AGRADECIMIENTOS

Mi agradecimiento a las personas que han colaborado en este proyecto ya que sin ellas no habría sido posible su realización: en primer lugar al grupo de Robòtica i Visió del departamento de Electrònica, Informàtica i Automàtica de la Universitat de Girona, que me brindó la oportunidad y los recursos; a mi tutor Lluís Magí, por compartir sus conocimientos y por el tiempo que le ha dedicado; a mi familia y amigos, por su apoyo moral y por los ánimos en los momentos difíciles, en especial a mi madre y a la Señora Lidia y a Manuel Toro, ya que sin su apoyo no habría llegado hasta el final; y finalmente, a compañeros y amigos como Iolanda Bayó, David Moreno o Javier de Haro, por su ayuda, interés y preocupación en que todo saliera perfecto. En definitiva, a todos los que me han apoyado, que son muchos aunque no pueda nombrarlos a todos, y a la gente desconocida de internet, que comparte sus conocimientos de forma desinteresada.

ÍNDICE.

1. Introducción

1.1 Antecedentes

El grupo de robótica y visión por Computador del Departamento de Electrónica, Informática y Automática de la Universitat de Girona desarrolló la “Tarja MAGCL”, un hardware específico para sistemas de visión y control encarado principalmente al tratamiento de imágenes en tiempo real.

Esta tarjeta dispone de una serie de conectores: puertos de entrada y salida para el desarrollo de sus funciones básicas; un conector IDC20 como bus de entrada de video por el que se reciben los datos de las imágenes a tratar; un DSP (Digital Signal Processor) que permite filtrar, digitalizar y tratar la señal en tiempo real para devolver el resultado procesado de nuevo en forma de señal digital por el bus de salida, formado por otro conector IDC20. El DSP puede programarse a través de un puerto serie RS232.

Al igual que muchos otros dispositivos destinados al desarrollo de aplicaciones contiene una FPGA (field-programmable gate array) sobre la que se puede cargar cualquier programa o función combinacional en forma de funciones lógicas simples, programando las interconexiones de sus bloques lógicos mediante un cable de download, un dispositivo SPROM EPC1 o vía JTAG, es precisamente la capacidad de reprogramarla en función del esquema diseñado, lo que la convierte en una herramienta para desarrollar futuras utilidades una vez manufacturada la placa.

Los puertos que componen la placa tienen una función predefinida para realizar su tarea principal de proceso de datos, pero para la realización de nuevas aplicaciones se previó un tercer conector IDC20 que se une con parte del bus de datos de la FPGA.

Las nuevas tareas que se diseñen puede que requieran la comunicación de la placa con otros elementos exteriores como un ordenador personal y según su función utilizar diferentes medios de comunicación.

1.2 Objetivos

En este proyecto se pretende diseñar e implementar un conjunto de periféricos de comunicación para poder utilizarlos como si se tratase de unas librerías hardware y facilitar el desarrollo de nuevos diseños que aportarán mejoras y utilidades añadidas.

Para realizar estos módulos será necesario realizar un bloque lógico en lenguaje VHDL que nos permitirá, independientemente del hardware utilizado, el acceso y control del periférico estudiado. Programando el software sobre el dispositivo de la placa encargado, que en nuestro caso práctico se trata de una FPGA.

Será necesario a su vez el diseño e implementación de una placa de pruebas encargada de adaptar el periférico al conector de nuestra placa, para poder usar, testear y debugar el módulo de software.

Concretando, en este proyecto se realizará el estudio e implementación de los módulos RS232 y USB (universal serial bus), dos puertos de comunicación serie diferentes, uno que funciona en singled-ended y el otro en modo diferencial.

1.3 Estructura del trabajo

En el capítulo 2 se comentan dos elementos principales con los que se desarrolla este proyecto, el lenguaje de programación VHDL con el que se realiza el software para las librerías hardware y el programa MAX PLUS II, un entorno gráfico para diseñar el programa conectando los módulos, testearlo, compilarlo y cargarlo sobre el chip especificando qué pins de éste corresponden a las entradas y salidas del programa.

No se comenta nada del programa PCAD 2001, quizá el tercer elemento más importante del desarrollo porque se deja para los apartados en los que se explique la implementación del adaptador de cada puerto.

En el capítulo 3 se desarrolla el puerto RS232, un protocolo sencillo para empezar a conocer las herramientas de trabajo, el programa de diseño de circuitos impresos PCAD 2001, el lenguaje de programación VHDL y el programa MaxPlus II que confecciona los módulos y los carga sobre la tarjeta gráfica.

Tras realizar un repaso general al protocolo, se explicará como se fabricó la placa física para adaptar las señales de la tarjeta gráfica a las requeridas por el puerto serie y como se implementó el programa para controlarla.

En el capítulo 4 se desarrolla el bus USB, un protocolo que, lejos de conformarse con una simple especificación de las características físicas de la comunicación y una estructura de paquete, define todo el modo de comportamiento de una capa superior para el reconocimiento de un dispositivo y la definición completa de éste al host.

A diferencia de otros protocolos no se trata simplemente de una estructura para la comunicación sino que conlleva una serie de pautas a seguir. Especifica las típicas características físicas, tanto mecánicas como eléctricas, un modelo de control de flujo, una capa de protocolo basada en paquetes de datos y un sistema de trabajo para los dispositivos.

Se explicará como se desarrolló el hardware para el testeo de este puerto, no mucho más complicado a nivel físico de lo que será posteriormente a nivel de programa. A medida que se avance en la explicación del software se irán introduciendo nuevos conceptos del protocolo para su comprensión.

En el capítulo 5 se recogen las conclusiones del proyecto, las impresiones del programador y posibles vías para continuar ampliándolo en un futuro.

2. Introducción al entorno de trabajo

2.1 Introducción al lenguaje VHDL

El lenguaje VHDL son las siglas de la combinación VHSIC y HDL, Very High Speed Integrated Circuit el primero y Hardware Description Language el segundo, que se utiliza para diseñar circuitos digitales. Una vez descrito un circuito podrá programarse sobre la FPGA de la tarjeta gráfica.

Sin entrar en detalle en la nomenclatura de este lenguaje es necesario conocer su funcionamiento para comprender las explicaciones de los programas que controlarán los puertos.

A diferencia de los lenguajes de programación secuenciales, sus procesos actúan en paralelo activados por señales. Estos procesos se agrupan en módulos para dividir el diseño en partes más simples y poder seguir una metodología de trabajo top-dowm o bottom-up. Durante la explicación se hablará de módulos y procesos, que como se puede intuir, forman un todo que sin su visión general, resultaría difícil deducir el comportamiento de cada uno individualmente. Aunque estén divididos según su función, actúan entre ellos activando señales que dificultan su seguimiento por no ser su comportamiento secuencial. De hecho todas las asignaciones de un proceso se realizan simultáneamente.

Cuando se diseña en este lenguaje, se convierte en una tarea muy difícil encontrar un error, pues nunca se conoce en qué momento o dónde se desvió el programa de su rumbo. Si a esto se le añade el hecho de que estamos programando sobre un hardware sin dispositivo de salida visual, a excepción de algunos leds que se puedan colocar en la placa de pruebas, se comprende la importancia de simplificar al máximo las tareas a realizar.

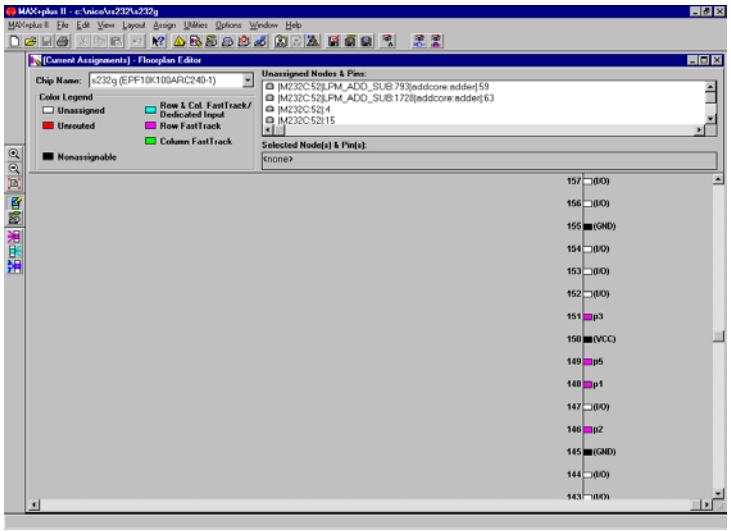
La razón del comportamiento del lenguaje VHDL es imitar un hardware real, de hecho durante la compilación, el programa se convierte en una serie de funciones lógicas simples que pasarán a ser parte del hardware en el momento que se programen sobre el dispositivo de puertas lógicas empleado.

Para establecer las conexiones entre los diferentes módulos existen herramientas con un entorno gráfico en las que se pueden colocar dichos módulos y dibujar las líneas que los unen, a su vez, incluye herramientas de testeo. El entorno sobre el que se trabaja es el programa MAX PLUS II, que tiene incorporadas librerías del chip programable de Altera que se utiliza.

2.2 Introducción al entorno de trabajo MAX PLUS II

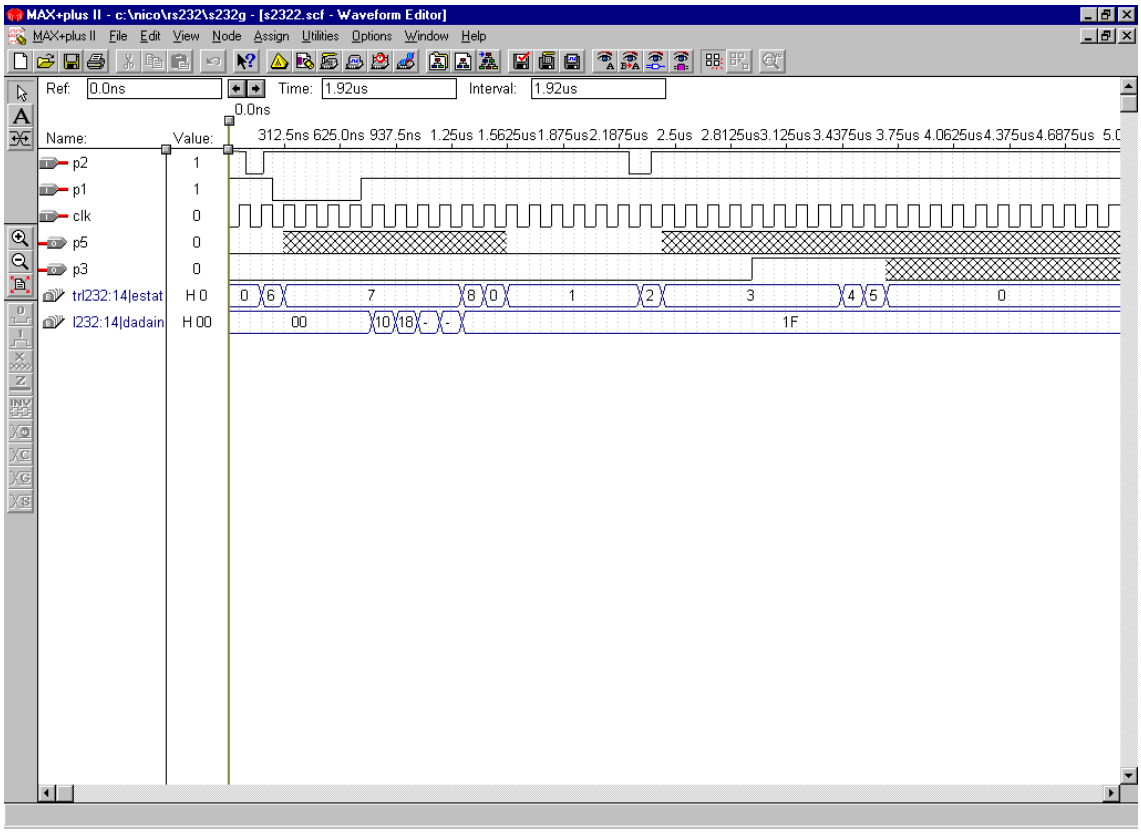
El programa MAX PLUS II es una herramienta para diseñar y programar un circuito digital. Sobre un entorno gráfico permite colocar módulos programados en VHDL e interconectarlos entre ellos.

MAX PLUS II será el programa encargado de compilar y programar la FPGA de la placa. Durante la compilación convierte el programa en una serie de funciones lógicas que posteriormente se cargarán sobre el dispositivo de puertas programables, y para ello tiene una serie de librerías con los posibles chips a utilizar. Habrá que definir los pins del chip que representan las entradas y salidas del programa.



También tiene herramientas de testeo en las que preconfigurando las señales de entrada, imita el comportamiento que tendrá el software. Esto será muy útil en el puerto RS232 cuya especificación solamente considera un tipo de paquete, aunque en el puerto USB no será de mucha utilidad, pues su protocolo incluye instrucciones para el flujo de datos, siendo estos datos paquetes enteros de bits y su comportamiento no depende exclusivamente del dispositivo, sino del host que controle este flujo.

Se debe realizar un esquemático, archivo .sch, en el que se configuren las señales de entrada del programa y se definan qué salidas o variables se quieren visionar. Cuando se ejecuta, el resultado viene a ser como el de la figura.



3. Desarrollo del Protocolo RS232

3.1 Introducción al protocolo RS232

A principio de los 60, el organismo actualmente conocido como Electronic Industries Association (EIA) desarrolló una interficie estándar para la transmisión de datos entre equipos. Para comprender el RS232, hay que tener en cuenta las necesidades de esa época, en que las comunicaciones se establecían entre una computadora principal centralizada y computadoras terminal, o en ocasiones entre dos terminales sin ningún computador de por medio. En la mayoría de los casos, los dispositivos se conectaban mediante la línea de teléfono y requerían módems en los extremos de la transmisión, y también empezaba a ser necesario un estándar que asegurase la comunicación correcta entre dispositivos de diferentes fabricantes. Hasta aquel momento, se empleaban complejos diseños para evitar los errores característicos de la transmisión de datos por canales analógicos, y por ello nació la idea del RS232, uno de los primeros protocolos de comunicación. El RS232 especifica el voltaje de la señal, los tiempos de transmisión, funciones para diferentes señales, un protocolo para el intercambio de datos y para los conectores físicos.

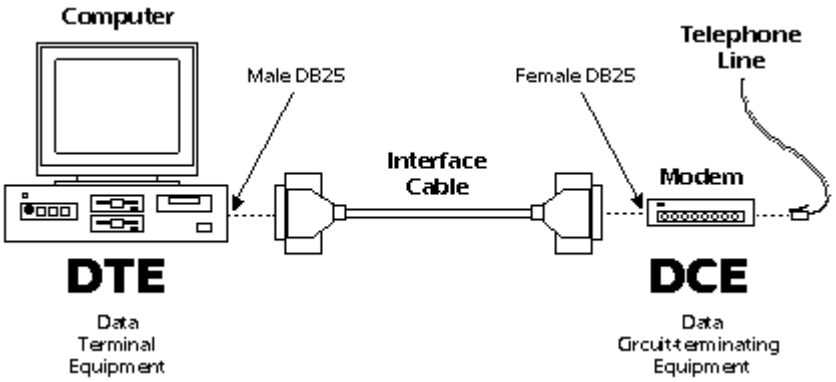
A lo largo de losaños transcurridos desde aquel momento, la especificación ha cambiado en tres ocasiones, llamándose la versión más reciente EIA232F. No solo ha cambiado el nombre del protocolo de RS232 a EIA232, sino también algunas de las señales que usa.

En la actualidad se ha simplificado su diseño para diferentes aplicaciones en las que no era necesario implementar todo el protocolo, aplicaciones que no se previeron en los 60. No ha habido un acuerdo para estos sistemas más simples,por lo que se pueden encontrar dificultades e incompatibilidades usando este puerto, a pesar de que siempre coincida el protocolo, los voltajes y los conectores.

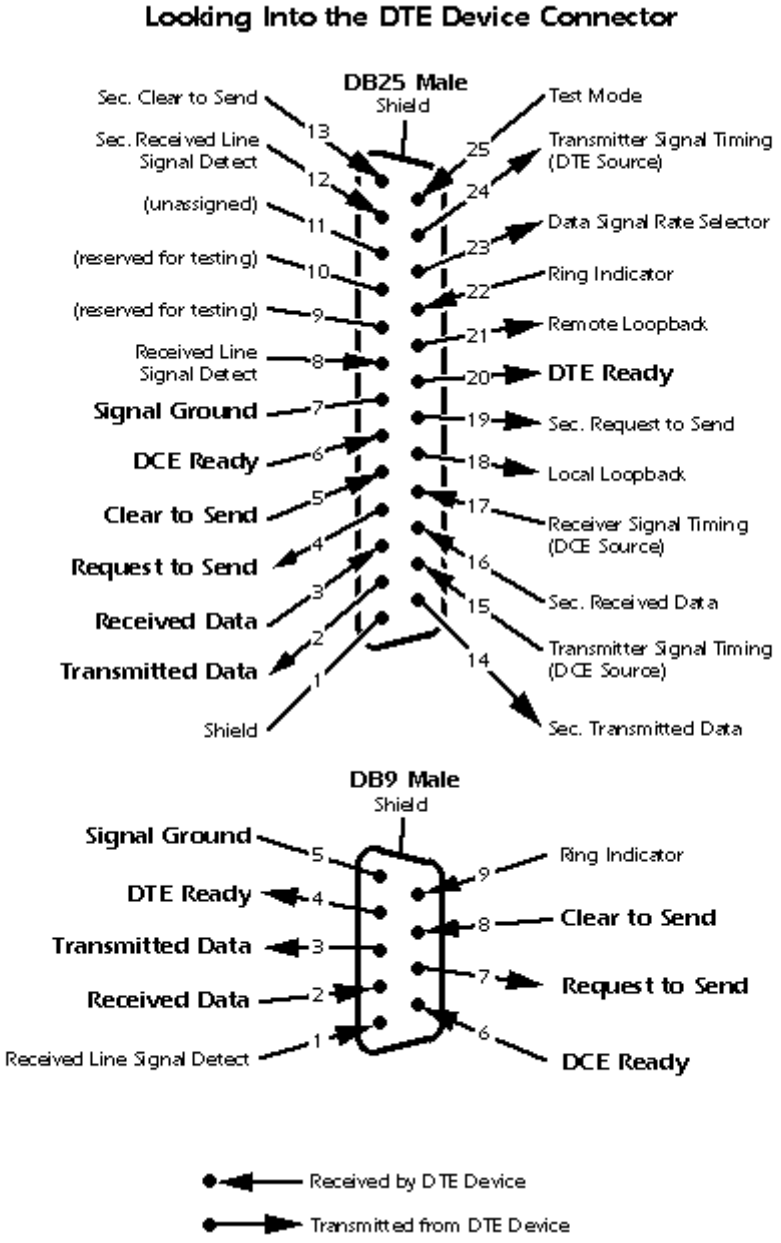
Se pueden encontrar problemas en el control del flujo por desconexión de pines, provocando overflows en el buffer. También es posible encontrar conectados dos dispositivos cuyas funciones DTE o DCE no sean las correctas según el cable y se reviertan las líneas de transmisión-recepción o incluso conectores macho o hembra

colocados de forma incorrecta o con la configuración de pins cambiada, inutilizando las funciones del cable. Los circuitos son suficientemente tolerantes a estas malas conexiones, que podrían incluso conectar pins a masa, aunque en caso de un mal funcionamiento es aconsejable desconectar el cable.

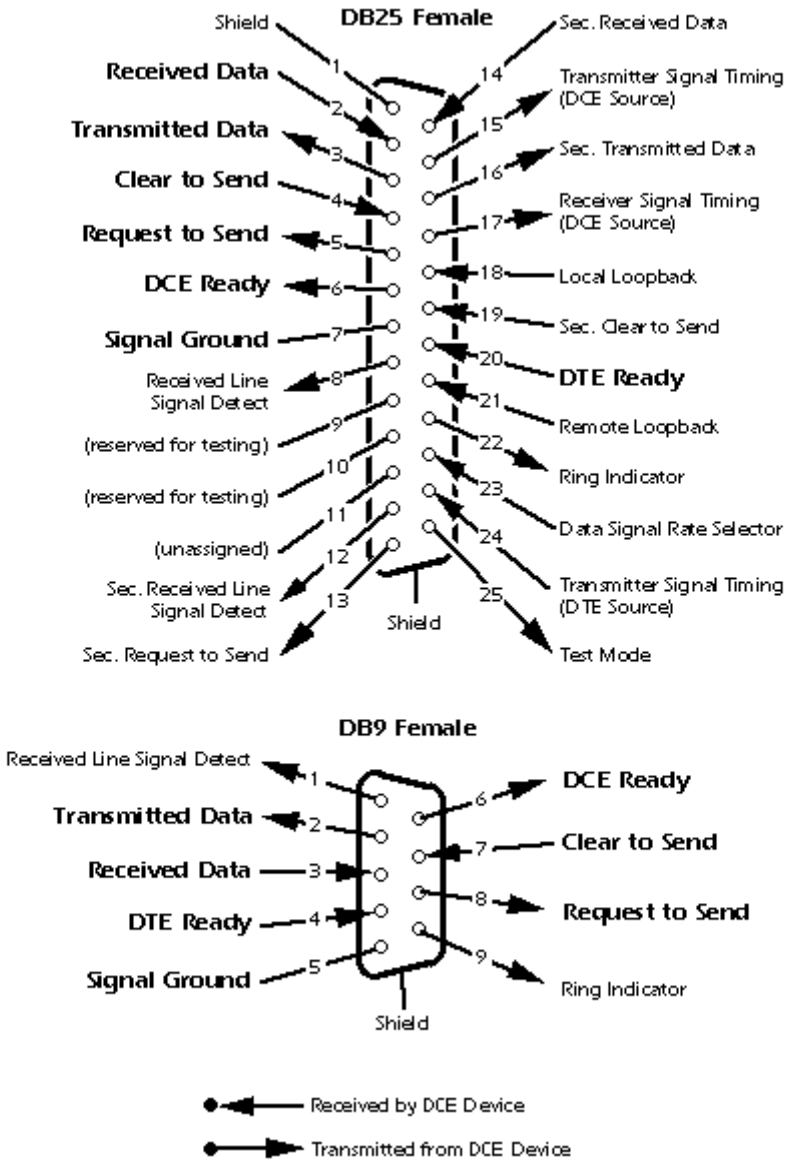
El estándar EIA completo se define en un conector de 25 pins, aunque la mayoría de estos sirven para funciones en las que la comunicación se realiza a través de un módem, por lo que en muchos casos es suficiente utilizar el conector de 9 pins DB9, también incluido en el estándar. Se coloca un macho en el lado de servidor o terminal DTE (Date Terminal equipement) donde usualmente se encuentra un ordenador o Terminal y un conector hembra en el lado DCE (Data Circuit-terminating Equipement) El cable de enlace es un cable con todas la líneas paralelas sin cruces ni interconexiones en los conectores. Si todos los cables siguen este estándar, no habrá errores de conexión.



Cuando se usa el DB25, sus pines realizan funciones como administrar el módem, cambiar los baudios de la transmisión o retransmisiones en caso de encontrar errores en la paridad entre otras, pero para una función no referida a un módem, las señales necesarias son menos.



Las señales que implican una dirección en la transmisión se especifican desde el punto de vista del DTE. Aunque en el DCE deberían mantenerse los mismos nombres y sentidos, no se suele poner en práctica porque nunca queda claro cuando se tratará de un dispositivo DTE o DCE, y así en el DCE se les dan otros nombres en función de su sentido.



Las funciones del protocolo EIA232 se pueden dividir en seis categorías que se nombran a continuación, aunque solo se explicarán con detalle las utilizadas en la versión simplificada para esta implementación. La primera categoría son la señal de masa y la malla, que supone la protección frente a interferencias. La malla es la carcasa del conector, y aunque en principio está separada, normalmente se conecta al pin 5 o masa del conector y a la malla del cable. Por otro lado, el pin de masa puede estar conectado a la masa protegida del dispositivo que conlleva el conector. La segunda categoría es el canal primario de comunicación para el intercambio de datos, con sus señales propias de control de flujo. La tercera categoría corresponde al canal secundario de comunicación mediante el cual se controla el módem, se realizan las

peticiones cuando hay errores y se inicializa el canal principal. La cuarta categoría la constituyen diferentes señales para conocer el estado del módem, chequearlo y controlarlo. La quinta categoría son señales de tiempo de transmisión y recepción para la comunicación síncrona a diferentes baudios. La sexta categoría son señales de testeo del canal para comprobar antes de la transmisión que todo está conectado y capaz de soportar la frecuencia máxima decidida.

En la versión simplificada se ignoran las señales relativas al módem como el pin 1 “received line signal detect” o el pin 9 “Ring indicator”. Otras señales como el DTE Ready están en principio definidas en el estándar para un módem, aunque se les podría dar otro uso, concretamente DTR se coloca a “0” lógico, voltaje positivo, para indicar al módem que se conecte a los circuitos telefónicos. Si se coloca a “1”, el módem se desconecta, cuelga. El DCE Ready (DSR) lo utiliza el módem para indicar al terminal que está conectado a la línea telefónica, en modo de datos (ni voz ni llamada) y que la llamada ha sido realizada y está generando un tono de respuesta,.Otras funciones pueden ser decididas por el ingeniero para estas señales, en función del DCE del que se trate. En la implementación se ofrece la posibilidad de conectar estas señales a la placa de la tarjeta gráfica, aunque lo que realmente interesa son las cuatro señales de la transmisión primaria.

El pin 2 o Transmitted Data, TxD, se activa para enviar datos del DTE al DCE, y si no se están enviando datos se mantiene a “1” lógico, voltaje negativo.

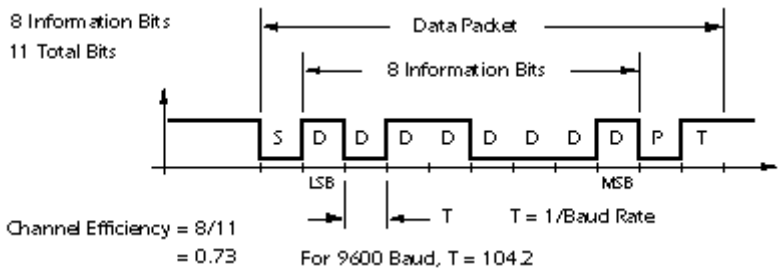
El pin 3 o Received Data, RxD, se activa cuando el DTE recibe datos del DCE, nótese que en el lado del DCE normalmente se etiquetan estos dos pins al revés porque desde su punto de vista la comunicación es a la inversa.

El pin 4 o Request to Send (RTS) se activa a “0” lógico por el DTE para avisar al DCE cuando le quiere enviar datos.

El pin 5 o Clear to Send (CTS) se activa a “0” lógico por el DCE para avisar al DTE que está listo para recibir los datos. En definitiva, son señales para el handshake o el control en la transmisión de datos. En caso de poder ir en ambas direcciones, será el primer dispositivo en activar su RTS el encargado de emitir y el segundo el receptor.

Como observamos, los significados de los pins en cada punto pueden variar en función de quien actúe, pero siempre se mantiene la dirección de la señal o, lo que es lo mismo, quién gobierna ese cable.

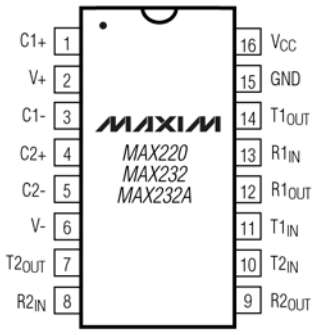
RS232 es una comunicación serie, porque solo puede transmitir los bits uno a uno, y de tipo full-duplex, un cable para cada uno de los dos sentidos de la comunicación, con una tasa de entre 300 y 120.000 bits por segundo o baudios, los más comunes son, 300, 1.200, 2.400, 9.600, 19.200, 38.400, 57.600 y 115.200, y en la implementación la comunicación es a 9.600 baudios. La eficiencia del canal se mide por la cantidad de bits útiles transmitidos, quitando los de formación de paquetes o separación en partes y los de detección de errores. En la comunicación RS232 se envía la información por bytes, con un bit de inicio de paquete y otro de fin, además de un bit de paridad para la detección de errores, es decir, ocho útiles de once o eficiencia 0,73. Esta es la configuración más usual y la que se utilizará en la implementación, aunque puede variar según el dispositivo. Se trata también de una transmisión asíncrona por lo que no hay señales añadidas que controlen los tiempos, el emisor y el receptor deben estar preconfigurados en las mismas características para que la comunicación sea correcta.



3.2 Diseño e implementación del adaptador para el puerto RS232

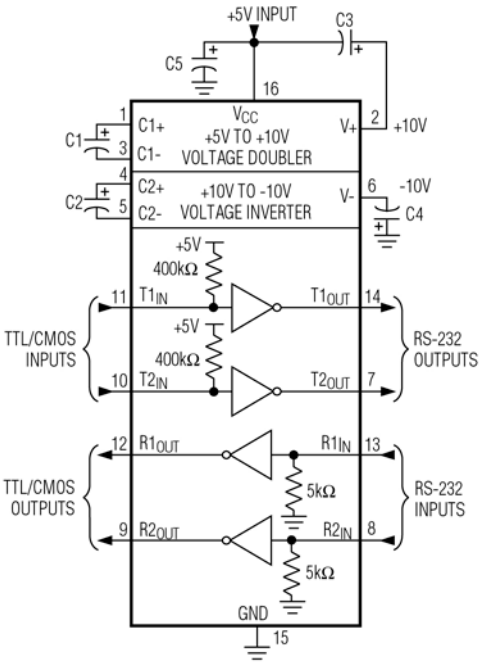
El primer paso previo al diseño consiste en estudiar las tareas que el adaptador hardware debe realizar. Para desarrollar el protocolo RS232 únicamente hay que controlar 4 líneas, dos por las que se reciben y transmiten datos y otras dos para el control de la transmisión, para lo que no parece necesario ampliar elhardware, pues el control puede ser por software. El único inconveniente que hallamos es el voltaje que

utiliza: este puerto serie trabaja a +12v como nivel lógico ”1” y -12v como nivel lógico “0”, mientras que los pines que salgan de la tarjeta trabajarán entre 0 y 5 voltios, y para ello necesitaremos un transistor inversor. El chip que se utiliza es el MAX232 que está alimentado con un voltaje de 5v+, posee un doblador y un inversor de la señal para adaptar las entradas CMOS de 0-5 voltios a las salidas hacia el puerto RS232, y las entradas del puerto serie a las salidas hacia la FPGA de la tarjeta gráfica.

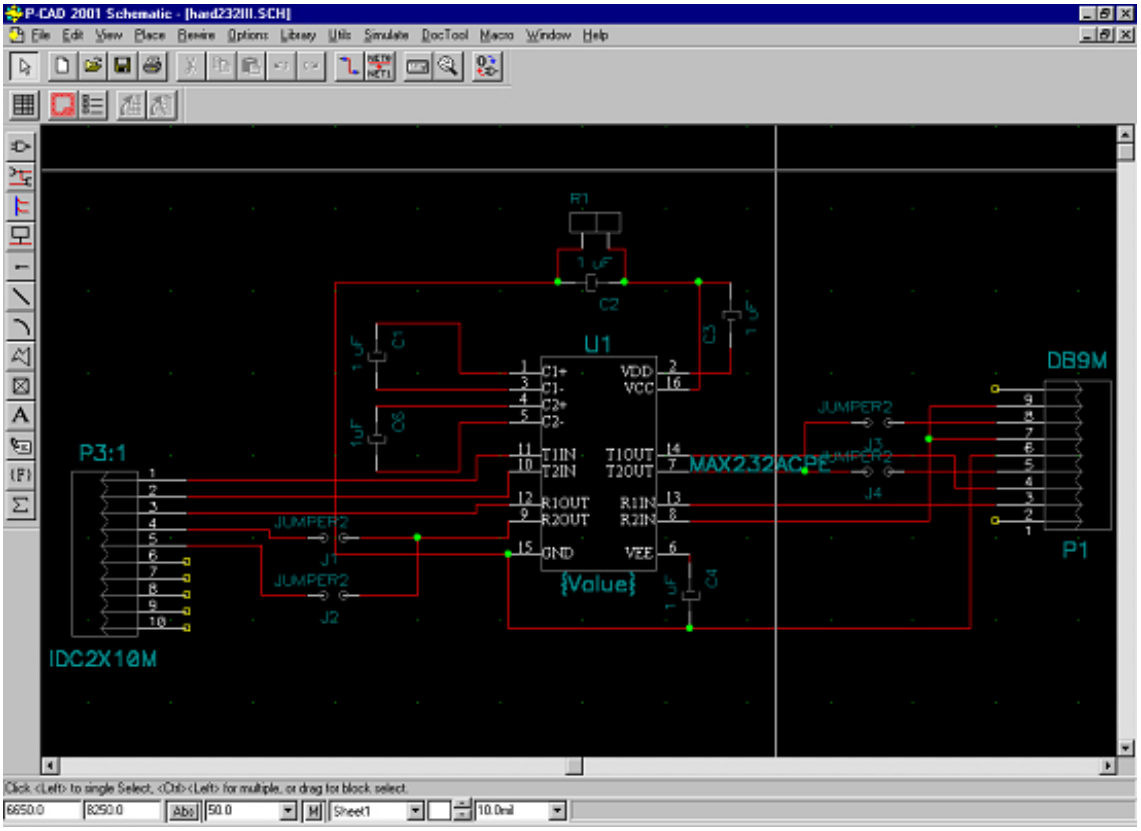


DIP/SO

CAPACITANCE (μF)					
DEVICE	C1	C2	C3	C4	C5
MAX220	4.7	4.7	10	10	4.7
MAX232	1.0	1.0	1.0	1.0	1.0
MAX232A	0.1	0.1	0.1	0.1	0.1

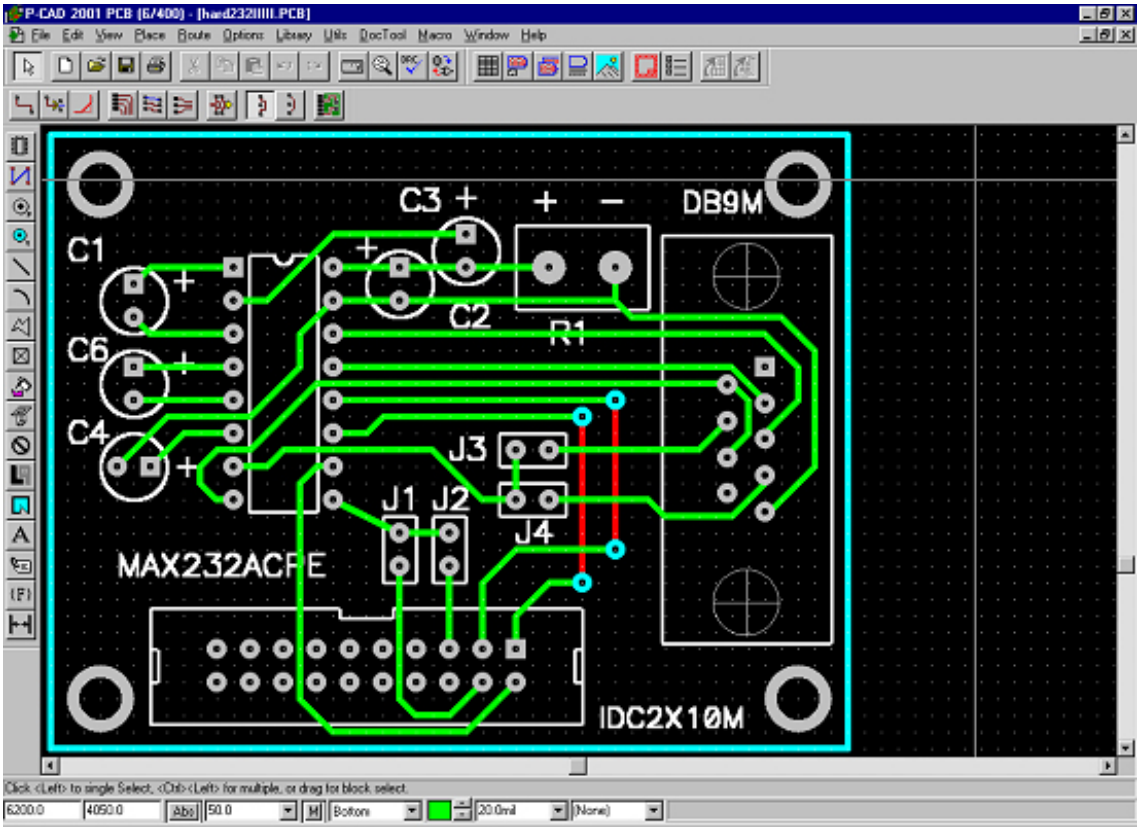


El diseño se realiza a través del entorno de trabajo para circuitos integrados PCAD 2001. En la primera fase se realiza el diseño del circuito con el PCAD 2001 Schematic, donde se decidirán qué elementos intervienen en la placa y cómo se conectan entre ellos sin tener en cuenta como se colocarán físicamente. La placa contendrá el chip MAX232, cinco condensadores de 1.0 μF, tal y como especifica en el datasheet del chip, el conector DB9 para el lado RS-232 y el IDC-10 para la conexión con el bus que proviene de la placa. También tiene un conector para la alimentación y cuatro jumpers para que el control de la transmisión sea a través de las señales RTS y CTS o a través de DTE Ready y DCE Ready.



Una vez realizado el esquema, se genera la netlist, donde quedan especificados los elementos y las conexiones entre ellos. Este archivo sirve para pasar el diseño al formato pcb para su uso en la herramienta PCAD 2001 PCB (printed circuit board), a través de la cual se realizará la impresión del circuito en papel, para su posterior impresión sobre una placa.

Tras repartir de forma adecuada los elementos en lo que más adelante será el dibujo de la placa, se definen los anchos y se dibujan las líneas de conexión, se decide el diámetro de cobre de los pads a través de los cuales luego se realizarán agujeros e interesa que no se pierda todo el cobre en el proceso, y se dibujan sobre el otro lado de la placa las líneas que, de otro modo, inevitablemente se cruzarían. Estas líneas quedan representadas en otro color, en este caso el rojo, y cuantas menos haya, mejor, pues la placa está recubierta de cobre por un único lado y el otro lado deberá realizarse a mano. Como último detalle, el ángulo de las líneas es preferible que sea superior a noventa grados para evitar que se deterioren fácilmente.



3.3 Librería hardware del RS232

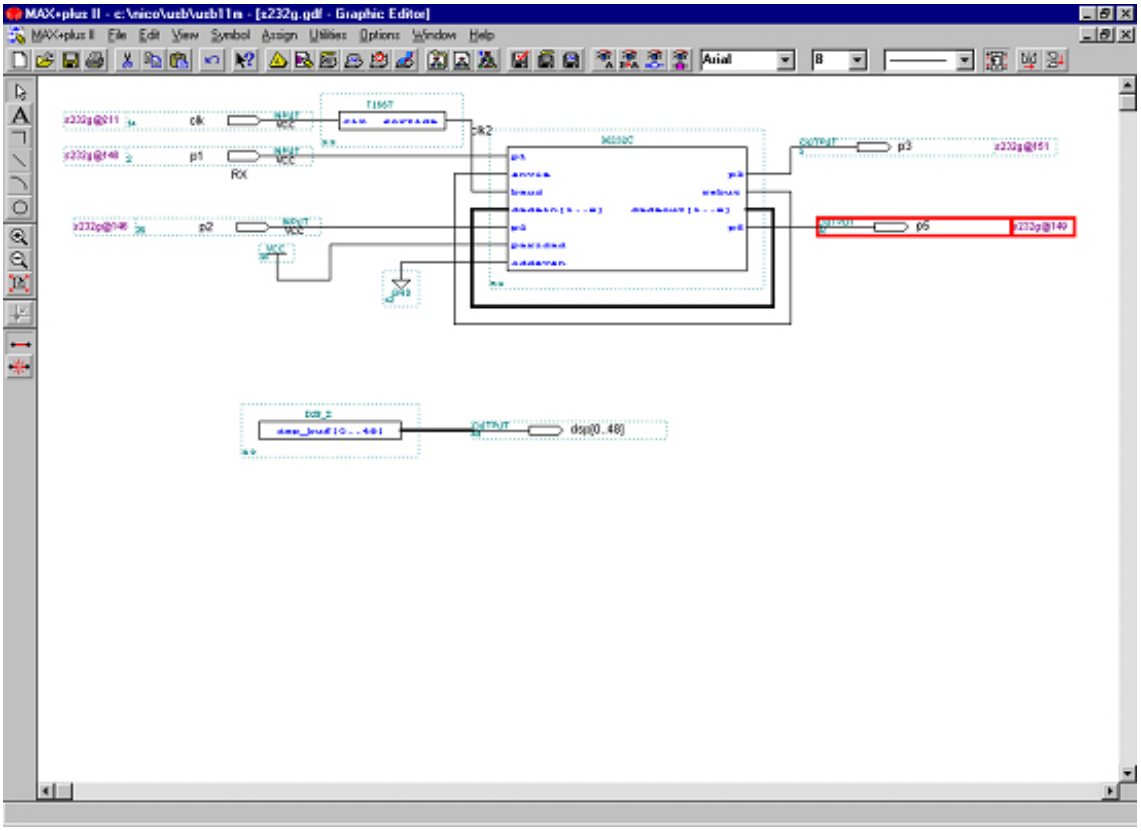
En la implementación del software de control se distinguen 3 bloques diferentes, el primero, DSP_Z, coloca en alta impedancia los pines de la FPGA que se conectan con el DSP para que, en caso de que éste este activado, no se sobrecalienten los chips por cortocircuitos entre ellos. El segundo es el T1667, y se trata de un timer que adapta la señal de reloj de la placa a la frecuencia de trabajo del puerto serie. Como se pretende transmitir a 9600 baudios y la frecuencia de la placa es de 32 MHz, cada período del nuevo reloj será de 3333,33 señales del anterior, es decir 1667 clocks en “0” y 1667 en “1”. Si el reloj de la placa fuese a 24 MHz, cada período sería de 2500 clocks, 1250 a nivel alto y 1250 a nivel bajo. Esa es la tarea del timer, cambiar el nivel de la señal cada semiperíodo para formar un nuevo reloj de menor frecuencia, 9600 baudios.

El tercer bloque controla el puerto serie y consta de tres procesos, el primero se encarga de controlar el modo de transmisión, enviar o recibir datos. Para su testeo se ha preparado el programa de forma que reciba un byte y lo retransmita a

continuación, por lo que será la señal de salida “rebut” situada en el último estado de la recepción la que active el modo de envío, pues está conectada exteriormente a la señal de entrada “envía” que rige este proceso.

El segundo proceso controla la transmisión, en cada flanco de subida del clock interno se ejecuta la maquina de estados. En el estado inicial o estado 0 del modo de recepción, se espera que la entrada p2 o el RTS del DTE correspondiente se active colocándose a “0” lógico, momento en el que p5value y p5control pasarán a ser “0” también o activo. P5value y P5control se utilizan para asignar el valor a las señales de salida, donde P5 es el CTS. En VHDL no se puede dejar una señal en alta impedancia durante un flanco de subida Como eso es necesario en las líneas de comunicación del puerto serie, se controlan en el tercer proceso mediante una señal de control PXcontrol, que decide bien dejar la línea en alta impedancia, “Z”, o asignarle el valor de PXvalue. Una vez el DTE detecta su señal CTS activa, continúa enviando el bit de start “0”. Al recibirlo la maquina de estados pasará al estado 1 en el que permanecerá ocho clocks recibiendo el byte de datos, almacenándolo en el buffer dadaout(i) y calculando la paridad. La paridad cuenta si el número de unos lógicos que se reciben es par o impar. También dependiendo de si el modo preconfigurado de paridad es ODD o EVEN, se representará el par con un “0” o con un “1” y el impar al revés. Este cálculo se puede realizar bit a bit con una suma o XOR de la que solo interesa el bit de resto. En función de si el modo preconfigurado es con o sin paridad,se esperará el bit de comprobación pasando al estado 3 previo al 2 o se pasará directamente al estado 2, donde esperará recibir el bit de stop representado por un “1” lógico.

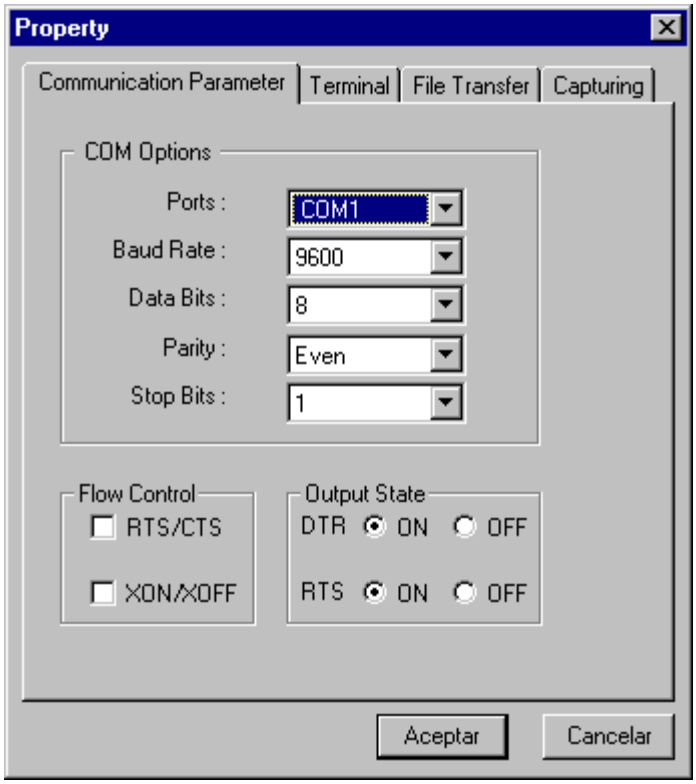
En el estado 3 se activa la señal “rebut” que a su vez activa el envío de datos, es el momento de enviar utilizando la línea P3 que corresponde a la del envío de datos RxD hacia el DTE, esta vez primero se deben colocar en el estado 0 las señales P5value y P5control a “0”, P5 es ahora el RTS del DCE, nuestro dispositivo. En el estado 1 se espera que el CTS se active antes de enviar el bit de start “0” y pasar al estado 2 donde se enviarán los datos y se calculará su paridad. Si la paridad está activa, se pasa al estado 5 y si no al 3, que envía el bit de stop. En el estado 5, en función de si la paridad es odd o even y si el cálculo es par o impar, se enviará el bit de paridad correspondiente y se pasará al estado 3. Desde el estado 3 se pasa al 6 en el que se desactiva el RTS, señal P5.



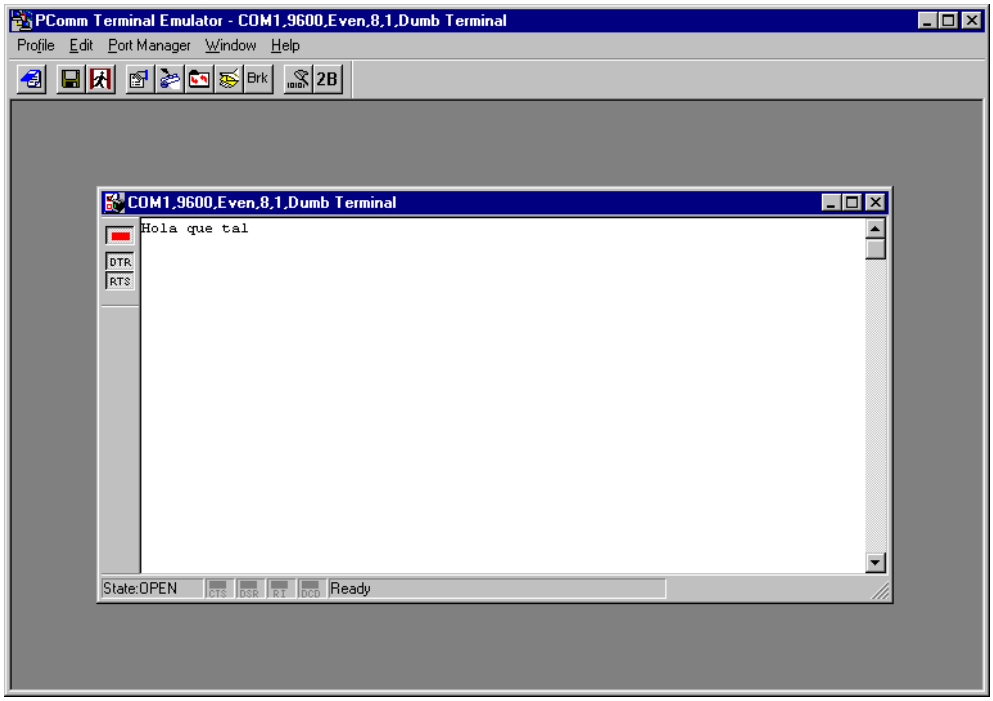
3.4 Comprobación del puerto RS232

Para la el testeo de este puerto se utiliza el emulador del puertos Pcomm Lite 2.3. Previamente habremos implementado la librería, de forma que al recibir un byte se coloque en el buffer de salida y se active la señal rebut, el buffer de salida se conecta al de entrada y la señal rebut a envía, y el módulo reenviará el byte a continuación.

En este protocolo deben preconfigurarse ambas partes con el mismo modelo para que la comunicación sea correcta, es decir, en el módulo la señal paridad está a “1” lo que significa que se enviará este bit, la señal oddeven conectada a masa significa even, par, el clock T1667 lo hace trabajar a 9600 baudios y se pretende recibir un byte, 8 bits. En el Pcomm Lite se debe configurar igual.



El resultado es que cada byte que se envía correspondiente a una tecla, automáticamente se reenvía de vuelta. A medida que se escribe, lo que se recibe en pantalla será como la muestra.



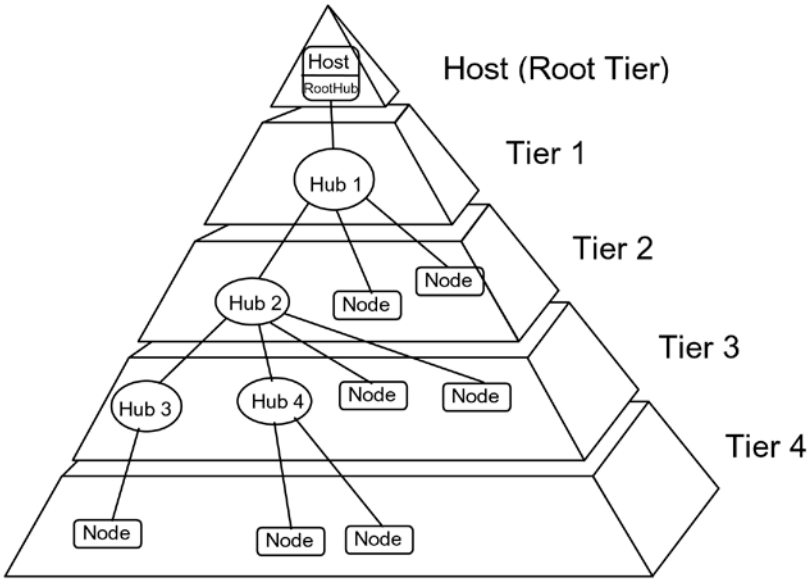
4. Desarrollo del protocolo USB

4.1 Introducción al protocolo USB 1.1

En este capítulo se pretende dar un repaso rápido al protocolo USB dando a conocer su arquitectura y los conceptos más importantes, necesarios para la comprensión de las diferentes partes del proyecto. Muchos otros detalles se añadirán en el resto de apartados, siendo de interés su conocimiento sólo en esa parte del estudio y otros muchos no se llegarán a mostrar, quedando disponibles en la especificación del Universal Serial Bus.

Este protocolo permite el intercambio de datos entre el host y un amplio número de periféricos, siguiendo un esquema basado en el paso de testigo organizado por el host. Permite que los periféricos sean conectados, configurados, utilizados y desconectados mientras el resto sigue operando.

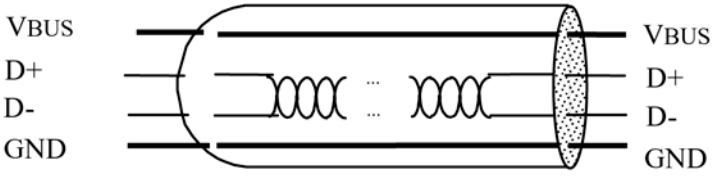
El protocolo USB dispone a los dispositivos físicamente interconectados en formal de árbol con diferentes capas, el nodo de cada ramificación es un hub (repetidor) encargado de aumentar el número de interconexiones con la siguiente capa. En la capa superior de la pirámide tenemos el host, único en cualquier sistema usb del que cuelga solamente su hub (repetidor), único elemento de la segunda capa. En la tercera capa podemos encontrarnos más hubs o funciones, siendo una función el dispositivo final de la comunicación o periférico. El número máximo de capas son siete debido a las constantes de tiempo que se emplean y los retardos que se generan en los hubs, por lo que en la séptima capa ya solo pueden quedar funciones, habiendo pasado la comunicación por 5 hubs antes de llegar a esas funciones.



Topología del Bus

El host inicia las transferencias, mediante polling (sondeo, llamada) pasa el testigo a los diferentes dispositivos, que en función de su respuesta aceptan o no la petición.

El cable que se usa para las conexiones es de cuatro hilos, uno conectado a voltaje (Vbus), otro a masa (GRD) permitiendo al cable alimentar el dispositivo en el caso que sea necesario, y dos cables que transportan la señal diferencial (D+,D-), permitiendo trabajar en tres posibles velocidades: high-speed que transfiere a 480 Mb/s, full-speed a 12 Mb/s y low-speed a 1.5 Mb/s, ésta última destinada a dispositivos que utilicen poco ancho de banda y no ralenticen el sistema.



Cable USB

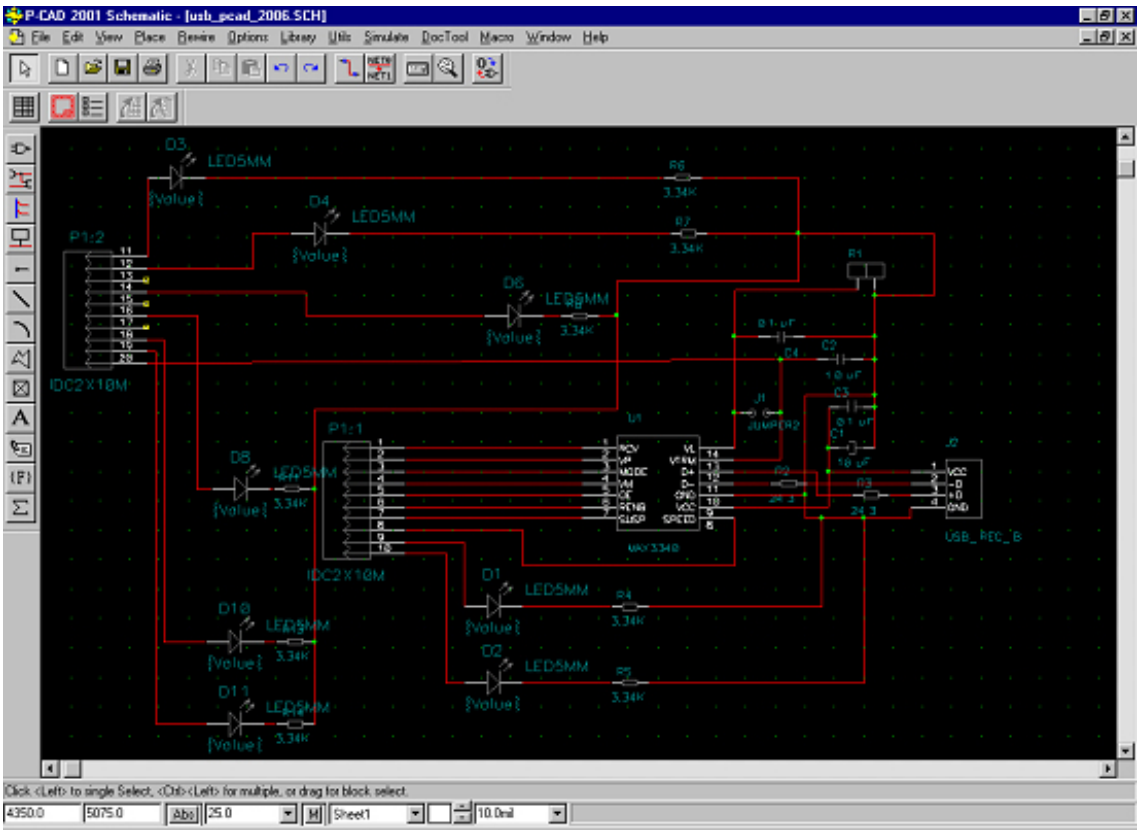
4.2 Diseño e implementación de la placa adaptador del bus USB

Para diseñar la placa se utilizan las herramientas que ofrece el entorno de diseño para circuitos impresos PCAD 2001. En la primera fase se realiza el diseño del circuito con el PCAD 2001 Schematic, programa con el que determinaremos los elementos a utilizar en la placa y como están interconectados, sin tener en cuenta aquí los aspectos físicos de la implementación.

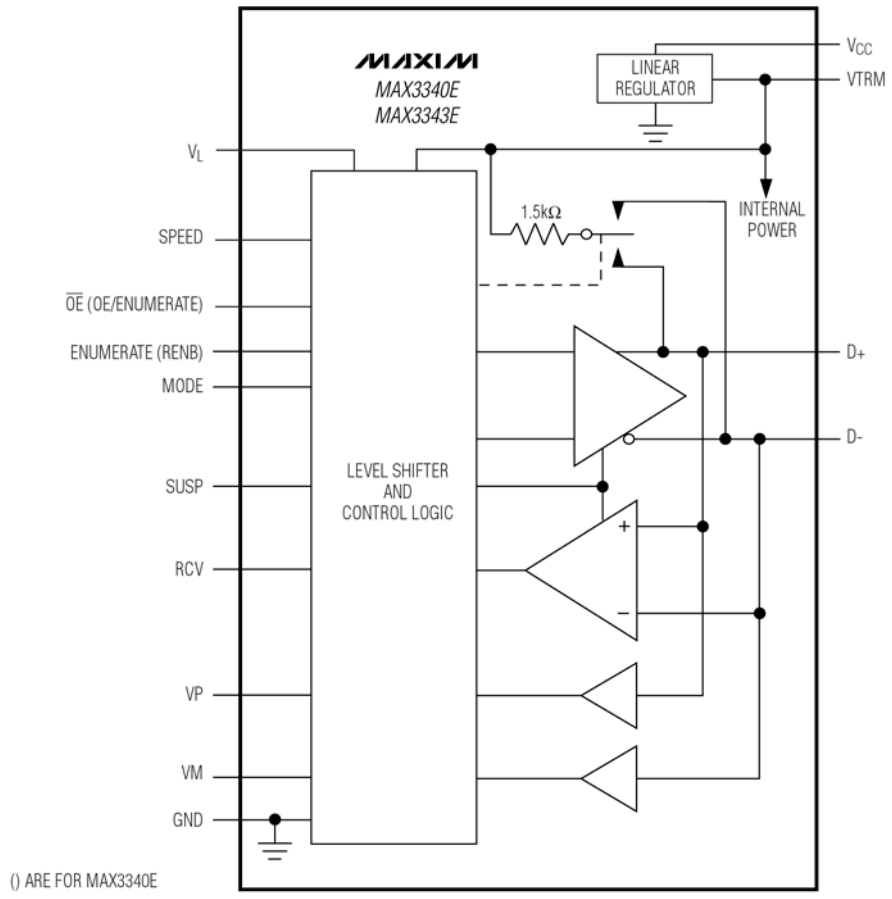
Utilizando las librerías de elementos, se colocan las piezas previamente diseñadas. El objetivo de esta placa es adaptar la señal diferencial que proviene del bus usb a las señales digitales que utilizaremos como entradas de la FPGA y para ello se utilizará el circuito integrado MAX3340.

En un extremo de la placa estará la conexión usb y en el otro, el conector IDC20 por el que se conectará el circuito a la tarjeta gráfica. Los 8 ocho leds conectados a masa a través de sus resistencias son señales para testear y debugar el programa de la FPGA. Los condensadores de desacople se colocan siguiendo las especificaciones del chip y por último las líneas D- y D+ deben estar conectadas al bus a través de dos resistencias de 24,3 Ohms para cumplir con la impedancia en la línea especificada en el protocolo USB. El esquema del chip es el siguiente:

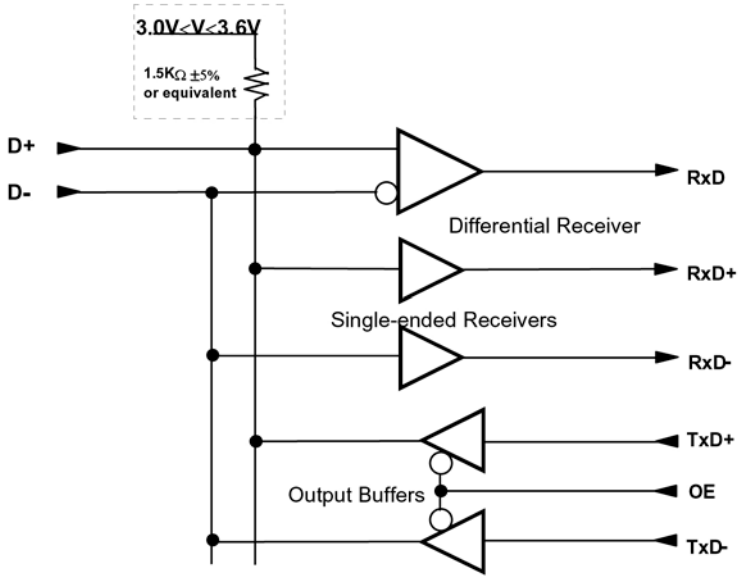
Quedando el diseño en el PCAD 2001 de la siguiente forma:



Las señales D+ y D- del chip se conectan al conector USB a través de sus resistencias, internamente están conectadas a dos diferenciadores operacionales, uno para el envío y otro para la recepción de la señal diferencial, y a los dos diodos receptores singled-ended.



Según la velocidad de transmisión a la que se trabaje, los voltajes utilizados son distintos: en el caso de full speed D+ se conecta a 3,3 V a través de una resistencia de 1,5 K; en low speed será D- la conectada. El diferencial debe ser siempre superior a 200 mV.



Full-Speed port transceiver

VCC y GRD se conectan a las líneas de masa y voltaje del bus, el chip provee una salida VTRIM de señal regulada a 3.3 V, alimentada por el VCC del bus USB que permite alimentar dispositivos externos. Se recomienda que sean tolerantes a +5V o que tengan una protección zener para una posible caída del usb. Colocaremos un jumper entre esta señal y la entrada VL que provee el voltaje al lado lógico del chip por si se quiere auto alimentar este (1,8 - 3,6 V).

RCV, VP, VM, MODE, OE, RENB, SUSP y SPEED son las señales que salen del bloque lógico del chip para gobernarlo y comunicarse con la FPGA de la placa.

RCV es la salida del receptor que responde a la señal diferencial.
VP y VM son las entradas / salidas de datos del lado lógico, en función de OE/ENUMERATE, que controla la dirección de la transmisión, actuarán como entradas, si este pone a ‘0’, o como salidas si se pone a nivel alto.

El MODE nos indica el modo de funcionamiento, si está a 0 se selecciona el modo “single-ended” y si está a 1 se selecciona el modo diferencial.

Como se ha comentado, la entrada OE/ENUMERATE a nivel alto activa las señales de salida, a nivel bajo activa las señales como entradas y en alta impedancia desconecta la resistencia de pull-up interna, lo que es útil cuando se entra en estado suspendido para dejar la línea totalmente libre y consumir menos.

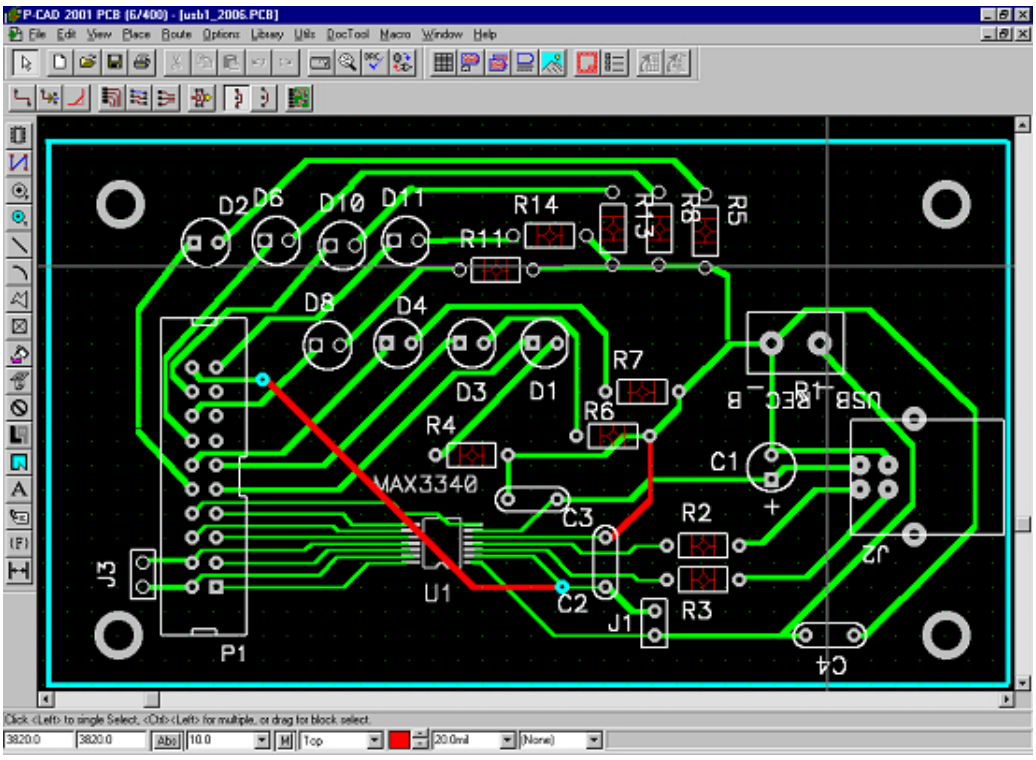
RENB es el enable del receptor, si está a nivel alto RCV, VP y VM responden a D+ y D-, si está a nivel bajo solo responde RCV, VP y VM quedan en alta impedancia. Suele estar conectado a OE/ENUMERATE.

La entrada SUSP a “1” activa el estado de bajo consumo, en ese momento debe consumir menos de 200mA de VCC, y deja OE/ENUMERATE en alta impedancia, fuerza RCV a nivel bajo, las señales D+ y D- quedan en alta impedancia pero VP y VM continúan comportándose como receptores y se sigue escuchando del bus, se pone a “0” para el uso normal.

La entrada SPEED selecciona la velocidad, a nivel alto se selecciona full-speed que trabaja a 12Mbs y conecta la resistencia de pull-up interna de 1,5K a D+. Si se pone a nivel bajo se selecciona low-speed (1,5Mbs) y la resistencia se conecta a D-.

Una vez realizado el esquema, se genera la netlist, donde quedan especificados los elementos y las conexiones entre ellos. Este archivo sirve para pasar el diseño al formato pcb para su uso en la herramienta PCAD 2001 PCB (printed circuit board), a través de la cual se realizará la impresión del circuito en papel, para su posterior impresión sobre una placa recubierta de cobre.

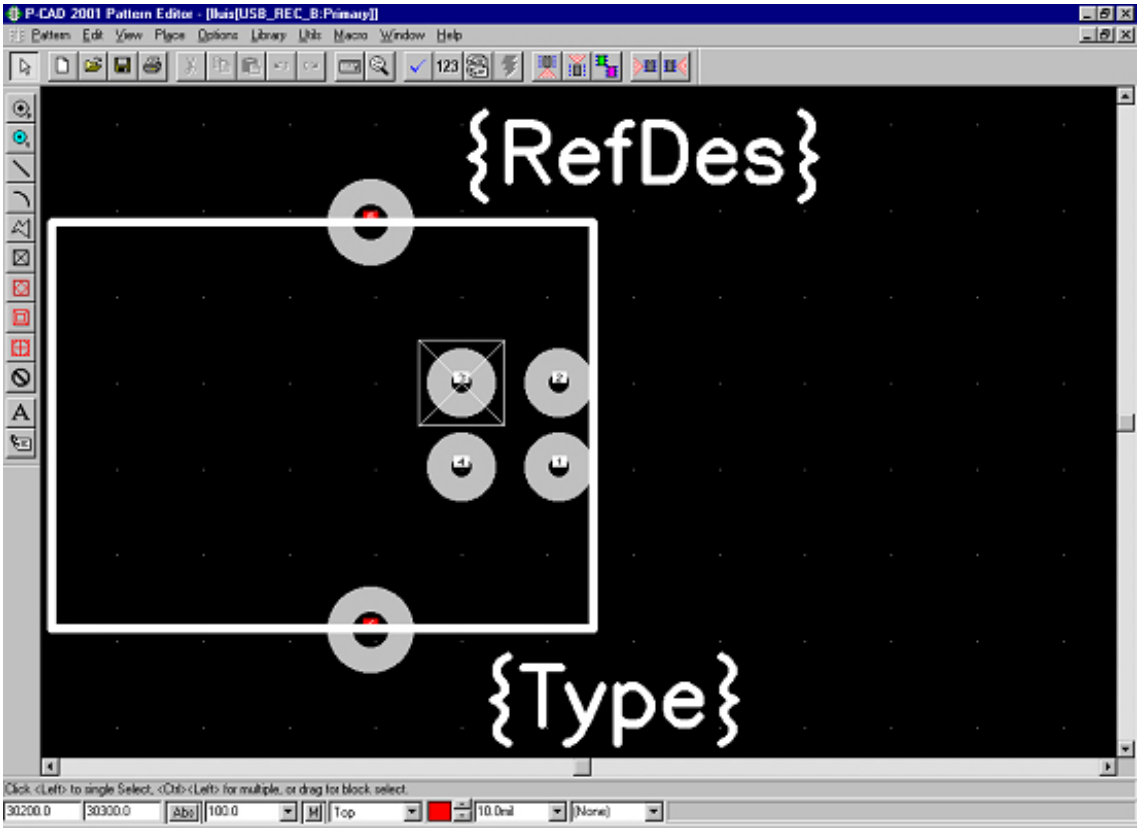
Una vez distribuidos los elementos y dibujadas las conexiones de cobre, el esquema a imprimir es el de la figura.



Básicamente hay que definir el ancho de las líneas 20 mils para las normales y 12 para las más finas que llegan al chip, dejar un buen grosor de cobre en los pads (agujeros de la placa) para que luego no fallen las conexiones al perderse cobre durante el proceso de perforación, y decidir cómo se repartirán las líneas en la capa superior y en la inferior de modo que no se crucen y no dejar ángulos rectos que se deterioran mas fácilmente.

Tras imprimir la cara más compleja de este circuito en una placa de cobre de una sola cara, la parte superior hay que realizarla a mano con alambre. En este caso serán simplemente las dos líneas rojas, el elemento J3 se colocó con posterioridad al diseño, para facilitar la tarea práctica de comprobación de las dos señales VP y VM durante el testeo.

Otra herramienta que se utilizó durante la implementación fue el PCAD 2001 Pattern editor que permite diseñar elementos como el conector USB que no se encuentren en las librerías. Se dibuja la base del conector y se definen las distancias entre los pads y el grosor de cobre de su circunferencia.



4.3 Implementación del software para el control del bus USB

Para estructurar el control del bus USB hay que definir previamente todos los módulos o bloques que formarán el sistema, y organizar los procesos que desarrollará cada uno de estos módulos.

En un primer intento se organizan los bloques imitando un hardware real, como pretende el lenguaje VHDL, dividiendo los módulos en las diferentes partes que compondrían una placa física. Sería necesaria una unidad de control que dirigiera el sistema a partir de unas señales que le indicaran el estado en que se halle en cada momento, un buffer de entrada para almacenar los datos recibidos, un módulo de proceso de datos para tratar el mensaje recibido, otro módulo de control de errores

inmediatamente después del buffer de entrada para descartar rápidamente los paquetes de datos erróneos, un buffer de salida para colocar los datos una vez decidido el paquete a enviar y una memoria RAM como dispositivo de almacenamiento.

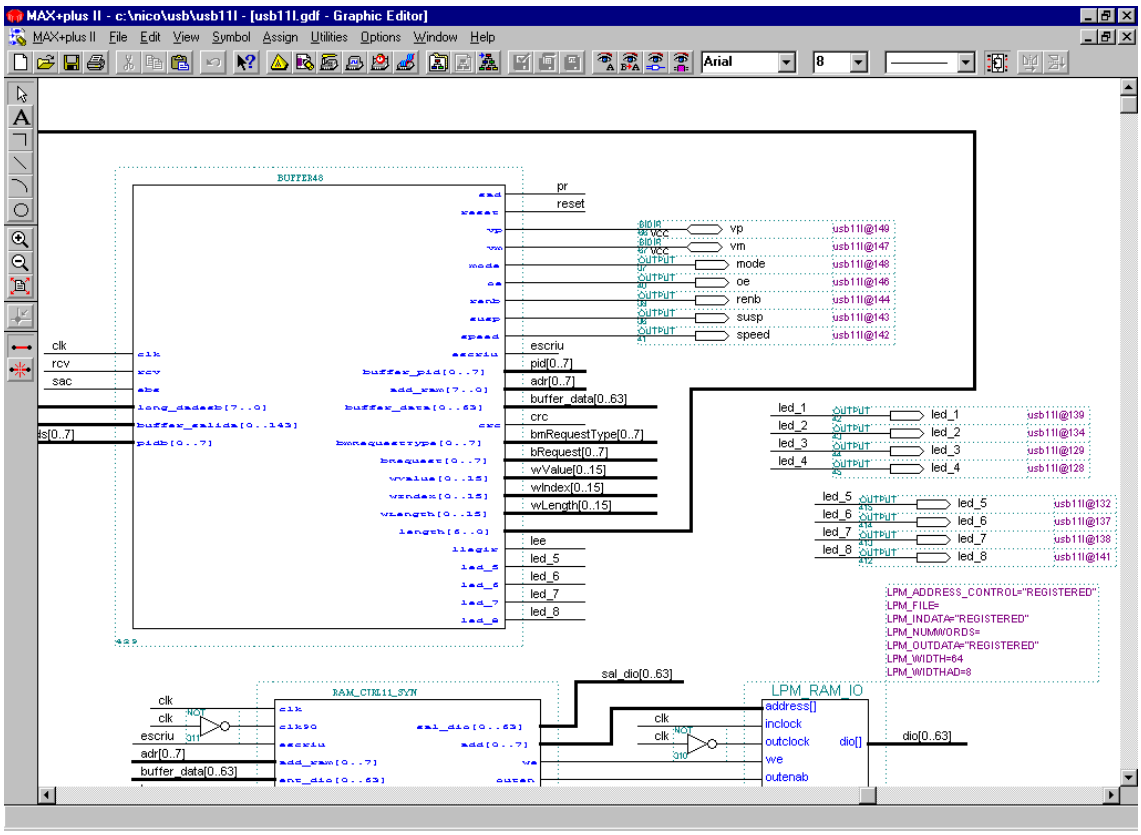
La ingobernabilidad del sistema descrito en el párrafo anterior, con una gran cantidad de señales que deben actuar en el momento preciso, nos obliga a pensar en un sistema mucho más sencillo. Uno de los principales problemas es el tratamiento del bus de datos, que suele ser de entrada-salida debido a que los datos pueden circular en uno u otro sentido, pero los diferentes módulos tienen que estar perfectamente sincronizados para no acceder a él simultáneamente. Por otro lado se dan casos en que los datos deben permanecer disponibles en el bus durante un período suficiente de tiempo para que varios módulos actúen sobre ellos. El lenguaje VHDL o por lo menos la interficie MAX PLUS II no ofrece soluciones suficientes para mantener este buffer con los datos estables en el bus cuando lo gobiernan varios bloques.

El modelo final se compone de dos módulos principales, pensados a partir de las diferentes capas en las que se basa un sistema de comunicación. Las dos capas a implementar serán la capa física que se concentra en la recepcióncorrecta de la señal y la composición del paquete de datos y en segundo lugar, la capa lógica encargada del control del proceso a partir de los diagramas de estados definidos.

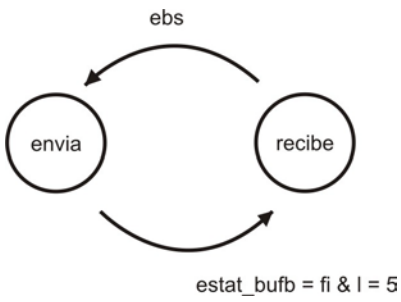
Otros bloques secundarios que ayudan en tareas auxiliares son el módulo que controla el clock del sistema, una memoria RAM lógica y un módulo controlador de esta memoria.

4.3.1 El módulo de capa física.

Este módulo se encarga de las funciones que se realizan sobre la señal física, es decir, la recepción y el envío de datos, la descomposición del paquete recibido en sus diferentes partes, la corrección de errores desestimando los paquetes incorrectos y el almacenamiento de los datos para ser tratados en otros bloques. Se han dividido estas funciones en cuatro procesos.



Un primer proceso típico en muchos de los módulos implementados en VHDL se encarga del encendido y apagado del módulo, o mejor dicho de los demás procesos de este modulo. Básicamente se sincronizan los diferentes módulos a través de una señal para que actúen en el momento que les corresponde. En este caso el bloque permanecerá encendido constantemente a la escucha del bus y cambiará al modo de envío cuando detecte la señal de envío de datos (ebs), que proviene del modulo de control. Al finalizar el envío, estado del buffer “fin” y tras enviar el quinto bit en este estado, I=5, vuelve al modo de recepción.



Maquina general del buffer

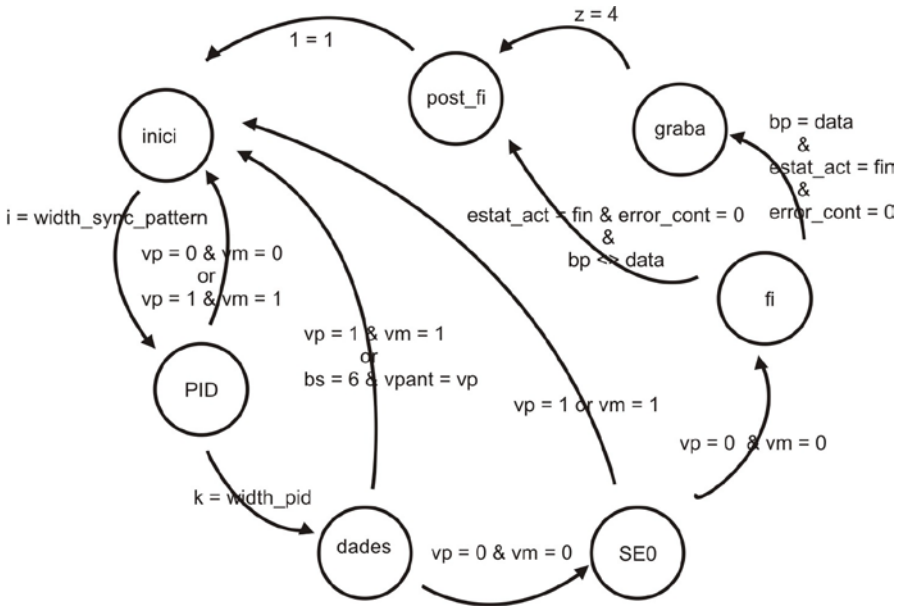
Un segundo proceso se encarga del control de la placa física implementada, activando las señales necesarias que controlan el chip físico según nos encontremos en modo de recepción, en modo de envío de datos o en estado suspendido del sistema.

Adicionalmente en el modo de envío se coloca en el buffer de salida el bit a enviar, este bit se coloca aquí porque en VHDL no es posible controlar un buffer tristate durante el flanco de subida (rising_edge), VHDL no permite dejar la línea de entrada-salida en estado de alta impedancia si no se controla por una señal completa y no únicamente por su rising_edge. En este proceso se trabaja con la señal clock completa.

Concretando se colocan los pins de la placa susp a “0”, speed a “1” y mode a “0” que permanecen igual tanto en el envío como en la recepción. Cuando recibimos oe y renb se ponen a “1” y a “0” cuando enviamos, los datos los colocamos en vp y vm que representan D+ y D- en el lado USB del chip.

El tercer proceso se encarga de la recepción y envío de datos, la descomposición de estos y su almacenamiento. Para simplificar el control y la coordinación con otros bloques se decide que el dispositivo permanezca a la escucha del bus de datos en todo momento hasta que reciba la señal de envío de la unidad de control, momento en el que empezará a trasmitir el paquete de datos para volver a la escucha inmediatamente después. Favorece la elección de este sistema el hecho que nunca deberá emitir dos paquetes seguidos debido a que las transferencias están dirigidas por el host y siempre se recibirá mínimo un paquete después de cada paquete enviado, ya sean paquetes de acknowledge (ACK) o de testigo (TOKEN).

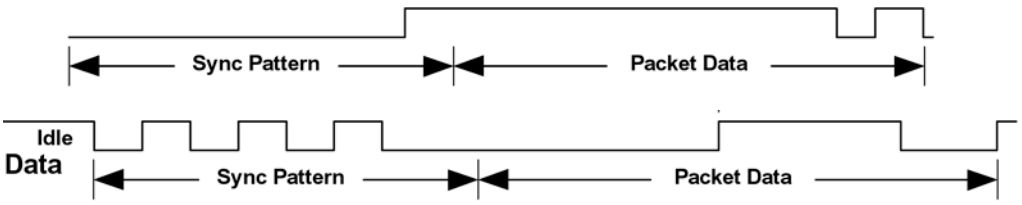
Se divide pues el proceso en dos grandes partes el envío y la recepción, siendo la recepción el estado inicial en el que se encuentra el dispositivo.



Máquina de estados de la recepción de datos.

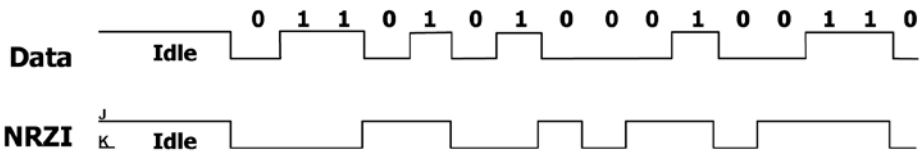
En el estado inicial de este proceso (“inici”) se pueden recibir dos tipos de señales a atender, la señal de sincronización (SYNC) o un reset. La primera indica que los bits que vienen a continuación forman un paquete de datos y la segunda sirve para devolver el dispositivo a su estado inicial (default).

La señal de sincronización aparece en el bus como un estado IDLE (alta impedancia) seguido de la cadena binaria “KJKJKJKK”, en su codificación NRZI.



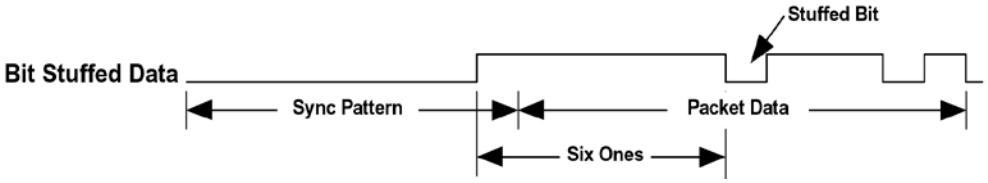
Señal de inicio de paquete (SYNC)

Los paquetes de datos en el protocolo USB van codificados en el sistema NRZI, consiste en cambiar el nivel de la línea cuando se trata de un “0” lógico y mantenerlo igual en caso de un “1” lógico, lo que significaría que en una cadena de unos la línea permanecería constantemente en el mismo estado.



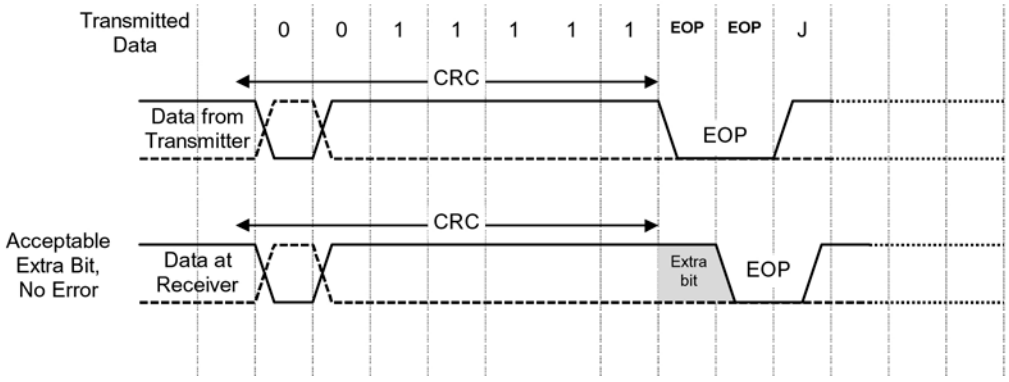
Codificación NRZI

Para asegurar una señal adecuada en la transferencia de datos y una buena sincronización del clock se emplea el “bit stuffing”, que consiste en insertar un cero cada 7 unos lógicos, de este modo se fuerza un cambio en la línea al menos cada 7 bits. Cuenta como el primer “1” el ultimo bit de la cadena de sincronización, que antes de la codificación NZRI son 7 ceros y un uno, no hay excepciones posibles en la inclusión de este cero ni aun tratándose del último bit del paquete antes de la señal de final de paquete (EOP).



Bit stuffing

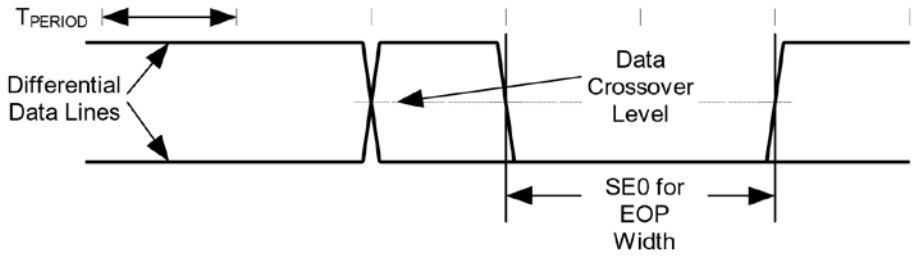
Puede darse el caso práctico en el que aparezca un séptimo uno justo antes de la señal de final de paquete, pero no se trata de otro bit sino de un alargo del último uno que se puede producir en algún hub al pasar a la señal de fin de paquete EOP (End of packet), el hecho de detectar éste posible error justo al final del paquete nos permite desestimar este último bit sin considerarlo erróneo.



Extra bit

La segunda señal que se puede recibir en el estado inicial es el reset, que esta formado por lo que se denomina un SE0 (single-ended cero) de 50 ms, que no tiene porque ser continuo. En caso que no lo sea, los intervalos sin SE0 no deben ser superiores a 3 ms. Un dispositivo que detecta una señal de reset durante mas de 2,5 μs puede considerarla ya como tal y debe finalizar su tratamiento antes de que termine dicha señal.

Una señal SE0 no es más que poner a cero las dos líneas D+ y D- el tiempo que duraría un bit.



Final de paquete

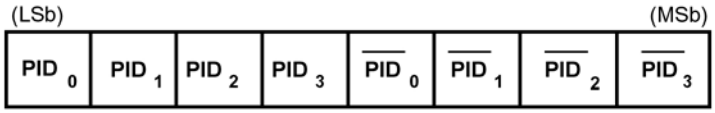
Debe considerarse también en este estado de escucha del proceso que si la línea permanece en alta impedancia durante mas de 3 ms, el dispositivo empezará la transición al estado suspendido (suspended) que deberá haber completado a los 10 ms, consumiendo a partir de ese instante solamente la corriente especificada para el estado, y sale de él en el momento que detecta de nuevo actividad en el bus y en un máximo de 20 ms debe volver a su estado normal de funcionamiento.

En el caso que se reciba la señal de sincronización, se debe proceder a la recepción del paquete datos, entrando en el estado del programa “PID”.

Un paquete puede estar formado por 3 partes o menos, en función de su tipo, acabando todos con la señal de fin de paquete. Los paquetes se emiten en bytes, cuyos bits se emiten y reciben empezando por el de menor peso (LSb) y acabando por el de mayor peso (MSb).

La primera parte del paquete es el PID (Paquet identifier field) formado por 8 bits, 4 bits que nos indican qué tipo de paquete es, y con ello qué formato y

queésistema de corrección de errores tendrá, y 4 bits que aseguran la recepción correcta de los primeros. Estos 4 últimos bits son los complementarios de los primeros, se duplica la información para la detección de errores. Las partes que siguen al PID varían en función del tipo de paquete.



Identificador de paquete

En la capa física no es necesario preocuparse de la información del paquete que se está recibiendo, de eso se encargará la unidad de control, en este módulo simplemente se descompone el paquete en partes. Veamos los posibles paquetes que podemos encontrarnos en cuanto a su estructura y no utilidad.

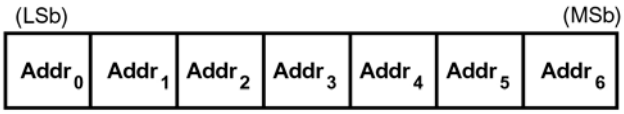
Tipo de pid	Nombre	PID [3..0]	Descripción
TOKEN	OUT	0001B	Dirección+ nº endpoint
	IN	1001B	Dirección+ nº endpoint
	SOF	0101B	Start-of-frame
	SETUP	1101B	Dirección + nº endpoint
DATA	DATA0	0011B	Paquete de datos
	DATA1	1011B	Paquete de datos
HANDSHAKE	ACK	0010B	Paquete correcto
	NACK	1010B	No se pueden recibir o enviar datos
	STALL	1110B	Endpoint inoperativo o petición no soportada
SPECIAL	PRE	1100B	Inicia el trafico en dispositivos low-speed

Tabla de tipos de paquete

En la transmisión en full-speed solo se usan los tres primeros tipos de paquete, el primer tipo que se define son los paquetes de token.

Para dirigirse a una función o endpoint, el host emite un paquete de token y para direccionarla correctamente necesita el campo de dirección del dispositivo y el nº de endpoint dentro de este dispositivo. El campo de dirección está formado por 7 bits y es único para cada dispositivo, por lo que el numero máximo que se puede tener en un

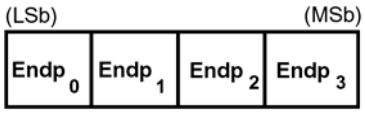
sistema USB son 127 sin contar la dirección 0. Al conectarse cualquier dispositivo, empieza con la dirección 0 y se le asigna otra en el proceso inminente de enumeración.



Campo de direccionamiento

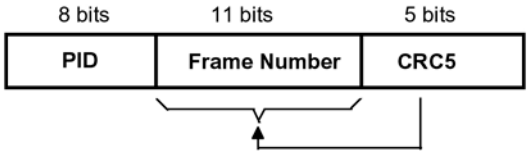
Si el host quiere recibir un paquete, emitirá un token out; si quiere enviarlo emitirá un token in y si el paquete a enviar es uno de los paquetes estándar para el control del dispositivo, este será un paquete de SETUP.

Puede ser que un dispositivo tenga diversas funciones y que cada una de ellas requiera uno o más endpoints, que se pueden definir como buffers lógicos de recepción de datos. El numero de endpoint está formado por 4 bits, por lo que el numero máximo que puede tener un dispositivo son 15, sin contar el endpoint por defecto 0.



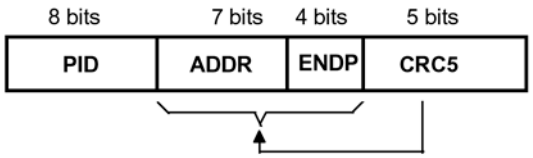
Campo de endpoint

Los paquetes de token tienen, por lo tanto, un total de 11 bits de datos, aunque en el paquete SOF (start-of-frame), los 11 bits representan el número de frame en la que nos encontramos. El sof es un tipo de paquete especial que se usa solamente en el modo de transferencia asíncrono, en el resto de modos se ignora, aunque secundariamente evita que los dispositivos entren en estado suspended. En la velocidad de transmisión Full-Speed se emite al inicio de cada frame, siendo un frame una división en el tiempo de 1ms y en la velocidad High-Speed se divide el tiempo en micro-frames enviando un paquete SOF cada 125 microsegundos.



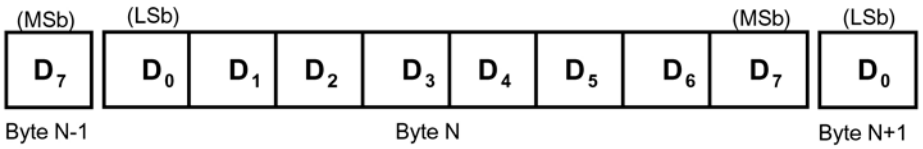
Paquete start-of-frame

Los paquetes de token tienen al final un vector de 5 bits llamado crc5 que se usa para la comprobación de errores en la dirección y el endpoint transmitido, o el número de frame en caso de ser un sof, el pid tiene su propio chequeo.



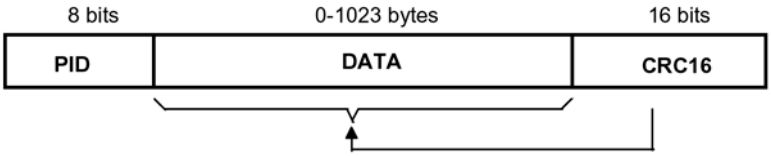
Comprobación crc

El siguiente tipo de paquete que se puede recibir son los paquetes de datos, tienen un rango de entre 0 y 1023 bytes y cada uno se envía empezando por el bit de menor peso.



Orden de los bytes

Los paquetes DATAX quedan compuestos por el pid, un campo de datos de 0 o más bytes y el crc para chequear errores, que en este caso es de 16 bits.



Paquete de datos

El último tipo de paquete son los paquetes de handshake, que consisten únicamente en un PID.

Conocida la estructura de los datos, seguimos con las fases del modo de recepción, después del pid entramos en el estado “dades”.

En el estado “dades” se recogerán los bits que siguen al PID. En función de éste se activará la unidad de corrección de errores en modo crc5 o crc16, cada tipo de paquete puede tener uno de estos dos sistemas de corrección de errores.

Aquí se presenta un problema importante, ya que no hay tiempo suficiente entre la recepción de algunos paquetes para efectuar los cálculos del crc después de recibirlo, por lo que la corrección de errores debe realizarse a medida que llegan los bits.

A este se le añade otro problema, ya que el campo de comprobación del paquete llega a continuación de los datos sin ningún tipo de diferenciación, por lo que no se le identifica hasta que finalmente termina el paquete con la recepción de la señal de final de paquete EOP (end of packet).

En el sistema que se usaba en un principio, se retrasaba el inicio del cálculo de errores algún bit más de la longitud del vector de comprobación, de modo que al final del paquete diese tiempo a parar el proceso de corrección antes de que terminase con los datos y continuase por error con el vector de crc. Para solucionar el primer problema se almacenaban en un buffer los últimos 16 bits recibidos y así realizar una comprobación rápida entre el resultado de la fórmula del crc16 y el vector final recibido. Una vez se comprende el modelo matemático del sistema de corrección, se descubre un método más sencillo que se explicará en el proceso de comprobación de errores.

No se ha encontrado en la especificación del USB el tiempo que debe pasar entre dos paquetes, solo el tiempo que debe pasar entre que se recibe y se envía un paquete, 8 bits. Fue gracias a la experimentación que se dedujo el corto espacio de tiempo que pasaba entre la recepción de un paquete de SETUP y el siguiente de DATA0 enviados seguidos por el host, y mas adelante la casualidad hizo que en una

parada manual del osciloscopio se captasen estos dos paquetes y se pudiese comprobar que pasaban solo 5 señales de reloj (clocks), el tiempo máximo para solucionar el crc y entrar en espera del siguiente paquete.

Otras tareas que se realizan durante la recogida de datos (fase “dades”) son el almacenamiento en la memoria RAM de los datos recogidos para su tratamiento posterior. La descomposición de los primeros 64 bits del paquete en variables, es una tarea solo útil si se trata del paquete de datos que sigue al de setup, pues estos paquetes están predefinidos y sus bytes tienen un significado concreto. Se calcula numéricamente el valor de uno de estos bytes que representa la longitud del paquete de datos que pide o quiere enviar el host a continuación, del mismo modo que la anterior operación solo es útil si se trata del paquete de setup. Constantemente se está comprobando que se cumpla con el “bit stuffing”.

En resumen, el proceso descompone el paquete en sus partes para que la unidad de control tenga la información que necesita lista en el momento que ejecuta el cambio de estado.

Los datos se van grabando en la RAM lógica en bloques de 64 bits durante la fase de recogida de datos. En el estado de fin se graban los bits del ultimo paquete que todavía no habían formado un bloque completo de 64, esperando el tiempo oportuno si este último paquete es muy pequeño y la memoria aun está grabando el bloque anterior.

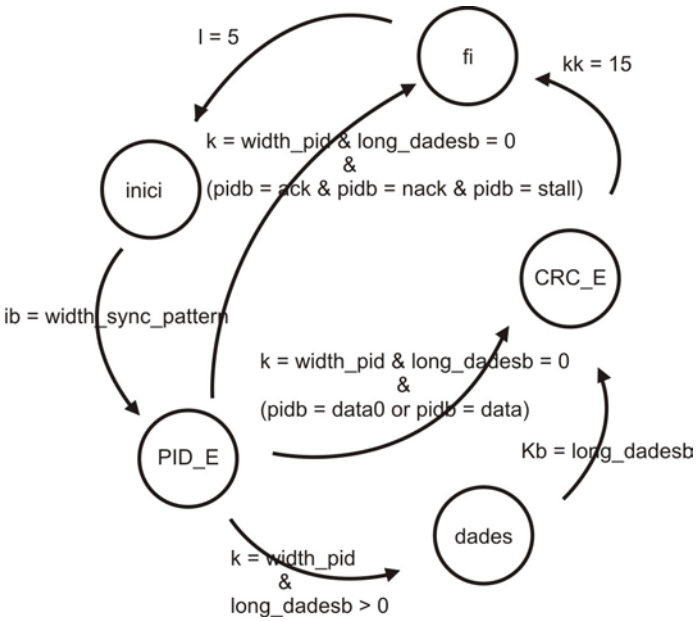
En el momento que se recibe el primer bit del SE0 se pasa a la siguiente fase, “SE0”, en la que simplemente se espera recibir el segundo bit de SE0 correspondiente al fin de paquete. En este momento se aprovecha para hacer un recálculo de la longitud del paquete y guardar información en algunas variables auxiliares.

En la fase de fin de la recepción y tras acabar el proceso que controla el crc, se comprueba que el paquete sea correcto y también su longitud en los paquetes de longitud fija token o handshake, y se envía una señal de recepción de paquete correcto (sad) a la unidad de control, que ejecutará sus máquinas de estado para moverse al que corresponda, ordenando al bloque que ejerce de buffer si debe permanecer recibiendo o enviándole un paquete para transmitir.

En los paquetes de datos puede ser necesario, antes de finalizar, pasar a otro estado “graba”, donde se grabarían los bits del último paquete antes de dar la señal de paquete recibido. Este retraso se debe a que la controladora de la memoria RAM tarda un tiempo en grabar un paquete, 4 o 5 clocks. Si el número de bits de la última porción fuese muy pequeña y se intentasen grabar justo al terminar el paquete la memoria, podría no haber acabado aun con la tanda anterior.

Al dar la señal de paquete correcto se pasa al estado “postfi” en el que se pone a “0” la señal de paquete recibido y automáticamente se vuelve a la fase inicial.

Cuando la máquina de estados de la unidad de control decide que toca enviar un paquete, pondrá el pid, los datos y su longitud en su buffer de salida y activara a “1” la señal ebs que colocará este módulo en el estado inicial de envío “inici” con los datos colocados en su buffer de entrada.



Máquina de estados del envío de datos

La máquina de estados de la unidad de control define en un solo clock su nuevo estado y como proceder, a partir de los datos proporcionados por el paquete recibido. Es por lo que en este estado inicial esperaremos 6 bits a enviar el paquete, y cumplir con seguridad con el byte de espacio que hay que dejar entre la recepción y el envío de datos. Tras la espera se inicia el proceso que calcula el crc del paquete que tenemos en el buffer de entrada y a su vez se empieza a enviar el paquete de

sincronización. En el momento que se envía el último bit de sincronización, se pasa al estado de envío del pid “pid_e”.

En el estado pid_e se envían los ocho bits que forman el pid y durante el envío del último bit se decide el siguiente estado en función del pid y la longitud de paquete. Si se trata de un paquete de datos de longitud 0, se apaga el proceso de crc, que no habrá efectuado cambios y se pasa al envío del crc, estado “crc_e”. Si se trata de un paquete de datos de longitud superior a cero, se pasa primero al envío de datos, estado “dades” y si se trata de un paquete de handshake, se procede directamente al envío de la señal de fin de paquete. En todo momento durante el envío se están sumando los unos que hay seguidos para realizar el bit stuffing si llegan a 6.

Al finalizar el envío de datos se apaga el crc y se salta al estado de “crc_e”, en el que se enviarán los datos del vector de comprobación invertido. El hecho de que se invierta se debe a su significado matemático, que se estudiará más adelante.

Tras el crc se envía la señal de fin de paquete que, aunque teóricamente con 2 ciclos de SE0 sería suficiente, experimentalmente se comprueba que son necesarios 4 ciclos de reloj para que el host lo detecte bien, seguidos de un uno lógico.

El primer proceso que se explicó era el encargado de detectar el fin del envío y colocará en ese momento la máquina de estados en el modo de recepción de paquete, cuyo estado inicial colocará a su vez el envío en su posición inicial.

El chip tiene dos modos de envío de datos singled-ended (mode 0) o diferencial (mode 1), se envía en modo 0 regido por las siguientes tablas de verdad, de ahí los valores que se colocan sobre vp y vm en la implementación del programa.

Table 1a. MAX3340E Truth Table, Transmit (MODE = 0)

OE/ENUMERATE = 0 (TRANSMIT), RENB = 0					
INPUT		OUTPUT			RESULT
VP	VM	D+	D-	RCV	
0	0	0	1	0	Logic 0
0	1	0	0	X	SEO
1	0	1	0	1	Logic 1
1	1	0	0	X	SEO

Table 1b. MAX3340E Truth Table, Transmit (MODE = 1)

OE/ENUMERATE = 0 (TRANSMIT), RENB = 0					
INPUT		OUTPUT			RESULT
VP	VM	D+	D-	RCV	
0	0	0	0	X	SEO
0	1	0	1	0	Logic 0
1	0	1	0	1	Logic 1
1	1	1	1	X	Undefined

Para la recepción hay un único modo.

Table 1c. MAX3340E Truth Table, Receive

OE/ENUMERATE = 1 (RECEIVE), RENB = 1					
INPUT		OUTPUT			RESULT
D+	D-	VP	VM	RCV	
0	0	0	0	X	SEO
0	1	0	1	0	Logic 0
1	0	1	0	1	Logic 1
1	1	1	1	X	Undefined

El último proceso de este módulo se encarga del control de errores, calculando el crc de los paquetes que se envían y reciben, para comparar el resultado con la cadena de bits que se recibe al final del paquete o añadirlo al paquete enviado.

El crc (cyclic redundancy check) se basa principalmente en el concepto de la división de enteros, cuando un dividendo se divide por un divisor obtenemos un cociente y un resto concretos. A partir de un dividendo y con un divisor conocido por ambas partes se puede recalcular la división en el destino para comprobar que los restos son iguales o dividir la diferencia del dividendo menos el resto y comprobar que dé cero.

El concepto de división entera se puede aplicar también a la división de polinomios que conservan las mismas propiedades: considerando que los bits de una cadena representan coeficientes de un polinomio se obtiene un dividendo que dividido por otro polinomio divisor al que llamaremos polinomio generador, éste conocido por ambas partes de la transmisión, obtendremos un cociente y un resto únicos.

Facilita la división el hecho que al estar trabajando con ceros y unos, en base 2, las sumas y restas de la división se convierten en simples XOR.

El crc es capaz de detectar errores simples, dobles y algunos múltiples en cantidades pequeñas de bits. En el protocolo USB se considera suficiente un campo de chequeo de 5 bits para los paquetes de token que ocupan solo once bits y 16 bits para los paquetes de datos que ocupan hasta 1023 bytes.

Como el resto es siempre de grado inferior al divisor, se utiliza el polinomio generador X5 + X2 + X0 para que la longitud del resto no sea superior a 5 bits y el polinomio X16 + X15 + X2 + X0 para que la cadena resultante no sea superior a 16 bits.

La forma más intuitiva para generar el crc es dividir la cadena de datos considerando el primer bit que llega el más significativo (MSB) del polinomio y así podemos empezar la división al comienzo de la recepción, de forma que si llega un “1” se resta el divisor del dividendo y se desplaza el resto a la izquierda, como en una división de polinomios, y si llega un “0” simplemente se desplaza el resto a la izquierda, y así sucesivamente hasta el bit de menor significado (LSB). En ese momento el resto será un polinomio de grado igual o inferior al polinomio generador. El crc que se calcula según el protocolo USB tiene algunas propiedades extra, matemáticamente poco importantes, pero que mejoran el sistema.

La implementación comienza con el resto cargado de unos en vez de ceros, que sería lo lógico matemáticamente, y esto es equivalente a sumar al polinomio que nos llega un polinomio de grado superior, que sería el que nos habría dejado esos unos en el residuo, y es un polinomio calculable y por lo tanto conocido, pero su valor no es lo importante, pues al comprobar el crc se emplea el mismo sistema, ya que de otra forma los ceros que hubiese al principio del polinomio enviado no quedarían protegidos por la formula del crc (por una división).

La segunda característica en la implementación real aprovecha las propiedades conmutativas y asociativas del XOR. La resta del dividendo y el divisor no se realiza de una sola vez sino bit a bit, acumulando sobre el resto lo correspondiente a los siguientes bits y desplazándolo a la izquierda, es decir, no se resta sobre el dividendo porque aun no ha llegado sino que se acumula sobre el resto. De esta forma no se tiene que esperar a recibir un grupo suficiente de bits para realizar la resta sino que

cada vez que llega uno se opera sobre él y se acumula el valor correspondiente para los siguientes bits en el resto.

La última característica especial consiste en invertir el resto antes de colocarlo al final de la cadena de bits enviada, que se supone el dividendo. Matemáticamente equivale a sumar una constante conocida al resto. La razón de esta operación es evitar que el resultado final sea cero, pues ceros añadidos al final del paquete no se detectarían como errores, y es que si se añade al polinomio su resto tal cual, los ceros añadidos erróneamente al final darían como residuo igualmente cero al continuar la división.

El paquete estará formado por el polinomio con su resto invertido a continuación, sin ninguna diferenciación, de forma que en su chequeo hay que dividirlo todo para obtener el valor de la constante que equivale a la inversión del resto. Este valor residual constante es 01100 (X4 + X3) para los paquetes de token y 1000000000001101 (X15+X3+X2+x0) para los de datos.

La generación del crc se calcula antes de generar el bit sttufing, del mismo modo tendrá que deshacerse este antes de la comprobación.

En la implementación se divide el cálculo del crc en 5 estados. En el primero, “datos”, se inicializan las variables y se rellena el resto de unos para pasar al siguiente estado, “inici”. En el estado “inici” se comprueba ya el primer bit que llega, se calcula el XOR de este bit con el bit mas alto del residuo, y en función de este resultado se actuará de una manera u otra, no hay que considerar simplemente el bit que llega sino que hay que tener en cuenta lo acumulado en el resto. En el siguiente estado “comprobación”, si el resultado fue un cero, se desplaza simplemente el resto a la izquierda rellenándose el bit de menor peso con un cero, y si el resultado del XOR fue un uno, se realiza una XOR completa entre el resto y el polinomio generador después de desplazar el resto.

También se calcula el resultado XOR del siguiente bit recibido con el bit de mayor peso del residuo que nos queda para saber como se operará en el siguiente clock avanzándose un paso.

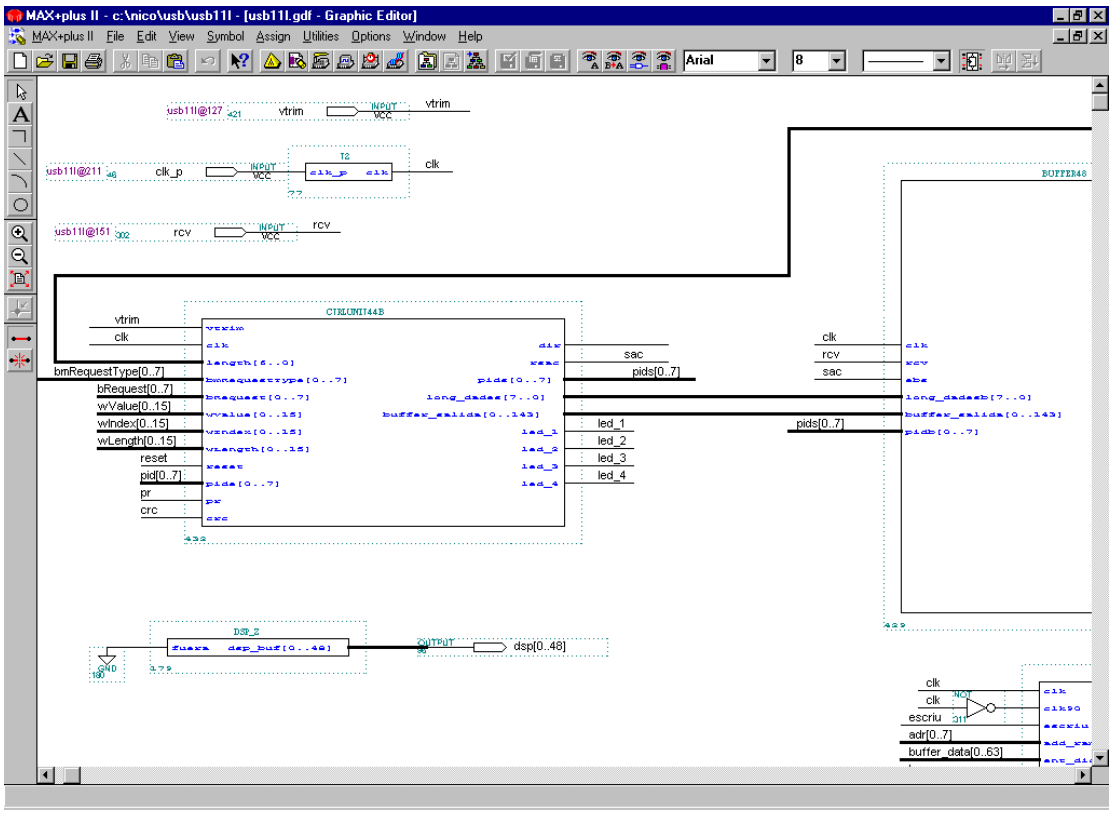
En el penúltimo estado, “es_correcto”, se comprueba el resultado que nos da con el residuo conocido. En el último estado, “fin”, se espera a que otro proceso recoja el resultado antes de parar la máquina.

Este proceso de corrección de errores es el último que se realiza en el módulo “buffer”, y el siguiente paso es estudiar la unidad de control que dará una idea de cómo funciona el proceso lógico del protocolo USB y las máquinas de estados que lo rigen.

4.3.2 El módulo de capa lógica

Se puede dividir un dispositivo USB en tres capas. A la capa más baja o capa física la representa el módulo buffer50 que se ha estudiado en el capítulo anterior, realiza tareas de buffer enviando y recibiendo paquetes del bus y controlando errores de transmisión. Este capítulo explica el módulo que implementa la capa media, que controla el movimiento de datos entre la capa baja o la interficie de bus y los diferentes endpoints del dispositivo, o puntos finales a los que debe llegar un paquete. La capa alta representa la función final del dispositivo, es decir, la utilidad que le da a esos datos y variará según el uso que se le quiera dar al dispositivo en un futuro.

El módulo de capa lógica “ctrl_unit” se encarga principalmente del control de las diferentes máquinas de estados que rigen el comportamiento del protocolo USB, que está dividido en dos grandes procesos, el primero gestiona el estado general en el que se encuentra el dispositivo, y el segundo controla el estado dentro de una transmisión.



La máquina de estados que implementa el primer proceso consta de cinco estados: “conectado” (Attached), “powered” (Encendido), “por defecto” (Default), “direccionado” (Address) y “configurado” (Configured), dejando el caso especial suspendido (Suspended) para el buffer.

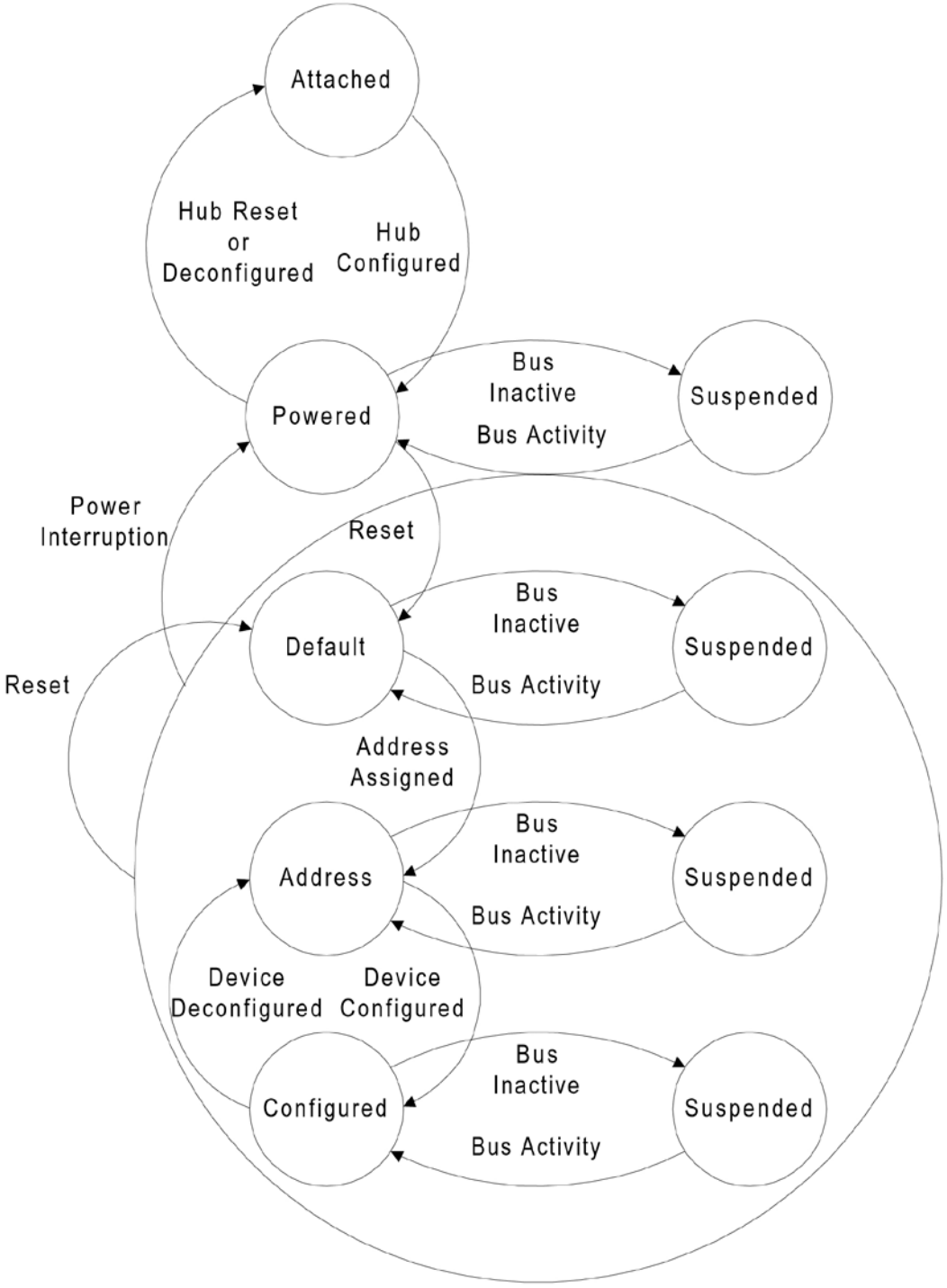


Diagrama de estados de un dispositivo.

El primer estado “Attached” indica que el dispositivo está conectado físicamente al bus USB y será el estado inicial, el comportamiento del dispositivo previo a este momento no está especificado en el protocolo.

Este estado es detectado por el chip, mediante unos amplificadores operacionales, cuando cambia el voltaje del bus al enchufarse el cable en el otro

extremo de la conexión donde las líneas de datos se conectan a masa a través de unas resistencias.

Se considera en el segundo estado, “powered”, cuando el dispositivo empieza a recibir Vbus a través del cable USB y se estabiliza, el host puede detectar el dispositivo antes de ser encendido.

En la implementación se detecta este estado a partir de una señal de salida del chip regulada a 3,5 V, señal “vtrim”, que se activa cuando el chip comienza a recibir voltaje, y también puede ser aprovechada para autoalimentar el dispositivo si se conecta el jumper de la placa que aprovecharía el Vbus, aunque en principio se utiliza el mismo alimentador que el de la tarjeta gráfica.

Una vez el dispositivo está “powered”, no puede responder a ninguna transacción en el bus hasta que reciba una señal de reset, momento en el que pasará al estado de “default”. En ese estado ya podrá responder a las request (peticiones) del host, que comenzará pidiendo el descriptor de configuración y devolverle la información requerida.

Durante el estado “default”, el dispositivo debe escuchar las peticiones a la dirección por defecto 00H y será a través del proceso de enumeración cuando el dispositivo adquiera una dirección única para él, momento en el que pasará a estado “adressed”.

En la implementación, el primer proceso es una estructura condicional muy simple que distingue las fases y tiene variables para pasar entre los estados que provienen de otros procesos, módulos o señales directas del hardware. El reset proviene del modulo de capa física en el que se detecta esta señal única, las dos líneas estarán en nivel bajo durante un período determinado. El vtrim es una de las entradas de la FPGA que proviene del chip, el “dir_valida” y el “configurado” provienen de la otra máquina de estados de este mismo módulo.

En el estado “adressed” se pedirán al dispositivo los descriptores para conocer su comportamiento, estas peticiones son estándar y los descriptores que se envían sirven para definir el dispositivo.

Las peticiones se realizan mediante lo que se llama una transmisión de control a la dirección actual del dispositivo. En una de las fases de la transmisión, el host envía un paquete de datos con un formato predefinido de ocho bytes de longitud en la que se encuentra la petición concreta. El formato de esta petición está preestablecido, el primer byte del paquete es el bmRequestType, que indica la dirección de la transmisión, envío o recepción, el tipo de paquete, las peticiones estándar son de tipo cero, y el tipo de recipiente final. Una petición puede dirigirse al dispositivo general, a una interficie de éste o al endpoint de una interficie. El segundo byte es el bRequest que indica la petición concreta al dispositivo. Los bytes tercero y cuarto son el wvalue, un parámetro cuyo significado dependerá de cada petición. Los bytes quinto y sexto son el wIndex, otro parámetro que depende de la petición, pero que suele referirse a un índice u offset. Lo dos últimos bytes son el wLength, que representan la longitud de datos a transmitir durante la siguiente fase de la transmisión en la que se procede con la petición.

Offset	Field	Size	Value	Description
0	*bmRequestType*	1	Bitmap	Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6...5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4...0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	*bRequest*	1	Value	Specific request (refer to Table 9-3)
2	*wValue*	2	Value	Word-sized field that varies according to request
4	*wIndex*	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	*wLength*	2	Count	Number of bytes to transfer if there is a Data stage

4.3.2.1 Peticiones Estándar

Las peticiones estándar que debe conocer y responder cualquier dispositivo USB en sus estados default, direccionado o configurado se definen a continuación junto al comportamiento básico que debe adoptar.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B 00000001B 00000010B	CLEAR_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
10000000B	GET_CONFIGURATION	Zero	Zero	One	Configuration Value
10000000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
10000001B	GET_INTERFACE	Zero	Interface	One	Alternate Interface
10000000B 10000001B 10000010B	GET_STATUS	Zero	Zero Interface Endpoint	Two	Device, Interface, or Endpoint Status
00000000B	SET_ADDRESS	Device Address	Zero	Zero	None
00000000B	SET_CONFIGURATION	Configuration Value	Zero	Zero	None
00000000B	SET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
00000000B 00000001B 00000010B	SET_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
00000001B	SET_INTERFACE	Alternate Setting	Interface	Zero	None
10000010B	SYNCH_FRAME	Zero	Endpoint	Two	Frame Number

Peticiones estándar de dispositivo

bRequest	Value
GET_STATUS	0
CLEAR_FEATURE	1
Reserved for future use	2
SET_FEATURE	3
Reserved for future use	4
SET_ADDRESS	5
GET_DESCRIPTOR	6
SET_DESCRIPTOR	7
GET_CONFIGURATION	8
SET_CONFIGURATION	9
GET_INTERFACE	10
SET_INTERFACE	11
SYNCH_FRAME	12

Códigos estándar de petición

Para referenciar cada recipiente final hay una codificación en hexadecimal.

Descriptor Types	Value
DEVICE	1
CONFIGURATION	2
STRING	3
INTERFACE	4
ENDPOINT	5

Tipos de descriptor

Los diferentes tipos de petición definirán el dispositivo, informarán de sus características y tratarán los registros de estado para habilitar las opciones de éste.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B 00000001B 00000010B	CLEAR_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None

La petición Clear Feature se utiliza para deshabilitar características, cada una de ellas tiene un tipo de recipiente concreto, dispositivo, interficie o endpoint. Si no coinciden característica y recipiente o si el dispositivo o el endpoint no están especificados en su configuración, se responde con un error a la petición. Las dos peticiones estándar para deshabilitar la capacidad de remote wake up o finalizar el estado halt, “colgado”, de un endpoint se definen en la tabla siguiente.

Feature Selector	Recipient	Value
DEVICE_REMOTE_WAKEUP	Device	1
ENDPOINT_HALT	Endpoint	0

Selectores de característica estándar

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B	GET_CONFIGURATION	Zero	Zero	One	Configuration Value

En la petición Get Configuration se devuelve el valor o ID de la configuración actual del dispositivo si este está configurado. Si solamente está direccionado, devolverá el valor cero.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID (refer to Section 9.6.5)	Descriptor Length	Descriptor

En la petición Get Descriptor se devuelve el descriptor especificado en el wvalue, si existe. El byte alto de wvalue representa el tipo de descriptor y el bajo su índice, wIndex especifica el lenguaje en los descriptores de cadena (string

descriptors), cero para el resto, y wLength la longitud de datos que se pretende recibir. Si ésta es mayor que la del descriptor, el dispositivo devuelve solo la longitud de éste, acabando con un short packet. Se define short packet como un paquete más corto que la longitud máxima de datos de ese endpoint y se reconoce como el último paquete de la fase de datos. Cuando el descriptor es mayor, solo se envían la cantidad de bytes pedidos.

Hay tres tipos de petición de descriptores, de dispositivo (device), de configuración (configuration) y de cadena (string), pero cuando se pide el de configuración se devuelven también los de interficie (interface) y a continuación de cada uno de ellos siguen sus descriptores de puntos finales (endpoints).

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000001B	GET_INTERFACE	Zero	Interface	One	Alternate Setting

En la petición Get Interface, se devuelve el valor o ID de la interficie pedida dentro de las posibles que se han definido durante la configuración.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B 10000001B 10000010B	GET_STATUS	Zero	Zero Interface Endpoint	Two	Device, Interface, or Endpoint Status

En la petición Get Status se devuelve el registro de estado del recipiente que se especifica en el bmRequestType, y si no existe ninguno que coincida con el wIndex pedido, se devuelve un error.

Cuando se pide el estado de un dispositivo se devuelven dos bytes de información durante la fase de datos, el estándar solo define el significado de los dos bits de menor peso del primer byte. El bit 0 representa la característica self powered (autoalimentado), si es cero significa autoalimentado y si es uno, que se alimenta del bus USB, que no puede cambiarse con el set_feature() o el clear_feature(). El bit 1 representa el remote wakeup, que empieza con valor 0 o deshabilitado, y se puede

cambiar con el set_feature() y el clear_feature(). Cuando un dispositivo se resetea vuelve a valor 0.

El estándar también define el bit 0 de una petición Get_status a un endpoint para transacciones de interrupción o en masa, y representa el estado Halt de la línea. Cuando ha habido un error, se coloca en valor uno y los paquetes que se reciban se responderán con un paquete de handshake STALL. Opcionalmente, puede implementarse en las transacciones de control de la Default Control Pipe, como se llama al canal por defecto de transmisión.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	SET_ADDRESS	Device Address	Zero	Zero	None

En la petición Set Address, se direcciona el dispositivo que adopta como dirección el valor de wvalue cuando termine esta transacción de control, es decir, tras finalizar la fase de status. Más adelante se explicarán los diferentes tipos de transacciones.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	SET_CONFIGURATION	Configuration Value	Zero	Zero	None

En la petición Set Configuration, se coloca al dispositivo con la configuración indicada en el wValue, que representa el ID de la configuración dentro de las definidas en el descriptor, si wValue es 0 el dispositivo vuelve al estado direccionado.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B	SET_DESCRIPTOR	Descriptor Type and Descriptor Index	Language ID (refer to Section 9.6.5) or zero	Descriptor Length	Descriptor

En la petición Set Descriptor se pueden modificar los descriptores o añadir nuevos, wValue especifica en su byte alto el tipo del descriptor y en el bajo el índice

dentro del descriptor, wIndex es cero excepto en los descriptores de cadena (string),donde especifica el ID del lenguaje y wLength indica la longitud en bytes que tendrá el descriptor.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B 00000001B 00000010B	SET_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None

Con la petición Set Feature se habilita una característica, el selector de ésta debe coincidir con el recipiente seleccionado en wIndex. Si no coinciden o el recipiente no existe, se devolverá error o un STALL en la fase de status de la transacción.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000001B	SET_INTERFACE	Alternative Setting	Interface	Zero	None

En la petición Set Interface se permite al host seleccionar una de las diferentes interficies alternativas definidas en el descriptor, si solo soporta una por defecto o la pedida no existe, se devolverá error.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000010B	SYNCH_FRAME	Zero	Endpoint	Two	Frame Number

En transacciones síncronas, se utiliza la petición Synch Frame para decidir en qué frame empezar el patrón a repetir.

4.3.2.2 Tipos de Transacción

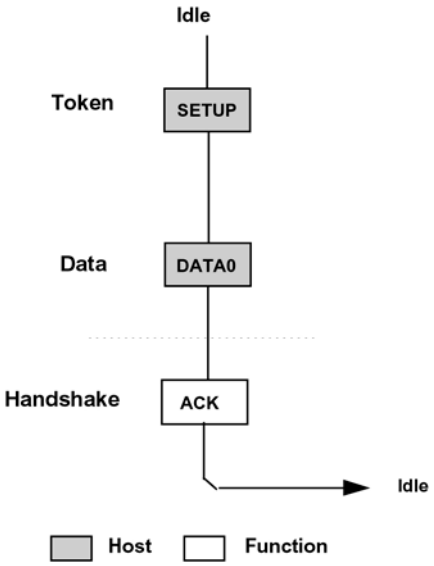
Conocidas las peticiones deben explicarse los diferentes tipos de transacción antes de continuar con la explicación del segundo proceso de la unidad de control, donde se implementa la maquina de estados que rige las transacciones.

En los procesos iniciales, tras la conexión de un dispositivo USB se utiliza la transmisión de control sobre un canal definido por defecto, la Default Control Pipe. Aparte hay tres tipos más para funciones adicionales que pueda tener, el primero es la transmisión en masa (Bulk Transactions) para dispositivos que transmitan grandes cantidades de datos sin necesidad de un control exhaustivo, como las impresoras; el segundo, es la transmisión síncrona (Isochronous Transactions), cuando se deba controlar el momento exacto en el que transcurre una transacción; el tercero, es la transmisión por interrupciones (Interrupt transactions), para dispositivos que no necesiten un estado constante de comunicación o cuando esta se realice con períodos muy largos de espera, los dispositivos de interfaz humana como ratones o teclados (HID – human interface device) son un buen ejemplo.

Aunque la estructura queda abierta para añadir el resto de tipos de transacción y al final del proyecto se explicará detalladamente como añadir nuevas funciones al dispositivo, en esta implementación solo se encuentra la transmisión de control. El motivo principal es que para el objetivo de este proyecto no son necesarias las otras tres y al no ser utilizadas tampoco podría comprobarse su correcto funcionamiento, de cualquier modo, con las máquinas de estados del protocolo su implementación sería sencilla una vez conocido el lugar adecuado donde situarlas, en el código quedan abiertos los huecos para su futura realización.

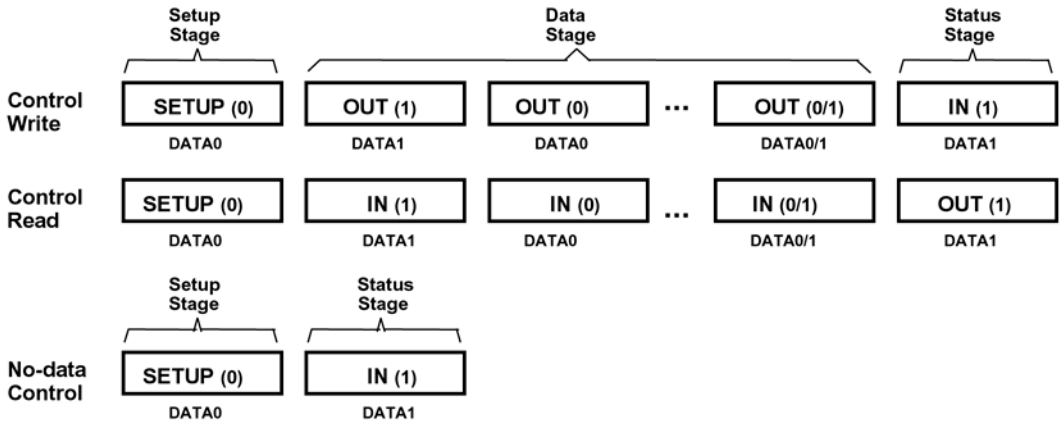
Una transmisión de control se divide en tres partes: la etapa de setup, la etapa de datos y la etapa de status. Durante la etapa inicial (setup stage), el host envía una petición al endpoint o función final del dispositivo. Esta fase comienza con un paquete con el pid (packet identifier) “setup” y de la misma forma que con los paquetes de envío de datos, token out, el dispositivo esperará a continuación un paquete de datos

con el formato de las peticiones explicadas anteriormente y cuyo significado deberá conocer e interpretar. Le indicarán el principal motivo de la transacción y la manera de proceder, y según la información recibida en esta fase, el dispositivo sabrá si en la siguiente etapa de datos debe enviar o recibir información, o si directamente debe esperar una etapa de estatus. Al final de la etapa de setup el dispositivo enviará un paquete de confirmación o handshake para indicar que ha recibido bien los datos de esta fase.



fase de setup

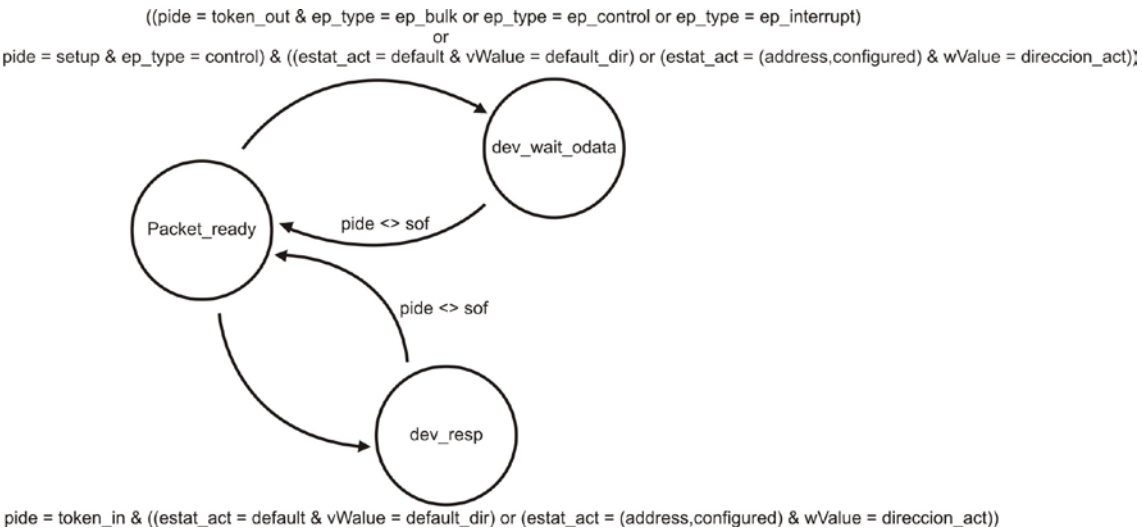
En la segunda etapa (data stage) se envían o reciben los datos de la petición, el host controla la transacción, por eso la fase comienza con un paquete de token in o token out, enviado por el host en función del sentido de la transacción y a continuación se transferirá el paquete de datos y el punto encargado de recibirlos devolverá al final un paquete de handshake. Esta operación se repetirá hasta transmitir la totalidad de datos siendo el máximo de bytes en cada paquete el especificado en el descriptor, en su defecto ocho. Para reconocer errores en la transmisión se alternan los paquetes de datos de cada repetición, siendo un paquete data0 el encargado de transportar los datos en la fase de setup, un data1 el primero de la fase de datos, un data0 el segundo y así sucesivamente.



Transmisión de control

En la última etapa (status stage), se invierte el sentido de la transmisión, por lo que si antes se recibían token out, en esta será un token in y se responde a este paquete con un paquete de datos de longitud cero y el receptor de los datos devuelve el último handshake. Si ha habido un error en la transmisión y se ha recibido un paquete stall durante el setup o los datos, debe responderse en esta fase final también con un paquete stall.

La máquina de estados implementada en el segundo proceso de la unidad de control debe controlar en qué fase de la transacción está y dentro de la fase qué paquete toca emitir o recibir. Para ello es muy útil seguir las máquinas de estados que nos especifica el protocolo USB en su capítulo 8, dedicado a esta capa, aunque aquí solo se explicará la parte implementada dedicada a la transmisión de control.



Máquina de estados de la transmisión de control

Como el host será el encargado de iniciar cada transmisión en cada una de sus fases, se empieza siempre en el estado “paquet_ready” esperando el primer paquete. Cuando llega un paquete a la dirección actual del dispositivo y se trata de un setup o token out, se espera al siguiente paquete de datos en el estado “dev_wait_odata”. Cuando llega se estudia que tipo de paquete es, se ejecuta su función, se prepara el paquete de salida si así lo requiere la petición y se vuelve al estado inicial tras enviar el paquete de handshake correspondiente ack, nack o stall. Cuando se recibe un token in quiere decir que el host espera datos, y previamente preparados en función de la petición inicial, se envían y se espera el paquete de handshake en el estado “dev_resp”.

Para el control de la etapa en la que se encuentra la transmisión se utiliza la variable “fasesc” que comienza con valor “setup_ctrl”. Tras recibir el primer paquete de datos ya se conoce la dirección que tendrá la etapa de datos, si es que esta existe y se pasa a estado “dataout_ctrl” o “datain_ctrl”. En cada uno de estos dos estados solo se esperan recibir paquetes en la dirección correspondiente, y en cuanto ésta cambie nos encontramos en la fase de estatus “status_ctrl” en la que se envía o recibe el paquete de datos de longitud cero. En una recepción, se vuelve al estado “setup_ctrl” tras enviar el paquete de longitud cero de la status stage y en un envío, tras el paquete de confirmación.

Es interesante notar que la máquina se ejecuta en un solo clock, o mejor dicho durante el rising edge de la variable “pr” que indica paquete recibido y que proviene del buffer, con lo que vuelve a verse la utilidad de descomponer el paquete en la recepción, pues se conoce entre otras cosas wvalue(0..10) que representa la dirección en paquetes de token. La variable “dentro” nos ayuda a inicializar la señal de envío “rsac” a cero antes de cada paquete y permite que el buffer espere un “rsac” en su flanco de subida.

La variable “pide” es el pid del paquete actual y “token_pid” el del anterior, lo que permitirá tratar correctamente los paquetes de datos recibidos en la segunda parte de cada transmisión. Ttoggle” es una variable que controla la alternancia entre paquetes data0 y data1 que se produce durante todo el proceso de transmisión de control.

La variable “ep_type” nos indica el tipo de transmisión en la que se encuentra el dispositivo, en principio solo control. Data_avail” indica que el dispositivo no es capaz de recibir datos por el momento y “ep_trouble” cuándo ha habido un error en la transmisión.

En el estado “dev_wait_odata” se encuentra toda la casuística necesaria para, en función de la petición recibida, tratar los datos actualizando los registros del dispositivo o enviando los descriptores de información del sistema para que el host conozca sus características. El significado de estos descriptores está explicado a continuación junto a los valores adoptados en el programa. En este estado se prepara ya la salida de información de la siguiente fase colocándola en el buffer de salida junto a su longitud “lengthw”, se decide que paquete responder en el handshake de la fase, “pids”, y se envía poniendo “rsac” a nivel alto. Long_dades” es la longitud del paquete que sale, en este momento 0 por tratarse de un paquete de handshake.

Para ello se estudian todas la variables involucradas, el tipo de petición que viene dado por “brequest” y “bmrequest_type”, y los parámetros “wvalue”, “windex” y “wlength”, se utiliza también “length”, el cálculo numérico de wlength, en las comparaciones de enteros para conocer la longitud de datos a enviar.

En el último estado del programa “dev_resp” simplemente se espera la respuesta de confirmación del host o paquete de handshake. En el caso particular de tratarse de la petición de set_adress, se actualiza la variable “dir_valida” a uno que indica que el dispositivo está direccionado y adopta su dirección actual guardada en “dirección_act”. En el proceso de enumeración que se explica al final del capítulo se verá el momento en el que ocurre está petición.

4.3.2.3 Descriptores estándar

Los descriptores son paquetes de datos con una estructura predeterminada que define todas las características del dispositivo. A través de ellos, en algunas de las peticiones estándar el dispositivo informa al host de sus atributos. Se encuentran entre los descriptores tipos definidos en función de la clase de dispositivo o vendedor,

tratados en especificaciones posteriores, así como otros informativos en forma de cadena de caracteres para su posible lectura por un humano, pero aquí se estudian los principales o estándar para la definición del dispositivo, pues ni su función específica ni su vendedor están aun decididos.

El primer descriptor a definir es el device descriptor o descriptor de dispositivo, cada dispositivo tiene uno único y se transmite en las fases iniciales de la comunicación a través de la Default Control Pipe, en un proceso llamado enumeración.

El formato de este descriptor es de dieciocho bytes, y en el programa es la rama del condicional en la que bmRequestType es GETZERO, bRequest es GET_DESCRIPTOR y wValue es igual a UNO15, que indica el tipo de descriptor de la petición, DEVICE. El primer byte de este descriptor es el blength que da la longitud total del descriptor, en este caso dieciocho; el segundo byte es el bDescriptorType que indica su tipo, DEVICE; los dos siguientes, tercer y cuarto byte, son el bcdUSB, cuyo significado es la versión del protocolo USB que cumple el dispositivo, usb 1.1, el primer byte sería el 01 y el segundo el 10, es decir usb 0110. El quinto byte es el bDeviceClass, que indica el código de clase que, como no se ha precisado, permanece a cero; el sexto y el séptimo byte son el bDeviceSubClass y el bDeviceProtocol, que indican el código de subclase y el de protocolo y del mismo modo que el anterior quedan a cero. El octavo byte es el bMaxPacketSize0, que indica el tamaño máximo en bytes de la Default Control Pipe o Endpoint zero, en el programa ocho bytes que coincide con el valor por defecto de la pipe al iniciar la enumeración. Los dos siguientes bytes, el noveno y el décimo son el ID Vendor, asignado por la asociación que lleva el USB. Una empresa puede comprar un ID para clasificar sus dispositivos dentro de unos estándares. Los dos siguientes bytes son el idProduct, un número designado por cada empresa a su producto; los dos siguientes el bcdDevice, designan el dispositivo, y el siguiente el iManufacturer, representa el índice dentro de la tabla de strings o cadenas de caracteres en la que se describe al que ha manufacturado el dispositivo. El siguiente byte es el iProduct, otro índice de la tabla de cadenas que describe el producto y a continuación está el iSerialNumber ídem para el número de serie. En el programa estos últimos bytes permanecen a cero. El último byte del descriptor de dispositivo es el bNumConfigurations, que da el numero total de configuraciones posibles del dispositivo, cada una de ellas son diferentes modos de

funcionamiento de éste y describen características diferentes, en este caso hay una única configuración.

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	DEVICE Descriptor Type
2	*bcdUSB*	2	BCD	USB Specification Release Number in Binary-Coded Decimal (i.e., 2.10 is 210H). This field identifies the release of the USB Specification with which the device and its descriptors are compliant.
4	*bDeviceClass*	1	Class	Class code (assigned by the USB). If this field is reset to zero, each interface within a configuration specifies its own class information and the various interfaces operate independently. If this field is set to a value between 1 and FEH, the device supports different class specifications on different interfaces and the interfaces may not operate independently. This value identifies the class definition used for the aggregate interfaces. (For example, a CD-ROM device with audio and digital data interfaces that require transport control to eject CDs or start them spinning.) If this field is set to FFH, the device class is vendor-specific.
5	*bDeviceSubClass*	1	SubClass	Subclass code (assigned by the USB). These codes are qualified by the value of the *bDeviceClass* field. If the *bDeviceClass* field is reset to zero, this field must also be reset to zero. If the *bDeviceClass* field is not set to FFH, all values are reserved for assignment by the USB.

Offset	Field	Size	Value	Description
6	*bDeviceProtocol*	1	Protocol	Protocol code (assigned by the USB). These codes are qualified by the value of the *bDeviceClass* and the *bDeviceSubClass* fields. If a device supports class-specific protocols on a device basis as opposed to an interface basis, this code identifies the protocols that the device uses as defined by the specification of the device class. If this field is reset to zero, the device does not use class-specific protocols on a device basis. However, it may use class-specific protocols on an interface basis. If this field is set to FFH, the device uses a vendor-specific protocol on a device basis.
7	*bMaxPacketSize0*	1	Number	Maximum packet size for endpoint zero (only 8, 16, 32, or 64 are valid)
8	*idVendor*	2	ID	Vendor ID (assigned by the USB)
10	*idProduct*	2	ID	Product ID (assigned by the manufacturer)
12	*bcdDevice*	2	BCD	Device release number in binary-coded decimal
14	*iManufacturer*	1	Index	Index of string descriptor describing manufacturer
15	*iProduct*	1	Index	Index of string descriptor describing product
16	*iSerialNumber*	1	Index	Index of string descriptor describing the device's serial number
17	*bNumConfigurations*	1	Number	Number of possible configurations

Descriptor de Dispositivo Estándar

El descriptor siguiente es el de configuración. En el programa es la rama en la que bmRequestType es GETZERO, bRequest es GET_DESCRIPTOR y wValue es igual a DOS15, que indica el tipo de descriptor de la petición, CONFIGURATION. El primer byte de este descriptor es el bLength que representa la longitud total en bytes que ocupa, en este caso nueve, el segundo byte es el bDescriptorType, que da el tipo

de descriptor, es decir, CONFIGURATION (que como indica la tabla de tipos es el numero dos), los dos siguientes bytes son el wTotalLength, (la longitud total de todos los descriptores a enviar en esta petición). Cuando el host pide el descriptor de configuración, no se devuelve únicamente éste sino que a continuación se envían los descriptores de interficie y los descriptores de endpoint, y además podría haber varias configuraciones. El orden para el envío es el siguiente: primero se envía la primera configuración, después el descriptor de su primera interficie, a continuación de cada interficie se envían los descriptores de sus endpoints, se continúa con la segunda interficie y sus endpoints, hasta finalizar, para proceder con la siguiente configuración. En este programa solo hay una configuración, con una interficie y con dos endpoints de control. En el wTotalLength se especifica el total de bytes de todos estos descriptores, en este caso treinta y dos. El quinto byte es el bNumInterfaces, en el programa valor uno, el sexto byte es el bConfigurationValue, que es el valor para identificar ésta configuración entre el resto, su nombre. El siguiente byte es el iConfiguration, que representa el índice de la tabla de cadenas de caracteres que describe esta configuración. El octavo byte es el bmAttributes que define atributos estándar que tenga la configuración, solo están definidos los bits 5 y 6 que representan el Remote WakeUp y el self-powered respectivamente, con valor uno están activados. El último byte representa la cantidad máxima de corriente que absorbe el dispositivo del bus USB, básicamente para que el sistema pueda controlar si soportará más dispositivos conectados.

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	CONFIGURATION Descriptor Type
2	*wTotalLength*	2	Number	Total length of data returned for this configuration. Includes the combined length of all descriptors (configuration, interface, endpoint, and class- or vendor-specific) returned for this configuration.
4	*bNumInterfaces*	1	Number	Number of interfaces supported by this configuration
5	*bConfigurationValue*	1	Number	Value to use as an argument to the SetConfiguration() request to select this configuration
6	*iConfiguration*	1	Index	Index of string descriptor describing this configuration

Offset	Field	Size	Value	Description
7	*bmAttributes*	1	Bitmap	Configuration characteristics D7: Reserved (set to one) D6: Self-powered D5: Remote Wakeup D4...0: Reserved (reset to zero) D7 is reserved and must be set to one for historical reasons. A device configuration that uses power from the bus and a local source reports a non-zero value in *MaxPower* to indicate the amount of bus power required and sets D6. The actual power source at runtime may be determined using the GetStatus(DEVICE) request (see Section 9.4.5). If a device configuration supports remote wakeup, D5 is set to one.
8	*MaxPower*	1	mA	Maximum power consumption of the USB device from the bus in this specific configuration when the device is fully operational. Expressed in 2mA units (i.e., 50 = 100mA). Note: a device configuration reports whether the configuration is bus-powered or self-powered. Device status reports whether the device is currently self-powered. If a device is disconnected from its external power source, it updates device status to indicate that it is no longer self-powered. A device may not increase its power draw from the bus, when it loses its external power source, beyond the amount reported by its configuration. If a device can continue to operate when disconnected from its external power source, it continues to do so. If the device cannot continue to operate, it fails operations it can no longer support. The USB System Software may determine the cause of the failure by checking the status and noting the loss of the device's power source.

Descriptor de Configuración Estándar

El descriptor siguiente es el de interficie, que se envía tras el descriptor de configuración en su petición. El primer byte de este descriptor es el bLength que representa la longitud total en bytes del descriptor, el segundo byte es el bDescriptorType, su tipo, en este caso INTERFACE. El tercer byte es el bInterfaceNumber que es el ID asignado a esta interficie para su diferenciación con las otras. El cuarto byte es el bAlternateSetting. Una misma interficie puede tener varios

modos de funcionamiento y estos pueden definirse en configuraciones alternativas de la interficie que modifican solo una parte de su configuración sin que el resto se vea afectado, el bAlternateSetting identifica esta configuración entre el resto. El quinto byte es el bNumEndpoints que indica el numero de funciones finales que tiene la interficie, a parte del endpoint por defecto, de los que en el programa se han definido dos. El sexto byte es el bInterfaceClass, el séptimo bInterfaceSubClass y el octavo bInterfaceProtocol, parámetros definidos por el protocolo USB según su función, que aun no ha sido concretada. El último byte es el iInterface, el índice dentro de la tabla de cadenas de caracteres para describir la interficie.

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	INTERFACE Descriptor Type
2	*bInterfaceNumber*	1	Number	Number of interface. Zero-based value identifying the index in the array of concurrent interfaces supported by this configuration.
3	*bAlternateSetting*	1	Number	Value used to select alternate setting for the interface identified in the prior field
4	*bNumEndpoints*	1	Number	Number of endpoints used by this interface (excluding endpoint zero). If this value is zero, this interface only uses the Default Control Pipe.
5	*bInterfaceClass*	1	Class	Class code (assigned by the USB). A value of zero is reserved for future standardization. If this field is set to FFH, the interface class is vendor-specific. All other values are reserved for assignment by the USB.
6	*bInterfaceSubClass*	1	SubClass	Subclass code (assigned by the USB). These codes are qualified by the value of the *bInterfaceClass* field. If the *bInterfaceClass* field is reset to zero, this field must also be reset to zero. If the *bInterfaceClass* field is not set to FFH, all values are reserved for assignment by the USB.

Offset	Field	Size	Value	Description
7	*bInterfaceProtocol*	1	Protocol	Protocol code (assigned by the USB). These codes are qualified by the value of the *bInterfaceClass* and the *bInterfaceSubClass* fields. If an interface supports class-specific requests, this code identifies the protocols that the device uses as defined by the specification of the device class. If this field is reset to zero, the device does not use a class-specific protocol on this interface. If this field is set to FFH, the device uses a vendor-specific protocol for this interface.
8	*iInterface*	1	Index	Index of string descriptor describing this interface

Descriptor de Interficie Estándar

El siguiente descriptor es el de endpoint, que se envía tras el descriptor de interficie en la petición de descriptor de configuración. El primer byte es el bLength que indica la longitud total de bytes de este descriptor, en este caso siete. El bLength representa su longitud en bytes, el bDescriptorType su tipo, ENDPOINT. El tercer byte es el bEndpointAdress, los bits de 0 a 3 indican su dirección y el séptimo byte si es de entrada o de salida, en el programa se define uno de cada tipo. El cuarto byte es el bmAttributes, el estándar solo utiliza los dos primeros bits para indicar el tipo de transmisión que usa, en el programa se usa la de control o valor 00. Los dos siguientes bytes son el wMaxpaquetsize, que indican el tamaño máximo de bytes que este endpoint puede transmitir en un paquete. El último byte es el bInterval que solo se utiliza en la transmisión por interrupciones, en el programa cero.

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	ENDPOINT Descriptor Type
2	*bEndpointAddress*	1	Endpoint	The address of the endpoint on the USB device described by this descriptor. The address is encoded as follows: Bit 3...0: The endpoint number Bit 6...4: Reserved, reset to zero Bit 7: Direction, ignored for control endpoints 0 = OUT endpoint 1 = IN endpoint
3	*bmAttributes*	1	Bitmap	This field describes the endpoint's attributes when it is configured using the *bConfigurationValue*. Bit 1..0: Transfer Type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt All other bits are reserved.
4	*wMaxPacketSize*	2	Number	Maximum packet size this endpoint is capable of sending or receiving when this configuration is selected. For isochronous endpoints, this value is used to reserve the bus time in the schedule, required for the per-frame data payloads. The pipe may, on an ongoing basis, actually use less bandwidth than that reserved. The device reports, if necessary, the actual bandwidth used via its normal, non-USB defined mechanisms. For interrupt, bulk, and control endpoints, smaller data payloads may be sent, but will terminate the transfer and may or may not require intervention to restart. Refer to Chapter 5 for more information.
6	*bInterval*	1	Number	Interval for polling endpoint for data transfers. Expressed in milliseconds. This field is ignored for bulk and control endpoints. For isochronous endpoints this field must be set to 1. For interrupt endpoints, this field may range from 1 to 255.

Descriptor de Endpoint Estándar

El último tipo de descriptor que se utiliza en el programa son los string descriptors o descriptores de cadena. Cuando se peticiona el string de índice cero se devuelve el descriptor de códigos LANGID que representa los lenguajes soportados por el dispositivo, en las demás peticiones el host pedirá el string en el lenguaje deseado.

Offset	Field	Size	Value	Description
0	*bLength*	1	N+2	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	STRING Descriptor Type
2	*wLANGID[0]*	2	Number	LANGID code zero
...	...	...	...	...
N	*wLANGID[x]*	2	Number	LANGID code *x*

Descriptor de Códigos de Lenguaje Soportados

En el programa se define únicamente el 0x0c0a Spanish (Modern Sort). En la rama del condicional del estado dev_wait_Odata donde bmRequestType es GETZERO, bRequest es GET_DESCRIPTOR y wValue es igual a tres que indica el tipo de descriptor STRING, se devuelve el UNICODE string descriptor correspondiente cuando el índice es diferente a cero. Blength es el tamaño del descriptor, dieciocho, bDescriptorType el tipo, STRING, y el resto son los bytes de la cadena de caracteres en unicode, dos bytes cada carácter. Cuando el wValue es 0, se devuelven los lenguajes soportados.

Offset	Field	Size	Value	Description
0	*bLength*	1	Number	Size of this descriptor in bytes
1	*bDescriptorType*	1	Constant	STRING Descriptor Type
2	*bString*	N	Number	UNICODE encoded string

Descriptor de String

4.3.2.4 Proceso de enumeración

Cuando un dispositivo se conecta a un sistema USB, el host comienza un proceso llamado enumeración en el que se identificará y se estudiarán sus características, para finalmente asignarle una dirección y seleccionar su configuración.

Aunque cada sistema operativo lleva a cabo este proceso de una forma muy determinada, el proceso teórico por pasos es el siguiente:

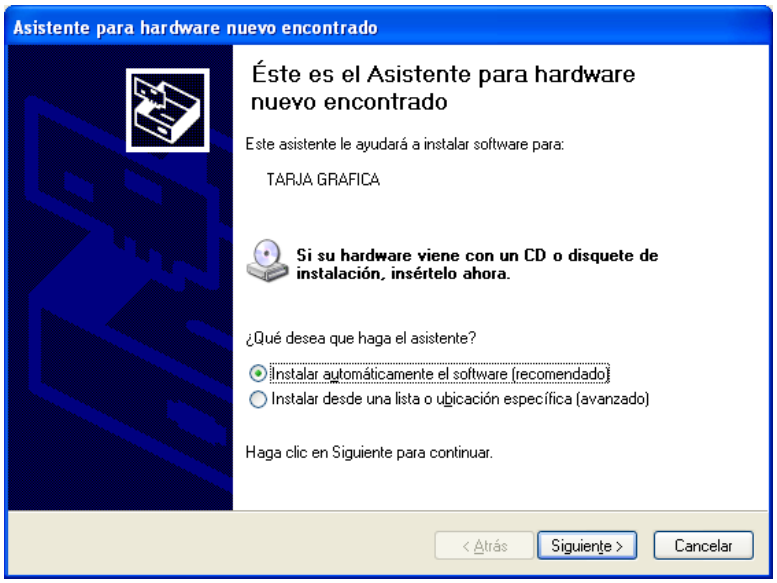
1. El hub al que ha sido conectado el dispositivo informa al host a través de un cambio en su canal de estado. En este momento el dispositivo está en el estado Powered.
2. El host interroga al hub para conocer el motivo de su cambio.
3. Cuando el host se da cuenta de que se ha conectado un dispositivo, espera 100 ms para que la conexión se termine y se estabilice el voltaje en el dispositivo para enviar un reset sobre el puerto.
4. El hub mantiene el reset durante 10 ms. Al finalizar el puerto está ya habilitado y el dispositivo en estado “default”, en el que no puede consumir mas de 100 mA del bus, los registros se resetean y se esperan transmisiones por el canal por defecto, default control pipe.
5. El host asigna una dirección por defecto al dispositivo, que pasará al estado direccionado, “addressed”.
6. El host lee el descriptor de dispositivo para conocer su carga máxima de transferencia en el canal por defecto que continua activo.
7. El host lee la información de todas las configuraciones del dispositivo.
8. Estudiadas las posibilidades y en función de la tarea que vaya a realizar el dispositivo, el host le asigna una de sus configuraciones, momento en el que se encuentra en el estado configurado, en el que todos los endpoints de esa configuración pueden activarse y en el puede pasar a consumir el voltaje especificado en el determinado descriptor de configuración. El dispositivo está listo para su uso.

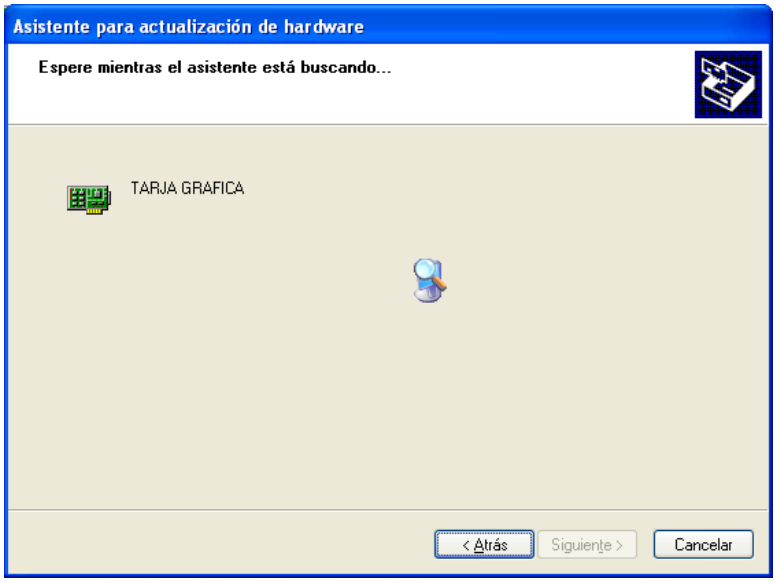
Aunque la teoría es la que se acaba de exponer, cada sistema la lleva a cabo a su manera, y es interesante conocer como lo hace windows, pues en fases de desarrollo puede que el comportamiento no sea el esperado. Por ejemplo, en el momento en el que se espera un paquete de set address, es decir en el punto 5, windows realiza previamente un get device descriptor para leer únicamente sus 8 primeros bytes y

conocer ya la carga máxima que puede transmitir, lo que lleva a confusión hasta que se conoce este hecho. Más detalladamente los pasos que sigue windows son:

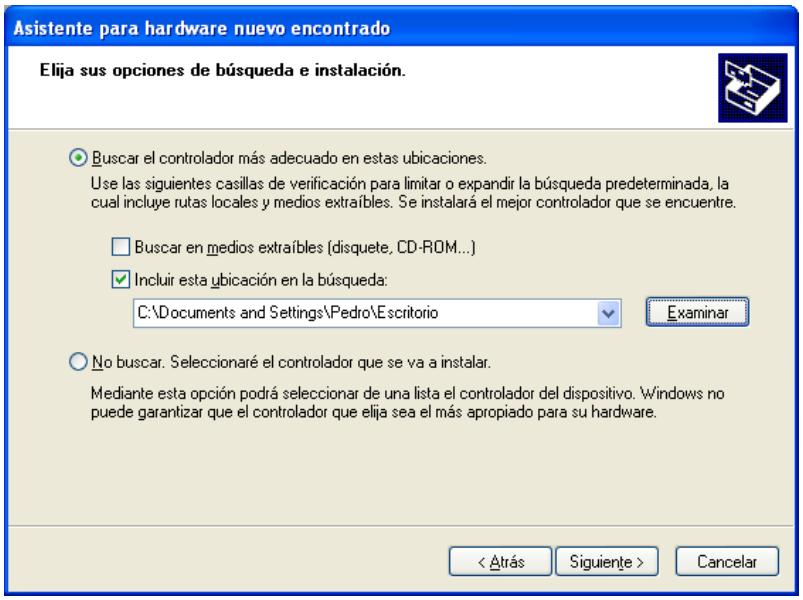
1. El host o hub detecta la conexión del dispositivo a través de sus resistencias de pull-up en el par de transmisión. El host espera 100 ms para que finalice la conexión y se estabilice la carga.
2. El host envía una señal de reset al bus para colocar el dispositivo en estado direccionado y que responda a la dirección por defecto.
3. El host en MS Windows pregunta por los 64 primeros bytes del descriptor de dispositivo.
4. Tras recibir los 8 primeros el host envía de nuevo un reset al bus.
5. El host envía una petición de set address que colocará el dispositivo en estado direccionado.
6. El host pregunta por los 18 bytes del descriptor de dispositivo.
7.El host pregunta por los 9 primeros bytes del descriptor de configuración para conocer su tamaño.
8. El host pregunta por los 255 bytes del vector de configuración.
9. El host pide los string descriptors que se hayan especificado.

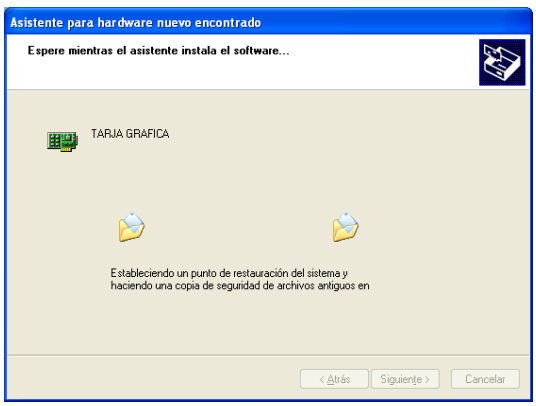
Al final de este paso nueve, Windows pedirá el driver del dispositivo y puede ser que repita de nuevo peticiones para todos los descriptores antes de realizar el set configuration.

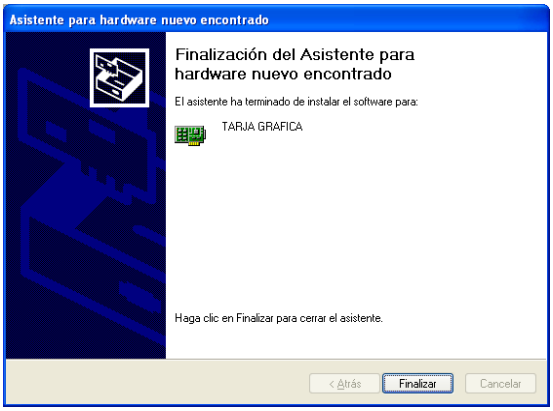




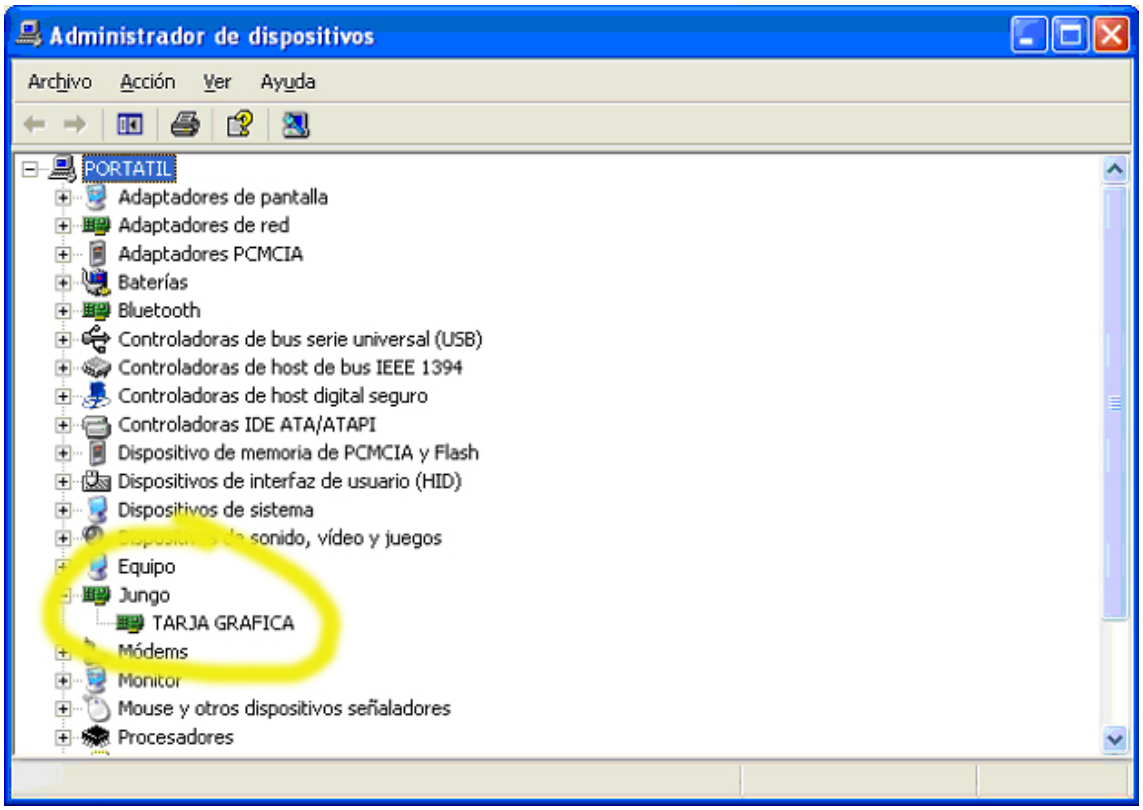
Se recoge el driver implementado con el programa Windriver del directorio en el que esté.







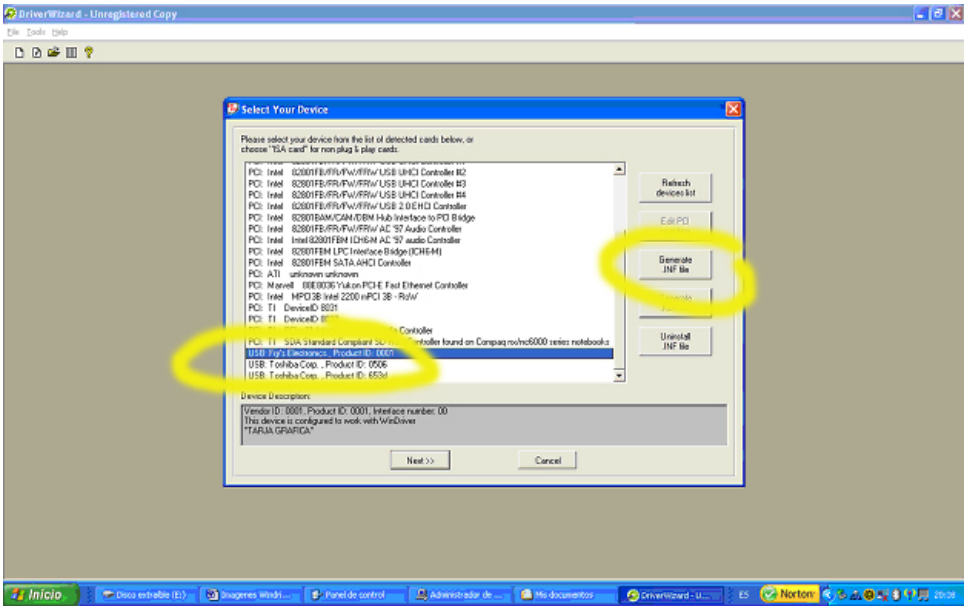
Finalmente tenemos la tarjeta reconocida por Windows y en perfecto funcionamiento.



4.3.2.5 Comprobación con Windriver

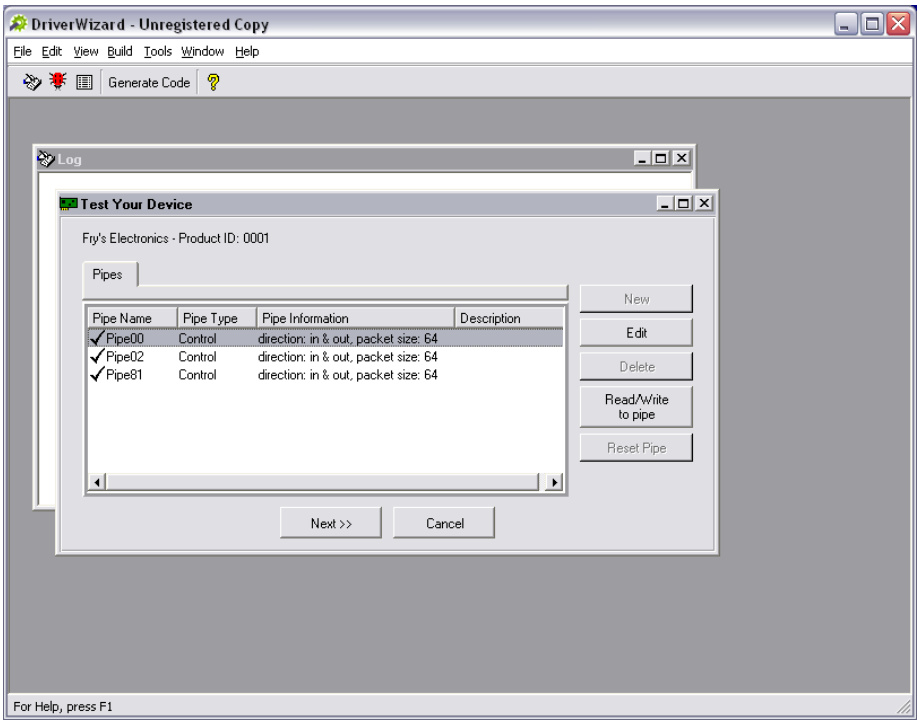
Windriver es un programa de Jungo Ltd. para la creación de drivers de puertos en windows, y el posterior desarrollo de aplicaciones sobre ellos.

Una vez reconocido por windows el dispositivo USB, se podrá generar un driver automáticamente con este programa, los únicos datos que pide son un nombre del dispositivo, otro del fabricante y los ID de Class y de Vendor, que en este caso los inventaremos pues estos se deben comprar a la entidad encargada de su distribución, aunque es posible que exista alguno destinado al desarrollo.

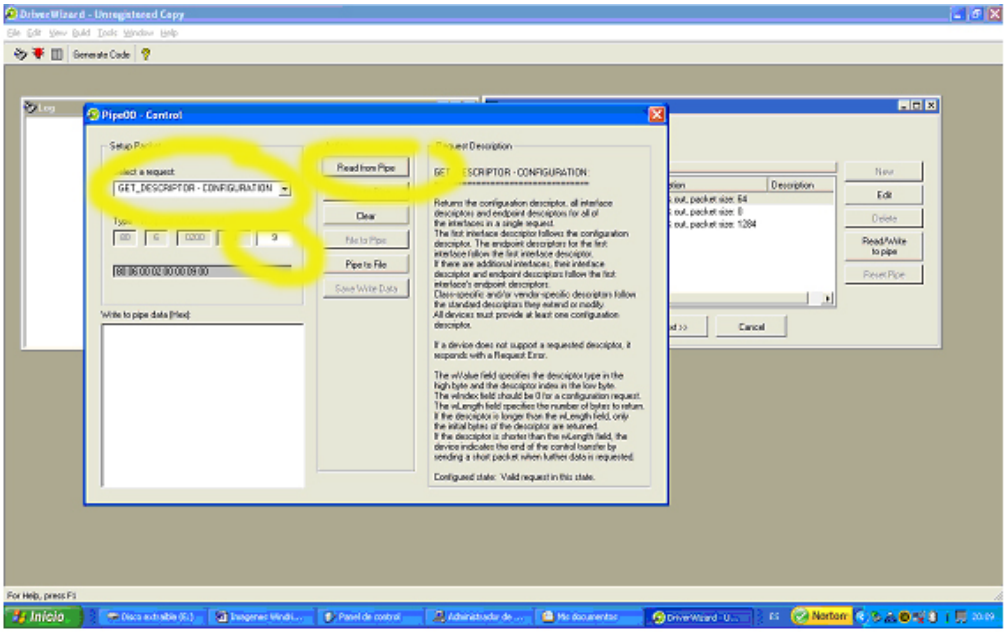


Windriver ofrece una herramienta mediante la cual se pueden enviar peticiones al dispositivo. Desde este punto se hicieron las pruebas para comprobar su perfecto funcionamiento y desde donde se puede continuar generando y probando nuevos tipos de peticiones para funciones futuras no estándar.

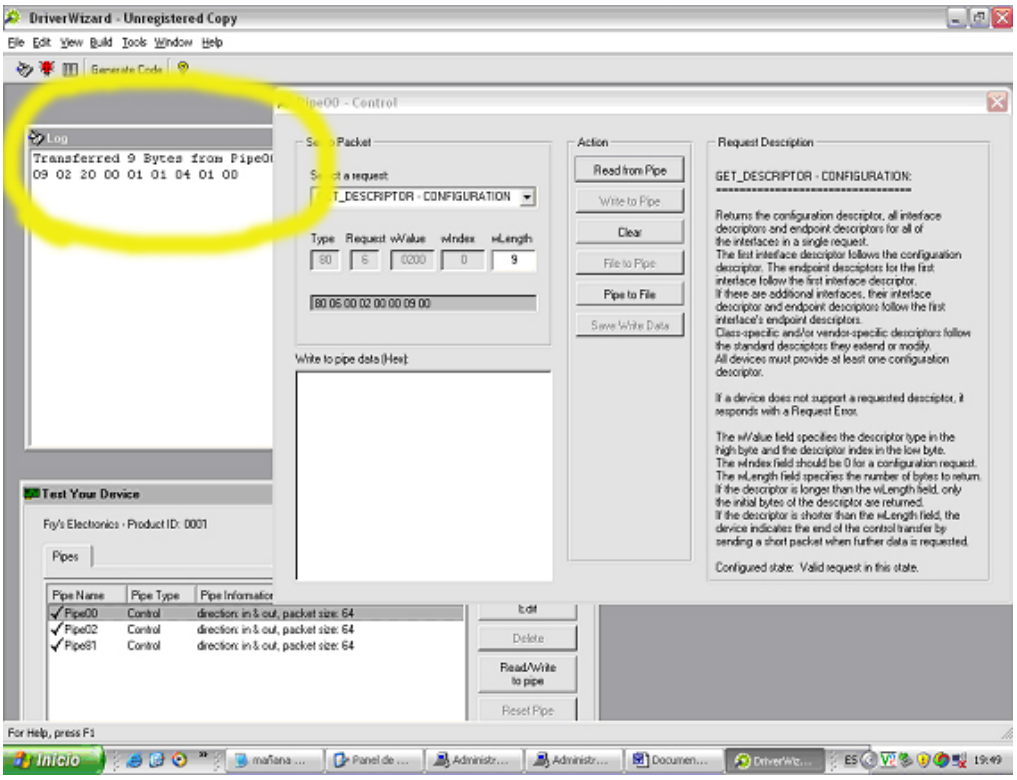
En esta figura eliges la pipe o endpoint por el que se emitirá la petición de paquete dentro de los tres endpoints definidos en el configuration descriptor.



En la siguiente figura se define el tipo de paquete, por ejemplo la bRequest Get_descriptor_configuration, que deja como único valor no fijo el wLength. Se puede peticionar la cantidad de bytes que se deseen, en este caso 9. También deja la opción de rellenar completamente todos los campos para peticiones no estándar.



La respuesta llega correctamente, únicamente los 9 primeros bytes del descriptor, 09 02 20 00 01 01 04 01 00.



Se puede comparar el valor respondido con la configuración que se introduce en el programa, 09 es la longitud del descriptor, 02 indica que es el de configuración, 20 00 la longitud total, en decimal 32, primero se envía el byte de menor peso, aunque aquí ya está representado cada byte en el orden de su significado, siendo el 2 el de mayor peso, byte a byte se establece en el orden que llega.

4.3.3 Módulos auxiliares

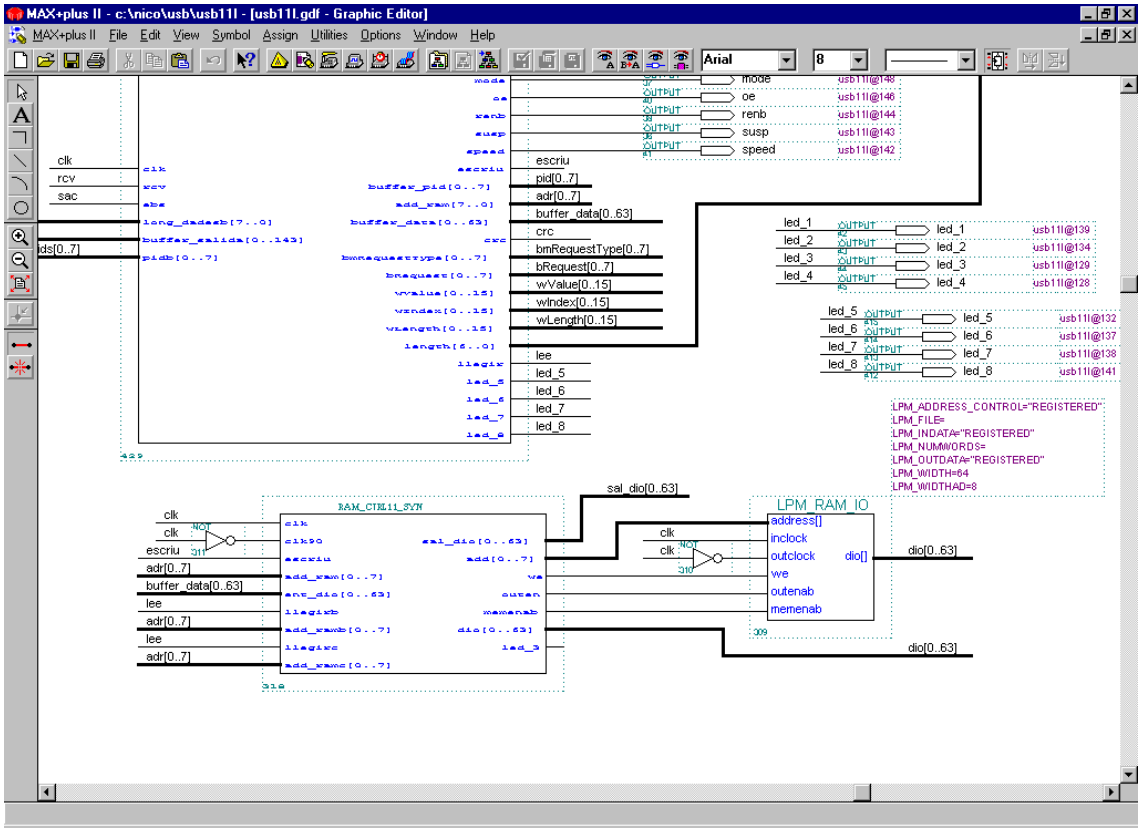
Hay cuatro pequeños módulos más en la implementación, T2, DSP_Z, RAM_CTRL11_SYN y LPM_RAM_IO cuyas funciones se explican a continuación.

El T2 es un timer o clock que adapta el período de la señal de la placa al período utilizado en el conjunto del modulo. Como para la transmisión USB se trabaja a 12MHz y el clock de la placa lo hace a 24 Mhz, hay que dividir entre 2 su período, y para ello se cambia el nivel del clock interno cada flanco de subida del clock general, de este modo convertimos un período en medio.

El DSP_Z es un módulo encargado de dejar en alta impedancia las salidas dirigidas al DSP_Z evitando que cuando este esté en uso, diferencias de nivel sobrecalienten el chip.

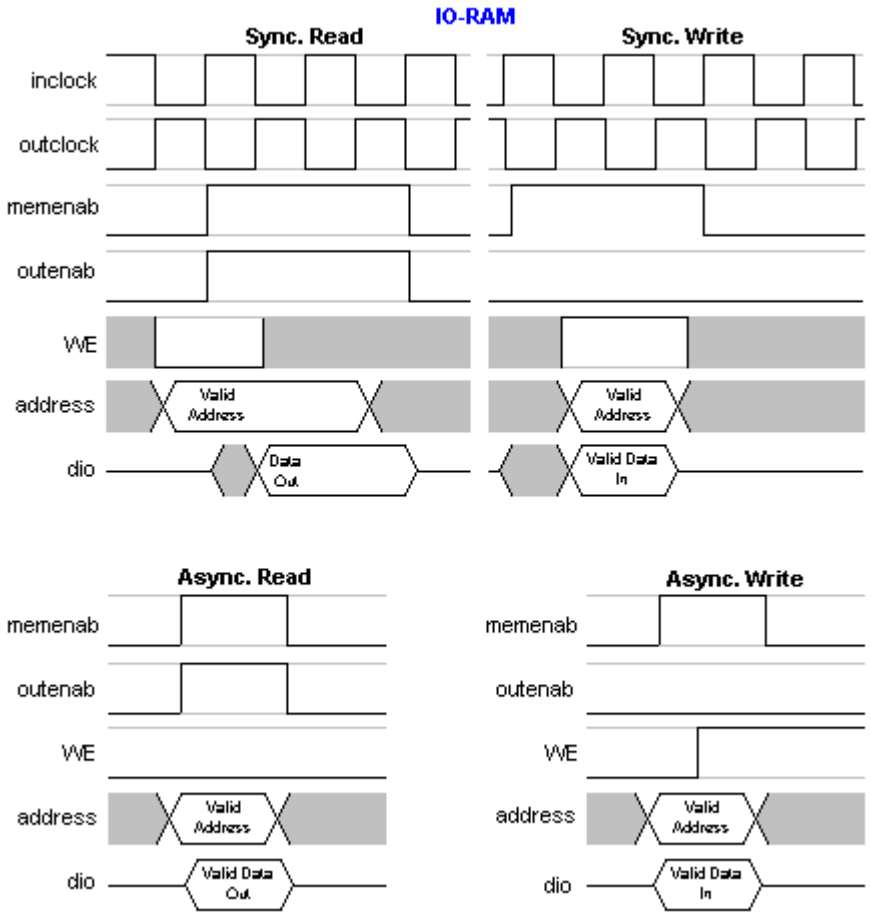
LPM_RAM_IO es un módulo preconstruido que implementa una memoria RAM lógica, y puede tener varias formas y usos en función de su configuración previa. Se configura con los siguientes parámetros: LPM_WIDTH igual a 64 que será el buffer de salida; LPM_WIDTHAD igual a ocho que será el tamaño del buffer que direcciona todas las posibles celdas de 64 bits de la RAM; LPM_INDATA con valor REGISTERED especifica que el valor de los datos de entrada deben fijarse en el bus por el inclock o el outclock antes de realizar la operación; LPM_OUTDATA con valor REGISTERED especifica que el valor de los datos de salida deben fijarse en el bus durante el flanco de subida del outclock antes de aparecer en el buffer,;LPM_ADDRESS_CONTROL con valor REGISTERED especifica que las señales we, memenab y address[ ] solo tendrán efecto durante el flanco de subida del inclock, es decir, estos parámetros colocan la RAM en un modo de funcionamiento síncrono. Es totalmente desaconsejable tratar de utilizar la memoria en modo asíncrono pues el control del

buffer en cada momento se convierte en una dura tarea. Controlar el momento en el que los datos están estables en el bus es muy complicado con tantas señales en el conjunto del programa, es por ello que también se implementa una controladora de la RAM para reducir las posibilidades de error en el control del buffer de entrada salida, implementando en ella tres bufers únicamente de escritura y uno de lectura.



RAM_CTRL11_SYN controla la RAM en modo síncrono, las señales de entrada son un buffer de 64 bits, tres posibles direcciones de 8 bits y tres señales para comenzar una lectura o escritura, de salida tenemos un buffer de 64 bits, las señales que controlan la RAM y el buffer de entrada salida de la memoria RAM.

Para realizar una lectura, se coloca sobre el bus correspondiente la dirección deseada y se pone a nivel alto la señal de lectura o escritura correspondiente. En ese momento la controladora pasa por una serie de estados, empezando por el estado “atent” del clock, que mueven las señales de la RAM para cumplir con el esquema de la lectura / escritura síncrona. El cambio de las señales se divide en dos procesos, uno controlado por el flanco de subida del inclock y el otro por el del outclock, por lo que la operación dura dos clocks del reloj general.



Esquema de tiempo de las operaciones en la RAM

5. Conclusiones y desarrollos posteriores

5.1 Conclusiones

Básicamente, la complejidad de este proyecto es la comunicación USB.No se trata simplemente de una transmisión serie de un paquete de datos con un formato específico, sino que tiene detrás todo un protocolo de comunicación mediante paquetes.

La diversidad de opciones que ofrece el protocolo USB es inmensa, abarcando todo tipo de dispositivos y utilidades del mercado, por lo que dejar el programa cerrado resulta imposible sin una función final a realizar. Recordemos que el principal objetivo del proyecto era conseguir la comunicación con el PC sin ninguna tarea especifica, e implementar una única función recortaría en gran medida sus posibilidades, y es por ello que se decidió dejarlo en este punto. En este capítulo intentaré explicar como desarrollar futuras aplicaciones a partir de esta base de programa.

Podrían haberse implementado otros formatos de comunicación serie, pero al haber seleccionado el USB como el segundo a realizar, abarcó demasiado tiempo en su realización y aun así, lejos de completar todas sus posibilidades se queda en una mínima parte de lo que puede realizar. Suficiente para establecer una comunicación y realizar tareas de control, pero también con opciones sin implementar como transferencias en masa o por interrupciones.

Se propusieron otros protocolos diferentes a continuación, pero sin duda ninguno sería tan complejo como éste. Quizás si se hubiesen desarrollado anteriormente sí se hubiesen implementado. Pero si tuviese que tomar una decisión, para continuar preferiría avanzar en este protocolo descrito, que es y será en los próximos años uno de los principales sistemas de comunicación entre dispositivos.

El protocolo RS485 es uno de los que se propusieron a continuación, pero solamente especifica las características mecánicas y físicas de la comunicación, e implementar la tarjeta adaptadora sin realizar un software para la comunicación no tenía mucho sentido, de hecho la forma en la que se transmiten los datos por el puerto

no está definida, cada uno en particular elegiría cómo transmitir los datos poniendo de acuerdo a ambas partes de la comunicación. El mismo programa que se utiliza en el RS232 sería perfectamente válido para este otro puerto, añadiendo una señal que controle el sentido de la comunicación.

5.2 Desarrollos posteriores

Al margen de la necesidad de desarrollar nuevos puertos para la tarjeta gráfica del departamento, como el Ethernet, IRDA o algún otro protocolo serie de la EIA (que tiene varios), en este capítulo intentaré detallar como evolucionar en éste en el que he estado trabajando.

Las modificaciones más sencillas serán las que utilicen el sistema de transmisión ya implementado, es decir, si nos interesa aumentar el número de tipos de petición sobre un canal de control, solo es necesario aumentar la casuística en la fase dev_wait_odata de la unidad de control. Las peticiones implementadas son las estándar, pero nada impide crear peticiones nuevas según la aplicación que quieras darle al dispositivo y confeccionar las respuestas de la placa. Solo habrá que comparar los nuevos valores decididos para bRequest, bmRequest, wValue, wIndex y wLength y devolver la información que se especifique, ya sea en la unidad de control o en cualquier otro módulo que se añada.

Si se desea establecer diferentes comportamientos a la vez y para ello crear nuevas configuraciones, interficies o endpoints deberá modificarse la información enviada en los descriptores del dispositivo en la petición get_descriptor y cambiar su longitud.

Para implementar el resto de tipos de comunicación habrá que ampliar la máquina de estados creando estados nuevos para cada tipo de comunicación y actualizando variables como ep_type que nos indica el tipo de endpoint o fasec que nos indica la fase dentro de la comunicación.

Las opciones para el envío de datos abarcan multitud de posibilidades, recoger y transmitir una señal en tiempo real, que aportaría sus datos desde el exterior del modulo, se podrán también recoger los datos de la memoria RAM exterior al módulo

que tiene la placa física o de la misma memoria RAM implementada en el módulo previamente cargándola con la información. Para tratar estos datos u otros que provengan del host deberían implementarse nuevos módulos en el interior del programa.

Otras opciones irán a la inversa, una vez implementada la enumeración del dispositivo, en resumen, el proceso a través del cual windows lo reconoce. Puede reducirse este programa al mínimo necesario incluso extrayendo partes de él, de forma que pudiese ser programado en un chip lo más pequeño posible y optimizarlo para uso en dispositivos más económicos.

Aunque en determinados aspectos el protocolo puede parecer rebuscado y complejo, ello es porque se pretende un dispositivo adaptable y compatible, lo que en la industria se traduce en económico, pues ya no son necesarios diferentes puertos para cada tipo de periférico: realizando éste se abarca una inmensa gama de dispositivos.

Es adaptable porque gran parte del protocolo queda abierto a especificaciones futuras, por ejemplo bits de los registros estándar o de los campos de las peticiones. De hecho, nuevos aspectos, como mayores velocidades en el USB 2.0 o modos de comunicación en el USB wireless, se han implementado ya y seguro que en un futuro próximo serán necesarios muchos más.

El protocolo desarrollado es compatible, porque las nuevas versiones que se realicen de los diferentes periféricos que surjan pueden interactuar con las anteriores versiones gracias a él, lo que lo coloca en situación ventajosa frente a protocolos incompatibles o que no preveieron correctamente su evolución.

De todo lo expuesto, de sus especiales características y de su adaptabilidad, se deduce que las posibilidades de este protocolo son muy amplias.

6. Bibliografía

6.1 Libros

VHDL
Douglas L. Perry
Editorial McGraw-Hill, 1994

VHDL Programming by example
Douglas L. Perry
Editorial McGraw-Hill, cop. 2002

6.2 Manuales

Tarja MAGCL
Manual de uso
Universitat de Girona

Universal Serial Bus Specification
Compaq, Intel, Microsoft, NEC.

WinDriver USB v7.00 User's Guide
Jungo Ltd.

6.3 Internet

http://www.beyondlogic.org/usbnutshell/
Página muy útil para desarrolladores del puerto USB.
http://www.graphics.cornell.edu/~westin/canon/ch03.html
Aporta conocimientos prácticos de los paquetes USB.
http://www.lvr.com/usb.htm
Magnífico ejemplo del proceso de enumeración en windows.
http://www.jungo.com/support/documentation/windriver/801/wdusb_man_mhtml/
Guía de Jungo sobre el USB, con casos prácticos extraidos a través de su programa.
http://www.microsoft.com/whdc/system/bus/USB/IAD.mspx#EBAA
Para conocer mejor el comportamiento de windows con el USB
http://www.usb.org/home
Página principal de la USB Implementers forum, Inc desarrolladores de su especificación.
http://www.altera.com/
Página principal de Altera, fabricadores del chip y el programa de desarrollo.
http://www.camiresearch.com/Data_Com_Basics/data_com_tutorial.html
Una de las mejores páginas encontradas que explican el protocolo RS232.
http://www.arcelect.com/rs232.htm
Otra página con explicaciones del protocolo RS232, este protocolo se aprendió desde un inicio a través de la web.
http://www.euskalnet.net/shizuka/rs232.htm
la tercera de la páginas empleadas en el RS232, que complementa a las otras dos.

6.4 Datasheets

max220-max249.pdf
MAXIM
Manual técnico del chip Max232, que se usa en el protocolo RS232.

MAX3340E-MAX3343E.pdf
MAXIM
Manual técnico del chip utilizado para el bus USB

Muchos otros manuales cada uno aportando pequeños detalles del protocolo se encuentran en el CD del proyecto, desde explicaciones del chequeo CRC hasta la lista de LANGIDs o idiomas soportados por el USB.