SISTEMAS INFORMÁTICOS 2008-2009

Sistema de comunicación de sistemas reconfigurables basado en PCI con sistemas de sobremesa

Memoria del Proyecto

18/09/2009

Autores:

Pablo Albaladejo Mestre
Gerardo Martín Moreno
Pablo Sánchez Escapa

Directores:

David Atienza Alonso
Miguel Peón Quirós

Departamento de Arquitectura de Computadores y Automática
Facultad de Informática

Índice

1 – Agradecimientos 5
2 – Resumen 6
3 – Abstract 6
4 – Palabras clave 7
5 – Introducción 8
6 – Acceso por DMA 11
7 – Puente PCI – Especificación PCI IP Core 13
7.1 – ¿Qué es un puente PCI? 13
7.2 – Introducción al PCI IP Core 13
7.3 – Descripción general 13
7.4 – Unidad Wishbone esclava 14
7.4.1 – Arquitectura de la unidad esclava Wishbone 14
7.5 – Unidad PCI objetivo 15
7.5.1 – Arquitectura de la unidad PCI objetivo 16
8 – Wishbone 18
8.1 – Funcionamiento General 18
8.1.1 – Operación de Reset 18
8.1.2 – Iniciación del ciclo de transferencia 18
8.1.3 – Protocolo Handshaking 18
8.2 – Ciclos de Lectura/Escritura simple 19
8.2.1 – Ciclo de Lectura simple 19
8.2.2 – Ciclo de Escritura simple 20
8.3 – Ciclos de Lectura/Escritura de bloque 21
8.3.1 – Ciclo de Lectura de bloque 22
8.3.2 – Ciclo de Escritura de bloque 24
8.4 – Fichero pci_user_constants 27
8.4.1 – Puente PCI 27
8.4.2 – Unidad Wishbone 28
8.4.3 – Cabecera de dispositivo 29
8.5 – Implementación 29
8.5.1 – Memoria 29
8.5.1 – Controlador 29
8.5.1 – Informe 30
9 – Driver en C para Windows 32
9.1 – Introducción 32
9.1.1 – Herramientas usadas 32
9.2 – Arquitectura de OS *Windows XP* 33

9.3 – Tipos de drivers en *Windows XP* 33
9.4 – Comunicación entre modo kernel y modo usuario 34
9.5 – Sistema de ficheros del Driver 38
9.6 – Funciones relativas al Driver 38
9.6.1 – DriverEntry 39
9.6.2 – AddDevice 40
9.6.3 – SSIIUnload 41
9.6.4 – CreateSSII 42
9.6.5 – CloseSSII 43
9.6.6 – ShutdownSSII 43
9.6.7 – WriteEvent 44
9.6.8 – HandlePnP 44
9.6.9 – HandleSystemControl 47
9.6.10 – ManejarIOControl 48
9.6.11 – SSIIDrvIOGetPCIResources 50
9.6.12 – SSIIDrvIOAccessDevMem 50
9.6.13 – SSIIDrvIOAccessDMA 51
9.7 – Archivo INF 52
9.7.1 – Sección Strings 53
9.7.2 – Sección VERSIÓN 54
9.7.3 – Sección MANUFACTER 55
9.7.4 – Archivos INF multiplataforma 56
9.7.5 – Sección de modelos 57
9.7.6 – Sección DDInstall 57
9.7.7 – Sección SourceDiskNames 59
9.7.8 – Sección SourceDiskFiles 59
9.7.9 – Sección ClassInstal32 59
10 – Aplicación de usuario 61
11 – Datos obtenidos y comparativas 62
11.1 – Datos de lectura del dispositivo implementado 62
11.2 – Datos de escritura del dispositivo implementado 62
11.3 – Comparativa 62
11.4 – Conclusiones de la implementación 63
12 – Bibliografía 65
13 – Licencia 66

1 – Agradecimientos

En primer lugar nos gustaría comenzar agradeciendo a David Atienza Alonso por brindarnos la posibilidad de realizar este proyecto que comenzamos con tanto entusiasmo y a Miguel Peón Quirós que, gracias a su constante apoyo y dedicación hasta el último momento, ha hecho posible la conclusión del mismo.

En especial nos gustaría enviar un especial agradecimiento a nuestros familiares y amigos, cuyo apoyo incondicional y sus palabras de ánimo en los momentos más difíciles, nos ayudaron a en la consecución del proyecto. Por último no podemos olvidar al personal de la facultad, los técnicos que nos dieron soporte cuando lo necesitamos y, en particular, al personal de seguridad que nos trató con tanta amabilidad y paciencia.

2 – Resumen

En este proyecto vamos a desarrollar un disco duro de estado sólido utilizando como soporte una tarjeta reconfigurable, o FPGA, con un chip SpartanII. Dada la limitación de lógica reconfigurable de la que disponemos vamos a desarrollar una memoria pequeña, de tan solo 2kbytes, pero que será suficiente para comprobar su funcionamiento y como ejercicio de comprensión y adquisición de experiencia en la elaboración de este tipo de componentes. La finalidad de este proyecto es cubrir todos los aspectos en el desarrollo de un componente periférico, desde su implementación en VHDL, su conexión y comunicación con el ordenador, hasta el acceso al mismo desde una aplicación de usuario.

Nos hemos servido del puente Wishbone, obtenido de OpenCores, para realizar la comunicación con el PCI. En la FPGA implementamos la memoria, la configuración y protocolos de comunicación necesarios para conectar con la interfaz del Wishbone y comunicarnos con el PC a través del bus PCI. Para hacer posible que el sistema sea reconocible por el computador, y pueda interactuar con éste, desarrollamos un driver que encapsula la funcionalidad necesaria para instalar el dispositivo y realizar las transacciones a través del controlador DMA. Para tratar de obtener el mayor rendimiento, el traspaso de datos entre nuestro sistema y la computadora se realiza a través del DMA que nos permite escribir y leer directamente de la memoria RAM con mayor rapidez y, por tanto, liberar al procesador de la carga masiva de interrupciones que le supondría intervenir en las transferencias. Por último desarrollamos una aplicación de usuario que interactúa con los datos existentes en la memoria RAM y envía peticiones de lectura y escritura, por sectores, a nuestro dispositivo que realiza las transacciones en bloques de 512bytes.

3 – Abstract

By this project we are developing a solid state hard drive using a Field Programmable Gate Array, or FPGA, with a SpartanII chip. As the reconfigurable logic available is not enough we will rise a small memory of only 2kbytes but large enough to check its functionality and as understanding and experiencing exercise in development of such components. The purpose of this project is to cover all aspects in the development of a peripheral component, its implementation in VHDL, its connection and communication with the computer, even the access through from software.

We use the bridge Wishbone, obtained from OpenCores, to guarantee the connection to the PCI. Memory, configuration and connection protocols required by the Wishbone interface are implemented within the FPGA card to establish the communication with the PC via the PCI bus. To enable the system to be recognized by the computer, and interact with it, we developed a driver with all the required functionality to install the device and perform transactions with the DMA controller. Data transfers between our system and the computer are granted by DMA increasing the performance of the device allowing us to read and write data directly to RAM memory faster and therefore, release the processor of being interrupted massively during transactions. Finally we developed user application software that interfaces with existing data within RAM memory and sends reading and writing requests to our device which performs transactions in 512bytes of block size.

4 – Palabras clave

FPGA, PCI, Wishbone, VHDL, Driver, DMA, Archivo INF, Bus, Maestro/Esclavo, Sistema reconfigurable, IP Core, BAR0, BAR1, DeviceIoControl, Controlador de dispositivo, Gestor I/O, DriverEntry, AddDevice, HandlePnP.

5 – Introducción

La motivación de este proyecto surge de nuestro interés por completar los conocimientos adquiridos a lo largo de la carrera sobre la arquitectura de computadores aplicándolos al desarrollo completo de un componente periférico: desde su implementación lógica hasta el desarrollo de una aplicación de acceso al mismo pasando por todo el proceso de comunicación del sistema con el PC a través del bus PCI.

Las asignaturas cursadas en la Ingeniería Informática se centran en el funcionamiento y la arquitectura de un computador y en alguna de ellas se hace breve mención al funcionamiento de componentes periféricos. Nuestro interés reside en la integración de una tarjeta configurable FPGA, a través del puerto PCI, con un computador actual sobre plataforma Windows.

Así como en otras asignaturas hemos aprendido a realizar diseños lógicos de circuitos, para poder integrar en éste tipo de tarjetas, consideramos adecuado completar todos estos conocimientos aplicándolos a un ámbito real y completándolos con la integración de los mismos en un sistema. Esta cuestión del desarrollo de componentes hardware no queda cubierta en la carrera ya que no hay ninguna asignatura que se centre en el desarrollo de drivers.

Para realizar la comunicación del sistema con el computador a través del bus PCI hemos utilizado el puente Wishbon-PCI de *Opencores.org*. Este proyecto, pretende aportar una interfaz estándar que abstraiga de los protocolos de comunicación del bus PCI y haga que los sistemas SoC sean portables y de fácil integración.

Nuestro proyecto utiliza el puente Wishbone-PCI para implementar un disco de estado sólido sobre la memoria RAM estática integrada en la propia FPGA, que sea accesible para una aplicación estándar Windows que se ejecute sobre el procesador del sistema. El mayor inconveniente de los discos duros magnéticos es el tiempo de búsqueda que, actualmente, está en el orden de milisegundos. En las últimas décadas, la velocidad de los componentes informáticos ha aumentado considerablemente pero este factor, al depender de factores físicos como la velocidad de rotación del disco, ha sufrido pocas mejoras. Los discos de estado sólido o, en nuestro caso, implementados sobre memoria RAM, permiten obtener tiempos de acceso del orden de microsegundos, varios órdenes de magnitud inferiores a los obtenidos con los discos magnéticos.

Es importante señalar que nuestro proyecto no se limita a utilizar parte de la memoria del PC para realizar un "disco RAM" sino que implementa toda la funcionalidad en hardware dedicado, utilizando recursos de DMA (Bus Master en el PCI) y memoria externa al sistema. Las ventajas de nuestra aproximación son varias:

- Al utilizar memoria externa, no se perjudica el rendimiento del sistema al no disminuir la cantidad de recursos disponibles para las aplicaciones.
- Al utilizar componentes hardware dedicados, el procesador no tiene que dedicar ciclos a realizar las transferencias.
- Al tratarse de memoria externa al PC, es posible alimentarla con baterías para que no pierda su contenido en casa de fallos eléctricos o, como está sucediendo recientemente a nivel comercial, sustituirla por módulos de memoria flash (no volátil).

Nuestro proyecto se compone de los siguientes pasos:

1.- Diseño, implementación y verificación del hardware que gestiona el disco RAM. Este desarrollo se ha realizado en VHDL sobre una Spartan-2 300, utilizando las herramientas de Xilinx. El mecanismo de comunicación es la conexión al bus Wishbone.

2.- Adaptación, configuración y conexión del "IP core" que hace de puente entre el bus PCI del sistema PC anfitrión y nuestro hardware, proporcionando un interfaz de bus Wishbone. Este desarrollo se integra en el proyecto VHDL del módulo anterior, utilizando las herramientas de Xilinx.

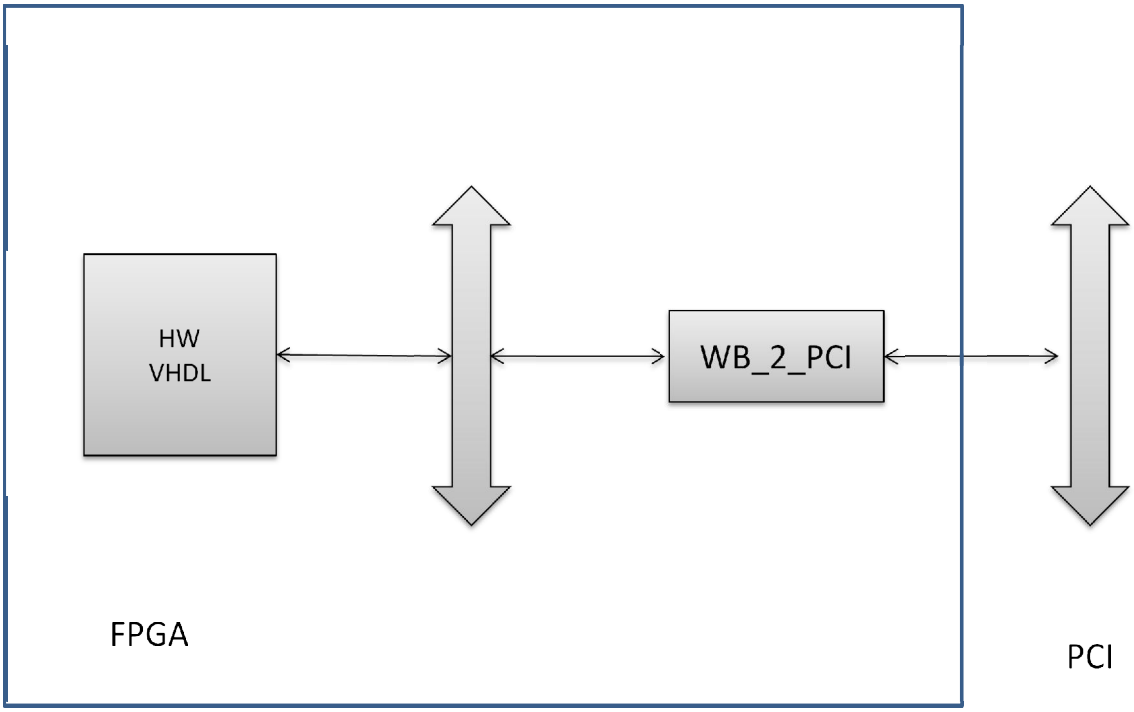
3.- Desarrollo, implementación y verificación de un controlador de dispositivo para Win32 que interactúe con el hardware y ofrezca a las aplicaciones un interfaz de acceso al hardware, a través del mecanismo de acceso a drivers del API Win32 (funciones IoCTL).

4.- Implementación de una aplicación de usuario, escrita en C++ del modo habitual, que haga uso de los servicios de nuestro hardware, a través del interfaz ofrecido por el controlador de dispositivo. Esta aplicación realiza transferencias de datos entre bloques de usuario y el disco externo, realizando mediciones del en el tiempo de acceso y la tasa de transferencia.

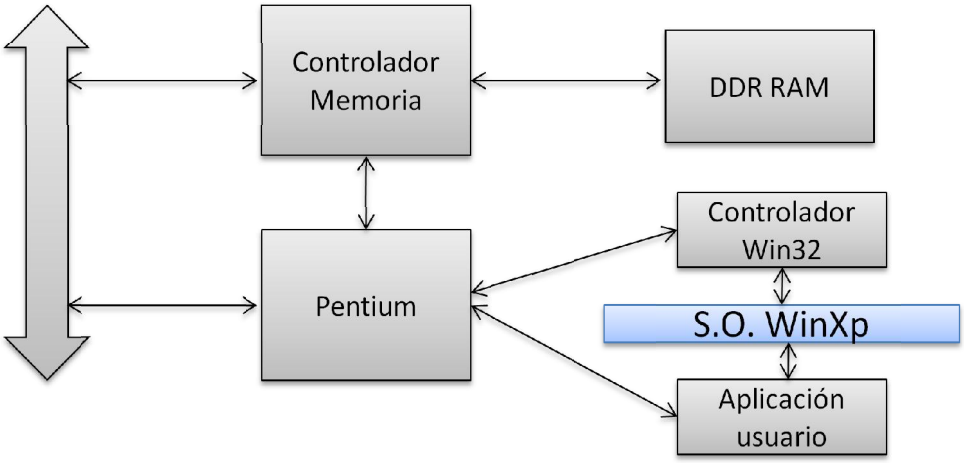
En los siguientes apartados se detalla el trabajo realizado en cada uno de los puntos anteriores.

La arquitectura del diseño global queda tal y como sigue:

- Conexión del HW con el bus PCI



- Conexión del PC al PCI



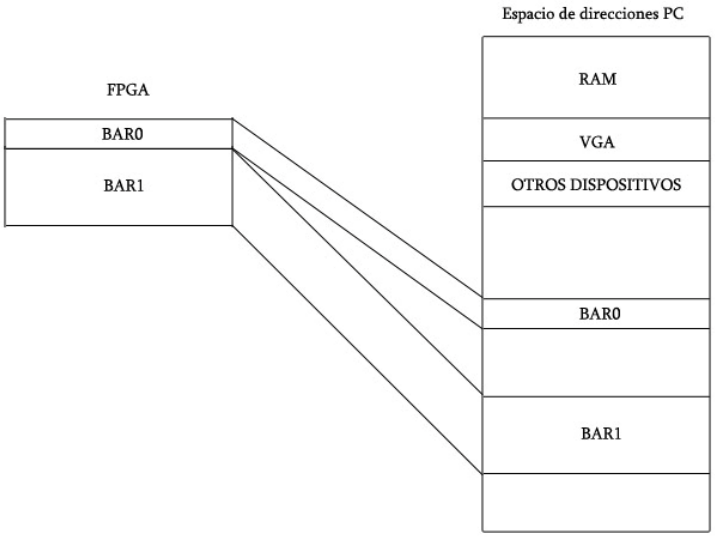
6 – Acceso por DMA

En este apartado se va a tratar de explicar la manera en la que se trata el espacio de direcciones del computador para que el dispositivo obtenga los recursos necesarios para su funcionamiento.

Utilizamos un computador que solamente trabaja con direcciones de 32 bits lo que supone que el sistema cuenta con un espacio de memoria de 2^32 palabras de 1 byte. Dado que no contamos con la intención de portar el sistema a un computador de 64 bits trabajaremos con direcciones de 32 bits.

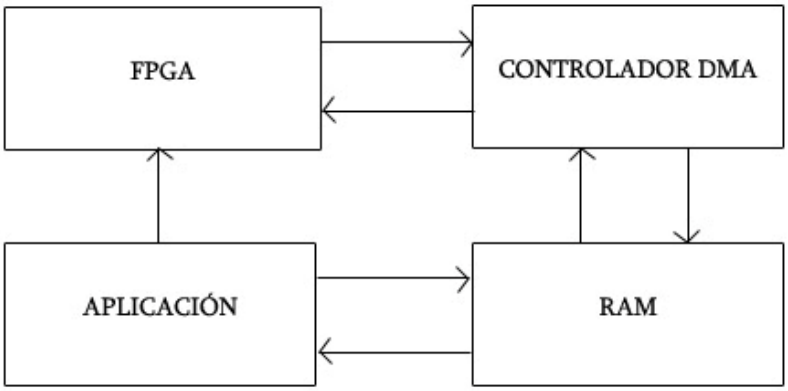
Hay que tener en cuenta que cuando nos referimos al espacio de memoria no hacemos referencia a la memoria física, o memoria RAM, sino a la que es capaz de direccionar el computador y que es utilizada tanto para ubicar la memoria RAM como otros componentes que soliciten espacio como la tarjeta de vídeo VGA o nuestro propio dispositivo.

Como se ha explicado en la sección del Wishbone nuestro sistema reserva dos espacios de direcciones, uno de 4 kbytes para el BAR0 o registros de configuración y otro de 32 Mbytes para la gestión de nuestro dispositivo o BAR1, que son asignados por el sistema operativo. Nuestro objetivo es desarrollar una aplicación que pueda trabajar con el dispositivo que hemos implementado por lo que tiene que ser capaz de comunicarse con él, sin embargo, no se puede acceder directamente a la memoria que tenemos en la tarjeta a través de la dirección física asignada sino que debemos solicitar una traducción a un espacio virtual con el que podamos trabajar. Desde nuestro driver podemos obtener una traducción lógica de las direcciones físicas e implementar sendas funciones de lectura y escritura accesibles desde la aplicación para realizar transferencias a y desde la tarjeta, no obstante, vamos a utilizar el controlador DMA de la placa base para realizar las transferencias aprovechando las ventajas de rendimiento que esto supone. Si utilizamos estas direcciones lógicas una operación de lectura y escritura estaría accediendo directamente a la memoria física que tenemos implementada en nuestra VGA haciendo uso del procesador del computador. Para evitarlo utilizamos DMA que, reservando un espacio de memoria físico en la RAM, nos permitirá realizar transferencias entre la tarjeta y la memoria RAM utilizando el controlador de DMA liberando al procesador de esta carga. No obstante utilizaremos escrituras directas para establecer registros de configuración en nuestra tarjeta.



Para poder utilizar DMA debemos solicitar un espacio contiguo de memoria en la RAM de 4 kbytes específico para este propósito. Al realizar la petición obtenemos dos direcciones, una que será transferida al dispositivo para que pueda acceder a la memoria y otra asignada al driver para el mismo propósito. Nuestra intención es poder acceder al espacio DMA y al BAR1 de nuestro dispositivo utilizando nuestra aplicación. ¿Qué problema supone esto? Supone que ya no trabajamos a nivel de núcleo como en el driver si no a nivel de usuario por lo que el espacio de direcciones accesible desde la aplicación está limitado por el propio espacio asignado al proceso. Esto quiere decir que no podemos utilizar directamente la dirección virtual obtenida en el driver sino que debemos solicitar que ésta sea mapeada a una dirección accesible desde el proceso. Por ello, implementamos en el driver una función que nos permita solicitar este mapeo y nos pase una traducción de las direcciones del BAR1 y del espacio DMA útiles desde la aplicación.

Una vez resueltas todas las cuestiones de direccionamiento el traspaso de información entre la tarjeta conectada al puerto PCI y nuestra aplicación se realiza mediante la memoria RAM y utilizando el controlador DMA. Nuestro dispositivo a petición de la aplicación escribirá datos en la RAM que serán accesibles desde el modo usuario, pudiendo modificarlos directamente y, ordenar a la tarjeta que los lea cuando sea necesario.



7 – Puente PCI – Especificación PCI IP Core

7.1 – ¿Qué es un puente PCI?

Los puentes PCI son usados en aplicaciones y dispositivos que quieren acceder a recursos proporcionados por un bus local PCI. Los sistemas que disponen de múltiples buses deben de proporcionar una interfaz que conecte el bus interno al bus PCI local, con el fin de habilitar las comunicaciones entre ambos. Los puentes PCI proporcionan dicha interfaz.

7.2 – Introducción al PCI IP Core

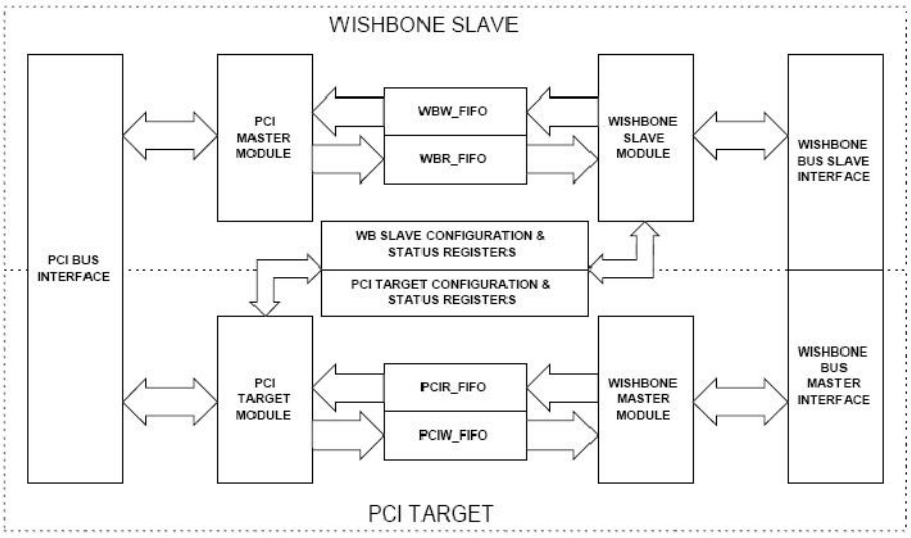
El PCI IP Core (puente PCI) proporciona una interfaz entre el bus System On Chip (SoC) Wishbone, y el bus PCI local. Se compone de dos unidades independientes, una que gestiona las transacciones originadas en el bus PCI, y otra que gestiona las transacciones originadas en el bus Wishbone.

7.3 – Descripción general

El puente PCI consiste en dos unidades: la unidad PCI objetivo, y la unidad Wishbone esclavo. Cada unidad dispone de su propio juego de funciones para soportar las operaciones de puente desde el Wishbone al PCI y del PCI al Wishbone. La unidad esclava Wishbone actúa como esclavo en la parte del Wishbone del puente, e inicia transacciones como maestro en la parte del bus. La unidad PCI objetivo actúa como objetivo en la parte PCI del puente y como maestro en su lado Wishbone. Ambas unidades operan como unidades independientes entre sí. La unidad objetivo PCI implementa el interfaz objetivo en el bus PCI y la interfaz maestra del bus Wishbone. La unidad esclava Wishbone implemente la unidad esclava del bus Wishbone y la interfaz maestra en el bus PCI.

Dicha interfaz PCI es compatible con la *especificación 2.2 PCI* y mientras que el Wishbone es compatible con una *Arquitectura de Interconexión Wishbone SoC Revisión B*. El Wishbone soporta operaciones de bus de 32 bits y de ninguna otra anchura.

La siguiente imagen ilustra una descripción general de la arquitectura:



Arquitectura del núcleo del puente PCI

7.4 – Unidad Wishbone esclava

Los elementos del bus Wishbone pueden acceder al bus PCI a través de la unidad esclava Wishbone. Se pueden usar entre una y cinco imágenes de configuración para acceder al espacio de direcciones del PCI.

Cada imagen se compone de:

- Registro de dirección base
- Registro de máscara de dirección
- Registro de traducción de dirección
- Registro de control de la imagen
- Decodificador

A la dirección base, almacenada en el registro de dirección base, se le enfrenta el valor de la máscara almacenada en el registro de máscara de dirección. El decodificador compara dicha dirección con la dirección de bus Wishbone para identificar ciclos válidos Wishbone. Si fuera necesario, la dirección Wishbone puede ser traducida a un valor diferente antes de acceder al bus PCI. El valor de una dirección presentada en el bus PCI es almacenada en los registros de traducción de dirección. El registro de control de imagen se usa para configurar el comportamiento de la imagen.

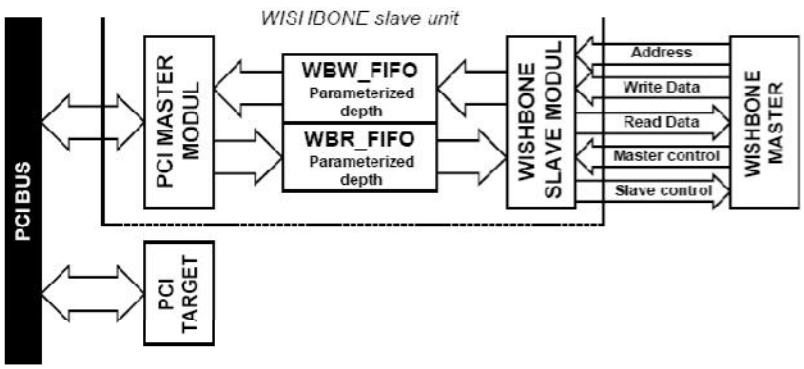
Cada imagen se puede configurar como memoria accesible o como espacio de dirección de entrada/salida en el bus PCI.

Los ciclos de escritura a través de la unidad esclava WB son procesados como postescrituras y los ciclos de lectura como lectura retardada. Las lecturas pueden ser pre-buscadas si el acceso a la imagen es configurado correctamente. La única excepción para esta regla es la configuración de escritura, la cual es iniciada por un mecanismo especial y por tanto queda fuera de esta explicación.

La Cola de Escritura Wishbone (WBW_FIFO) es usada para las postescrituras que actúan en el bus Wishbone; la Cola de Lectura Wishbone (WBR_FIFO) acumula las lecturas prebuscadas. La unidad esclava Wishbone conecta a las unidades maestras Wishbone actuando como esclavo, es decir, almacena los ciclos iniciados por un dispositivo maestro Wishbone que deben ser transmitidos al bus PCI.

7.4.1 – Arquitectura de la unidad esclava Wishbone

La unidad esclava Wishbone se compone de unos pocos elementos funcionales permitiendo al maestro Wishbone realizar accesos de lectura/escritura al bus PCI. La estructura de dichos elementos se muestra en la siguiente imagen.



Arquitectura de la unidad esclavo del Wishbone

7.4.1.1 – Módulo Wishbone esclavo

El módulo Wishbone esclavo, el cual incluye de una a seis unidades de imagen, es una interfaz esclava Wishbone de 32 bits como se define en la *Especificación Wishbone revisión 1B*. Este módulo maneja los ciclos de lectura/escritura hacia las imágenes del espacio de direcciones PCI y los accesos al espacio de configuración.

7.4.1.2 – WBW_FIFO

El módulo esclavo Wishbone usa la WBW_FIFO para el acceso a la memoria y para los ciclos de escritura de entrada/salida realizados por el Wishbone maestro. La profundidad parametrizada proporciona la opción de definir la WBW_FIFO de acuerdo con las necesidades específicas de la aplicación sobre el número de ciclos de escritura generados.

El bus Wishbone determina la velocidad de los ciclos de escritura hacia el WBW_FIFO, mientras que el bus PCI regula la velocidad de los ciclos de escritura desde el WBW_FIFO.

7.4.1.3 – WBR_FIFO

El módulo esclavo Wishbone usa una WBR_FIFO para almacenar datos leídos desde los objetivos PCI.

EL bus PCI determina la velocidad de los ciclos de lectura hacia la WBR_FIFO, y el bus Wishbone regula la velocidad de los ciclos de lectura desde la WBR_FIFO.

7.4.1.4 – Módulo Maestro PCI

El módulo maestro PCI usa la información proporcionada por el modulo esclavo Wishbone para realizar los ciclos de bus PCI. Es un iniciador de interfaz de 32 bits a 66MHz (33MHz en FPGA), compatible con las especificaciones de Bus Local PCI revisión 2.2.

7.5 – Unidad PCI objetivo

Los dispositivos PCI pueden acceder al bus Wishbone a través de la unidad objetivo PCI del puente, la cual proporciona de una a seis imágenes del espacio de memoria del lado del Wishbone. Cada imagen es seleccionada por una dirección producida durante la fase de dirección en el bus PCI. Dicha dirección es comparada con la dirección base a través de la máscara de dirección, cuyo valor está almacenado en los Registros de Configuración del PCI, y puede ser mapeada en memoria o en el espacio de estrada/salida. Una dirección también puede ser traducida a un valor almacenado en el registro de Traducción de Dirección, en el caso de que la imagen esté configurada correctamente.

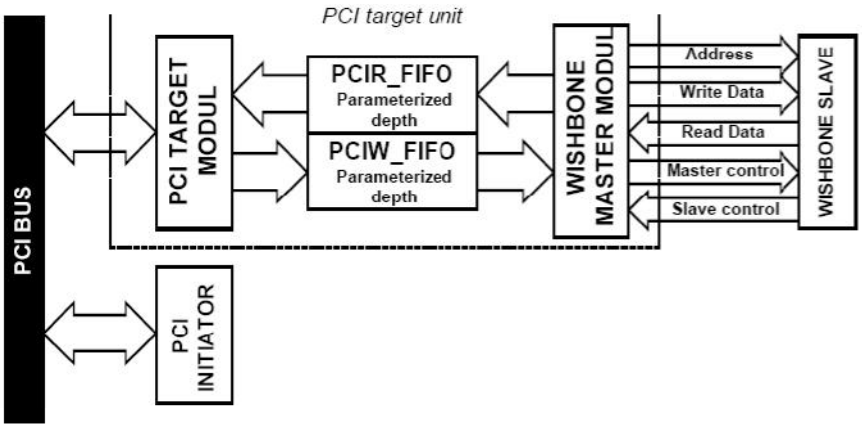
Los ciclos de escritura a través de la unidad PCI objetivo son tratados como postescrituras. Los ciclos de lectura pueden ser prebuscados.

La PCIW_FIFO almacena los ciclos de post escrituras; la PCIR_FIFO guarda los ciclos de lectura prebuscada.

7.5.1 – Arquitectura de la unidad PCI objetivo

La unidad objetivo PCI se compone de un conjunto de partes funcionales, las cuales permiten las inicializaciones PCI para llevar a cabo los accesos de lectura/escritura al bus Wishbone.

El módulo PCI objetivo es una interfaz objetivo compatible con la Especificación de Bus Local PCI revisión 2.2 de 32 bits a 66MHz (33Mhz en la FPGA), la cual incluye de dos a seis unidades de imagen para traducciones de dirección procedentes del bus PCI. Por tanto, este módulo gestiona los ciclos de lectura/escritura hacia el espacio de direcciones del Wishbone y los accesos a espacios de configuración.



Visión general de la arquitectura de la unidad objetivo PCI

7.5.1.1 Módulo PCI Objetivo

El módulo PCI objetivo usa una PCIW_FIFO para el acceso a memoria y ciclos de escritura de entrada/salida realizados por el iniciador PCI. La profundidad parametrizada ofrece la posibilidad de definir la PCIW_FIFO con acuerdo a las necesidades específicas de la aplicación para ajustar los ciclos de escritura.

El bus PCI determina la velocidad de los ciclos de escritura hacia en la PCIW_FIFO, mientras el bus Wishbone regula la velocidad de los ciclos de escritura desde la PCIW_FIFO.

7.5.1.2 PCIR_FIFO

El módulo maestro Wishbone usa una Cola PCIR_FIFO para almacenar los datos leídos desde los esclavos Wishbone.

El bus Wishbone determina la velocidad de los ciclos de lectura hacia la PCIR_FIFO, y el bus PCI regula la velocidad de los ciclos de lectura desde la PCIR_FIFO.

7.5.1.3 Módulo Wishbone Maestro

El módulo Wishbone maestro es una interfaz Wishbone maestro de 32 bits definida según la especificación Wishbone revisión 1B. A través del dispositivo maestro Wishbone, el núcleo envía las solicitudes al bus Wishbone.

El siguiente diagrama muestra la arquitectura completa del puente PCI

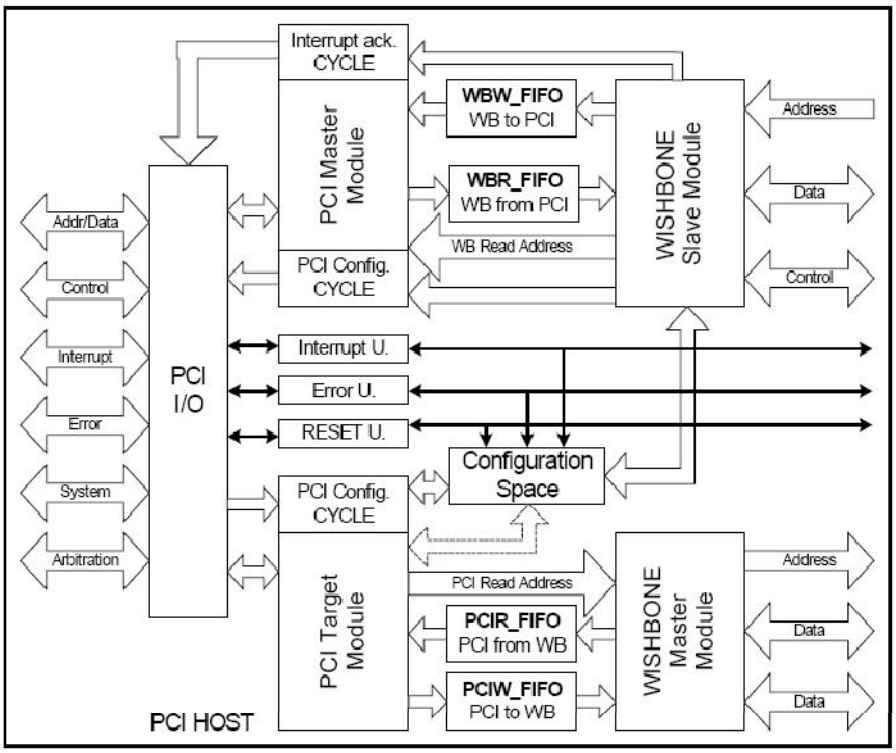


Diagrama completo del puente PCI del PCI IP Cores

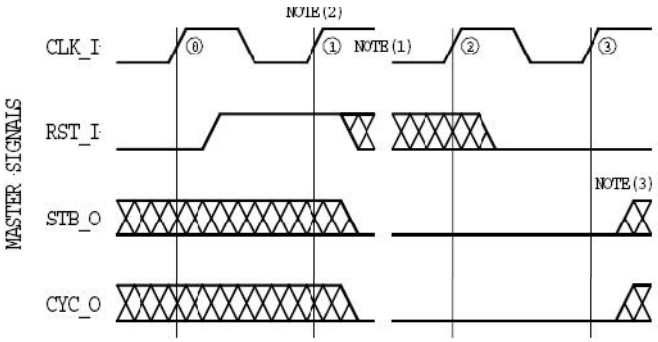
8 – Wishbone

8.1 – Funcionamiento General

Los interfaces maestro y esclavo están interconectados con un conjunto de señales que las permiten intercambiar datos. Estas señales se agrupan descriptivamente como un *bus*, y están contenidas dentro de un modulo funcional llamado INTERCON. Direcciones, datos y otros tipos de información están representados sobre este bus en forma de *ciclos de bus*.

8.1.1 – Operación de Reset

Todas las interfaces hardware son inicializadas en un estado predefinido. Esto se logra con la señal de reset [RST_O], la cual puede ser llamada en cualquier momento. También es usada para fines de testeo, inicializando todas las máquinas de estado con auto inicializado y los contadores que puedan ser usados en el diseño. La señal [RST_O] es controlada por el modulo SYSCON. Este modulo está conectado a todas las señales [RST_I] de todos los interfaces maestro y esclavo. La siguiente figura muestra el ciclo de reset:



8.1.2 – Iniciación del ciclo de transferencia

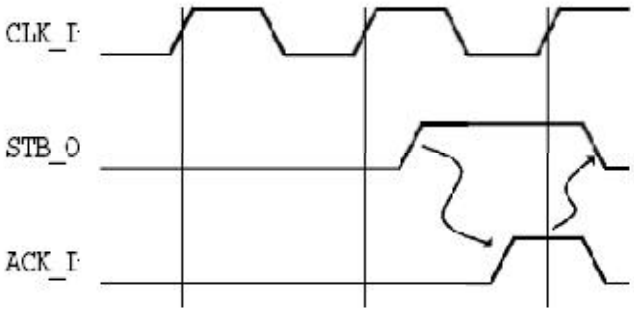
Las interfaces maestras inician un ciclo de transferencia enviando la señal [CYC_O]. Cuando [CYC_O] aparece negada, todas las otras señales del maestro son invalidadas.

Las interfaces esclavo responden a otra señal de un esclavo solo cuando [CYC_I] es enviada.

Las señales de SYSCOM y las respuestas hacia SYSCOM no se ven afectadas.

8.1.3 – Protocolo Handshaking

Todos los ciclos de bus usan un protocolo de Handshaking entre la interfaz maestro y la esclava. Como se muestra en la siguiente figura, el maestro envía la señal [STB_O] cuando está listo para transferir datos. La señal [STB_O] se mantiene hasta que el esclavo envía una de las señales de terminación de ciclo [ACK_I], [ERR_I] o [RTY_I]. En cada flanco de subida de [CLK_I] la señal de terminación es muestreada. Si esta es enviada, entonces [STB_O] se niega. Esto proporciona a ambas interfaces, maestro y esclavo, la posibilidad de controlar la velocidad a la que se transfieren los datos.



Protocolo Handshaking de bus local

8.2 – Ciclos de Lectura/Escritura simple

Los ciclos de lectura/escritura simple proporcionan un dato cada vez. Estos son los ciclos básicos usados para proporcionar las interconexiones en el Wishbone. En lo que sigue, la señal [CYC_O] no aparece para mantener los diagramas temporales lo más simples posible. Por ello, se asume que la señal [CYC_O] está continuamente activa.

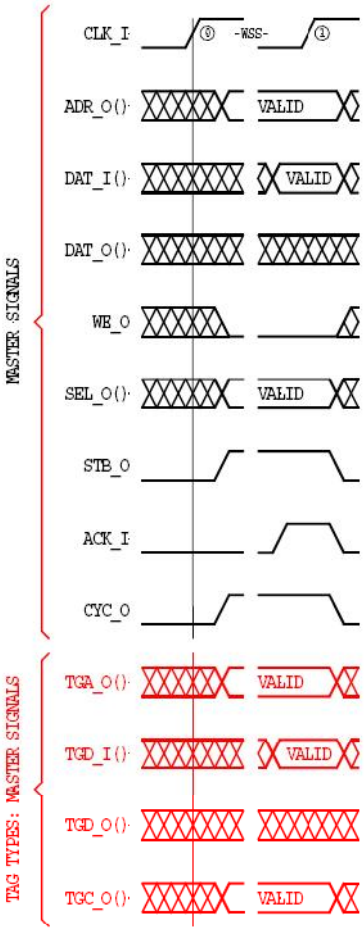
8.2.1 – Ciclo de Lectura simple

La siguiente figura muestra un ciclo de una lectura simple. El protocolo de bus funciona tal como sigue:

- Flanco de reloj 0:
  - El maestro presenta una dirección válida en [ADR_O()] y [TGA_O()].
  - El maestro niega [WE_O] para indicar que es un ciclo de lectura.
  - El maestro presenta en el banco de selección [SEL_O()] para indicar dónde se espera el dato.
  - El maestro envía [CYC_O] y [TGC_O()] para indicar el comienzo del ciclo.
  - El maestro envía [STB_O] para indicar el comienzo de la fase.

- Flanco de reloj 1. Setup:
  - El esclavo decodifica las entradas, y como respuesta envía [ACK_I].
  - El esclavo presenta una dato válido en [DAT_I()] y [TGD_I()].
  - El esclavo envía [ACK_I] en respuesta a [STB_O] para indicar un dato válido.
  - El maestro monitoriza [ACK_I], y se prepara para recibir los datos por [DAT_I()] y [TGD_I()].
  - Nota: el esclavo debe de insertar estados de espera antes de enviar [ACK_I], de este modo actúa en forma de cuello de botella. Varios estados pueden ser añadidos.

- Flanco de reloj 1:
  - El maestro mantiene la entrada en [DAT_I()] y [TGD_I()].
  - El maestro niega [STB_O] y [CYC_O] para indicar el final del ciclo.
  - El esclavo niega [ACK_I] en respuesta a la negación [STB_O]



Ciclo de lectura simple

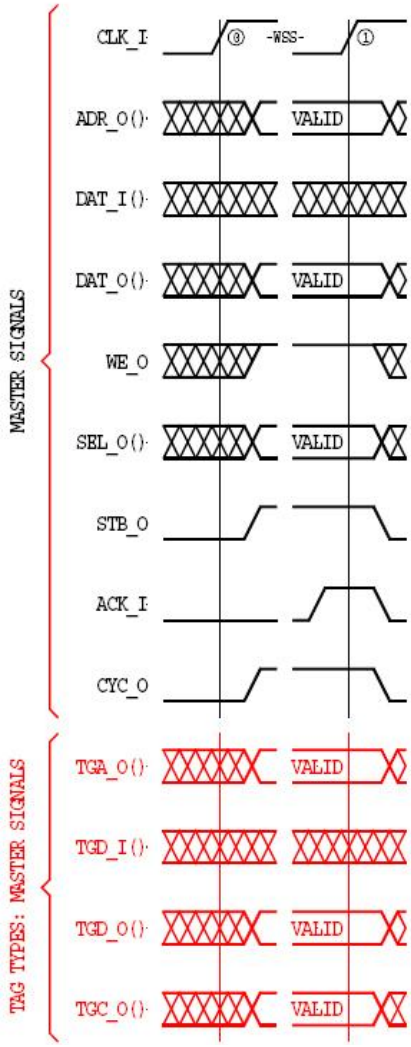
8.2.2 – Ciclo de Escritura simple

La siguiente figura muestra un ciclo de una lectura simple. El protocolo de bus funciona tal como sigue:

- Flanco de reloj 0:
  - El maestro presenta una dirección válida en [ADR_O()] y [TGA_O()].
  - El maestro envía [WE_O] para indicar un ciclo de escritura.
  - El maestro presenta la selección del banco [SEL_O()] para indicar donde es enviado el dato.
  - El maestro envía [CYC_O] y [TGC_O()] para indicar el inicio del ciclo.
  - El maestro envía [STB_O] para indicar el inicio de la fase.
  -
- Flanco de reloj 1. Setup:
  - El esclavo decodifica las entradas, y como respuesta el esclavo envía [ACK_I].
  - El esclavo se prepara para leer el dato en [DAT_O()] y [TGD_O()].
  - El esclavo envía [ACK_I] en respuesta a [STB_O] para indicar que el dato se está leyendo
  - El maestro monitoriza [ACK_I] y se prepara para terminar el ciclo.

- Nota: El esclavo debe insertar estados de espera antes de enviar [ACK_I], de modo que se puede generar un cuello de botella en la velocidad del ciclo.

- Flanco de reloj 1:
  - El esclavo deja los datos en [DAT_O()] y [TGD_O()].
  - El maestro niega [STB_O] y [CYC_O] para indicar el final del ciclo.
  - El esclavo niega [ACK_I] en respuesta a la señal negada [STB_O].



Ciclo de escritura simple

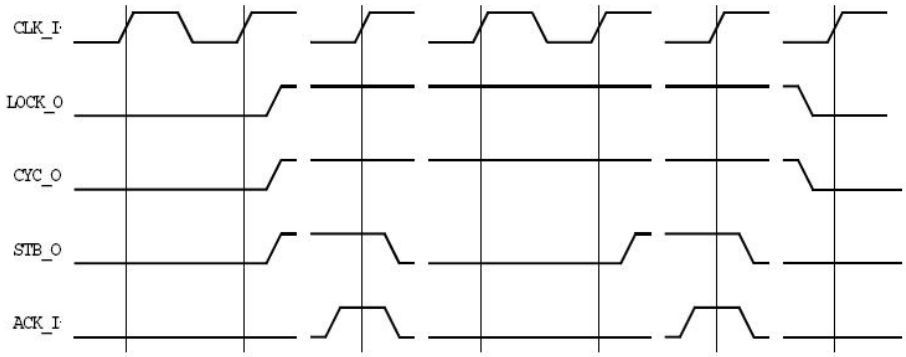
8.3 – Ciclos de Lectura/Escritura de bloque

Las transferencias de tamaño de bloque ofrecen transferencias de múltiples datos. Son muy similares a los ciclos de escritura y lectura simples, pero tienen unas pocas modificaciones para poder soportar las transferencias múltiples.

Durante los ciclos de bloque, la interfaz básicamente implementa ciclos de lectura/escritura como se describió anteriormente. Sin embargo, los ciclos de bloque han sido

modificados un poco para que esos ciclos individuales (denominados *fases*) sean combinados conjuntamente para formar un solo ciclo de bloque. Esta función es especialmente útil cuando múltiples maestros se usan en las interconexiones. Por ejemplo, si el esclavo actúa como memoria compartida (mediante un puerto dual), un árbitro para dicha memoria puede determinar cuando un maestro ha terminado su acceso y cualquier otro puede obtener el acceso a la memoria.

Como se muestra en la siguiente figura, la señal [CYC_O] es activada durante el ciclo de bloque. Esta señal puede ser usada para solicitar permiso de acceso a un recurso compartido a un árbitro local. Para mantener el acceso hasta el final del ciclo, la señal [LOCK_O] debe de permanecer activa. Durante cada una de las fases de transferencia de datos (entre la transferencia de bloque), el protocolo de handshaking habitual se mantiene entre las señales [STB_O] y [ACK_I].



Uso de la señal [CYC_O] durante los ciclos de bloque

8.3.1 – Ciclo de Lectura de bloque

La siguiente figura muestra el ciclo de lectura de un bloque. El ciclo de bloque es capaz de una transferencia de datos en cada ciclo de reloj. Sin embargo, este ejemplo también muestra cómo la interfaz maestro y esclavo pueden reducir la velocidad de transferencia de bus insertando estado de espera. Se muestran en total cinco transferencias (fases). Después de la segunda transferencia, el maestro inserta un estado de espera. Después de la cuarta transferencia, el esclavo también inserta un estado de espera. Dicho ciclo termina después de la quinta transferencia. El protocolo para esta transferencia funciona tal y como sigue:

- Flanco de reloj 0:
  - El maestro presenta una dirección válida en [ADR_O()] y [TGA_O()].
  - El maestro niega [WE_O] para indicar ciclo de lectura.
  - El maestro selecciona el banco con [SEL_O()] para indicar dónde se esperan los datos.
  - El maestro envía [CYC_O] y [TGC_O()] para indicar el comienzo del ciclo.
  - El maestro envía [STB_O] para indicar el inicio de la primera fase.
  - Nota: el maestro envía [CYC_O] y/o [TGC_O()] en el flanco de reloj 1 o en cualquier momento anterior.

- Flanco de reloj 1. Setup :
  - El esclavo decodifica las entradas, y el esclavo responde enviando [ACK_I].

- El esclavo presenta datos válidos en [DAT_I()] y [TGD_I()].
- El maestro monitoriza [ACK_I], y se prepara para leer de [DAT_I()] y [TGD_I()].

- Flanco de reloj 1:
  - El maestro lee los datos en [DAT_I()] y [TGD_I()].
  - El maestro presentas nuevas [ADR_O()] y [TGA_O()].
  - El maestro presenta una nueva selección de banco en [SEL_O()] para indicar dónde se esperan los datos.

- Flanco de reloj 2. Setup:
  - El esclavo descodifica las entradas, y responde enviando [ACK_I].
  - El esclavo presenta datos válidos en [DAT_I()] y [TGD_I()].
  - El maestro monitoriza [ACK_I], y se prepara para leer [DAT_I()] y [TGD_I()].

- Flanco de reloj 2:
  - El maestro lee los datos de [DAT_I()] y [TGD_I()].
  - El maestro niega [STB_O] para introducir un estado de espera (-WSM-).
  -
- Flanco de reloj 3. Setup:
  - El esclavo niega [ACK_I] en respuesta a [STB_O].
  - Nota: cualquier número de estados de espera pueden ser insertados por el maestro.

- Flanco de reloj 3:
  - EL maestro presenta nuevas direcciones en [ADR_O()] y [TGA_O()].
  - El maestro presenta una nueva selección de banco [SEL_O()] para indicar donde se esperan los datos.
  - El maestro envía [STB_O].

- Flanco de reloj 4. Setup:
  - El esclavo decodifica las entradas, y responde enviando [ACK_I].
  - El esclavo presenta datos válidos en [DAT_I()] y [TGD_I()].
  - El maestro monitoriza [ACK_I] y se prepara para leer de [DAT_I()] y TGD_I()].

- Flanco de reloj 4:
  - El maestro lee datos desde [DAT_I()] y [TGD_I()].
  - El maestro presenta [ADR_O()] y [TGA_O()].
  - El maestro presenta una nueva selección de banco en [SEL_O()] para indicar donde es esperan los datos.

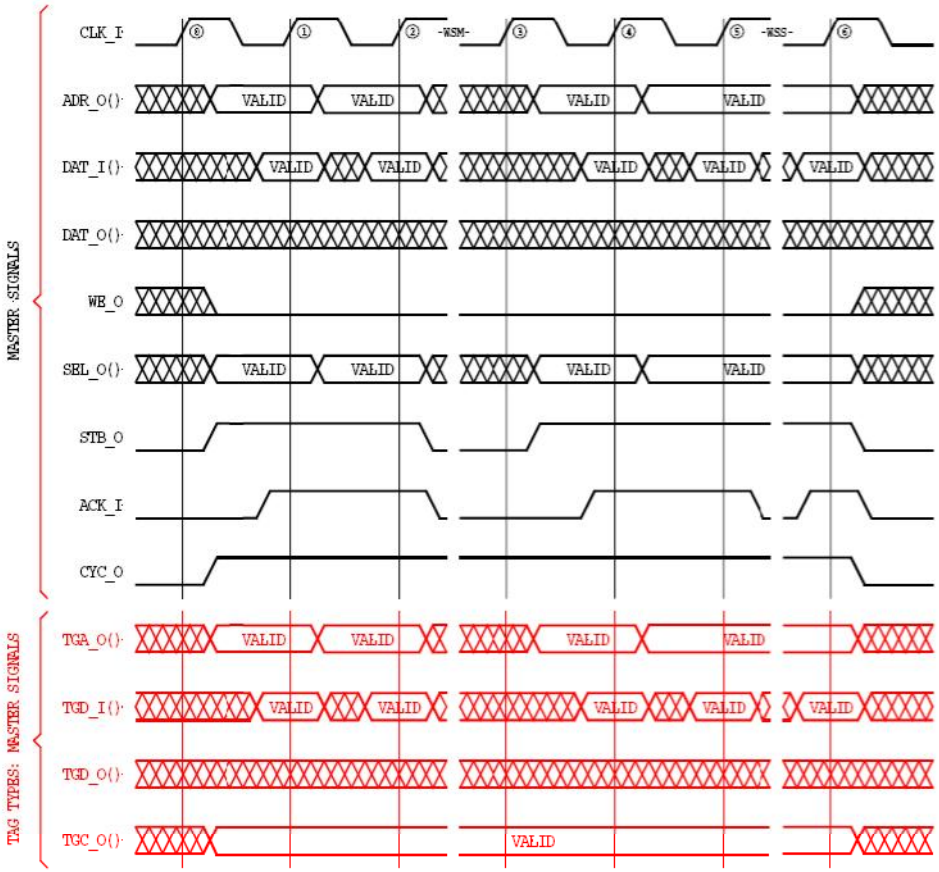
- Flanco de reloj 5. Setup:
  - El esclavo decodifica las entradas, y responde enviando [ACK_I].
  - El esclavo presenta datos válidos en [DAT_I()] y [TGD_I()].
  - El maestro monitoriza [ACK_I] y se prepara para leer [DAT_I()] y [TGD_I()].

- Flanco de reloj 5:
  - El maestro lee datos de [DAT_I()] y [TGD_I()].
  - El esclavo niega [ACK_I] para introducir un estado de espera.

- Nota: cualquier número de estados de espera pueden ser insertados por el esclavo en este punto.

- Flanco de reloj 6. Setup:
  - El esclavo decodifica las entradas, y responde enviando [ACK_I].
  - El esclavo presenta datos válidos en [DAT_I()] y [TGD_I()].
  - El maestro monitoriza [ACK_I] y se prepara para leer de [DAT_I()] y [TGD_I()].

- Flanco de reloj 6:
  - El maestro lee datos desde [DAT_I()] y [TGD_I()].
  - El maestro termina el ciclo negando [STB_O] y [CYC_O].



Ciclo de lectura de bloque

8.3.2 – Ciclo de Escritura de bloque

La siguiente figura muestra un ciclo de escritura de bloque. El ciclo de bloque es capaz de una transferencia de datos cada ciclo de reloj. Sin embargo, en este ejemplo también muestra cómo las interfaces maestro y el esclavo pueden reducir la velocidad de transferencia del bus por la inserción de estado de espera. Se muestran un total de cinco transferencias. Después de la segunda transferencia el maestro inserta un estado de espera. Después de la cuarta

transferencia el esclavo inserta un estado de espera. El ciclo termina en la quinta transferencia. El protocolo funciona tal y como se muestra a continuación:

- Flanco de reloj 0:
  - El maestro presenta [ADR_O()] y [TGA_O()].
  - El maestro envía [WE_O] para indicar que se trata de un ciclo de escritura.
  - El maestro presenta la selección de banco [SEL_O()] para indicar donde se envía el dato.
  - El maestro envía [CYC_O] y [TGC_O()] para indicar el comienzo del ciclo.
  - El maestro envía [STB_O].
  - Nota: el maestro envía [CYC_O] y/o [TGC_O()] en el flanco de reloj 1, o en cualquier instante anterior.

- Flanco de reloj 1. Setup:
  - El esclavo decodifica las entradas, y responde enviando [ACK_I].
  - El esclavo se prepara para leer los datos de [DAT_O()] y [TGD_O()].
  - El maestro monitoriza [ACK_I], y se prepara para terminar la fase de datos actual.
  -
- Flanco de reloj 1:
  - El esclavo lee los datos de [DAT_O()] y [TGD_O()].
  - El maestro presenta las direcciones en [ADR_O()] y [TGA_O()].
  - El maestro presenta una nueva selección de banco en [SEL_O()] para indicar donde es enviado el dato.

- Flanco de reloj 2. Setup:
  - El esclavo decodifica las entradas, y responde enviando [ACK_I].
  - El esclavo se prepara para leer de [DAT_O()] y [TGD_O()].
  - El maestro monitoriza [ACK_I],y se prepara para terminar la fase de datos actual.

- Flanco de reloj 2:
  - El esclavo lee desde [DAT_O()] y [TGD_O()].
  - El maestro niega [STB_O] para introducir un estado de espera.

- Flanco de reloj 3. Setup:
  - El esclavo envía [ACK_I] en respuesta a [STB_O].
  - Nota: cualquier número de estados de espera pueden ser introducidos por el maestro en este punto.

- Flanco de reloj 3:
  - El maestro presenta [ADR_O()] y [TGA_O()].
  - El maestro presenta la selección de banco en [SEL_O()] para indicar donde se mandan los datos.
  - El maestro envía [STB_O].

- Flanco de reloj 4. Setup:
  - El esclavo descodifica las entradas, y responde enviando [ACK_I].
  - El esclavo se prepara para leer datos desde [DAT_O()] y [TGD_O()].
  - El maestro monitoriza [ACK_I], y se prepara para terminar la fase de datos.

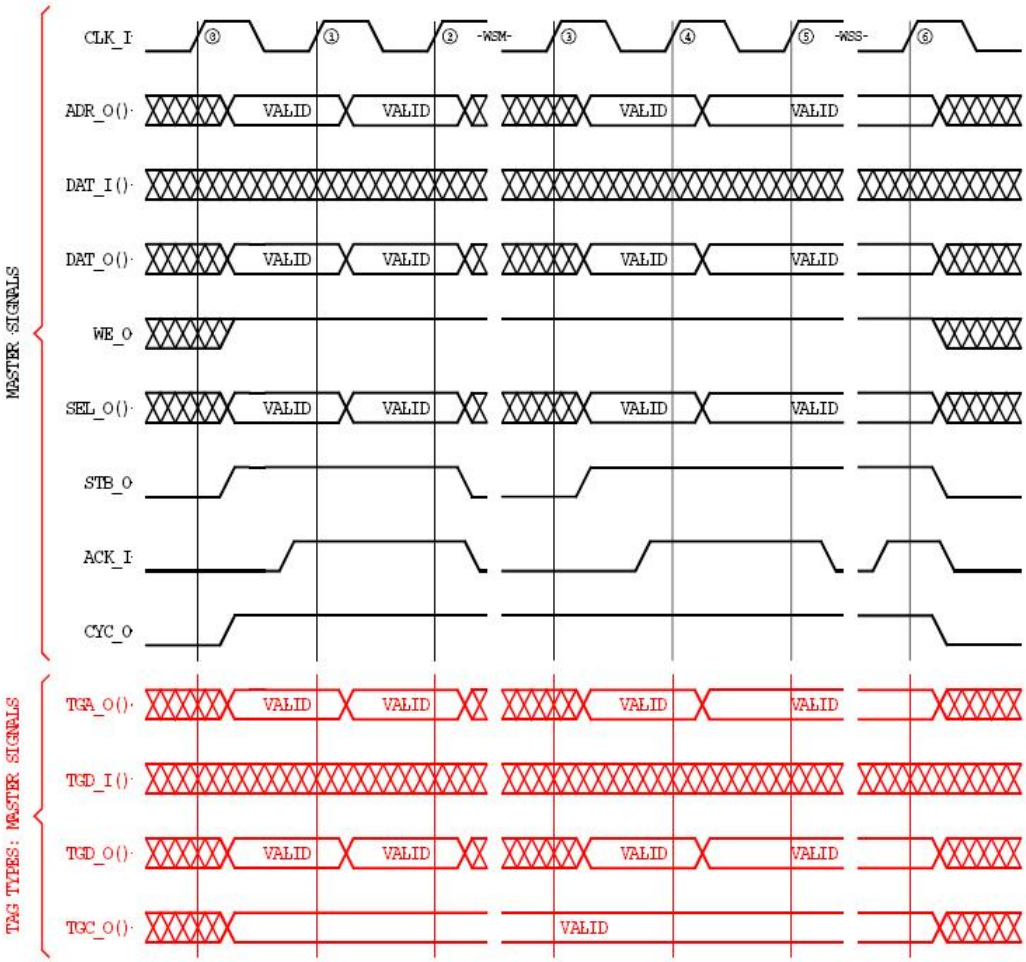
- Flanco de reloj 4:
    - El esclavo lee los datos de [DAT_O()] y [TGD_O()].
    - El maestro envía [ADR_O()] y [TGA_O()].
    - El maestro presenta una nueva selección del banco de datos [SEL_O()] para indicar donde son enviados los datos.

- Flanco de reloj 5. Setup:
    - El esclavo decodifica las entradas, y responde enviando [ACK_I].
    - El esclavo se prepara para leer desde [DAT_O()] y [TGD_O()].
    - El maestro monitoriza [ACK_I], y se prepara para terminar la fase de datos.

- Flanco de reloj 5:
    - El esclavo lee los datos desde [DAT_O()] y [TGD_O()].
    - El esclavo niega [ACK_I] para indicar un estado de espera.
    - Nota: cualquier número de estados de espera pueden ser insertandos por el esclavo en este punto.

- Flanco de reloj 6. Setup:
    - El esclavo decodifica las entradas, y responde enviando [ACK_I].
    - El esclavo se prepara para leer datos desde [DAT_O()] y [TGD_O()].
    - El maestro monitoriza [ACK_I], y se prepara para terminar la fase de datos.

- Flanco de reloj 6:
    - El esclavo lee datos desde [DAT_O()] y [TGD_O()].
    - El maestro termina el ciclo negando [STB_O] y [CYC_O].



Ciclo de escritura de bloque

8.4 – Fichero pci_user_constants

Como se ha comentado en puntos anteriores, un conjunto de parámetros tanto del puente PCI, como del bus Wishbone, pueden ser definidos y modificados para ajustar la arquitectura al caso particular de las necesidades de la aplicación.

Dichos parámetros se encuentran definidos en el fichero pci_user_constants.v proporcionado en el paquete de IP Cores.

A continuación se comentan brevemente los más relevantes en nuestro diseño hardware.

8.4.1 – Puente PCI

En el punto 5 de la memoria, PCI IP Core, se comenta que los tamaños de las estructuras son parametrizables, como por ejemplo las colas que implementan los accesos al bus.

Dicha configuración se realiza mediante la edición de la sección descrita a continuación dentro del fichero pci_user_constants.v.

Para las placas Xilinx, las instancias de las memorias tipos FIFOS se realiza mediante BRAMs. Debido a que se usan anchuras de 16 bits, se disponen de hasta 8 bits de dirección que se configuran con PCI_FIFO_RAM y WB_FIFO_RAM.

En este caso, WBW y WBR se refieren a las colas de escritura y de lectura que dispone la unidad Wishbone Esclava comentada en los puntos 7.4.1.2 y 7.4.1.3.

De la misma manera, PCIW y PCIR hacen referencia a las colas de escritura y lectura que dispone el módulo PCI objetivo comentado en los puntos 7.5.1.1 y 7.5.1.2.

```
`define WBW_ADDR_LENGTH 4
`define WBR_ADDR_LENGTH 4
`define PCIW_ADDR_LENGTH 3
`define PCIR_ADDR_LENGTH 3
`define PCI_FIFO_RAM_ADDR_LENGTH 8
`define WB_FIFO_RAM_ADDR_LENGTH 8
`define PCI_XILINX_RAMB4
`define WB_XILINX_RAMB4
```

Además, con la siguiente línea podemos configurar el tamaño de cada imagen del dispositivo en el PCI.

```
` define PCI_NUM_OF_DEC_ADDR_LINES 7
```

De esta manera, los siete primeros bits (los más significativos) de una dirección en el PCI se usan para decodificar el acceso, comparándolos con la dirección base. Por ello, cada imagen activa del dispositivo PCI tendrá un tamaño de 32MB (25 bits para desplazamiento dentro de la imagen). En nuestro caso, la única imagen implementada es BAR1.

Este parámetro influye en la velocidad de acceso, puesto que cuanto menor sea el número de líneas a decodificar, más rápido se efectuará.

8.4.2 – Unidad Wishbone

En este fichero, también se proporciona la posibilidad de configurar el espacio de memoria y de direcciones relativo a todo el dispositivo.

Es posible configurar la dirección base del bus Wishbone con:

```
`define WB_CONFIGURATION_BASE 20'h0000_0
```

Dicha configuración no puede ser cambiada en tiempo de ejecución.

Para poder acceder al bus sin ninguna configuración software, se pueden modificar las direcciones base de los registros BAR de cada imagen.

```
`define  WB_BA1    20'hEA00_0
`define  WB_BA2    24'h0000_00
`define  WB_BA3    24'h0000_00
`define  WB_BA4    24'h0000_00
`define  WB_BA5    24'h0000_00
```

En nuestro caso, la inicialización del WB_BA1, se hará en la fase de carga del driver por el sistema operativo, y cuando se disponga de la información de los recursos otorgados al dispositivo.

En este fichero también se pueden gestionar las máscaras de dirección dentro del bus Wishbone, pero como en nuestro caso no tenemos más que un elemento (una memoria) conectada al bus, no es necesario modificar los valores por defecto.

8.4.3 – Cabecera de dispositivo

Los siguientes parámetros permiten definir la cabecera del dispositivo PCI, tales como el identificador de Vendor, o el identificador de dispositivo. Ambos parámetros serán requeridos en fases posteriores, como por ejemplo en la programación del driver de dispositivo.

```
`define HEADER_VENDOR_ID         16'h1895
`define HEADER_DEVICE_ID         16'h0010
`define HEADER_REVISION_ID       8'h01
`define HEADER_SUBSYS_VENDOR_ID 16'h1895
`define HEADER_SUBSYS_ID         16'h0001
`define HEADER_MAX_LAT           8'h1a
`define HEADER_MIN_GNT           8'h08
```

8.5 – Implementación

A parte del fichero de configuración pci_user_constants, se han desarrollado dos módulos en VHDL: uno referente a la implementación de la memoria del disco, y otro referente al controlador.

8.5.1 – Memoria

El diseño de la memoria que implementa el dispositivo FPGA corresponde a una RAM de 4KB, divido en 4 sectores de 128 palabras de 4 bytes. Por ello, la unidad de transferencia mínima entre la FPGA, el DMA y la aplicación queda definida por sectores de 512 Bytes, que resultan del mismo tamaño de sector que los discos duros convencionales.

Por otro lado, el disco RAM ofrece un banco de 32 registros de 32 bits, donde se sitúan los flags de control del dispositivo. Dichos registros, son accesibles desde la memoria del procesador.

8.5.1 – Controlador

El módulo controlador incluido en el diseño VHDL gestiona las comunicaciones de los módulos Maestro y Esclavo con el bus Wishbone. Implementa las máquinas de estados descritas por la especificación de lectura de bloque de los apartados anteriores y es el encargado de gestionar las acciones entre el modulo Maestro con el acceso DMA.

8.5.1 – Informe

El reporte de la síntesis e implementación que devuelve ISE, con las estadísticas de utilización, es el que sigue:

proyecto Project Status (09/17/2009 - 07:11:12)			
Project File:	proyecto.ise	**Current State:**	Programming File Generated
Module Name:	top_arena	• **Errors:**	No Errors
Target Device:	xc2s300e-6fg456	• **Warnings:**	751 Warnings
Product Version:	ISE 10.1.03 - WebPACK	• **Routing Results:**	All Signals Completely Routed
Design Goal:	Balanced	• **Timing Constraints:**	All Constraints Met
Design Strategy:	Xilinx Default (unlocked)	• **Final Timing Score:**	0 (Timing Report)

proyecto Partition Summary	[-]
No partition information was found.	

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	1,417	6,144	23%	
Number used as Flip Flops	1,411			
Number used as Latches	6			
Number of 4 input LUTs	2,343	6,144	38%	
Logic Distribution				
Number of occupied Slices	2,273	3,072	73%	
Number of Slices containing only related logic	2,273	2,273	100%	
Number of Slices containing unrelated logic	0	2,273	0%	
Total Number of 4 input LUTs	3,604	6,144	58%	
Number used as logic	2,343			
Number used as a route-thru	173			
Number used for 32x1 RAMs	1,088			
Number of bonded IOBs				
Number of bonded	57	325	17%	
Number of Block RAMs	12	16	75%	
Number of GCLKs	2	4	50%	
Number of GCLKIOBs	2	4	50%	

Performance Summary			[-]
Final Timing Score:	0	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

9 – Driver en C para Windows

9.1 – Introducción

El controlador de dispositivo (driver) permite al SO interactuar con un periférico, en nuestro caso la FPGA, proporcionando una interfaz para poder utilizarlo cómodamente abstrayéndose del funcionamiento interno del periférico.

Para realizar la comunicación entre el modo kernel y el modo usuario se emplea la función del API, DeviceIoControl.

La función DeviceIoControl envía un código de control directamente al driver del dispositivo especificado y este realiza las operaciones correspondientes. La funcionalidad concreta se explicara más en detalle más adelante.

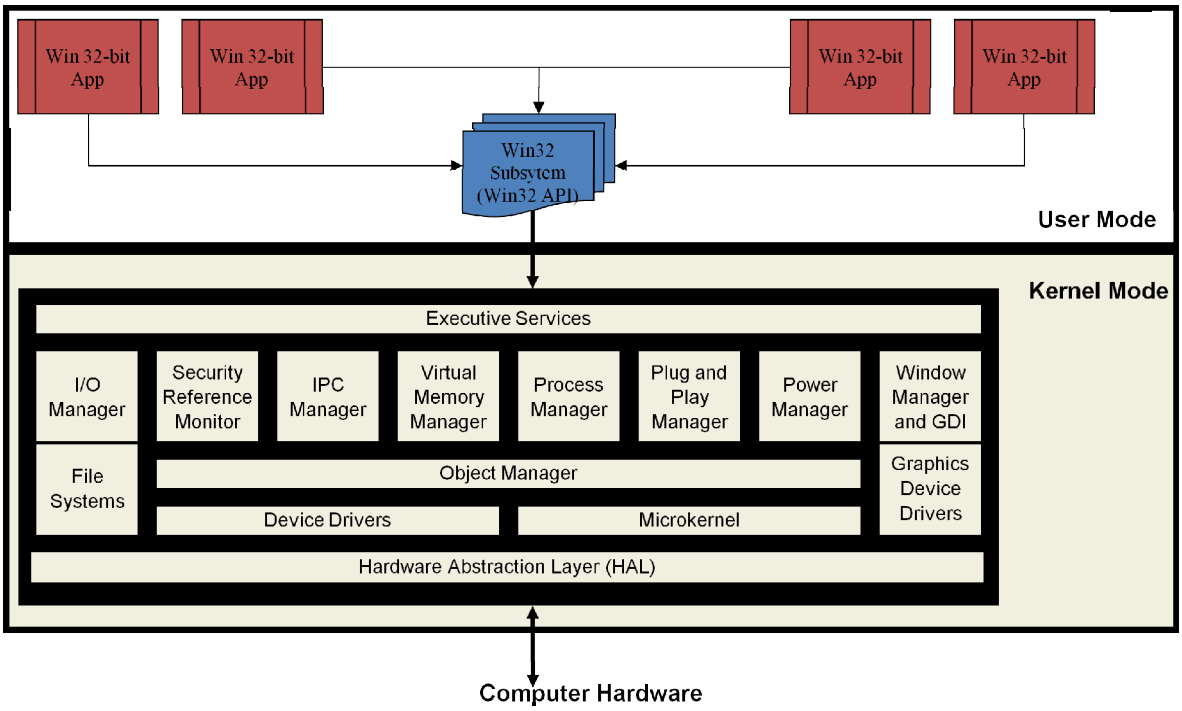
9.1.1 – Herramientas usadas

Las herramientas usadas para la realización del driver son:

- Windows Driver Kit (WDK)
    - El WDK es un completo sistema de desarrollo de controladores que contiene el Windows Driver kit (DDK) y las pruebas para la estabilidad y la fiabilidad de los controladores de Windows, incluyendo las siguientes:
        - Windows Driver Foundation (WDF)
            - ✓ Simplifica el desarrollo y el soporte de los controladores de Windows.
            - ✓ Header file refactoring (Windows Vista y versiones posteriores)
            - ✓ Reduce la complejidad del archivo de cabecera a través de una estructura más simple de directorios, eliminando los conflictos de declaraciones, y utilizando un único conjunto de archivos de cabecera para todas las versiones de Windows.
        - Installable File System (IFS) Kit
            - Incluye cabeceras, bibliotecas, ejemplos, y la documentación que se distribuyen como parte de la WDK.
        - Herramientas de verificación y análisis estático
            - Como PREfast y Static Driver Verifier, que ayudan a encontrar errores en tiempo de compilación.

9.2 – Arquitectura de OS *Windows XP*

En el siguiente diagrama se puede observar la arquitectura completa de Windows XP



Fuente: Programming the Microsoft Windows Driver Model

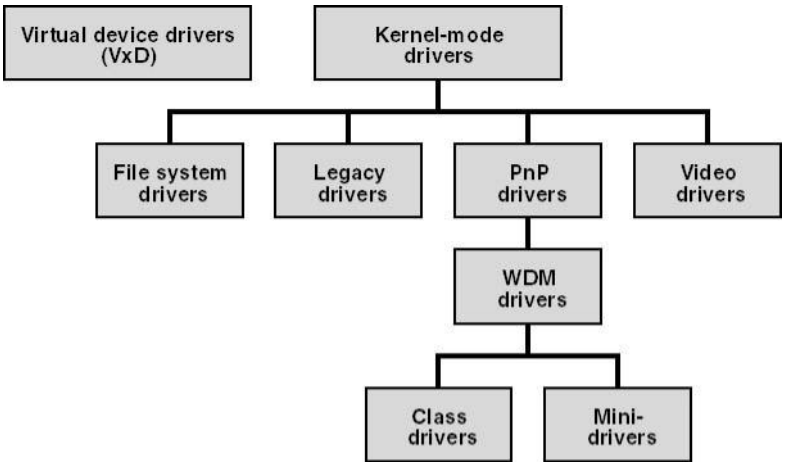
Como se puede observar en el esquema de la arquitectura Windows XP está estructurado en capas:

- User Mode(Modo usuario):
    - La capa más cercana al usuario, en esta capa se encuentra las aplicaciones de usuario y programas de soporte para aplicaciones conocido como Windows XP Subsystems, que incluye servicios para todas las aplicaciones como servicios cliente, servidor o de seguridad.
- KernelMode(modo núcleo):
    - La capa más cercana al hardware, en esta capa encontramos los servicios del SO, los controladores de dispositivo y el micro núcleo entre otros.

Esta estructura separa las aplicaciones de usuario del núcleo del sistema y sobre todo del hardware dejando la comunicación entre capas limitada a un conjunto de llamadas que proporciona la API del sistema operativo.

9.3 – Tipos de drivers en *Windows XP*

En la siguiente imagen podemos ver una clasificación de los driver en un sistema Windows:

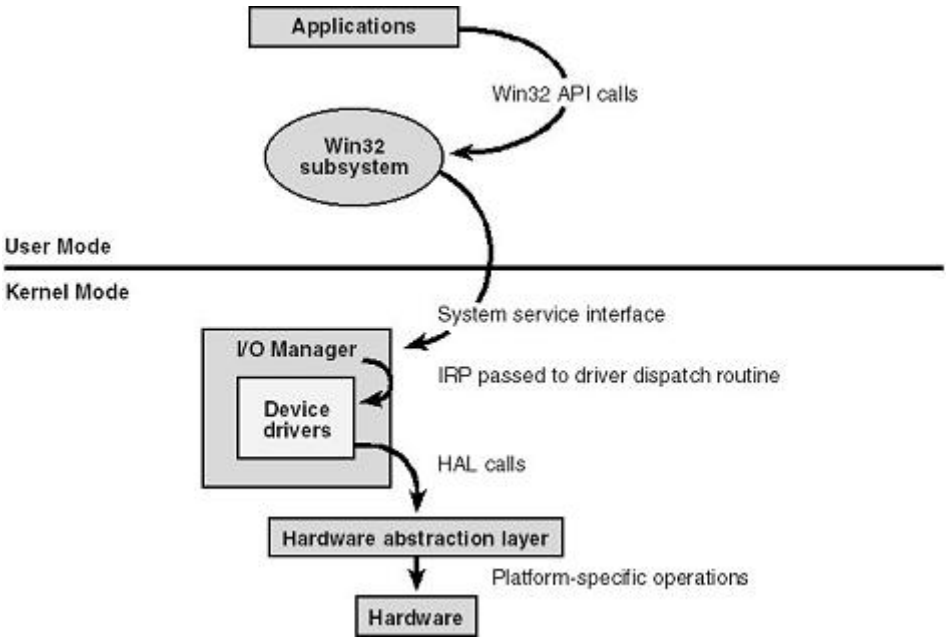


Fuente: Programming the Microsoft Windows Driver Model

El driver implementado en nuestro caso entra dentro de la clase PnP drivers que es un tipo de controlador de modo núcleo que incorpora el protocolo Plug and Play de Windows.

9.4 – Comunicación entre modo kernel y modo usuario

En la siguiente imagen vemos un esquema de cómo funciona la comunicación desde una aplicación en modo usuario hasta el hardware.



Fuente: Programming the Microsoft Windows Driver Model

La aplicación de usuario usa una llamada de la API para poder acceder al dispositivo, esta llamada llega al Gestor de entrada/salida que ya opera en modo núcleo para poder interactuar con el dispositivo. Este gestor valida los parámetros para garantizar que la operación no va a generar ningún fallo de seguridad, ni intenta acceder a datos para los cuales el usuario no tiene

permisos. A continuación crea una estructura de datos denominada I/O Request Packet (IRP) que pasa a un punto de entrada del controlador del dispositivo correspondiente.

El comportamiento de la rutina de tratamiento del controlador puede variar, pero lo normal es que devuelva a la aplicación de modo usuario una indicación del estado de la operación a realizar, en función de esta indicación (puede ser error, se está ejecutando la petición, la petición ha terminado, etc.) la aplicación de usuario dependiendo de la respuesta del controlador actuara en consecuencia.

Los controladores a pesar de que ejecutan en modo kernel y pueden comunicarse directamente con el hardware, usan los mecanismos que proporciona la capa de abstracción de hardware (HAL) para acceder al hardware. La rutina HAL usa un método dependiente de la plataforma.

Cuando el controlador termina con una operación de entrada/salida completa el IRP con una llamada a una rutina de servicio particular del modo kernel. Esta finalización es el último paso y permite a la aplicación que está esperando la respuesta, recibir el resumen de la ejecución de la operación de Entrada/Salida.

Como se comentó en la introducción, en la construcción del driver, para implementar la comunicación entre modo kernel y modo usuario se optó por la utilización de la función API DeviceIoControl.

La función DeviceIoControl tiene la siguiente sintaxis:

```
BOOL WINAPI DeviceIoControl(
  __in         HANDLE hDevice,
  __in         DWORD dwIoControlCode,
  __in_opt     LPVOID lpInBuffer,
  __in         DWORD nInBufferSize,
  __out_opt    LPVOID lpOutBuffer,
  __in         DWORD nOutBufferSize,
  __out_opt    LPDWORD lpBytesReturned,
  __inout_opt  LPOVERLAPPED lpOverlapped);
```

Esta función envía un código de control a un controlador de dispositivo, haciendo que el dispositivo realice la operación correspondiente. La función devuelve un buffer de salida que puede utilizarse para devolver resultados hacia la aplicación que ejecuta en modo usuario.

El prototipo de llamada a esta función desde una aplicación de usuario es el siguiente:

```
result = DeviceIoControl(Handle,Code,InputData,InputLength,
  OutputData, OutputLength, &Feedback, &Overlapped);
```

Handle es un identificador abierto a un dispositivo, este manejador se obtiene llamando a la función CreateFile, un prototipo de esta llamada en una aplicación de usuario sería el siguiente:

```
Handle    =    CreateFile("\\\\.\\IOCTL",    GENERIC_READ    |
GENERIC_WRITE,
```

```
  0, NULL, OPEN_EXISTING, flags, NULL);
if (Handle == INVALID_HANDLE_VALUE)
  <error>
...
CloseHandle(Handle);
```

Mientras se mantenga el identificado, se pueden hacer llamadas a ReadFile, WriteFile, o DeviceIoControl. Cuando haya terminado de acceso del dispositivo, de forma explícita, se cierra el identificador llamando a CloseHandle.

Argumentos de la llamada DeviceIoControl:

- Code(DWORD)
  - Es un código de control que indica qué operación de control se desea realizar

- InputData(PVOID)
  - Tipo de datos que se envía al controlador del dispositivo.

- InputLength(DWORD)
  - Indica el tamaño de los datos que se pasan en la llamada DeviceIoControl en el argumento InputData.

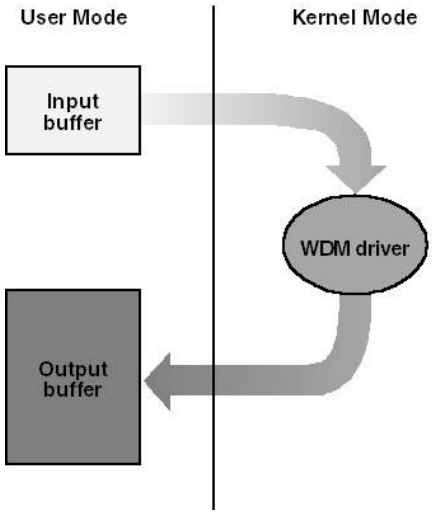
- OutputData(PVOID)
  - Tipo de datos que el controlador puede rellenar total o parcialmente con la información que desea enviar de vuelta a la aplicación que realizo la llamada DeviceIoControl.

- OutputLength
  - Indica el tamaño de los datos que se pasan el la llamada DeviceIoControl en el argumento OutputData.

- Feedback(DWORD)
  - Esta variable de actualiza desde el controlador e indica cuántos bytes de datos de salida se devuelven.

- Overlapped(OVERLAPPED)
  - Esta estructura se utiliza para ayudar a controlar una operación asíncrona. Si se ha especificado el flag FILE_FLAG_OVERLAPPED en la llamada CreateFile(), se debe especificar el puntero a la estructura OVERLAPPED. En caso contrario se puede pasar como nulo (NULL) este último argumento, ya que el sistema lo ignorara de todos modos.

La siguiente imagen ilustra la comunicación entre la aplicación y el driver.



Fuente: Programming the Microsoft Windows Driver Model

La estructura utilizada para la comunicación es la estructura IRP, esta estructura, que ya se escuetamente comento anteriormente, representa un paquete de peticiones de entrada/salida, esta estructura la utiliza el Gestor de entrada/salida (I/O Manager) para comunicarse con los controladores, y los mismos controladores para comunicarse con ellos.

Un paquete IRP consta de dos partes diferentes:

- Cabecera o parte fija del paquete.
  - Esta parte es utilizada por el Gestor de entrada/salida para almacenar información sobre la solicitud original, tales como los parámetros que se han pasado en la llamada original, la dirección del objeto que realizo la llamada, etc.

- Las posiciones de la pila entrada/salida:
  - Después de la cabecera se encuentra un conjunto de posiciones de la pila de entrada/salida, una por cada controlador vinculado con la petición. Cada posición contiene los parámetros, códigos de función, y el contexto que utiliza el controlador correspondiente para determinar lo que debe hacer.

El siguiente código resume la parte de la estructura del IRP que puede usar un driver.

```
typedef struct _IRP
  PMDL  MdlAddress;
  ULONG  Flags;

  union {
    struct _IRP  *MasterIrp;
    PVOID  SystemBuffer;
  } AssociatedIrp;

  IO_STATUS_BLOCK  IoStatus;
  KPROCESSOR_MODE  RequestorMode;
  BOOLEAN PendingReturned;
  BOOLEAN  Cancel;
  KIRQL  CancelIrql;
  PDRIVER_CANCEL  CancelRoutine;
```

```
  PVOID UserBuffer;

  union {
    struct {
    union {
      KDEVICE_QUEUE_ENTRY DeviceQueueEntry;
      struct {
        PVOID  DriverContext[4];
      };
    };
    PETHREAD  Thread;
    LIST_ENTRY  ListEntry;
    } Overlay;
  } Tail;
} IRP, *PIRP;
```

No se va comentar cada parte de la estructura del IRP, solamente la parte que es relevante para el driver que implementamos, para obtener información más concreta, en el apartado bibliografía, se pueden encontrar enlaces en los que localizar dicha información.

9.5 – Sistema de ficheros del Driver

El código del driver está escrito en C. La estructura de nuestro driver es la siguiente:

- MakeFile:
  - Contiene las reglas de dependencia y las reglas de construcción.
- Sources:
  - Contiene los nombres de los archivos que van a ser compilados.
- ssiidrv.h:
  - Contiene las cabeceras de las funciones que serán implementadas en el archivo ssiidrv.c
- ssiidrv.c:
  - Contiene la implementación de las funciones del driver.
- ssiidrv_public.h:
  - Contiene las cabeceras de las funciones compartidas por el código del driver y el código de la aplicación.
- ssiidrv_msg.mc:
  - Contiene la definición de los mensajes que el driver devolverá como eventos y que pueden ser visualizados desde el visor de eventos de Windows.

9.6 – Funciones relativas al Driver

Las funciones implementadas en el driver son las siguientes:

9.6.1 – DriverEntry

Esta rutina es la primera que se llama después de cargar el controlador desde el Gestor de Entrada/Salida. Esta rutina es la responsable de la inicialización del controlador. Su cabecera es la siguiente:

```
DRIVER_INITIALIZE DriverEntry;

NTSTATUS
  DriverEntry(
    __in struct _DRIVER_OBJECT  *DriverObject,
    __in PUNICODE_STRING  RegistryPath
    )
  {...}
```

Parámetros:

- DriverObject:
  - Puntero a una estructura DRIVER_OBJECT. Es el objeto que representa al controlador.

- RegistryPath:
  - Puntero a una cadena Unicode, especifica la ruta de acceso de la clave del registro del driver.

La estructura DRIVER_OBJECT será completada en el cuerpo de la función DriverEntry asignando las funciones que trataran cada uno de los códigos de operación que admitirá el controlador.

El código que implementa esta función es el siguiente:

```
////////////////// DriverEntry()/////////////////

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

 WriteEvent(MSG_DRIVERENTRY_START, DriverObject);

 DriverObject->DriverExtension->AddDevice = AddDevice;
 DriverObject->MajorFunction[IRP_MJ_CREATE] = CreateSSII;
 DriverObject->MajorFunction[IRP_MJ_CLOSE] = CloseSSII;
 DriverObject->MajorFunction[IRP_MJ_SHUTDOWN] = ShutdownSSII;
 DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]          =
ManejarIOControl;
 DriverObject->MajorFunction[IRP_MJ_PNP] = HandlePnP;
 DriverObject->MajorFunction[IRP_MJ_SYSTEM_CONTROL]          =
HandleSystemControl;
 DriverObject->DriverUnload = SSIIUnload;

 WriteEvent(MSG_DRIVERENTRY_END, DriverObject);
 return STATUS_SUCCESS;
}
```

9.6.2 – AddDevice

Esta rutina es invocada por Windows por cada dispositivo detectado por el sistema. La principal responsabilidad de esta rutina es llamar a la rutina IoCreateDevice para crear varios objetos de dispositivos que representan los dispositivos físicos, lógicos o virtuales. Las aplicaciones pueden usar el dispositivo simbólico creado por IoCreateDevice para tener acceso al controlador.

La cabecera de esta función es la siguiente:

```
DRIVER_ADD_DEVICE AddDevice;

NTSTATUS AddDevice(
    __in struct _DRIVER_OBJECT  *DriverObject,
    __in struct _DEVICE_OBJECT  *PhysicalDeviceObject
    )
  {...}
```

Parámetros:

- DriverObject DriverObject:
  - Puntero a una estructura DRIVER_OBJECT. Es el objeto que representa al controlador.

- PhysicalDeviceObject :
  - Representa el objeto físico del controlador creado por un controlador de nivel inferior.

El código que implementa esta función es el siguiente:

```
//////////////////// AddDevice() ////////////////////
NTSTATUS AddDevice(IN PDRIVER_OBJECT DriverObject, IN
PDEVICE_OBJECT pdo)
{
 UNICODE_STRING deviceName, deviceLink;
 NTSTATUS status;
 PDEVICE_OBJECT deviceObj;
 TSSIIDeviceExtension * devExt;

 WriteEvent(MSG_ADDDEVICE_START, DriverObject);

 // Crear dispositivo simbólico para que las aplicaciones
puedan abrirlo.
 RtlInitUnicodeString(&deviceName, DEVICE_NAME);
 status = IoCreateDevice(DriverObject,
sizeof(TSSIIDeviceExtension), &deviceName,
                  FILE_DEVICE_UNKNOWN, 0, TRUE,
                  &deviceObj);

 if (NT_SUCCESS(status))
 {
       DriverObject->DeviceObject = deviceObj;
       devExt = deviceObj->DeviceExtension;
```

```
        RtlZeroMemory(devExt, sizeof(TSSIIDeviceExtension));
        devExt->physDevObj = pdo;
        //DMA
        devExt->nMapRegs = 0;
        devExt->dmaLength = 4*1024;

        RtlInitUnicodeString(&deviceLink, DEVICE_LINK_NAME);
        status = IoCreateSymbolicLink(&deviceLink,
&deviceName);

        if (NT_SUCCESS(status))
        {
              devExt->NextLowerDevice =
IoAttachDeviceToDeviceStack(deviceObj, pdo);
              status = (devExt->NextLowerDevice == NULL ?
STATUS_NO_SUCH_DEVICE : STATUS_SUCCESS);
        }

        if (!NT_SUCCESS(status)) {
              IoDeleteDevice(deviceObj);
              WriteEvent(MSG_ADDEVICE_NO_SUCH_DEVICE,
DriverObject);
        }
 }

 if (NT_SUCCESS(status))
 {
        deviceObj->Flags |= DO_DIRECT_IO;
        WriteEvent(MSG_ADDDEVICE_END, DriverObject);
        deviceObj->Flags &= ~DO_DEVICE_INITIALIZING;
 }
 else
 {
        WriteEvent(MSG_ADDDEVICE_END_FAILURE, DriverObject);

 }

 return status;
}
```

9.6.3 – SSIIUnload

Esta función es invocada cuando el dispositivo del driver es desconectado. Esta rutina elimina los recursos específicos del controlador como la memoria, hilos y eventos que son creados en la rutina DriverEntry y AddDevice

El código que implementa esta función es el siguiente:

```
//////////////////////SSIIUnload//////////////////////

void SSIIUnload(IN PDRIVER_OBJECT DriverObject)
{
 UNICODE_STRING deviceLink;
 TSSIIDeviceExtension * devExt;
```

```
 PAGED_CODE();

 RtlInitUnicodeString(&deviceLink, DEVICE_LINK_NAME);
 IoDeleteSymbolicLink(&deviceLink);
 if (DriverObject->DeviceObject != NULL)
 {
       devExt = DriverObject->DeviceObject->DeviceExtension;

       // Unchain the driver from the stack.
       if (devExt->NextLowerDevice != NULL)
             IoDetachDevice(devExt->NextLowerDevice);
       IoDeleteDevice(DriverObject->DeviceObject);
 }
}
```

9.6.4 – CreateSSII

Esta función es invocada cuando un programa de usuario crea con la llamada CreateFile() haciendo referencia al dispositivo.

Un ejemplo de un posible código que realice esta llamada es el siguiente:

```
// Abrir el controlador de dispositivo PCI.
 driverSSII = CreateFile(PUBLIC_DEVICE_NAME, GENERIC_READ |
GENERIC_WRITE, 0,
       NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
```

No se van a comentar cada uno de los argumentos de la llamada, solo comentar que el primer argumento PUBLIC_DEVICE_NAME se declarara en un archivo de cabeceras que sea accesible tanto desde el código de la aplicación como desde el código del driver para su compilación, en este caso se encuentra en el archivo *ssiidrv_public.h* que se comenta en capítulos anteriores. Esta declaración indicará la dirección del controlador al que se desea acceder, en este caso:

```
//  Nombre con el que abrir un manejador al dispositivo.
#definePUBLIC_DEVICE_NAME      "\\\\.\\SSIIDRV"
```

El código que implementa la función CreateSSII es el siguiente:

```
////////////////////// CreateSSII() ////////////////////

NTSTATUS CreateSSII(IN PDEVICE_OBJECT DeviceObject, IN PIRP
Irp)
{
 PAGED_CODE();

 IoCompleteRequest(Irp, IO_NO_INCREMENT);
 Irp->IoStatus.Information = 0;
 return STATUS_SUCCESS;
}
```

9.6.5 – CloseSSII

Función invocada por Windows cuando un programa de usuario cierra el manejador (handle) del controlador invocando la función Close(). En esta función también se desmapean los recursos asignados para DMA, esto evita que los recursos asignados en cada ejecución se queden ocupados.

La llamada desde la aplicación quedaría:

```
CloseHandle(driverSSII);
```

Siendo *dirverSSII* un identificador abierto a un dispositivo.

El código que implementa la función ClosedSSII es el siguiente:

```
////////////////// CloseSSII() //////////////////
NTSTATUS CloseSSII(IN PDEVICE_OBJECT DeviceObject, IN PIRP
Irp)
{
 TSSIIDeviceExtension * devExt;

 PAGED_CODE();
 devExt = DeviceObject->DeviceExtension;
 //Liberar DMA
 WriteEvent(MSG_CLOSE_ENTRY, DeviceObject->DriverObject);
 if(devExt->adapterObj !=NULL)
       (*devExt->adapterObj->DmaOperations->PutDmaAdapter)
             (devExt->adapterObj);
 devExt->adapterObj=NULL;
 WriteEvent(MSG_CLOSE_EXIT, DeviceObject->DriverObject);
 IoCompleteRequest(Irp, IO_NO_INCREMENT);
 Irp->IoStatus.Information = 0;


 return STATUS_SUCCESS;
}
```

9.6.6 – ShutdownSSII

Al invocar esta función desde Windows desconecta el dispositivo.

El código que implementa esta función es el siguiente:

```
////////////////// ShutdownSSII() //////////////////

NTSTATUS ShutdownSSII(IN PDEVICE_OBJECT DeviceObject, IN PIRP
Irp)
{
 PAGED_CODE();
```

```
 IoCompleteRequest(Irp, IO_NO_INCREMENT);

 return STATUS_SUCCESS;
}
```

9.6.7 – WriteEvent

Esta función envía un evento un mensaje en función del código de error que recibe. Los mensajes de error del driver están definidos en el archivo *ssiidrv_msg.mc*

El código que implementa esta función es el siguiente:

```
////////////////////// WriteEvent() ////////////////////

BOOLEAN WriteEvent(IN NTSTATUS ErrorCode, IN PVOID IoObject)
{
 PIO_ERROR_LOG_PACKET packet;
 UCHAR packetSize = sizeof(IO_ERROR_LOG_PACKET);
 BOOLEAN res = FALSE;

 PAGED_CODE();

 packet = IoAllocateErrorLogEntry(IoObject, packetSize);
 if (packet != NULL)
 {
       RtlZeroMemory(packet, sizeof(PIO_ERROR_LOG_PACKET));
       packet->ErrorCode = ErrorCode;
       IoWriteErrorLogEntry(packet);
       res = TRUE;
 }

 return res;
}
```

9.6.8 – HandlePnP

Gestiona los eventos Plug and Play relacionados con el dispositivo. Concretamente esta función busca el espacio de memoria asignado al dispositivo (BAR0 y BAR1) de la lista de recursos y los almacena en una estructura para poder utilizarlos cuando sea necesario.

Por otra parte, esta función también hace una petición de un espacio para DMA para que la aplicación y el controlador puedan transmitirse datos el uno al otro. La dirección del espacio concedido para DMA se envía al controlador (esta dirección se guarda en un registro del BAR0) para que sepa donde debe escribir o leer los datos para intercambio con la aplicación, dicha dirección es almacenada en una estructura para poder enviársela a la aplicación de usuario cuando esta la solicite con una petición usando la función DeviceIoControl.

El código que implementa la función HandlePnP es el siguiente:

```
 NTSTATUS HandlePnP(IN PDEVICE_OBJECT DeviceObject, IN PIRP
Irp)
{
```

```
 NTSTATUS status = STATUS_SUCCESS;
 PIO_STACK_LOCATION irpStack;
 PCM_PARTIAL_RESOURCE_LIST raw, translated;
 CM_PARTIAL_RESOURCE_DESCRIPTOR * resource;
 TSSIIDeviceExtension * devExt;
 DEVICE_DESCRIPTION dd;
 PHYSICAL_ADDRESS tmpAddr;
 unsigned int ii;

 WriteEvent(MSG_HANDLEPNP_START, DeviceObject->DriverObject);
 devExt = DeviceObject->DeviceExtension;
 irpStack = IoGetCurrentIrpStackLocation(Irp);
 switch (irpStack->MinorFunction)
 {
       ////// Sub-función para añadir dispositivos al
sistema.
       case IRP_MN_START_DEVICE:
             // Obtener lista de recursos en modo raw (bus) y
traducidos.
             raw = &irpStack-
>Parameters.StartDevice.AllocatedResources-
>List[0].PartialResourceList;
             translated = &irpStack->Parameters.StartDevice.
                         AllocatedResourcesTranslated-
>List[0].PartialResourceList;

             // Hay que buscar los recursos de memoria BAR0 y
BAR1. Identificados por su tamaño.
             for (ii = 0; ii < translated->Count; ++ ii)
             {
                   resource = &(translated-
>PartialDescriptors[ii]);
                   if (resource->Type ==
CmResourceTypeMemory)
                   {
                         if (resource->u.Memory.Length ==
4096)
                         {
                               devExt->BAR0Addr =
(PVOID)resource->u.Memory.Start.LowPart;
                               devExt->BAR0Length = resource-
>u.Memory.Length;
                         }
                         else
                         {
                               devExt->ssiiPCIAddr =
(PVOID)resource->u.Memory.Start.LowPart;
                               devExt->ssiiPCILength =
resource->u.Memory.Length;
                         }
                   }
             }

             // Mapear los recursos de memoria BAR0 y BAR1.
             if ( (devExt->BAR0Addr == NULL) || (devExt-
>ssiiPCIAddr == NULL) )
                   status = STATUS_UNSUCCESSFUL;

             if (NT_SUCCESS(status))
             {
                   tmpAddr.LowPart = (ULONG)devExt->BAR0Addr;
```

```
                  tmpAddr.HighPart = 0;
                  devExt->BAR0KernelAddr =
                        MmMapIoSpace(tmpAddr, devExt-
>BAR0Length, MmNonCached);
                  if (devExt->BAR0KernelAddr == NULL)
                        status = STATUS_UNSUCCESSFUL;
            }

            if (NT_SUCCESS(status))
            {
                  tmpAddr.LowPart = (ULONG)devExt-
>ssiiPCIAddr;
                  tmpAddr.HighPart = 0;
                  devExt->ssiiPCIKernelAddr =
                        MmMapIoSpace(tmpAddr, devExt-
>ssiiPCILength, MmNonCached);
                  if (devExt->ssiiPCIKernelAddr == NULL)
                        status = STATUS_UNSUCCESSFUL;
            }

            //Solicitud de DMA
            RtlZeroMemory(&dd, sizeof(dd));
            dd.Version = DEVICE_DESCRIPTION_VERSION;
            dd.Master = TRUE;
            dd.DmaWidth = Width32Bits;
            dd.MaximumLength = devExt->dmaLength;
            dd.InterfaceType = PCIBus;
            dd.Dma32BitAddresses = TRUE;

            devExt->adapterObj = IoGetDmaAdapter(devExt-
>physDevObj, &dd, &devExt->nMapRegs);

            //Asignación de buffer para DMA
            if( devExt->adapterObj != NULL ) {
                  devExt->dmaKernelAddr = (*devExt-
>adapterObj->DmaOperations->AllocateCommonBuffer)(devExt-
>adapterObj, devExt->dmaLength, &devExt->dmaDeviceAddr,
TRUE);
                  if (devExt->dmaDeviceAddr.LowPart != NULL)
{
                        *((unsigned int *)((devExt-
>ssiiPCIKernelAddr + 0x0000000C))) = devExt-
>dmaDeviceAddr.LowPart;
                        *((unsigned int *)((devExt-
>BAR0KernelAddr + 0x00000188))) = devExt-
>dmaDeviceAddr.LowPart;
                  } else {
                        WriteEvent(MSG_COMMON_BUFFER_ERROR,
DeviceObject->DriverObject);
                  }
            } else {
                  //Meter mensaje de error
                  WriteEvent(MSG_DMA_ERROR, DeviceObject-
>DriverObject);
                  status = STATUS_UNSUCCESSFUL;
            }

            //  Si se produce algún error, liberar todos los
recursos obtenidos
            // hasta el momento.
            if (!NT_SUCCESS(status))
```

```
            {
                  if (devExt->BAR0KernelAddr != NULL)
                  {
                        MmUnmapIoSpace(devExt-
>BAR0KernelAddr, devExt->BAR0Length);
                        devExt->BAR0KernelAddr = NULL;
                  }
                  if (devExt->ssiiPCIKernelAddr != NULL)
                  {
                        MmUnmapIoSpace(devExt-
>ssiiPCIKernelAddr, devExt->ssiiPCILength);
                        devExt->ssiiPCIKernelAddr = NULL;
                  }
            }

            IoCompleteRequest(Irp, IO_NO_INCREMENT);
            break;

      ////// Subfunción para desconectar dispositivos del
sistema.
      case IRP_MN_STOP_DEVICE:
            WriteEvent(MSG_IRP_MN_STOP_DEVICE_ENTRY,
DeviceObject->DriverObject);
            if (devExt->BAR0KernelAddr != NULL)
            {
                  MmUnmapIoSpace(devExt->BAR0KernelAddr,
devExt->BAR0Length);
                  devExt->BAR0KernelAddr = NULL;
            }
            if (devExt->ssiiPCIKernelAddr != NULL)
            {
                  MmUnmapIoSpace(devExt->ssiiPCIKernelAddr,
devExt->ssiiPCILength);
                  devExt->ssiiPCIKernelAddr = NULL;
            }
            if (devExt->dmaKernelAddr != NULL)
            {
                  (*devExt->adapterObj->DmaOperations-
>FreeCommonBuffer)(devExt->adapterObj, devExt->dmaLength,
devExt->dmaDeviceAddr, devExt->dmaKernelAddr, TRUE);
            }
            WriteEvent(MSG_IRP_MN_STOP_DEVICE_EXIT,
DeviceObject->DriverObject);
      default:
            IoSkipCurrentIrpStackLocation(Irp);
                  status = IoCallDriver(devExt-
>NextLowerDevice, Irp);
            break;
 }

 WriteEvent(MSG_HANDLEPNP_END, DeviceObject->DriverObject);
 return status;
}
```

9.6.9 – HandleSystemControl

Esta rutina delega los mensajes de control del sistema al siguiente en la pila de controladores.

El código que implementa a la función HandleSystemControl es el siguiente:

```
////////////////// HandleSystemControl() /////////////////

NTSTATUS HandleSystemControl(IN PDEVICE_OBJECT DeviceObject,
IN PIRP Irp)
{
 NTSTATUS status;

 IoSkipCurrentIrpStackLocation(Irp);
 status = IoCallDriver(((TSSIIDeviceExtension*)DeviceObject-
>DeviceExtension)->NextLowerDevice, Irp);

 return status;
}
```

9.6.10 – ManejarIOControl

Esta función es de total relevancia, debido a que es el punto donde se actúa como interfaz con las aplicaciones de usuario.

Recibe un código de operación de una llamada a la función DeviceIoControl desde modo usuario y en función del código de operación selecciona la acción a realizar y llama a la función adecuada para realizar dicha acción.

Los códigos de operación estarán declarados en un archivo de cabeceras que sea accesible tanto desde el código de la aplicación como desde el código del driver para su compilación, en este caso se encuentra en el archivo *ssiidrv_public.h*. En el siguiente código se observa cómo definir un código de operación, en dicho código también se puede observar que se ha definido una estructura debajo de dicha definición, esta estructura se utilizara para pasar información entre la aplicación y el controlador del dispositivo o viceversa.

```
#define SSIIDRV_IO_GET_PCI_RESOURCES \
 CTL_CODE(FILE_DEVICE_UNKNOWN, 2048, METHOD_OUT_DIRECT,
FILE_ANY_ACCESS)
typedef struct
{
 PVOID BAR0Addr, ssiiPCIAddr;
 ULONG BAR0Length, ssiiPCILength;
} TSSIIDRV_IO_GET_PCI_RESOURCES;
```

El resto de operaciones y estructuras definidas en el archivo *ssiidrv_public.h* se puede observar a continuación:

```
 //  Función para acceder al área de mamoria (BAR1) del
dispositivo.
#defineSSIIDRV_IO_ACCESS_DEVMEM \
 CTL_CODE(FILE_DEVICE_UNKNOWN, 2049, METHOD_OUT_DIRECT,
FILE_ANY_ACCESS)
typedef struct
{
 unsigned long offset;
```

```
 unsigned long value;
 BOOLEAN write;
} TSSIIDRV_IO_ACCESS_DEVMEM;
//  Función para acceder al DMA
#defineSSIIDRV_IO_ACCESS_DMA \
 CTL_CODE(FILE_DEVICE_UNKNOWN, 2050, METHOD_OUT_DIRECT,
FILE_ANY_ACCESS)
typedef struct
{
 PVOID mappedBar1;
 PVOID mappedDMAAddr;
} TSSIIDRV_IO_ACCESS_DMA;
```

El código completo que implementa la función ManejarIOControl es el siguiente:

```
/////////////// ManejarIOControl() ///////////////
NTSTATUS ManejarIOControl(IN PDEVICE_OBJECT deviceObject, IN
PIRP irp)
{
 NTSTATUS status = STATUS_SUCCESS;
 PIO_STACK_LOCATION irpStack;

 PAGED_CODE();

 irpStack = IoGetCurrentIrpStackLocation(irp);
 switch (irpStack->Parameters.DeviceIoControl.IoControlCode)
 {
 case SSIIDRV_IO_GET_PCI_RESOURCES:
       status = SSIIDrvIOGetPCIResources(deviceObject, irp,
irpStack);
       break;

 case SSIIDRV_IO_ACCESS_DEVMEM:
       status = SSIIDrvIOAccessDevMem(deviceObject, irp,
irpStack);
       break;

 case SSIIDRV_IO_ACCESS_DMA:
       status = SSIIDrvIOAccessDMA(deviceObject, irp,
irpStack);
       break;

 default:
       status = STATUS_INVALID_DEVICE_REQUEST;
       break;
 }

 // Completar la operación sin incrementar la prioridad de la
hebra del cliente.
 IoCompleteRequest(irp, IO_NO_INCREMENT);

 return status;
}
```

Como se puede observar en el código de esta función hay dos posibles funciones que se deseen ejecutar desde el modo usuario, estas funciones se verán a continuación

9.6.11 – SSIIDrvIOGetPCIResources

Función para enumerar los recursos asignados al dispositivo PCI.

El código que implementa esta función es el siguiente:

```
/////////////// SSIIDrvIOGetPCIResources() //////////////

static NTSTATUS SSIIDrvIOGetPCIResources(IN PDEVICE_OBJECT
deviceObject, IN PIRP irp,
                        IN PIO_STACK_LOCATION irpStack)
{
 NTSTATUS status = STATUS_SUCCESS;
 ULONG outputLength;
 PVOID output;
 TSSIIDeviceExtension * devExt;

 devExt = deviceObject->DeviceExtension;
 outputLength = irpStack-
>Parameters.DeviceIoControl.OutputBufferLength;

 if (outputLength < sizeof(TSSIIDRV_IO_GET_PCI_RESOURCES))
       status = STATUS_INVALID_BUFFER_SIZE;
 else
 {
       output = MmGetSystemAddressForMdlSafe(irp->MdlAddress,
NormalPagePriority);
       if (output == NULL)
       {
             status = STATUS_INSUFFICIENT_RESOURCES;
       }
       else
       {
             ((TSSIIDRV_IO_GET_PCI_RESOURCES*)output)-
>BAR0Addr = devExt->BAR0Addr;
             ((TSSIIDRV_IO_GET_PCI_RESOURCES*)output)-
>BAR0Length = devExt->BAR0Length;
             ((TSSIIDRV_IO_GET_PCI_RESOURCES*)output)-
>ssiiPCIAddr = devExt->ssiiPCIAddr;
             ((TSSIIDRV_IO_GET_PCI_RESOURCES*)output)-
>ssiiPCILength = devExt->ssiiPCILength;
             irp->IoStatus.Information =
sizeof(TSSIIDRV_IO_GET_PCI_RESOURCES);
       }
 }

 return status;
}
```

9.6.12 – SSIIDrvIOAccessDevMem

Esta función accede a la zona de memoria del dispositivo.

El código que implementa esta función es el siguiente:

```
/////////////// SSIIDrvIOAccessDevMem() ///////////////

static NTSTATUS SSIIDrvIOAccessDevMem(IN PDEVICE_OBJECT
deviceObject, IN PIRP irp,
                        IN PIO_STACK_LOCATION irpStack)
{
 NTSTATUS status = STATUS_SUCCESS;
 ULONG outputLength;//, inputLength;
 TSSIIDeviceExtension * devExt;
 TSSIIDRV_IO_ACCESS_DEVMEM * userBuffer;
 volatile PULONG ssiiAddr;

 devExt = deviceObject->DeviceExtension;
 outputLength = irpStack-
>Parameters.DeviceIoControl.OutputBufferLength;

 if (outputLength < sizeof(TSSIIDRV_IO_ACCESS_DEVMEM))
       status = STATUS_INVALID_BUFFER_SIZE;
 else
 {
       userBuffer = (TSSIIDRV_IO_ACCESS_DEVMEM *)
                  MmGetSystemAddressForMdlSafe(irp-
>MdlAddress, NormalPagePriority);
       if (userBuffer == NULL)
             status = STATUS_INSUFFICIENT_RESOURCES;
       else if (userBuffer->offset >= devExt->ssiiPCILength)
             status = STATUS_UNSUCCESSFUL;
       else
       {
             userBuffer->offset &= 0xFFFFFFFC;
             ssiiAddr = (unsigned long *)(devExt-
>ssiiPCIKernelAddr + userBuffer->offset);
             if (userBuffer->write)
                   *ssiiAddr = userBuffer->value;
             else
                   userBuffer->value = *ssiiAddr;
             irp->IoStatus.Information =
sizeof(TSSIIDRV_IO_ACCESS_DEVMEM);
       }
 }

 return status;
}
```

9.6.13 – SSIIDrvIOAccessDMA

Esta función devuelve las direcciones del espacio mapeado para DMA a la aplicación de usuario para que la aplicación pueda leer o escribir en dicho espacio.

```
/////////////// SSIIDrvIOAccessDMA () ///////////////

static NTSTATUS SSIIDrvIOAccessDMA(IN PDEVICE_OBJECT
deviceObject, IN PIRP irp,
                        IN PIO_STACK_LOCATION irpStack)
```

```
{
 NTSTATUS status = STATUS_SUCCESS;
 ULONG outputLength;//, inputLength;
 TSSIIDRV_IO_ACCESS_DMA * userBuffer;
 TSSIIDeviceExtension * devExt;
 PMDL tmpMDL;
 volatile PULONG ssiiAddr;

 devExt = deviceObject->DeviceExtension;
 outputLength = irpStack-
>Parameters.DeviceIoControl.OutputBufferLength;

 if (outputLength < sizeof(TSSIIDRV_IO_ACCESS_DMA))
       status = STATUS_INVALID_BUFFER_SIZE;
 else
 {
       userBuffer = (TSSIIDRV_IO_ACCESS_DMA *)
                    MmGetSystemAddressForMdlSafe(irp-
>MdlAddress, NormalPagePriority);

       devExt->dmaMdlCommonBuffer = IoAllocateMdl(devExt-
>dmaKernelAddr, devExt->dmaLength, FALSE, FALSE, NULL);
       MmBuildMdlForNonPagedPool(devExt->dmaMdlCommonBuffer);
       userBuffer->mappedDMAAddr =
MmMapLockedPagesSpecifyCache(devExt->dmaMdlCommonBuffer,
UserMode, MmCached, NULL, FALSE, NormalPagePriority);

       devExt->dmaMdlCommonBuffer = IoAllocateMdl(devExt-
>ssiiPCIKernelAddr, devExt->ssiiPCILength, FALSE, FALSE,
NULL);
       MmBuildMdlForNonPagedPool(devExt->dmaMdlCommonBuffer);
       userBuffer->mappedBar1 =
MmMapLockedPagesSpecifyCache(devExt->dmaMdlCommonBuffer,
UserMode, MmCached, NULL, FALSE, NormalPagePriority);
 }

 return status;
}
```

9.7 – Archivo INF

En este apartado se detallan las secciones principales que pueden ser usadas en el desarrollo de un archivo INF. Cada sección describirá los aspectos que se consideren importantes para describir el controlador en el sistema y definirá valores en el sistema operativo que permitirán clasificar el controlador y hacerlo accesible por las aplicaciones que lo precisen.

La siguiente figura muestra la estructura completa de un archivo INF:

En primer lugar y para la implementación de cualquier archivo INF hay que tener en cuenta ciertos aspectos; las cabeceras de definición de secciones se encuadran entre corchetes y no son sensibles a mayúsculas por los que serán aceptadas igualmente entradas como version, Version o VERSION. Por otro lado el orden en el que se especifiquen estas secciones no es relevante ya que el programa de instalación busca las secciones por el nombre de forma no secuencial.

Se tratará de detallar primero las secciones requeridas y prioritarias para terminar comentando la utilidad de otras menos importantes.

9.7.1 – Sección Strings

Aunque esta sección suele situarse por convenio al final del archivo comenzaremos explicándola en primer lugar dado que se hará referencia a su contenido en muchas de las secciones.

Cada entrada de esta sección es de la forma:

```
Nombre_etiqueta = “cadena”
```

Cada *Nombre_cadena* especifica una clave única que identificara la cadena. Se puede hacer referencia a las cadenas sustituyéndolas en cualquier parte del archivo por la clave que las identifica encerrada entre símbolos de porcentaje '%' (%nombre_etiqueta%). Las comillas realmente solo son necesarias si la cadena empieza o finaliza con espacios en blanco ya que el parseador del programa instalador podría eliminarlos al tratar el archivo INF, no obstante, se recomienda su uso en cualquier caso.

9.7.2 – Sección VERSIÓN

Esta sección es requerida en cualquier archivo INF y aunque generalmente se escribe al principio como se ha comentado anteriormente no es importante.

```
Signature="valor"
```

Esta entrada es obligatoria e identifica el sistema operativo para el cual está escrito el driver. Los valores válidos son:

- $Windows NT$
  - Especifica sistemas operativos basados en NT.
- $Windows 95$
  - Sistemas 9x/Me
- $Chicago$
  - Válida para todos los sistemas operativos Windows.

Los símbolos '$' son obligatorios y las cadenas deben de estar escritas entre comillas dobles y aunque éstas no son sensibles a las mayúsculas si la cadena no es una de las descritas anteriormente el archivo INF no será aceptado.

Aunque esta entrada es requerida es curioso que el programa de instalación no le presta mucha atención y realmente no importa para la instalación cuál de los valores descritos arriba esté presente ya que aunque cada sistema escogerá un driver con la etiqueta adecuada para él se conformará con cualquier otro si es la única opción. Es posible que tenga un carácter informativo para especificar para qué sistema operativo se ha desarrollado el sistema que se pretende instalar.

```
Class=nombre_de_la_clase
```

Esta entrada especifica el nombre de clase predefinido para todos los dispositivos estándares. Si se define esta entrada se debe especificar también la entrada ClassGUID. Incluir estas entradas es útil para el programa de instalación ya que permitirá optimizar la búsqueda en el archivo INF y agilizará la instalación.

Se puede especificar una nueva clase y en ese caso se debe de escribir un nombre único de máximo 32 caracteres que no sea igual a los ya predefinidos e igualmente generar un nuevo valor para la entrada ClassGUID. En caso de definir una nueva clase será necesaria una sección ClassInstall32 que no será especificada en la memoria. Para obtener un identificador GUID es

necesaria una aplicación se generación de claves. En nuestro caso hemos obtenido la nuestra en internet, hay múltiples webs que ofrecen este servicio de forma gratuita.

```
ClassGUID={nnnnnnnn-nnnn-nnnn-nnnn-nnnnnnnnnnnn}
```

- Especifica el valor asociado a la clase en dígitos hexadecimales.

```
Provider=Proveedor
```

- Identifica al proveedor o creador del controlador.
- Generalmente es de la forma %Etiqueta% ya que se implementa mediante una etiqueta definida en la sección Strings que se detallará más adelante.
- Estas etiquetas serán usadas a lo largo de la descripción.

```
CatalogFile=filename.cat
```

- Especifica un archivo de catalogo aportado en las distribuciones que han superado el test del Laboratorio de Calidad del Hardware Windows (WHQL) que contiene firmas digitales e información de validación de la instalación.

```
DriverVer=mm/dd/yyyy[,x.y.v.z]
```

- Esta entrada informa de la versión del controlador a instalar por el archivo INF.

- mm/dd/yyyy:
  - Este valor especifica la fecha del paquete de instalación y debe de ser la más reciente de todos los ficheros presentes en el paquete de instalación.

- [,x.y.v.z]:
  - Este valor opcional especifica la versión del controlador. Si se establece, el valor x es obligatorio mientras que el resto son opcionales. Se debe establecer para evitar problemas ya que el sistema operativo la utiliza para determinar el controlador más reciente, a la hora de realizar actualizaciones, combinada con la fecha.

```
DriverPackageDisplayName="descripción_del_controlador"
```

- A modo descriptivo para los usuarios del propósito del controlador.

9.7.3 – Sección MANUFACTER

La sección *Manufacturer* identifica al fabricante de uno o más de los dispositivos que serán instalados por el archivo INF:

```
Nombre_fabricante | %clave_nombre% = sección_de_modelos
[,extensión_SO]…
```

Cada entrada identifica un fabricante al que se le asocia una sección de definición de modelos que debe de existir en el archivo. A continuación se le pueden agregar opcionalmente extensiones para soporte multiplataforma.

9.7.4 – Archivos INF multiplataforma

Es posible crear archivos INF para distintos sistemas operativos y plataformas y definir secciones específicas para cada uno de ellos agregando extensiones a las cabeceras de las mismas. Estas extensiones deben ser especificadas en la sección *Manufacturer*:

```
[Manufacturer]
; The manufacturer-identifier for the Abc device.
%ManufacturerName%=AbcModelSection,ntia64,ntamd64
```

La siguiente tabla resume las extensiones soportadas para las distintas plataformas:

Extension	Uso
.ntamd64	La sección contiene instrucciones para instalar un dispositivo o establecer un dispositivo compatible con sistemas x64 soportados por Windows XP o superior.
.ntia64	La sección contiene instrucciones para instalar un dispositivo o establecer un dispositivo compatible con sistemas Itanium soportados por Windows XP o superior.
.ntx86	La sección contiene instrucciones para instalar un dispositivo o establecer un dispositivo compatible con sistemas x86 soportados por Windows XP o superior.
.nt	La sección contiene instrucciones para instalar un dispositivo o establecer un dispositivo compatible con todos los sistemas soportados por Windows 2000 o superior.
(no platform extension)	La sección contiene instrucciones para instalar un dispositivo o establecer un dispositivo compatible con todos los sistemas soportados por Windows 95 o superior.

En el momento de la instalación el software buscará primero las secciones definidas para esa plataforma y en última instancia buscará en las secciones sin extensión en caso de no haberse especificado. En nuestro caso y dado que no necesitamos soporte multiplataforma no hemos especificado ninguna extensión salvo en casos muy concretos.

Las extensiones pueden contener además parámetros de definición de arquitectura, versión, etc., pero dado su rareza ya que definen secciones puntuales para una determinada versión de SO no profundizaremos en ellas.

9.7.5 – Sección de modelos

En esta sección se identifican los dispositivos para los cuales se instalan los drivers definidos en el archivo INF y asociados al fabricante que referencia la sección de fabricante.

```
descripción_de_dispositivo|%clave_descripción%=
 nombre_sección_DDInstall , hw-id[, compatible-id]…
```

Cada una de las entradas de esta sección mapea una cadena de descripción del dispositivo con el ID de dispositivo y el nombre de la sección *DDInstall* que contiene las instrucciones de instalación para el dispositivo, es decir, cada una de las entradas identifica un dispositivo.

Cada sección de modelos debe de estar definida en la sección correspondiente de su fabricante. Es necesaria una sección de modelos por cada sección de fabricante definida. En cada una de éstas se especifican los detalles para la instalación del driver de cada dispositivo de ese fabricante.

El parámetro *hw-id* es muy importante ya que supone el hardware ID, una cadena de identificación del dispositivo del fabricante y, que el manejador PnP usará para encontrar y asociar el dispositivo a instalar con el archivo INF. El identificador ha de seguir el siguiente formato:

```
enumerador\enumerador_específico_de_dispositivo
```

El primer parámetro, *enumerador*, es un identificador genérico del tipo de dispositivo USB, PCI … El segundo parámetro es el identificador propio del fabricante y contiene valores como el valor del fabricante y valores de versión. Así por ejemplo en nuestro dispositivo definimos las siguientes cadenas de identificación:

```
`define HEADER_VENDOR_ID         16'h1994
`define HEADER_DEVICE_ID         16'h0010
`define HEADER_REVISION_ID       8'h01
`define HEADER_SUBSYS_VENDOR_ID 16'h1994
`define HEADER_SUBSYS_ID         16'h0001
```

Y por tanto nuestra cadena de identificación de dispositivo es la siguiente:

```
PCI\VEN_1994&DEV_0010&SUBSYS_00011994&REV_01
```

Por último la cadena compatible-id se utiliza para especificar identificadores de dispositivos compatibles separados por coma y que serán controlados por el mismo driver.

9.7.6 – Sección DDInstall

La sección *DDInstall* mencionada en la sección de modelos contiene directivas que referencian secciones adicionales en el archivo INF. Éstas directivas pueden estar presentes en las secciones referenciadas por la sección DDInstall. Las directivas más comunes son:

```
DriverVer = mm/dd/yyyy[,x.y.v.z]
```

- Esta directiva ofrece información adicional de la versión del driver.

```
CopyFiles = seccion_de_archivos[,seccion_de_archivos]
```

- Esta directiva hace referencia a una o más secciones donde se especifican los archivos que deben ser copiados.

Se pueden declarar secciones DDInstall con el sufijo *.Services*. Éstas declaran directivas AddService que referencian otras secciones o archivos inf con información adicional.

```
AddService=Nombre_del_servicio,[flags],nombre-sección[,
          sección-registro-eventos
[,[TipoEventoLog][,NombreEvento]]]...
```

La sección de registro de eventos incluye las entradas que deben introducirse en el registro del sistema y gracias a las cuales podrán visualizarse lo mensajes generados por el driver en el visor de sucesos del sistema. Como en las demás secciones ésta también puede hacer referencia a otra sección, en la que se describirán éstas entradas mediante la directiva AddReg:

```
AddReg = Nombre_sección_entradas_registro
```

Cada entrada de registro es de la forma:

```
reg-root, [subkey], [value-entry-name], [flags], [value]
```

Donde:

- *Reg-root*:
  - Identifica la raíz del árbol de registro y puede ser uno de los siguientes valores:
    - HKCR – Abreviación para HKEY_CLASSES_ROOT
    - HKCU – Abreviación para HKEY_CURRENT_USER
    - HKLM – Abreviación para HKEY_LOCAL_MACHINE
    - HKU – Abreviación para HKEY_USERS
    - HKR – Indica que las claves que especifiquen esta abreviación serán relativas a la clave asociada a la sección en la que la directiva AddReg aparezca según se indica en la siguiente tabla.

Sección INF que contiene la directiva AddReg	Clave de registro referenciada por HKR
INF *DDInstall* section	The device's software key
INF *DDInstall*.HW section	The device's hardware key
INF *DDInstall*.Services section	The Services key

- *Subkey*
  - Valor opcional que identifica una subclave del registro.

- *value-entry-name*
  - Este valor opcional o bien referencia un valor existente dado ya en una subclave o bien un nuevo valor para añadir a la subclave dada.

- *Flags*
  - Valor hexadecimal expresado como máscara O-Logica que define valores preestablecidos del sistema.

- *Value*. Valor que se desea asignar a dicha clave del registro.

9.7.7 – Sección SourceDiskNames

Identifica las unidades, disquetes, CD-ROM y o rutas donde se encuentran los archivos requeridos por el driver. Cada entrada asigna un identificador a cada dispositivo:

```
Identificador=descripción,,,ruta_de_acceso
```

9.7.8 – Sección SourceDiskFiles

Identifica los archivos usados durante la instalación. Se utiliza el identificador dado en la sección *SourceDiskNames* para indicar el dispositivo en el que se encuentra:

```
Id Identificador=nombre_de_archivo
```

9.7.9 – Sección ClassInstal32

Esta sección es utilizada para declarar, generalmente, una nueva clase de dispositivos. Ésta nueva clase aparecerá en al Administrador de Dispositivos del sistema. En el caso de desarrollar un driver para un dispositivo de una clase existente bastará con especificar ésta en la sección Version. Éste aspecto no es relevante para el funcionamiento del dispositivo pero si es importante para clasificar e identificar el mismo. Una vez más en esta sección se suele referenciar a otra mediante la directiva AddReg en la cual estarán las entradas del registro

para la nueva clase y que deberán de especificar entre otros aspectos el nombre de la clase, el tipo de clase y el icono que se mostrará en el árbol de dispositivos.

10 – Aplicación de usuario

La aplicación de usuario realizada para la conclusión de este proyecto hace uso de la llamada a la función del API DeviceIoControl, que se ha comentado a lo largo de esta documentación, para poder acceder a la funcionalidad del controlador. Concretamente esta aplicación hace uso de la función SSIIDrvIOAccessDMA para obtener la dirección de los recursos asignados para DMA y de los registros del BAR1 del controlador. A través de dichas direcciones la aplicación puede comunicarse con el controlador para realizar lecturas y/o escrituras de datos utilizando DMA.

En esta aplicación se han medido datos interesantes para comprobar el correcto funcionamiento del sistema. Para que el cálculo de los datos sea lo más fiable posible se han realizado múltiples medidas y con todas ellas se ha obtenido la media. Estos datos se expondrán en los siguientes apartados, y el rendimiento obtenido se comparara con dispositivos que actualmente se encuentran a la venta.

11 – Datos obtenidos y comparativas

En esta sección se compararán los datos obtenidos del dispositivo que hemos implementado con dispositivos de almacenamiento actuales.

11.1 – Datos de lectura del dispositivo implementado

La lectura de los datos contenidos en la memoria del dispositivo usando DMA desde la aplicación deja los siguientes resultados:

- Frecuencia :
  - 1652,71 MHz
- Número de ciclos necesarios para leer 512 bytes:
  - 75578 ciclos
- Tiempo de acceso a un sector de 512 bytes:
  - 43,611 µs
- Tasa de transferencia sostenida:
  - 11,2 MB/seg

11.2 – Datos de escritura del dispositivo implementado

La escritura de los datos contenidos en la memoria RAM del Pc a la memoria del dispositivo usando DMA deja los siguientes resultados:

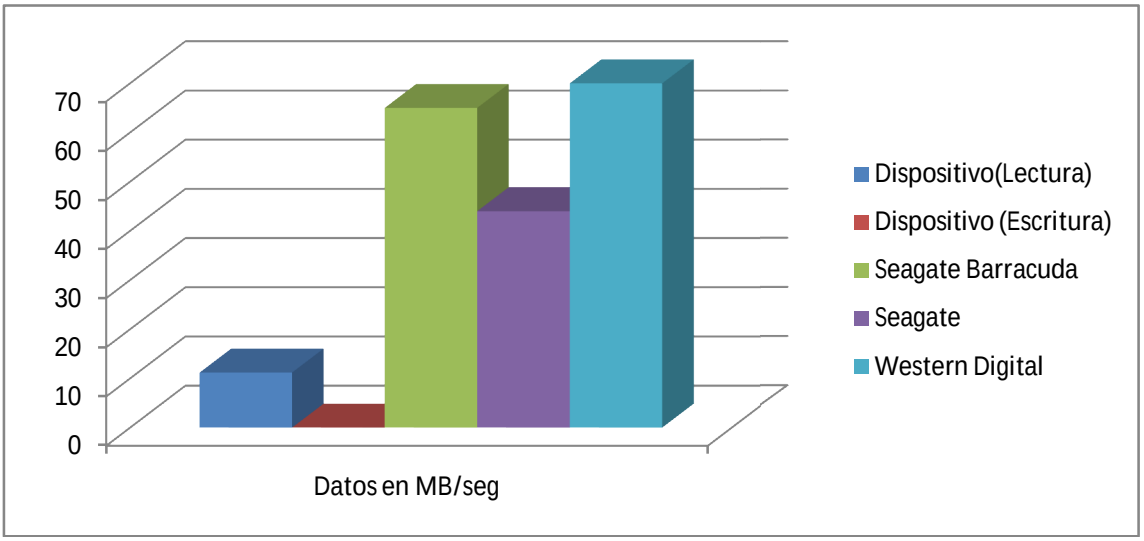
- Frecuencia :
  - 1652,71 MHz
- Número de ciclos necesarios para leer 512 bytes:
  - 433750545 ciclos
- Tiempo de acceso a un sector de 512 bytes:
  - 250288,83 µs
- Tasa de transferencia sostenida:
  - 2 KB/seg

11.3 – Comparativa

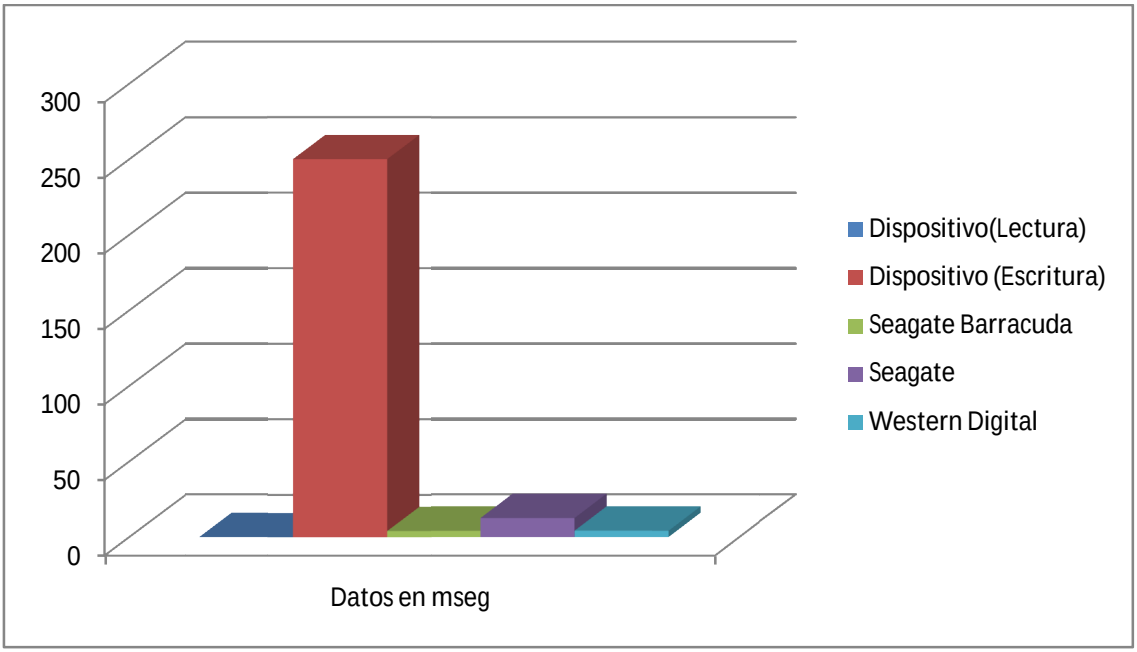
En este apartado se comparan los datos obtenidos por el dispositivo implementado con otros dispositivos del mercado. Los dispositivos de almacenamiento elegidos para la comparación son los siguientes:

- Seagate ST9160821A 2,5¨ 160GB
- Seagate Barracuda 7200.8 400GB
- Western Digital 750 GB Caviar SE16

La siguiente grafica muestra una comparativa de la tasa de transferencia de los distintos dispositivos, nuestro dispositivo se separa en datos de lectura y escritura:



La siguiente grafica muestra una comparativa de la tasa de transferencia de los distintos dispositivos, nuestro dispositivo se separa en datos de lectura y escritura:



11.4 – Conclusiones de la implementación

La primera conclusión clara que se extrae de los apartados anteriores es que existe una gran diferencia entre la lectura y la escritura en el dispositivo que hemos implementado, esta diferencia deriva en que los datos de escritura son erróneos por un fallo en la implementación, dicho fallo no se ha podido subsanar en el plazo de entrega.

Obviando el error en las escrituras, podemos ver en la comparativa anterior que la tasa de transferencia de nuestro dispositivo es inferior al resto, pero el tiempo de acceso es también muy inferior como cabria esperar dada la implementación de nuestro dispositivo. El tiempo de acceso de la implementación se acerca al de los discos duros de estado sólido que se pueden encontrar en el mercado actual, con lo que podemos concluir que nuestra implementación, dentro de sus limitaciones, cumple el objetivo de asemejar su

comportamiento con el de un disco de estado sólido, que era uno de los objetivos del proyecto.

12 – Bibliografía

Enlaces

- http://www.microsoft.com/whdc/devtools/wdk/default.mspx
- http://msdn.microsoft.com

Libros

- *Programming the Microsoft Windows Driver Model*. Walter Oney
- *Connecting a SoC to a PC through the PCI bus*. Miguel Peón
- *PCI DataBook*. PCI Bridge IP Core. Rev. 2.2. OpenCores.org
- *PCI DataSheet*. PCI Bridge IP Core. Rev. 2.2. OpenCores.org
- *PCI IP Core Design document.* Rev. 0.1. OpenCores.org
- *PCI IP Core Specification*. Rev. 1.2. OpenCores.org

13 – Licencia

Nosotros, los autores del proyecto “Sistema de comunicación de sistemas reconfigurables basado en PCI con sistemas de sobremesa”, autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado

Firmado:

Pablo Albaladejo Mestre *Gerardo Martín Moreno* *Pablo Sánchez Escapa*

Madrid, a_____ de _____ de 2009