

Módulo uUSB-HS

Por Michael Kusch
tintronic@yahoo.com
Versión preliminar 0.2

Introducción

Ciertamente la interfaz más utilizada para conectar dispositivos externos a un PC hoy en día es el Universal Serial Bus o USB. Esta interfaz utiliza sólo un par trenzado para comunicación en modo *half duplex*. Además posee 2 cables de alimentación que proporcionan un mínimo de 4.4V y hasta 500mA para alimentar periféricos de bajo consumo sin la necesidad de una fuente externa.

Una comunicación USB requiere de un dispositivo maestro llamado HOST que normalmente es un PC, el cuál inicia absolutamente todas las transferencias, y uno o más dispositivos esclavos. Hasta 127 esclavos pueden ser comandados por una interfaz USB HOST.

Debido a que el HOST inicia todas las transferencias, no existe comunicación directa entre dos dispositivos esclavos. Debido a esto, todo el grueso del procesamiento se hace en el PC HOST, simplificando los controladores USB esclavos, reduciendo así enormemente el costo de los chips de interfaz USB. Gracias a esto, hoy en día se encuentran innumerables dispositivos USB a muy bajo costo, como lo son cámaras web, teclados, Mouse, scanner y los últimamente populares Pendrive, o memorias flash externas, cuyo costo va en rápido descenso a la vez que su capacidad aumenta exponencialmente. Incluso ya existen controladores tipo *bridge* entre USB e IDE, los cuáles poseen pines para el conector USB y pines para el conector IDE, y no requieren de ningún procesador adicional que haga la traducción entre ambos protocolos. Sólo requieren unas pocas componentes externas.

Resulta entonces lógico pensar que una forma barata y simple de agregar funcionalidades a un microcontrolador sea utilizando una interfaz USB HOST. De esta manera puede crearse un *datalogger* con capacidad virtualmente ilimitada mediante el uso de un Pendrive o un disco duro USB, ó agregarse una cámara de video sin la necesidad de buscar un chip digitalizador de video ni tener que construir una interfaz compleja. Conectar un disco duro o un Pendrive a un microcontrolador sólo requiere de programar las funciones mínimas de la especificación USB para dispositivos de almacenamiento masivo o *Mass Storage Device Class*.

Hoy en día existen muchos microcontroladores que incluyen una interfaz USB como si fuera un puerto serial más y existen muchos puentes USB-RS232 y USB-Paralelo, pero son todos para dispositivos esclavos, es decir, son para construir equipos periféricos para ser conectados a un PC. Por esta razón no pueden iniciar una transferencia USB, sólo pueden responder a comandos enviados por el HOST.

La mayor parte de los pocos controladores USB HOST que pueden encontrarse en Internet incluyen un microcontrolador, típicamente un 8051 modificado, el cuál se conecta internamente al controlador USB. Entonces, para que un microcontrolador externo pueda implementar un HOST, debe comunicarse con el controlador USB a través del 8051. La desventaja es que el 8051 es un microprocesador y no posee los periféricos normalmente disponibles en los microcontroladores actuales, lo que implica que no puede sustituir a un microcontrolador. Esto implica tener que programar 2 procesadores y sus lenguajes respectivos. Además, el hecho de poseer un 8051, aumenta el costo del chip y el costo total del equipo en construcción. La única ventaja de este tipo de configuración radica en poder programar el *stack* USB en el 8051, liberando al microcontrolador de esta tarea.

Se encontró un único controlador USB HOST que no incluyera un procesador y que estuviera disponible para el mercado minorista a través de un distribuidor en EEUU. Corresponde al SL811HS de Cypress, el cuál puede ser configurado tanto en modo HOST como en modo esclavo.

El funcionamiento de este chip es similar al controlador de ethernet CS8900A en cuanto a que genera automáticamente los bits de preámbulo y los CRC correspondientes. Se podría decir que también es un *packet whacker* como el CS8900A, pero para USB.

Diseño esquemático

A continuación se encuentran los esquemáticos proporcionados por el fabricante para el modo HOST y esclavo en el documento “SL811HS - Interfacing an External Processor”

Con un rápido vistazo a ambos circuitos puede apreciarse que la diferencia es poca. Básicamente, el HOST debe ser capaz de proporcionar una alimentación de 5 volts a través de un switch de poder, mientras que el esclavo puede utilizar los 5 volts del puerto USB como fuente de alimentación. El estándar USB obliga a incluir un condensador de 120uF a la salida de la línea de +5 volts del puerto USB-A.

La otra diferencia, aparte del tipo de conector, está en las resistencias conectadas a las líneas D+ y D- del bus USB. El Host tiene 2 resistencias de 15[kΩ] a tierra, mientras que el esclavo tiene una resistencia de 1.5[kΩ] conectada a +3.3 volts. Dependiendo de la velocidad del esclavo, la resistencia se conecta a D+ (*full-speed*) o a D- (*low-speed*).

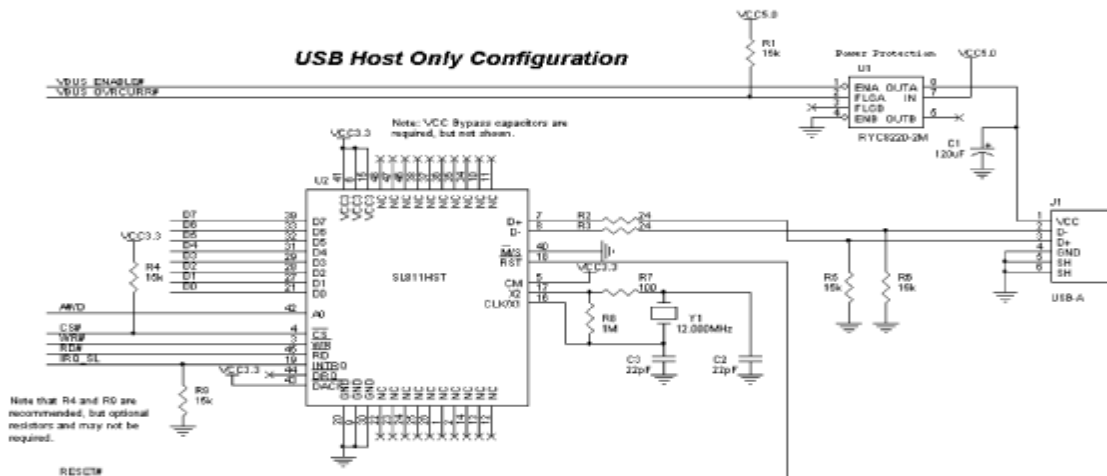


Figure 5. SL811HS in a Typical USB Host Configuration

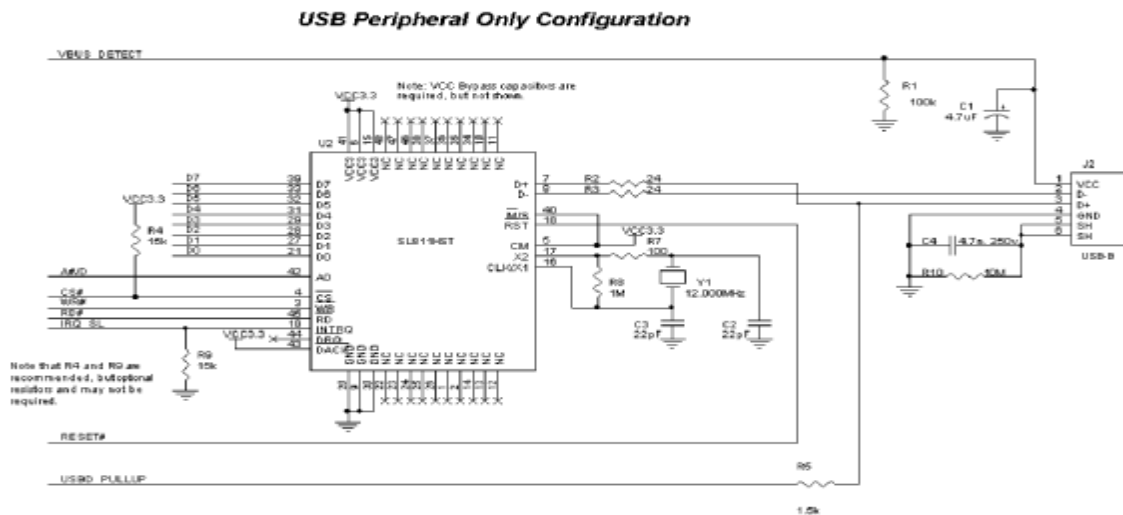
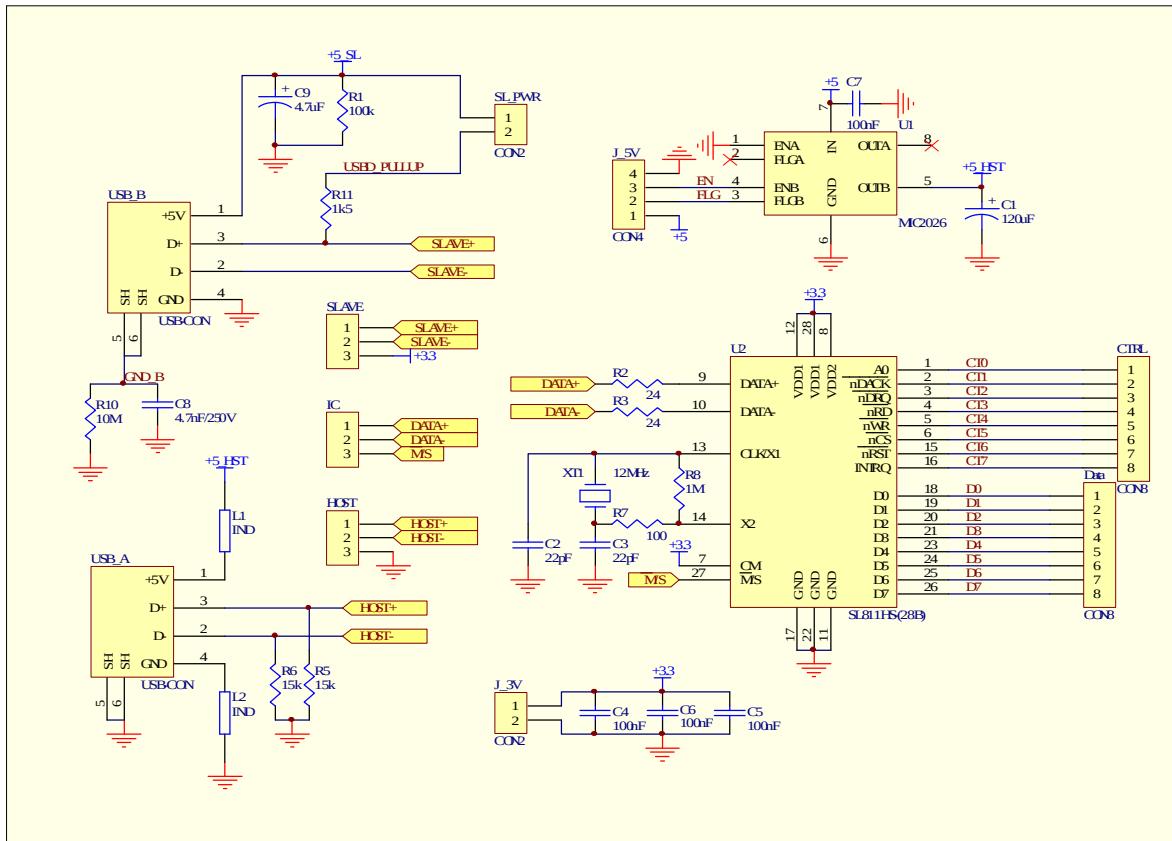


Figure 6. SL811HS/S in a Typical USB Peripheral Configuration

El SL811HS se alimenta con 3.3 volts, lo que significa que puede ser alimentado directamente del USB en modo esclavo, a través de un regulador de voltaje lineal de *low dropout* o baja caída. Además, sus entradas son tolerantes a 5 volts, haciéndolo compatible con microcontroladores de 5V. Los bits de control de salida son de colector abierto, pudiendo entonces conectar una resistencia de *pull-up* al voltaje de alimentación del microcontrolador. Si bien las salidas de datos son de 3.3 volt, éste voltaje alcanza a ser leído como un 1 válido para lógica TTL de 5V.

Aprovechando la similitud entre las conexiones HOST y esclavo, se diseñó un módulo que permitiera la operación en cualquiera de los 2 modos, seleccionables a través de 3 jumpers. El circuito esquemático del módulo se presenta a continuación.



Características del conector USB-A

Este conector hembra de forma rectangular indica que es un puerto HOST. Los pines 1 y 4 proveen alimentación de 5 volt al periférico. Posee inductancias en serie para la supresión de EMI. Los pines 2 y 3 son D- y D+ respectivamente. Todo host posee una resistencia de 15[kΩ] entre estos pines y tierra. El blindaje va conectado directamente a la tierra del circuito.

Características del conector USB-B.

Este conector de forma cuadrada indica que es un puerto esclavo. El pin 1 corresponde a la alimentación de al menos 4.4[V], con la cuál puede ser alimentado el SL811HS y el resto del circuito. Si el circuito es alimentado de forma externa, éste pin sirve para detectar la presencia de un HOST al otro extremo del cable. Este pin se encuentra disponible a través de un condensador y una resistencia conectados en paralelo cuya función es filtrar la línea para evitar la detección de falsos picos de voltaje que inducirían a una desconexión y reconexión del circuito esclavo. El pin 4 es tierra y va conectado directamente a la tierra del circuito esclavo.

Una resistencia de 1.5[k Ω] va conectada al pin D+. Al ser ésta conectada a +3.3[V], indica que el esclavo es un dispositivo de 12Mbps o “full speed”.

A diferencia del circuito maestro, el blindaje no va conectado directamente a tierra, sino que va conectado a través de una resistencia de 10[M Ω] en paralelo con un condensador cerámico de 4.7[nF], como lo recomienda el fabricante. Esto porque se trata del blindaje y su función es proteger la línea de transmisión de fuentes externas de interferencia y no debe ser usado como línea de tierra de potencia.

Características del switch de potencia.

Existen en el mercado switches de potencia especialmente diseñados para el estándar USB. Son circuitos integrados que poseen 2 pines de alimentación y dos switches de potencia. La entrada a cada switch proviene directamente de la alimentación. Cada switch posee un pin de salida, un pin de habilitación, y un pin de error. Estos circuitos integrados poseen una limitación de corriente continua de 500[mA] y protección contra cortocircuito y sobrecalentamiento. Cualquier error polariza el transistor de colector abierto, conectando el pin FLG a tierra. Estos chips pueden alimentarse entre 3 y 5 [V].

La razón por la cuál poseen 2 switches se debe a que en un PC, todos los puertos HOST aparecen de a pares. Internamente, cada controlador HOST está conectado a un HUB de 2 puertos. Al conectar un nuevo esclavo USB, el HOST encuesta al dispositivo cuánta corriente consume. El esclavo retorna un byte con el consumo en unidades de 2[mA]. Si el consumo es mayor al que puede entregar ese puerto, el HOST puede desconectar la alimentación a ese dispositivo.

La elección del switch de poder se hizo buscando primero los que tenía disponibles el distribuidor MOUSER de EEUU, donde se han comprado la mayor parte de las componentes no adquiribles en Chile. El switch de potencia elegido es el MIC2026 del fabricante MICREL. Sus principales características son:

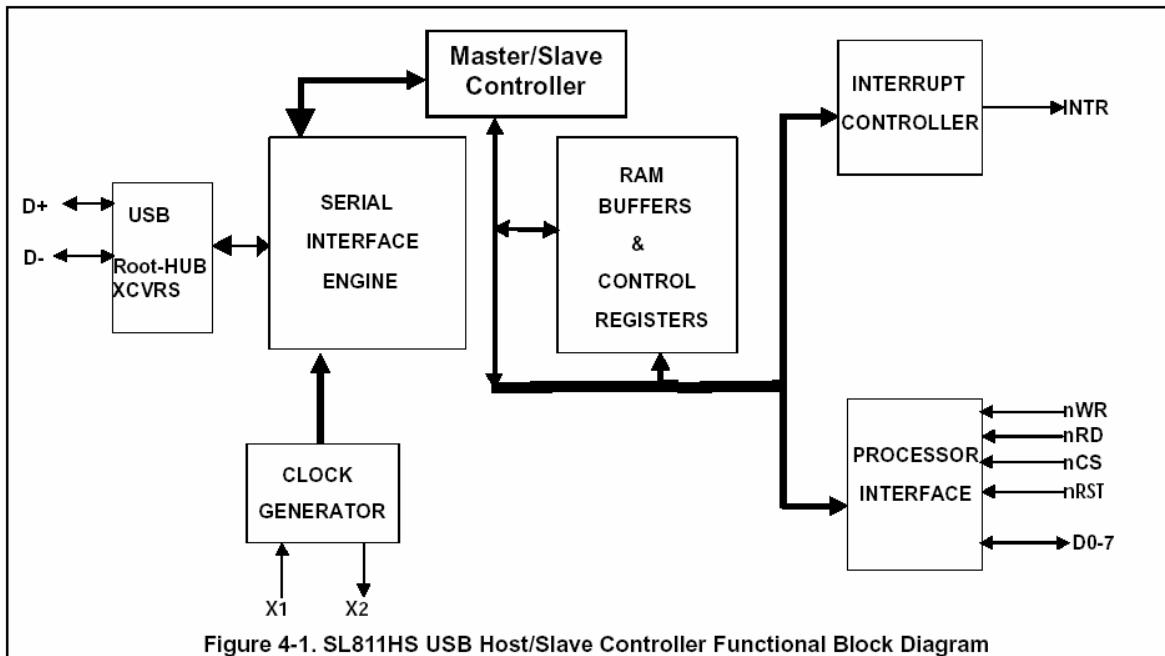
- Voltaje de operación: 2.7 a 5.5 [V].
- Resistencia máxima del switch: 140[m Ω].
- Corriente continua mínima: 500[mA] por canal.
- Protección ante cortocircuito y sobrecalentamiento.
- Disponible en empaque DIP-8 y SOIC-8.

Este módulo posee un único puerto USB-A, por lo cuál se utiliza sólo un switch. Se eligió el switch B por motivos de ruteo del circuito impreso. Los pines de alimentación y control se encuentran disponibles en la regleta de conexión. Debe recordarse conectar una resistencia de pull-up al pin FLG. También es recomendable conectar una resistencia de *pull-down* al pin ENB, ya que mientras el microcontrolador no inicialice el respectivo pin, éste estará como entrada que en la práctica es un estado de alta impedancia, quedando indeterminado el estado del switch.

La especificación USB requiere que todo puerto HOST tenga una capacitancia de salida de 120 μF para la alimentación de 5V.

Características del controlador SL811HS.

A continuación se presenta un diagrama de bloques del SL811HS, extraído directamente de la hoja de datos de éste.



Los pines D+ y D- se encuentran ruteados hacia los conectores USB-A y USB-B. A través de 2 *jumper* se selecciona cuál conector se utilizará. Los pines X1 y X2 se encuentran conectados al circuito del cristal. Todas las señales de control restantes y de datos se encuentran disponibles en la regleta del módulo. Para modo HOST, las señales de DMA llamadas nDREQ y DACK no se utilizan. Existen otros 3 pines que no se ven en este diagrama: M/S, A0 y CM. Estos serán explicados a continuación.

El pin A0 selecciona si lo que se está transfiriendo por el bus de datos es una dirección de memoria o un dato, parecido al pin RS de un display LCD inteligente. Las señales nRD y nWR son las de lectura y escritura. El pin nRST es de reset, activo en bajo. El pin nCS es de *chip select*, habilitando el funcionamiento de todas las demás señales y sacando al bus de datos de su estado de alta impedancia. Este chip posee un único pin de petición de interrupción para todas las señales internas de interrupción.

El pin M/S selecciona el modo de trabajo HOST(*master*) o esclavo(*slave*). Éste se conecta a Vcc o GND por medio de un jumper. Sin embargo, el funcionamiento como HOST o esclavo puede ser cambiado por software a través del bit 7 del registro CONTROL2 (0x0F).

El pin CM selecciona la frecuencia del cristal a utilizar como fuente de reloj. El SL811HS requiere de una señal de reloj de 48[MHz]. Ésta puede provenir de un cristal de 48[MHz] ó, gracias a un PLL interno, de un cristal de 12[MHz], que es más barato y fácil de encontrar. Además, de la figura siguiente se aprecia que el circuito recomendado para conectar un cristal de 48[MHz] es más complejo y requiere de una bobina, la cuál en empaque superficial es normalmente cara y más grande y difícil de soldar.

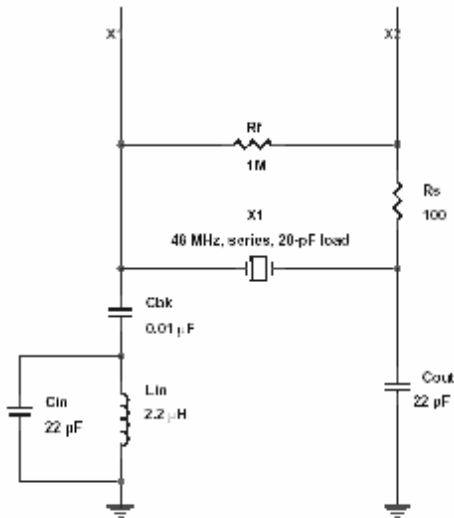


Figure 4-2. Full-Speed 48-MHz Crystal Circuit

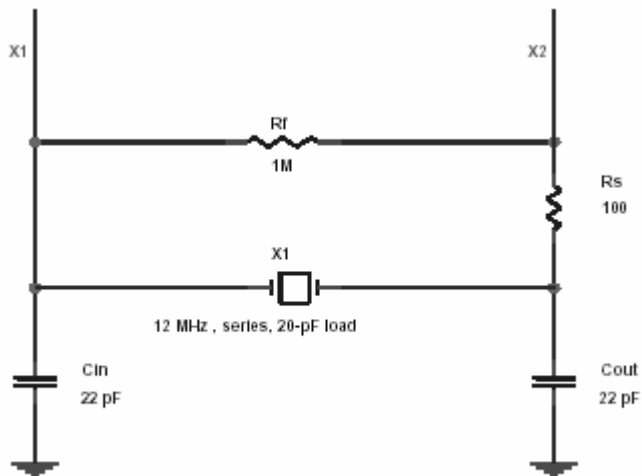


Figure 4-3. Optional 12-MHz Crystal Circuit

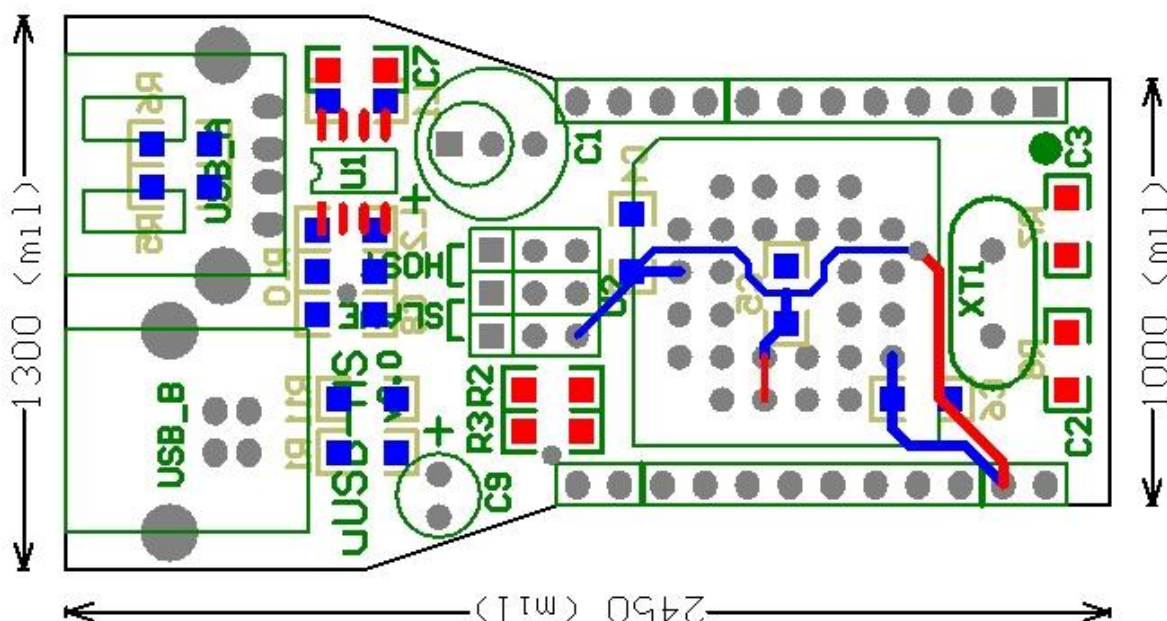
Se han colocado 3 condensadores de 100[nF] lo más cercano posible a los pines de alimentación del SL811HS, ya que el PLL interno de éste es muy sensible al ruido de alta frecuencia en la alimentación. Esto está recomendado por el fabricante.

Diseño del circuito impreso.

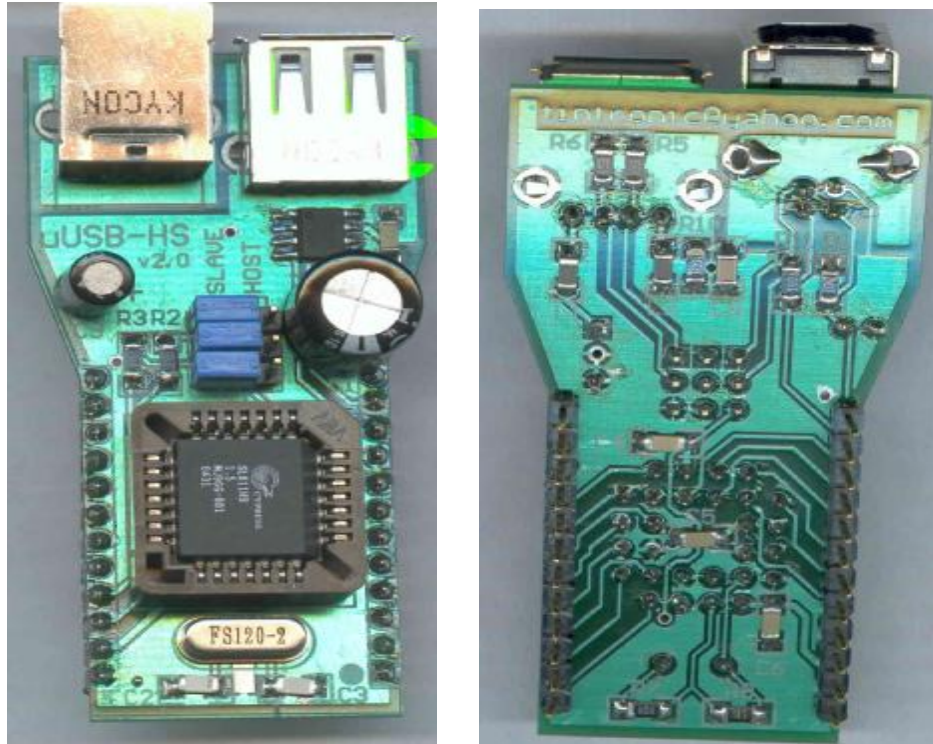
Para hacer el módulo lo más compacto posible se utilizaron componentes de montaje superficial donde fuera posible, y se diseñó el circuito impreso con componentes por ambas caras. La distancia entre las 2 regletas de conexión debía ser tal que el módulo pudiese conectarse a un protoboard, por lo cual se partió utilizando las medidas del módulo *Header MSP430*. El socket PLCC-28 para el chip SL811HS impidió poder hacer el circuito impreso aún más angosto. Sin embargo, al igual que el módulo Header MSP430, éste cabe en un solo protoboard, dejando una fila a cada lado para conectar cables ó un bus de datos hacia el microcontrolador. También tiene el ancho justo para poder conectarlo entre 2 protoboards, teniendo así el máximo de 4 puntos libres por pin para el conexionado.

Debido al porte de los conectores USB, fue necesario ensanchar el circuito impreso. Esto no presenta ningún inconveniente, ya que los conectores USB se encuentran lejos de las regletas de conexión al protoboard. Además, el ensanchamiento de ese extremo del circuito impreso provee de espacio suficiente para poner las demás componentes.

Se puso especial énfasis en lograr un buen filtrado de la línea de alimentación. Para ello se trató de llevar la línea de +3.3[V] lo más directamente posible a cada pin de alimentación del SL811HS. Se posicionaron los condensadores de filtraje C4, C5 y C6 lo más cerca posible de los pines de alimentación. Finalmente se ruteó la línea de alimentación de manera tal que una vez que pasara por el *pad* del condensador se fuera directamente al pin de alimentación y terminara en él. La idea es que ojalá cada pin de alimentación del chip estuviese conectado primero a un condensador y luego directamente al pin de alimentación de la regleta. Esto para que el condensador sólo filtre el ruido para el respectivo pin, y no deba además encargarse de filtrar el ruido de otras componentes que también estén conectadas a +3.3[V]. A continuación se muestra el ruteo de la línea de +3.3[V]. En rojo se muestra la cara superior, mientras que en azul se muestra la cara inferior.



El circuito impreso con las componentes montadas puede verse en la siguiente figura. A la izquierda se encuentra la cara superior. En ella pueden verse los conectores USB-A (derecha) para la interfaz HOST y USB-B (izquierda) para la interfaz esclava. Los *jumpers* azules están puestos en configuración esclavo.



Se eligió un empaque PLCC (de 28 pines) para el chip SL811HS por dos motivos:

- El socket PLCC-28 es mucho más fácil de soldar que el chip de empaque TQFP de 48 pines, ya que este último posee un espaciado de apenas 10[mil](milésimas de pulgada) entre pines y un grosor de pines de también 10[mil]
- En caso de quemarse el SL811HS por conexiónado, el chip es fácilmente reemplazable ya que no va soldado al circuito impreso. No existe un socket para el empaque TQFP, por lo que este debería haberse soldado directamente al circuito impreso.

Stack USB HOST

Implementar un stack USB HOST para un microcontrolador alcanza para una memoria por sí sólo y escapa al objetivo de esta Memoria. Es el equivalente a desarrollar el stack TCP/IP *uIP*, compilado para la MSP.

Se presenta un código de ejemplo que permite utilizar un teclado USB con un microcontrolador MSP430.

Conocimientos básicos del protocolo USB

El bus USB está pensado específicamente para la conexión de dispositivos periféricos a un computador. Por esta razón, todo bus USB se compone de un único dispositivo maestro llamado HOST, y de uno hasta 127 periféricos, incluidos Hubs. Al conectar un dispositivo esclavo a un bus USB, el Host inicia un proceso de configuración del dispositivo. En primer lugar se corre un proceso llamado enumeración, en el cuál a cada dispositivo conectados al bus le es asignado un único número que lo identifica. A diferencia de ethernet, un HUB USB no es sólo un repetidor de señal, sino un dispositivo periférico en sí que también debe ser enumerado. Al principio, todos los esclavos responden a la dirección #0.

Además de la dirección, cada dispositivo esclavo posee un número limitado de puertos, llamados *endpoints*, que pueden ser de entrada ó de salida. Cada *endpoint* es visto como un buffer receptor ó proveedor de datos. Entonces, la comunicación no se realiza sólo con un dispositivo, sino con un determinado *endpoint* del dispositivo.

Los tipos de paquetes que se transmiten por el bus USB son los siguientes:

- SETUP
Indica la transferencia de un comando de configuración.
- IN
Indica que el siguiente paquete es una lectura de datos que serán transferidos desde el esclavo hacia el host.
- OUT
El siguiente paquete será de datos desde el host hacia el esclavo.
- SOF/EOF
Start of Frame. Paquete de inicio de FRAME. Es transmitido una vez cada 1[ms] para evitar que los esclavos entren en modo de bajo consumo. Posee un índice de tiempo que cicla cada 2048[ms] para coordinar dispositivos de tiempo real. EOF es la versión de baja velocidad del SOF.
- Preamble
Es transmitido para indicar que la siguiente transferencia será con un dispositivo de baja velocidad.
- ACK
Indica que el último comando fue reconocido y ha comenzado a ser ejecutado. *No* indica que ya se terminó de procesar.

- **NAK**
El destinatario no pudo procesar el paquete debido a que se encuentra ocupado. La operación deberá ser efectuada nuevamente. Esto no se hace de forma automática.
- **STALL**
El comando no es conocido. Es equivalente a un STOP, pero el dispositivo esclavo puede seguir funcionando.
- **DATA0 ó DATA1**
Estos paquetes son transmitidos después de un paquete de comando IN, OUT ó SETUP y contiene los datos referentes a esos comandos. Al enviar o recibir datos, se transmiten alternadamente como paquete DATA0 ó DATA1, para así saber si se ha perdido un paquete entremedio.

Toda transferencia es iniciada por el HOST, el cuál envía un paquete llamado TOKEN. Éste contiene el tipo de transferencia (SETUP/IN/OUT/SOF/EOF), la dirección y puerto (*endpoint*) de destino y un CRC5 del paquete. Luego se transfiere un paquete de datos, ya sea desde (OUT) o hacia (IN) el HOST. Finalmente se envía un paquete de *handshake* (ACK,NACK,STALL) que contiene el resultado de la operación.

Es importante hacer notar que el NACK no se envía en respuesta a un paquete que ha llegado con errores, sino que indica que el paquete no pudo ser procesado porque el dispositivo se encuentra ocupado. Si un paquete llega con errores, el dispositivo simplemente lo descarta y no envía respuesta alguna.

El HOST transmite un TOKEN SOF (*Start of Frame*) cada 1[ms] indicando el comienzo de cada *frame*. Por lo tanto, cada *frame* dura 1[ms]. Toda transferencia debe realizarse dentro de un mismo frame y no puede pasarse a otro.

La revisión 1.1 del protocolo USB establece 2 velocidades de comunicación:

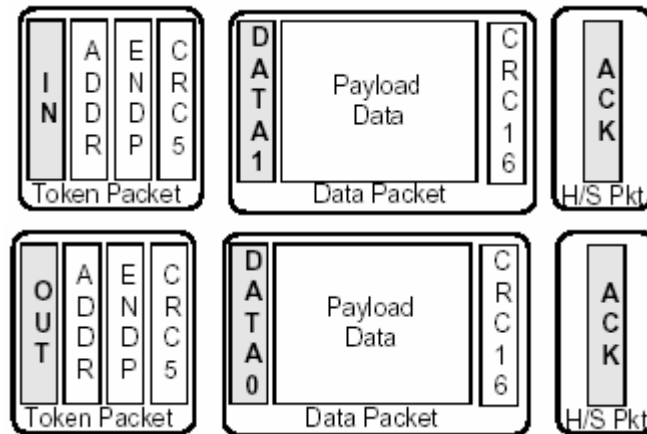
- **Velocidad completa ó *Full-Speed***: La comunicación se efectúa a 12Mbps. Utilizada para dispositivos rápidos como discos duros, scanners, dispositivos de audio y video.
- **Velocidad baja o *Low Speed***: La comunicación se efectúa a 1.5Mbps y debe ir presidida por un *preamble* para indicar que el paquete a continuación es de baja velocidad. Es utilizada básicamente por dispositivos de interfaz hombre-maquina, como mouse y teclados.

Existen 4 tipos de transferencia:

1. *Bulk Transfers*

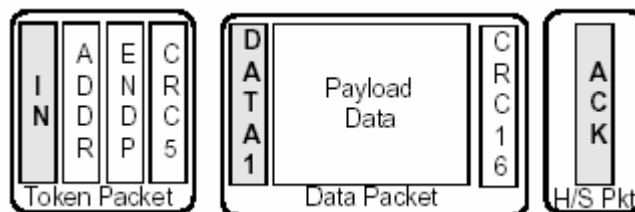
Son transferencias tipo ráfaga, cuyo paquete de datos contiene 8, 16, 32 ó 64 bytes. Puede ser de tipo IN (slave a host) o de tipo OUT (host a slave). Cada transferencia se compone de 3 paquetes: *Token* (IN ó OUT), *Payload Data* (datos) y *Handshake* (ACK, NACK ó STILL).

El protocolo USB tuvo muy en mente el concepto de errores de transmisión, tratándolos como un hecho de la vida en vez de un caso fortuito. Por esta razón, además del CRC16, se utiliza un mecanismo llamado *Data Toggle*, alternando el ID de paquete ó *PID* entre DATA0 y DATA1, cada vez que se transfiere un paquete de datos.



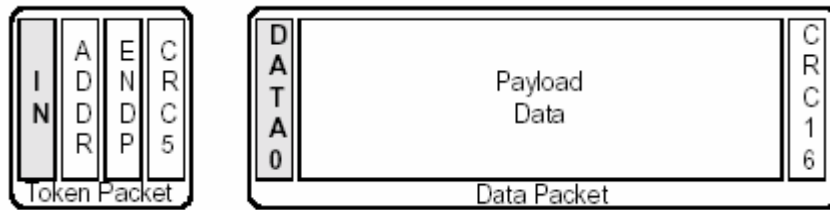
2. Interrupt Transfers

Este tipo de transferencia fue agregado en la revisión 1.1 del protocolo USB. Es parecido al *bulk transfer*, pero sólo existe para paquetes tipo IN. Pueden transferirse de 1 a 64 bytes de datos. Debido a que toda comunicación es iniciada por el HOST, los *endpoints* de tipo interrupción tienen asociado un intervalo de encuestamiento o *ping*, para asegurar que sean encuestados por el host.



3. Isochronous Transfers

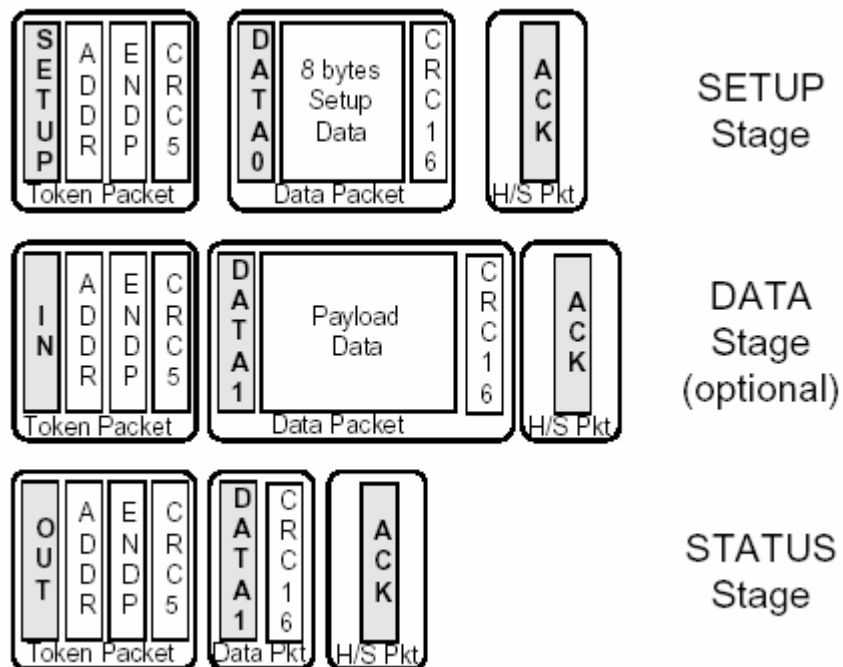
Transferencia isocrónica es utilizada para aplicaciones de tiempo real, como audio y video. El bajo retardo y un ancho de banda asegurado son las características más importantes en este tipo de aplicaciones. Para ello se reserva una cierta cantidad de ancho de banda para transferencias isocrónicas, las cuáles se realizan al principio de cada frame (después de un paquete SOF). Para disminuir el *overhead*, este tipo de transferencias no posee un paquete de *handshake* (ACK, NACK, STALL), no implementa el mecanismo de *data toggle* (siempre es DATA0) ni posee retransmisiones. El único mecanismo de detección de errores que posee es el CRC16 intrínseco a un paquete de datos. El máximo largo son 1024 bytes de datos.



4. Control Transfers

Este tipo de transferencias se utiliza para controlar y enviar comandos a los dispositivos esclavos. Debido a su extrema importancia, de este tipo de transacción emplea la mayor verificación de errores que el protocolo USB puede ofrecer. Las transferencias de control son ejecutadas de acuerdo a una “ley del mejor esfuerzo”, proceso de 6 pasos definidos en la sección 5.5.4 de la revisión 1.1 del estándar USB.

Una transferencia de control consta de 2 ó 3 pasos. Primero se envía una *transferencia* de SETUP, en la cuál se envían 8 bytes de datos de control. En caso de que se necesiten enviar más datos, se ejecuta una nueva transferencia tipo IN. Finalmente se ejecuta una transferencia de estado, en la cuál el dispositivo esclavo confirma la ejecución satisfactoria de un comando de control.



Funcionamiento del controlador Host/Slave SL811HS de Cypress.

El controlador SL811HS posee un bus de 8 bits compartido para direcciones y datos. El tipo de información a transferir se elige con el pin A0. Cuando A0=0, se accesa a un puntero el cuál contiene la dirección de memoria a escribir/leer. Cuando A0=1, se accesa a la información contenida en la dirección del puntero. Al leer o escribir cualquier dirección de memoria, el puntero se incrementa automáticamente. Sin embargo, un documento de Erratas del SL811HS publicado por Cypress, declara que el puntero podría no incrementarse, por lo cuál no es recomendable utilizar la propiedad de autoincremento del puntero.

Debido a que el puntero es de 8 bits, el tamaño de memoria del SL811HS es de 256 bytes. Los primeros 16 bytes corresponden a registros de configuración. Los siguientes 240 bytes corresponden al buffer de datos para la comunicación USB. Este buffer posee un puntero que indica la posición del primer byte dentro del buffer de los datos a enviar ó recibir. Así pueden armarse varios paquetes a la vez, ó armarse un paquete mientras otro se está transfiriendo, ya que la mayoría de los paquetes de datos son de largo máximo 64 bytes (de datos).

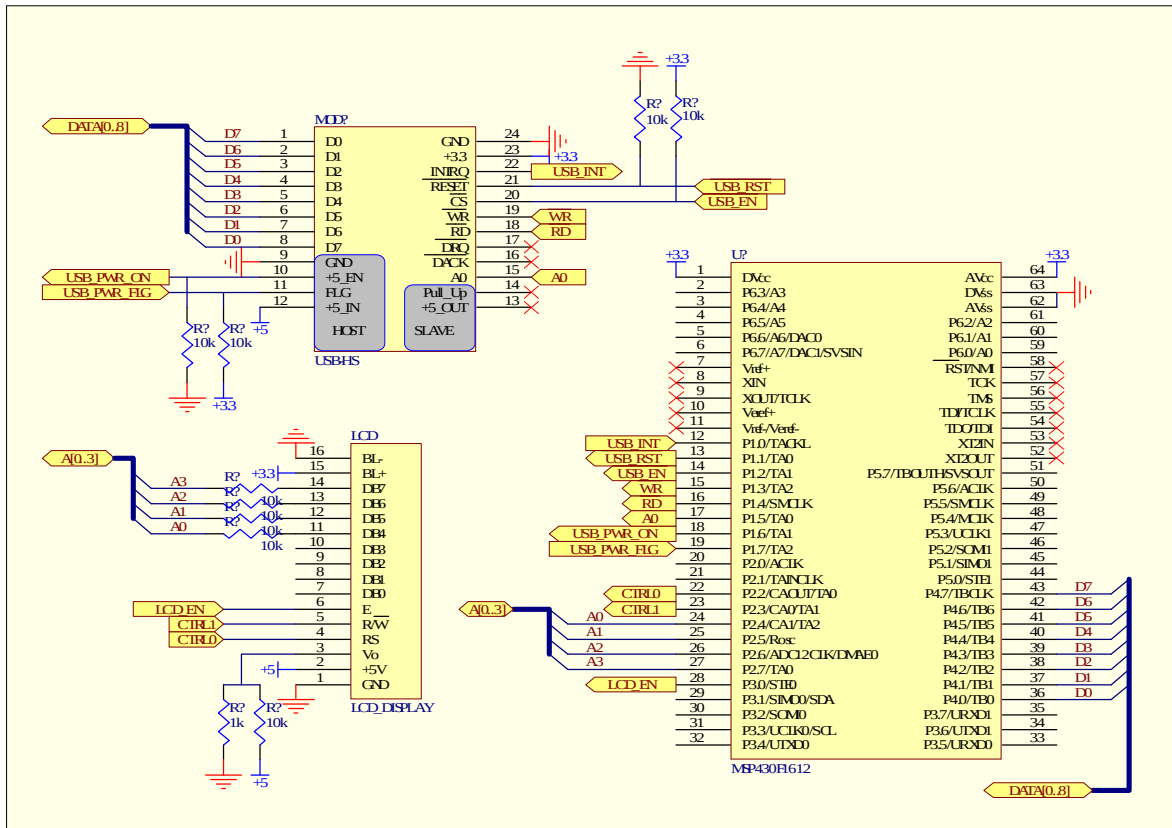
El SL811HS posee un único pin de petición de interrupción, el cuál se pone en uno cuando se activa cualquiera de sus banderas de interrupción, las cuáles pueden ser enmascaradas por software.

Además de la interfaz lógica, el SL811HS efectúa muchas de las funciones del protocolo USB por sí sólo:

- Generación automática de paquetes de inicio de frame (SOF/EOF).
- Generación automática de *preamble* para paquetes de baja velocidad.
- Cálculo automático del CRC5 para paquetes TOKEN.
- Cálculo automático de CRC16 para paquetes de datos.

Programación del SL811HS con un microcontrolador MSP430.

El conexionado utilizado incluye una MSP430F1612, un Módulo “uUSB-HS v2.0” y un display LCD inteligente de 4 líneas por 20 caracteres utilizado en modo bidireccional. El diagrama esquemático del conexionado se presenta a continuación:



Primero debemos definir las funciones básicas que leen y escriben un registro del SL811HS. Para ello se han definido los pines de comunicación entre la MSP y el SL811HS de la siguiente forma:

```
//Bus de datos/control
#define USB_IN  P4IN
#define USB_OUT P4OUT
#define USB_SEL P4SEL
#define USB_DIR P4DIR
//Pin de interrupción. Asignación exclusiva.
#define USB_INT P1IN&BIT0 //Active High
#define USB_INT_B BIT0 //Active High
#define USB_INT_init P1DIR&=~BIT0;P1SEL&=~BIT0; // IN; DigitalIO;
#define USB_INT_EN P1IE|=BIT0;P1IES&=~BIT0;
//pin de reset. Asignación exclusiva.
#define USB_RST_ON P1OUT|=BIT1
#define USB_RST_OFF P1OUT&=~BIT1
#define USB_RST_init P1DIR|=BIT1;P1SEL&=~BIT1; // OUT; DigitalIO;
//pin de ENABLE o CHIP_SELECT. Asignación exclusiva.
#define USB_EN_ON P1OUT|=BIT2
#define USB_EN_OFF P1OUT&=~BIT2
#define USB_EN_init P1DIR|=BIT2;P1SEL&=~BIT2; // OUT; DigitalIO;
//Pin de escritura. Multiplexable.
#define USB_WR_ON P1OUT|=BIT3
#define USB_WR_OFF P1OUT&=~BIT3
```



```

#define USB_WR_init P1SEL&=~BIT3;    // DigitalIO;
#define USB_WR_OUT P1DIR|=BIT3;
#define USB_WR_IN P1DIR&=~BIT3;
//Pin de lectura. Multiplexable.
#define USB_RD_ON P1OUT|=BIT4
#define USB_RD_OFF P1OUT&=~BIT4
#define USB_RD_init P1SEL&=~BIT4;    // DigitalIO;
#define USB_RD_OUT P1DIR|=BIT4;
#define USB_RD_IN P1DIR&=~BIT4;
//Pin de dirección. Selecciona DAT0 o REGISTRO. Multiplexable.
#define USB_A0_ON P1OUT|=BIT5
#define USB_A0_OFF P1OUT&=~BIT5
#define USB_A0_init P1SEL&=~BIT5;    //OUT; DigitalIO;
#define USB_A0_OUT P1DIR|=BIT5;
#define USB_A0_IN P1DIR&=~BIT5;
//Pin de habilitación del switch de poder hacia el esclavo
#define USB_PWR_ON P1OUT|=BIT6
#define USB_PWR_OFF P1OUT&=~BIT6
#define USB_PWR_init P1DIR|=BIT6;P1SEL&=~BIT6;    //OUT; DigitalIO;
//Pin de error del switch de poder
#define USB_PWR_FLG P1IN&BIT7 //Active Low
#define USB_PWR_FLG_init P1DIR&=~BIT7;P1SEL&=~BIT7;    //IN;
DigitalIO;

//USB_SEL First makes bits WR RD and A0 outputs. Then it enables
SL811HS
#define USB_SELECT USB_WR_ON;USB_WR_OUT;USB_RD_ON;USB_RD_OUT;
USB_A0_OUT;USB_EN_OFF;

//USB_DES First disables SL811. Then it turns bits WR RD and A0
back to inputs (for multiplexing)
#define USB_DESELECT USB_EN_ON;USB_WR_IN;USB_RD_IN;USB_A0_IN;

```

Ahora la definición de los 16 registros.

```

#define CTL 0x00 // write this register to kick off a transfer
#define BUFADR 0x01 // start of internal data buffer
#define BUFLen 0x02 // length of internal buffer
#define PID_EP 0x03 // name when written--PID and Endpoint
for next xfr
#define PKTSTAT 0x03 // name when read--status of last
transfer
#define FNADDR 0x04 // name when written--USB function
address
#define CTL1 0x05 // more control stuff
#define INTSTATUS 0x0D // Interrupt request status bits. We use
DONE and SOF.
#define SOFCT_L 0x0E // SOF (EOP) time constant low byte
#define SOFCT_H 0x0F // name when written--EOP time constant
high byte

```

Algunas definiciones para los datos a escribir en registros.

```
#define IN_PID      0x90  // PID (Packet ID) constants
#define SETUP_PID  0xD0  // for the 'set address' request
#define SOF_PID     0x05  // constants for 811 CTL1 register
#define USB_RESET  0x08  // SIERES=1
#define USB_OPERATE 0x21  //Low Speed=1(b5),SOF(EOP)EN=1(b0)
```

Finalmente las funciones básicas.

```
void wr811(BYTE a, BYTE d)
```

```
{
    USB_SELECT;      //Make SL811 ready to communicate with.

    USB_DIR=0xFF;    //Data port to outputs
    USB_OUT=a;
    USB_A0_OFF;      //Write address to pointer
    USB_WR_OFF;
    USB_WR_ON;

    USB_OUT=d;
    USB_A0_ON;        //Write data to register
    USB_WR_OFF;
    USB_WR_ON;
    USB_DIR=0x00;    //Data port to inputs (for multiplexing)

    USB_DESELECT;    //Liberate used signals
}
```

```
BYTE rd811(BYTE a)
```

```
{
    unsigned char d;
    USB_SELECT;      //Make SL811 ready to communicate with.

    USB_DIR=0xFF;    //Data port to outputs
    USB_OUT=a;
    USB_A0_OFF;      //Write address to pointer
    USB_WR_OFF;
    USB_WR_ON;

    USB_DIR=0x00;    //Data port to inputs
    USB_A0_ON;        //Read data from register
    USB_RD_OFF;
    d=USB_IN;         //Read DATA
    USB_RD_ON;

    USB_DESELECT;    //Liberate used signals
    return d;
}
```

```

void addr811(BYTE a)
{
    USB_SELECT;        //Make SL811 ready to communicate with.

    USB_DIR=0xFF;      //Data port to outputs
    USB_OUT=a;
    USB_A0_OFF;        //Write address to pointer
    USB_WR_OFF;
    USB_WR_ON;

    USB_DIR=0x00;      //Data port to inputs
    USB_DESELECT;      //Liberate used signals
}

```

La rutina de inicialización del chip SL811HS retorna el número de versión de hardware del SL811HS:

- 00h: SL11
- 01h: SL811 rev. 1.2
- 02h: SL811 rev. 1.5

La primera función que debe ejecutarse es una inicialización de los pines de comunicación, ejecutar un reset del chip SL811HS y obtener el número de versión, para así asegurarse de que el SL811HS está correctamente conectado.

```

unsigned char USB_init(void)
{
    unsigned char rev;
    USB_RST_OFF;        //SL811 Reset enabled.
    USB_RST_init;
    USB_EN_ON;          //SL811 Disabled.
    USB_EN_init;
    USB_PWR_OFF;        //Power to USB-A disconnected.
    USB_PWR_init;

    USB_WR_init;
    USB_RD_init;

    USB_INT_init;        //interrupt input;
    USB_PWR_FLG_init;    //USB_+5V Error (overload) input

    Delayx100us(250);    //Delay 25ms
    USB_RST_ON;          //Disable Reset.

    USB_SEL=0x00;        //DigitalIO
    rev=( (rd811(0x0E)&0xF0) >>4 ); //Get HW rev. number
    USB_PWR_ON;          //Enable +5V to USB Slave
    return rev;
}

```

Para enviar un comando USB y esperar hasta que la operación haya sido terminada (comando + datos + ack), se utiliza la función `go()`, la cuál retorna el registro de estado `STATUS`.

```
BYTE go(BYTE cmd)                // Launch an 811 operation.
{
    wr811(INTSTATUS,0x01); // clear the DONE bit
    wr811(CTL,cmd);        // start the operation
    Delayx100us(1);
    while(rd811(INTSTATUS) & 0x01 == 0)
    {;}                    // spin while "done" bit is low
    return rd811(PKTSTAT); // return the status of this transfer
}
```

El siguiente ejemplo asume que hay un teclado USB conectado directamente al módulo USB HOST al encender la MSP, y que no hay HUB de por medio.

Antes de utilizar un dispositivo USB, éste debe ser enumerado y de ser necesario, configurado. Para ello, primero se envía un reset del bus USB, que es distinto a un reset del SL811HS. Un reset USB se envía a un puerto específico de un HUB o en este caso al dispositivo conectado directamente al módulo USB. Un reset USB causa un reset en el dispositivo esclavo, el cuál responde ahora a la dirección #0 del bus USB.

Primero se reservan 8 bytes del buffer del SL811HS para datos. Esto se hace estableciendo primero la dirección donde se guardarán los datos a ser enviados. Como se vio anteriormente, el buffer comienza en la dirección 0x10, por lo que se eligió esta dirección, escribiendo el registro `BUFADR(0x00)`. Luego se escribe el largo del paquete en el registro `BUFLEN(0x02)`.

```
wr811(BUFADR,0x10); // start of SETUP/IN internal data buffer
wr811(BUFLEN,0x08); // reserve 8 bytes
```

Luego se genera una condición de reset del bus USB durante unos 25[ms], para dar tiempo al esclavo de resetearse.

```
// (1) Reset the USB device. This makes it listen to address 0.
wr811(CTL1,USB_RESET); // Speed=1(L), JKFORCE=1, SOFEN=1
Delayx100us(250);      // about 18 milliseconds
wr811(CTL1,USB_OPERATE);
```

Luego se habilita la generación automática de paquetes SOF. Como ya se explicó, estos paquetes se envían una vez cada 1[ms] al inicio de cada frame. De esta manera se evita que el esclavo entre en modo suspendido. El valor de SOF se basa en una señal de reloj de 12MHz, por lo cuál el valor correcto es 12000, ó 0x2EE0. Debe tenerse cuidado al modificar el registro `SOFCT_H(0x0F)`, ya que los bits 6 y 7 de éste configuran la inversión de polaridad D+/D- y el modo Master(HOST)/Slave, respectivamente. El pin de selección M/S configura este bit (bit 7 del registro 0x0F) cuando el SL811HS sale de reset. Luego el pin M/S es ignorado y sólo cuenta el estado del mencionado bit.

```
// Enable sending 1 msec EOP's (the low-speed version of SOF's)
wr811(SOFCT_L, 0xE0); // Set the SOF generator time constant
wr811(SOFCT_H, 0x2E | 0xC0); // 1 ms SOF rate, b7=HOST, b6=POLSWAP
```

Luego se ennumeran los dispositivos USB. Como se asume de antemano que ya existe un único teclado conectado directamente al Módulo USB, puede asignársele la dirección #1, sin necesidad de preguntar primero qué tipo de dispositivo es para buscar el driver adecuado. La enumeración se realiza enviando un paquete tipo SETUP, con el comando set_address, especificado en el capítulo 9 del estándar USB rev1.1. Para ello debe primero llenarse el buffer del SL811HS con los 8 bytes de datos con los campos del comando set_address de la siguiente forma:

```
// (2) Issue a SET_ADDRESS USB request, setting the peripheral
// address to 1 From the USB spec, Chapter 9, here is the data
// for a "SET_ADDRESS" request:
wr811(0x10,0x00); // bmRequestType (h->d,std request,device is
// recipient)
wr811(0x11,0x05); // bRequest (SET_ADDRESS)
wr811(0x12,0x01); // wValueL (device address)--we're setting
// it to ONE
wr811(0x13,0x00); // wValueH (zero)
wr811(0x14,0x00); // wIndexL (zero)
wr811(0x15,0x00); // wIndexH (zero)
wr811(0x16,0x00); // wLengthL (zero)
wr811(0x17,0x00); // wLengthH (zero)
```

Debido a que el teclado acaba de ser reseteado, éste responde a la dirección #0, por lo tanto el comando set_address se envía a esa dirección. Una vez armado el paquete, éste es enviado a través de la función go, cuyo resultado debe ser un ACK, de lo contrario significa que el teclado no está conectado.

```
wr811(FNADDR,0x00); // USB address zero
wr811(PID_EP,SETUP_PID | 0x00); // SETUP PID, EP0
result=go(0x07); // DIREC=1(out), ENAB=1, ARM=1
```

Una vez que el teclado ha recibido satisfactoriamente el comando set_address, envía de vuelta un ACK y recién entonces comienza a configurarse para responder a la dirección #1. Es importante notar que el ACK no significa que haya terminado la ejecución del comando enviado, por el contrario, significa que ha reconocido el comando y va a comenzar a ejecutarlo.

Luego se le pide un paquete de datos al dispositivo, el cuál contiene el estado de éste. En la configuración de los dispositivos HID, éste es un paquete sin datos. Si el teclado responde con un NACK, significa que aún está procesando el comando anterior, por lo que el último comando debe ser reenviado hasta que el teclado responda con un ACK.

```

result = 0;
while( !(result & 0x01) )
{
    // STATUS stage is a no-data IN to EP0
    wr811(PID_EP, IN_PID | 0x00); // IN PID, EP0
    result=go(0x03); // Don't sync to SOF, DIREC=0(in), ENAB, ARM
    //display(result);
}

```

Ahora el teclado debe ser configurado. Si no se envía el comando de configuración, el teclado se encontrará en un estado “desconfigurado” y probablemente no retorne ningún dato.

```

// (3) Send a CONTROL transfer to select configuration #1. Until
// we do this the device is in an "unconfigured" state and
// probably won't send data.
// Again, from USB spec Chapter 9:
wr811(0x10, 0x00); // bmRequestType (h->d, std request, device is
                    // recipient)
wr811(0x11, 0x09); // bRequest (SET_CONFIGURATION)
wr811(0x12, 0x01); // wValueL (configuration = 1)
wr811(0x13, 0x00); // wValueH (zero)
wr811(0x14, 0x00); // wIndexL (zero)
wr811(0x15, 0x00); // wIndexH (zero)
wr811(0x16, 0x00); // wLengthL (zero)
wr811(0x17, 0x00); // wLengthH (zero)

wr811(FNADDR, 0x01); // now talking to USB device at address 1
wr811(PID_EP, SETUP_PID | 0x00); // OR in the endpoint (zero)
result=go(0x07); // DIREC=1(out), ENAB=1, ARM=1
// STATUS stage is a no-data IN to EP0
wr811(PID_EP, IN_PID | 0x00); // IN PID, EP0
result=go(0x03); // DIREC=0(in), ENAB=1, ARM=1

```

Ahora el teclado ya se encuentra configurado y podemos iniciar una lectura periódica. El SL811HS generará automáticamente un EOF cada 1[ms] y pondrá al mismo tiempo la bandera EOF del registro STATUS en 1. Esta bandera puede ser utilizada como una señal de reloj de 1[kHz]. Una lectura típica de una tecla se hace cada 4 a 20 [ms]. En este ejemplo se hará una lectura cada 4 banderas EOF, o sea, cada 4 [ms]. Una vez leída la bandera, ésta debe ser sobrescrita con un 1 para ser borrada.

```

wr811(PID_EP, IN_PID | 0x01); // set up for IN PIDS to endpoint 1
while(1)
{
    addr811(0x10); //reset pointer to beginning of internal buffer
    waitframes(4); // send the IN requests every n milliseconds
    result=go(0x03); // DIREC=0(in), ENAB=1, ARM=1
    if (result & 0x01) // look only for ACK
        decode();
}

```

Se han creado 2 nuevas funciones. waitframes(n) espera a que pasen n frames. decode() hace la decodificación del paquete de 8 bytes retornado por el teclado.

```
void waitframes(BYTE num)
{
    int j=0;
    BYTE result;
    while (j < num)
    {
        wr811(INTSTATUS,0xFF); // clear the int request flags
        while (1)
        {
            result = (rd811(INTSTATUS)); // hang while SOF flag
                                         low
            result &= 0x10;                // SOF flag is bit 4
            if (result == 0x10) break;
        }
        j++;
    }
    Delayx100us(1);
}

void decode(void)
{
    unsigned char i,len;
    len=rd811(BUFLen); //number of bytes returned by keyboard
    for(i=0;i<len;i++) //copy data to local buffer
        USB_BUF[i]=rd811(0x10+i);
    SEND_CMD(LINE3);
    for(i=2;USB_BUF[i]>0;i++) //print decimal value
        printf("%02d ",(int)USB_BUF[i]);
    for(;i<8;i++)
        printf(" "); //clear rest of line
    SEND_CMD(LINE4);
    for(i=2;USB_BUF[i]>0;i++) //print table character
        printf(" %c ",tec2char(USB_BUF[i]));
    for(;i<8;i++)
        printf(" "); //clear rest of line

    SEND_CMD(LINE2);
    printf(" ");
    for(i=0x80;i>0;i>>=1) //print flags of byte 0 from msb to lsb
    {
        if(USB_BUF[0]&i)
            SEND_CHAR(0xFF); //print black rectangle
        else
            SEND_CHAR(' '); //print blank space
    }
    printf(" ");
}
```

```

    for(i=0x80;i>0;i>=1) //print flags of byte 0 from msb to lsb
    {
        if(USB_BUF[1]&i)
            SEND_CHAR(0xFF); //print black rectangle
        else
            SEND_CHAR(' '); //print blank space
    }
}

```

Un rápido vistazo al paquete enviado por el teclado durante un par de pruebas muestra que las teclas pulsadas aparecen desde el byte 2 al 7, dando un máximo absoluto de 6 teclas que pueden ser presionadas simultáneamente. Todas las teclas retornan un código de posición, exceptuando las teclas Ctrl, Alt, Shift y Windows, las cuáles aparecen como banderas en el byte 0, 4 para las izquierdas y 4 para las derechas. El orden en el que aparecen las hasta 6 teclas, es el orden en que fueron presionadas. Quien haya trabajado con teclados PS/2 se dará cuenta que su funcionamiento es totalmente distinto a un teclado USB.

Recuérdese que un teclado de PC es leído de forma matricial por el controlador embebido en él, ya sea éste PS/2 ó USB. Por esta razón existe también un máximo de teclas por sector que pueden ser presionadas al mismo tiempo. Si el límite es excedido, el teclado USB responde con un código de error, poniendo los 6 bytes de teclas en 0x01.

La función `decode()` está hecha para un display de 4 líneas por 20 caracteres. En la 2ª línea se despliegan los *flags* del byte 0, representando las teclas Shift, Alt, Ctrl y Windows derecha e izquierda. La 3ª línea muestra los números de las teclas presionadas, mientras que la 4ª línea utiliza la función `tec2char()` para convertir el número de tecla en el carácter ASCII correspondiente. De esta manera puede implementarse rápidamente una conversión número de tecla a carácter, dependiendo de cuáles quiera implementarse en el microcontrolador. Además, debe tenerse en cuenta que la función `tec2char()` debe ser implementada para el teclado que se está utilizando. Un teclado inglés es distinto a uno en español.

A continuación se muestra una implementación matemática de conversión de tecla a carácter, la cual es preferible a utilizar una tabla, puesto que una conversión matemática requiere menos memoria y la diferencia en tiempo de procesamiento es poca para la mayoría de las teclas. Podría implementarse una mezcla de conversión matemática para números y letras, y conversión por tabla para signos de puntuación.

Esta función `tec2char()` retorna el código ASCII para las letras mayúsculas y los números que se encuentran sobre las letras y en el pad numérico de la derecha. Esta función puede ser ampliada por ejemplo para incluir la tecla Shift y diferenciar entre letras mayúsculas y minúsculas.


```

char tec2char (unsigned char c)
{
    if( (c>=4) & (c<=29) )
        return (c+61);
    else if( (c>=89) & (c<=97) )
        return (c-40);
    else if(c==98)
        return '0';
    else if( (c>=30) & (c<=38) )
        return (c+19);
    else if(c==39)
        return '0';
    else
        return 0;
}

```

Otro Ejemplo.

En el CD se suministra un 2º ejemplo el cual detecta la conexión/desconexión de un dispositivo esclavo al puerto USB. Sólo detecta el cambio en la conexión del dispositivo y configura la interfaz del SL811HS apropiadamente, pero no inicializa el dispositivo conectado. Este ejemplo utiliza las opciones de interrupción que provee el SL811HS. Al detectar un cambio en el estado de la conexión, la MSP muestra el nuevo estado en un display LCD. Los estados son:

- Waiting: Ningún dispositivo está conectado.
- Full Speed: Hay conectado un dispositivo de velocidad completa.
- Low Speed: Hay conectado un dispositivo de baja velocidad.

Conexión de Mouse

El mismo programa para leer un teclado USB puede utilizarse para leer un mouse USB. La diferencia radica en el tipo de datos en el paquete de respuesta. El mouse responde con 3 bytes que se describen a continuación:

- Byte 0: Reporte de botones:
 - o Bit 0: Botón Primario
 - o Bit 1: Botón Secundario
 - o Bit 2: Botón Terciario
- Byte 1: Variación del eje X. Valor entre -127 y +127. Positivo hacia la derecha.
- Byte 1: Variación del eje Y. Valor entre -127 y +127. Positivo hacia abajo.