Enlace para Comunicación entre PC y FPGA a través de Interfaz USB

Proyecto Fin de Carrera

de

D. Javier Porrino Pérez

Para la Obtención del Título de

Ingeniero de Telecomunicación

Dirigido por D. Miguel Ángel Aguirre Echánove

Departamento de Ingeniería Electrónica

Universidad de Sevilla

Septiembre 2002

1. Introducción

En las páginas de este documento se recoge la información relativa al Proyecto Fin de Carrera asignado a Javier Porrino Pérez, consistente en la implementación de un enlace USB para comunicar una placa de desarrollo de lógica reconfigurable con un PC.

Este Proyecto se ha llevado a cabo entre los meses de Octubre de 2001 y Julio de 2002, y el autor ha sido dirigido en su realización por D. Miguel Ángel Aguirre Echánove, profesor del Departamento de Tecnología Electrónica de la Universidad de Sevilla, al que le agradece toda la atención y ayuda prestadas en estos meses.

El espíritu con el que se ha desarrollado el interfaz de comunicaciones que se describe en esta memoria es que sirva como punto de partida a futuros diseños y no como un diseño ya cerrado. Se ha realizado un sistema de trasvase de datos entre un PC y una placa de desarrollo que se pretende sea lo más general posible, para así poder adaptarse a cualquier entorno de trabajo y sobre el que se puedan apoyar diversas aplicaciones para realizar su intercambio de información.

Esta memoria responde a ese mismo espíritu, por lo que se ha hecho especial hincapié en aquellos aspectos que son susceptibles de ser modificados para adaptarlos a un entorno de trabajo definitivo. Por tanto, además de exponer el diseño en sí, se intenta dar ideas sobre posibles modificaciones y mejoras.

La memoria se ha estructurado de la siguiente manera:

En el capítulo 2, “Planteamiento del problema”, se definen los requisitos previos y los objetivos del proyecto así como la forma general de abordarlo. También quedará estipulado el alcance del proyecto, es decir, qué se ha hecho y qué no se ha hecho. Por tanto, aquí se esbozan las líneas principales por las que se guiará el desarrollo del documento.

En el capítulo 3 se detallan las fases en que se ha estructurado el desarrollo del proyecto, y el modo de abordar cada una de ellas. Se esbozan también los criterios que se han tenido en cuenta a la hora de tomar decisiones sobre la línea a seguir en el diseño. Por último, se hace una temporización para cada una de estas fases.

En el siguiente capítulo se introducen los conceptos teóricos necesarios para el entendimiento del proyecto. Se hace una aproximación a los aspectos que nos incumben del funcionamiento del sistema USB, explicando en aquellos puntos que sean pertinentes qué soluciones se van a aplicar a nuestro caso.

El capítulo 5 es una descripción de las herramientas, fundamentalmente de tipo software, que se han empleado a lo largo del desarrollo del proyecto.

Y por fin llegamos a la descripción del producto resultado del proyecto: en los capítulos 6 a 9 se explican las diferentes vertientes de la solución final, y por tanto constituyen el núcleo de la memoria, además de acaparar la mayor parte de su extensión. Los capítulos 6 y 7 nos dan una descripción exhaustiva de la parte hardware, y en el capítulo 8 se detalla todo el software del interfaz. El capítulo 9 explica cómo poner a funcionar el interfaz en una tarjeta de desarrollo de Xess, que es la que se ha usado para el desarrollo.

Por último, el capítulo 10 da algunas directrices para los futuros diseños que se basen en este interfaz, indicando aquellos puntos que son susceptibles de modificaciones y cómo hacer estas modificaciones.

2. Planteamiento del Problema

2.1. Marco del Proyecto

Este proyecto se enmarca dentro del proceso de creación de una placa de desarrollo basada en la FPGA Virtex 800 de Xilinx que está llevando a cabo el Grupo de Tecnología Electrónica de la Universidad de Sevilla a cargo del Profesor D. Miguel Ángel Aguirre Echánove. El objetivo de dicha placa es proporcionar un entorno de diseño y depuración de circuitos digitales de gran tamaño. Para ello, se aprovecha la gran capacidad de reconfiguración de la FPGA antes mencionada, a la vez que se requiere de un sistema de comunicación con el PC que sea rápido y fiable.

En este punto es donde entra en acción el presente proyecto. Su finalidad es crear un interfaz USB para comunicar la tarjeta con un PC mediante este protocolo serie de alta velocidad, que se añadirá al ya existente interfaz EPP a través del puerto paralelo. De este modo, una vez implementado el enlace USB, por él se podrá acceder a la tarjeta con el fin de programarla, intercambiar datos o realizar cualquier operación con los distintos componentes que incluye (FPGAs, oscilador programable, FLASH,...).

2.2. Objetivo del Proyecto

El objetivo de este proyecto es proporcionar un enlace USB que sea capaz de intercambiar una gran cantidad de datos entre un PC y un dispositivo de lógica reconfigurable (FPGA, CPLD, ...) a alta velocidad.

Para lograr una funcionalidad completa, el proyecto incluye tanto el diseño y desarrollo de la circuitería necesaria, que se describirá con VHDL y se implementará mediante las herramientas software que Xilinx pone a nuestra disposición, como la parte de software necesaria para comunicarse con la tarjeta. Dicho software se desarrollará en principio bajo C y C++ y tendrá como entorno operativo el sistema Windows 98 de

Microsoft, por ser el primero que incluye soporte completo para USB y estar a nuestra disposición en los laboratorios del departamento.

2.3. Requisitos

¿Qué se le pide a nuestro sistema? Las características que debe cumplir son las siguientes:

- Que sea capaz de intercambiar grandes cantidades de datos.
- Que la transferencia sea fiable, es decir, libre de errores.
- Que consiga una buena tasa de transferencia.
- Que la circuitería sea lo más general posible, ya que el entorno de desarrollo (la tarjeta XSV 800 de Xilinx sobre la que se va a realizar el proyecto) es diferente del entorno de trabajo final del sistema (la tarjeta de desarrollo diseñada por el Grupo de Tecnología electrónica). Por tanto, se pide que realizar esta transición no dé ningún tipo de problemas, pueda hacerse automáticamente.
- A su vez, que la modificación de esta circuitería para adaptarla a su uso final sea lo más sencilla posible y esté bien documentada, para que las personas que quieran usar este interfaz para sus propios desarrollos no pierdan mucho tiempo en la comprensión del circuito.
- Que se proporcione un software libre de fallos y una interfaz de programación sencilla.

2.4. Elección de la tecnología. Restricciones

Actualmente la especificación USB con la que se está trabajando en el ámbito comercial es la 2.0, que data de Abril de 2000 y en la que se define el modo "high-speed", con una tasa de bits de 480 Mbits/seg. Sin embargo, aunque lo que se busca con este proyecto es una velocidad elevada y ésta sería más que suficiente, no se va a adoptar esta especificación por tener poca difusión en el mercado: hay a la venta pocos equipos que funcionan en modo high-speed, las tarjetas controladoras de USB 2.0 son costosas y los sistemas operativos actuales (Windows XP, 2000) no disponen todavía de los drivers apropiados para hacerlas funcionar correctamente. Ni que decir tiene que

nunca existirá este software para Windows 98, por lo que para soportar USB 2.0 se necesitaría un sistema operativo más moderno.

Por los motivos expuestos anteriormente, este trabajo se va a basar en la especificación USB 1.1, que salió en 1998 y con la que se corrigen los fallos encontrados en versiones anteriores. En esta especificación se definen dos modos de funcionamiento, "low-speed", que trabaja con un reloj de 1,5 Mbits por segundo, y "full-speed", con un reloj de bus de 12 Mbits/s. Es en este último modo en el que nos centraremos.

Por último, indicar que no se pretende que el diseño final sea cien por cien compatible con la especificación USB, ya que ello implicaría, entre otras cosas, la necesidad de incluir en la placa final un microcontrolador que implementara las instrucciones a las que debe responder obligatoriamente cualquier dispositivo USB, y se restaría eficacia al resultado. Por el contrario, el objetivo es eliminar todas aquellas características que no sean estrictamente necesarias para comunicar el PC con la placa, ganando así probablemente en velocidad, sin que se degraden otras características de USB como puede ser la conexión de distintos equipos simultáneamente al mismo bus. Nuestro límite serán las restricciones impuestas por el controlador del bus USB, que es el hardware del PC que controla todas las transacciones por la línea física y que a su vez está controlado por el sistema operativo, por lo que nos tendremos que adaptar a su forma de trabajar.

2.4.1. Justificación de la solución elegida

¿Por qué USB 1.1? Esta es la pregunta a la que intentaremos dar respuesta en estos párrafos. En primer lugar, comentaremos el aspecto de la velocidad. Actualmente, con el interfaz EPP implementado en la actual versión de la tarjeta de desarrollo se consiguen tasas de transferencia de información del orden de los 2 Mbytes/s. Trabajando con USB en el modo "full-speed", la velocidad del bus es fija, 12 Mbits/s como ya se ha comentado, pero el ancho de banda resultante para una aplicación concreta depende de muchos factores como pueden ser el número de dispositivos conectados al bus o el tipo de transferencias a realizar (como se verá en el apartado

4.2.3, USB tiene 4 modos de transferencia). En todo caso, debido al protocolo USB, que incluye cabeceras, asentimientos, retransmisiones, etc., esto nos lleva a estimar que en las condiciones más óptimas y con un diseño muy depurado, se pueden llegar a alcanzar los 10Mbits/s, lo que equivaldría a unos 1,25 Mbytes/s. Por tanto, la velocidad de transferencia será del orden de la actual, aunque se deja la puerta abierta a que en un futuro se adapte el sistema a la especificación USB 2.0, con lo que esta velocidad se multiplicaría.

Pero principalmente han sido otros factores los que han llevado a elegir esta solución. Veamos algunas de las virtudes del bus USB (Universal Serial Bus):

- Conector estándar: todos los dispositivos llevan el mismo conector, por lo que se evita el problema de a qué conector va este enchufe.
- Actualmente, el número de dispositivos que se puede conectar a un PC es limitado, cada dispositivo tiene un conector distinto y no hay espacio. Con USB, a un solo puerto puedo conectar hasta 127 dispositivos mediante hubs.
- Por tanto, se evitan los conflictos por compartir interrupciones, ya que con una sola línea de petición de interrupción (la del controlador USB) se gestionan múltiples dispositivos.
- Los dispositivos se detectan y configuran automáticamente en USB, evitándole al usuario esta tediosa tarea. Por tanto, se permite el enchufe y desenchufe en caliente, sin necesidad de reinicios.
- Es una solución que alcanza un buen compromiso entre coste y prestaciones.

Todas estas características han hecho que USB sea una tecnología en auge, y cada vez son más los dispositivos de uso cotidiano que aparecen en el mercado. Por tanto, es de esperar que a largo plazo acabe reemplazando a los actuales puertos de comunicaciones, y es éste factor el que nos ha hecho decidirnos por crear una solución USB para comunicar un PC y el sistema de desarrollo. No pretende de momento sustituir al interfaz paralelo, pero sí complementarlo.

2.5. Alcance del Proyecto

En este apartado se va a definir qué es lo que se ha incluido en este proyecto y qué es lo que no, para que sirva de referencia para la evaluación de las distintas fases del mismo. El proyecto terminado incluye los siguientes componentes:

2.5.1. Código VHDL sintetizable

Se incluyen varios ficheros de código VHDL en los que se describen los distintos componentes que se han implementado:

- el nivel físico de USB.
- la parte imprescindible del nivel lógico de USB (USB device framework), para el funcionamiento dentro del sistema USB.
- El nivel de dispositivo USB, es decir, nuestra aplicación.

Se incluye también un fichero con las restricciones en cuanto al patillaje del circuito, específico para la FPGA Virtex 800.

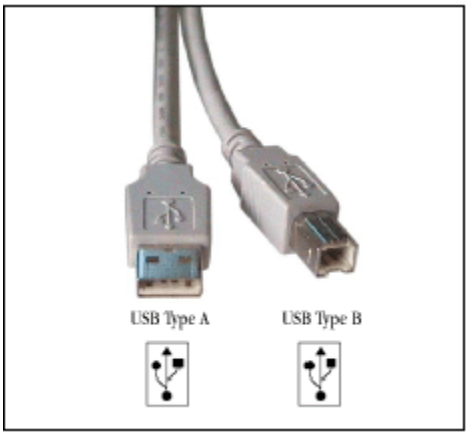
En lo que se refiere al tamaño del circuito final, tras realizar las debidas optimizaciones éste ha quedado en torno a las 8500 puertas lógicas, lo cual es un tamaño bastante considerable. Esto es debido a que el control del bus serie es más complejo que el del paralelo, porque el primero puede ser compartido por distintos dispositivos y por tanto, entre otros, necesita mecanismos de identificación, así como convertidores serie/paralelo y viceversa, etc. Por otro lado, el nivel lógico de USB está pensado para ser implementado por un microcontrolador y una ROM que procesen instrucciones, mientras que aquí todo eso se hace a nivel de hardware, lo que ha incrementado mucho el tamaño del resultado final.

2.5.2. Elementos hardware adicionales

Como se ha mencionado anteriormente, el bus USB funciona a una velocidad de 12 MHz. Esta restricción ha hecho que se optara por diseñar el circuito para funcionar

con un reloj de 12MHz. Cualquier otra frecuencia de reloj que se le aplicase al circuito haría que éste no funcionase. El problema está en que con el oscilador programable que viene en la tarjeta XSV-800 la frecuencia más cercana que se puede conseguir es 12.5 MHz, pero esta frecuencia está fuera del rango de tolerancia del circuito, y por lo tanto no funciona. La solución que se ha adoptado ha sido diseñar una pequeña placa impresa que incluye un oscilador programable del mismo modelo que el de la XSV (Dallas DS1075M), pero con una frecuencia base de 60 MHz, de modo que al dividirlo por 5 obtenemos los 12 MHz que necesitamos. Este circuito se puede programar directamente desde la XSV, de forma que su uso es muy sencillo, y se explica en el capítulo 7 de este documento. En el proyecto se incluye por tanto esta placa con el oscilador, así como un fichero con el diseño del PCB en formato del programa TANGO.

Por otro lado, también es necesario en el proyecto el cable USB que comunique el PC con la tarjeta de desarrollo. Normalmente, para la conexión de los periféricos mediante USB se usan cables asimétricos: tienen un conector distinto por cada lado, ya que los receptáculos del PC y del periférico son distintos. Hay conectores tipo 'A' (el del PC) y tipo 'B' (el del periférico). Estos conectores son los que se ven en la figura.



Este cable sería un cable A-B, y es el estándar de USB. En nuestro caso, no podemos usar este cable ya que el receptáculo que trae la tarjeta XSV-800 es de tipo 'A', por lo que necesitamos un cable tipo A-A, es decir, con ambos conectores de tipo 'A'. Aunque este tipo de cable está expresamente prohibido por la recomendación USB, es fácil encontrarlo en cualquier tienda de accesorios informáticos.

2.5.3. Software

Se deberá entregar el código fuente del software necesario para implementar el trasvase de información entre PC y placa de la forma más eficiente posible, es decir, consiguiendo una tasa de transferencia optimizada. En principio se barajaron dos alternativas posibles para la estructura de estos drivers:

1. Que estos programas interactúen directamente con el hardware del "host controller", es decir, controlando los registros del controlador del bus, lo cual requeriría programación a bajo nivel, que sería una solución mucho más efectiva al manejar directamente el ancho de banda del bus, pero mucho más costosa en tiempo de desarrollo y en complejidad. Además, en el mercado hay definidos dos estándares de controlador totalmente diferentes,"UHC" (Universal Host Controller) y "OHC" (Open Host Controller) por lo que el diseño para manejar uno de ellos no valdría para el otro. Resultado: el sistema no funcionaría en todos los equipos.
2. Que el software interactúe con los drivers de USB (system software), lo cual generaría un código mucho más sencillo y válido para cualquier equipo (sobre el que corra Windows 98), aunque cabe esperar que menos eficiente.

Finalmente se ha optado por la segunda solución, por dos razones: la relativa sencillez de la programación y para obtener un resultado lo más genérico posible, que valga para todo equipo que soporte USB y tenga un sistema operativo Windows XX.

Por tanto, se entrega el código fuente de dos programas: un driver que interactúa con la pila de drivers de USB, llevando la gestión básica del dispositivo, y un programa de aplicación a modo de ejemplo, con interfaz gráfica de usuario al estilo de Windows. Esta aplicación sirve para demostrar cómo debe escribirse un programa que interactúe con el enlace USB. Junto con los códigos fuente comentados se entregan los ejecutables y los archivos necesarios para la correcta instalación del driver. Ambos programas se explicarán más tarde en esta memoria.

3. Estrategia de desarrollo

Una vez hemos introducido el problema y hemos definido las herramientas que hemos utilizado, es momento de exponer cómo se ha abordado la solución. Para ello, vamos a diferenciar una serie de fases que se han ido completando hasta llegar al final del desarrollo, es decir, al término del proyecto. Dichas fases, que se describirán con profundidad en los siguientes párrafos, son las siguientes:

- **Fase 0: Estudio del problema.**
- **Fase 1: Desarrollo de un sistema básico de comunicaciones por el Bus Serie Universal que sea operativo.**
- **Fase 2: Optimización de este sistema de comunicaciones.**
- **Fase 3:Realización de una aplicación de ejemplo.**
- **Fase 4: Documentación y presentación de resultados.**

Cabe señalar que cada una de las fases de desarrollo expuestas arriba abarca dos vertientes: diseño del hardware y diseño del software. El hardware será el que se implemente en la tarjeta de desarrollo, en este proyecto la FPGA Virtex 800 de Xilinx, y el software será el que corra en el PC (host en terminología de USB), y será el encargado de coordinar la comunicación entre PC y placa de desarrollo. No se pueden separar estos dos aspectos en distintas fases ya que cada uno necesita del otro para el funcionamiento del sistema, es decir, sin el software no se puede probar el hardware y viceversa. Por tanto, ambas vertientes se irán desarrollando en paralelo.

A continuación se va a exponer el contenido de cada una de las fases de desarrollo del proyecto. Como ya se ha comentado, el desarrollo del hardware y del software se irá llevando a cabo de forma simultánea en cada una de las fases. Indicaremos también las herramientas utilizadas en cada parte del proceso.

3.1. Fase 0: Estudio del Problema

Esta fase consiste en la recopilación y estudio de toda la información necesaria para poder empezar a trabajar en el proyecto de forma práctica, así como de toda aquella materia y herramientas que puedan ser útiles en fases posteriores. Se pretende documentarse y entender las distintas facetas que presenta el problema a resolver. Para ello, desde el inicio de la fase se recopiló información en forma de libros, documentos escritos y se realizó una extensa búsqueda por internet. Asimismo, se consiguieron varias utilidades software para el desarrollo y testado de sistemas USB, tanto en la vertiente del hardware con en la del software. La búsqueda de información se concentró sobre todo en los siguientes campos:

- Sistema USB: Se recopilaron las especificaciones 1.1 y 2.0, varias notas de modificaciones, las especificaciones para los controladores (UHC y OHC), documentos para el desarrollo de dispositivos estándar, etc. También se encontró un interfaz USB escrito en VHDL para ser implementado en la placa XSV de Xess, con licencia GNU General Public License, de la empresa Trenz electronic, que nos puede ser útil a modo de ejemplo.

- Diseño software: Información diversa relacionada con la implementación de drivers USB en Windows 98, así como para el desarrollo de drivers en general bajo C++. También sobre cómo está soportado USB en Windows 98, es decir, sobre el "USB system".

- Tarjetas de desarrollo: Se recopiló el manual de la tarjeta XSV800 de Xess que será empleada en las primeras fases del proyecto.

- Utilidades USB: Se han recopilado diversas, como filtros para analizar los paquetes que circulan por el bus USB, herramientas automáticas de generación de drivers, analizadores del estado del sistema USB en un PC, etc, que han sido útiles en las diversas fases del proyecto.

El resultado del desarrollo de esta fase fue un anteproyecto que se presentó en diciembre de 2001, y que ha servido como base para el desarrollo de este documento.

Con el análisis de toda esta información estamos en condiciones de avanzar hasta la siguiente fase.

3.2. Fase 1: Desarrollo de un sistema básico de comunicaciones por el USB que sea operativo.

En esta fase se pretende implementar un código en VHDL y un software básico que sean capaces de intercambiar datos de forma lógica. Es decir, queremos conseguir trasvasar datos entre PC y dispositivo USB. Para ello, inicialmente utilizaremos como soporte del hardware la placa de desarrollo XSV800 de Xess, que tiene una FPGA Virtex 800 de Xilinx y soporte para USB. Sobre ella iremos probando circuitos escritos en VHDL hasta conseguir el intercambio de información con el PC.

Esta fase sirve también para determinar las vías más adecuadas para nuestro diseño: en hardware, se ha de decidir si se usa solamente el "pipe" de control, o si además de éste hay que implementar algún otro para otro tipo de transferencias, ya sean de control, bulk, interrupción o incluso isócronas. (para entender la terminología de USB, ver el capítulo "Introducción a USB". Finalmente, se ha optado por implementar solamente el "pipe" de control, dado su buen resultado.

En cuanto al software, se diseñó un programa sencillo en C que permitía escribir o leer bytes del dispositivo. Este programa interactúa con un driver, también esbozado en C, que es el que se comunica con la pila de drivers de Windows 98. Esta es la forma recomendada para trabajar con USB, y la más sencilla.

En esta fase se probó el interfaz USB de trenz electronic (mencionado en el punto anterior), aunque pronto hubo que desecharlo debido a que el módulo creado por trenz electronic funciona con un reloj de 48 MHz y la placa de Xess tiene un oscilador que la frecuencia más cercana que da es 50 MHz, lo que hizo que el sistema no funcionara correctamente.

Además, como ya se ha mencionado anteriormente, para obtener la frecuencia de 12 MHz necesaria hubo que construir una pequeña placa impresa con un oscilador programable y conseguir que suplantara al oscilador original. Esto fue fácil debido a la gran capacidad de configuración de la tarjeta XSV de Xess, y su sistema de jumpers. El proceso de conexión así como la placa están descritos en un apartado posterior.

3.3. Fase 2: Optimización

Ya tenemos un sistema que funciona. El siguiente paso es optimizarlo para conseguir una tasa de transferencia de datos lo mejor posible. Para ello, se estudiaron dos vertientes:

En cuanto al hardware, por un lado intentaremos meter el máximo de datos en cada paquete USB. Es decir, en la fase anterior usábamos un paquete de datos para enviar una sola instrucción de lectura o escritura (una dirección y un dato). Ahora, como el tamaño máximo de datos en un paquete es definible por el usuario, intentaremos meter en un mismo paquete el máximo posible de parejas dirección / dato. En cada instrucción de lectura se pueden leer simultáneamente hasta 4 direcciones, y en cada instrucción de escritura se pueden escribir hasta 254 datos. Con esto conseguiremos el máximo aprovechamiento del ancho de banda.

Por lo que respecta al software, éste será posiblemente el cuello de botella del sistema y el que limitará el ancho de banda disponible, dadas las restricciones que impone el sistema operativo elegido. Se diseña de tal modo que su funcionamiento concuerde con el del hardware tal como se ha explicado en el párrafo anterior, es decir, empaquetando los datos de forma que se envíen paquetes lo más grandes posible.

3.4. Fase 3:Realización de una aplicación de ejemplo.

En esta fase se ha diseñado una aplicación que demuestra el uso del interfaz. Esta aplicación consta tanto de hardware escrito en VHDL como de un programa de

ejemplo, y lo que se hace es intercambiar datos entre un búfer de 256 bytes en la RAM de la placa y el programa. Esta aplicación tiene como finalidad servir como base de futuros desarrollos basados en el interfaz USB. Se explica más detenidamente en el apartado 6.5. en su vertiente hardware (implementada en la FPGA), y en el apartado 8.4. en cuanto al software.

3.5. Fase 4: Documentación y Presentación

En este punto hemos llegado al término del desarrollo del proyecto. Por último, se elaborará la correspondiente documentación del resultado y se expondrá ante el tribunal pertinente para su aprobación. En caso de que fuese necesario, se volvería sobre fases anteriores para hacer las modificaciones pertinentes para la aprobación del documento.

3.6. Temporización del Proyecto

En este punto se expone un resumen de la duración e incidencias de cada una de las fases, así como de los tiempos en que se han ido completando:

- **Fase 0:** Esta fase se inició a principios de octubre de 2001 y concluyó sin problemas a mediados de diciembre, con la presentación de un anteproyecto. Su duración, por tanto, 2 meses y medio.
- **Fase 1:** Esta ha sido la fase más larga, dado que se ha empleado bastante tiempo en analizar las distintas alternativas que se presentaban. También porque conlleva el grueso del proyecto: diseñar el motor del interfaz en VHDL, y el esbozo del driver. Se inició a principios de enero, y concluyó a mediados de abril. 3 meses y medio.
- **Fase 2:** Esta fase es bastante más corta, dado que ya sólo implica modificaciones sobre el original. No hubo incidencias y concluyó en 15 días, a principios de mayo.

- **Fase 3:** En esta fase se diseñó el nivel de "aplicación", por lo que de nuevo hubo que diseñar hardware y programar la aplicación de ejemplo. Se construyeron los circuitos definitivos y se depuró hasta que el sistema funcionó completamente. Llevó aproximadamente un mes, concluyendo a finales de mayo.
- **Fase 4:** Es la fase actual. Hay que escribir toda la documentación y realizar los trámites administrativos, así como la lectura del proyecto. Se prevé que concluya en la primera quincena de Julio.

Como resumen, veamos un cuadro con la temporización:

Planificación del Proyecto			
Fase	**Duración**	**Inicio**	**Fin**
Fase 0	2 meses y medio	Octubre	Mediados diciembre
Fase 1	3 meses y medio	Inicio enero	Mediados abril
Fase 2	15 días	Mediados abril	Final abril
Fase 3	1 mes	Principio mayo	Final mayo
Fase 4	1 mes y medio	principio junio	Principio julio

4. Introducción a USB

A continuación se expone un breve resumen de la especificación 1.1 de USB. La intención es simplemente dar a conocer aquellos conceptos básicos para el entendimiento del proyecto, por lo que nos centraremos en éstos y dejaremos de lado el resto de particularidades de la especificación. Para conocer la especificación completa, ésta se puede consultar y descargar en la web oficial de USB, http://www.usb.org/developers/.

Comenzaremos describiendo los distintos componentes que intervienen en un bus USB, para pasar a ver el modelo de comunicación. Antes que nada, hay que decir que en USB todas las transacciones las inicia el software del host (el PC), por lo que un dispositivo sólo puede enviar datos cuando se lo indique el software del host.

4.1. Componentes del sistema USB

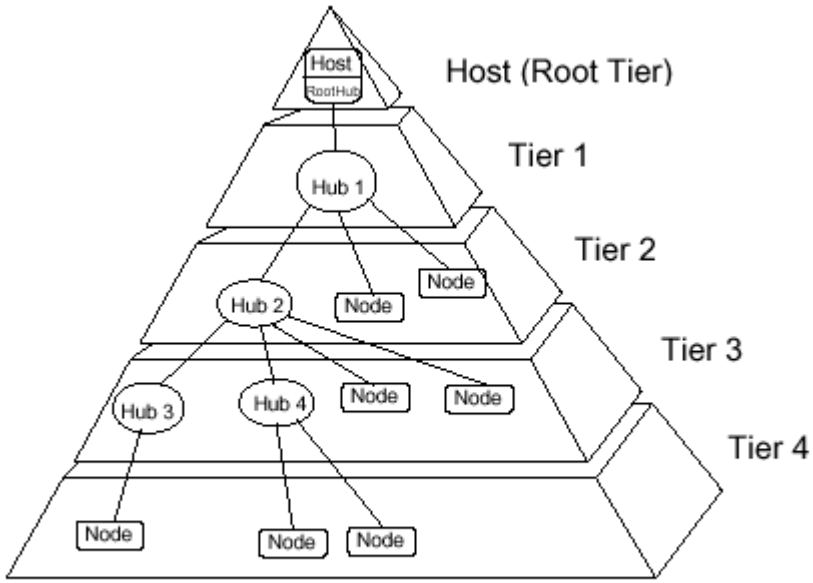


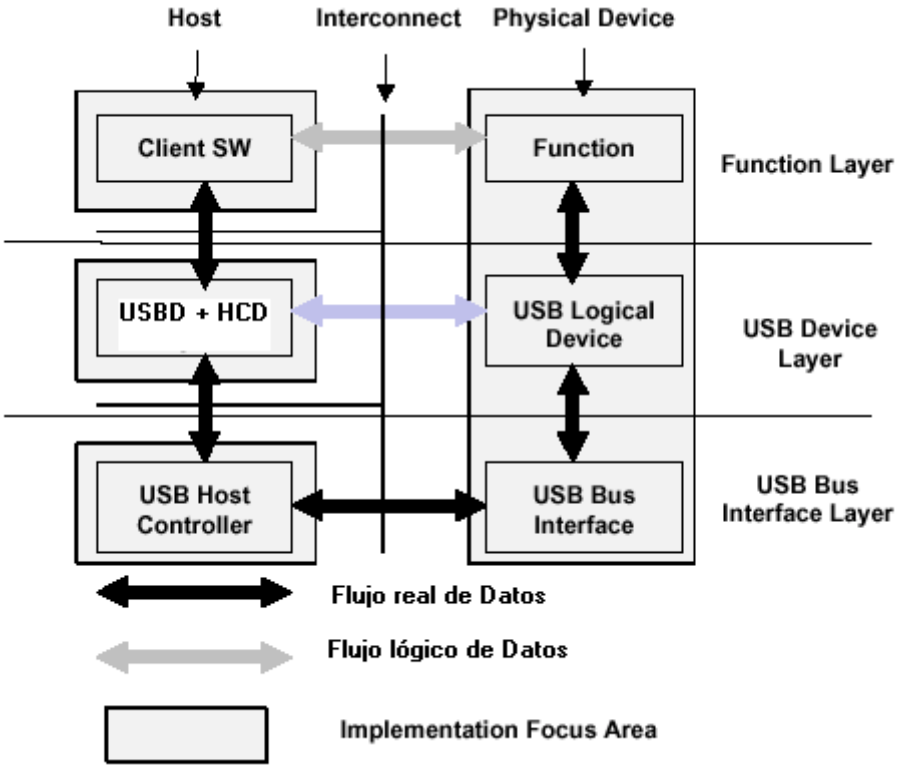
Figure 4-1. Bus Topology

En esta figura podemos observar la topología básica de un sistema USB. Fundamentalmente tenemos en el nodo superior el host que es el que dirige todo el

proceso de intercambio de información (en nuestro caso sería el PC). A él se conectan el resto de los nodos con una topología en forma de estrella, para intercambiar información con el host. Hay dos tipos de nodos, hubs y dispositivos. Los hubs hacen posible conectar tantos dispositivos como queramos, ampliando el número de puertos disponibles para conectar dispositivos. El objetivo del sistema es que los dispositivos intercambien información con el host. Para ello, tenemos los siguientes componentes:

- **Componentes Hardware**
  - USB Host controller o controlador (chip en el PC)
  - Dispositivo USB (en nuestro caso la placa XSV)
- **Componentes software**
  - Drivers del dispositivo (client software)
  - Driver USB (USBD)
  - Driver del controlador (Host Controller Driver, HCD)

La relación entre todos ellos la podemos ver en la siguiente gráfica:



4.1.1. Componentes Software

- **Drivers del dispositivo**: Es el objeto del proyecto, en su vertiente de software. En nuestro caso, comprende tanto el driver (bulkusb.sys) como el programa de aplicación. Envían peticiones de transferencia de datos (IRPs) al driver USB. Estas peticiones inician una transferencia entre el host y un dispositivo determinado. Al enviar la IRP al driver USB (USBD), hay que proporcionar un búfer en memoria donde se almacenan los datos a enviar o recibir. Este componente se comunica con el USBD y no tiene conocimiento de los mecanismos de transferencia USB.

- **Driver USB (USBD)**: No es objeto de nuestro desarrollo, sino que forma parte del Sistema Operativo. Conoce las características del dispositivo objetivo de la transferencia, ya que en un paso previo ha configurado este dispositivo. Cada dispositivo se puede configurar con una serie de "canales" (pipes) de distintas características (ancho de banda, prioridad, etc.), de modo que el USBD sabe qué canal o canales tiene abiertos el dispositivo y adapta la información a transferir a este formato. Para esto, tiene en cuenta las capacidades del dispositivo anteriormente mencionadas, las necesidades del software cliente y la ocupación del bus. El USBD es independiente de la implementación concreta del controlador, aunque forma parte del Sistema Operativo de la máquina (en Windows 98, usbd.sys).

- **Driver del controlador**: Se encarga de organizar las distintas transacciones a realizar en el tiempo. Para ello, divide el tiempo en intervalos de 1ms., y la información a transmitir la separa en transacciones individuales que se llevarán a cabo en uno o varios marcos de 1 ms. (El bus USB se temporiza en intervalos de 1 ms.,"frames"). Así, va colocando las transacciones para ir rellenando los intervalos. Esta ordenación depende de muchos factores, como el número de dispositivos conectados al bus y el ancho de banda requerido por cada uno. Por eso, no se puede saber cuánto tardará en llevarse a cabo una determinada transferencia completa, ya que se puede dividir en distintas transacciones en distintos intervalos. El HCD es dependiente de la implementación física del

controlador, ya que accede directamente a sus registros y conoce su modo de funcionar.

4.1.2. Componentes Hardware

- **USB Host Controller:** Forma parte del hardware del PC, y lleva a cabo las transferencias programadas por el HCD. Cada transferencia se identifica por una serie de datos, como la dirección del dispositivo, el tipo de transferencia y la dirección del búfer de memoria para los datos. El host controller trasvasa los datos necesarios del búfer al dispositivo o viceversa, controlando para ello directamente las líneas del bus. Se han desarrollado dos tipos de controladores, el Universal (UHC, Universal Host Controller), y el abierto (OHC, Open Host Controller). Ambos hacen el mismo trabajo, aunque de diferente manera.

- **Dispositivo USB:** Es el objetivo principal de nuestro diseño. Es cualquier dispositivo que realiza una función determinada, independiente de su sistema USB (ej: en nuestro caso, la función, que se implementa a modo de ejemplo, es un búfer de lectura y escritura). Esta función necesita comunicarse con el host y para ello dispone de un interfaz físico USB, que transmite datos en la forma necesaria, y de un "dispositivo lógico USB", que sería aquella parte del dispositivo que entiende las instrucciones que le llegan por el bus para hacer posible la comunicación. El dispositivo lógico debe guardar una serie de descriptores, que son estructuras de datos en los que se indica cómo funciona el dispositivo (ancho de banda requerido, tipo de transferencia, etc.). Al enchufar el dispositivo al host, éste detectará la conexión y le pedirá estos descriptores al dispositivo, para saber cómo se va a realizar la comunicación. Con esta explicación se puede entender mejor la parte correspondiente al dispositivo USB en la figura anterior.

  Ya se ha explicado anteriormente que los dispositivos USB pueden ser "full-speed" (funcionan con una tasa de bits de 12 Mb/s), o "low-speed" (1,5 Mb/s).

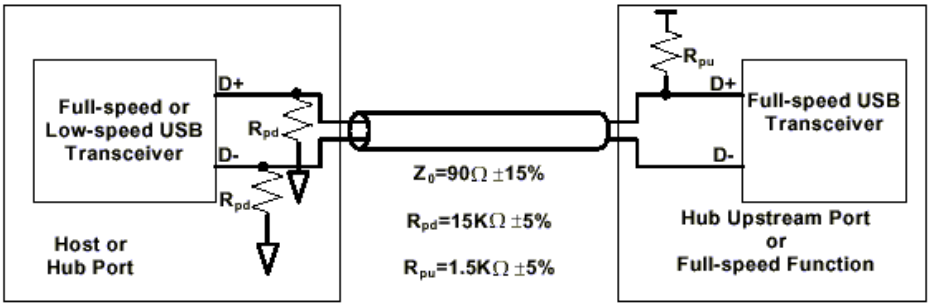
4.2. Modelo de Comunicación USB

En este apartado se va a explicar cómo se trasvasa la información a través del bus, del host al dispositivo y viceversa. Veremos los requisitos físicos, el formato de las transacciones y el proceso de configuración y puesta en marcha de un dispositivo.

4.2.1. Breve descripción del nivel eléctrico

El bus USB está formado por 4 líneas: Una transporta alimentación (Vcc = 5 V) y otra tierra (GND), para que el dispositivo se pueda alimentar directamente del bus y no requiera una fuente externa. Esta alimentación puede tener restricciones, por lo que no la vamos a usar en nuestro diseño, dado que disponemos de la alimentación general de la tarjeta XSV. Por otro lado tenemos dos líneas de datos diferenciales (D+ y D-), por las que se transmiten los datos serie con codificación NRZI. Cuando ambas líneas tienen valores de tensión opuestos (una 0 V y la otra aprox. 3 V) , señalizan un "0" o un "1", pero también se utilizan otros valores: ambas líneas puestas a tierra se entienden como un valor "SE0", que se usa para señalizar el fin de un paquete o un reset proveniente del host. Igualmente, hay otra serie de valores codificados para realizar acciones como suspender la actividad en el bus, reanudarla, etc. El chip PDIUSBP11A que lleva la tarjeta y el módulo VHDL "adaptador" (perteneciente al proyecto) se encargan de aislarnos de estas particularidades, de modo que a la salida de este componente tenemos una señal de datos que ya no es diferencial.

En cuanto a las terminaciones de las líneas, hay que decir que en nuestro diseño la línea D+ está conectada a una resistencia de pull-up de 1,5KΩ, como se ve en la siguiente figura. Esto hace que al conectar el dispositivo al host, éste detecte automáticamente su presencia por un cambio en la tensión de la línea, y que se de cuenta de que lo que se ha conectado es un dispositivo "full-speed", ya que la resistencia está conectada a la línea D+. Si estuviese conectada a la línea D-, esto denotaría que el dispositivo es "low-speed". La tarjeta XSV permite configurar el dispositivo como "full-speed" o como "low-speed", mediante un jumper.



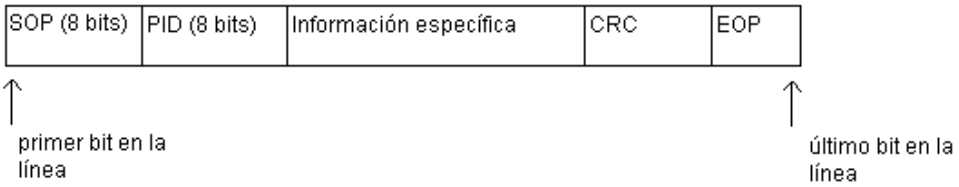
Conexión de un dispositivo full-speed

4.2.2. Nivel físico

La codificación de línea es NRZI. En el código NRZI, los '0' se codifican con un cambio en el valor de la línea, y los '1' hacen que la línea permanezca como estaba. Por ello, como muchos dispositivos sacan el reloj de la transmisión a partir de los cambios en la línea, si se transmitiesen muchos '1' seguidos la línea no cambiaría de valor y se perdería el sincronismo de la comunicación. Para que esto no ocurra, tras seis '1' consecutivos siempre se introduce un '0' en la secuencia de bits, independientemente del bit que venga después de los seis '1'. Este mecanismo se llama "bit stuffing", y hay que tenerlo en cuenta tanto en el receptor como en el transmisor.

La información por el bus se envía en forma de paquetes. Hay muchos tipos de paquetes dependiendo del tipo de transferencia que se esté realizando, pero todos ellos comienzan con un campo "SOP" (Start Of Packet), que no es más que una secuencia definida de 8 bits, y terminan con un "EOP" (End Of Packet), que está formado por dos SE0 y un '1'. Tras el "SOP", viene un campo de 8 bits para identificar el tipo de paquete (PID, Packet Identification), y después la información varía con cada tipo de paquete. Sin embargo, la mayoría de paquetes llevan sus campos protegidos por códigos de redundancia cíclica para evitar errores. Este mecanismo de seguridad también se ha implementado en el dispositivo, tanto para transmisión como para recepción, ya que el controlador lo supervisa inevitablemente. No vamos a entrar ahora en el mecanismo concreto de generación y comprobación del CRC, lo veremos cuando expliquemos el módulo VHDL correspondiente.

El formato general de los paquetes es el que se ve a continuación:



Formato general de los paquetes USB

Vamos a ver los tipos de paquetes más importantes. Los podemos clasificar en paquetes "token", paquetes de datos y asentimientos.

- **Paquetes "token":** Son el primer paquete de cualquier transmisión. Se emplean para indicar el dispositivo destino de la siguiente transmisión, y la dirección de esta transmisión (si es hacia o desde el dispositivo). Son los siguientes:
  - **Setup:** Indica el comienzo de una nueva transferencia.
  - **IN:** Indica que el siguiente paquete (que será de datos) va desde el dispositivo hacia el host.
  - **OUT:** Indica que el siguiente paquete va desde el host hacia el dispositivo.

El formato general de los paquetes token es el siguiente:

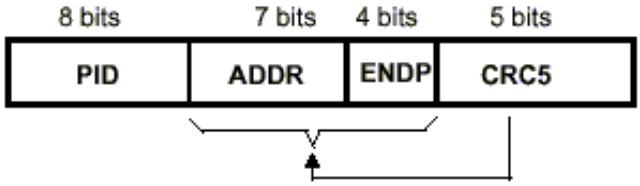


Figure 8-5. Token Format

En la figura faltan los campos SOP y EOP. El campo ADDR indica la dirección del dispositivo, y el campo ENDP (endpoint), el canal de comunicación dentro del dispositivo. El campo CRC5 protege los campos señalados.

- **Paquetes de datos:** Se emplean para transmitir datos de cualquier longitud, en nuestro caso con un máximo de 8 bytes de datos. Hay de dos tipos, **DATA0** y

DATA1. Ambos son exactamente iguales, con la diferencia de que los PIDs son distintos. Esto permite implementar un mecanismo de recuperación de errores que se explicará posteriormente, llamado "data toggle". El formato es el siguiente:

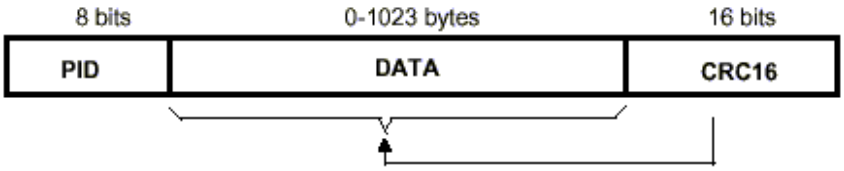


Figure 8-7. Data Packet Format

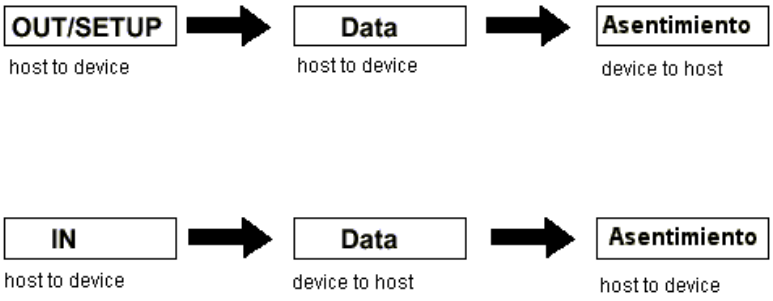
En este caso el CRC es de 16 bits y protege los bytes de datos.

- **Asentimientos:** Los únicos campos que lleva son SOP, PID y EOP. Sirven para asentir o rechazar la información enviada en un paquete de datos anterior. Aunque hay tres tipos, sólo nos interesan dos:
  - **ACK:** Asiente el paquete de datos. Va en la dirección contraria a éste.
  - **NACK:** Rechaza el paquete de datos por estar el dispositivo ocupado procesando el paquete anterior. Sólo lo puede enviar el dispositivo USB, nunca el host, y sirve para indicar a éste que reenvíe el paquete más tarde.

Aparte de estos paquetes, hay otro, con PID "SOF" (Start Of Frame), que se difunde por el bus para señalizar el comienzo de un nuevo marco, y que nuestro dispositivo deberá ignorar.

Estos paquetes no viajan individualmente, sino que se agrupan para formar "transacciones"; una transacción consta de un token, un paquete de datos y un asentimiento. El token indica el destinatario y dirección de la transacción, y el asentimiento asiente o rechaza la transacción. Estos tres paquetes viajan invariablemente en ese orden, y no se puede romper la unidad de una transacción, es decir, una vez que se envíe un token, el siguiente paquete en circular por el bus será el paquete de datos y el siguiente, el asentimiento. Ningún otro dispositivo USB aparte del seleccionado podrá acceder al bus durante la transacción. Además, hay restricciones para el tiempo que puede pasar entre un paquete y otro dentro de una transacción. Si un dispositivo, por ejemplo, no envía un asentimiento (ACK o NACK), quiere decir que el

paquete que le ha llegado estaba corrupto. El PC detecta un "time-out" en el bus (expira un contador sin llegar asentimiento) e inicia la retransmisión.



Transacciones en USB

Por último, estas transacciones se agrupan de distintas formas para formar distintos tipos de transferencias de datos, que son los modos de trasvasar información que tiene USB. Nosotros sólo emplearemos las llamadas "transferencias de control".

4.2.3. Tipos de transferencias

Con los distintos tipos de paquetes se implementan los cuatro distintos tipos de transferencias que soporta USB. Cada uno de ellos se adapta a las distintas necesidades de comunicación que puede tener un dispositivo. Estos tipos son:

- **Transferencias isócronas**
- **Transferencias "Bulk"**
- **Transferencias de interrupción**
- **Transferencias de control**

Veamos cada una de ellas:

4.2.3.1. Transferencias isócronas

Este tipo está pensado para dispositivos que necesiten transmitir datos en tiempo real. Por ejemplo, dispositivos de audio. Los datos se envían con una tasa constante y no están sometidos a control de errores, ya que el reenvío de datos no es factible. Es la forma de conseguir el mayor ancho de banda posible, aunque su implementación puede

ser compleja porque hay que mantener la generación y/o recepción de datos con la velocidad correcta, para evitar llenar o vaciar las colas. Junto con las transferencias de interrupción, son las transferencias con más prioridad en el bus. No se contempla para este proyecto este tipo de transferencia.

4.2.3.2. Transferencias de Interrupción

Están pensadas únicamente para consultar al dispositivo si tiene algo que transmitir, es decir, si tiene una petición de interrupción presente. Por tanto, este tipo de transferencia sólo se da en el sentido dispositivo → host: Cada cierto número de marcos, que se puede configurar, el host envía un paquete **IN**. Si el dispositivo tiene información disponible, la envía con paquetes **DATA**, y el host los asiente con un paquete **ACK**. Si el dispositivo no tiene información, simplemente no hace nada. Por tanto, este tipo de transferencia tiene recuperación de errores. Si hay error, la transmisión se reintenta en la siguiente oportunidad de transmisión.

En el proyecto no se emplea este tipo de transferencia, aunque si fuera necesario que la placa avisase al PC para transmitir algo, se podría utilizar este mecanismo.

4.2.3.3. Transferencias "Bulk"

Pensado para dispositivos que necesiten enviar datos en grandes cantidades, como por ejemplo impresoras. Es el tipo con menos prioridad, de hecho sólo se llevan a cabo si cuando se han hecho todas las transacciones de los otros tipos programadas para un marco, en ese marco queda tiempo disponible. Por eso, al pedir una transferencia de este tipo, no se sabe a priori cuántos marcos va a tardar en completarse. Estas transferencias sí están sometidas a control de errores y mecanismos de reenvío, por lo que hay que implementar, entre otros elementos, temporizadores de expiración.

El mecanismo es como sigue: transferencias en dirección host→ dispositivo: consiste en sucesivas transacciones con token **OUT,** sin ninguna otra señalización o particularidad. Si el receptor recibe un paquete con errores, no emite ningún asentimiento y deja que expire el temporizador del host, tras lo cual éste retransmite el/los paquete de datos y prosigue con la transmisión del resto. Se sabe que ha

terminado la transmisión cuando se recibe un paquete DATA con menos datos de la cuenta o sin datos.

En transferencias dispositivo → host: Igual que el anterior, pero con transacciones de tipo **IN.** Es decir, este tipo de transferencia consiste en sucesivas transacciones de tipo IN o OUT.

Este tipo de transferencia no se emplea en nuestro diseño, aunque sería la más adecuada, pero dado que para transferencias masivas no consigue una mejora sustancial con respecto a las transferencias de control, y en cambio complicaría bastante el diseño hardware del sistema, se ha optado por desecharla.

4.2.3.4. Transferencias de Control

Este tipo de transferencia se usa para configurar todo dispositivo USB, luego cada dispositivo USB tendrá que tener al menos un mecanismo de transferencia de control (control endpoint). Por tanto, dado que tenemos que implementar por fuerza un controlador de este tipo de transferencia, lo vamos a usar también para realizar el trasvase de datos que necesitamos, ahorrándonos el tener que implementar un controlador de transferencias tipo "bulk" adicional.

Las transferencias de control tienen reservado un 10% del ancho de banda que siempre tendrán disponible, aunque si sobra tiempo en un marco puede ocupar más ancho de banda. Como en las transferencias "bulk", un ancho de banda mayor del 10% no está garantizado, pero en las condiciones óptimas (si sólo hay transferencias de control y no hay más dispositivos conectados al bus) puede ocupar hasta el 100% del ancho de banda del bus. Por tanto, en este tipo de transferencias se basa nuestra solución. Pasemos a ver la estructura de las transferencias de control:

Cada transferencia se compone de dos o tres "**fases**":

- **Fase de setup**
- **Fase de datos (opcional)**
- **Fase de Estado**

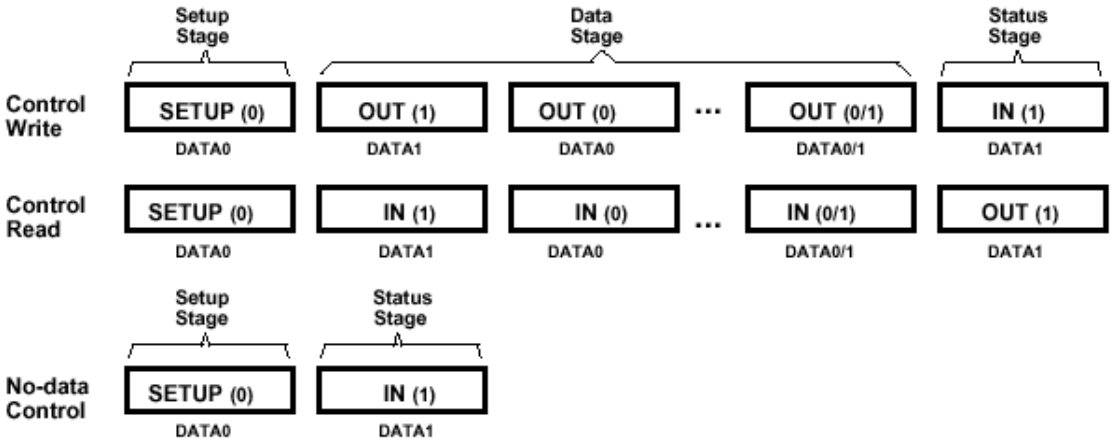
A su vez, cada una de estas fases se compone de una o más transacciones: un paquete "**token"** (**IN, OUT** o **SETUP**), seguido de un paquete de datos (**DATA**), y por último un asentimiento. Las fases de setup y estado constan de una sola transacción, mientras que la de datos puede tener cuantas transacciones sean necesarias, o incluso ninguna.

En la fase de setup, el host envía un paquete **SETUP** que indica el inicio de una transferencia de control. Tras este, envía un paquete **DATA** con 8 bytes de información que contienen la instrucción a cumplir. Esta instrucción puede ser propia de USB para configurar el dispositivo, o una instrucción nuestra como por ejemplo, "escribir estos datos en estas direcciones". Esto es lo que se hace en el proyecto. Así, en una misma transferencia podemos incluir varias órdenes de lectura o escritura, con el fin de optimizar el ancho de banda. Por otro lado, cuando el receptor recibe esta instrucción, sabe en qué dirección viajarán los datos en la fase de datos.

La fase de datos no es obligatoria. Dependiendo de la instrucción, será necesaria o no. En esta fase los paquetes DATA se pueden desplazar en sentido host → dispositivo (irían precedidos por un paquete **OUT** y seguidos del correspondiente asentimiento), o en la dirección dispositivo → host (precedidos por **IN** y seguidos de asentimiento del host). Por ejemplo, si la orden fuese "asignar dirección X al dispositivo" (este sería el mensaje de la fase de setup), no sería necesaria esta fase, se pasaría directamente a la de estado. Si la orden fuese "leer estas direcciones", entonces en la fase de datos se mandarían paquetes **DATA** desde el dispositivo al host con los resultados de las lecturas.

Por último, la fase de estado sirve para cerrar la tranferencia, y consta de una transacción en la dirección contraria a la de la fase de datos. Por ejemplo, si en la fase de datos el token fue **IN**, ahora el token sería **OUT.** El paquete **DATA** de esta última fase viajaría viaja sin datos, tanto en un sentido como en el otro.

En la siguiente figura se intentan reflejar las transferencias de control:



Las dos primeras serían transferencias con las tres fases (stages), y la última sin la fase de datos. Cada recuadro representa una transacción(un token, el indicado en el cuadro, un paquete DATA, y un ACK).

Dado que en este tipo de transferencias se pueden producir pérdidas de sincronización (por ejemplo: el host envía un paquete de datos y el dispositivo lo recibe bien; envía el asentimiento pero éste llega corrupto al host. Éste se cree que ha fallado la transmisión y la reintenta, lo que provoca que el dispositivo acepte dos veces los mismos datos; igualmente, se puede producir la pérdida de un paquete), se implementa un mecanismo de seguridad para mantener al dispositivo sincronizado con el host. Este mecanismo se llama "data toggle", y explica el porqué de tener dos tipos de paquetes de datos, **DATA0** y **DATA1**.

Básicamente consiste en que cuando hay que enviar datos, estos se envían alternativamente en paquetes **DATA0** y **DATA1**, de forma que si se diera el error anterior, el dispositivo sabría que el paquete que le ha llegado es de nuevo el anterior, porque esperaba un **DATA1** y ha llegado un **DATA0**. El paquete de datos que sigue a un token **SETUP** es siempre **DATA0**, y el de la fase de estado es siempre **DATA1**. Con este mecanismo se consigue evitar la pérdida de sincronización, a la vez que se implementa otro mecanismo de reenvío.

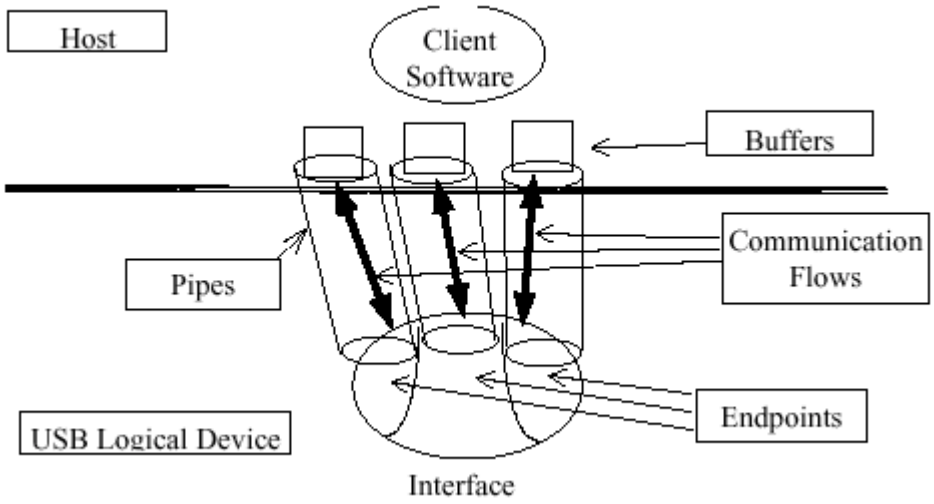
4.2.4. Proceso de Configuración

¿Qué ocurre cuando se conecta un dispositivo al bus USB? El host necesita saber cómo se comunican los drivers del usuario con el dispositivo. Para ello, pide esta información al nivel lógico del dispositivo. Éste debe almacenar una serie de "**descriptores**" que no son más que estructuras de datos que contienen esta información.

Por otro lado, cualquier dispositivo USB puede tener distintas **configuraciones**, es decir, distintos modos de funcionar. Por ejemplo, un dispositivo podría tener una configuración normal, y otra de ahorro de energía para el caso en que el bus no pudiera aportarle la energía suficiente. Un dispositivo no puede tener más de una configuración activa a la vez, y en nuestro caso, el dispositivo sólo tiene una configuración, por lo que no hay que implementar mecanismos de selección de la configuración.

Cada una de estas configuraciones tendrá a su vez uno o más **interfaces**, que son conjuntos de canales (pipes) de comunicación entre el dispositivo y el host, cada uno de los cuales puede implementar cualquier tipo de transferencia. Por ejemplo, un CD-ROM USB tendría dos interfaces: uno de datos, que usaría un canal "bulk", y otro de audio que tendría canales isócronos. Cada interfaz lo maneja por tanto un programa software diferente. Nuestro dispositivo sólo tiene un interfaz en su única configuración, y este interfaz está formado por un canal de transferencias de control.

A su vez, cada uno de estos canales se llaman **"pipes"**, y tienen un extremo en el dispositivo USB (este extremo se llama **"endpoint"**), y el otro disponible para el driver de dispositivo encargado de ese interfaz. Así, el software y el dispositivo se comunican directamente a través del "pipe". La siguiente gráfica pretende ilustrar esto:



Volviendo al proceso de configuración, los descriptores del dispositivo contienen toda la información relativa a las distintas configuraciones, los interfaces de cada una y los canales de cada interfaz. Así, al enchufar el dispositivo al bus, el host detecta una conexión nueva y manda un reset al dispositivo. Tras resetearse, éste responde a la dirección 0, que es la correspondiente al canal de control que es de obligatoria implementación. Este canal emplea únicamente transferencias de control, y por él el host le pide al dispositivo que envíe sus descriptores. Éste los envía, y el software de USB los analiza. Seguidamente, se asigna una dirección distinta de la cero al dispositivo. A partir de ese momento, el dispositivo sólo atenderá a esa dirección y no a cualquier otra. Una vez el software analiza las distintas configuraciones, elige una en función de los requisitos del programa cliente y de la ocupación del ancho de banda del bus, y envía al dispositivo orden de emplear esta configuración. A partir de ese momento, el programa cliente puede comunicarse libremente con el dispositivo en la forma determinada. Si el bus no pudiese soportar ninguna de las configuraciones, el dispositivo quedaría sin configurar y se notificaría al usuario, el cual no podría usar el dispositivo.

En nuestro proyecto, en el hardware del dispositivo se han implementado dos descriptores: el **descriptor del dispositivo** y un **descriptor de configuración.** El primero es único para cada dispositivo, e indica, entre otras cosas, el tipo de dispositivo USB, la identificación del producto, mediante la cual se relaciona al dispositivo con los drivers de usuario adecuados, el número de configuraciones y el número máximo de bytes que puede llevar un paquete de datos, que en nuestro caso es 8.

Como nuestro dispositivo sólo tiene una configuración, sólo tiene un descriptor de configuración, en el que se incluyen los datos sobre el interfaz, el canal de control y las características de energía.

5. Herramientas usadas en el Proyecto

Antes de pasar a describir la solución para los distintos componentes del proyecto, me gustaría explicar qué herramientas (generalmente software) se han empleado para el desarrollo de cada uno de éstos componentes. Vamos a separar entre componentes hardware y software.

5.1. Herramientas para Hardware

Básicamente se han empleado tres grupos de herramientas:

- **Paquete Xilinx Foundation F3.1i.** Con este paquete software se ha realizado la síntesis e implementación del código VHDL que posteriormente se ha bajado a la FPGA Virtex 800. También se ha usado su simulador para realizar las simulaciones de algunos módulos.

- **Paquete de herramientas GXSTOOLS de Xess.** El fabricante de la tarjeta XSV800 nos proporciona estos programas para descargar los ficheros de configuración sobre la FPGA o la CPLD, para comprobar el estado de la tarjeta, o para programar el oscilador de la XSV800, entre otras cosas. Cabe destacar que ésta última herramienta, la de programación del oscilador, también se ha empleado para programar el oscilador que hemos usado, el que se ha implementado en una placa impresa.
- **Accel Tango PCB.** Este editor de PCBs se ha empleado para diseñar el PCB de la placa del oscilador.

5.2. Herramientas para software

Para generar el software se han empleado dos herramientas; una para el driver y otra para la aplicación:

- **Microsoft Windows 98 Driver Development Kit (DDK).** Ésta es la herramienta es necesaria para construir drivers para Windows 98/NT, y por tanto es la que se ha empleado para la generación del driver. Éste ha sido desarrollado a partir de un ejemplo de driver encontrado en la documentación del DDK. El código fuente está escrito en C, pero para que funcione como driver e interactúe con el resto de drivers de USB, debe poder ejecutarse en modo 'kernel', y para ello es necesario compilarlo con esta utilidad. Es de libre distribución y se puede encontrar en la red, en la página indicada al final de este documento.

- **Microsoft Visual C++ 6.0.** Con este paquete se ha desarrollado la aplicación de ejemplo, el programa "USBLink". Éste está escrito en C++ y basado en la MFC (biblioteca de clases de base de Microsoft), por lo que su entendimiento puede ser bastante complicado, pero el código que realmente nos interesa, el de manejo del interfaz USB, es realmente sencillo y será la parte en la que nos centremos cuando expliquemos el programa. Además, esta parte puede ser implementada con C puro. Por lo tanto, lo interesante es el modo de acceder a la placa, y conociendo el funcionamiento, se pueden desarrollar programas con cualquier otra herramienta.

5.3. Otras herramientas

Para el desarrollo del proyecto he empleado muchas otras herramientas, aunque sólo voy a citar aquí las que realmente me han sido de utilidad:

- **SniffUSB 1.0.** Es un filtro para mensajes USB. Captura los paquetes de datos, mensajes del sistema operativo, etc., que vayan dirigidos a un dispositivo USB concreto, y analiza su significado. Es una herramienta útil sobre todo para la

depuración del software (tanto driver como aplicación), aunque también puede ayudar en el desarrollo del hardware. Es de libre distribución y uso. Su referencia se encuentra en la bibliografía.

- **Intel USB system check.** Esta herramienta es útil para determinar si el sistema USB funciona bien en un PC. Analiza tanto el software (pila de drivers) como el hardware (controlador USB) del PC para ver si es capaz de soportar dispositivos USB sin problemas.

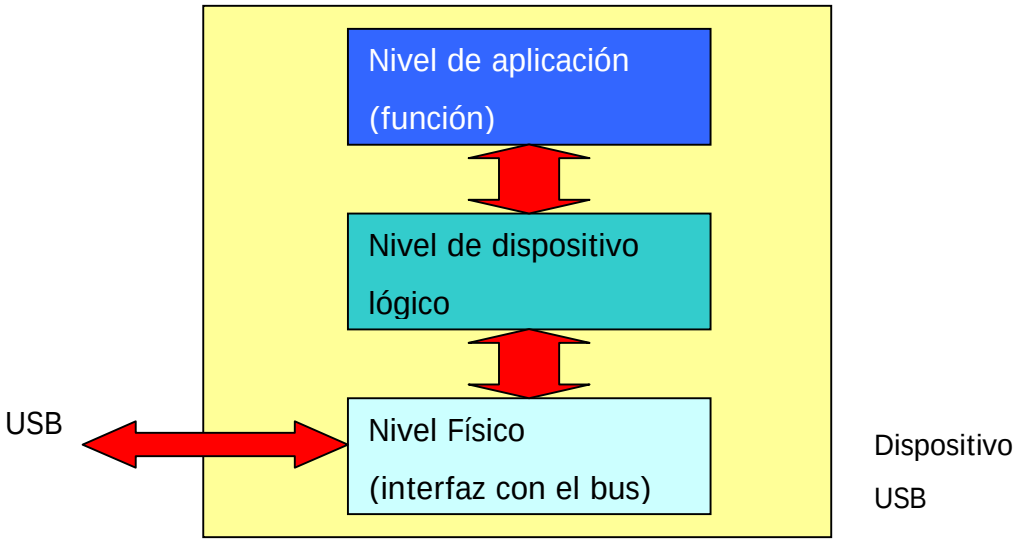
6. Solución Final. Interfaz físico USB en VHDL

A partir de este punto, el presente documento se centra en la explicación de la solución adoptada para resolver el problema expuesto anteriormente, en sus distintas áreas; en este capítulo se expone la parte correspondiente al desarrollo del hardware para implementar sobre una FPGA, y en el siguiente nos centraremos en la explicación de la estructura, funcionamiento y uso del software asociado.

6.1. Introducción a la solución

Ésta no es la primera implementación de un interfaz USB pensado para funcionar sobre un dispositivo de lógica reconfigurable, como una FPGA. No es difícil encontrar algunos ejemplos más navegando por la red. Sin embargo, sí es el primero que adopta una solución distinta a las ya existentes, y esto es debido a la necesidad de simplicidad, de obtener un resultado lo más compacto posible y que empleara el mínimo número de recursos posibles.

Lo que sí tiene en común nuestra solución con las anteriores es la estructura general. Como se ve en la figura, se divide la funcionalidad del interfaz USB en distintas capas.



El nivel físico es el encargado de manejar directamente las líneas del bus, y de aplicar todas las funciones que se han descrito para este nivel (ver apartado 4.2.2), como son la detección de inicio y fin de paquete, el control del bit-stuffing, generación y comprobación de CRCs, y en nuestro caso, de permanecer sincronizados con el host durante una transferencia de control.

El nivel lógico, por tanto, ya puede comunicarse con los drivers del sistema operativo. Su función es la de responder a los comandos que le lleguen desde éste, en un primer momento para realizar la configuración del sistema (leer descriptores, asignar dirección, etc.), y posteriormente, en nuestro caso, para atender las peticiones de lectura y escritura. Es decir, proporciona los medios para realizar un trasvase de datos entre el PC y el nivel de aplicación.

Por último, el nivel de aplicación o de función recoge la funcionalidad del dispositivo, que no tiene nada que ver con USB. Es aquello para lo que está hecho el dispositivo. Por ejemplo, podría ser una impresora. En nuestro caso no existe ya que la tarjeta es de propósito general, puede implementarse sobre ella cualquier función. Por tanto, nuestro objetivo son los dos primeros niveles.

Pues bien, las soluciones existentes hasta ahora lo que hacen es implementar en VHDL u otro lenguaje de descripción de hardware tan sólo el nivel físico, ya que el nivel de dispositivo lógico se presta más bien a ser implementado mediante un microcontrolador, porque su función es ejecutar instrucciones. Por tanto, este nivel lo implementan con un microcontrolador y una ROM en la que se escribe el firmware, junto con los descriptores y otros datos. Dado que nosotros queríamos un diseño compacto, y que no necesitábamos que el dispositivo resultante fuese compatible con todas las normas de la especificación USB sino que funcionara bien, optamos por reducir mucho el tamaño del nivel lógico de forma que lo pudiésemos describir mediante VHDL y así implementarlo sobre la misma FPGA.

Y eso es lo que se ha hecho. La solución final sólo consta del código escrito en VHDL e implementado sobre una FPGA, con lo que es un diseño muy simple. No necesitamos ninguna circuitería adicional, tan solo un transceptor USB, que nos proporcione el medio físico adecuado. En este proyecto se ha usado el PDIUSBP11A de

Philips porque es el que trae la tarjeta XSV800 de Xess, pero también existen otros como el MAX3340E de Maxim, que tiene un funcionamiento similar.

A cambio esta simplicidad, la reducción del nivel lógico hace que se respondan a muy pocos comandos de USB, tan sólo a los que realmente son necesarios. Esto no es una pérdida de funcionalidad, ya que el resto de los comandos simplemente no se usan en nuestra aplicación. Si fuese necesario, de todas formas, que el dispositivo respondiese a comandos adicionales, hacerlo sería tan simple como añadir una (o más) nueva instrucción al nivel lógico de esta solución, que se describirá en un apartado posterior.

Por otro lado, las anteriores implementaciones de un interfaz USB en VHDL trabajaban con hasta cuatro señales de reloj distintas: necesitaban un reloj de 48 MHz para extraer a partir de él la señal del reloj en recepción, mediante un DPLL, luego estaba el propio reloj extraido, otro reloj de 12 MHz para la transmisión y otro adicional de 24 MHz para otras partes del circuito. Nuestro diseño tiene la ventaja de que sólo necesita de una señal de reloj de 12 MHz para el funcionamiento de todas las partes del sistema, tanto en transmisión como en recepción, sin que ello suponga un aumento del número de errores de recepción. Esto es debido a que el sistema se sincroniza con cada nuevo paquete que llega debido a que su primer campo, el SOP (Start of Packet) no es más que una secuencia de sincronización. Por tanto, conseguimos así un importante aumento de la simplicidad del circuito, con la reducción de circuitería que ello conlleva.

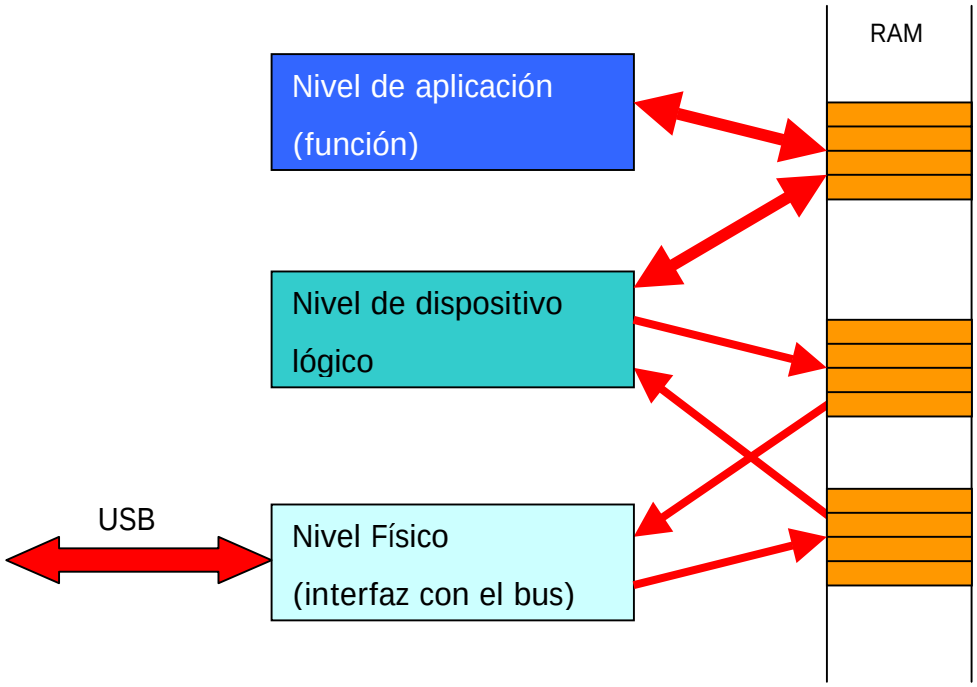
Por último, indicar que al contrario que las anteriores soluciones, que implementan un nivel físico muy genérico y dan la posibilidad de usar hasta cuatro “endpoints” (aunque no implementan los controladores de estos endpoints), en nuestra solución sólo hay un endpoint, que se usa tanto para configuración del dispositivo como para trasvase de datos. Aunque esto resta generalidad, ganamos mucho en simplicidad y podemos atender satisfactoriamente todas nuestras necesidades.

Una vez hecha esta introducción, pasamos a describir la solución nivel a nivel, explicando los distintos módulos que la componen.

6.2. Arquitectura

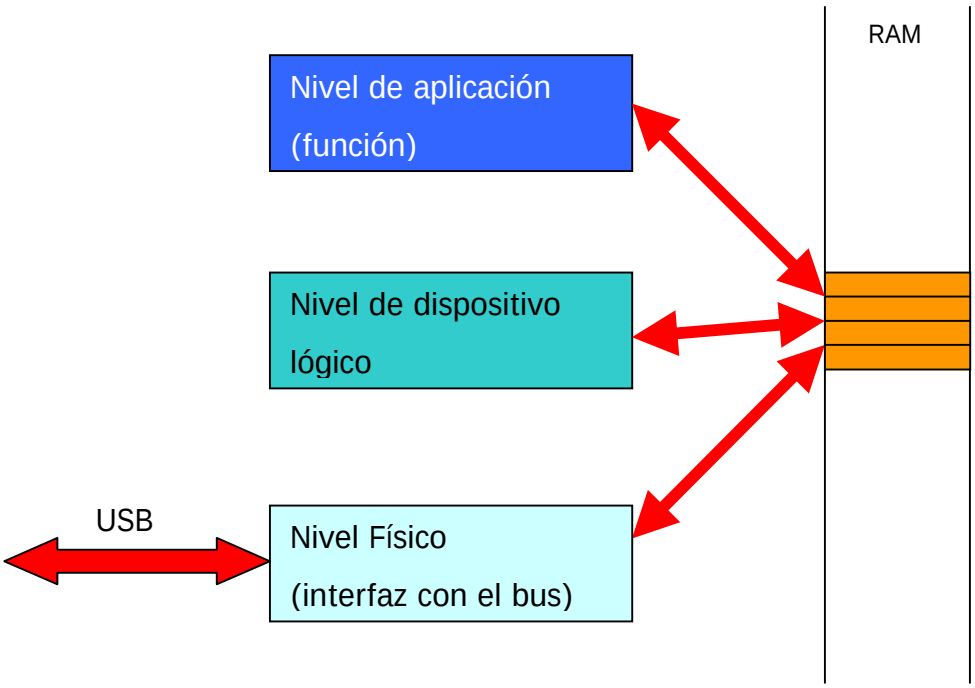
Como hemos visto en el punto anterior, el diseño del interfaz USB se ha estructurado en tres niveles, centrando dos de ellos, el físico y el lógico, el grueso de la circuitería. En el nivel de aplicación se ha implementado una función muy simple a modo de ejemplo, que tan sólo coge datos (procedentes del bus) y los escribe en RAM; también lee datos de la RAM y los prepara para ser enviados al PC.

Hay que señalar que los tres niveles acceden a la RAM que implementa la tarjeta XSV800: el nivel físico guarda en un búfer de 8 posiciones (bytes) las instrucciones que llegan por el bus, para que el nivel lógico las interprete, y también lee de otro búfer de 8 posiciones los datos que hay que enviar al PC. Por su parte, el nivel lógico lee del primer búfer las instrucciones y escribe en el segundo búfer los datos para que el nivel físico los envíe al PC. Además usa un tercer búfer de 8 posiciones para pasarle al nivel de aplicación los datos/direcciones que tiene que leer/escribir.



Se han escogido estos tres búfers por separado para mayor simplicidad de los módulos, aunque los tres niveles podrían compartir un único búfer de 8 posiciones para sus entradas y salidas, y esto funcionaría bien porque cada uno accede a la RAM en distintos instantes y en el orden correcto. Por tanto, sólo nos hace falta un búfer de 8

bytes que a falta de RAM, se podría implementar en VHDL como 8 registros y un decodificador de direcciones.

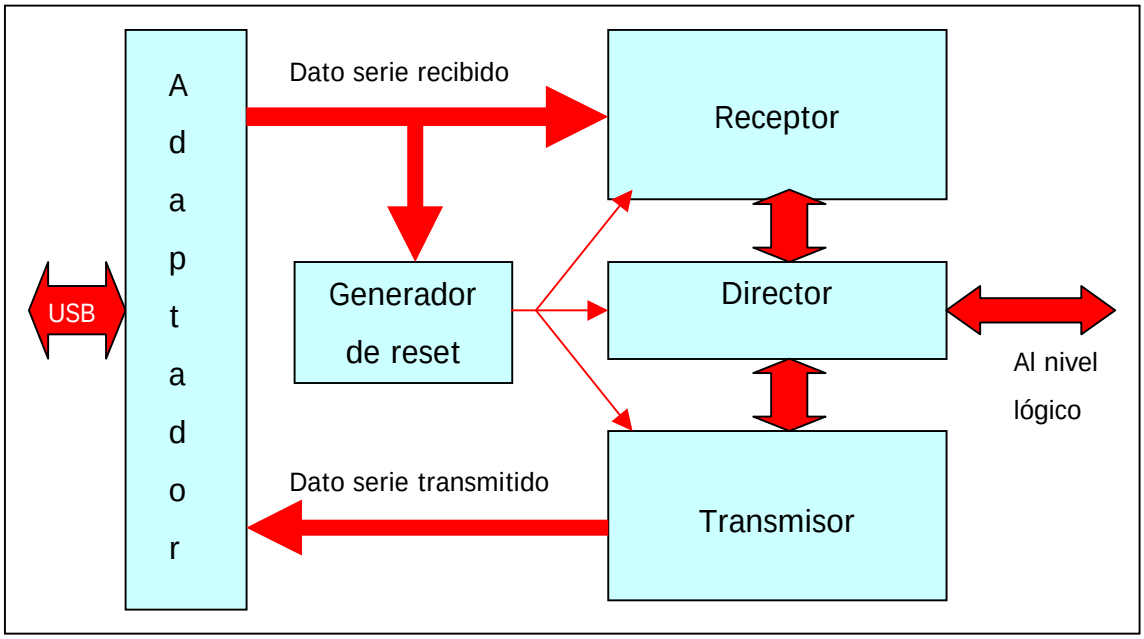


Ya podemos ver por tanto la descripción de cada nivel. Empecemos por el físico, para ir subiendo en la pila de niveles.

6.3. Nivel Físico

Hemos explicado anteriormente la función que debe cumplir este nivel: presentar al nivel lógico los datos que van llegando por el bus, libres de errores, y enviar al PC los datos que provengan de los niveles superiores. Para ello, tiene que implementar todas las funciones definidas en la especificación, como son codificación/decodificación NRZI, bit-stuffing, control de los códigos de redundancia cíclica, etc. Y la más importante: llevar la sincronización de una transferencia de control al ir avanzando por sus diferentes etapas. Asimismo, se realizan otras funciones de carácter general, como la generación de la señal de reset que afecta a todo el circuito.

El diagrama de bloques de este nivel es el siguiente:

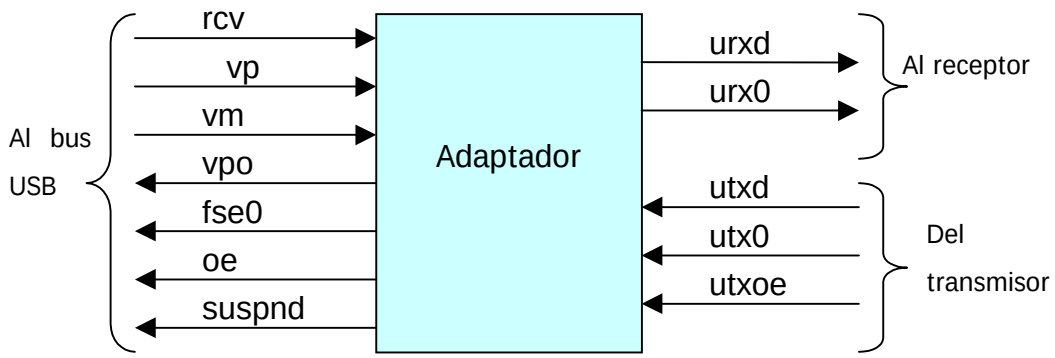


Los bloques que lo componen son los siguientes:

- **Adaptador:** Este módulo es el encargado de traducirnos las señales provenientes del transceptor (y por lo tanto del bus) a niveles lógicos sencillos (‘0’ y ‘1’).
- **Generador de reset:** Este modulo detecta que desde el PC se está dando la orden de resetear todo el circuito y genera una señal de reset. Su salida es el reset que alimenta el resto del circuito (también a los niveles lógico y de aplicación).
- **Receptor:** Se encarga de recibir los datos que llegan por el bus, decodificarlos, etc, y almacenarlos en la memoria.
- **Transmisor:** Se encarga de enviar al bus asentimientos y paquetes de datos con bytes recogidos de memoria. Realiza las operaciones de codificación de línea, generación de CRCs y bit-stuffing.
- **Director:** Es el encargado de controlar al transmisor y al receptor, y de gestionar las transferencias de datos (lleva la secuencia de fases en una transferencia de control). Interactúa con el nivel lógico.

Vamos a ver cada uno de ellos:

6.3.1. Adaptador



Este componente está descrito en el archivo *adaptador.vhd* que se entrega con la presente documentación. Las líneas de la izquierda representan el interfaz con el transceptor. Los nombres se corresponden con las patillas del transceptor PDIUSBP11A a la que se tiene que conectar cada una, para mayor simplicidad. Por lo tanto, estas líneas provienen directamente de las líneas del bus, y son el único patillaje externo que necesita nuestro diseño, aunque en el proyecto también se han usado las líneas que conectan la Virtex con dos displays de 7 segmentos. Estos displays son innecesarios y se han puesto en el diseño para ver si la placa funciona.

La función que realiza este módulo es, como ya se ha dicho, convertir los valores del bus a valores lógicos que podamos usar. Es un bloque puramente combinacional. En recepción, la tabla de conversión es la siguiente:

entrada		**salida**		
vp	**vm**	**urxd**	**urx0**	**significado**
‘1’	‘0’	‘1’	‘0’	“1”
‘0’	‘1’	‘0’	‘0’	“0”
‘0’	‘0’	‘0’	‘1’	“SE0”
Otros		-	‘0’	-

Por tanto, nuestro dato serie lo tenemos en la línea ***urxd***, y la línea ***urx0*** nos señaliza el bit “SE0”, que se emplea para señalizar el fin de un paquete o el reset del

dispositivo. Con estas dos líneas podemos recibir toda la información que viaja por el bus. La línea de entrada desde el bus *vp* representa a la línea del bus D+, y *vm* representa a D-; la línea *rcv* es la entrada diferencial, y no se usa en nuestro diseño, por lo que aquí no se hace nada con ella.

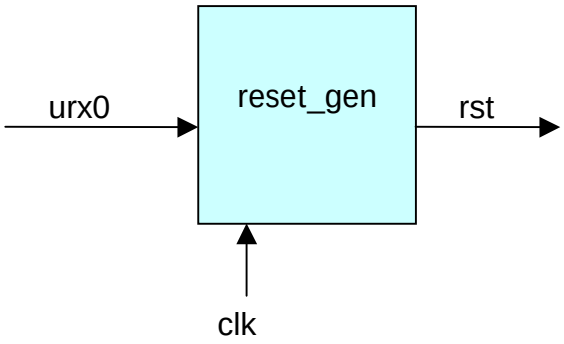
En transmisión, no se hace prácticamente nada: la línea *utxd* se conecta a *vpo* (es el dato a transmitir, '0' o '1'), y *utx0* se conecta a *fse0* (cuando está a '1' señaliza "SE0"). La línea *utxoe* habilita la transmisión cuando está a '1', hay que conectarla a *oe* pero ésta es activa a nivel bajo, por lo que se hace

```
oe <= not(utxoe);
```

Estas tres líneas provienen directamente del transmisor. Por último, la señal *suspnd* hay que ponerla a '0'.

Si en vez del transceptor de Philips se usara el de Maxim, este módulo sería lo único que habría que tocar de todo el circuito para adaptar la lógica a la del nuevo transceptor. Éstos suelen tener varios modos de funcionamiento, por lo que probablemente se podría encontrar uno que fuera compatible con esta lógica y ni siquiera habría que tocar esto.

6.3.2. Generador de reset



Este componente está implementado en el fichero *reset_gen.vhd*. Su función es la de generar la señal de reset para todo el circuito, y lo hace a partir de la línea *urx0*. La especificación indica que un dispositivo debe ser reseteado cuando vea en el bus la señalización "SE0" durante 2,5 μs. de forma continuada. Por lo tanto, nuestro dispositivo genera un pulso en la señal *rst* cuando la línea *urx0* está a '1' durante 2,5 μs, es decir, durante 30 ciclos de reloj (de 12 MHz) consecutivos.

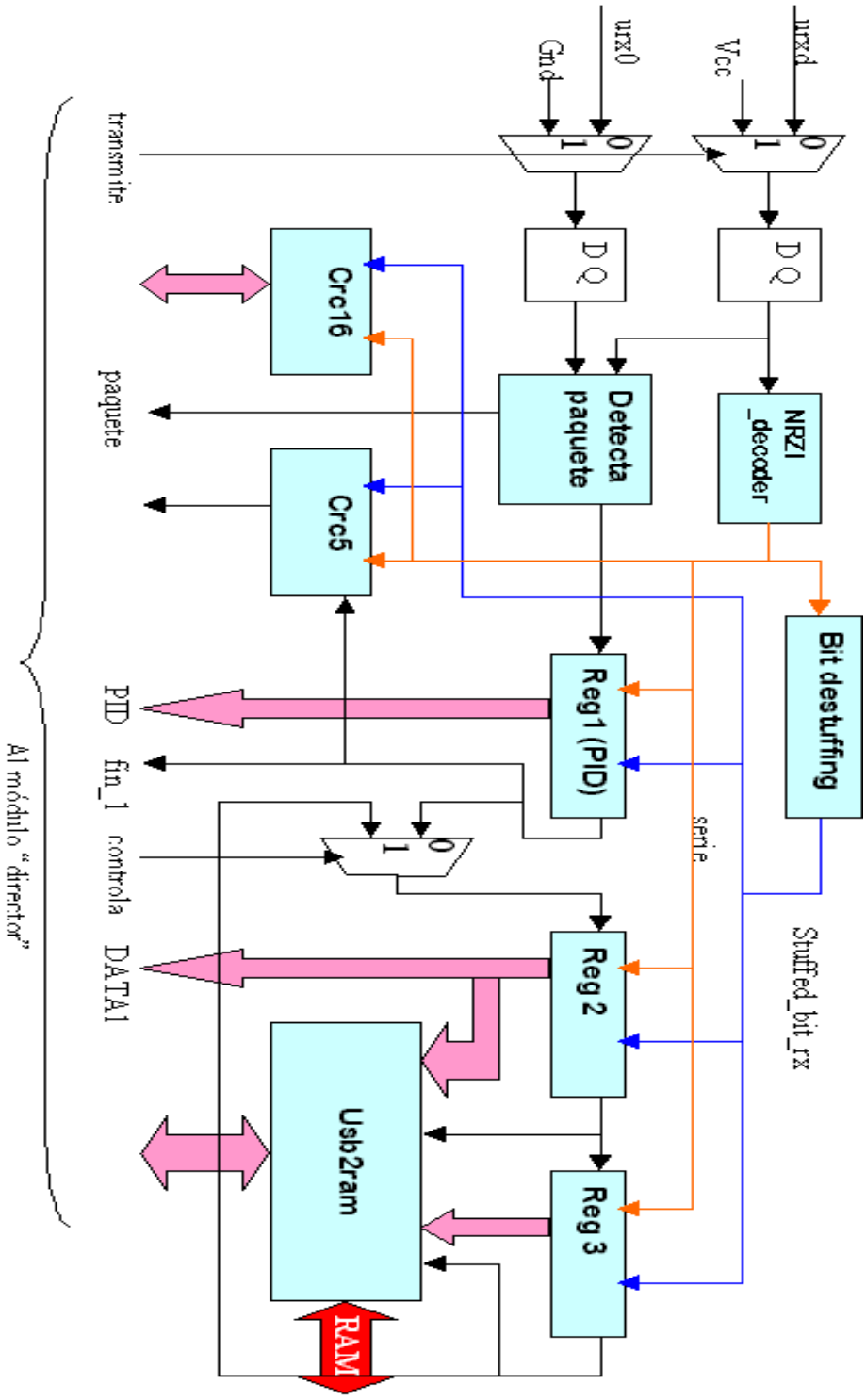
Esto se hace mediante un simple contador. Cuando *urx0* vale '0' o pasa a valer '0', inmediatamente el contador se pone a cero para iniciar una nueva cuenta. Si *urx0* vale '1', el contador se incrementa. Si alcanza el valor 30, la señal *rst* se pone a '1', y así permanece mientras la señal *urx0* no pase a '0'.

Tanto el ancho del contador como el valor límite de la cuenta son datos "generic", de forma que el usuario puede establecer su valor al instanciar el componente en el módulo *USBtop*. En nuestro caso el ancho es 6 bits, y el límite, 30, de forma que se señaliza el reset a los 2,5 μs. Si cambiase la frecuencia del reloj, habría que cambiar estos valores.

Por último, señalar que este componente tiene una entrada adicional, que no se ha recogido aquí. Se llama *rst_in*, y es un reset para este módulo. Es meramente formal y no sirve para nada, por lo que al instanciar este componente hay que tener cuidado de poner esta línea a tierra.

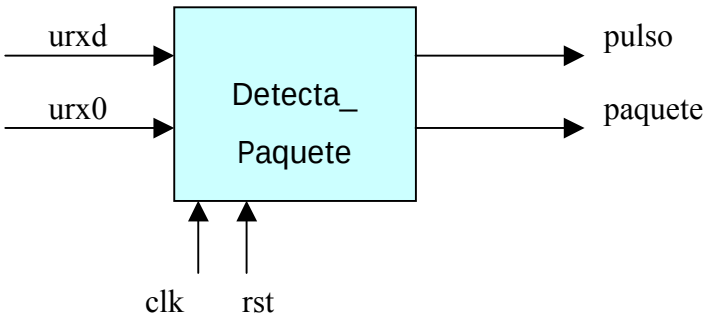
6.3.3. Receptor

Este módulo constituye el corazón del diseño, y no está implementado en un fichero específico sino que sus componentes están directamente instanciados en el módulo *USBtop*, en *usbtop.vhd*. El esquema es el siguiente (no hemos colocado las señales de reloj y reset, pero estas llegan a todos los módulos):



Para explicar su funcionamiento global, vamos a ver cada uno de sus componentes por separado (los bloques azules del esquema). Una vez los hayamos visto, será más fácil entender el esquema del receptor.

6.3.3.1. Módulo Detecta_Paquete



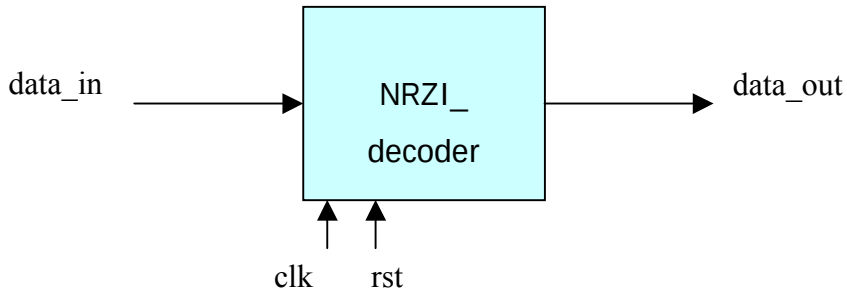
Este módulo se implementa en el archivo *detecta_paquete.vhd*. Su función es señalizar al resto del receptor el inicio y el fin de un paquete. Señaliza todos los paquetes que ve, incluso los que son para otras direcciones y los de dispositivos low-speed. El *director* se encarga de descartar los paquetes que no son válidos. Toma como entrada las señales *urxd* (datos) y *urx0* (SE0), provenientes del bus. Su funcionamiento es el siguiente:

Es una máquina de estados que cuando detecta una secuencia SOP (Start Of Packet) correcta (ésta secuencia es "01010100") pone la señal *pulso* a '1' durante un ciclo de reloj, y pone *paquete* a '1'. Cuando detecta el fin del paquete (EOP, "End Of Packet", se señaliza con urx0 a '1'), desactiva la señal *paquete*.

Por lo tanto, la señal *pulso* es un pulso al inicio del paquete, que coincide exactamente con el primer bit de dato útil en la línea *urxd*, y sirve para indicar al primer registro de desplazamiento que empiece a recoger datos. La señal paquete está activa durante toda la duración del paquete, y no se usa en el módulo receptor, sino en el *director*.

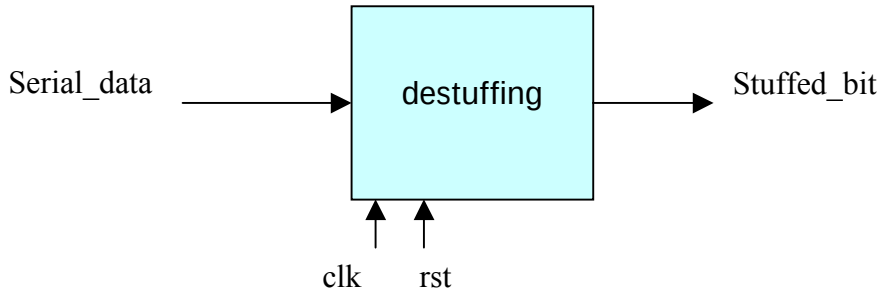
NOTA: En reposo, la línea *urxd* está a '1', es decir, el bus señaliza un 1 lógico en reposo.

6.3.3.2. NRZI_decoder



Este módulo se implementa en el fichero *nrzi_decoder.vhd*. Su función es realizar la decodificación de línea. Hasta llegar aquí, los datos viajan por el bus con codificación NRZI, es decir, los '0' se representan por un cambio en el valor de la línea, y los '1' no alteran este valor. Este módulo realiza el proceso inverso. La señal de entrada, *data_in*, es la señal codificada y proviene del módulo *adaptador* (tras pasar por un biestable que lo que hace es estabilizar el valor del bus en cada ciclo, ver esquema del receptor), y a la salida tenemos la señal *data_out*, que representa el valor lógico correspondiente al valor del bus en ese mismo instante. Es importante resaltar que para realizar la decodificación no se produce ningún retraso en la línea. Por lo tanto, esta señal de salida es la que alimenta el resto del receptor, como se ve en el esquema de la página 44. Se corresponde con la red señalada en naranja (alimenta a los registros de desplazamiento, al detector del bit-stuffing, y a los módulos de comprobación de CRC).

6.3.3.3. Módulo destuffing

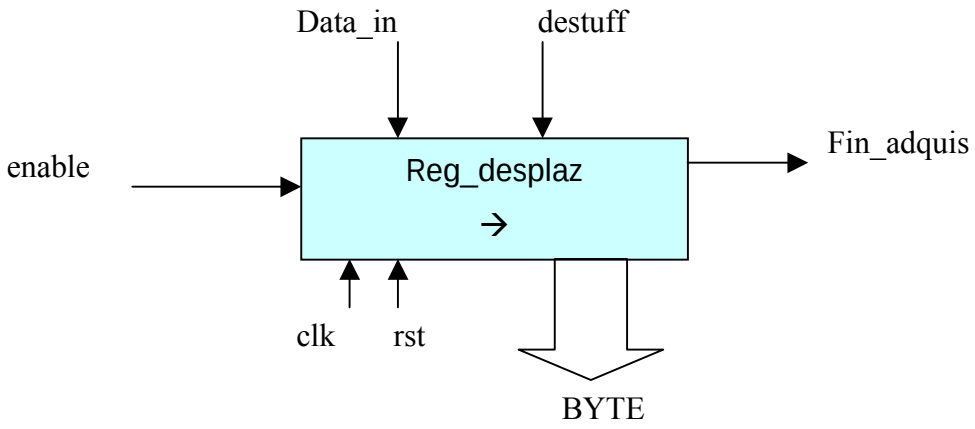


Este componente se implementa en el fichero *destuffing.vhd*. Su función es señalizar los bits de stuffing para que tanto los registros de desplazamiento que vienen a

continuación como los módulos de cálculo de CRC los tengan en cuenta y los eliminen convenientemente. Toma como entrada la línea serie procedente del módulo anterior (la línea de datos), y la salida es una señal que se pone a '1' para indicar que el bit que hay en la línea en ese periodo es de stuffing. Esta señal alimenta, por tanto, a los tres registros de desplazamiento y a los módulos de comprobación del CRC (red azul en el esquema del receptor).

Recordar que el bit de stuffing era un bit a '0' que se introducía en la línea tras seis '1' consecutivos, para evitar perder la información de sincronismo. Por lo tanto, este módulo cuenta el número de '1' consecutivos en la línea, y si llega a 6, el siguiente bit es de stuffing. Es un simple contador.

6.3.3.4. Registro de desplazamiento

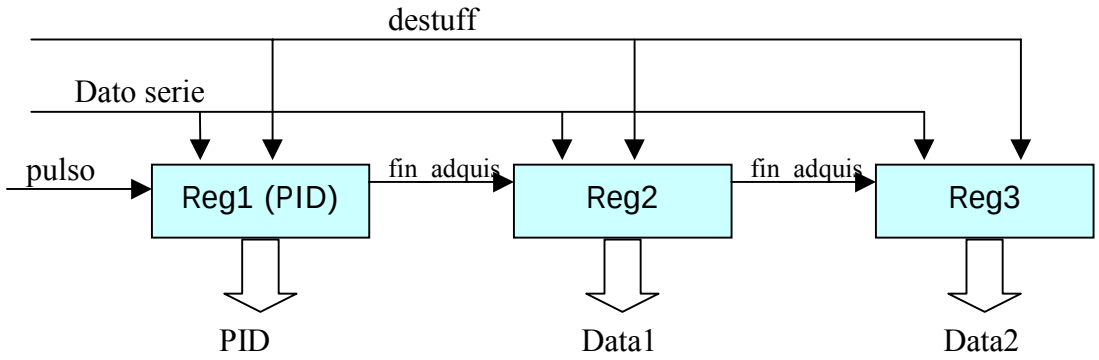


Este componente se implementa en el fichero *Reg_desplaz.vhd*. Es un registro de desplazamiento que recoge 8 bits de la línea serie (*data_in* está conectada a la salida del decodificador de línea, red naranja, para los tres registros) al activarse una señal (*enable*). El primer bit recogido es el que está en la línea cuando *enable* se activa. Esta señal deberá ser un pulso, o al menos estar a '0' antes de 7 ciclos. Cuando se recogen los 8 bits, se produce un pulso en la línea *fin_adquis*. El byte recogido se saca en paralelo por BYTE hasta que se recojan nuevos datos. Cuando se activa la línea *destuff*, el bit actual es de stuffing, por lo que ese bit ('0') se elimina de la secuencia. Esta línea *destuff* está lógicamente conectada a la salida del módulo *destuffing* (red azul en el esquema del receptor).

El objetivo de la línea *fin_adquis* es poder poner varios de estos registros en cascada: se activa justo en el bit siguiente al último que ha recogido, por lo que sirve de enable para el siguiente registro. Así, por ejemplo, si ponemos tres registros en cascada, recogemos 24 bits de datos consecutivos. En el receptor se implementan tres de estos registros, con dos posibles configuraciones según una línea de control (*controla*) procedente del módulo director. Esto se explicará más tarde.

La estructura interna de este registro es el propio registro de desplazamiento, controlado por una circuitería que lo activa o desactiva. Ésta se basa en un contador que se habilita con la señal *enable*, y que cuenta hasta 7, deteniendo el registro justo al recoger 8 bits. La necesidad de tener en cuenta los bits de stuffing complica un poco la circuitería: si se activa la señal *destuff*, se congelan el registro y el contador durante ese ciclo. Hay que tener especial cuidado con los bits de stuffing justo al final de los 8 bits. Para mayor información, véase el fichero *Reg_desplaz.vhd*.

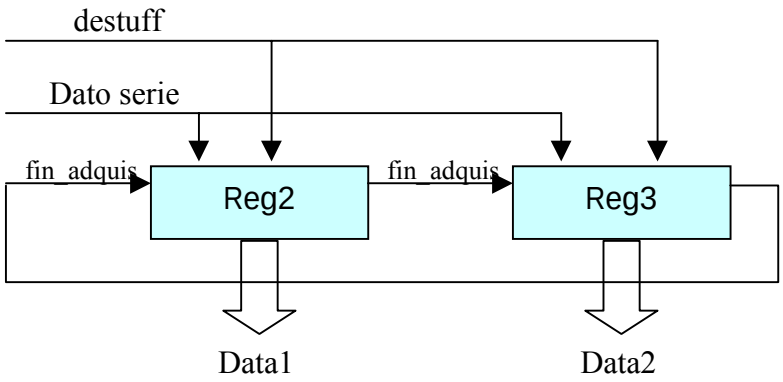
Como se ha mencionado anteriormente, el receptor tiene tres registros, que se pueden configurar de dos maneras según la línea *controla*. Si ésta vale '0', el multiplexor deja pasar la señal *fin_adquis* del primer registro como enable del segundo (ver esquema del receptor), por lo que la topología resultante es de tres registros en cascada:



Esta topología se usa para recoger los paquetes "tokens" y los asentimientos. Cuando la línea pulso se activa, el primer registro empieza a recoger bits, y recoge los 8 correspondientes al PID del paquete, por lo que podemos identificar el tipo de paquete. Este PID va directamente al módulo *director*, que se encarga de interpretarlo. Cuando termina el primer registro, empieza el segundo, que recoge los 8 siguientes bits. Si el

paquete es un token, los 7 bits de la derecha corresponden a la dirección del dispositivo que debe recoger la transacción, por lo que el *director* debe analizar si es nuestra dirección. Cuando este registro termina, el tercero empieza a recoger otros 8 bytes, aunque en este caso ya no nos interesa esa información, por lo que no se mira (Si es un token, ahí está recogido el "endpoint" objetivo, que no nos interesa ya que sólo hay uno, y el crc del paquete, que no nos interesa porque el módulo *crc5* lo analiza por su cuenta; si el paquete es un asentimiento este campo no contiene información válida).

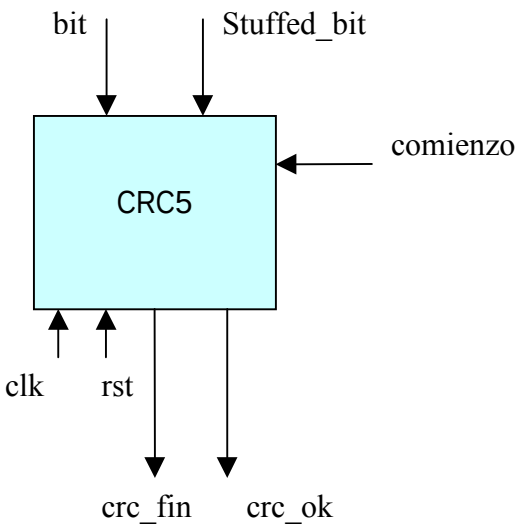
Si el *director* detecta que el PID corresponde a un paquete de datos, y quiere recoger estos datos, entonces pone la línea *controla* a '1', por lo que cambia la configuración de los registros, pasando el primer registro a quedar aislado de los otros dos, que quedan como sigue:



Por lo tanto, ambos registros recogen un byte de datos alternativamente, sin ninguna pérdida. Una vez uno de ellos acaba de recoger un byte, el componente que vaya a procesar esta información (*usb2RAM*) dispone de al menos 8 ciclos de reloj para hacerlo, que es el tiempo que el otro registro estará funcionando antes de pasar el testigo de nuevo al primero. Este tiempo es más que suficiente para nuestro propósito, que es almacenar esta información en memoria.

Pero en este caso, ¿cómo se da el impulso inicial para que uno de los dos registros empiece a funcionar? Cuando se cambió a esta configuración, el registro *reg2* ya se encontraba en funcionamiento, recogiendo el primer byte, ya que se cambia cuando se acaba de recoger un PID y se detecta que es de datos. Por lo tanto, es el registro *reg1* el que da el primer impulso.

6.3.3.5. CRC5



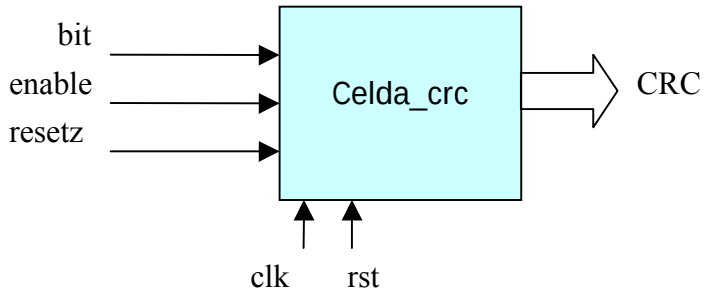
Este componente se implementa en el fichero *crc5.vhd*. Su función es comprobar el código de redundancia cíclica de los paquetes "token" que lleguen al receptor. Para obtener una información completa acerca de los CRC en USB, consultar la especificación de USB. La idea es la siguiente: cuando hay que generar un CRC, a partir los datos a proteger se van realizando una serie de operaciones (básicamente XORs y desplazamientos sobre un polinomio inicial), de forma que el polinomio resultante se invierte y se envía detrás de los datos. En recepción, sólo hay que aplicar el mismo algoritmo a los datos que van llegando, incluido el campo con el CRC, de forma que al final de éste, si la transferencia ha sido correcta, nos queda un polinomio fijo. Con sólo ver si el resultado es este polinomio, sabemos si los datos se han transmitido bien.

En nuestro caso, sólo necesitamos la comprobación, no generar un CRC. Los datos a proteger son los 11 bits que vienen inmediatamente después del PID, y que son la dirección del dispositivo y el endpoint destino. Como el CRC son 5 bits, la secuencia total que hay que chequear es de 16 bits. Este módulo, cuando se activa la línea *comienzo*, empieza a realizar la operación durante 16 ciclos (sin contar los ciclos de los bits de stuffing), y tras estos 16 ciclos comprueba si el polinomio restante es "01100", que es el valor residual que debe quedar. Si la operación es correcta, pone las señales *crc_fin* y *crc_ok* a '1' durante un ciclo; si no lo es, sólo activa la línea *crc_fin*, aunque en nuestro caso esta señal no es usada por *director* porque éste está perfectamente sincronizado con este módulo.

Para un funcionamiento correcto, la señal *comienzo* debe ser el *fin_adquis* del primer registro de desplazamiento, ya que los datos de interés vienen justo a continuación del PID (ver página 22, formato de un token). La señal *bit* es el dato proveniente del decodificador, y *stuffed_bit* viene de *destuffing*.

Internamente, este componente está constituido por un contador que cuenta los 16 bits, y por una celda genérica de cálculo de CRC, *celda_crc*, que es la que realiza el cálculo tanto en este módulo como en el CRC16 y que describiremos a continuación. El resto es circuitería de puesta en marcha y de paro, y para tener en cuenta los bits de stuffing. De nuevo, se soluciona bien el problema de los bits de stuffing al final del proceso.

Celda_crc



Este bloque está implementado en el fichero *celda_crc.vhd*. Su función es el cálculo de códigos de redundancia cíclica, y está implementado mediante generics, de forma que se emplea tanto para calcular CRCs de 5 bits de ancho (en *crc5*), como de 16 bits de ancho (en *crc16*). Como en ambos casos se necesita un polinomio generador distinto, éste también es un generic (el valor decimal del polinomio, internamente se realiza la conversión). Su funcionamiento es el siguiente: sólo calcula polinonios mientras la línea *enable* está a '1'. Cuando esto ocurre, según el valor que haya en la línea *bit* (es el dato serie procedente del bus), realiza una operación u otra según el algoritmo de cálculo. La línea *resetz*, si se activa, devuelve el registro interno a su estado inicial, con lo que el bloque queda listo para un nuevo cálculo. En cualquier momento, podemos obtener el valor de este registro, que es el resultado de la operación, en el bus *crc*.

El algoritmo, tal como viene en la especificación, es el siguiente:

1. Se carga el registro con todos los bits a '1'. (el registro tiene 5 bits si es para crc5 o 16 para crc16).
2. Para cada bit recibido, se hace un XOR entre este bit y el MSB del registro.
3. El registro se desplaza una posición a la izquierda y el LSB se rellena con un '0'.
4. Si el resultado del XOR es '1', se hace un XOR entre el registro y el polinomio generador, y el resultado se vuelve a guardar en el registro; si el resultado del XOR del paso 2 es '0', no se hace nada.
5. Si no se ha terminado, se vuelve al paso 2.
6. Para la generación de CRC, en este punto se niega el registro bit a bit y se envía al bus, primero el MSB (esto es al revés que en el resto de los campos, donde primero se envía el LSB). Si es en recepción, si ya hemos recibido el último bit del CRC, en el registro debe quedar el polinomio residual.

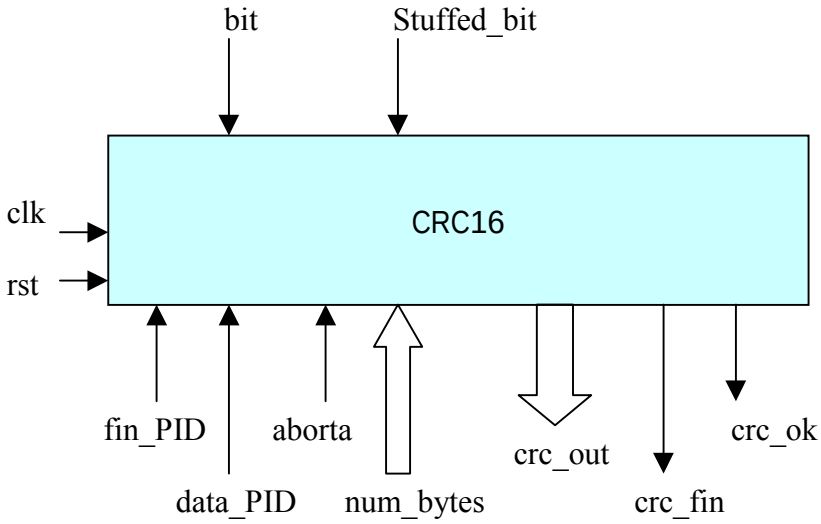
Los polinomios generadores son:

- Crc5: "00101" (decimal 5)
- Crc16: "1000000000000101" (decimal 32773)

Los polinomios residuales son:

- Crc5: "01100"
- Crc16: "1000000000001101"

6.3.3.6. Módulo CRC16



Este módulo se implementa en el fichero *crc16.vhd*. Lo primero es indicar que este componente se comparte por el receptor (que lo usa para comprobar crc's de cualquier numero de bytes) y por el emisor, que lo usa para generar crc´s. Esto conlleva que se tenga que implementar una lógica (en el módulo *usbtop*) para tomar el control del módulo. Esto se hace mediante un multiplexor controlado por la línea *transmite*, proveniente de *director*. Si *transmite*='0', estamos en recepción, y por lo tanto el receptor controla el módulo (el mutiplexor selecciona las líneas *bit*, *stuffed_bit*, *fin_PID* y *data_PID* provenientes del receptor), mientras que si *transmite*='1', entonces estamos en fase de transmisión y estas líneas se toman desde el transmisor.

Por otro lado, las líneas *aborta* y *crc_out* solo las usa el transmisor, por lo que no las veremos ahora. El funcionamiento en recepción es idéntico al de *crc5*, salvo que ahora trabajamos con anchos de 16 bits. El módulo se emplea para comprobar el crc de los paquetes de datos, que pueden tener desde 0 a 8 bytes de datos. Las diferencias con crc5 son las siguientes:

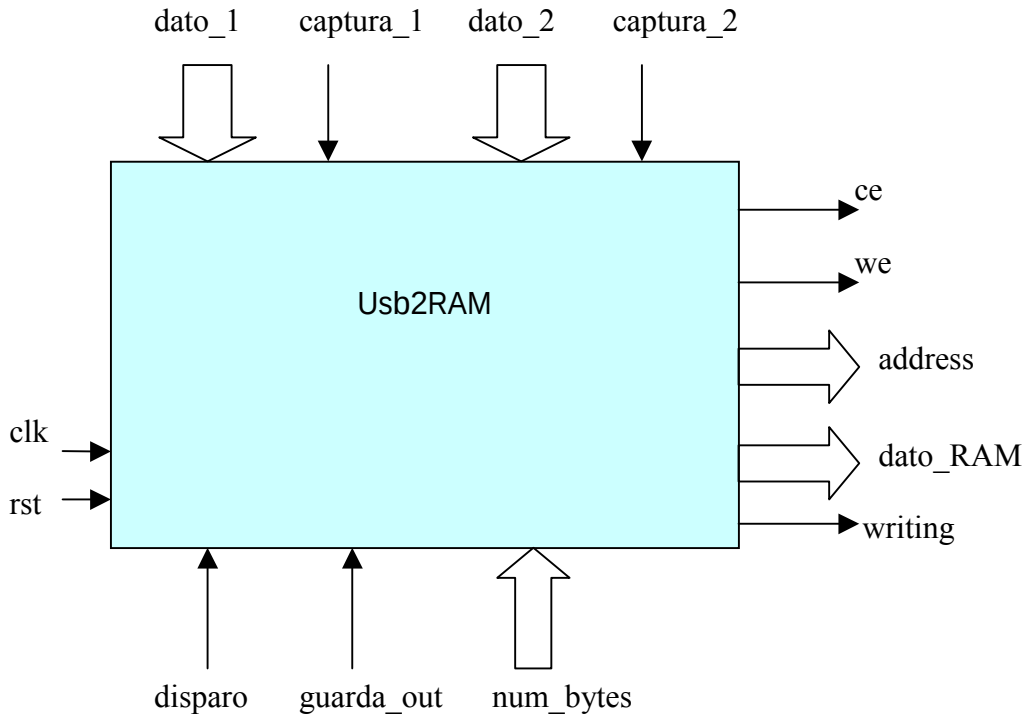
- El proceso se dispara ahora mediante las líneas *fin_PID* y *data_PID*. La primera va conectada a la línea *fin_adquis* del primer registro de desplazamiento, y la segunda, a la línea *controla*, que seleccionaba entre modo de recepción de tokens y modo de datos (ver esquema del receptor). Para disparar el proceso, se tiene que

poner *data_PID* a '1' justo un ciclo después de que lo haya hecho". Esto pasa justo cuando llega un paquete de datos que queremos recoger.

- El número de bits que hay que comprobar es variable. La señal de entrada *num_bytes* indica el número de bytes a comprobar. A estos se le suman internamente los 16 bits del crc. Por lo tanto, se pueden calcular crc's de desde 0 bytes hasta 8 bytes (tamaño máximo de un paquete).

- En este caso sí se usan las líneas *crc_fin* y *crc_ok*; si ambas se activan a la vez, el proceso ha terminado y los datos son correctos. Si *crc_ok* no se activa, ha habido errores en la transmisión.

6.3.3.7. Módulo usb2RAM

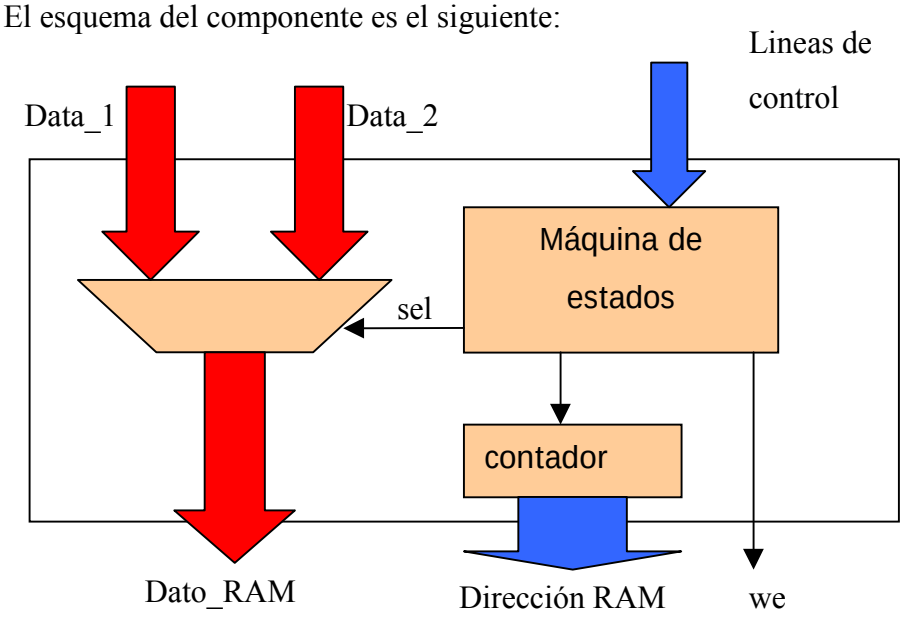


Este módulo se implementa en el fichero *usb2ram.vhd*. Su función consiste en guardar en memoria los bytes de datos de un paquete de datos. Esto se hace independientemente de que los datos sean correctos o no, posteriormente se comprueba

su CRC y se decide si los datos almacenados en memoria son válidos o no. Todo esto lo gestiona el módulo *director*. En el esquema anterior, las entradas del lado superior provienen del propio receptor, en concreto de los registros de desplazamiento *reg2* y *reg3* (*dato_1* y *dato_2* son los bytes recogidos, y *captura_1* y *captura_2* son las respectivas señales *fin_adquis*); las señales del lado inferior provienen de *director*, y las de la derecha son las líneas a conectar con la RAM (Chip Select, Write Enable, bus de datos y bus de direcciones); la señal *writing* está activa cuando los datos en los buses de la RAM son válidos. Se usa para adjudicar el control de la RAM a este módulo. Esto se hace en el componente *usbtop*, donde se regula la compartición de la RAM entre todos los módulos que la usan.

El módulo funciona de dos maneras: si se activa la línea *disparo*, entonces *num_bytes*, que indica el número de bytes a recoger, está puesta a 8. Se recogen los ocho bytes que lleguen a los registros (recordemos que éstos recogen bytes alternativamente) y se almacenan en la RAM, en las posiciones de la 0 a la 7. Estos bytes recogidos son los que iban en el paquete **DATA0** que sigue a un **SETUP**, y por lo tanto constituyen una nueva instrucción que los niveles superiores tendrán que ejecutar. Si por el contrario se activa la línea *guarda_out*, entonces *num_bytes* indica el número de bytes a recoger (de 0 a 8). Se recogen estos bytes y se guardan en otro búfer distinto, de las posiciones 16 a 23 de la RAM. Esto es porque ahora los bytes recogidos son directamente datos de la aplicación, y es ésta la que los debe gestionar.

El esquema del componente es el siguiente:

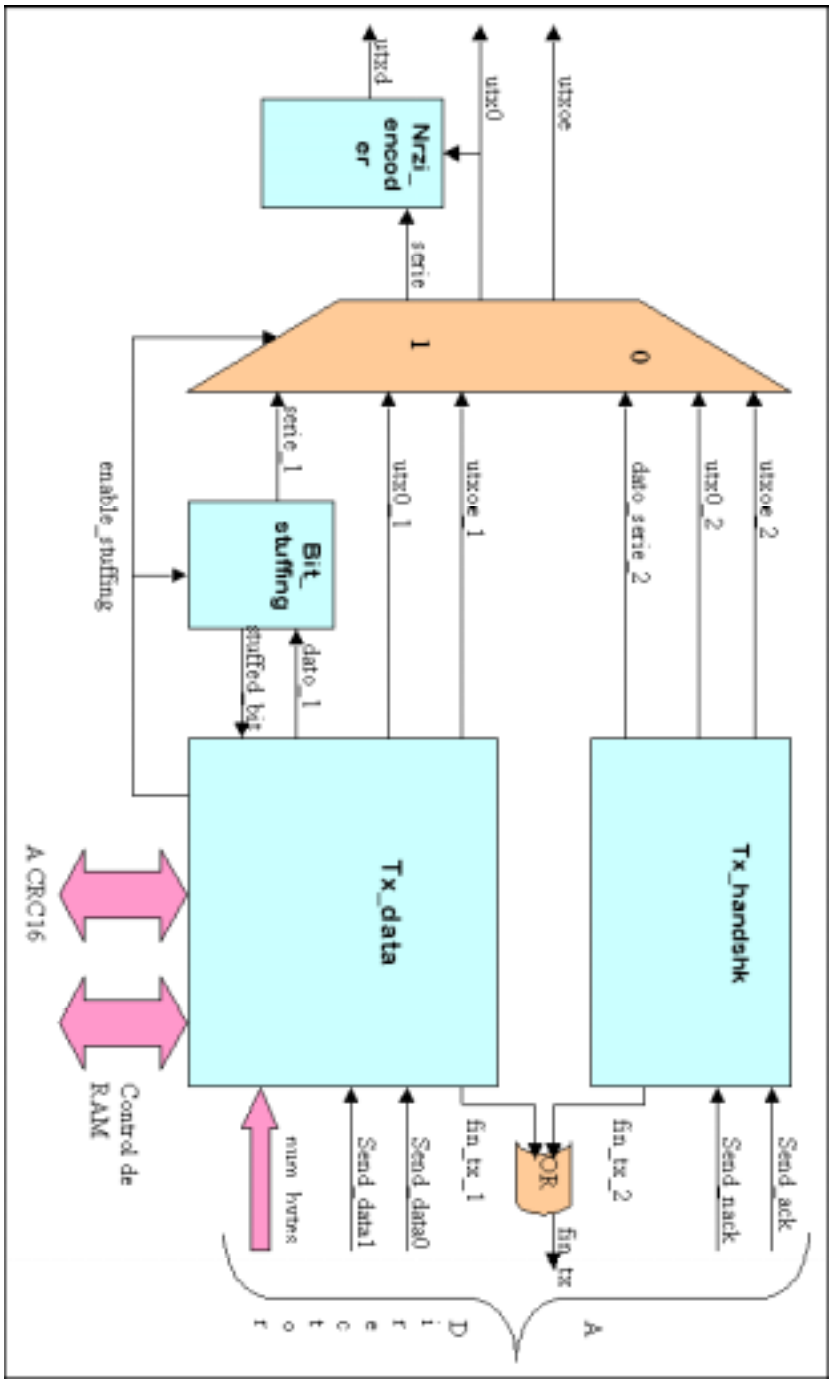


El funcionamiento es el siguiente: al iniciarse el proceso, la máquina de estados espera a que *reg2* termine de recoger un byte; esto se indica cuando *captura_1* se pone a '1'. Entonces ese byte se puede almacenar, por lo que el multiplexor lo selecciona y lo envía al bus de datos de la RAM. Se realiza la escritura. Mientras, *reg3* está recogiendo otro byte, por lo que la máquina espera a que termine, el multiplexor selecciona el nuevo dato (*dato_2*) y se vuelve a repetir el proceso, tantas veces como bytes haya que recoger. El contador cuenta de 0 a 7, y el resto de las líneas de dirección las controla la máquina de estados para seleccionar un búffer o el otro.

Con este componente se ha acabado de exponer el receptor; sólo queda aclarar un pequeño detalle: si se observa de nuevo el esquema del receptor (pag. 42), se verá que a la entrada hay dos biestables; se usan para muestrear las señales de entrada del bus, *urxd* y *urx0*, y así mantenerlas estables durante todo el ciclo del reloj. Las salidas de estos biestables son las que se usan en el resto del receptor. Pero a la entrada cada biestable tiene un pequeño multiplexor controlado por la señal *transmite*, proveniente del receptor. Si *transmite* = '0', estamos en recepción y por tanto a los biestables llegan *urxd* y *urx0*, y el funcionamiento es correcto; si *transmite* = '1', nosotros estamos transmitiendo y ocupando el bus. En este caso, la entrada del primer biestable se pone a '1' (que es el estado de reposo de la línea *urxd*), y la del segundo a '0' (para que no se señalice un SE0). Por lo tanto, la misión de estos multiplexores es que el receptor no vuelva a recoger el paquete que estamos enviando, y así el módulo *director* no se pueda equivocar. Lo que se hace es desconectar el receptor del bus durante la transmisión.

6.3.4. Transmisor

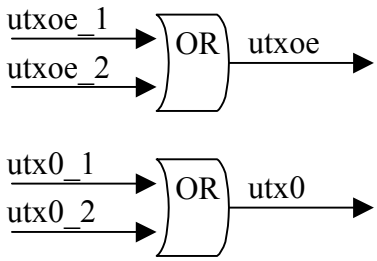
Este componente se encuentra implementado en el fichero *transmisor.vhd*, y su función es enviar todos los paquetes que tenga que enviar la tarjeta a través del bus USB. Estos paquetes sólo pueden ser asentimientos (ACK y NACK), y de datos (DATA0 y DATA1), con datos desde 0 bytes hasta 8 bytes. Su estructura es la siguiente (apaisado):



Las tres líneas de salida por la izquierda, *utxoe*, *utx0* y *utxd* son las que van al módulo *adaptador* y por tanto directamente al bus (la última de ellas es la señal a transmitir, codificación de línea incluida). El resto de conexiones van al módulo *director*, que controla todo el proceso de transmisión, a la RAM, para leer los datos que hay que enviar, y al módulo *crc16*, que no hay que olvidar que también forma parte del transmisor. Cuando este módulo está en funcionamiento, la línea *transmite* está activa, por lo que el control del módulo *crc16* lo tenemos aquí. Las líneas que van al *crc16* son las que parten del módulo *tx_data*, y además la línea *dato_1*, que va de *tx_data* a *bit_stuffing*, y que contiene los bits a transmitir, y la línea *stuffed_bit*, que va en sentido contrario a la anterior y que se activa cuando se está transmitiendo un bit de stuffing. Con todo esto podemos utilizar correctamente el módulo *crc16*.

El módulo *Tx_handshk* se encarga de transmitir un paquete asentimiento (ACK o NACK), el módulo *Tx_data* transmite un paquete de datos (DATA0 o DATA1), el módulo *bit_stuffing* inserta los bits de stuffing que sean necesarios en los paquetes de datos (en los asentimientos no es necesario ya que son cadenas de bits determinadas en las que nunca hay seis '1' consecutivos), y tras esto hay un multiplexor que selecciona las líneas de uno u otro bloque para pasarlas al transceptor. En ningún momento se transmiten dos paquetes a la vez, por lo que no hay problema de tener que elegir entre uno u otro.

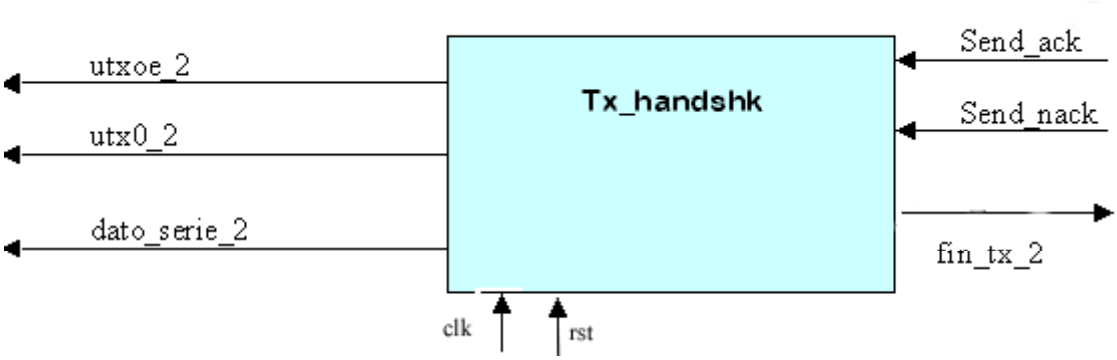
De hecho, realmente este multiplexor sólo existe para las líneas *dato_serie_1* y *dato_serie_2*. Para las líneas *utxoe_n* y *utx0_n* lo único que hay son puertas or, ya que nunca van a estar activas las dos simultáneamente (por ejemplo, nunca se activan *utxoe_1* y *utxoe_2* a la vez), por lo que nos basta con la puerta or.



Por último, tras el multiplexor, el módulo *NRZI_encoder* realiza la codificación de línea sobre los datos a transmitir, por lo que éstos ya se pueden enviar al bus.

Veamos los módulos del transmisor con más detalle:

6.3.4.1. Tx_handshk



Este módulo está implementado en el fichero *tx_handshk.vhd*. Su función, como acabamos de decir, es generar asentimientos. Cuando se activa la línea *send_ack* genera un paquete ACK, y cuando se activa *send_nack*, genera un paquete NACK. Cuando termina la transmisión, da un pulso en la línea *fin_tx* (en el transmisor se llama *fin_tx_2*).

La secuencia de salida (*dato_serie_2*) debe ir al codificador de línea, mientras que las líneas *utxoe_2*, que habilita el transceptor para transmitir si vale '1', y *utx0_2*, que señaliza un "SE0" cuando vale '1', van directamente al módulo *adaptador*.

Recordar que un asentimiento simplemente son 16 bits (los campos SOP y PID) más dos bits "SE0" y un '1', que señalizan el final del paquete. Por lo tanto, si hay que transmitir un ACK, la secuencia a enviar es

00000001	01001011	SE0 SE0 1
SOP	PID	EOP

Y si hay que enviar un NACK,

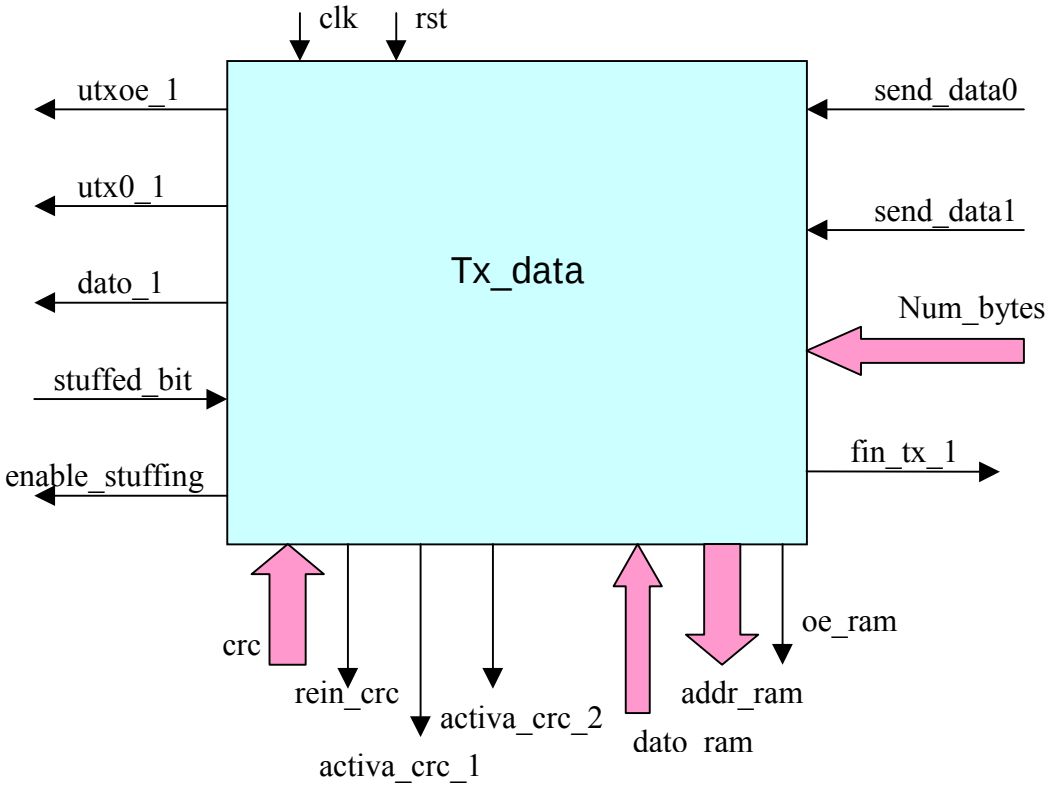
00000001	01011010	SE0 SE0 1
SOP	PID	EOP

Por tanto, el funcionamiento del módulo es muy simple: es una máquina de estados que cuando se acciona empieza a sacar por la línea serie una de las dos cadenas,

ayudada por un contador. A su vez, debe controlar las líneas *utxoe* y *utx0*, y su sincronización con la línea de datos. Aquí hay que tener cuidado, ya que como el módulo *NRZI_encoder* introduce un retardo de un ciclo en la línea de datos, éstos deben ir adelantados un ciclo con respecto a *utxoe* y *utx0*.

En este caso no hace falta ni cálculo de CRC ni generación de bit_stuffing, como ya se ha explicado anteriormente.

6.3.4.2. Tx_data



Este módulo se implementa en el fichero *tx_data.vhd*. Su función es generar un paquete de datos por la línea *dato_1* cuando se activan las señales *send_data0* o *send_data1*. Lógicamente, si se activa la primera se enviará una paquete DATA0, y si se activa la segunda, un paquete DATA1. Transmite el número de bytes que se indiquen en la línea *num_bytes* al comienzo del proceso, que puede ir de 0 a 8. Cuando termina, pone la señal *fin_tx* a '1' durante un periodo.

La estructura de un paquete de datos es la siguiente:

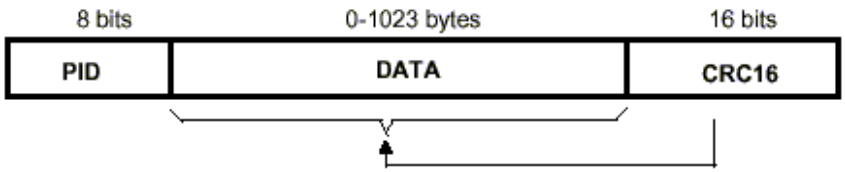


Figure 8-7. Data Packet Format

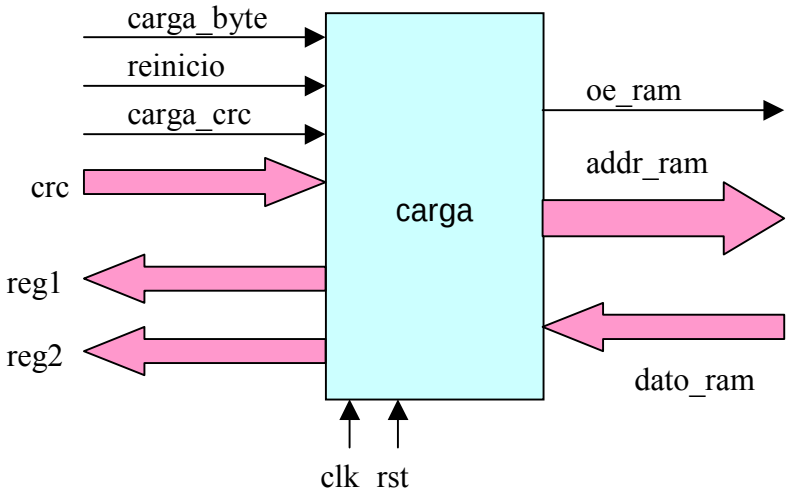
En esta figura, falta el campo SOP (Start of Packet), precediendo al paquete, y el EOP (End of Packet) al final.

El funcionamiento de este módulo es muy parecido al de *tx_handshk*. En primer lugar, se empieza a mandar a la línea el campo SOP y el PID de la misma forma, a través de una cadena predeterminada. Cuando se termina con esto, se empiezan a mandar los bytes de datos. Estos están inicialmente almacenados en la RAM, de las posiciones 8 a 15. Se recogen de ahí y se van enviando. A la vez, se va calculando el CRC. Cuando se terminan de enviar los datos, se envían los 16 bits del CRC calculado y se da la señalización de EOP del mismo modo que en *tx_handshk*. Como la línea de datos va al generador de bits de stuffing, si éste está introduciendo un bit de stuffing, activa la línea *stuffed_bit* y el módulo *tx_data* detiene todo el proceso durante ese ciclo. La señal *enable_stuffing* habilita la inserción de bits de stuffing, y está activada durante todo el proceso.

Del mismo modo que en el módulo anterior, los datos en la línea de salida van un periodo de reloj adelantados con respecto a las señales *utxoe* y *utx0*, para compensar el retraso que introduce el codificador de línea.

¿Cómo se leen los bytes a enviar de la memoria? Esto no lo hace este módulo directamente, sino que implementa en su interior otro componente, llamado *carga*, que lee los bytes de la RAM y se los pasa a nuestro módulo. Por tanto, *Tx_data* no maneja directamente las líneas de acceso a la RAM, sino que lo hace *carga*. Veámoslo:

carga



Este módulo se implementa en el fichero *carga.vhd*. En la figura, las líneas de la izquierda provienen del módulo *tx_data*, y las líneas de la derecha van directamente hacia la RAM. El elemento principal de este bloque son dos registros de 8 bits cada uno: reg1 y reg2. Cuando se da un pulso en *carga_byte*, se lee una posición de memoria y se almacena en uno de los dos registros. La primera posición que se lee es la 8, que corresponde con el inicio del búfer de transmisión, que va de la 8 a la 15. Este primer byte se guarda en reg1, y puede ser leído desde fuera del módulo. Por cada pulso en *carga_byte*, se lee la siguiente posición del búfer y se almacena alternativamente en reg1 y reg2. Si se da un pulso en la línea *reinicio*, el contador de posición se pone a cero (la siguiente lectura será en la posición 8), y el siguiente registro escrito será reg1.

Si se da un pulso en *carga_crc*, el funcionamiento es distinto: los registros se configuran como si fueran uno solo de 16 bits de ancho, y se almacena el valor que entre por el bus *crc* en ese momento. Mejor dicho, se almacena el valor del bus, pero negado bit a bit y en el orden inverso. Este es el código:

```
...elsif carga_crc='1' then
        p_reg1(0)<=not(crc(15));
        p_reg1(1)<=not(crc(14));
        p_reg1(2)<=not(crc(13));
        p_reg1(3)<=not(crc(12));
```

```
                p_reg1(4)<=not(crc(11));
                p_reg1(5)<=not(crc(10));
                p_reg1(6)<=not(crc(9));
                p_reg1(7)<=not(crc(8));
                p_reg2(0)<=not(crc(7));
                p_reg2(1)<=not(crc(6));
                p_reg2(2)<=not(crc(5));
                p_reg2(3)<=not(crc(4));
                p_reg2(4)<=not(crc(3));
                p_reg2(5)<=not(crc(2));
                p_reg2(6)<=not(crc(1));
                p_reg2(7)<=not(crc(0));
        end if;...
```

Esto se hace así porque es como hay que enviar el CRC, según se vio en el punto 6 del algoritmo en la página 52. De esta forma, el módulo *tx_data* utiliza este componente de dos maneras: primero va leyendo los bytes de datos a enviar de estos registros (alternativamente), y cuando acaba con los datos, carga el crc y empieza a enviarlo de la manera correcta. Posteriormente, reinicia el componente para volver a usarlo en el siguiente paquete.

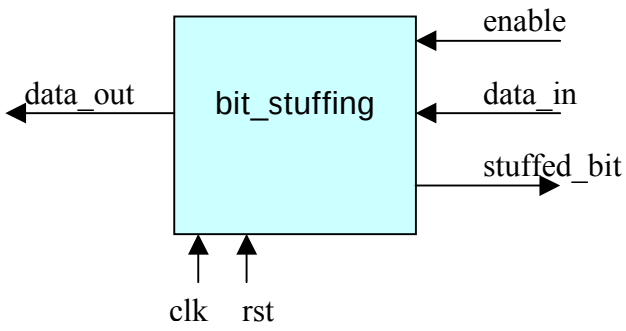
Crc16

Nos queda por resolver una cuestión en *tx_data*; ¿cómo se calcula el crc? Para resolverlo, recordemos el módulo *CRC16* en el apartado 6.3.3.6. Vimos que en transmisión, las líneas *bit*, *stuffed_bit*, *fin_PID* y *data_PID* provenian del transmisor. Pues bien, la primera, dado que es la línea de datos serie, en transmisión se conecta a la salida *dato_1* de este módulo (por donde sale el paquete generado); la segunda se conecta a la línea *stuffed_bit* proveniente del módulo *bit_stuffing* (lógico), y la tercera y la cuarta servían para iniciar el cálculo. Primero se activaba *fin_PID*, y en el ciclo siguiente, *data_PID*. Ahora estas líneas se conectan a *activa_crc_1* y *activa_crc_2* respectivamente, y la máquina de estados de *tx_data* las maneja de la forma adecuada para iniciar el cálculo cuando empiecen a salir los bits de datos.

Por otro lado, teníamos las líneas *aborta* y *crc_out* del módulo *CRC16*, que sólo usaba el transmisor. Estas líneas están permanentemente conectadas al módulo *tx_data*, *crc_out* es la salida de los 16 bits del crc calculado, y se conecta a la señal *crc* , de donde se toma el resultado del cálculo del crc, y la señal *aborta* se conecta a *rein_crc*.

Se usa para poner el módulo *CRC16* en su estado inicial y poder volver a usarlo sin error. Esto es porque al iniciar el cálculo del crc en transmisión, se le dice a *crc16* que lo calcule para 8 bytes, independientemente del número de bytes real. Así, cuando terminamos de enviar datos, se recoge el crc correcto y se envía; mientras *crc16* sigue calculando, por lo que hay que pararlo. Se da un pulso en *rein_crc* y se devuelve el módulo a su estado de reposo.

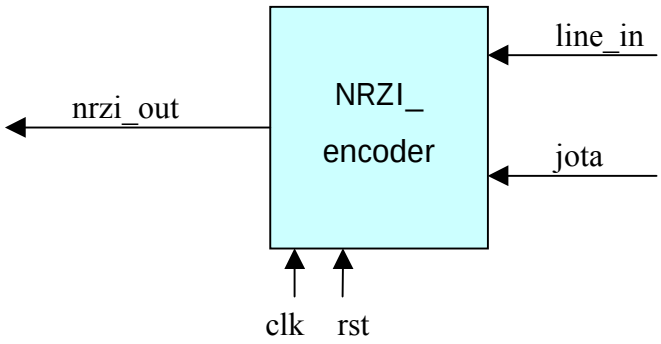
6.3.4.3. Bit_stuffing



Este módulo se implementa en el fichero *bit_stuffing.vhd*. Cuando la señal *enable* está a '1', va comprobando los bits que entran por *data_in*, y los va sacando por *data_out*. Si llegan seis '1' consecutivos, inserta un '0' en *data_out* (bit de stuffing) y durante ese periodo pone *stuffed_bit* a '1' para que los bloques *tx_data* y *crc16* sepan que se está insertando el bit y se bloqueen durante ese periodo de reloj.

Su implementación es bastante simple, no es más que un contador y un poco de lógica combinacional. En cuanto a la entrada hay un '0', el contador se pone a '0'. Si *enable* = '0', la secuencia de entrada se pasa directamente a la salida. Esta salida está ya lista para ser enviada al codificador de línea.

6.3.4.4. NRZI_encoder



Este componente se implementa en el fichero *NRZI_encoder.vhd*. Es un codificador de línea que emplea el código NRZI, que consiste en que si el bit es un '0', se produce un cambio en la línea, y si es un '1', no. Hay que recordar que en reposo, la línea de salida debe de estar siempre a '1' (estado 'J', frente al '0', que se llama 'K' en USB), independientemente del último paquete. Por eso el formato de un carácter EOP (End of Packet) es "SE0 SE0 J"; la ultima J hace que la línea quede a '1' tras la transmisión.

En nuestro caso, los datos entran por *line_in* provenientes de *tx_handshk* o de *tx_data*, y la salida *nrzi_out* es la misma secuencia pero con el código de línea, de forma que está lista para ser enviada. De aquí va directamente al bus USB, pasando por el módulo *adaptador*, que no la toca, y por el transceptor, que la convierte de valores lógicos a valores diferenciales que manda por D+ y D-, las líneas del bus.

La línea *jota*, cuando se pone a '1', fuerza un '1' ('J') en la línea de salida. Como está conectada a la señal *utx0*, esto hace que tras el SE0 que señaliza el final de un paquete, la línea quede en el estado 'J', que es lo correcto como se ha explicado anteriormente.

Hay que decir que la señal *nrzi_out* es directamente la salida de un biestable. Esto le da estabilidad al valor que se pone en el bus y por tanto evita errores en la recepción en el PC, pero introduce un retraso de un periodo de reloj en la línea que hay que tener en cuenta a la hora de sincronizar la secuencia de datos con las señales *utxoe* y *utx0*.

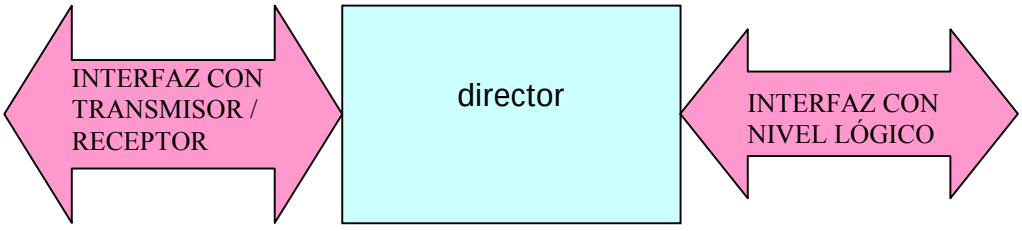
Con este módulo, ya hemos visto el transmisor completamente. Por lo tanto, pasamos a ver el último bloque del nivel físico del interfaz USB, que es el módulo *director*.

6.3.5. Director

Este módulo es el "cerebro" del nivel físico. Se implementa en el fichero *director.vhd*, y no es más que una máquina de estados que controla el resto de bloques del nivel físico, es decir, el receptor y el transmisor. Un nombre mejor para este componente sería "controlador del endpoint". Su función es dirigir transferencias de control, tanto de lectura como de escritura, y tener en cuenta todos los aspectos del nivel físico: comprobación de crc's, mantenimiento del mecanismo "data-toggle", comprobación de la dirección de los paquetes, control de la expiración de contadores en el bus, reenvío de datos, etc. Para el entendimiento completo de este módulo, conviene revisar la estructura de las transferencias de control en el apartado 4.2.3.4 de este documento.

En primer lugar, veremos las distintas líneas de entrada / salida del módulo, luego veremos cómo son las transferencias que implementa y cómo debe ser manejado este dispositivo desde el nivel lógico, y por último un diagrama de flujo que aproxime su funcionamiento.

6.3.5.1. Interfaces



Como se observa en la figura, tenemos dos grupos de líneas de entrada y salida: aquellas que van al receptor o al transmisor, y que son todas aquellas de las que ya

hemos hablado en sus apartados correspondientes, y las que implementan la comunicación con el nivel lógico, que van al módulo *USBdevice*.

Interfaz con transmisor y receptor

Este interfaz consta de las siguientes líneas:

- ***Paquete*** : (entrada). Proviene del módulo *detecta_paquete*. Indica que se está recibiendo un paquete. Cuando se desactiva, la recepción del paquete ha terminado.
- ***PID*:** (entrada). Salida paralelo primer registro del receptor. Por tanto, cuando se ha recibido un paquete, contiene el PID.
- ***PID_catch:*** (entrada). Es la señal *fin_adquis* del primer registro de desplazamiento del receptor. Un pulso indica que se acaba de recoger el PID, pero que el resto del paquete sigue recibiéndose en los otros registros. (no ha acabado el paquete).
- ***ADDR*:** (entrada). Es la salida paralelo del segundo registro del receptor. Cuando llega un paquete tipo token, recoge la dirección del dispositivo destino de la transacción. Así "director" comprueba si es para nosotros.
- ***Crc5_ok*:** (entrada). Es la señal *crc_ok* del módulo *crc5*. Si vale '1', el token se ha recibido correctamente. Valida la dirección de la transacción.
- ***Crc16_fin*:** (entrada). Da un pulso cuando el módulo *crc16* ha terminado de calcular el crc de un paquete de datos que ha llegado.
- ***Crc16_ok*:** (entrada). Si da un pulso a la vez que la señal anterior, el crc es correcto y por tanto los datos también. Si no lo da, los datos están corruptos y hay que esperar su retransmisión.
- ***Enable_memoria*:** (salida). Es la señal *disparo* del módulo *usb2RAM*. Le indica a este módulo que recoja 8 bytes de datos y los guarde en la RAM.
- ***Enable_mem_2*:** (salida). Es la señal *guarda_out* del módulo *usb2RAM*. Le indica a éste que recoja datos y los guarde en RAM. La señal anterior era para recoger una nueva instrucción, y ésta para recoger bytes de datos. El número de bytes a recoger se indica en el bus *num_datos*.
- ***Num_datos:*** (salida). Indica el número de bytes a recibir al receptor, y el número de bytes de datos a transmitir al transmisor.

- ***Control:*** (salida). Esta señal selecciona el modo de funcionamiento de los registros de desplazamiento del receptor (para recoger datos o tokens). De paso, sirve para activar el módulo *crc16* en recepción.
- ***Transmite:*** (salida). Indica transmisión ('1') o recepción ('0'). Deshabilita el paso de paquetes al receptor cuando está en transmisión.
- ***Send_ack:*** (salida). Ordena al transmisor enviar un asentimiento positivo.
- ***Send_nack:*** (salida). Idem, para asentimiento negativo.
- ***Send_data0:*** (salida). Idem, para paquete "DATA0".
- ***Send_data1:*** (salida). Idem, para paquete "DATA1".
- ***Fin_tx:*** (entrada). El transmisor indica que ha terminado de enviar cualquiera de los paquetes anteriores.

Ya hemos visto el funcionamiento de estas líneas, por lo que no requieren explicación adicional. El diagrama de flujo de la máquina de estados aclarará como las usa el módulo *director*.

Interfaz con el nivel lógico

Estas líneas van todas entre el módulo *director* y el módulo *USBdevice*, en el que está implementado el nivel lógico de dispositivo. Sirven para establecer un protocolo de comunicación entre los dos componentes, con el fin de secuenciar las distintas fases de una transferencia de control sin pisar los datos que se guardan en la RAM. Estas señales son las siguientes:

- ***Setup_stage:*** (salida de "director"). Cuando se activa, indica al nivel lógico que ha llegado una instrucción nueva. Los 8 bytes de esta instrucción están guardados en las posiciones 0 – 7 de la RAM.
- ***En_ctrl_rd:*** (entrada). Si se activa, el nivel lógico le indica a *director* que ha leído la instrucción y que ha guardado datos en la RAM (posiciones 8 – 15) para que el transmisor los envíe al PC. Es decir, da paso a la fase de datos de una transferencia de lectura (los datos van del dispositivo al PC).
- ***En_ctrl_wr:*** (entrada). Si se activa, el nivel lógico le indica a *director* que ha leído la instrucción y que pase a recibir el siguiente paquete de datos que llegue y lo guarde en memoria (posiciones 16 – 23, ya que son datos para la aplicación). Es

decir, da paso a la fase de datos de una transferencia de escritura (los datos van del PC al dispositivo).

- ***D_stage:*** (salida). Indica al nivel lógico que se han enviado (lectura) o recibido (escritura) los datos a/desde el PC, y que éste pide más o envía más. Es decir, que prosigue la fase de datos con otra transacción IN o OUT. El nivel lógico debe procesar esto y volver a activar alguna de las dos señales anteriores para que siga la fase de datos.
- ***St_stage:*** (salida). Indica al nivel lógico que ha terminado la fase de datos y que pase a la fase de estado de la transferencia de control (indica que se ha recibido un token en sentido contrario a los anteriores).
- ***En_status:*** (entrada). Cuando se activa, el nivel lógico autoriza a proseguir con la fase de estado. Para él, la transferencia de datos ha terminado.
- ***Error:*** (salida). Indica al nivel lógico que mientras se procesaba la transferencia ha llegado un nuevo paquete SETUP, indicativo e una nueva transferencia. El nivel lógico debe abortar la primera instrucción y atender la nueva. Esto es útil si la máquina de estados pierde la sincronización de la transferencia. El PC reintentará la transferencia y entonces la máquina se vuelve a sincronizar, sin necesidad de resetear el dispositivo.

Adicionalmente, en este interfaz tenemos dos buses con otros objetivos:

- ***Dev_dir:*** (entrada a *director*). Contiene la dirección del dispositivo. Director debe comparar la de los paquetes que lleguen (señal ADDR) con ésta para determinar si somos el objetivo de la transmisión.
- ***Bytes_in:*** (entrada). Indica a *director* el número de bytes de datos a transmitir o a recibir. *director* pasa este dato al transmisor o al receptor en *num_datos*.

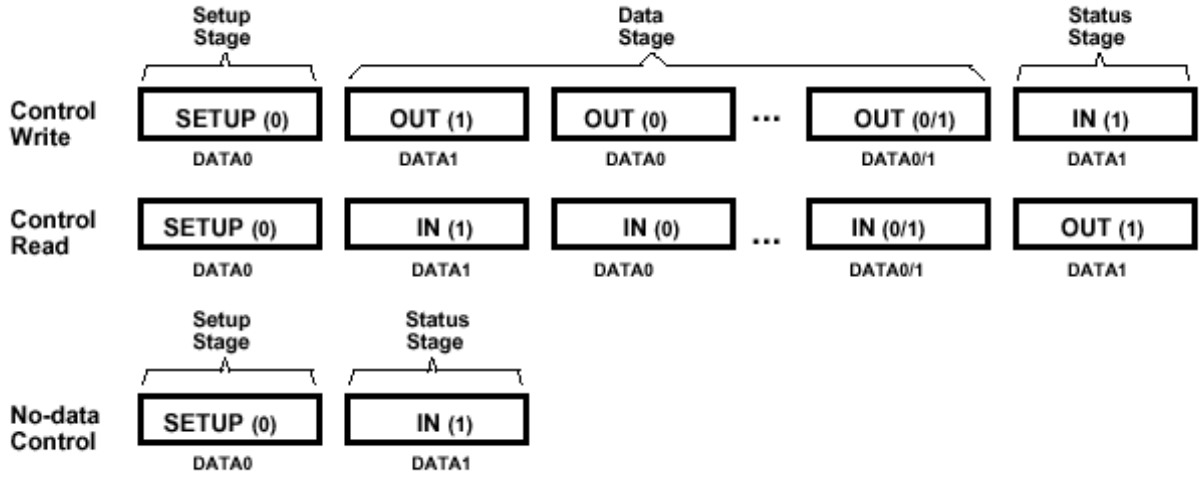
Ya hemos visto todas las líneas de entrada y salida del módulo *director*. Ahora vamos a ver cómo son las transferencias que implementa y cómo son los protocolos para controlar estas transferencias desde el nivel lógico.

6.3.5.2. Manejo de *director*. Transferencias de control

El módulo *director* puede controlar tres tipos de transferencias de control distintas:

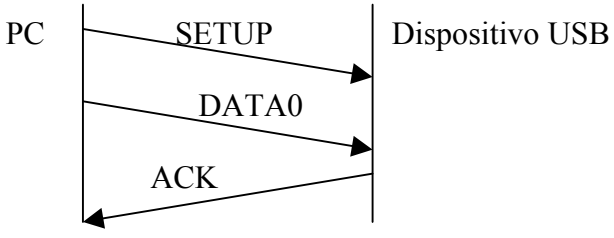
- Transferencia de lectura o "control read": En la fase de datos, éstos van del dispositivo al PC.
- Transferencia de escritura o "control write": Los datos van del PC al dispositivo.
- Transferencia sin fase de datos o "no-data control": no tiene fase de datos.

Recordemos que la estructura de estas transferencias era la siguiente:



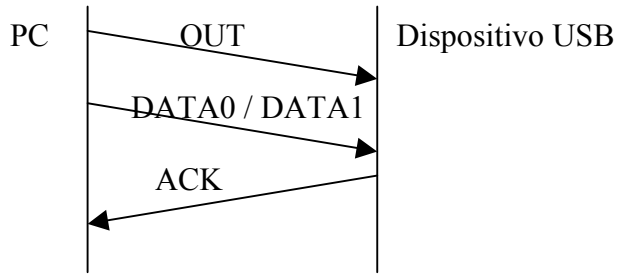
En la figura se pueden ver los tres tipos de transferencias, y las fases de cada una. Cada recuadro representa una transacción, y por lo tanto está formado por tres paquetes: un token, un paquete de datos (DATA0 o DATA1) y un asentimiento. El titulo que se indica en cada recuadro es el token de esta transacción, y el valor entre paréntesis indica el tipo de paquete de datos (DATA0 o DATA1).

Así, la primera transacción es la misma para cada tipo de transferencia:



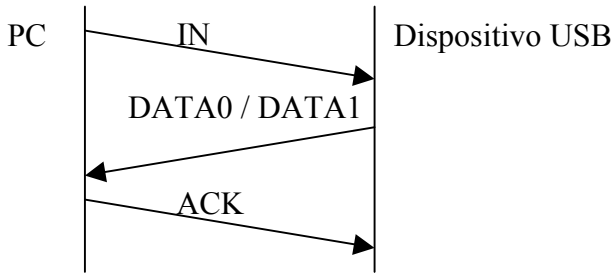
El diagrama anterior indica todos los paquetes que componen la transacción de la fase de SETUP. El paquete DATA0, en su campo de datos, contiene los 8 bytes que componen la instrucción. Veremos el formato de estas instrucciones en el capitulo dedicado al nivel lógico. Este paquete no puede ser DATA1, siempre DATA0, ya que aquí se inicia la secuencia de data-toggle. Por otro lado, es obligatorio enviar el asentimiento, y éste solo puede ser ACK, nunca NACK. Es decir, no se puede rechazar este paquete de datos.

En las transferencias de escritura, la fase de datos está formada por transacciones OUT, de la forma que sigue:

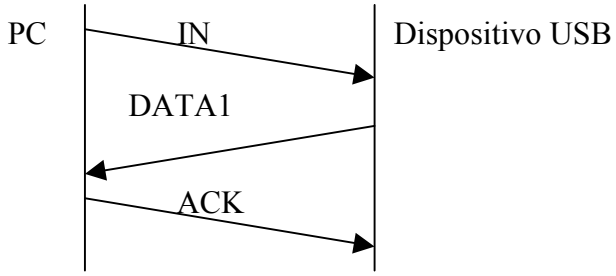


Ahora el token es OUT y el paquete de datos será DATA0 o DATA1 según corresponda en la secuencia del data-toggle (se alternan paquetes DATA0 y DATA1). Como el de la fase de SETUP fue DATA0, el primero de la fase de datos será DATA1, el siguiente DATA0, y así sucesivamente. En cuanto al asentimiento, éste puede ser ACK, como en la figura, si el receptor ha recibido los datos correctamente y no hay errores, o bien NACK, si el receptor todavía no puede recibir (porque todavía no se hayan terminado de procesar los datos anteriores, para no machacar el búfer de recepción). Si se envía NACK, el PC volverá a intentar la transacción más tarde. Si hubiera errores en los datos, no se envía ningún asentimiento. El PC detecta esto porque expira un contador, y reintenta la transacción.

En las transferencias de lectura, la fase de datos está formada por una serie de transacciones IN, como se ve en la siguiente figura. En este caso, se cumplen las mismas reglas para los paquetes de datos que en el caso anterior, pero en los asentimientos es distinto: El PC no puede enviar NACK, sólo ACK. Si los datos fuesen erróneos, no se enviaría asentimiento y el dispositivo tendría que detectar un “time-out” (expira un contador), con lo cual en la siguiente transacción IN se reenviarían los datos.

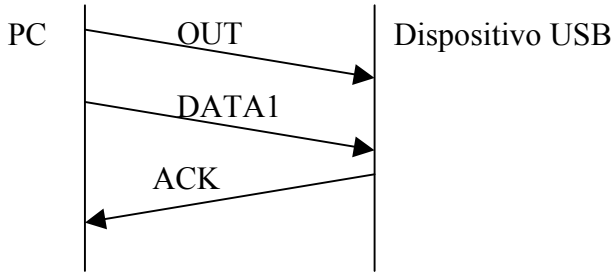


Veamos por último las transacciones de las fases de estado. Siempre son transacciones con el token contrario al de la fase de datos. En las transferencias de escritura y en las transferencias sin datos, tenemos un token IN:



El paquete de datos siempre debe ser DATA1, para finalizar la secuencia de data-toggle, y en su campo de datos no contiene ningún byte de datos. Sirve para que el dispositivo indique al PC que ha cumplido la instrucción correctamente. De nuevo, si el PC no envía el ACK, es que ha habido errores y se reintentará la transacción.

En las transferencias de lectura, la fase de estado es así:



De nuevo, el paquete de datos debe ser DATA1 y no contener datos, y nosotros podemos enviar ACK para asentir, con lo que finaliza la transferencia, NACK para

rechazar, con lo que reintentaremos la fase de estado más tarde, o no asentir nada, lo que indica que ha habido errores y que se debe reintentar la fase de estado.

Pues bien, el control de todo esto es lo que implementa nuestro módulo *director*. Veamos a continuación el protocolo que debemos seguir para implementar estas transferencias desde el nivel lógico, es decir, cómo debemos usar el componente *director*.

Transferencias de lectura (control read)

1. **Esperar a que *setup_stage* valga '1'**. Entonces, leer la nueva instrucción del búfer de memoria. Esta instrucción pide enviar datos al PC.
2. Escribir 8 bytes o menos en el búfer de transmisión. Tras esto, **poner la señal *en_ctrl_rd* a '1'**. Esto hace que se transmitan los datos al PC. (antes de esto, poner el número de bytes a enviar, de 0 a 8, en el bus *bytes_in*)
3. **Esperar a que se ponga a '1' o bien *d_stage*, o bien *status_stage*.** Cuando una de las dos se active, **poner *en_ctrl_rd* a '0'**, ya que estaba a '1'. Si se activa la primera, el PC sigue pidiendo datos, por lo que volvemos al paso 2. Si la que se activa es *status_stage*, hemos pasado a la fase de estado (puede que no hayamos enviado todos los datos, es igual). Ir al paso 4.
4. **Poner *en_status* a '1'**. Esto indica que se ha entendido el fin de la transferencia.
5. **Esperar a que *status_stage* valga '0'** y entonces **poner *en_status* a '0'** y volver al reposo. FIN.

Transferencias de escritura (control write)

1. **Esperar a que *setup_stage* valga '1'**. Entonces, leer la nueva instrucción del búfer de memoria. Esta instrucción pide recibir datos del PC.
2. **Poner el número de bytes a recibir en el bus *bytes_in*** y **poner *en_ctrl_wr* a '1'.** Esto autoriza la recepción de datos en el búfer de recepción de datos (posiciones 16 – 23 de la RAM). El número de bytes a

recibir se lee de la instrucción. Si es mayor que 8, en *bytes_in* se escribe 8, ya que cada transacción lleva 8 bytes de datos como máximo.

3. **Esperar a que se ponga a '1' o bien *d_stage*, o bien *status_stage*.** Cuando una de las dos se active, **poner *en_ctrl_wr* a '0'**, ya que estaba a '1'. Si se activa la primera, indica que se han recibido datos en el búfer. Procesar estos datos. Volver al paso 2. Si la que se activa es *status_stage*, no había más datos que recibir, por lo que hemos pasado a la fase de estado. Ir al paso 4.
4. **Poner *en_status* a '1'**. Esto indica que se ha entendido el fin de la transferencia.
5. **Esperar a que *status_stage* valga '0'** y entonces **poner *en_status* a '0'** y volver al reposo. FIN.

Nótese que los pasos 1, 4 y 5 son iguales en ambos tipos de transferencias. Eso es porque ambos protocolos usan las líneas *setup_stage*, *st_stage* y *en_status* del mismo modo. El tercer tipo de transferencia hace un uso distinto de las líneas de la fase de estado:

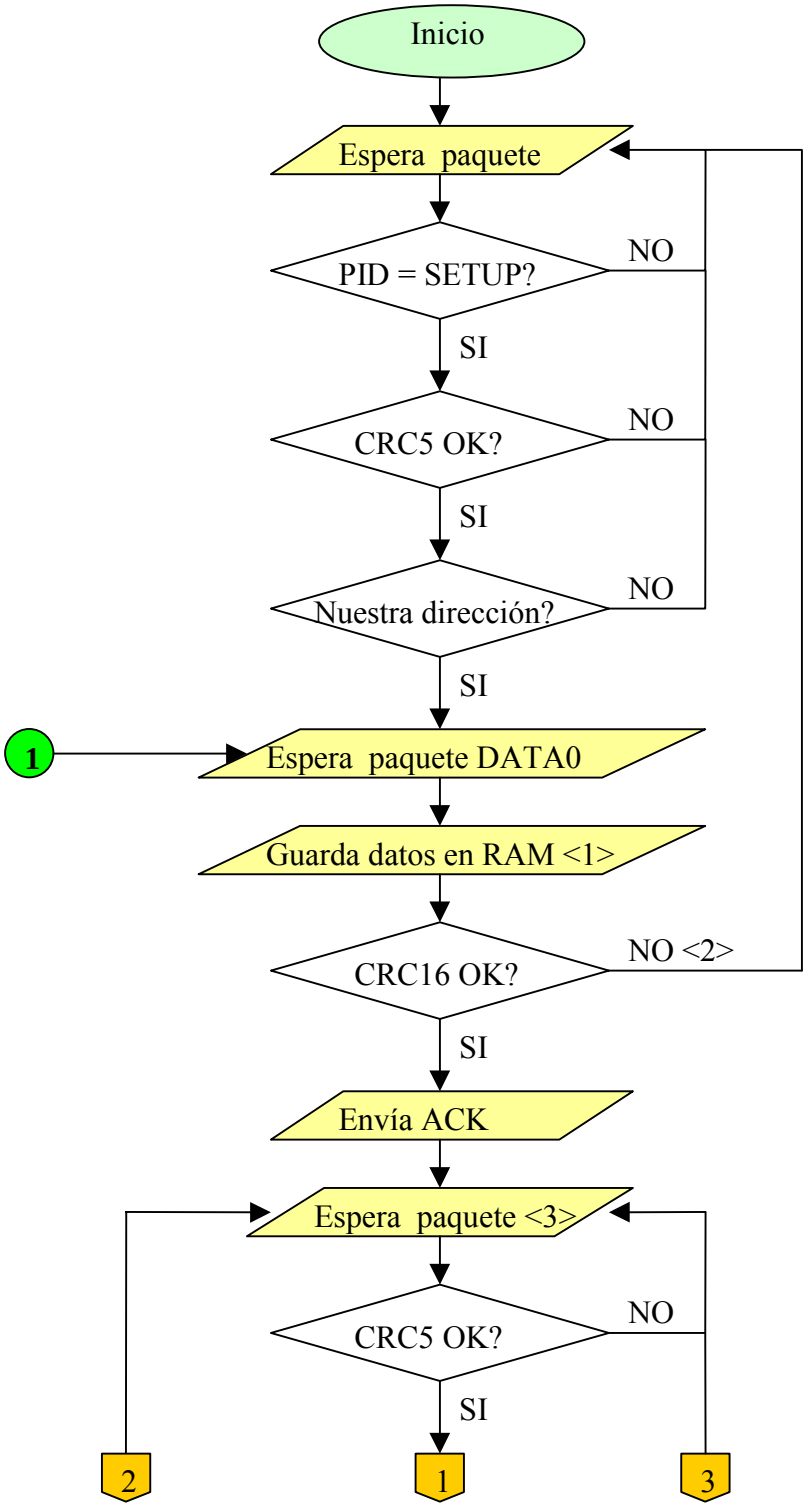
Transferencias sin datos (no-data control)

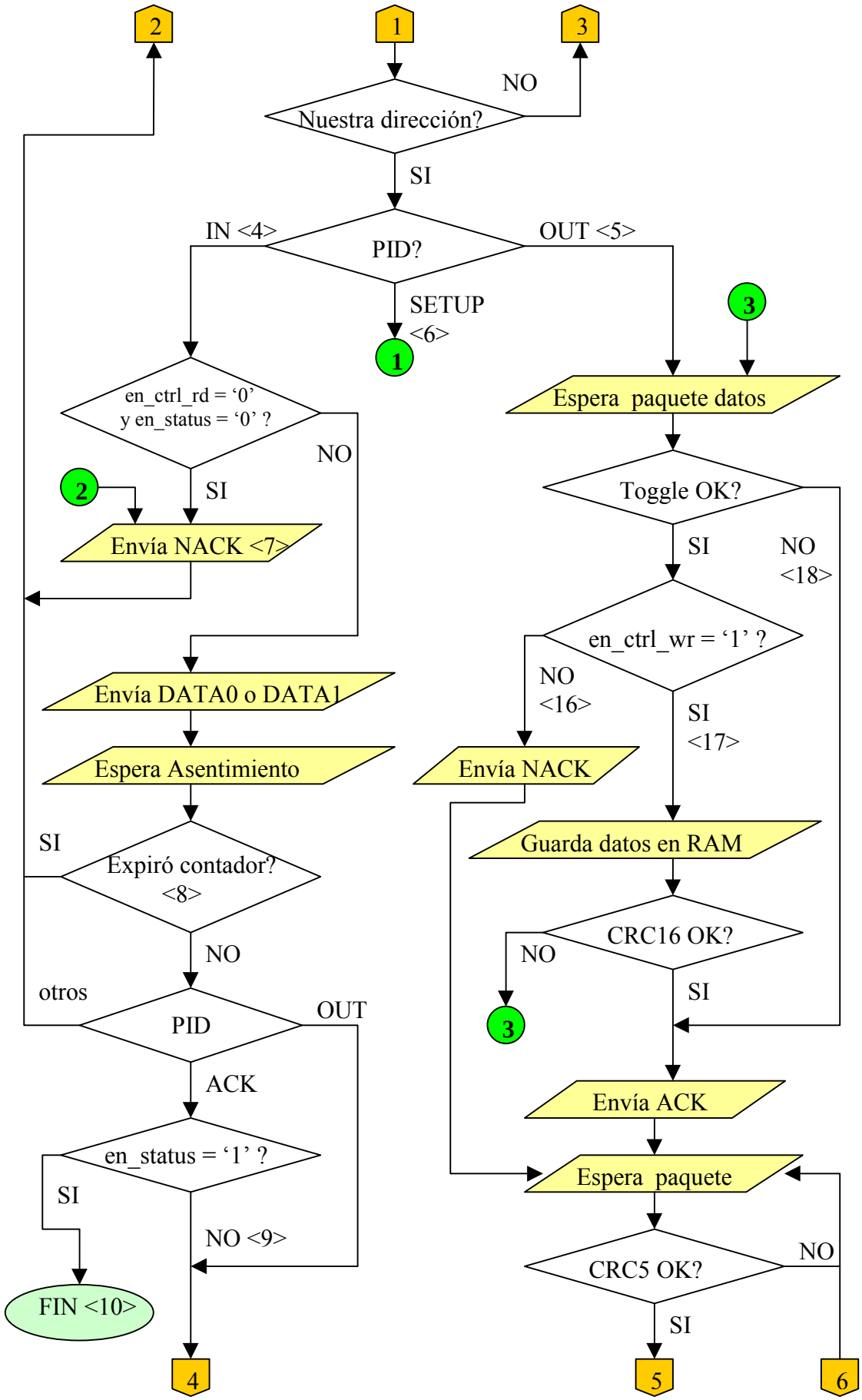
1. **Esperar a que *setup_stage* valga '1'**. Entonces, leer la nueva instrucción del búfer de memoria. Esta instrucción no pide ni recibir datos del PC, ni enviarlos. Ejecutar la instrucción.
2. **Poner *en_status* a '1'**.
3. **Esperar a que *status_stage* valga '1'**.
4. **Poner *en_status* a '0'** y volver al reposo.

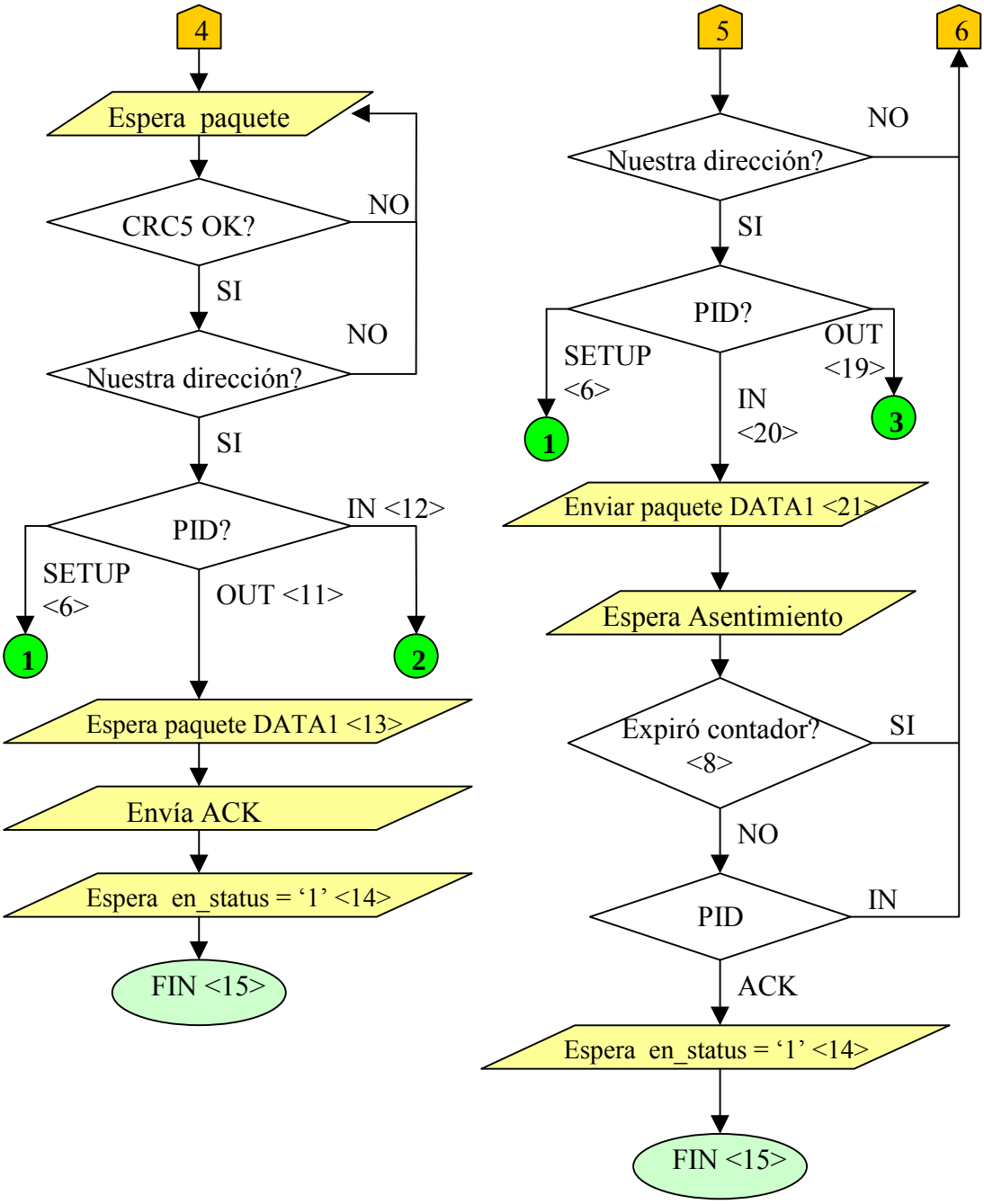
Como en este caso no hay fase de datos, la transferencia se simplifica mucho.

6.3.5.3. Diagrama de flujo

Para terminar de caracterizar el componente *director*, vamos a ver un diagrama de flujo que nos muestra su funcionamiento. Dada su complejidad, se han introducido algunos saltos que se relacionan a través de círculos numerados como este: 1
Los números entre símbolos "<" y ">" indican notas al final del diagrama.




2
1
3
NO
Nuestra dirección?
SI
IN <4>
PID?
OUT <5>
SETUP
<6>
1
3
en_ctrl_rd = '0'
y en_status = '0' ?
NO
SI
2
Envía NACK <7>
Espera paquete datos
Toggle OK?
SI
NO
<18>
en_ctrl_wr = '1' ?
NO
<16>
SI
<17>
Envía DATA0 o DATA1
Espera Asentimiento
Envía NACK
Guarda datos en RAM
SI
Expiró contador?
<8>
NO
CRC16 OK?
NO
SI
otros
PID
OUT
3
ACK
Envía ACK
en_status = '1' ?
Espera paquete
SI
NO <9>
CRC5 OK?
NO
FIN <10>
SI
4
5
6



NOTAS

<1> Se activa la señal *disparo* del módulo *usb2RAM*, lo que hace que los datos se almacenen en las posiciones 1 – 8 de la RAM (instrucción). También se activa el módulo *crc16*.

<2> Si el CRC de los datos no es correcto, no se envía ningún asentimiento, por lo que el PC reiniciará la transmisión. Nos ponemos a la espera.

<3> Fin de la fase de setup, entramos en la de datos. Ponemos la señal *setup_stage* a '1' para avisar al nivel lógico.

<4> El token recibido es IN. Puede ser una transferencia de lectura o una transferencia sin datos.

<5> El token es OUT. Tenemos una transferencia de escritura.

<6> El token es SETUP, por lo que abortamos la transferencia e iniciamos una nueva. Se da un pulso en la línea *error*, para avisar al nivel lógico.

<7> Si *en_ctrl_rd* y *en_status* son ambas '0', no podemos pasar ni a la fase de datos (en transf.. de lectura) ni a la de estado (en transf.. sin datos). Mientras dure esto, tenemos que seguir contestando a los tokens IN entrantes con rechazos NACK (dispositivo ocupado).

<8> Tras enviar un paquete de datos, tenemos que esperar un asentimiento, pero este puede perderse. Para controlar esto, justo al terminar de transmitir el paquete ponemos en marcha un contador. El tiempo que hay que esperar está definido en la especificación USB. Si el contador expira, pasamos a esperar de nuevo el inicio de la misma transacción. El contador está implementado en el mismo fichero *director.vhd*, y cuenta hasta 32 antes de expirar. Este valor tiene en cuenta el retraso que hay desde que realmente empieza a llegar el asentimiento hasta que lo detectamos.

<9> Si *en_status* = '0', significa que es una transferencia de lectura, por lo que pasamos a esperar el siguiente token.

<10> En cambio, si *en_status* = '1', estamos terminando una transferencia sin datos. Ponemos *status_stage* a '1' durante un ciclo para asentir al nivel lógico el fin de la transferencia, y nos vamos al estado de reposo, a esperar una nueva transacción (a "inicio").

<11> El token es OUT, por lo que se ha terminado la fase de datos y entramos en la de estado (es una transferencia de lectura). Ponemos *status_stage* a '1' y vamos a esperar la llegada del paquete DATA1.

<12> El token es IN, por lo que seguimos en la fase de datos. Damos un pulso en la señal *d_stage*, para indicar al nivel lógico que cargue más datos en el búfer de transmisión, y a este paquete IN contestamos siempre con un NACK.

<13> Este paquete DATA1 de la fase de estado no contiene datos, por lo que ni nos molestamos en comprobarlo.

<14> Esperamos que el nivel lógico nos asienta el fin de la transferencia poniendo *en_status* a '1', y cuando lo hace podemos poner *status_stage* a '0'.

<15> Transacción terminada con éxito. Volvemos al estado de reposo, a esperar el inicio de una nueva transferencia de control.

<16> Si *en_ctrl_wr* = '0', no podemos almacenar los datos que lleguen en la RAM porque los niveles superiores todavía están procesando los datos anteriores. Tenemos que rechazar los paquetes de datos enviando NACK.

<17> Si *en_ctrl_wr* es '1', ya podemos recoger los datos. Los guardamos en la RAM a través del módulo *usb2RAM*, en el búfer de aplicación, posiciones 16 a 23 de la RAM. Damos un pulso en la señal *d_stage* para indicar al nivel lógico que ya tiene datos disponibles para la aplicación.

<18> Hay un error de toggle, esperábamos un paquete DATA0 y ha llegado DATA1 o al revés. Esto quiere decir que la transacción actual es el reenvio de la anterior, ya que el ACK que enviamos se perdió y el PC supuso que hubo error. Por tanto, ignoramos los datos y asentimos el paquete de datos, para que el PC vea que ya se ha recibido.

<19> El token es OUT, por lo que continuamos en la fase de datos. Pasamos a esperar otro paquete de datos.

<20> El token es IN, por lo que pasamos a la fase de estado. Ponemos *status_stage* a '1' para avisar al nivel lógico.

<21> Enviamos el paquete DATA1 que no contiene datos de la fase de estado.

Fin del nivel físico.

6.4. Nivel de dispositivo lógico

Ya hemos visto en su totalidad el nivel físico del dispositivo. Podemos hablar ahora del nivel lógico. Ya se introdujo en el apartado 4.2.3.4 que su función es responder a los comandos o instrucciones que le lleguen a través del bus. Estas instrucciones se originan en el software que se ejecuta en el PC, y se clasifican en tres grupos: instrucciones estándar, instrucciones de clase e instrucciones de fabricante. Las que a nosotros nos interesan son las estándar y las de fabricante.

Las instrucciones estándar son aquellas que debe ejecutar obligatoriamente todo dispositivo que sea totalmente compatible con USB, y se definen en la especificación USB. En nuestro caso, implementamos las mínimas para que se pueda realizar un trasvase de datos correcto, y así ganamos en simplicidad, reducimos mucho este nivel. Aunque el producto resultante no se puede llamar 100% compatible con USB, en la práctica funciona perfectamente incluso cuando hay otros dispositivos conectados al bus al mismo tiempo. Por lo tanto, no perdemos las buenas características del USB. Las instrucciones estándar las genera el software del sistema encargado de gestionar los recursos, es decir, el sistema operativo. En el caso de Windows 98, este encargado es el driver "USBD.sys". Esta es la razón de que estas instrucciones sean obligatorias, para no tener que escribir software particular para cada dispositivo, y que el sistema operativo sepa cómo comunicarse con él con sólo enchufarlo al PC.

Por otro lado, las instrucciones del fabricante ("vendor" en la terminología USB) son todas aquellas que el fabricante del dispositivo quiera añadir a un producto particular además de las estándar. En nuestro caso, hemos añadido dos: una instrucción para llevar a cabo lecturas de datos, y otra para escrituras. Hay que recordar que las instrucciones sólo pueden ir por un canal de control, como es nuestro caso. Estas instrucciones las genera el software particular del dispositivo, es decir, nuestro driver, que será explicado en un apartado posterior.

Como ya hemos visto, una instrucción está formada por ocho bytes, con el siguiente formato:

Table 9-2. Format of Setup Data

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
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

(NOTA: La tercera columna indica el tamaño de cada campo en bytes)

Éstos son los datos que hay en las posiciones 0 – 7 de la RAM cuando se activa la señal *setup_stage*. Como se ve, tenemos cinco campos en cada instrucción. El primero de ellos, *bmRequestType*, indica la dirección de la instrucción, así como el tipo de instrucción (estándar o "vendor"). Aunque este campo se rellenará correctamente por nuestro software, no aporta información relevante, por lo que el nivel lógico no lo usa.

El campo *bRequest* (dirección 1 de la RAM) es el código de la instrucción, nos dice de qué instrucción se trata. Los códigos para nuestras instrucciones (las del fabricante) son los siguientes:

- Instrucción de escritura → *bRequest* = 19 (decimal, 13h hexadecimal)
- Instrucción de lectura → *bRequest* = 20 decimal, 14 hex.

El código de las instrucciones estándar lo veremos en una tabla a continuación.

Los campos *wValue* y *wIndex* dependen de cada instrucción. Cada uno es de dos bytes, y hay que recordar que en USB primero viaja el byte menos significativo, por lo que el búfer se ordena así:

- dirección 2: byte menos significativo de *wValue*
- dirección 3: byte más significativo de *wValue*
- dirección 4: byte menos significativo de *wIndex*
- dirección 5: byte más significativo de *wIndex*

En nuestra instrucción de fabricante "escritura" no se hace uso de estos dos valores; sin embargo, en "lectura", aquí vienen las direcciones a leer. Pueden ir desde 1 hasta 4 direcciones de lectura en cada instrucción, por tanto. Cada dirección es un byte, luego se pueden direccionar 256 posiciones.

Por último, el campo *wLength* nos indica el número de bytes que hay que transferir en la fase de datos de la transferencia, en un sentido o en otro. Si vale 0, esta instrucción no necesita transferencia de datos (se implementa una transferencia sin datos, no-data control). Nuestro dispositivo sólo lee el byte menos significativo, en la dirección 6 de la RAM, por lo que como máximo admite transferencias de 255 bytes de longitud. El driver de usuario se encarga de que no se supere esta longitud, limitando las transferencias a 254 bytes. Es muy fácil adaptar tanto hardware como software para permitir transferencias más largas, sólo hay que leer la dirección 7 de la instrucción y formar un dato *wLength* de dos bytes.

A continuación vamos a ver un cuadro con los códigos de todas las instrucciones estándar. En nuestra solución sólo se implementan las siguientes:

- **SET ADDRESS**: cambia la dirección del dispositivo
- **GET DESCRIPTOR:** envía un descriptor al PC
- **SET CONFIGURATION:** con esta orden se da por terminado el proceso de configuración.

Estas tres son suficientes para funcionar correctamente, nunca se van a enviar ninguna de las otras.

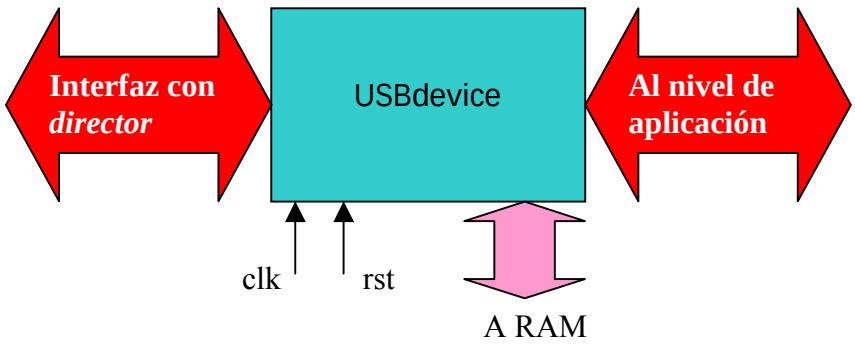
Table 9-4. Standard Request Codes

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

Con toda esta información sobre las instrucciones, podemos ver cómo funciona nuestro dispositivo lógico.

6.4.1. El componente USBdevice

El nivel lógico de nuestra solución está constituido un solo componente, *USBdevice*, que se implementa en el fichero *device.vhd*.



Veamos cada uno de estos interfaces:

- El interfaz con el componente *director* tiene dos grupos de señales: por un lado, están los buses *dev_dir*, que le indica a *director* nuestra dirección, *num_datos*, que contiene el número de bytes a recibir, y el bus *num_bytes*, que no hemos mencionado antes y que no va a *director*, sino directamente al transmisor para indicarle el número de bytes a transmitir (es una excepción). Por otro lado, están las líneas para implementar los protocolos de transferencia con *director* que ya hemos visto: *setup_stage*, *data_stage*, *status_stage* y *error* de entrada, y *en_ctrl_rd*, *en_ctrl_wr* y *en_status* de salida hacia *director*.

- El interfaz con la RAM contiene las líneas de comunicación con ésta para lectura y escritura: bus de direcciones, bus de datos de entrada y de salida, habilitadores de lectura y escritura, y la señal *writing*, que sirve para tomar posesión de los buses de la RAM en escritura, y se usa en el módulo *usbtop*.

- El interfaz con el nivel de aplicación está formado por cuatro líneas:
  - *orden_app* (salida) sirve para activar el módulo de aplicación que realiza escrituras.
  - *app_ok* (entrada) se pone a '1' para indicar que el módulo de aplicación ha terminado las escrituras.
  - *orden_app_rd* (salida) sirve para activar el módulo de aplicación que realiza escrituras.
  - *app_ok_rd* (entrada) se pone a '1' para indicar que el módulo de aplicación ha terminado las lecturas.

Este componente es una máquina de estados cuyo funcionamiento es el siguiente: espera en reposo a que *setup_stage* valga '1'. Cuando esto pasa, hay una instrucción disponible en memoria. La lee, y según el código de instrucción pasa a ejecutar una secuencia de acciones u otra. Cuando termina con la instrucción, vuelve al reposo. Por otro lado, si *error* se pone a '1', se va inmediatamente al estado de reposo (se aborta la ejecución de la instrucción) y se espera la nueva instrucción.

Vamos a ver cómo se ejecuta cada instrucción en este módulo:

6.4.1.1. Instrucciones Estándar

Se responde a las instrucciones **Get Descriptor, Set Address** y **Set Configuration**.. La primera se implementa mediante una transferencia de lectura, y las otras dos mediante transferencias sin datos.

GET DESCRIPTOR

El formato de esta instrucción es el siguiente:

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10000000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero	Descriptor Length	Descriptor

El módulo *USBdevice* lee la dirección 2 de la RAM y encuentra el valor 6, por lo que sabe que la orden es Get Descriptor y que debe enviar un descriptor. A continuación lee las posiciones 6 y 7 para saber el número de bytes a enviar (puede que no se le pida el descriptor completo, de hecho ésta es la primera instrucción que se ejecuta al enchufar el dispositivo, y esta primera vez sólo se le piden los 8 primeros bytes del descriptor, que tiene 18 bytes). Posteriormente, lee el byte más significativo de *wValue* (posición 3), que nos indica el descriptor a enviar. Con estos datos, escribe los bytes del descriptor pedido en la RAM, en el búfer de transmisión (direcciones 8 a 15), y da la orden de enviarlos al PC (pone la línea *en_ctrl_rd* a '1'). Si se han pedido más de 8 bytes, se separa la fase de datos de la transferencia en varias transacciones.

Este dispositivo tiene dos descriptores, que son los que el módulo *USBdevice* puede enviar. Hay muchos más tipos de descriptores, pero debido a la sencillez de nuestro dispositivo, con dos descriptores nos basta para describirlo. Estos descriptores son el "device descriptor" y el "configuration descriptor". Ambos, casualmente, tienen 18 bytes. Veámoslos:

Device descriptor: Cada dispositivo USB tiene uno y solo uno. Debe ser de 18 bytes. El nuestro es el siguiente:

Offset	Campo	Longitud (bytes)	Valor (hex)	Descripción
0	Longitud	1	12	Longitud del descriptor en bytes (18 en decimal)
1	Tipo	1	01	Indica device descriptor
2	Especificación	2	01 00	Especificación USB codificada en BCD Indica Revisión 1.00
4	Clase	1	00	No indica ninguna clase. No tocar
5	Subclase	1	00	No indica ninguna subclase. No tocar
6	Protocolo	1	00	No se usa ningún protocolo de clase
7	Tamaño máximo de paquete	1	08	Ocho bytes por paquete de datos como máximo. Es lo primero que lee el PC
8	Fabricante	2	03 32	Código de fabricante. He elegido éste arbitrariamente. Lo usa el software para elegir el driver de dispositivo correcto
10	Producto	2	00 03	Código de producto. Igual que el anterior
12	Dispositivo	2	00 00	Versión del dispositivo en BCD. Ésta es la 0.0
14	Índice de fabricante	1	00	Índice que apunta a una cadena que describe al fabricante. No usamos ninguna cadena descriptiva
15	Índice de producto	1	00	Idem
16	Numero de serie	1	00	Idem
17	Numero de configuraciones	1	01	Solo tenemos una configuración

Recordar que en cada campo de dos bytes, se envía primero el menos significativo. Así, la cadena de bytes que se envía es:

12 01 00 01 00 00 00 08
32 03 03 00 00 00 00 00
00 01

Necesitamos por tanto tres transacciones para enviarlo completo.

Configuration descriptor: Cada dispositivo tiene uno por configuración. En él se incluyen los descriptores de todos los interfaces que tiene esa configuración, y los de los endpoints que tiene cada interfaz (no se incluye el endpoint de control, éste no hay que describirlo). Por tanto, en nuestro descriptor de configuración se incluye un descriptor de interfaz y ninguno de endpoints. Es el siguiente:

Offset	Campo	Longitud (bytes)	Valor (hex)	Descripción
0	Longitud	1	09	Longitud del descriptor en bytes (9 en decimal, sólo el de configuración)
1	Tipo	1	02	Indica configuration descriptor
2	Longitud total	2	00 12	Longitud total del descriptor (18 en decimal)
4	Nº de interfaces	1	01	Un solo interfaz
5	Identificador de configuración	1	02	Este valor sirve para seleccionar esta configuración desde el software, mediante la instrucción "set configuration". El valor es arbitrario
6	Índice de configuración	1	00	No hay cadena descriptiva
7	Atributos de energia	1	40	Es un mapa de bits. Indica que el dispositivo tiene su propia alimentación y que no puede despertar al sistema
8	Potencia máxima	1	00	Indica que no consume ninguna intensidad a través de las líneas del bus
9	Longitud	1	09	Longitud del descriptor de interfaz (éste empieza aquí)
10	Tipo	1	04	Indica "interface descriptor"
11	Numero de interfaz	1	00	El primero y único es el 0
12	"Alternate Setting"	1	00	No nos interesa este valor
13	Numero de endpoints	1	00	Sin contar el de control, 0 endpoints
14	Clase del interfaz	1	00	No nos interesa
15	Subclase del interfaz	1	00	No nos interesa
16	Protocolo del interfaz	1	00	Idem
17	Índice del interfaz	1	00	No hay cadena descriptiva

Por tanto, esta instrucción envía uno de estos dos descriptores al PC.

SET ADDRESS

Su formato es el siguiente:

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

Cuando el módulo lee el código de esta instrucción, lee la dirección 2 (byte menos significativo de *wValue*), donde viene la dirección a la que tendremos que responder a partir de ahora. Entonces lo primero es terminar la transacción, que es sin datos, según el protocolo indicado en el punto 6.3.5.2, y a continuación cambiar la dirección del dispositivo sacando el nuevo valor por el bus *dev_dir*. A partir de ahora sólo contestaremos a la nueva dirección.

NOTA: Inicialmente el dispositivo debe contestar a la dirección 0, que es la dirección por defecto. En el proceso de configuración se le asignará una nueva dirección. Así, inicialmente todos los dispositivos responden a la dirección 0.

SET CONFIGURATION

El formato de esta instrucción es como sigue:

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

Cuando llega esta instrucción, el dispositivo no hace absolutamente nada, ya que no tiene que adoptar ninguna medida especial para esta configuración, que además es la única. Por eso ni se molesta en leer el valor de la configuración que viene en el campo *wValue*, ya que ya sabe que es el 2 (ver el descriptor). Por tanto, lo único que hace es terminar el proceso de la transferencia sin datos, poniendo *en_status* a '1' y esperando a que *status_stage* sea '1'.

6.4.1.2. Instrucciones del fabricante

Como vimos anteriormente, se han implementado dos instrucciones adicionales: **lee datos** y **escribe datos**. Los códigos de instrucción son respectivamente 20 y 19 (en decimal).

LEE DATOS

El objetivo de esta instrucción es que el PC lea los datos que hay en de 1 a 4 direcciones, que pueden ser de RAM, como en el ejemplo que se implementa, pero que también podrían ser registros o cualquier otro dispositivo de almacenamiento. Las direcciones a leer van en los campos *wValue* y *wIndex* de la instrucción, rellenando con 0 las partes que no necesitemos. Por ejemplo, si sólo queremos leer la posición 3, *wValue* valdría 03 00 y *wIndex* = 00 00. Si quisiéramos leer las direcciones 3, 6 y 39, entonces *wValue* valdría 03 06 y *wIndex* = 27 00 (hex). El numero de bytes que se devuelve es el doble que el de direcciones leídas, ya que se devuelven en formato dirección – dato – dirección – dato ..., ocupando cada dirección y cada dato un byte. Así, en el primer ejemplo anterior, los datos devueltos serían 2 bytes, el primero un 3 (la dirección leída), y el segundo el valor que se ha leído en esa dirección. En el segundo ejemplo se devolverían 6 bytes. El dispositivo puede saber el número de lecturas porque el campo *wLength* lleva el número de bytes a devolver.

Cuando *USBdevice* detecta que ha llegado esta instrucción, en primer lugar lee la posición 6 (campo *wLength*), donde se indica el número de bytes que tiene que devolver el dispositivo. Le da este valor al transmisor y al módulo de aplicación encargado de hacer las lecturas, y llama a éste módulo (da un pulso en la línea *orden_app_rd*). Cuando el módulo termina (da un pulso en *app_ok_rd*), ya están los datos listos en el búfer de transmisión para enviárselos al PC. Por lo tanto, *USBdevice* activa *en_ctrl_rd* para que empiece la fase de datos y luego termina la transferencia de lectura. En la fase de datos sólo hay una transacción, ya que el máximo número de bytes que se pueden devolver es 8 (4 direcciones leídas).

ESCRIBE DATOS

Esta instrucción implementa la escritura de datos, es decir, trasvase de datos en dirección PC → dispositivo USB. Lo que se haga después con esos datos ya es cosa del nivel de aplicación. En este caso, no se usan los campos *wValue* ni *wIndex*, sino que las direcciones y datos a escribir se proporcionan en la fase de datos de la transferencia. Cuando se llega a la fase de datos, el nivel físico escribe los datos que le llegan en el búfer de recepción de datos, entre las posiciones 16 y 23 de la RAM. A partir de aquí, el módulo *USBdevice* tiene que tomar estos datos y procesarlos de forma adecuada. Los datos se almacenan en la RAM en la forma dirección – dato – dirección – dato ..., ocupando cada dirección y cada dato un byte. Por ejemplo, para escribir en la dirección 7 un 45 y en la dirección 100 un 0, se transfiere por el bus y se almacena en la RAM la siguiente secuencia: 7 45 100 0, donde cada número es un byte. Por tanto, las direcciones van de 0 a 255, por lo que se pueden direccionar hasta 256 posiciones, registros, etc. Los datos no tienen ningún formato, luego son valores entre 0 y 255, pero que pueden representar cualquier otra cosa.

Por otro lado, hay que decir que en una misma transferencia el número máximo de bytes que se pueden transmitir es de 255. Como el formato es dirección-dato, el número de bytes que se transmite es par, por lo que el máximo es 254. Esto supone 32 transacciones OUT en la fase de datos de una transferencia de escritura. El mínimo número de bytes que se pueden transmitir es 2, es decir, una escritura.

Cuando llega esta instrucción al dispositivo, el módulo *USBdevice* lee el campo *wLength* de la instrucción para saber la longitud total de la transferencia. Si es mayor que 8, la instrucción se ejecutará por segmentos de 8 bytes, que es la cantidad que vendrá en cada transacción. Así, activa la señal *en_ctrl_wr* para dar paso a la fase de datos y espera a que llegue el primer grupo de bytes. Cuando llegan (se pone *data_stage* a '1'), da la orden al nivel de aplicación de procesarlos (activa *orden_app*) y espera a que el procesamiento termine. Cuando el nivel de aplicación termina (pone *app_ok* a '1'), ya se pueden machacar los datos de la RAM, por lo que se vuelve a poner *en_ctrl_wr* a '1' para volver a recibir datos y así sucesivamente hasta que termina la transferencia. Entonces asiente la fase de estado y vuelve al reposo.

Ya hemos visto todas las instrucciones implementadas en el módulo *USBdevice*. Es importante no tocar las instrucciones estándar, ya que de ellas depende el funcionamiento del sistema (si no se configura bien, no funcionará). Sin embargo, las órdenes del fabricante sí se pueden manipular para adaptarlas a las necesidades de cada usuario, haciendo los correspondientes cambios en el software para el dispositivo.

También se pueden añadir instrucciones nuevas, tanto estándares como del fabricante. Es tan sencillo como añadir una entrada para la nueva instrucción en el estado *lee_inst* de este módulo y escribir su correspondiente rutina. Las que ya hay implementadas deben servir como modelos de cómo implementar los tres tipos de transferencias de control.

6.4.2. Proceso de configuración

Para terminar con el nivel lógico, vamos a ver qué es lo que pasa cuando se enchufa el dispositivo al PC a través del cable USB. En primer lugar, el controlador del bus detecta un cambio en la impedancia de la línea, gracias a las resistencias que tiene el dispositivo a la entrada. Además sabe si el dispositivo conectado es “low-speed” o “full-speed”, ya que el primero cambia la impedancia de la línea D – , y los “full-speed” cambian la de la línea D+.

Una vez detectada la presencia del dispositivo, se informa al driver USBD, del sistema operativo. Éste ordena enviar un reset al dispositivo. Tras el reset, el dispositivo está preparado para responder a la dirección 0. Se le envía la orden GET DESCRIPTOR, pidiendo el “device descriptor”. Sólo se leen los primeros 8 bytes. Esto se hace para conocer el tamaño máximo de paquetes que soporta el dispositivo, que es el dato que va en el octavo byte del descriptor.

A continuación se vuelve a resetear el dispositivo, y se envía la orden SET ADDRESS, con una dirección distinta de 0. A partir de aquí, el dispositivo responde a la nueva dirección. La siguiente orden es de nuevo GET DESCRIPTOR, se vuelve a pedir el “device descriptor”, pero esta vez se lee entero.

Luego viene la orden GET DESCRIPTOR, pero esta vez se lee el descriptor de configuración, para saber cómo tratar el dispositivo. Sólo se lee el descriptor de configuración propiamente dicho, es decir, los primeros 9 bytes. Así el PC puede saber cuánto ocupa el descriptor entero y reservar espacio. Por tanto, se vuelve a enviar la instrucción GET DESCRIPTOR, pero esta vez se lee el descriptor de configuración entero.

En este punto, el PC ya sabe cómo tratar al dispositivo y a qué driver de usuario tiene que llamar para que lo maneje. Aquí termina la parte genérica del proceso de configuración. Lo que falta es que una vez que se cargue el driver de usuario, lo primero que haga éste sea enviar la orden SET CONFIGURATION al dispositivo para que sea plenamente funcional.

Por tanto, un resumen del proceso de configuración es el siguiente:

- Reset
- GET DEVICE DESCRIPTOR. Lee solo 8 bytes.
- Reset
- GET DEVICE DESCRIPTOR. Lo lee entero.
- GET CONFIGURATION DESCRIPTOR. Lee 9 bytes.
- GET CONFIGURATION DESCRIPTOR. Lo lee entero.

Fin del nivel lógico.

6.5. Nivel de Aplicación

Como se dijo en la introducción al principio de este capítulo, este nivel ya no pertenece a USB, sino que representa la funcionalidad que tenga el dispositivo. Aunque para poder comunicarse con el PC, tiene que hacer uso de los niveles funcionales de USB. Por tanto, el nivel de aplicación no es objeto de este proyecto, y el que se ha implementado en nuestra solución es muy básico y sólo sirve como ejemplo de comunicación entre el dispositivo y una aplicación corriendo en el ordenador.

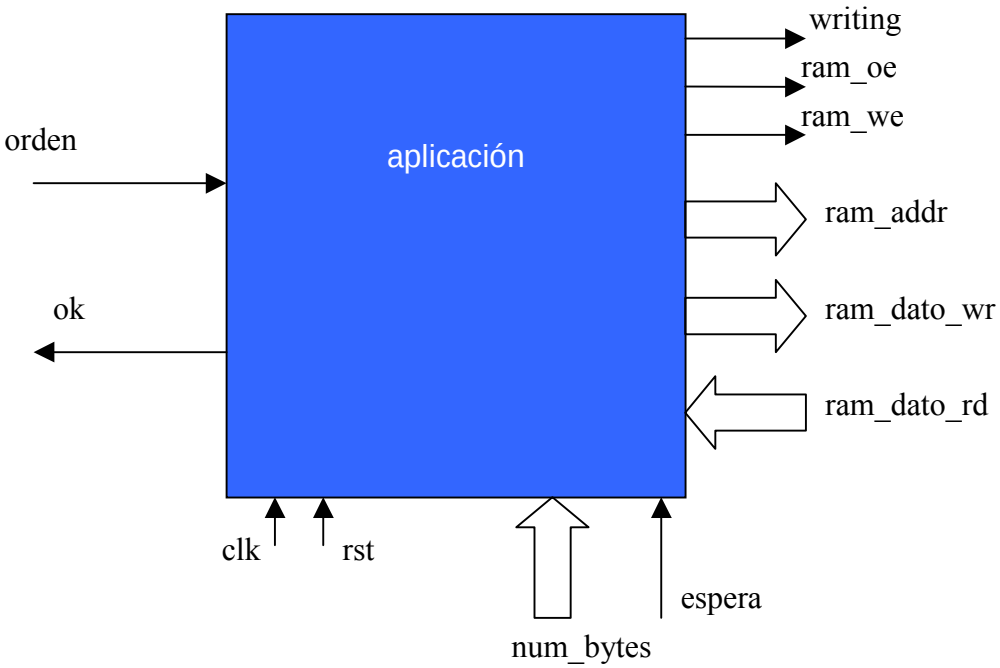
La función de los dos módulos implementados en este nivel es escribir y leer datos provenientes de la aplicación en un búfer de memoria. Este búfer se constituye a partir de 256 posiciones de memoria en la RAM que trae la tarjeta XSV800, en concreto entre las posiciones 256 y 511 de la RAM. Para la aplicación, la posición 256 de la RAM constituye la dirección 0 del buffer, y la 511 se representa por la dirección 255. Los datos y direcciones a escribir vienen por bytes, alternándose una dirección y un dato (cada uno de un byte, por eso van de 0 a 255). Este formato es el mismo en el que hay que enviar las lecturas a la aplicación en el PC.

Como hemos dicho en el párrafo anterior, este nivel está constituido por dos módulos. Uno realiza las lecturas, y otro las escrituras. Vamos a verlos:

6.5.1. Módulo de escritura

Este módulo se implementa en el fichero *aplicacion.vhd*, y su interfaz es como se ve en la página siguiente. Su función es realizar escrituras en el búfer.

Las líneas *orden* y *ok* están conectadas al módulo *USBdevice*, en concreto a las señales *orden_app* y *app_ok* respectivamente. Así, cuando se activa la línea *orden*, lee el numero de bytes indicado en *num_bytes* (*num_bytes* proviene de *USBdevice* y va a *director*, pero la podemos aprovechar también para comunicar el número de bytes a procesar a los módulos de aplicación) del búfer de recepción (posiciones 16 a 23 de la RAM). Los lee de dos en dos. En cada grupo de dos bytes, el



primero representa una dirección (de 0 a 256) y el segundo un valor a escribir en esa dirección. Así, tras leer los dos bytes, ejecuta la operación solicitada (escribe el segundo byte en la dirección indicada por el primer byte, en el búfer de la aplicación. Cuando procesa todas las parejas de bytes, activa la línea *ok*.

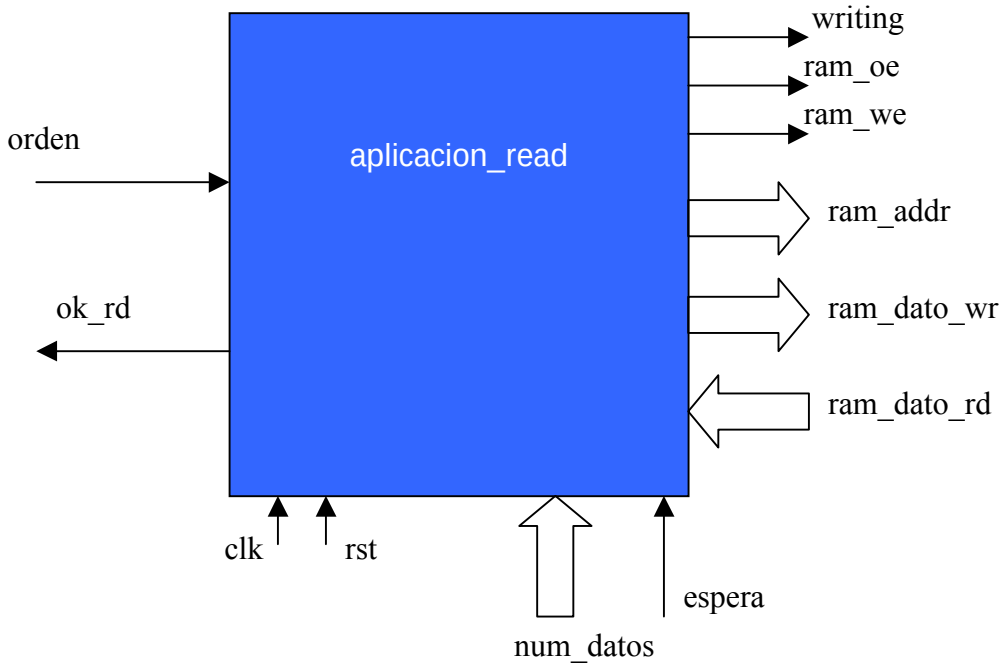
Si en el transcurso del proceso se activa la línea *espera*, quiere decir que hemos perdido el control del bus de la RAM, por lo que la operación se retrasa hasta que la RAM se libere (*espera* está conectada a la señal *transmite*; en transmisión, el control de la RAM lo tiene el transmisor).

6.5.2. Módulo de lectura

De nuevo, su interfaz se ve en la página siguiente. Este módulo se implementa en el fichero *aplicación_read.vhd*. Su función es leer datos del búfer de aplicación y mandarlos al programa.

Cuando se activa la línea *orden*, que está conectada a *orden_app_rd* en *USBdevice*, realiza entre 1 y 4 operaciones de lectura en el búfer de aplicación. El

número de operaciones a realizar viene indicado en *num_datos* (*num_datos* indica el numero de bytes a devolver, por lo que las operaciones son la mitad de este valor).



En cada operación, se lee de la instrucción recibida (posiciones 0 a 7 de la RAM) la dirección a leer. Recordar que estas direcciones estaban en los campos *wValue* y *wIndex* de la instrucción. A continuación se lee dicha dirección del búfer de aplicación, y se almacena su contenido en el búfer de transmisión, entre las posiciones 8 y 15 de la RAM. También se guarda en el búfer de transmisión la dirección leída, de forma que los bytes se organizan de la forma dirección - dato. Cuando se leen todas las posiciones requeridas, se activa la línea *ok_rd* (conectada a *app_ok_rd* del nivel lógico). En este momento el búfer de transmisión está listo para que el módulo transmisor lo envíe al PC.

NOTA: Este módulo, al contrario que el de escritura (en *aplicacion.vhd*), no requiere de una señal de espera que retenga su operación mientras que otro módulo accede a la RAM, ya que en este caso no se dará esta situación.

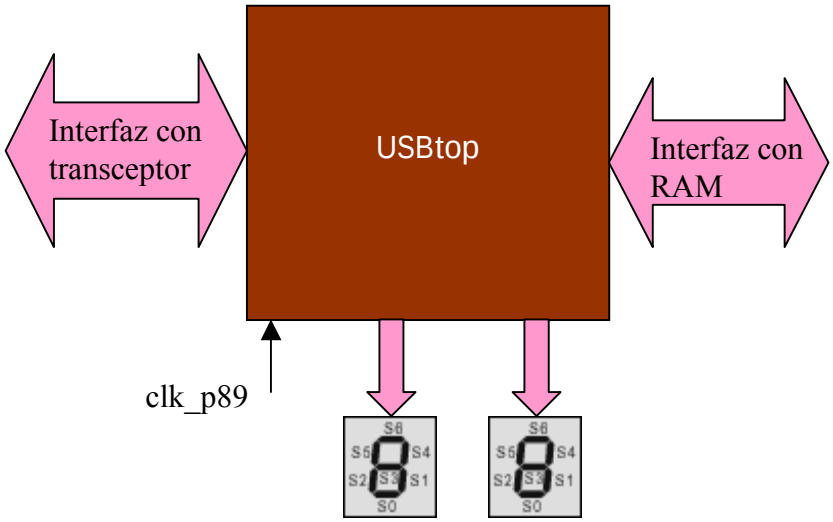
Fin del nivel de aplicación.

6.6. Implementación del dispositivo USB

Ya hemos visto todo el diseño en VHDL. En este capítulo vamos a ver la estructura del fichero *usbtop.vhd*, en donde se instancian todos los componentes, y posteriormente veremos los requisitos para la implementación y síntesis del diseño sobre la tarjeta XSV800.

6.6.1. USBtop

El componente *USBtop*, que es el nivel superior de la jerarquía de componentes de este diseño, se implementa en el fichero *usbtop.vhd*. Sus conexiones con el exterior son las siguientes:



El interfaz con el transceptor (sea cual sea, en el caso de la XSV800 es el Philips) está formado por las líneas para el control del bus, y junto con el reloj son las únicas conexiones imprescindibles del diseño. Estas señales son *rcv*, *vp* y *vm* de entrada (aunque la primera no se usa), y *vp0*, *fse0*, *oe* y *suspnd* de salida. Todas vienen del módulo *adaptador*.

El reloj se toma del exterior, y debe ser de 12 MHz. Como ya dijimos, no nos vale el del oscilador que trae la XSV800; por tanto, hay que tomar uno externo. Esto se

explicará en el siguiente capítulo. En todo caso, en este diseño el reloj entra por la patilla 89 de la FPGA Virtex 800.

El interfaz con la RAM está formado por las líneas *ce*, que viene del módulo *usb2RAM*, *we*, *oe_RAM*, *ADDR* y *dato*, ésta última definida como INOUT. Estas líneas se gestionan en el módulo *USBtop*, asignándolas a cada módulo cuando las necesite y si no, poniéndolas en alta impedancia. No hay conflictos entre los distintos módulos porque cada uno accede a la RAM en un tiempo distinto, luego no se superponen los accesos. El código del módulo *USBtop* para esto es el siguiente:

```
dato_lec <= dato;                -- Dato de la RAM en lecturas

dato <= dato_wr_1 when esc_1='1' else      --Dato a la RAM en escrituras; las líneas
        dato_wr_2 when esc_2='1' else      --"esc_n" son las señales "writing" de cada
        dato_wr_4 when esc_4='1' else      --módulo según la numeración que se
        dato_wr_5 when esc_5='1' else      --muestra abajo.
        "ZZZZZZZZ";

ADDR <= ADDR_1 when esc_1='1' else                        -- usb2ram    (1)
        ADDR_2 when (esc_2='1' or oe_ram_2='0') else      -- USBdevice     (2)
        ADDR_3 when oe_ram_3='0' else                     -- transmisor (3)
        ADDR_4 when (esc_4='1' or oe_ram_4='0') else      -- aplicacion_read  (4)
        ADDR_5 when (esc_5='1' or oe_ram_5='0') else      -- aplicacion_write (5)
        "ZZZZZZZZZZZZZZZZZZZZ";

we <= we_1 when esc_1='1' else
we_2 when esc_2='1' else
we_4 when esc_4='1' else
we_5;

oe_ram <= (oe_ram_2 and oe_ram_4 and oe_ram_3 and oe_ram_5);
```

Como hemos dicho, el acceso a la RAM no es imprescindible. Se podrían usar registros internos o incluso implementar una pequeña memoria RAM en VHDL a base de registros. Ya se vio cómo reducir el uso de la RAM en el apartado 6.2.

Por último, nos quedan las líneas que controlan los dos displays de 7 segmentos que trae la tarjeta XSV800. El uso de estos displays no es en absoluto necesario, simplemente se han implementado para animar un poco la tarjeta. Muestran, en hexadecimal, el valor de un registro de 8 bits que se implementa en *USBtop*, y que copia el valor de la posición 1 del búfer de aplicación (dirección 257 de la RAM). La salida de este registro se manda a los displays a través de dos módulos *bin2led*, que convierten la señal de binaria a 7 segmentos. Así, en todo momento vemos el valor de esta posición de memoria. Si escribimos en ella, este registro captura el nuevo valor y lo muestra por los displays. Podemos usar esto para comprobar el funcionamiento de las escrituras. El código es:

```
p1: process(registro,we_5,ADDR_5,dato_wr_5)
begin
        if (we_5='0' and ADDR_5(7 downto 0)="00000001") then
                p_registro<=dato_wr_5;
        else
                p_registro<=registro;
        end if;
end process;
```

En *USBtop* también se implementa directamente el multiplexor que selecciona las líneas de entrada al módulo *crc16*. Selecciona entre las entradas del receptor y las del transmisor.

6.6.2. Síntesis e implementación. Generación del fichero usb.bit

Ya tenemos el código VHDL completo. Vamos a ver cómo implementarlo para descargarlo a la FPGA Virtex 800, que es la que hemos usado en el proyecto. En primer lugar, ejecutamos el programa "PCM.exe" de Xilinx, y creamos un proyecto nuevo (supongamos que el nombre es usb). Añadimos los siguientes ficheros como código fuente:

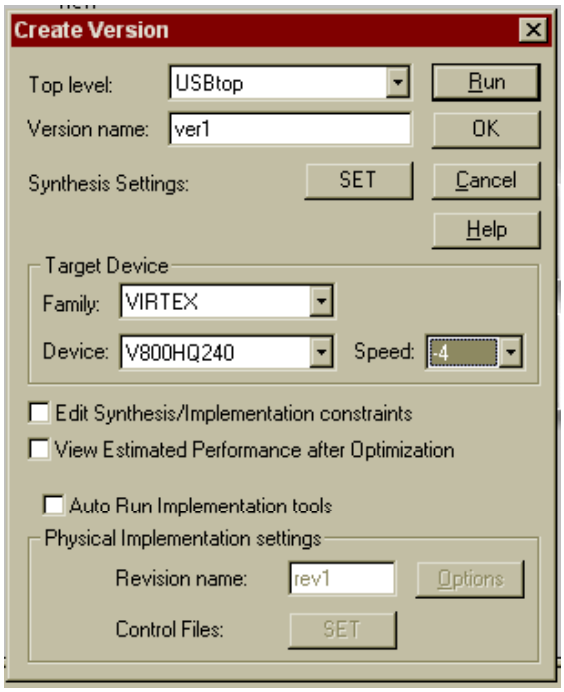
- ✓ Adaptador.vhd
- ✓ Aplicacion.vhd
- ✓ Aplicacion_read.vhd
- ✓ Bin2led.vhd
- ✓ Bit_stuffing.vhd
- ✓ Carga.vhd
- ✓ Celda_crc.vhd
- ✓ Contador.vhd
- ✓ Crc16.vhd
- ✓ Crc5.vhd
- ✓ Destuffing.vhd
- ✓ Detecta_paquete.vhd
- ✓ Device.vhd
- ✓ Director.vhd
- ✓ Nrzi_decoder.vhd
- ✓ Nrzi_encoder.vhd
- ✓ Reg_desplaz.vhd
- ✓ Reset_gen.vhd
- ✓ Transmisor.vhd
- ✓ Tx_data.vhd
- ✓ Tx_handshk.vhd
- ✓ Usbtop.vhd
- ✓ Usb2ram.vhd

A continuación, generamos un fichero *usb.ucf* que contenga las siguientes restricciones en cuanto al patillaje (esto sólo es válido para la tarjeta XSV800 de XESS):

```
NET "clk_p89" LOC = "p89";
NET "ce" LOC = "p186";
NET "we" LOC = "p201";
NET "oe_ram" LOC = "p228";
NET "dato<0>" LOC = "p202";
NET "dato<1>" LOC = "p203";
NET "dato<2>" LOC = "p205";
NET "dato<3>" LOC = "p206";
NET "dato<4>" LOC = "p207";
NET "dato<5>" LOC = "p208";
NET "dato<6>" LOC = "p209";
NET "dato<7>" LOC = "p215";
NET "ADDR<0>" LOC = "p200";
NET "ADDR<1>" LOC = "p199";
NET "ADDR<2>" LOC = "p195";
NET "ADDR<3>" LOC = "p194";
NET "ADDR<4>" LOC = "p193";
NET "ADDR<5>" LOC = "p192";
NET "ADDR<6>" LOC = "p191";
NET "ADDR<7>" LOC = "p189";
NET "ADDR<8>" LOC = "p188";
NET "ADDR<9>" LOC = "p187";
NET "ADDR<10>" LOC = "p238";
NET "ADDR<11>" LOC = "p237";
NET "ADDR<12>" LOC = "p236";
NET "ADDR<13>" LOC = "p235";
NET "ADDR<14>" LOC = "p234";
NET "ADDR<15>" LOC = "p232";
NET "ADDR<16>" LOC = "p231";
NET "ADDR<17>" LOC = "p230";
NET "ADDR<18>" LOC = "p229";
NET "fse0" LOC = "p17";
NET "oe" LOC = "p12";
NET "sl0_p177" LOC = "p177";
NET "sl4_p145" LOC = "p145";
NET "sl1_p167" LOC = "p167";
NET "sl6_p134" LOC = "p134";
NET "sl2_p163" LOC = "p163";
NET "sl5_p138" LOC = "p138";
NET "sl3_p156" LOC = "p156";
NET "sr0_p124" LOC = "p124";
NET "sr1_p132" LOC = "p132";
NET "sr2_p133" LOC = "p133";
NET "sr3_p139" LOC = "p139";
NET "sr4_p141" LOC = "p141";
NET "sr5_p144" LOC = "p144";
NET "sr6_p147" LOC = "p147";
NET "suspnd" LOC = "p176";
NET "vm" LOC = "p9";
NET "vpo" LOC = "p13";
NET "vp" LOC = "p10";
```

En esta lista podemos observar la patilla que le corresponde a cada entrada/salida.

Este fichero lo copiamos al directorio del proyecto, reemplazando el que ya existía. Con esto ya está todo listo para la implementación y síntesis. No hacen falta restricciones adicionales en cuanto a tiempos, retrasos, etc. Seleccionamos el componente *USBtop* como nivel superior de la jerarquía, e iniciamos la síntesis, seleccionando para ello la FPGA adecuada como se muestra en la figura:



El proceso de síntesis genera unos cuantos "warnings" que podemos ignorar con toda tranquilidad. Tras esto, ejecutamos el proceso de implementación, y obtenemos así el fichero *usb.bit* que contiene nuestro diseño listo para bajarlo a la FPGA. Las instrucciones para poner en marcha la placa las veremos en un apartado posterior.

El resultado de la implementación es el siguiente:

```
Design Summary:
  Number of errors:      0
  Number of warnings:    2
  Number of Slices:                  494  out of  9,408     5%
  Number of Slices containing
    unrelated logic:                   0  out of  494       0%
```

```
  Number of Slice Flip Flops:        468  out of  18,816      2%
  Total Number 4 input LUTs:         771  out of  18,816      4%
    Number used as LUTs:                770
    Number used as a route-thru:        1
  Number of bonded IOBs:              50  out of  166       30%
  Number of GCLKs:                     1  out of  4         25%
  Number of GCLKIOBs:                  1  out of  4         25%
Total equivalent gate count for design:  8,496
Additional JTAG gate count for IOBs:  2,448
```

Como se puede ver, el diseño final tiene 468 biestables y unas 8500 puertas equivalentes, por lo que con una FPGA de 10000 puertas equivalentes tendríamos suficiente para implementarlo. Por otro lado, si tuviésemos que implementar un búfer en VHDL para no usar una RAM externa como en este caso, el tamaño del diseño subiría algo.

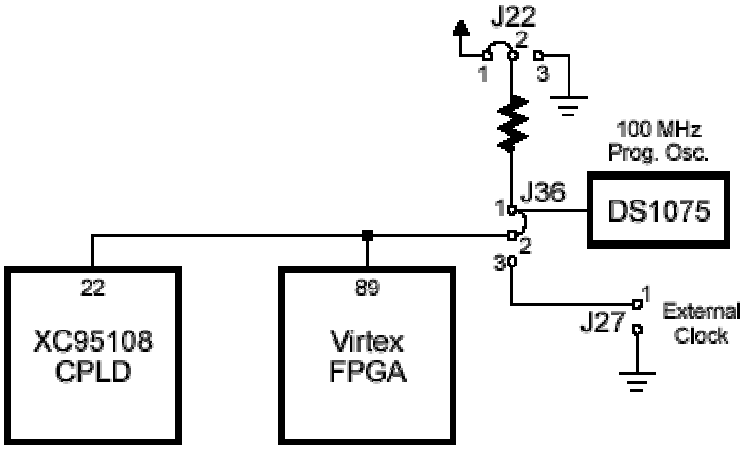
7. Generación de la señal de Reloj

Como se mencionó en el apartado 2.5.2, hay que alimentar la placa XSV800 de XESS con una señal de reloj externa. Esto es debido a que el oscilador interno de la placa nos da frecuencias de valor los divisores de 100 MHz, pero no nos puede dar los 12 MHz que necesitamos en este diseño. Por tanto, la solución elegida ha sido usar una placa con un oscilador del mismo modelo que el de la XSV, un DS1075 de Dallas Semiconductor, pero con una frecuencia base de 60 MHz, lo que nos permite obtener los 12 MHz necesarios al dividir esta frecuencia base por 5. El nombre del componente usado es *DS1075M-060*. La documentación completa de este componente se puede descargar desde http://pdfserv.maxim-ic.com/arpdf/ds1075.pdf.

La solución adoptada es tan simple que sólo requiere hardware: un pequeño circuito impreso con el oscilador. No tenemos que implementar ninguna circuitería adicional o software para la programación del dispositivo, ya que al ser el mismo modelo que el que trae la placa de XESS, podemos utilizar los recursos de ésta, incluido el software, para realizar la programación del oscilador. Es decir, lo que hacemos es que el software de programación crea que está programando el oscilador de la placa, cuando en verdad está programando nuestro oscilador. Esto es muy sencillo de hacer, y se explica a continuación.

7.1. El oscilador DS1075

El componente que trae la placa es el DS1075-100. Su topología es la siguiente:

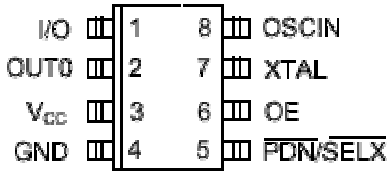


El jumper *J36* conecta el oscilador al resto de la placa cuando la pestaña está en la posición 1-2. Si la pestaña está en la posición 2-3 o no está puesta, el oscilador no alimenta a la FPGA.

Por otro lado, el jumper *J22* selecciona el modo de funcionamiento del oscilador. Si la pestaña está en la posición 1-2, la patilla de entrada/salida del oscilador se conecta a alimentación a través de una resistencia de pull-up. Esto pone al oscilador en modo de programación. Si por el contrario la pestaña se coloca en la posición 2-3, la patilla de entrada/salida queda libre y por ella sale la señal de reloj; éste es el modo de funcionamiento normal.

Pues bien, lo que hacemos nosotros es muy sencillo: se quita la pestaña del jumper *J36*, con lo que desconectamos el oscilador de la placa, y conectamos la señal de entrada/salida de nuestro circuito impreso al pin 2 de *J36*. Con esto tenemos la misma topología de la figura anterior, pero con nuestro oscilador en vez de el de la placa. El jumper de nuestro circuito impreso hace las funciones del jumper *J22*, seleccionando el modo de funcionamiento.

Antes de ver el circuito impreso, vamos a explicar el patillaje del oscilador: es un encapsulado de 8 patas, como se ve a continuación:



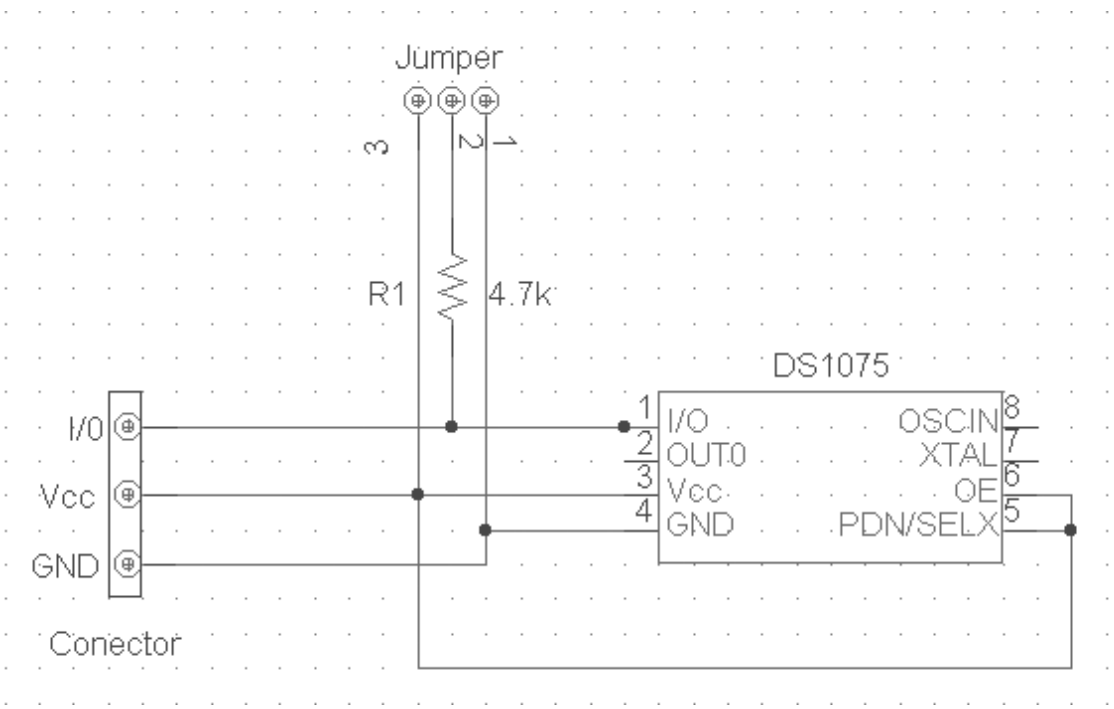
Las patas son las siguientes:

- *I/O*: Es la señal de entrada/salida. En modo de programación es entrada digital de datos, y en modo normal es salida, es la señal de reloj que alimenta todo nuestro circuito.
- *OUT0:* Señal de salida que no usamos, por lo que en el circuito impreso esta salida no se conecta a nada, se deja abierta.
- *Vcc:* Alimentación del oscilador. Debe ser 5V, y en nuestro caso se toma de la placa de desarrollo, de cualquiera de los pines de alimentación de los buses de expansión.
- *GND:* Entrada de tierra. También se conecta a cualquier pin GND de los buses de expansión de la tarjeta de XESS.
- *PDN/SELX:* (Power Down / Select Function). Este pin se pone a '1' para seleccionar el oscilador interno del componente. Si fuese '0', habría que colocar un oscilador de cuarzo entre las patas *OSCIN* y *XTAL*.
- *OE:* Output Enable. Esta entrada debe estar puesta a '1' para que a la salida haya señal de reloj.
- *XTAL:* Este pin no se conecta a nada, es para introducir un oscilador externo. En nuestro diseño esta patilla se deja abierta.
- *OSCIN:* igual que el anterior.

7.2. Nuestro circuito de reloj

Por tanto, nuestra tarjeta impresa debe hacer lo siguiente: proveer conectores para las patillas 1, 3 y 4, de forma que podamos conectar éstas (I/O, Vcc y GND) a la tarjeta de desarrollo, poner a Vcc las patillas 5 y 6, y dejar abiertas el resto (2, 7 y 8). También debe proporcionar la resistencia de pull-up para la programación, así como un

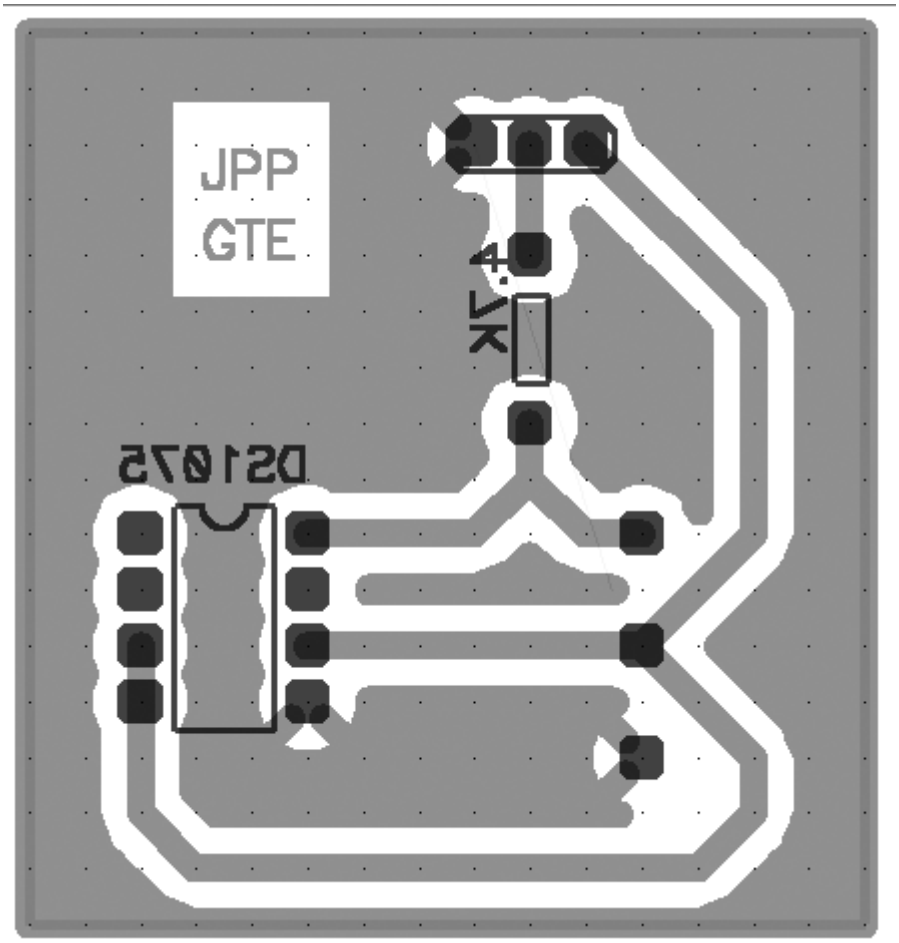
jumper para seleccionar entre modo normal y modo de programación. Por tanto, el esquemático resultante es el siguiente:



Si ponemos la pestaña en la posición 1-2 del jumper, la resistencia queda colocada entre la línea I/O y tierra, por lo que el oscilador está en modo normal y saca por la línea I/O el reloj. Si por el contrario la pestaña está en 2-3, la resistencia está entre I/O y Vcc = 5V, por lo que actúa como pull-up y el oscilador está en modo programación. En este modo, la línea I/O es de entrada y se usa para programar el divisor de frecuencia.

NOTA: Sólo se puede cambiar la posición del jumper con el oscilador desconectado de la alimentación.

La placa impresa que implementa este circuito se ha desarrollado con el programa Accel Tango PCB, y el PCB resultante, que está contenido en el fichero *dallas01.pcb* que se adjunta con esta documentación, es el siguiente:



Este PCB es para una placa con cobre por una sola cara. En esta cara (la de abajo) se plasma este circuito, y en la de arriba irían los componentes.

Las tres señales de salida, I/O, Vcc y GND, se sacan al exterior a través de un conector. En el PCB, están representadas por los tres pads que se ven a la derecha. El de arriba es I/O, el del medio es Vcc y el de abajo, GND. Por tanto, sólo nos queda explicar cómo usar esta placa.

7.3. Programación de la frecuencia de reloj

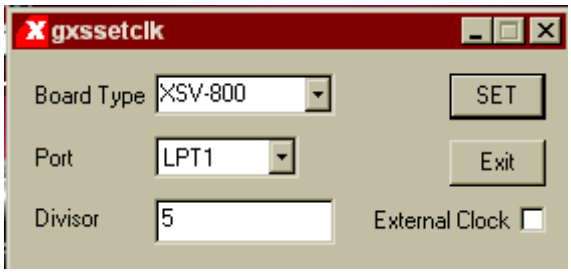
Este procedimiento será necesario para fijar el divisor programable del oscilador con el valor 5, de forma que la frecuencia de salida sea de 12 MHz. Dado que el valor del divisor es no volátil, solo hace falta realizar este proceso una vez.

1. En primer lugar, conectar el circuito impreso a la placa de desarrollo. Para ello, quitar la pestaña del jumper *J36* y conectar la salida I/O del circuito impreso al pin 2 de este jumper. Esto anula el oscilador de la tarjeta y conecta el nuestro. También hay que alimentar el circuito impreso. Para ello, conectar las señales Vcc y GND a cualquiera de los pines de los puertos de expansión que indiquen 5V y GND respectivamente.

2. A continuación, poner una pestaña en la posición 2-3 del jumper del circuito impreso. Esto selecciona el modo de programación.

3. Alimentar la tarjeta de desarrollo y conectar al PC mediante el cable paralelo.

4. Ejecutar el programa *gxssetclk.exe*. Rellenar los recuadros como se indica en la imagen. Pulsar *SET* . Esto lleva a cabo la programación. Cuando termine, pulsar *EXIT* para salir del programa.



5. Quitar la alimentación y el cable paralelo de la tarjeta de desarrollo.

6. Poner la pestaña en la posición 1-2 del circuito impreso. Esto devuelve el oscilador al modo normal. Al volver a alimentar la placa, el oscilador ya funciona normalmente, con una señal de 12 MHz. Esta señal llega a la FPGA a través de la patilla 89 de ésta.

7.4. Uso normal del circuito impreso

Una vez el oscilador ha sido programado, el uso de la tarjeta es muy sencillo:

1. Conectar el circuito impreso a la placa de desarrollo. Para ello, conectar la salida I/O del circuito impreso al pin 2 del jumper *J36* de la placa, y las señales Vcc y GND a cualquiera de los pines de los puertos de expansión que indiquen 5V y GND respectivamente. Asegurarse que la pestaña del jumper del circuito impreso está colocada en la posición 1-2, esto es, conecta la resistencia a GND.

2. Alimentar la tarjeta de desarrollo y conectar al PC mediante el cable paralelo.

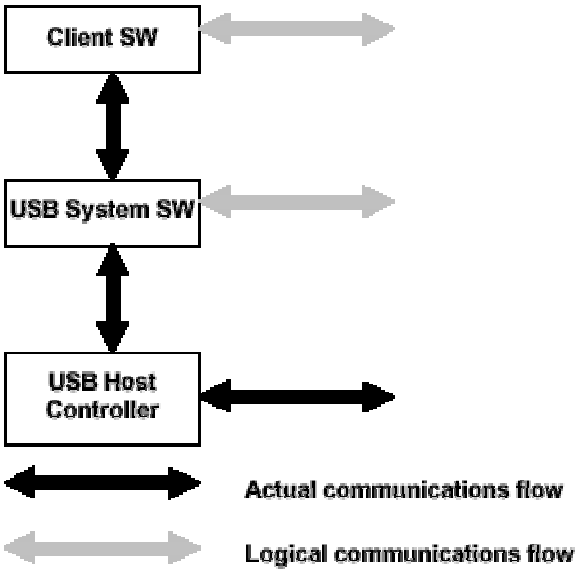
3. Usar la placa normalmente. Ésta está siendo alimentada por nuestro oscilador con una frecuencia de 12 MHz.

8. Solución final. Software

En este capitulo vamos a ver el software que se ha desarrollado para este proyecto. Lo primero es recordar que este software va a correr bajo Windows 98, luego éste será nuestro entorno de trabajo. En primer lugar veremos cómo gestiona Windows 98 el bus USB, y a continuación, expondremos los distintos aspectos de la solución. Por último, veremos cómo instalar y ejecutar correctamente este software.

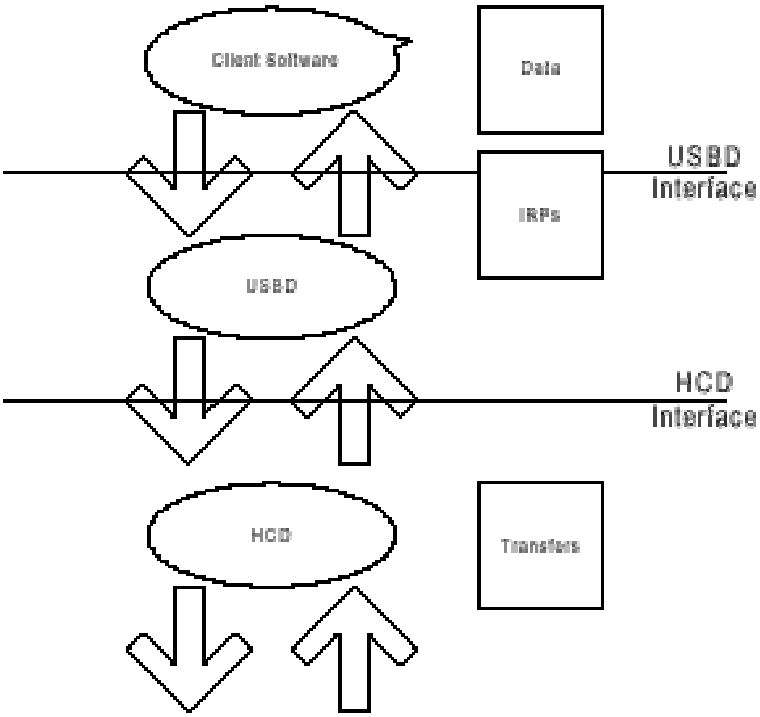
8.1. La pila de drivers USB

Ya vimos en el apartado 4.1 cómo se organizaba el software en el PC para llevar a cabo un proceso de comunicación a través del USB. El esquema era el siguiente:



Esta es la parte del PC (host). Únicamente los dos recuadros superiores son de software, el de abajo es el hardware del PC. El recuadro *USB System Software* representa el software perteneciente al sistema operativo, que está formado por los drivers *HCD* (Host Controller Driver) y *USBD* (USB driver), y el recuadro *Client*

Software representa el software del usuario, en este caso nosotros, por lo que ésta es la porción de software que tendremos que desarrollar. Antes de pasar a nuestra solución, hagamos un recordatorio del funcionamiento de los drivers del sistema operativo:



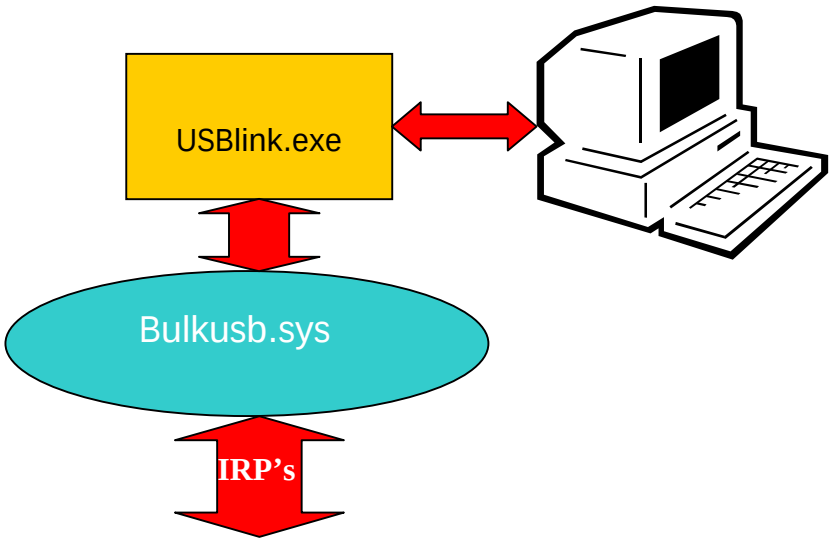
Software de usuario: Este software está compuesto por los drivers y aplicaciones que son específicos para la comunicación con un dispositivo USB concreto, por lo que también se le conoce como "device drivers" (drivers de dispositivo). Su función es enviar peticiones de transferencia de datos al driver USB (USBD). Estas peticiones son estructuras de datos llamadas IRP's ("I/O Request Packets", paquetes de petición de entrada/salida). Estas IRP's inician transferencias a / desde el dispositivo USB objetivo de la transacción. El software de usuario debe proporcionar el búfer de memoria en donde se almacenan los datos a enviar o recibidos, y no tiene conocimiento de los mecanismos de transferencia serie de USB. Cuando la transferencia termina, el USBD lo notifica devolviendo la IRP completada. Por tanto, el mecanismo de comunicación entre el software de usuario y el sistema operativo / bus USB es a través del intercambio de IRP's. Esto es lo que habrá de hacer el software diseñado para nuestro proyecto.

USB Driver (USBD): El USBD conoce las capacidades de comunicación del dispositivo USB objetivo de una transferencia, ya que es el receptor de los descriptores que se intercambian en el proceso de configuración. Por tanto, sabe cómo hay que comunicarse con el dispositivo. Cuando recibe una IRP del driver de dispositivo, el USBD organiza la petición en transacciones individuales según esta información que tiene del dispositivo, por lo que adapta la transferencia a las capacidades de éste. El USBD pasa estas nuevas transacciones al HCD para que las lleve a cabo.

Host Controller Driver (HCD): Este driver organiza todas las transacciones que se tienen que llevar a cabo para todos los dispositivos y se las pasa al host controller para que las ejecute. Por tanto, su función es controlar el hardware del host para ejecutar las transferencias necesarias entre el PC y todos los dispositivos conectados al bus.

8.2. Arquitectura del software de usuario

En este punto vamos a explicar la estructura adoptada para el software de usuario, que es el foco de nuestro proyecto en la vertiente del PC. Como hemos mencionado en el punto anterior, el objetivo es intercambiar IRP's con el driver USB para realizar el trasvase de información con nuestra tarjeta. Para ello, nuestra solución consta de un driver de usuario, cuyo ejecutable es el fichero *bulkusb.sys*, que se carga cuando se enchufa la placa al bus, y que gestiona el intercambio de IRP's, y de una aplicación de usuario, *usblink.exe*, que genera la información a enviar a la placa o que la recibe, y que interactúa con el usuario. El esquema, por tanto, es el siguiente:



Dado el carácter genérico de este proyecto, el foco principal es el driver *bulkusb.sys*, que se entrega en una versión definitiva que no es necesario modificar. En cambio, la aplicación *USBLink* se entrega a modo de ejemplo ilustrativo del manejo del driver. De ésta aplicación, que no hace nada útil, sólo nos interesa cómo gestiona el intercambio de información con nuestra tarjeta de desarrollo. Para ello, se comunica con nuestro driver de usuario a través de peticiones de lectura y/o escritura como si estuviera escribiendo o leyendo de un fichero corriente.

Mientras que la aplicación se puede construir con cualquier compilador de C o C++, el driver se ejecuta en modo "kernel", por lo que no nos vale con un compilador cualquiera. En nuestro caso, como el entorno objetivo es Windows 98, se ha empleado para la compilación el paquete "Windows 98 DDK" (Driver Development Kit), que proporciona las herramientas necesarias para ello. Por esta razón es conveniente usar directamente el driver *bulkusb.sys* tal como se entrega con el proyecto, y realizar cualquier aplicación conforme al interfaz que presenta este driver y que a continuación expondremos. Es decir, para usar este interfaz USB para un propósito concreto, se recomienda usar este driver con una aplicación específica que cree cualquier usuario.

8.3. El driver "bulkusb"

Como se ha mencionado anteriormente, el fin de este driver es proporcionar los medios para efectuar trasvases de información entre nuestro dispositivo USB y una aplicación de usuario. Por tanto, debe presentar dos interfaces: por un lado se comunica con dicha aplicación, y por el otro con la pila de drivers USB. Tanto en un interfaz como en el otro, todo el intercambio de información se realiza a través de IRP's (I/O Request Packets), que son estructuras de datos que contienen toda la información necesaria para el intercambio de los datos de usuario. El proceso es el siguiente:

1. Cuando la aplicación de usuario necesita leer o escribir datos en el dispositivo USB, hace llamadas a *ReadFile()* y *WriteFile()*, que son rutinas genéricas para la entrada / salida de datos del programa. Por tanto, el programa de usuario desconoce las características de la transmisión a través de USB. En esta llamada se pasan todos los datos necesarios para la transferencia, como son la dirección de los búferes de entrada y/o salida o la cantidad de datos a transferir.

2. El sistema operativo (en este caso Windows 98) empaqueta estos datos en una IRP y se los pasa a nuestro driver. Si la aplicación llamó a *ReadFile()*, la IRP que recibe el driver es del tipo "IRP_MJ_READ" (pide una lectura), mientras que si llamó a *WriteFile()*, la IRP es del tipo "IRP_MJ_WRITE" (para escrituras). Así, nuestro driver sabe la operación a realizar.

3. El driver *bulkusb.sys* procesa esta IRP con el conocimiento que tiene de nuestro dispositivo USB, y genera una nueva IRP que pasará a la pila de drivers USB. A continuación espera a que los niveles inferiores lleven a cabo la transferencia.

4. Cuando los niveles inferiores terminan, devuelven al driver la IRP completada (con la información leída si era una lectura, o con el número de bytes escritos si era una escritura). El driver debe ahora realizar la comprobación de errores (si realiza alguna), y completar la IRP que le envió la aplicación de usuario. Ésta recibe de esta manera la confirmación de la transferencia y prosigue su curso normal.

Hasta aquí se ha esbozado cómo realiza el driver el intercambio de información, que es su objetivo principal. Sin embargo, *bulkusb.sys* realiza otras funciones comunes a todos los drivers de Windows, como son la gestión de la energía y la gestión del sistema "Plug and Play" (PnP). Para estas funciones también se intercambian IRP's con el sistema operativo. Aunque en este capitulo trataremos estos puntos, no lo haremos tan a fondo como con la transferencia de datos. Para más información sobre estos aspectos, véanse los ejemplos que vienen en la documentación del paquete "Windows 98 DDK". Conviene indicar que este driver está ampliamente basado en uno de estos ejemplos, por lo que su lectura puede ser recomendable. Dicho ejemplo se encuentra, dentro del directorio raíz del DDK, en el directorio \src\usb\bulkusb.

8.3.1. Código fuente

Las rutinas que componen nuestro driver se encuentran repartidas en diferentes ficheros, según su funcionalidad. Estos ficheros son los siguientes:

- **Bulkusb.c** Contiene la rutina *DriverEntry*, que es la que se ejecuta al cargar el driver y que construye la tabla de encaminamiento de las IRP's. Esta tabla indica la rutina que hay que ejecutar cuando llegue una IRP determinada. Con esto se evita tener que escribir un bucle infinito que consuma todo el tiempo de CPU. Este fichero también contiene las rutinas que se encargan de enviar IRP's a los niveles inferiores de la pila USB.

- **BulkPwr.c** Contiene las rutinas que procesan las IRP's de gestión de energía.

- **BulkPnp.c** Contiene las rutinas que procesan las IRP's de Plug and Play.

- **IoctlBlk.c** Contiene rutinas para procesar IRP's de gestión del dispositivo, como pueden ser los resets.

- **OcrBlk.c** Contiene el código que más nos interesa, es decir, el que gestiona las IRP's de entrada / salida de datos.

- **BusbDbg.c** Contiene las funciones empleadas para depuración del programa.

A estos ficheros los acompañan los siguientes archivos de cabeceras:

- **Blk82930.h** Define diversas estructuras de datos, como son el "device extensión" (contiene información sobre el estado del dispositivo USB a lo largo del proceso) y BULKUSB_RW_CONTEXT (que guarda información sobre las IRP's de transferencia que se han enviado y esperan respuesta). También se incluyen los prototipos de todas las funciones.

- **Bulkusb.h** define diversos códigos que identificarán a las IRP's que enviemos (IOCTL's).

- **Guid829.h** define el GUID (globally unique identifier), que es la constante que se empleará para enlazar la aplicación de usuario con este driver. Cuando el programa de usuario quiera abrir el dispositivo USB para hacer transferencias de datos, dará este código, que lo relacionará con este driver y por tanto con nuestro dispositivo USB.

- **Busbdbg.h** contiene diversas definiciones empleadas en rutinas de depuración.

8.3.2. Funcionamiento del driver

En los siguientes apartados vamos a describir el funcionamiento de nuestro driver, haciendo especial énfasis en la gestión de las transferencias de datos. Aunque no se mencionarán todas las funciones que incluyen los ficheros de código, sí nos centraremos en las más importantes. Conviene resaltar aquí que algunas de estas funciones están pensadas para dispositivos con más de un "endpoint", por lo que no se emplean en esta implementación concreta del driver. Otras funciones tienen parte de su código encerrado entre símbolos de comentarios. Esto es por la misma razón: esas secciones de código están pensadas para dispositivos más complejos y no son necesarias en nuestro caso. No me ha parecido bien quitarlas pensando en futuras ampliaciones del enlace USB. En caso de que fuese necesario acudir al código del ejemplo inicial, éste se encuentra en el directorio antes mencionado en el DDK.

Muchas de las funciones a las que se llama en el código (todas las que no empiezan por BulkUsb_) no se definen en estos ficheros, sino que forman parte del interfaz que presenta el USBD (driver USB). Toda la información relativa a estas funciones (parámetros, valores devueltos, etc.) se puede consultar en la documentación del DDK. Ejemplos de este tipo de funciones son *IoGetCurrentIrpStackLocation* o *IoCallDriver*.

8.3.2.1. Carga del driver

Cuando el Sistema Operativo decide cargar el driver, llama a la rutina ***DriverEntry*** (en *Bulkusb.c*), que se encarga de asignar una rutina a cada tipo de IRP. Así, cuando al driver le llegue una IRP nueva, ésta será procesada por la rutina que se indique en la tabla creada por esta función. Esta tabla se guarda en el objeto *DriverObject* (nuestro driver), objeto que se coloca en la pila de drivers USB. También se fija una rutina para la llamada *AddDevice*, que se da cuando el dispositivo USB se enchufa al bus.

Nótese que en la carga del driver no se reserva ningún recurso, ya que no se puede suponer que nuestra tarjeta esté realmente conectada al bus en el momento de la

carga del driver. La reserva de recursos y la creación del FDO (F*unctional Device Object,* es el objeto que representa a nuestro dispositivo) no se llevan a cabo hasta que el dispositivo es detectado por la pila USB. Cuando esto sucede, el gestor de Plug and Play llama a la rutina ***BulkUsb_PnPAddDevice*** (en *Bulkpnp.c*), que se encarga de gestionar el arranque del dispositivo, llamando a las siguientes funciones:

En primer lugar llama a ***BulkUsb_CreateDeviceObject*** (en *bulkusb.c*), que crea el FDO (functional device object) y la estructura *DeviceExtension*. Entre ambas estructuras se tienen todos los datos necesarios para gestionar el dispositivo. El FDO contiene la parte de datos común a cualquier dispositivo de entrada/salida, y en *DeviceExtension* se guardan los datos particulares de nuestro dispositivo USB. Por eso en nuestros archivos de cabecera sólo se define éste último, ya que el FDO es general para Windows. El siguiente paso es relacionar este objeto dispositivo (FDO) con nuestro objeto driver. Para ello, se llama a la función ***BulkUsb_SymbolicLink*** (en *Bulkusb.c*), que crea un enlace al dispositivo basado en el código GUID que se define en el fichero *Guid829.h*. De este modo, cuando la aplicación de usuario quiera abrir el dispositivo, dará este código y así se le relacionará con el dispositivo adecuado y con este driver.

A continuación se llama a la función ***BulkUsb_QueryCapabilities*** (en *Bulkpwr.c*) que se encarga de consultar las capacidades de gestión de energía del dispositivo. Para ello, con la función *IoAllocateIrp()* crea una IRP de tipo IRP_MJ_PNP (función principal, "MaJor function") y IRP_MN_QUERY_CAPABILITIES (función secundaria, "MiNor function"), y la envía a la pila de drivers de USB, por medio de la función *IoCallDriver()*. Así es como se envían IRP's a la pila USB. La función de esta IRP es recoger la información de gestión de energía del dispositivo, que se obtiene de sus descriptores. Cuando la IRP es devuelta, la función puede disponer de esta información, que en nuestro caso indicará que el dispositivo no soporta gestión de la energía (ya que su alimentación es distinta que la del PC). Nótese cómo se realiza el proceso de envío y recepción de una IRP: antes de llamar a *IoCallDriver(),*se fija un evento que se activará cuando se devuelva la IRP, y se fija una rutina a la que se llamará cuando se active este evento y que procesará la IRP devuelta. Tras esto, se llama a *IoCallDriver()* y la función termina. Así no se ocupa la CPU durante el proceso de la

IRP por los niveles inferiores. Cuando se devuelve la IRP, se llama a la función indicada y así se reanuda el proceso.

Retomemos el proceso de inicialización del dispositivo. Tras llamar a la función *BulkUsb_QueryCapabilities*, la función *BulkUsb_PnPAddDevice* llama a ***BulkUsb_SelfSuspendOrActivate*** (en *Bulkpwr.c*), que intentará apagar el dispositivo hasta que haya que realizar una transferencia. Esto se hace porque puede pasar mucho tiempo entre que se enchufa el dispositivo y cuando realmente se usa, de modo que se intenta apagar para ahorrar energía. En nuestro caso, dado que el dispositivo no soporta esta función, esta rutina no hará nada. Nos aseguramos de eso incluyendo la línea

```
deviceExtension->PowerDownLevel = PowerDeviceD0;
```

que hace que la siguiente sentencia if sea cierta y se salga de la rutina sin intentar apagar (o encender) el dispositivo.

Aquí termina la función *BulkUsb_PnPAddDevice*, con lo que termina el proceso de creación del dispositivo funcional. Pero nos queda la fase de *setup*. Cuando la pila USB asigna algún recurso al dispositivo (como puede ser el FDO), le envía una nueva IRP, del tipo IRP_MJ_PNP. Por lo tanto, la función encargada de atenderla es ***BulkUsb_ProcessPnPIrp*** (en *Bulkpnp.c*), que lo que hace es enviarla a los drivers de los niveles inferiores para que la procesen, y a la vuelta, examina el campo *minor function*. Como éste es IRP_MN_PNP_START_DEVICE, se llama a la función ***BulkUsb_StartDevice*** (en *Bulkpnp.c*).

La función ***BulkUsb_StartDevice*** se va a encargar, en primer lugar, de pedirle a nuestra tarjeta el descriptor de dispositivo. Para ello, envía a los niveles inferiores una IRP. En este caso, queremos una transacción real por el bus USB, en la que le enviemos al dispositivo la orden GET_DEVICE_DESCRIPTOR, y él nos devuelva el descriptor pedido. Para pedir transacciones por el bus con cualquier comando, la IRP debe llevar el código IOCTL_INTERNAL_USB_SUBMIT_URB, y debe ir acompañada de una estructura llamada ***URB*** *(USB Request Block)*, en la que se indicará la transacción a ejecutar. Por tanto, lo que hace la función *BulkUsb_StartDevice* es eso: crea la URB específica para pedir el descriptor con la orden *UsbBuildGetDescriptorRequest(...)* y

luego llama a ***BulkUsb_CallUSBD(...)*** (en *Bulkusb.c*) que es la que se encarga de construir la IRP y enviarla. Así, cuando *BulkUsb_CallUSBD* devuelve el control a *BulkUsb_StartDevice,* ésta ya dispone del descriptor en el campo que indicó y puede hacer con él lo que quiera. Hemos realizado una transferencia a través del bus. Éste es el mecanismo que emplearemos para las transferencias de lectura y escritura.

En este punto, sólo queda seleccionar la configuración que queremos para que el enlace sea funcional. Como nuestro dispositivo sólo tiene una configuración con un solo interfaz, es ésta la configuración que hay que seleccionar. Para ello, la función *BulkUsb_StartDevice* llama a ***BulkUsb_ConfigureDevice*** (en *Bulkusb.c*), que crea una URB para pedir el descriptor de configuración. De nuevo se repite el mismo proceso: se llama a *BulkUsb_CallUSBD* pasándole la URB y ésta crea, envía y recibe la IRP con el descriptor. A continuación se llama a ***BulkUsb_SelectInterface*** (en *bulkusb.c*), que crea una URB para enviar al dispositivo la orden SET_CONFIGURATION. Cuando el dispositivo recibe esta orden, pasa a estar completamente configurado y por tanto el enlace ya es operacional. A partir de este punto, la aplicación puede intercambiar datos con el dispositivo a través del driver.

8.3.2.2. Abrir el dispositivo USB

Se va a explicar ahora con detalle cómo gestiona el driver las transferencias de datos que pide la aplicación de usuario. Ya hemos visto cómo se realiza la configuración inicial a nivel del driver, pero, ¿qué debe hacer una aplicación para intercambiar datos con el dispositivo USB? En primer lugar, la aplicación debe "abrir" el dispositivo, es decir, establecer un enlace con nuestra tarjeta. Esto es debido a que cuando se ejecuta la aplicación, el sistema operativo no la relaciona con ningún tipo de dispositivo de entrada/salida (la aplicación podría querer usar el puerto paralelo, el puerto USB o ningún puerto). Para establecer este enlace, la aplicación llama a la función *CreateFile(...)* para obtener un "manejador" (handle) del dispositivo. Una vez obtiene este manejador, se lo puede pasar a las funciones *ReadFile(...)* y *WriteFile(...)* para realizar transferencias.

Al llamar a *CreateFile* se le pasa como parámetro un nombre de fichero basado en la constante GUID de la que se habló en el apartado anterior. Por tanto, esta GUID, definida en el fichero *Guid829.h* , es la única forma de relacionar una aplicación con nuestro driver, y por lo tanto con nuestra tarjeta USB. La forma en que la aplicación genera el nombre de fichero a partir de la GUID es bastante compleja y se detalla en el apartado 8.4.2.1.

Cuando el S.O. recibe la llamada a *CreateFile*, lee el nombre de fichero pasado como parámetro y lo relaciona con *BulkUsb.sys*. Genera una IRP del tipo IRP_MJ_CREATE y se la envía a nuestro driver. Una vez en él, esta IRP se encamina a la rutina ***BulkUsb_Create*** (en *Bulkusb.c*). Nótese que la mayor parte del código de esta función está encerrado entre signos de comentarios. Esto es debido a que el procesamiento que se realiza en esta función está pensado para dispositivos USB que tengan varios "pipes". Aquí se determina a partir del nombre de fichero llamado qué pipe se quiere abrir y se actualiza su estado en el FDO. En nuestro caso, dado que sólo tenemos un pipe, no es necesario este procesamiento, por lo que esta función no hace absolutamente nada, se limita a devolver la IRP. En este punto no es necesaria ninguna transacción a través del bus, por lo que no se crea ni se envía ninguna URB.

Cuando la aplicación quiere cerrar la comunicación con el dispositivo (cerrar el pipe), llama a la función *CloseHandle()*, pasándole el manejador que obtuvo en *CreateFile()*. Esto genera una IRP del tipo IRP_MJ_CLOSE que se gestiona en la función ***BulkUsb_Close*** (en *Bulkusb.c*). Esta función cierra el pipe y, si era el último, llama a la función *BulkUsb_SelfSuspendOrActivate* para tratar de apagar el dispositivo. En nuestro caso, por tanto, esta función no tendrá ningún efecto.

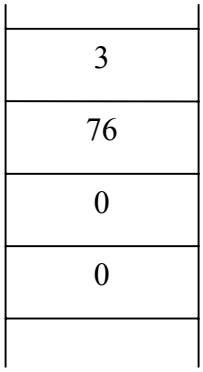
8.3.2.3. Operaciones de lectura

Pasemos ahora a explicar cómo se gestionan las operaciones de lectura de datos. Recordemos que nuestro dispositivo USB estaba preparado para ejecutar lecturas de hasta 4 bytes de datos en la misma transferencia, es decir, se podían leer hasta 4 direcciones con una sola instrucción (ver apartado 6.4.1.2). Los valores que deben tener los campos de esta instrucción USB son los siguientes:

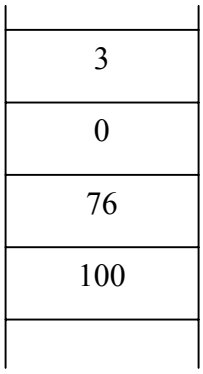
- *bmRequestType* → 0x11000000 (instrucción del fabricante, dirección dispositivo a PC).
- *bRequest* → 20 (decimal). Indica instrucción del fabricante "lee datos".
- *wValue* → contiene las dos primeras direcciones a leer. Como son dos bytes, el menos significativo contiene la primera dirección a leer (entero entre 0 y 256), y el más significativo, la segunda dirección. Si la aplicación sólo quiere leer 1 byte, este byte va a 0.
- *wIndex* → igual que el anterior. Contiene dos direcciones a leer, o va relleno de 0.
- *wLength* → número de bytes a devolver. Es el doble del número de lecturas que se vayan a realizar, ya que se devuelve un byte con la dirección leída, y a continuación el byte leído en esa dirección. Por tanto, puede valer 0, 2, 4, 6 ó 8.

Cuando la aplicación de usuario quiere hacer una lectura, llama a la función *ReadFile*, pasándole como parámetros el manejador que obtuvo al abrir el dispositivo y el búfer en el que se guardarán los datos. Dado que éstos se devuelven en formato dirección – dato, el tamaño del búfer debe ser el doble de bytes que se quieren leer. Por ejemplo, si queremos leer dos bytes, el búfer será de 4 bytes. Supongamos que

queremos leer las posiciones 3 y 76, que contienen respectivamente los valores 0 y 100. El búfer que la aplicación pasa al driver es



El driver deberá leer estos valores del búfer y empaquetarlos en los campos *wValue* (byte menos significativo = 3, byte más significativo = 76, es decir, *wValue* = 4C 03 hex.) y *wIndex* (00 00 hex.). El campo *wLength* valdrá 4. Así, el búfer que el dispositivo devolverá al driver y que el driver devolverá a la aplicación será



Vamos a ver cómo la función ***BulkUsb_Read*** (en *Ocrwblk.c*), que es la encargada de gestionar las lecturas, hace esto. Para ello expondremos su código C con los comentarios necesarios:

```
NTSTATUS
BulkUsb_Read(
   IN PDEVICE_OBJECT DeviceObject,
   IN PIRP Irp
   )

{

   NTSTATUS ntStatus = STATUS_SUCCESS;
   PURB urb;
   PDEVICE_EXTENSION deviceExtension = DeviceObject->DeviceExtension;
```

```
    ULONG siz, operaciones = 0, totalLength = 0;
    UCHAR reserved = 0, request = 20;
    USHORT value = 0,index = 0;
    PUCHAR p_aux = NULL;
    PVOID va;
    USHORT direcciones[4];
    ULONG i,j;

Irp->IoStatus.Information = 0;      //Indicará el número de bytes leidos con éxito
```

En primer lugar, se comprueba que el dispositivo no se haya desenchufado. Si esto sucede, se devuelve la IRP con un código de error. (Por simplicidad, he omitido el código de depuración y los comentarios originales).

```
 if ( !BulkUsb_CanAcceptIoRequests( DeviceObject ) ) {
      ntStatus = STATUS_DELETE_PENDING;
      Irp->IoStatus.Status = ntStatus;
      IoCompleteRequest (Irp, IO_NO_INCREMENT );
      return ntStatus;
   }
```

Ahora se determina el número de lecturas a realizar. Para ello, se obtiene el tamaño del búfer y se divide por 2. *MdlAddress* es un puntero a una estructura (MDL) que describe el búfer de la transferencia. *Irp* apunta a la IRP que nos ha llegado de la aplicación.

```
if ( Irp->MdlAddress ) {
         totalLength = MmGetMdlByteCount(Irp->MdlAddress);
         operaciones = totalLength / 2;
}
```

Si llega una petición de lectura de 0 bytes, se devuelve la IRP con éxito y no se pasa a los niveles inferiores.

```
if ( 0 == totalLength ) {
                  // allow 0-len read or write; just return success
      ntStatus = STATUS_SUCCESS;
      Irp->IoStatus.Status = ntStatus;

      IoCompleteRequest (Irp, IO_NO_INCREMENT );
      return ntStatus;
   }
```

Si se piden más de 4 direcciones, se devuelve un código de error.

```
if ( totalLength > 8 )
{
    ntStatus = STATUS_INVALID_PARAMETER;
    Irp->IoStatus.Status = ntStatus;
    IoCompleteRequest (Irp, IO_NO_INCREMENT );
    return ntStatus;
}
```

A continuación se inicializa la tabla auxiliar donde se guardarán las direcciones a leer. Como los campos *wValue* y *wIndex* se rellenan con 0 si no se usan, inicializamos la tabla a 0.

```
for ( j=0 ; j<4 ; j++ ) {
        direcciones[j] = 0;
}
```

Ahora se leen las direcciones a leer del búfer pasado por la aplicación. *p_aux* es un puntero a un byte, por lo que al incrementarlo pasamos al siguiente byte.

```
va = MmGetSystemAddressForMdl(Irp->MdlAddress);

p_aux = (PUCHAR) va;

for( i=0 ; i<operaciones ; i++ ) {
        direcciones[i] = (USHORT) *(p_aux++);
}
```

Ya tenemos las direcciones en la tabla de 4 bytes. Ahora, rellenamos los campos *wValue* y *wIndex* correctamente. Al multiplicar por 256, estamos desplazando 8 bits, por lo que pasamos al byte más significativo del campo. *Value* e *index* son valores USHORT, que tienen 16 bits.

```
value = direcciones[0];              //byte menos significativo
value+= (direcciones[1] * 256);      //byte más significativo
index = direcciones[2];
index+= (direcciones[3] * 256);
```

Ya tenemos todos los parámetros necesarios para construir nuestra instrucción; esto se hace creando una URB y llamando a la función *UsbBuildVendorRequest*, que nos construye una instrucción del fabricante y la rellena con los valores que le pasemos. Para ver la descripción de los parámetros con más detalle, consultar la documentación de la DDK.

```
siz = sizeof(struct _URB_CONTROL_VENDOR_OR_CLASS_REQUEST);

urb = BULKUSB_ExAllocatePool(NonPagedPool,siz);

if (urb) {
        UsbBuildVendorRequest(urb,
                URB_FUNCTION_VENDOR_DEVICE ,
                (USHORT) siz,
                USBD_TRANSFER_DIRECTION_IN |   USBD_SHORT_TRANSFER_OK,
                 reserved,
                 request,   //instrucción 20 (lee datos)
                 value,     // lleva 2 bytes de datos
                 index,     // otros 2 bytes (direcciones)
                 NULL,
                 Irp->MdlAddress,  //búfer
                 totalLength,
                 NULL);
```

Ya tenemos construida la URB con nuestra instrucción. Ahora hay que enviarla. Para ello, llamamos a *BulkUsb_CallUSBD*, que crea la IRP, la pasa a los niveles inferiores y espera a que sea devuelta. Por lo tanto, cuando termina y nos devuelve el control, ya tenemos el búfer relleno con los datos del dispositivo.

```
        BulkUsb_IncrementIoCount(DeviceObject);

        ntStatus = BulkUsb_CallUSBD(DeviceObject, urb);

        BulkUsb_DecrementIoCount(DeviceObject);
```

Liberamos la memoria de la URB

```
        BULKUSB_ExFreePool(urb);

} else {      //si no había suficiente memoria para la URB, se devuelve error.
        ntStatus = STATUS_INSUFFICIENT_RESOURCES;
}
```

Por último, sólo nos queda devolver la IRP a la aplicación. Ésta ya tiene la dirección del búfer porque lo ha reservado ella, así que no hay que hacer nada más.

```
   Irp->IoStatus.Status = ntStatus;
   IoCompleteRequest (Irp, IO_NO_INCREMENT );
   return ntStatus;

}
```

8.3.2.4. Operaciones de escritura

En el caso de las escrituras, el procesado debería ser más sencillo, porque en este caso no hay que adaptar la información a enviar como se hacía con las lecturas; sencillamente hay que construir la URB adecuada y pasarla a los niveles inferiores junto con el búfer que nos pasa la aplicación, sin tocar éste para nada. La instrucción USB del fabricante *escribe datos* se puede consultar en el apartado 6.4.1.2 de este documento. Permite hacer cuantas escrituras queramos en una sola transferencia USB. Los campos de la instrucción USB son los siguientes:

- *bmRequestType* → 0x01000000 (instrucción del fabricante, dirección PC a dispositivo).
- *bRequest* → 19 (decimal). Indica instrucción del fabricante "escribe datos".
- *wValue* → no se usa en este caso, por lo que va a 0.
- *wIndex* → igual que el anterior.
- *wLength* → número de bytes enviados. Es el doble del número de escrituras que se vayan a realizar, ya que el formato de una escritura es un byte de dirección seguido de un byte con el dato a escribir. El máximo es 254.

Sin embargo, el procesamiento que se realiza en el driver (rutina ***BulkUsb_write***, en el fichero *ocrwblk.c*) no es tan sencillo, debido a que se realiza una operación impuesta por el hardware: si el búfer que nos pasa el nivel de aplicación supera los 254 bytes (lo cual son 127 operaciones de escritura), la petición de escritura se "rompe" en segmentos de 254 bytes de largo, con lo cual la transferencia se realiza en varias etapas, es decir, se crean varias IRP's que se pasan a los niveles inferiores. Esto creará tantas transferencias distintas en el bus USB como IRP's creemos. Antes de devolver el

control a la aplicación, se espera a que retornen todas las IRP's, de modo que este procedimiento es transparente al usuario.

El motivo por el cual se hace esto es doble: por un lado, ya se vio que nuestro dispositivo USB sólo acepta transferencias de 254 bytes como máximo. Esto es debido a que sólo lee el byte menos significativo del campo *wLength*, por lo que el número máximo de bytes en una transferencia para no crear ambigüedades es 255; como el número de bytes de cada transferencia ha de ser par (1 dirección y 1 dato), el máximo entonces es 254. Esta limitación es fácilmente evitable: si se hace que el nivel lógico de nuestro dispositivo lea el campo *wLength* entero, se puede fijar el tamaño de los segmentos arbitrariamente.

El otro motivo para la segmentación es que al romper una petición de usuario demasiado extensa en varias peticiones en modo kernel más pequeñas, el *throughput* del S.O. es maximizado y se evita que un solo proceso monopolice los recursos del sistema. Otros procesos pueden hacer operaciones de entrada/salida entre las IRP's pendientes.

Este procesamiento complica bastante el código de la rutina *BulkUsb_write*, por lo que en vez de verlo completo vamos a mostrar un psudocódigo o guión de lo que se hace en la rutina. Con este guión es fácil entender el código completo, que lleva comentarios adicionales que facilitan su entendimiento. El pseudocódigo es el siguiente:

- Calcular la variable totalIrpsNeeded, el número de IRP's necesarias para romper la petición en segmentos de 254 bytes.
- Reservar memoria para una tabla de totalIrpsNeeded elementos tipo BULKUSB_RW_CONTEXT (ver nota 1). Rellenar con ceros.
- Para cada IRP a crear:
    - Reservar la memoria para la IRP con ***IoAllocateIrp***.
    - Obtener la dirección virtual del búfer de datos a partir de la IRP original, con ***MmGetMdlVirtualAddress***
    - Cada nueva IRP verá el búfer entero, pero se mapea su campo de lectura/escritura a una sola sección de 254 bytes dentro del búfer. Para ello se llama a ***IoAllocateMdl*** y a ***IoBuildPartialMdl***. Ver nota 2.

        - Llamar a la función ***UsbBuildVendorRequest*** para construir una URB del tipo URB_FUNCTION_VENDOR_DEVICE con los parámetros adecuados (***bmRequestType, bRequest, wValue, wIndex, wLength***).
        - Si se construye la URB, entonces:
            - Llamar a ***IoGetNextIrpStackLocation*** pasándole la IRP original para obtener ***nextStack***, que es un puntero al siguiente driver en la pila USB (el siguiente por debajo).
            - Rellenar los campos de la nueva IRP (ver nota 3)
            - Llamar a ***IoSetCompletionRoutine*** para fijar la rutina a la que se llamará cuando se devuelva la IRP. Esta rutina es ***BulkUsb_AsyncReadWrite_Complete***. Ver nota 4.
            - Llamar a ***IoCallDriver*** para pasar la nueva IRP a los niveles inferiores. Ver nota 5.
        - Fin entonces.
- Pasar a la siguiente IRP.
- Una vez procesadas todas las IRP's, si ha habido algún fallo, intentar resetear el dispositivo llamando a la función ***BulkUsb_ResetDevice*** (en ***IoctlBlk.c***). Ver nota 6.
- Fin de la rutina.

NOTAS

(1): Cada elemento BULKUSB_RW_CONTEXT es una estructura en la que se guarda la información de una IRP que se ha creado y enviado, y que está pendiente de retorno. Contiene un puntero a dicha IRP y un puntero a la URB asociada a la IRP, así como un puntero de retorno al FDO. Así el driver puede saber en cada momento cuantas transferencias quedan pendientes. El puntero a la tabla de BULKUSB_RW_CONTEXT se guarda en el FDO (objeto dispositivo).

(2): De este modo, sólo existe un búfer, el que nos pasa la aplicación de usuario, que puede ser de cualquier tamaño. Lo que se hace es dividirlo entre todas las IRP, de

forma que cada una envía al dispositivo su parte del búfer y por tanto se termina enviando el búfer completo.

(3): Se fija el código de la IRP a IRP_MJ_INTERNAL_DEVICE_CONTROL, que indica que la transferencia es una instrucción de control de un dispositivo, se le añade un puntero a la URB y se le da el código de control IOCTL_INTERNAL_USB_SUBMIT_URB, que indica que la instrucción está en la URB añadida. Así se consigue que se envíe la URB al dispositivo.

(4): Esta rutina (*BulkUsb_Write*) envía las IRP's a los drivers de nivel inferior y termina, con lo que no se ocupa la CPU mientras se espera. Cuando una IRP es devuelta, se llama a la rutina que se haya fijado con esta función, en este caso *BulkUsb_AsyncReadWrite_Complete* (en *Ocrwblk.c*), que realiza las siguientes funciones: decrementa el contador de IRP's pendientes, actualiza el número de bytes ya transmitidos al dispositivo, libera la memoria de la IRP terminada y si la IRP era la última, actualiza y devuelve la IRP original a la aplicación de usuario.

(5): Todo este procesado de IRP's es el mismo que se hace en la rutina *BulkUsb_CallUSBD*, pero en este caso lo hacemos directamente porque así no tenemos que esperar a que las IRP's sean devueltas.

(6): Esta función manda una IRP de control al driver que controla el puerto al que está enchufado el dispositivo (ya sea el root hub o un hub USB) para que éste señalice un reset y el dispositivo sea así reseteado.

8.3.2.5. Parada y desconexión del dispositivo

Esta función se gestiona a través de una serie de IRP's de Plug & Play. Éstas son enviadas a la función ***BulkUsb_PnPIrp***, que las reenvía a la función correcta según el tipo de IRP. Estas rutinas están ampliamente comentadas en el código, por lo que no nos vamos a centrar en ellas. Las IRP's que gestionan son las siguientes (según su función secundaria):

- IRP_MN_QUERY_REMOVE_DEVICE: Se envía para preguntar al driver si se puede desenchufar físicamente el dispositivo sin provocar errores en el software. Se suele enviar antes de IRP_MN_REMOVE_DEVICE en las desconexiones ordenadas, como cuando el usuario desinstala el dispositivo desde el Administrador de dispositivos. El driver espera a que terminen las transferencias pendientes y responde afirmativamente. En este punto, ya no puede aceptar más transferencias. Si responde negativamente, no se prosigue con el proceso de desconexión.

- IRP_MN_CANCEL_REMOVE_DEVICE: Se envía para anular la anterior, cuando ésta se ha respondido negativamente. Así el driver sabe que puede continuar con las transferencias.

- IRP_MN_REMOVE_DEVICE: Se envía para cerrar el driver. Puede ser en procesos ordenados tras una IRP_MN_QUERY_REMOVE_DEVICE contestada afirmativamente, o sin previo aviso (por ejemplo si el usuario desenchufa el dispositivo repentinamente). El driver debe abortar todas las transferencias pendientes y borrar todos los recursos, terminando por el FDO.

- IRP_MN_QUERY_STOP_DEVICE: Se envía para preguntar al driver si se puede parar el dispositivo para reconfigurarlo. Se envía justo antes de una IRP_MN_STOP_DEVICE. Si quedan transferencias pendientes, el driver responde negativamente y el proceso no prosigue. Si se responde afirmativamente, no se pueden aceptar más transferencias hasta que el dispositivo sea configurado de nuevo (se le envíe la orden SET_CONFIGURATION).

- IRP_MN_CANCEL_STOP_DEVICE: Se envía para anular la anterior.

- IRP_MN_STOP_DEVICE: Se envía antes de reconfigurar el dispositivo. No se puede asumir que se haya enviado antes una IRP_MN_QUERY_STOP_DEVICE (reconfiguración ordenada) por lo que si hay transferencias pendientes hay que cancelarlas. Se gestiona en la rutina ***BulkUsb_StopDevice***, en *Bulkpnp.c*.

8.3.2.6. Gestión de energía

Nuestro driver también debe responder a las IRP's de gestión de energía que le lleguen del sistema operativo. Estas IRP's son las siguientes:

- IRP_MN_QUERY_POWER
- IRP_MN_SET_POWER
- IRP_MN_WAIT_WAKE

El encaminamiento de estas IRP's se realiza en la función ***BulkUsb_ProcessPowerIrp*** (en *Bulkpwr.c*). El procesamiento básico para cada IRP es enviarla a los niveles inferiores y no hacer nada más, ya que el dispositivo sólo tiene un estado de energía y por tanto no soporta la gestión. Se han incluido estas funciones para que el driver cumpla con los requisitos de un driver de Windows 98.

8.3.3. Compilación

En este apartado se va a explicar cómo compilar el código fuente del driver para obtener el fichero ejecutable, *Bulkusb.sys*. En el apartado 8.5 se explicará el proceso de instalación del driver en el PC, para lo cual este fichero debe ir acompañado de otro, *Bulkusb.inf*, que es un fichero de texto que se puede modificar con cualquier editor de texto, y que relaciona el driver con el dispositivo físico al enchufar éste al PC.

Ya se ha mencionado que la forma de compilar el driver es a través de la aplicación *Windows 98 DDK*, que requiere de la instalación previa de Visual C++ de Microsoft. Aunque en la documentación se explicita que no vale la versión 6.0 del Visual C++, yo lo he podido compilar con esta versión sin problemas. También se admiten las versiones 4.0 y 5.0.

Existen dos modos de compilación: *Checked* y *Free*. Con el primero, se obtiene una versión del driver para depuración, en la que se incluyen todas las rutinas dedicadas a obtener mensajes de salida para depuración. De este modo, con un visor de mensajes del sistema operativo se pueden ver las salidas que va produciendo el driver durante su ejecución y conocer así su estado. Estas salidas típicamente indican las entradas y salidas de cada subrutina. En nuestro código, todas las llamadas a la función *BULKUSB_KdPrint* producen salidas de este tipo. Así podemos ver, por ejemplo, el valor de cualquier variable en un momento determinado.

El otro modo de compilación, *Free*, nos da la versión definitiva del driver. La diferencia con el anterior es que el driver compilado en este modo no produce salida de depuración, por lo que es más rápido y compacto. De este modo se debe construir la versión definitiva del driver, una vez depurada.

El proceso de compilación es el siguiente: una vez instalado correctamente el kit, copiar a un directorio cualquiera los ficheros con el código fuente y cabeceras, así como los ficheros *Makefile* y *Sources* , que también se adjuntan con el código fuente. PRECAUCIÓN: El path completo de este directorio debe cumplir la nomenclatura de MS-DOS, no la de Windows, es decir el nombre de cada directorio no debe superar los

8 caracteres ni incluir espacios. Si esto no se hace así, el compilador da errores diciendo que no puede abrir los archivos.

Desde la barra de tareas de Windows, ejecutar Inicio → Programas → Developments Kits → Windows 98 DDK → Free Build Environment (para compilación en modo *Free*) o bien Inicio → Programas → Developments Kits → Windows 98 DDK → Checked Build Environment (para compilación en modo *Checked*). Esto abrirá una ventana de MS-DOS y preparará las variables necesarias para la compilación. A continuación cambiaremos al directorio de nuestro driver, y teclearemos

```
> build –cZ
```

Esto lanzará el proceso de compilación. Con el código que se proporciona con este documento, el proceso no debe dar ningún error. Una vez terminada la compilación, podemos recoger nuestro driver ejecutable del directorio C:\98ddk\lib\i386\checked o de C:\98ddk\lib\i386\free, según el tipo de compilación que hayamos realizado. Este fichero se debe copiar al directorio c:\windows\system32\drivers (sólo si estamos actualizando el driver, si es la primera instalación ver el apartado 8.5.1).

8.4. La aplicación “USBLink.exe”

Como ya se mencionó en la introducción de este capítulo, esta aplicación se entrega a modo de ejemplo de cómo interactuar con el driver *bulkusb.sys* para intercambiar datos con nuestra tarjeta USB. Por tanto, sólo pretende indicar cómo hacer otras aplicaciones, y por ello no tiene ninguna finalidad concreta.

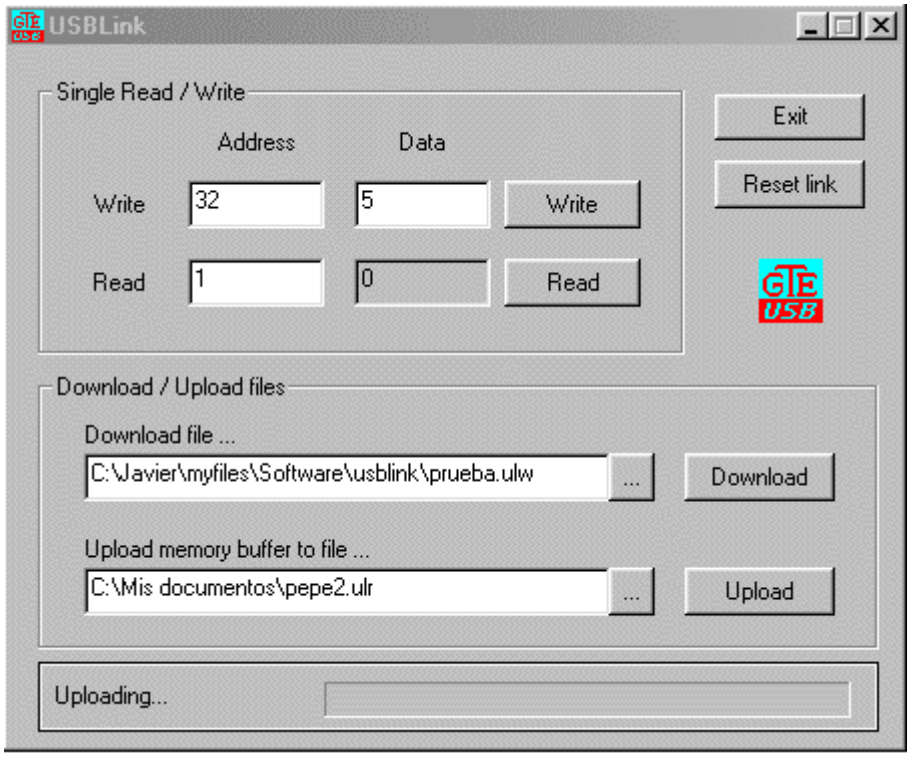
Está escrita en C++ y se basa en la MFC de Microsoft, por lo que su estructura es bastante compleja. Sin embargo, no se va a explicar aquí esa estructura, tan sólo la parte necesaria para entender el proceso de intercambio de información. Ésta parte sí es bastante sencilla y se basa únicamente en C puro, por lo que es fácil de comprender. El usuario que pretenda hacer otra aplicación la podrá hacer con la herramienta que desee, ya que el código que nos interesa no está ligado a ninguna plataforma concreta.

La función principal de la aplicación es generar datos para enviar al dispositivo USB, y recibir los que éste envíe. Conviene recordar que en el nivel de aplicación del dispositivo se implementaba un búfer de 256 posiciones (bytes) de RAM. Ahora iniciaremos lecturas y escrituras en este búfer. La aplicación ve el dispositivo como un fichero corriente de datos, por lo que el funcionamiento es tan sencillo como abrir el fichero, escribir o leer de él tal como se haría con un fichero normal, y cerrar el fichero.

En primer lugar veremos cómo funciona el programa, y luego pasaremos a describir las funciones del código que nos interesan. Por último, haremos un análisis de las prestaciones obtenidas.

8.4.1. El interfaz USBLink

Al ejecutar el archivo *usblink.exe*, nos aparece el cuadro de diálogo en el que se basa el programa. Este cuadro es el que se muestra a continuación:



En él se pueden observar tres zonas: el recuadro superior, titulado "*Single Read / Write*", el recuadro central, "*Download / Upload files*", y abajo tenemos una barra de estado. Ésta está formada por un indicador de la acción que se está llevado a cabo (en este caso, *Uploading...* indica que se están transfiriendo datos de la placa al PC), y por una barra de progreso que se va llenando conforme va avanzando la acción.

Además de éstos, tenemos los botones 

, que cierra la aplicación, y 

, que resetea el dispositivo. Esto último también está a modo de demostración de las capacidades del enlace, ya que nunca nos va a hacer falta resetear el dispositivo. Si en alguna transferencia se atasca, el PC lo detectaría, y enviaría de nuevo el paquete de inicio de la transferencia; al recibirlo, el interfaz USB de la placa se desbloquearía y la transferencia se llevaría a cabo. Si aún así fuese necesario resetearlo manualmente, USB permite desenchufar la tarjeta y volver a enchufarla. El PC envía entonces un reset al dispositivo y vuelve a configurarlo, por lo que pasa a estar operativo de nuevo.

El recuadro "*Single Read / Write*" nos permite realizar lecturas y escrituras de un solo dato, es decir, leemos o escribimos en una posición del búfer. Para hacer una

escritura, escribimos la dirección de destino (de 0 a 255) en el campo *Write Address* y el dato que queremos escribir (de 0 a 255, es un byte) en el campo *Write Data*. A continuación pulsamos 

, y esto provoca la escritura en el búfer del dispositivo. En el ejemplo de la figura, escribiríamos el dato *5* en la dirección *32*. Conviene recordar que los leds de la placa indican el valor contenido en la posición 1 del búfer, por lo que si escribimos un dato en la dirección 1 lo veremos en hexadecimal en los leds.

Para hacer una lectura, se escribe la dirección a leer en el campo *Read Address* y se pulsa el botón 

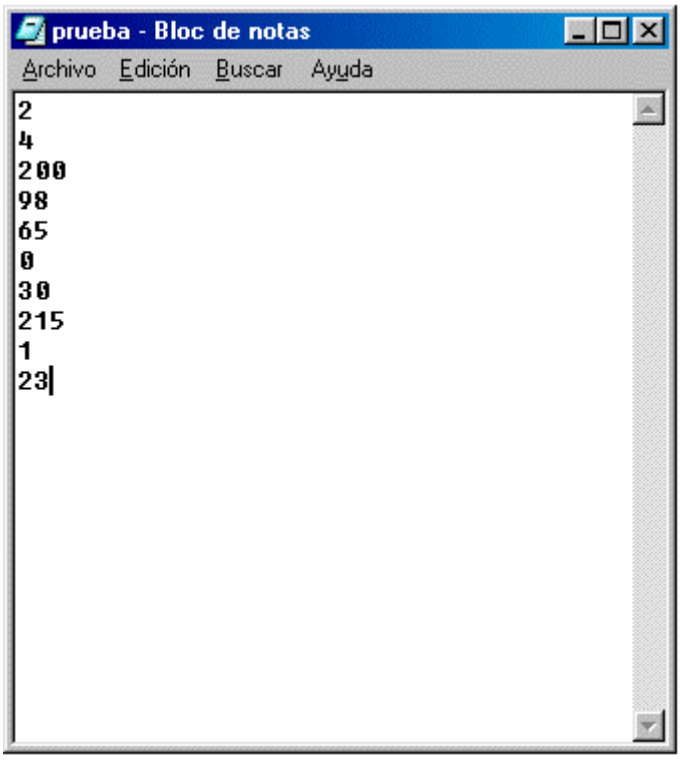
. El dato leído del búfer aparecerá en el campo *Read Data*.

El recuadro *"Download / Upload files"* sirve para hacer transferencias de datos masivas, entre el búfer y un fichero de texto. Así podemos escribir en un fichero cuantos datos queramos escribir en el búfer, y enviárselos al dispositivo todos juntos. De esta forma conseguimos maximizar el ancho de banda de la transmisión, ya que el número de datos útiles transmitidos aumenta mucho en relación al número de datos de carga (cabeceras, asentimientos,...).

Cuando queramos descargar un fichero a la placa (escritura), éste debe tener un formato determinado: su extensión debe ser *ulw,* y debe ser un fichero de texto que en las líneas impares contenga una dirección y en las pares un dato. Por ejemplo, si queremos hacer las siguientes escrituras simultáneamente,

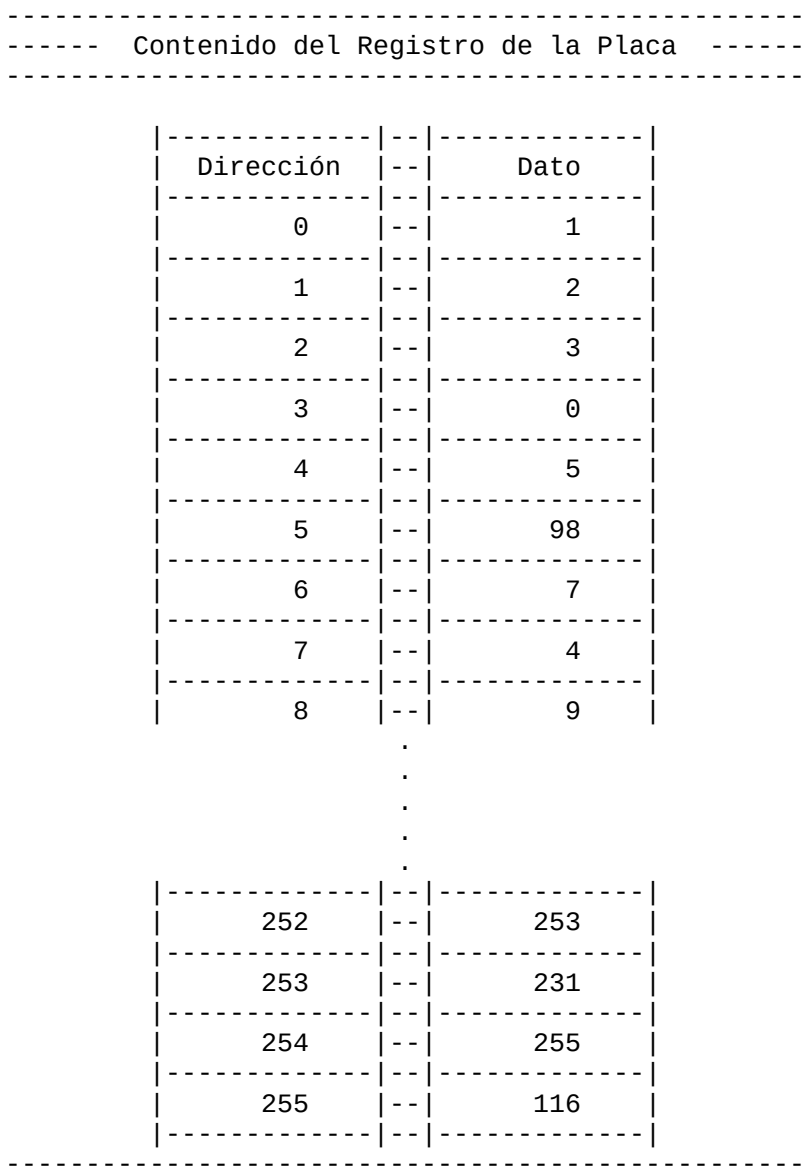
Direcciones	Datos
2	4
200	98
65	0
30	215
1	23

el contenido del fichero (en la gráfica, *prueba.ulw*) deberá ser el siguiente:



Una vez tenemos editado el fichero deseado, pulsamos el botón  junto al campo *Download file...* y seleccionamos el fichero que queremos enviar. La ruta completa de este fichero aparecerá en el cuadro de texto. Si el fichero seleccionado no tiene la extensión *ulw*, aparecerá un cuadro indicando el error. Por último, pulsamos en  y se inicia el proceso de escritura. Como el fichero puede ser todo lo largo que queramos, la escritura puede llevar un tiempo, por lo que en la barra de estado aparece el mensaje *Downloading...* y el indicador de progreso señalará el estado de la transferencia. Cuando ésta termine, aparecerá el mensaje *Ready*. En este proceso también se genera un fichero llamado *usblink.log*, que nos da estadísticas de la transferencia, y que veremos más adelante.

Al pulsar el botón  , el programa lee secuencialmente todas las posiciones del búfer de la placa y almacena los datos en el fichero que nosotros le hayamos indicado en el campo *Upload memory buffer to File...*. Este fichero deberá tener la extensión *ulr*. Por tanto, una vez hecho esto el fichero *ulr* contendrá el valor de todas las posiciones del búfer, de la siguiente forma:



NOTA: Si el fichero indicado para esto no existe, se crea nuevo; si ya existía, se sobrescribe.

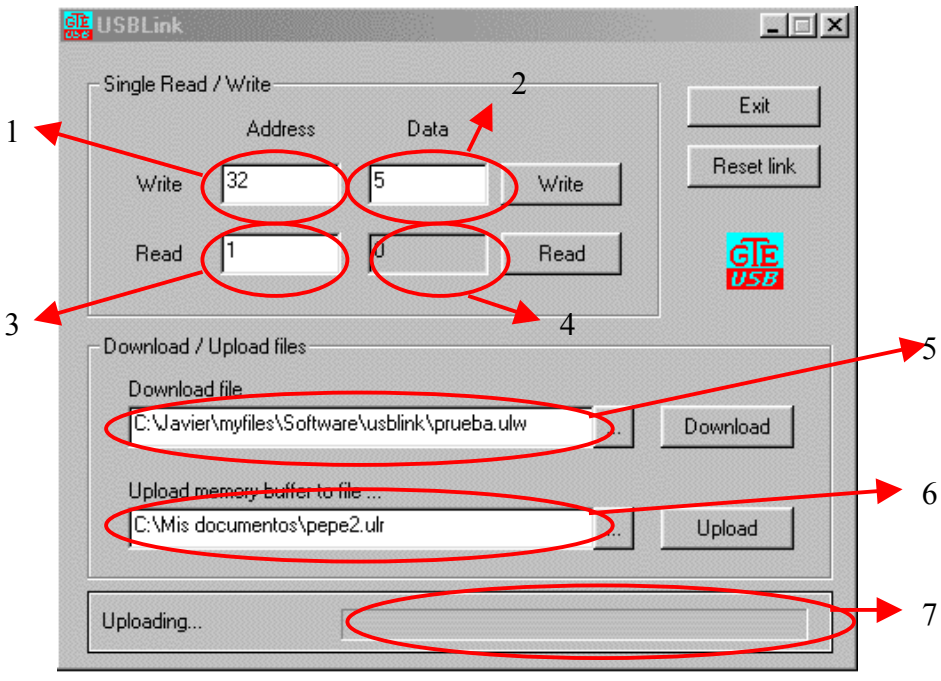
8.4.2. Código del programa

Vamos a ver ahora cómo está hecho el programa. Para ello, nos vamos a olvidar de toda la parafernalia de la MFC y nos vamos a centrar exclusivamente en el código que nos interesa.

Cuando ejecutamos el programa, se crea el objeto cuadro de diálogo correspondiente a la ventana que vemos en pantalla. Este objeto pertenece a la clase *CUsblinkDlg,* a la que pertenecen todas las funciones y datos miembros que nos interesan. Todos ellos están contenidos en el fichero *usblinkDlg.cpp*. Además, veremos algunas funciones globales que se recogen en el fichero *usblink.cpp*.

A cada uno de los campos de edición que aparecen en nuestro cuadro de diálogo le corresponde una variable (dato miembro de la clase *CUsblinkDlg*) que nos sirve para intercambiar datos entre el cuadro de diálogo y las funciones. Por ejemplo, al campo *Write Address* (perteneciente al recuadro *Single Read / Write*), le corresponde una variable tipo entero llamada *m_nDirWr*, de forma que cuando escribimos una dirección entre 0 y 255 en ese campo, el valor que escribimos se almacena en esta variable y así la función de escritura puede leer la dirección de la escritura de esta variable.

A continuación se muestran las variables que le corresponden a cada campo:



Campo	Nombre campo	Variable	Tipo	Observaciones
1	Write Address	m_nDirWr	Entero sin signo	Se comprueba que esté entre 0 y 255
2	Write Data	m_nDataWr	Entero sin signo	Igual que la anterior
3	Read Address	m_nDirRd	Entero sin signo	Igual que la anterior
4	Read Data	m_nDataRd	Entero sin signo	Solo lectura
5	Download file	m_strWrPath	Cadena de caracteres	Guarda la dirección completa del fichero .ulw
6	Upload file	m_strRdPath	Cadena de caracteres	Ruta del fichero .ulr que guarda el estado del registro
7	Barra de progreso	m_nProgress	Entero sin signo	El valor en porcentaje de la barra de progreso

Por otro lado, al pulsar cada uno de los botones del cuadro, se llama a su correspondiente función miembro, que realiza la función asociada al botón. Así por ejemplo, al pulsar el botón *Write*, se llama a la función miembro *OnButtonWr()*, que se encarga de escribir el valor de la variable *m_nDataWr* (correspondiente al campo *Write Data*) en la posición del búfer de la placa dada por la variable *m_nDirWr*. La función que le corresponde a cada botón la podemos ver en la siguiente tabla:

Botón	Función asociada	Descripción
	OnButtonWr()	Escribe un byte en la placa
	OnButtonRd()	Lee un byte de la placa
	OnDownload()	Hace las escrituras indicadas en un fichero .ulw
	OnBrowseWr()	Recoge la ruta completa del fichero seleccionado por el usuario
	OnUpload()	Lee todas las posiciones del búfer y las vuelca a un fichero
	OnBrowseRd()	Recoge la ruta del fichero en el que se quiere volcar el búfer
	OnReset()	Resetea el dispositivo

Estas funciones son las únicas que nos interesan, y están todas incluidas en el fichero *usblinkDlg.cpp*. Su estructura general es la siguiente:

- Abrir el dispositivo
- Realizar la lectura o escritura (o resetear)
- Cerrar el dispositivo

Antes de pasar a ver cada una de estas funciones, como todas tienen que abrir el dispositivo, vamos a ver cómo se realiza esto.

8.4.2.1. Apertura del dispositivo

Para abrir el dispositivo, todas las funciones llaman a la función o*pen_dev()*. Esta función es global, así como todas las subrutinas a las que llama, por lo que se encuentran en el fichero *usblink.cpp*. El objetivo es establecer un enlace con el driver *bulkusb.sys*. Como ya se vió, esto se hace a través de un código GUID que conocen tanto el driver como la aplicación. Por lo tanto, usaremos la misma constante GUID que se usaba en el driver.

El proceso de apertura del dispositivo es genérico e independiente de la plataforma de desarrollo, por lo que se recomienda copiar íntegramente estas funciones (están escritas en C) al desarrollar nuevas aplicaciones.

El código de la función *open_dev()* es el siguiente:

```
HANDLE open_dev()
{
        HANDLE hDEV = OpenUsbDevice( (LPGUID)&GUID_CLASS_I82930_BULK,
                                     completeDeviceName);
        return hDEV;
}
```

Como se puede ver, esta función pasa la llamada a otra rutina y devuelve un elemento tipo HANDLE, que es un manejador de fichero. Este manejador lo

guardaremos y es el que pasaremos a las funciones de lectura y escritura para vincular las transferencias a nuestro dispositivo USB.

A la función *OpenUsbDevice* le pasamos como parámetros un puntero a la GUID y una cadena de caracteres vacía (*completeDeviceName*). Por lo tanto, es aquí donde se establece el enlace entre el driver y la aplicación: ambos conocen la GUID. De hecho, para compilar este código necesitaremos el fichero de cabecera en donde se define la GUID (*GUID829.h*), perteneciente al código fuente del driver *bulkusb.sys*. En cuanto a la cadena de caracteres, la función *OpenUsbDevice* la rellena con el nombre completo que identifica a nuestro dispositivo, y se la pasa al driver para abrir el dispositivo correcto.

La función *OpenUsbDevice* no nos interesa en profundidad, por lo que la veremos por encima. En primer lugar llama a la función del sistema *SetupDiGetClassDevs(...)*, que nos crea una lista de todos los dispositivos conectados al sistema (PnP). A continuación, pasa esta lista junto con la GUID a la función *SetupDiEnumDeviceInterfaces(...),* que cada vez que es llamada mira un dispositivo de la lista y comprueba si es el que coincide con la GUID. Si devuelve un resultado negativo, se vuelve a llamar hasta que se encuentre nuestro dispositivo. Una vez que hemos encontrado nuestro dispositivo, llamamos a la función *OpenOneDevice(...)* para abrirlo.

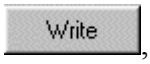
La función *OpenOneDevice* pertenece a este proyecto y se encuentra en el mismo archivo que las anteriores. Como hemos visto, su misión es abrir el dispositivo. Para ello, se le pasan los datos que hemos obtenido de la tabla anterior, que identifican a nuestro dispositivo. Con estos datos, esta función llama a *SetupDiGetInterfaceDeviceDetail(...)* dos veces (la primera es para leer el espacio que necesita reservar) para obtener detalles adicionales acerca del dispositivo. Entre estos detalles se encuentra el nombre del dispositivo, que es el que nos interesa para abrirlo. Guardamos este nombre en la cadena global que vimos anteriormente, y con él llamamos a la función *CreateFile(...)*, que establece el vínculo con el driver y nos devuelve el manejador (handle) del dispositivo. Ya vimos que la llamada a *CreateFile* provoca que se envíe una IRP al driver, de forma que este abre el dispositivo.

Con esto concluye el proceso de apertura del dispositivo. Devolvemos el manejador a la función llamante y si el proceso ha fallado en alguna de sus etapas (por ejemplo porque el dispositivo no esté enchufado o el driver no esté instalado), se devuelve el código INVALID_HANDLE_VALUE.

8.4.2.2. Rutinas de transferencia de datos

Ahora ya podemos ver cómo se hace una lectura o una escritura. Para ello, veremos cada una de las funciones que son llamadas al pulsar un botón de transferencia en nuestro cuadro de diálogo.

- Escritura de un solo byte.

Al rellenar los campos *Write Address* y *Write Data* y pulsar el botón 

, se llama a la función *OnButtonWr()*, que se encuentra en el fichero *usblinkDlg.cpp*. Esta función crea el búfer que hay que enviar al dispositivo (en este caso es de dos bytes, la dirección y luego el dato), y llama a *WriteFile(...)*, de modo que el búfer se envía a la placa a través del driver *bulkusb.sys*. Veamos su código:

```
void CUsblinkDlg::OnButtonWr()
{
        ULONG nBytesWrite = 0;          //Guardará el número de bytes escritos (2)
        CHAR *poutBuf = NULL;           //puntero al búfer de datos

        UpdateData(TRUE);               //Esta llamada hace que los valores que se han
                                          escrito en los campos del cuadro de diálogo se
                                          guarden en sus respectivas variables. Así
                                          disponemos de los datos que ha introducido el
                                          usuario.

        HANDLE hWrite = open_dev();     //Abrimos el dispositivo

        poutBuf = (PCHAR) malloc(2);    //Reservamos nuestro búfer de dos bytes

        if (poutBuf && hWrite != INVALID_HANDLE_VALUE) {

                                        // El if se cumple si las dos llamadas anteriores
                                        han sido correctas, es decir, si hay memoria libre
                                        y el dispositivo está enchufado y operativo.

                PUCHAR pOut = (PUCHAR) poutBuf;
```

```
            *pOut = (UCHAR) m_nDirWr;          //Pasamos la dirección al búfer
            *(pOut+1) = (UCHAR) m_nDataWr;  //Y ahora el dato; son UNSIGNED
                                                CHAR, es decir, bytes sin formato.

            WriteFile(hWrite,       //petición de escritura; así se inicia una transferencia
              poutBuf,              //de datos; se pasa el búfer (es el que llega al dispo-
              2,                    //sitivo tal cual), el tamaño (número de bytes a trans-
              &nBytesWrite,         //ferir), y en nBytesWrite se devolverá el numero de
              NULL);                //bytes realmente escritos.

            assert(2 == nBytesWrite);          //nos aseguramos que han llegado los
                                                dos; si no, dará error y el programa
                                                se interrumpirá.

        } else {                                // si la placa no se ha enchufado

            AfxMessageBox("Recursos insuficientes.\nNo se llevó a cabo la escritura");

                                                //esto muestra una pantalla con este
                                                  mensaje y un botón OK
        }

        if (poutBuf) {                          //liberar el búfer
            free(poutBuf);
         }

        if(hWrite != INVALID_HANDLE_VALUE)
            CloseHandle(hWrite);                //cerrar el dispositivo (esto llama a la
                                                rutina de cerrar el dispositivo del
                                                driver).

        UpdateData(FALSE);                      //copiamos los datos de nuestras
                                                variables a la pantalla. Esto sirve si
                                                se han modificado.

}
```

Por tanto, se ha visto que lo que hay que hacer es crear el búfer con el formato tal como lo espera el nivel de aplicación de la máquina VHDL de la placa, y al llamar a *WriteFile* este búfer llega a la placa tal cual (encapsulado en la instrucción del fabricante *escribe datos*).

- Lectura de un solo byte.

Para leer un solo byte del búfer de la placa, se escribe la dirección que se desee leer (entero entre 0 y 255) en el campo *Read Address* del cuadro de diálogo y se pulsa 

. Esto provoca una llamada a la rutina *OnButtonRd()*, cuyo código vamos a ver a continuación. Conviene recordar que para hacer una lectura, el búfer que se envía a la placa debe tener un tamaño de 2 x el número de lecturas (entre 1 y 4 lecturas por llamada), de forma que en nuestro caso tendrá 2 bytes. En el primer byte se enviará la dirección que se desea leer, y en el segundo, un 0. A la vuelta, en el primer byte del mismo búfer estará la dirección leída, y en el segundo, el dato leído.

```
void CUsblinkDlg::OnButtonRd()
{
	ULONG nBytesRead = 0;                                   //número de bytes leídos
	PUCHAR pbuff = NULL;                                    //puntero al búfer de usuario

	UpdateData(TRUE);                       //guardamos la dirección a leer en m_nDirRd

	HANDLE hRead = open_dev();                              //abrimos el dispositivo

	PCHAR pinBuf = (PCHAR) malloc(2);                       //búffer de 2 posiciones

	if (pinBuf) {

		pbuff = (PUCHAR) pinBuf;

		for (int j=0; j<2; j++) {                   //Rellenamos el búfer con ceros
			*(pbuff+j) = 0;
		}
	}

	if (pinBuf && hRead != INVALID_HANDLE_VALUE) {          // si recursos correctos

		*(pbuff) = (UCHAR) m_nDirRd;          //guardamos la dirección en el búfer

//A continuación hacemos la llamada al sistema para lecturas. Esto llama al driver para la lectura. Le pasamos el manejador del fichero (hRead), el búfer, el tamaño del búfer y nBytesRead, que contendrá el número de bytes leídos (que son dos, la dirección y el dato en sí). Devuelve un 1 si la lectura es correcta, y 0 en otro caso.

		BOOL success = ReadFile(hRead,
					pinBuf,
					2,
					&nBytesRead,
					NULL);

		if (success) {
```

```
                    m_nDirRd = *pbuff;                    //guardamos dirección leída
                    m_nDataRd = *(pbuff+1);   //guardamos el dato leído en su variable

            } else {                          //Mostramos un mensaje de lectura errónea

                    AfxMessageBox("Error en la lectura");
            }

    } else {
            AfxMessageBox("Recursos insuficientes.\nNo se llevó a cabo la lectura");
    }

    if (pinBuf) {                                          //liberamos los recursos
        free(pinBuf);
     }

    if(hRead != INVALID_HANDLE_VALUE)
            CloseHandle(hRead);

    UpdateData(FALSE);              //ponemos los valores devueltos en pantalla. Así
                                      podemos ver el dato leído en el campo Read
                                      Data.
}
```

Como se ha podido ver, hacer lecturas y escrituras es muy fácil. Esto es todo lo que hay que saber para poder hacerlas. Así se pueden leer hasta 4 posiciones del búfer de una sola transferencia y hacer todas las escrituras que queramos en una sola llamada a *WriteFile*. En las siguientes rutinas veremos cómo se hace esto.

- Descargar un fichero .ulw

Vamos a ver ahora procesos de transferencias múltiples. Para hacer una descarga (*download*), en primer lugar hay que seleccionar el fichero cuyo contenido se quiere descargar. La extensión de este fichero debe ser *ulw*, y en el apartado 8.4.1 se explica cuál debe ser su formato.

Hay dos formas de seleccionar el fichero: o bien se escribe su ruta completa en el campo *Download file...*, o bien se pulsa el botón  correspondiente. Esto abre un cuadro de diálogo *Abrir fichero* típico de Windows, y ahí elegimos el fichero del directorio correspondiente. Lo que ocurre al pulsar el botón es que se llama a la función *OnBrowseWr()*, la cual (ver su código en *usblinkDlg.cpp*) recoge la ruta completa del

fichero seleccionado y la almacena en la cadena de caracteres *m_strWrPath*. Luego muestra en el cuadro de diálogo dicha cadena. Es importante indicar que esto NO ABRE EL FICHERO, sólo recoge su nombre.

Una vez hemos seleccionado el fichero, pulsamos 

, lo que provoca una llamada a *OnDownload()*. El código de esta rutina debería ser muy similar al de *OnButtonWr()*, pero se complica al tener que atender una serie de cuestiones:

- Abrir, leer y cerrar el fichero ulw.
- Controlar el indicador de progreso, así como el indicador de estado.
- Calcular el tiempo de la operación y la tasa de transferencia
- Generar el fichero *usblink.log* con toda esta información
- Dividir la escritura en varios segmentos.

Del último punto no se había comentado nada todavía. En principio, esta función debería crear un búfer y volcar en él los datos del fichero tal como los va leyendo, y llamar a *WriteFile* para transferirlos todos de una sola vez. Sin embargo, se ha comprobado que para ficheros con más de 5000 bytes las transferencias empiezan a fallar, por lo que el enlace no es operativo (se suelen perder 10 o 20 bytes en cada transferencia errónea). Por ello, se ha optado por dividir los datos a transferir en bloques de tamaño fijo. El tamaño de cada bloque viene definido por la constante *PART_TAM*, que en el caso de la solución final vale 1016 (está definida en *usblink.h*). Por tanto, esta rutina se debe encargar también de dividir el búfer a enviar en bloques de este tamaño (menos el último, que será menor), e irlos enviando a la placa sucesivamente. Si un bloque falla, se vuelve a reenviar tantas veces como haga falta hasta que llegue correctamente. No se pasa al siguiente bloque hasta que el actual se ha entregado con éxito. Así nos aseguramos que todos los bytes llegan, y en el orden correcto.

Se ha elegido un tamaño de segmento de 1016 bytes por varios motivos: en primer lugar, al ser todas las transferencias de menos de 5000 bytes, prácticamente ninguna dará error. Por tanto, el enlace se vuelve mucho más fiable. Además este tamaño es múltiplo de 254, que es el tamaño de las particiones que hace el driver

bulkusb.sys. Conviene elegir siempre el tamaño del bloque como múltiplo de 254, porque así se maximiza la relación entre datos útiles y bytes de tara.

En el apartado 8.4.3 se realiza un estudio estadístico de errores para varios tamaños de bloque y varios tamaños de ficheros. Este estudio se ha basado en los resultados obtenidos en el fichero *usblink.log*, que recoge todas las estadísticas de cada descarga de ficheros. Dicho estudio demuestra que el valor de 1016 bytes es el óptimo para el tamaño de cada segmento.

Veamos por tanto el código de la rutina *OnDownload()*:

```
void CUsblinkDlg::OnDownload()
{
```

//En primer lugar, se inicializan las variables y se actualizan con los datos de pantalla.

```
    CHAR *fileName=NULL;              //puntero al nombre del fichero ulw
    FILE *pfile = NULL;               //puntero al fichero ulw
    FILE *pflog = NULL;               //puntero al fichero usblink.log
    int address = 0;
    ULONG nbytes = 0, totalBytes=0;
    ULONG nBytesWrite = 0;
    CHAR *poutBuf = NULL;             //puntero al búfer de datos a enviar
    HANDLE hWrite = INVALID_HANDLE_VALUE;  //manejador del dispositivo
    int totalParts = 1;
    ULONG tamano = 0;
    PUCHAR pOut = NULL;
    BOOL flag;
    clock_t start, finish, st_segment, fn_segment;  //variables para contar tiempo
    double seconds, sec_segment;
    int nFallos = 0;

    UpdateData(TRUE);
```

//El siguiente código controla la barra de progreso y el indicador de estado. Éste mostrará el mensaje "*Downloading...*" y la barra se pone al inicio.

```
    SetDlgItemText(IDC_STATIC_DWNLD, "Downloading...");
    m_nProgress = 0;
    CProgressCtrl *pProg = (CProgressCtrl *) GetDlgItem(IDC_PROGRESS1);
    pProg->SetPos(m_nProgress);
```

//A continuación se valida el nombre de fichero dado. Primero se comprueba que se haya seleccionado un fichero (longitud de la cadena distinta de 0) y luego que la extensión sea la correcta. Si no se cumple alguna de las condiciones, se muestra un mensaje de error.

```
    int k=m_strWrPath.GetLength();
    if (k==0) {
```

```
            AfxMessageBox("No seleccionó ningun fichero");
            SetDlgItemText(IDC_STATIC_DWNLD, " ");
            return;
        }

        fileName = (CHAR *) calloc(k+1,sizeof(CHAR));
        strcpy(fileName,m_strWrPath);
        fileName[k]='\0';

        if (fileName[k-1] != 'w' || fileName[k-2] != 'l' || fileName[k-3] != 'u') {
            AfxMessageBox("Formato de fichero incorrecto. Por favor seleccione un fichero
                        con extensión .ulw");
            free(fileName);
            SetDlgItemText(IDC_STATIC_DWNLD, " ");
            return;
        }
```

//Ya tenemos un fichero correcto. Podemos enviarlo a la FPGA. Para ello, primero abrimos el fichero indicado en modo lectura. También creamos el fichero *usblink.log,* y empezamos a rellenarlo.

```
        pfile=fopen(fileName, "r");
        pflog=fopen("usblink.log","w");

        fprintf(pflog,"Bajando archivo %s a FPGA...\n",fileName);

        if (pfile) {
```

//Las siguientes sentencias recorren el fichero de datos para contar los bytes que tiene, es decir, el número de bytes a enviar (*nbytes*). Luego situa el cursor al inicio del fichero.

```
            while(fscanf(pfile,"%d",&address) != EOF)
                nbytes++;

            fseek(pfile,0L,SEEK_SET);
```

//A continuación calculamos el número de segmentos necesarios, y escribimos estos datos en el fichero de resultados. Abrimos nuestro dispositivo USB, iniciamos el contador de tiempo global y nos metemos en un bucle *for* (para cada segmento).

```
            totalBytes = nbytes;
            totalParts = nbytes/PART_TAM;
            if (nbytes%PART_TAM)
                totalParts++;

            fprintf(pflog,"Numero total de bytes: %d\n", totalBytes);
            fprintf(pflog,"Numero total de segmentos: %d\n",totalParts);

            hWrite = open_dev();

            if (hWrite != INVALID_HANDLE_VALUE) {

                start = clock();
                for (ULONG j=0; j<totalParts; j++) {
```

//Guardamos en *tamano* el tamaño del bloque a enviar (es PART_TAM salvo que sea el último bloque), y decrementamos *nbytes* para que tenga el número de bytes que nos quedan. Reservamos el búfer que enviaremos a la placa.

```
if (nbytes<=PART_TAM) {
        tamano = nbytes;
} else {
        tamano = PART_TAM;
        nbytes -= PART_TAM;
}

poutBuf = (CHAR *)malloc(tamano);

if (poutBuf) {
```

//Rellenamos el búfer con el bloque de datos correspondiente, que se lee del fichero. Del bucle *while* no se sale hasta que este búfer se haya enviado correctamente.

```
pOut = (PUCHAR) poutBuf;

for(ULONG i=0; i<tamano; i++)
        fscanf(pfile,"%d",pOut++);

flag = TRUE;

while(flag)
{
```

//Iniciamos el contador de tiempo parcial (del segmento), y mandamos el búfer al driver para su transmisión. Cuando retomamos el control, paramos el contador y calculamos el tiempo de la operación, almacenando las estadísticas en *usblink.log*. Se comprueba si la transferencia ha sido correcta; si lo es, se actualiza la barra de progreso y se pasa al siguiente bloque de datos; si no, se reintenta la escritura. En cualquier caso, se libera el búfer.

```
st_segment = clock();

WriteFile(hWrite,
        poutBuf,
        tamano,
        &nBytesWrite,
        NULL);

fn_segment = clock();
sec_segment = (double)(fn_segment - st_segment)/CLOCKS_PER_SEC;

fprintf(pflog,"Segmento %d : Tiempo = %f s., tasa = %f
      bytes/s.\n",(j+1),sec_segment,(double)(tamano/sec_segment));

if(tamano == nBytesWrite) {
        flag = FALSE;
        m_nProgress = ((j+1)*512/totalParts);
        pProg->SetPos(m_nProgress);
} else {
        nFallos++;
}
```

```
                }

                free(poutBuf);
```

//Si no se pudo reservar memoria para el búfer, se muestra un mensaje de error y se sale de la función

```
                } else {
                        AfxMessageBox("No se ha realizado la operación. RECURSOS
INSUFICIENTES");
                        SetDlgItemText(IDC_STATIC_DWNLD, " ");
                        break;
                }
        }
```

//Hemos terminado la transferencia completa; mostramos el indicador *"Ready"* y paramos el contador de tiempo global; calculamos las estadísticas globales y las bajamos al fichero de resultados. También se vuelve a poner la barra de progreso a 0, y se liberan todos los recursos. Por último, se actualiza la pantalla.

```
                        SetDlgItemText(IDC_STATIC_DWNLD, "Ready");

                        finish = clock();
                        seconds = (double)(finish - start) / CLOCKS_PER_SEC;
                        fprintf(pflog,"\nTiempo total = %f segundos;\n", seconds);
                        fprintf(pflog,"Tasa total de transferencia = %f bytes/seg\n",
                                (double)(totalBytes/seconds));
                        fprintf(pflog,"Numero de paquetes retransmitidos: %d\n",nFallos);

                        CloseHandle(hWrite);

                } else {

                        AfxMessageBox("No se ha realizado la operación. RECURSOS
                                        INSUFICIENTES");
                        SetDlgItemText(IDC_STATIC_DWNLD, " ");
                }

                fclose(pfile);

                if(pflog) fclose(pflog);

        } else {

                AfxMessageBox("No se ha realizado la operación. RECURSOS INSUFICIENTES");
                SetDlgItemText(IDC_STATIC_DWNLD, " ");
        }

        free(fileName);

        m_nProgress = 0;
        pProg->SetPos(m_nProgress);

        UpdateData(FALSE);
}
```

El contenido del fichero *usblink.log* tras una descarga será parecido a éste:

```
Bajando archivo C:\porrino\software\culebra.ulw a FPGA...
Numero total de bytes: 100000
Numero total de segmentos: 10
Segmento 1 : Tiempo = 0.050000 s., tasa = 203200.000000 bytes/s.
Segmento 2 : Tiempo = 0.060000 s., tasa = 169333.333333 bytes/s.
Segmento 3 : Tiempo = 0.000000 s., tasa = 1.#INF00 bytes/s.
Segmento 4 : Tiempo = 0.000000 s., tasa = 1.#INF00 bytes/s.
Segmento 5 : Tiempo = 0.060000 s., tasa = 169333.333333 bytes/s.
Segmento 6 : Tiempo = 0.050000 s., tasa = 203200.000000 bytes/s.
Segmento 7 : Tiempo = 0.000000 s., tasa = 1.#INF00 bytes/s.
Segmento 7 : Tiempo = 0.050000 s., tasa = 203200.000000 bytes/s.
Segmento 8 : Tiempo = 0.060000 s., tasa = 169333.333333 bytes/s.
Segmento 9 : Tiempo = 0.050000 s., tasa = 203200.000000 bytes/s.
Segmento 10 : Tiempo = 0.000000 s., tasa = 1.#INF00 bytes/s.

Tiempo total = 0.660000 segundos;
Tasa total de transferencia = 151515.151515 bytes/seg
Numero de paquetes retransmitidos: 1
```

Como se puede ver, nos da la tasa de transferencia de cada segmento, así como la total, y el número de retransmisiones necesarias. Nótese en este ejemplo que el bloque 7 se ha enviado dos veces, ya que la primera fue errónea. En este caso el tamaño del bloque era de 10.000 bytes.

- Leer el búfer de la placa completo

Ahora vamos a ver qué pasa al pulsar el botón 

. Previamente hay que seleccionar el fichero de destino, que debe tener la extensión *.ulr*. Igual que antes, podemos escribir un nombre de fichero en el cuadro o pulsar 

, lo que llama a la función *OnBrowseRd()*; ésta abre un cuadro de diálogo *Guardar como...* y guarda la ruta del fichero en la cadena *m_strRdPath*. Si el fichero ya existe, se pregunta si se quiere sobrescribir. No se abre ni se crea el fichero.

Al pulsar *Upload*, se llama a la función *OnUpload()*. Ésta comprueba la validez del fichero indicado, lo crea, y empieza a hacer lecturas. En el apartado 8.3.2.3 se muestra el formato del búfer de lectura. Recuérdese que sólo se pueden hacer cuatro lecturas en una sola operación, por lo que de nuevo hay que leer el búfer de la placa por

partes. Cuando se han completado todas las lecturas, se vuelca el búfer leído al fichero destino. El código es el siguiente:

```
void CUsblinkDlg::OnUpload()
{
	CHAR *fileName=NULL;    //puntero al nombre del fichero
	FILE *pfile = NULL;     //puntero al fichero
	CHAR *pinBuf = NULL;    //puntero al búfer de 8 posiciones
	CHAR *pReg = NULL;      //puntero a un búfer de 512 posiciones
	ULONG nBytesRead;
	BOOL success=FALSE;
	ULONG j;

	UpdateData(TRUE);
```

//Preparar barra de estado y de progreso; comprobar la validez del fichero

```
	SetDlgItemText(IDC_STATIC_DWNLD, "Uploading...");
	m_nProgress = 0;
	CProgressCtrl *pProg = (CProgressCtrl *) GetDlgItem(IDC_PROGRESS1);
	pProg->SetPos(m_nProgress);


	int k=m_strRdPath.GetLength();
	if (k==0) {
		AfxMessageBox("No seleccionó ningun fichero destino");
		SetDlgItemText(IDC_STATIC_DWNLD, " ");
		return;
	}

	fileName = (CHAR *) calloc(k+1,sizeof(CHAR));
	strcpy(fileName,m_strRdPath);
	fileName[k]='\0';
```

//Ya tenemos un fichero correcto. Lo abrimos y reservamos la memoria para dos búferes. El primero, de 8 bytes, será el que se enviará a la placa y regresará con 4 posiciones leídas; en el segundo se guardan las direcciones leídas en los bytes impares y los datos leídos en los pares. Por eso tiene 512 bytes (2 x 256). También se abre el dispositivo.

```
	pfile=fopen(fileName, "w");

	pinBuf=(CHAR *)malloc(8);
	pReg=(CHAR *)malloc(512);
	HANDLE hRead = open_dev();

	if (pfile && pinBuf && pReg && hRead != INVALID_HANDLE_VALUE) {

		PUCHAR pbuff = (PUCHAR) pinBuf;
		PUCHAR pregaux = (PUCHAR) pReg;

		ULONG i = 0;
```

//El siguiente bucle realiza todas las lecturas, en bloques de 4

```
while (i<256) {

        for (j=0;j<8;j++)          //rellenamos el búfer con ceros
                *(pbuff+j) = 0;

        *(pbuff) = i;              //se guardan las direcciones a leer
        *(pbuff+1) = i+1;          //en los primeros 4 bytes del búfer
        *(pbuff+2) = i+2;
        *(pbuff+3) = i+3;

        success = ReadFile(hRead,
                                        pinBuf,
                                        8,
                                        &nBytesRead,
                                        NULL);
```

//Si la lectura ha sido exitosa, se copia el búfer al de 512 posiciones, se actualiza el índice y la barra de progreso; si falla, se reintenta.

```
        if (success) {

                for (j=0;j<8;j++)
                        *(pregaux++)=*(pbuff+j);

                i+=4;
                m_nProgress+=8;
                pProg->SetPos(m_nProgress);
        }
}
```

//La función *dumpRegToFile(…)* (en fichero *usblinkDlg.cpp*) vuelca los datos al fichero y los formatea correctamente. Devuelve 0 si hay error.

```
        int flag = dumpRegToFile(pfile,pReg);

        if (flag=0) {
                AfxMessageBox("Error escribiendo el fichero");
                SetDlgItemText(IDC_STATIC_DWNLD, " ");
        } else {
                SetDlgItemText(IDC_STATIC_DWNLD, "Ready");
        }

} else {
        AfxMessageBox("No se ha realizado la operación. RECURSOS INSUFICIENTES");
        SetDlgItemText(IDC_STATIC_DWNLD, " ");
}

if (fileName)
        free(fileName);

if (pinBuf)
        free(pinBuf);
```

```
        if (pReg)
                free(pReg);

        if (hRead != INVALID_HANDLE_VALUE)
                CloseHandle(hRead);

        if (pfile)
                fclose(pfile);

        m_nProgress=0;                  //volvemos la barra de progreso a 0
        pProg->SetPos(m_nProgress);
}
```

En la página 138 podemos ver cómo queda el fichero conteniendo el registro.

8.4.2.3. Generación de un reset

Por último, veamos cómo se genera un reset por software. Como el driver no tenía una entrada específica para crear señales de reset, hay que hacerlo a más bajo nivel. Al pulsar 

se llama a la función *OnReset()*, cuyo código es:

```
        void CUsblinkDlg::OnReset()
        {
                        HANDLE hdev = open_dev();
                        if (hdev) {
                                reset_device(hdev);
                                CloseHandle(hdev);
                        }
                        SetDlgItemText(IDC_STATIC_DWNLD, " ");
        }
```

Abre el dispositivo, llama a *reset_device(..)* y luego lo cierra. *reset_device* es una rutina global y por lo tanto está en *usblink.cpp*. Ésta es la siguiente:

```
void reset_device( HANDLE hDEV )
{

        UINT success;

        if (hDEV == INVALID_HANDLE_VALUE) {
               AfxMessageBox("No se puede resetear. ¿Está enchufada la placa?");
               return;
        }
```

```
        success = DeviceIoControl(hDEV,
                                  IOCTL_BULKUSB_RESET_DEVICE,
                                  NULL,
                                  0,
                                  NULL,
                                  0,
                                  NULL,
                                  NULL);

        return;
}
```

La llamada a *DeviceIoControl(...)* pasándole el manejador del dispositivo y el código IOCTL_BULKUSB_RESET_DEVICE hace que el dispositivo se resetee.

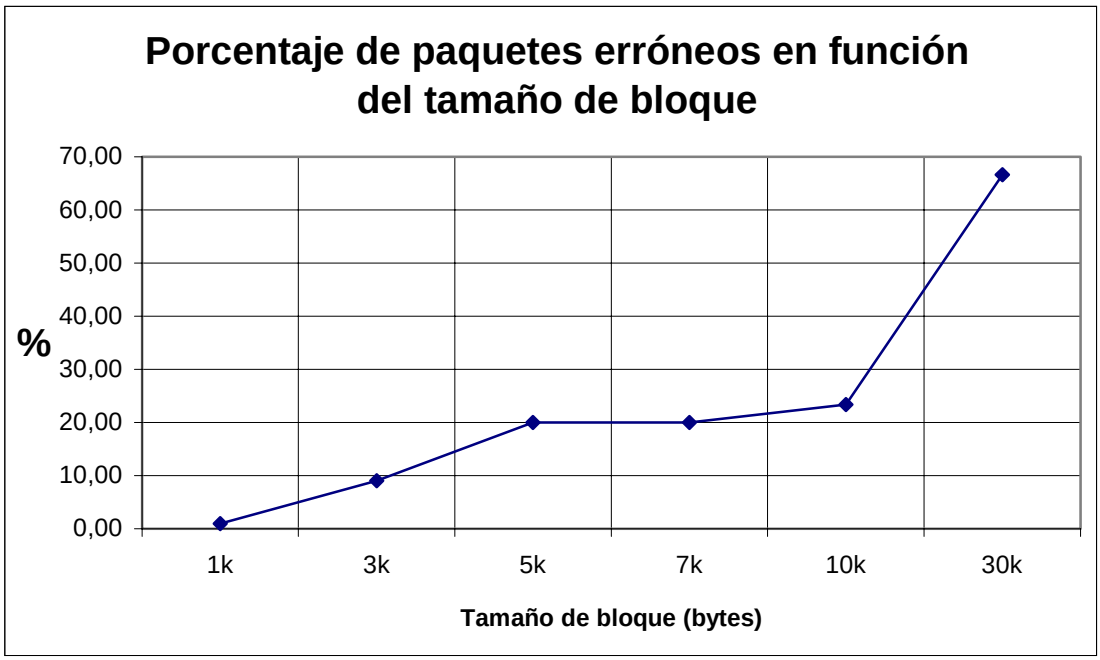
8.4.3. Estudio del tamaño de los segmentos

Vamos a exponer aquí los resultados obtenidos al realizar procesos de escritura masivos con distintos tamaños de bloque. Estos resultados se han obtenido a partir de los ficheros *usblink.log* resultado de cada descarga, y justifican que se haya elegido un tamaño de 1016 bytes/bloque. Se ha tomado como muestra un fichero de 100 kbytes. En la siguiente tabla se muestran estos resultados:

Tamaño Fichero	Tamaño Partición	Paquetes	Errores	% Errores	Error medio (%)	Bytes retransmitidos	Tasa final (bytes/s)
100k	1k	99	1	1,01	1,01	1.000	140.845
100k	3k	33	3	9,09	9,09	9.000	140.845
100k	5k	20	4	20	20	20.000	121.951
100k	7k	15	3	20	20	21.000	138.889
100k	10k	10	1	10	23,33	10.000	151.515
100k	10k	10	6	60		60.000	113.636
100k	10k	10	0	0		0	153.846
100k	30k	4	1	25	66,67	30.000	129.870
100k	30k	4	7	175		210.000	75.758
100k	30k	4	0	0		0	151.515

De la observación de estos datos, se pueden sacar las siguientes conclusiones:

- El porcentaje de errores (paquetes fallidos / paquetes correctos) aumenta con el tamaño del bloque, tal como se ve en la siguiente gráfica:



- Además de aumentar el número de errores, dado que cada paquete retransmitido tiene un número de bytes mayor al aumentar el tamaño del bloque, aumenta mucho el número de bytes que hay que retransmitir.

- En los casos de 10k y 30k, nótese que se han apuntado tres medidas: una con número de errores normal, otra sin fallos y otra con muchos fallos. Para tamaños de bloques grandes, aunque puede haber transmisiones que no tengan ningún fallo, esto se compensa con otras que tienen muchos fallos: por ejemplo, en el caso de 30k hay una transmisión en la que se retransmiten 210.000 bytes, cuando el tamño del fichero es de 100.000; en total, 310.000 bytes. Esto supone un incremento del 175% en el tamaño de la transmisión. Por tanto, la media de errores resultante sale muy alta, lo que desaconseja usar estos tamaños.

- La tasa total de cada transmisión sale parecida en todos los casos, salvo en aquellos en que hubo muchos errores. Esto es porque cuanto más pequeño es el

bloque, se transmiten menos datos (menos errores), pero al aumentar el número de particiones se aumenta el tamaño de la información no útil (protocolo) que se transmite.

- Por tanto, se aconseja usar tamaños de bloques de entre 1000 y 3000 bytes. Si queremos el menor número de bytes retransmitidos, se usará el tamaño de 1k; si queremos mayor aprovechamiento del bus, usaremos bloques más grandes. En ambos casos se consiguió una tasa total de unos 140 Kbytes/s, aunque en algunos de los bloques individuales se alcanzaron velocidades de hasta 200 Kbytes/s.

8.5. Instalación del software

En este apartado se va a contar qué es lo que hay que hacer para que este sistema funcione. En primer lugar, decir que sólo el driver requiere una instalación, la aplicación USBLink se ejecuta directamente. Se va a explicar la instalación para Windows 98.

Antes de proceder al proceso de instalación, comprobar que el sistema USB funcione correctamente en nuestro PC. Para ello, no olvidar habilitar el hardware USB del PC en la BIOS (suele venir deshabilitado). Si el sistema operativo está correctamente instalado, esto debería ser suficiente; para asegurarnos, ejecutar la aplicación *INTEL USB System Check* (ejecutable *usbready.exe,* ver apartado 5.3).

A continuación, alimentar y configurar la placa XSV800 de Xess con nuestra solución VHDL, tal como se explica en el capítulo 9 pero sin llegar a enchufarla al puerto USB.

La instalación se realiza suponiendo que no se han cambiado los códigos VID y PID que identifican a nuestro dispositivo, y que en la solución entregada son VID=0458 y PID=0003. Si se han cambiado, hay que modificar el fichero *bulkusb.inf*, que es la interconexión entre el dispositivo físico y el driver. Para ello, cambiar en la línea

```
%USB\VID_0458&PID_0003.DeviceDesc%=BULKUSB.Dev, USB\VID_0458&PID_0003
```

los valores de VID y PID.

8.5.1. Primera instalación

1. Copiar los ficheros *bulkusb.sys* y *bulkusb.inf* a un directorio temporal, desde el que se realizará la instalación.
2. Enchufar la placa al PC mediante el cable USB (no quitar la conexión por el puerto paralelo). En este momento, la pila de drivers USB detectará el dispositivo y le pedirá los descriptores. Una vez leido, nos aparece el cuadro de diálogo “Add New Hardware Wizard”, diciendo que va a buscar los drivers para nuestro dispositivo.

Pulsar “Siguiente”. En la siguiente pantalla, pulsar en “Buscar el mejor controlador para su dispositivo (Se Recomienda)”, y pulsar el botón “Siguiente”. En la siguiente pantalla, indicar el directorio de instalación en el que hemos puesto nuestros ficheros y pulsar “Siguiente”. En la siguiente pantalla, se indica que se va a proceder a la instalación. Comprobar que la ruta de *bulkusb.inf* es correcta y pulsar “Siguiente”. Se instalará el driver. En la última pantalla, aparecerá un mensaje afirmativo. Pulsar “Finalizar”. Si apareciese algún error o se pidiera reiniciar, es que ha habido algún fallo. Intentar corregirlo y seguir las instrucciones del apartado “Simular una primera instalación”.

8.5.2. Actualizar el driver

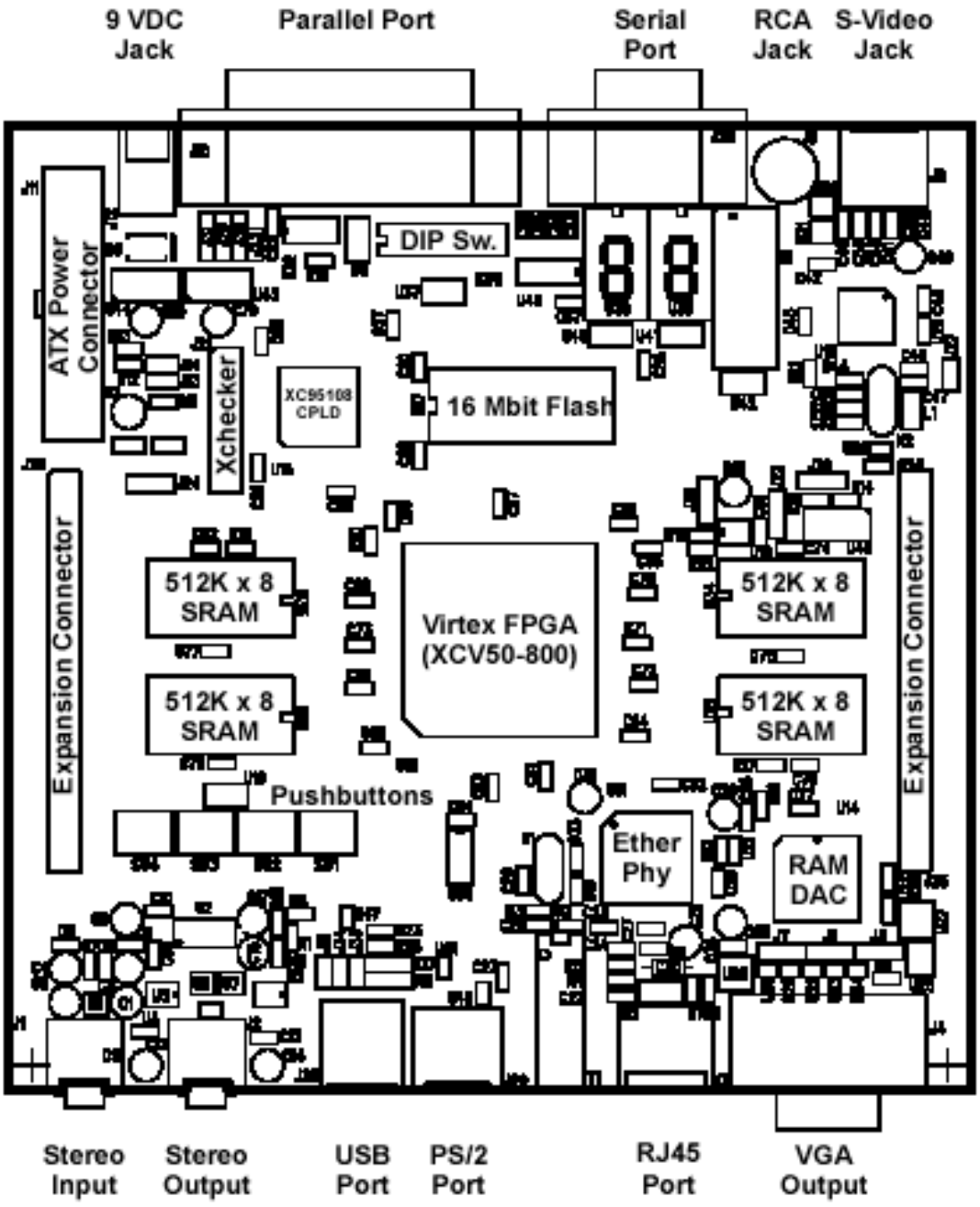
Si se ha compilado una nueva versión del driver tras una primera instalación satisfactoria, simplemente copiar el nuevo ejecutable (*bulkusb.sys*) al directorio C:\windows\system32\drivers. No es necesario reiniciar.

8.5.3. Simular una primera instalación

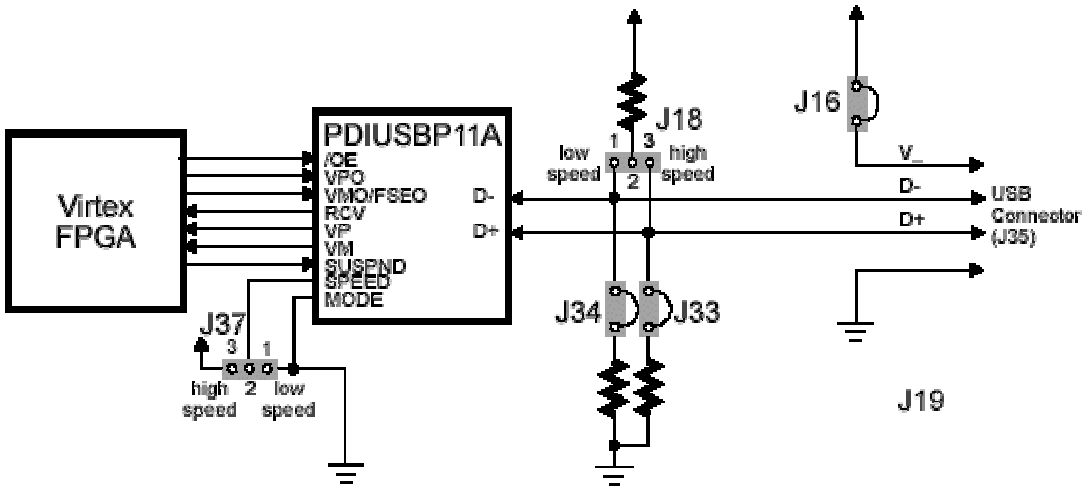
1. Borrar *bulkusb.inf* de C:\windows\inf.
2. Borrar *bulkusb.sys* de C:\windows\system32\drivers.
3. Usando Regedit, eliminar del registro la clave
   \LocalMachine\System\Enum\USB\VID_0458&PID_0003
4. Proceder a instalar como se indica en el apartado “Primera instalación”.

9. Configuración de la tarjeta XSV

En este capítulo se van a explicar los pasos necesarios para preparar la tarjeta XSV800 de Xess para utilizarla con nuestro diseño. En primer lugar, veamos el esquemático de la placa para poder identificar los jumpers.



1. En primer lugar, comprobar que todos los jumpers estén colocados correctamente:
   - Colocar un conector en el jumper J23.
   - Colocar un conector en la posición 1-2 del jumper J32 y quitarlos de J13 y J14.
   - Colocar conectores en la posición 2-3 de los jumpers J29, J30 y J31. El último es especialmente importante porque configura el puerto paralelo.
   - Quitar el jumper J36.
   - Para el interfaz USB, la colocación de los jumpers es la siguiente:



   - J16: sin conector. Es para usar la alimentación del PC a través del bus.
   - J33 y J34: Sin conector; estas resistencias se usan si la placa actúa como host USB, no como dispositivo.
   - J18: colocar un conector en la posición 2-3. Esto configura el dispositivo como "High speed".
   - J37: colocar conector en 2-3. Es para seleccionar el modo de funcionamiento del transceptor PDIUSBP11A de Philips.

2. Conectar nuestro PCB con el oscilador tal como se explica en el apartado 7.4.
3. Si es la primera vez, fijar la salida del oscilador a 12 MHz siguiendo el procedimiento indicado en el apartado 7.3.
4. Aplicar la alimentación de la placa. Para ello, conectar una fuente de alimentación tipo ATX al conector J11 de la placa. Se enciende el LED D2.

5. Conectar el cable paralelo al PC y a la placa. Cuidado: seguir siempre los pasos 4 y 5 en este orden. Al desconectar la alimentación, desconectar siempre después el cable paralelo.
6. Programar la CPLD. La CPLD gestiona la configuración de la FPGA a través del puerto paralelo. Para ello, hay que programarla con el interfaz necesario. Este interfaz está contenido en el fichero usbd*wnldpar.svf,* aunque también se podría usar el fichero *dwnldpar.svf* que viene con las herramientas *gxstools*. Se utiliza la utilidad *gxsload.exe*. Al ejecutarla, aparece la siguiente pantalla:

   Se selecciona el puerto paralelo y se arrastra el fichero que queramos descargar hasta la ventana blanca. Así se inicia el proceso de descarga.
7. Programar la FPGA. Para ello, se arrastra hasta la ventana de *gxsload* nuestro fichero *usb.bit*, que contiene el interfaz USB. Salimos de la aplicación *gxsload*. Con esto, ya tenemos listo nuestro dispositivo USB.
8. Se supone que ya se ha instalado el driver *bulkusb.sys*. Si no es así, seguir los pasos del apartado 8.5.1.
9. Enchufar la placa al puerto USB.
10. Ejecutar la aplicación *USBLink*, y ya está listo para usar.

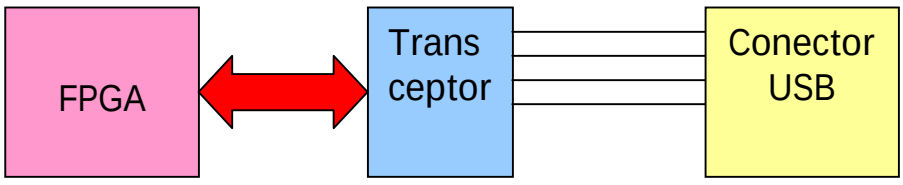
10. Futuras aplicaciones del interfaz

Como ya se ha expuesto en capítulos anteriores, este proyecto está pensado para ser aplicado como un interfaz más de la placa de desarrollo sobre la que se está trabajando en el Grupo de Tecnología Electrónica de la Universidad de Sevilla. Por ello, se ha tratado de obtener un producto lo más general posible para facilitar su incorporación a dicha placa de desarrollo. En este capítulo se pretende aportar una serie de guías de cómo debe ser esa adaptación. En primer lugar veremos los requisitos de componentes físicos (hardware), a continuación pasaremos a ver las adaptaciones necesarias tanto en el código VHDL como en el software, y por úlltimo se darán algunas ideas sobre posibles cambios que puedan mejorar las prestaciones del sistema.

10.1. Requisitos físicos del interfaz USB

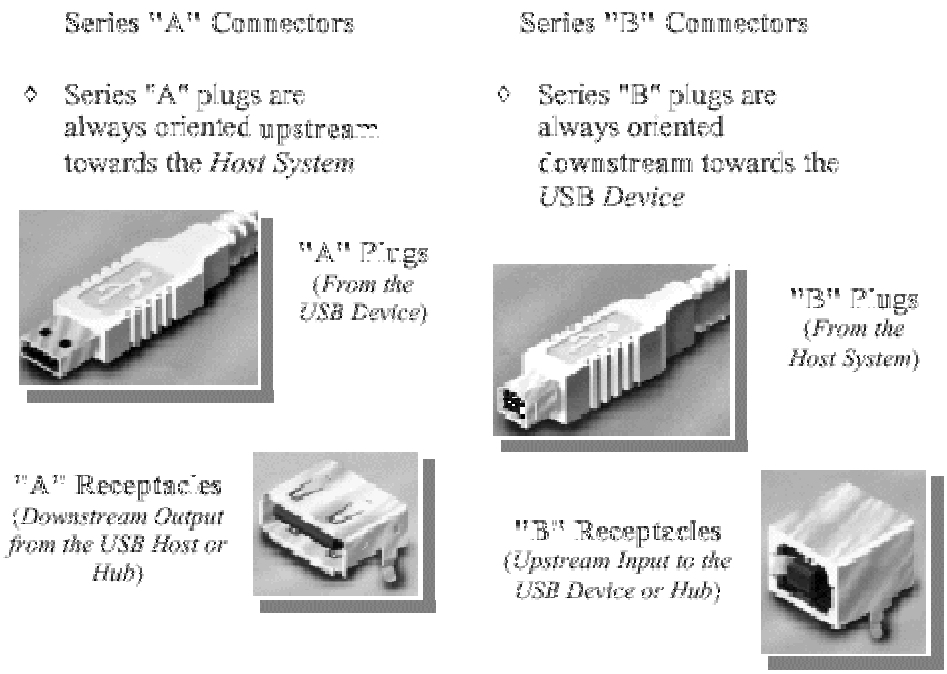
Para que la futura placa de desarrollo pueda comunicarse por USB, se necesitan cumplir una serie de requisitos físicos, como pueden ser la adaptación de impedancias correcta, o colocar correctamente las resistencias de pull-up que determinan la velocidad del dispositivo de cara al PC. La mayor parte de estos requisitos se cumplen colocando al final de la línea un transceptor USB, cuya única finalidad es ésa: adaptar la línea de transmisión para que cumpla las especificaciones USB, y de cara a la tarjeta, proporcionarnos unos valores de tensiones adecuados.

Por tanto, si suponemos que nuestro interfaz VHDL va a ir implementado sobre un dispositivo de lógica reconfigurable (CPLD o FPGA), el esquema que debemos tener será el siguiente:



El dispositivo de lógica reconfigurable deberá tener la capacidad suficiente para albergar nuestro diseño más el nivel de aplicación que se construya sobre él. Recordemos que nuestro circuito, incluyendo el nivel de aplicación que se ha diseñado a modo de ejemplo, tiene unas 8500 puertas lógicas.

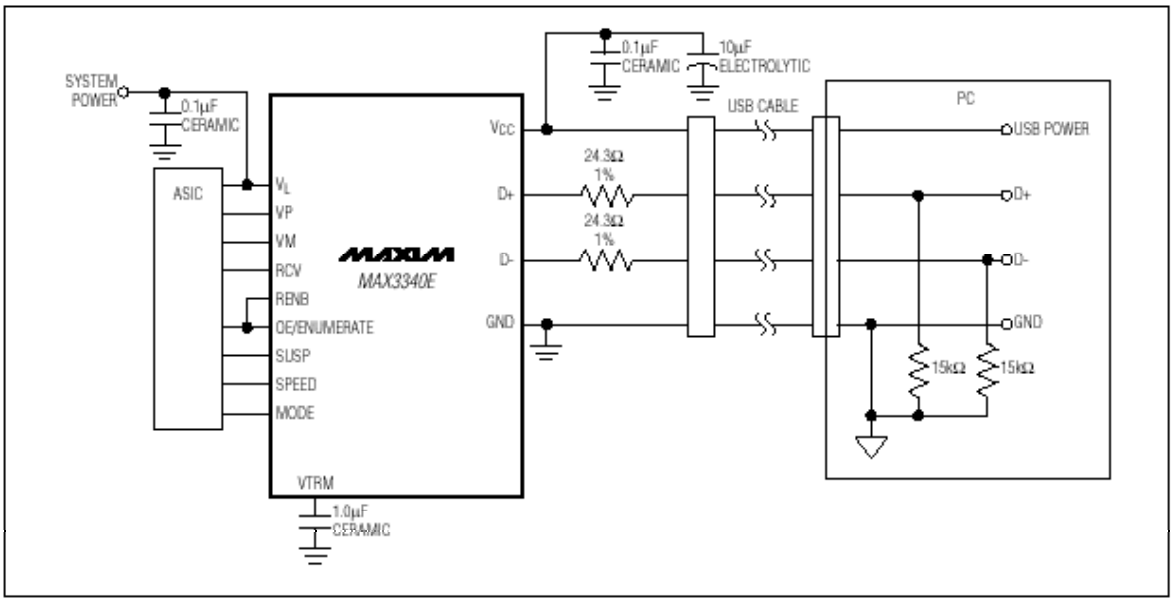
El conector USB será un receptáculo tipo A, de forma que en él entre el enchufe tipo A que se emplea para la comunicación con el PC.



De este modo, tendríamos que usar un cable tipo A-A para realizar la conexión con el PC. También podríamos usar un receptáculo tipo B y un cable A-B. Se recomienda usar el tipo A porque así también se pueden enchufar otros dispositivos a la placa, y ésta actuaría como *host* de la comunicación.

En cuanto al transceptor, en la tarjeta XSV800 se usa el modelo PDIUSBP11A de Philips. Por un lado tiene las 2 líneas de datos del bus USB (D+ y D-), que van directamente al conector, y por el otro lado las líneas que llegan al módulo *adaptador* de nuestro diseño. Además, tenemos dos líneas que determinan el modo de funcionamiento del dispositivo según se pongan a tierra o alimentación. (ver figura en página 162).

Para un nuevo diseño de placa, se recomienda usar el transceptor MAX3340E de *Maxim Integrated Products*, cuyo funcionamiento es muy similar al del producto de Philips, aunque ya lleva incluidas las resistencias de pull-up, de forma que al determinar en el chip el modo de funcionamiento a través de su patilla *speed*, éste conecta internamente la resistencia de pull-up a D+ o a D-, según se haya seleccionado "high speed" o "low speed". El esquemático quedaría como sigue:



Por lo tanto, habría que colocar dos resistencias de 24.3Ω entre el transceptor y el conector. También se podrían conectar resistencias de pull-down de 15KΩ entre las líneas D+ y tierra, y D- y tierra, con sus respectivos jumpers para conectarlas o quitarlas, como se ve en la figura de la página 162. Esto posibilitaría usar la placa como host USB.

10.2. Puntos a adaptar

Una vez tenemos el interfaz físico deseado, podemos construir sobre él el diseño de circuitería de trasvase de datos que queramos. En este punto vamos a exponer qué es lo que hay que tocar de nuestro diseño y qué no hay que tocar para construir sobre él la funcionalidad que queramos.

10.2.1. Adaptación del diseño VHDL

En la parte de VHDL, con este proyecto se entrega a modo de ejemplo un nivel de aplicación consistente en un búfer de 256 posiciones de RAM y la circuitería necesaria para hacer lecturas y escrituras. Esto es lo que habría que sustituir por otra circuitería que cumpla con los nuevos requisitos. Las instrucciones *lee datos* y *escribe datos* se podrían seguir manteniendo, pero adaptándolas a las nuevas necesidades. El búfer se sustituiría por la circuitería destino de los datos transmitidos. Por tanto, lo que hay que cambiar en cuanto al diseño VHDL es el nivel de aplicación.

10.2.2. Adaptación del software

Del mismo modo, la aplicación *USBLink* se entrega a modo de ejemplo de comunicación con el nivel de aplicación del interfaz VHDL. Por tanto, si cambiamos éste, debemos cambiar la aplicación para cumplir con los nuevos requisitos. Así, en cuanto al software lo que hay que cambiar es la aplicación. Se recomienda no tocar el driver a menos que queramos hacer mejoras en el propio interfaz USB.

10.3. Ampliación del interfaz

Con las adaptaciones explicadas en el punto anterior, conseguimos poder intercambiar información con las capacidades que se han expuesto a lo largo de esta memoria. Sin embargo, si estas prestaciones se quisiesen mejorar (fundamentalmente aumentar la tasa de transferencia), aquí se aportan algunas ideas que no se han llevado a cabo en este proyecto por la necesidad de obtener un producto que ocupase un área lo más reducida posible:

- Sustituir los tres búferes internos de 8 bytes cada uno que usa el interfaz por uno sólo. Esto es posible dado que no se accede a los tres simultáneamente. Sin embargo, hay que tener cuidado de no acceder al búfer común desde más de un

bloque a la vez. Esto reduciría el tamaño del circuito, aunque por otro lado se complicaría la circuitería de acceso a la RAM.

- En vez de usar la RAM que lleva la placa de desarrollo para implementar estos búferes, implementarlos en VHDL. Esto aumenta el tamaño del diseño pero probablemente aumente la velocidad del dispositivo, aunque ésta no es crítica. Esta solución solo sería útil para el caso en que no pudiésemos disponer de una RAM externa. En este caso, se recomienda adoptar también el punto anterior.

- Implementar un *pipe* adicional. Este pipe sería de tipo *bulk*, para aumentar la tasa de transferencia. Con esto dejaríamos el pipe de control (el único implementado en este proyecto) sólo para la configuración del dispositivo, y nos evitaríamos así desplazar una gran cantidad de información no útil (instrucciones, asentimientos, etc.). Sin embargo, esto complicaría mucho la lógica de control de nuestro dispositivo, dado que ahora tendría que atender a transacciones con tres direcciones diferentes (la del pipe de control, la del pipe *bulk* de entrada y la del pipe *bulk* de salida). Además, aumentaríamos el número de búferes necesarios.

- Evitar que el driver o la aplicación segmenten la información a transmitir, de modo que toda la información se intercambie en la misma transacción. Para ello, hay que hacer que el dispositivo lógico admita transferencias de más de 255 bytes. Esto implica incluir un mecanismo de control de errores que impida que se pierda la transacción completa. Con esto también ganamos en velocidad al acaparar todos los recursos del sistema y no liberarlos hasta el fin de la transferencia.

- Adaptar el dispositivo a la especificación 2.0 de USB, que incluye un nuevo modo de 480 Mb/s. Esto implicaría una remodelación más a fondo del interfaz.

11. Bibliografía

11.1. Bibliografía Impresa

En este apartado, he incluido tanto libros como documentos que se pueden obtener en la red. También se incluyen referencias a la documentación de algunos programas informáticos.

- **USB System Architecture.** Don Anderson. Ed. MindShare, Inc. Addison – Wesley.

- **Universal Serial Bus Specification Revision 1.1.** Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, NEC Corporation. 23 Septiembre 1998.

- **Full-Speed USB 1.1 Function Controller Product Specification.** Trenz Electronic.

- **XSV Board V1.0 Manual.** XESS Corporation.

- **Windows 98 DDK Documentation.** Microsoft. En especial, el documento *BULKUSB – Handling Asynchronous Bulk Data Transfer in a USB Minidriver*. Este documento se encuentra en el directorio C:\98DDK\src\usb\bulkusb\BULKUSB.HTM tras la instalación del DDK.

- **MSDN Library Visual Studio 6.0.** Microsoft.

- **Universal Serial Bus Specification Revision 2.0.** Compaq Computer Corporation, Hewlett-Packard Company, Intel Corporation, Lucent Technologies Inc, Microsoft Corporation, NEC Corporation, Koninklijke Philips Electronics N.V. 27 abril 2000.

- **Universal Host Controller Interface (UHCI). Design Guide. Revision 1.1**. Intel Corporation. Marzo 1996.

- **Open Host Controller Interface Specification for USB. Release: 1.0a.** Compaq Computer Corporation, Microsoft Corporation, National Semiconductor Corporation. 10 octubre 1996.

11.2. Recursos en la Web

- http://www.usb.org/developers/: Aquí se pueden obtener las diversas especificaciones USB.

- http://www.trenz-electronic.de: Han diseñado un interfaz USB en VHDL

- http://www.maxim-ic.com: Aquí se pueden obtener muestras de componentes electrónicos de MAXIM y Dallas Semiconductors.

- http://www.microsoft.com/ddk/ddk98.asp: de aquí se puede descargar el DDK de Windows 98.

- http://www.wingmanteam.com/usbsnoopy/: Para descargar el programa SniffUSB.

- http://www.usbman.com: aquí podemos obtener programas como el *Intel USB System Check.*

- http://www.microsoft.com/hwdev/usb/

- http://www.webopedia.com/TERM/U/USB.html

12. Indice