INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE INGENIERÍA MECÁNICA Y ELÉCTRICA

"SISTEMA DE ALMACENAMIENTO DE DATOS UTILIZANDO MEMORIA SDRAM, UN FPGA Y COMUNICACIÓN USB"

TESIS

QUE PARA OBTENER EL TÍTULO DE INGENIERO EN COMUNICACIONES Y ELECTRÓNICA

PRESENTA

JULIO GONZÁLEZ ARENAS

ASESORES:
ING. DANIEL CRUZ CLEOFAS
M.I. Ma. JOJUTLA OLIMPIA PACHECO ARTEAGA

MÉXICO, D.F. 2008

INSTITUTO POLITECNICO NACIONAL

ESCUELA SUPERIOR DE INGENIERIA MECANICA Y ELECTRICA
UNIDAD PROFESIONAL " ADOLFO LOPEZ MATEOS"

T E M A D E T E S I S

QUE PARA OBTENER EL TITULO DE INGENIERO EN COMUNICACIONES Y ELECTRÓNICA

POR LA OPCION DE TITULACION TESIS Y EXAMEN ORAL INDIVIDUAL

DEBERA(N) DESARROLLAR C. JULIO GONZALEZ ARENAS

"SISTEMA DE ALMACENAMIENTO DE DATOS UTILIZANDO MEMORIA SDRAM, UN FPGA Y COMUNICACIÓN USB"

DESARROLLAR UN SISTEMA DE ALMACENAMIENTO DE DATOS UTILIZANDO UNA MEMORIA SDRAM. UN FPGA PARA IMPLEMENTAR EL CONTROL DE LAS SEÑALES DE ACCESO A LA MEMORIA Y COMUNICACIÓN USB PARA LA TRANSFERENCIA DE DATOS DESDE Y HACIA UNA PC.

- ESPECIFICACION DEL SISTEMA
- EL BUS SERIE UNIVERSAL (USB)
- MEMORIA SDRAM
- IMPLEMENTACION DE LA COMUNICACIÓN USB
- IMPLEMENTACION DEL CONTROLADOR DE SDRAM
- INTEGRACION Y PRUEBAS

MÉXICO D.F., 17 DE ABRIL DE 2008.

A S E S O R E S

ING. DANIEL CRUZ CLEOFAS

M.I. MA. JOJUTLA OLIMPIA PACHECO ARTEAGA.

ING. MARCO ANTONIO CERECEDO DÍAZ
JEFE DEL DEPARTAMENTO ACADÉMICO
DE INGENIERÍA EN COMUNICACIONES Y ELECTRÓNICA

ESCUELA SUPERIOR DE INGENIERIA MECANICA Y ELECTRICA
ESTADOS UNIDOS MEXICANOS
JEFATURA DE I.C.E.

Agradecimientos

A Dios, por el regalo de dejarme abrir los ojos cada mañana y por esforzarme a lo largo de la elaboración de este trabajo y desde el primer día en que decidí confiar en El.

Todo lo puedo en Cristo que me fortalece.

Fil 4:13

A mis padres Julio y Sabina, por todo el apoyo y amor que siempre se siente aún estando lejos de ustedes mientras me daban la oportunidad de estudiar una carrera. También gracias por todas las enseñanzas que no se obtienen en ninguna escuela y que ustedes nos supieron dar. Respeto para ustedes!

Al Instituto de Investigaciones Eléctricas, por todas las facilidades que me brindaron para realizar este trabajo. Gracias a todas las personas dentro de esta institución que colaboraron para concluir ésto exitosamente, principalmente a la Maestra Jojutla Pacheco Arteaga por la paciencia, el apoyo y la asesoría brindados.

Al IPN, a la ESIME y a todas las personas que ponen al alcance de mucha gente la oportunidad de estudiar una carrera profesional en escuelas de educación pública. Gracias a los profesores en la ESIME que aportaron todo su trabajo en mi formación como ingeniero y en la de muchos otros compañeros.

A mis amigos y hermanos que alguna vez me dijeron: ¨Tu puedes hacerlo y lo harás bien¨, han sido parte importante en quien soy y en lo que hago. Gracias a Nelly, Oliva, Isabel, Laura, Christian, Pato, Josué, Mariana, Erendira, Diana, Celestina… a todos los amigos que conocí en el IIE y en la ESIME.

ÍNDICE

Índice de figuras

Índice de tablas.

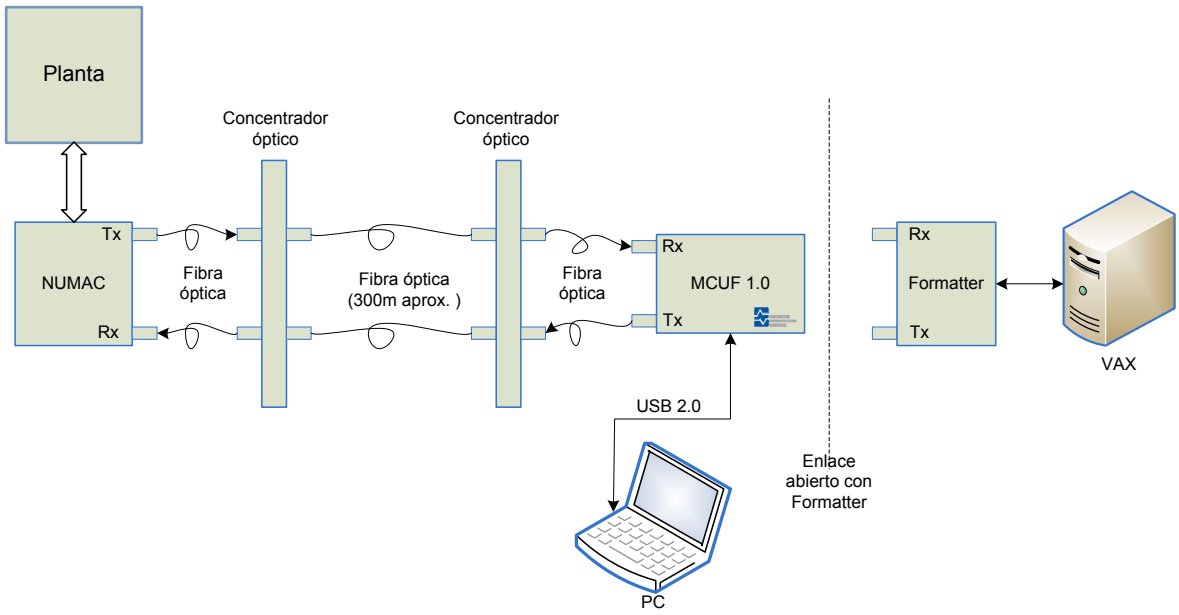
"SISTEMA DE ALMACENAMIENTO DE DATOS UTILIZANDO MEMORIA SDRAM, UN FPGA Y COMUNICACIÓN USB"

Problemática

La Central Nucleoeléctrica Laguna Verde (CNLV) cuenta en sus dos unidades de generación de energía eléctrica y en su Centro de Entrenamiento, con el SIIP o Sistema Integral de Información de Proceso para efectuar funciones de cálculo, monitoreo y apoyo en la correcta y óptima operación de cada unidad. Estas unidades estan equipadas con reactores de agua hirviente (BWR-5, Boiling Water Reactor) y cuentan con capacidad para generar 682.44 millones de watts. El reactor nuclear es de la marca GE y el conjunto de turbinas y turbogenerador son de la marca Mitsubishi. En la CNLV, la energía que mueve las turbinas para generación de energía eléctrica es obtenida mediante el calor generado en el proceso de fisión nuclear al interior del reactor. Este calor es utilizado para calentar agua y generar el vapor que mueve las turbinas. Las dos unidades de CNLV aportan aproximadamente el 5% de la energía eléctrica que requiere el país.

Después de más de diez años de funcionamiento continuo del SIIP en la planta CNLV, para prevenir futuros problemas de compatibilidad y obsolescencia de equipos, es necesario realizar el reemplazo del sistema de adquisición de datos del SIIP en ambas unidades de la CNLV. El sistema de adquisición de datos debe permitir la comunicación bidireccional con equipos NUMAC (Nuclear Measurement, Analysis and Control), es decir la recepción y envío de mensajes por medio de fibra óptica desde y hacia estos equipos.

Para la comunicación con el equipo NUMAC, es necesario desarrollar electrónica especial para recibir la información proveniente de estos equipos por fibra óptica, la cual actualmente es recibida por equipos llamados Formatters, transferirla a una PC y posteriormente al SIIP. Para cumplir con esta funcionalidad se requiere de un nuevo módulo electrónico que reciba los datos por fibra óptica, y los transfiera a un puerto USB.



Sustitución del Formatter en el SIIP

Este nuevo módulo electrónico debe satisfacer las necesidades de operación de tal forma que sea un dispositivo que tenga toda la capacidad para remplazar a los Formatters, debe realizar las tareas de codificación y decodificación de mensajes, almacenamiento temporal y conectividad USB de manera eficiente. Se sugiere el empleo del bus USB, ya que ya que este bus tiene la capacidad de conectar hasta 127 dispositivos utilizando un mismo puerto y estos pueden ser conectados y desconectados en cualquier instante sin necesidad de reiniciar la PC (*hot-plugging*), lo que ha hecho del USB unas de las interfaces más utilizadas en la comunicación con varios tipos de dispositivos periféricos, además de que en la actualidad la mayoría de los equipos de cómputo ya no cuentan con las

interfases tradicionales como el puerto serial o el paralelo. La comunicación USB y el almacenamiento de datos son requerimientos técnicos que debe cumplir el módulo electrónico para ser integrante del nuevo Sistema de Adquisición de Datos de la Central Núcleo eléctrica de Laguna Verde y son el tema que aborda el presente trabajo.

Objetivo

Desarrollar un sistema de almacenamiento de datos utilizando una memoria SDRAM, un FPGA para implementar el control de las señales de acceso a la memoria y comunicación USB para la transferencia de datos desde y hacia una PC. Este sistema se integrará al diseño del nuevo módulo electrónico que se empleará para sustituir a los Formatters dentro el proceso de reemplazo del Sistema de Adquisición de Datos de la Central Nucleoeléctrica de Laguna Verde.

Capítulo 1 Antecedentes

1.1 Especificación del módulo electrónico (MCUF)

El módulo electrónico que actualizará el sistema de adquisición de datos (figura 1.1) debe recibir la información proveniente de equipos NUMAC vía fibra óptica, almacenarla en una memoria local y posteriormente transferirla a una PC mediante un puerto USB 2.0. Así mismo, deberá procesar el envío de datos desde la PC al módulo mediante puerto USB 2.0 y la transferencia desde la memoria local del módulo hacia los equipos NUMAC mediante fibra óptica. La información enviada y recibida mediante fibra óptica debe cumplir con el protocolo IBM 3270, por lo que es necesario que se integren módulos de codificación y decodificación en el módulo electrónico para dar formato a los mensajes enviados y que éstos sean reconocidos por el equipo NUMAC, así como para obtener los datos de los mensajes recibidos.

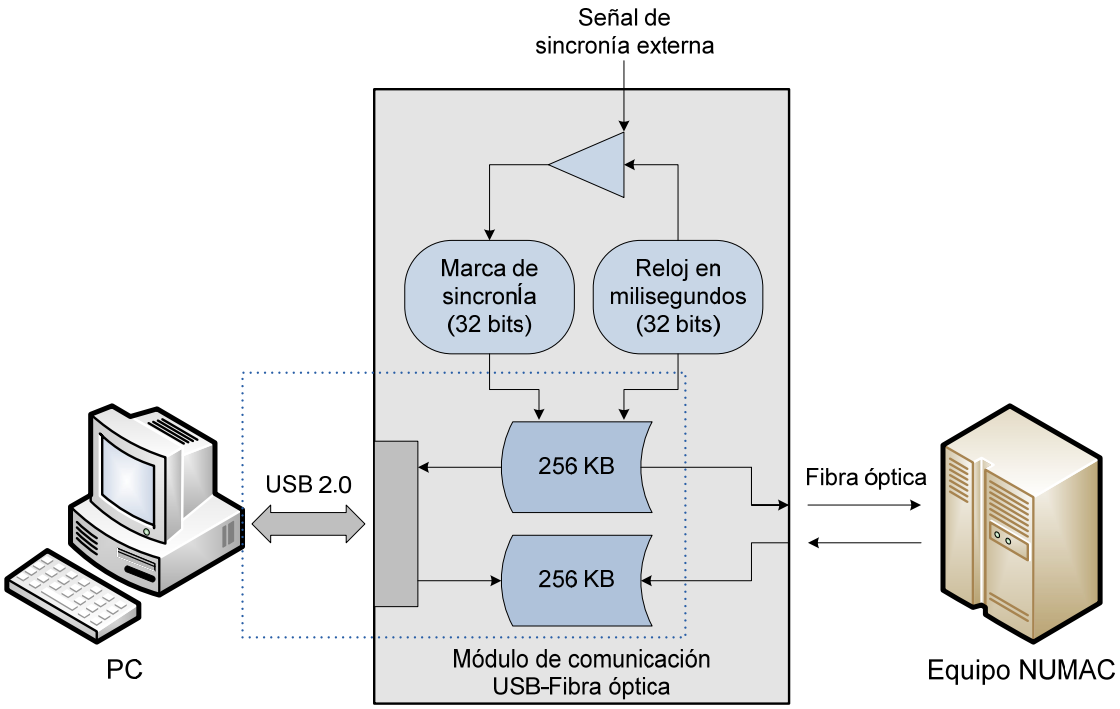


Figura 1.1 Módulo de comunicación USB-Fibra óptica.

La sección marcada con línea punteada en la figura 1.1 comprende la comunicación USB y el almacenamiento temporal de datos, lo cual concierne a este trabajo. La comunicación USB se basa en un controlador USB de Cypress®, concretamente el CY7C68013A (EZ-USB FX2LP) mientras que el almacenamiento se basa en una memoria SDRAM (MT48LC16M16A2) de Micron®. La configuración y acceso a la memoria se realiza mediante un FPGA Spartan3 de Xilinx®. El código que se utiliza para programar el controlador USB es C mientras que la programación del FPGA se realizó utilizando VHDL. De aquí en adelante el módulo electrónico al que se integrará el sistema que se desarrolla en el presente trabajo será referido como módulo de comunicación USB-Fibra óptica o MCUF.

1.2 Protocolo propietario de comunicación por puerto USB

El protocolo de usuario utilizado por el módulo de comunicación USB-Fibra óptica reconoce cuatro comandos enviados de la PC al módulo; ***Recibe***, ***Transmite***, ***Sincroniza*** y ***Reset*** a los cuales el módulo contesta con su respuesta correspondiente; ***Recibido***, ***Transmitido***, ***Sincronizado*** y ***Reinicio***.

La trama correspondiente a cada mensaje enviado al módulo tiene un formato específico el cual está integrado por tres partes básicas; un identificador de comando expresado en dos bytes, el número de bytes de datos a enviar también expresado en dos bytes y los bytes de datos que se desean enviar, esta última parte puede variar en tamaño o incluso no existir en algunos de los mensajes.

Figura 1.2 Formato de los mensajes enviados por la PC.

La respuesta a cada uno de los mensajes tiene un formato similar en el que se incluye un código de la respuesta del comando enviado (2 bytes), un grupo de 2 a 14 bytes

que representan parámetros del módulo y varios bytes de datos en el caso de que se trate de la respuesta a un comando ***Recibe***.

Figura 1.3 Formato de las respuestas a los comandos enviados por la PC.

Las tablas que se presentan a continuación contienen información referente a los datos en cada uno de los campos de la trama para los mensajes enviados, así como de los datos correspondientes a los campos de la trama de respuestas a los comandos. Como puede observarse, la longitud de los comandos enviados y las respuestas recibidas es variable dependiendo del comando del que se trate. Cada uno de los comandos y su respectiva respuesta es explicado posteriormente.

Tabla 1.1 Comandos enviados por la PC.

Comando	Identificador de comando	Número de bytes de datos	Bytes de datos	Tamaño
Recibe	0x0001	-	-	2 bytes
Transmite	0x0002	Variable	Variable	Variable
Sincroniza	0x0003	0x0004	4 bytes	8 bytes
Reset	0x0004	-	-	2 bytes

Tabla 1.2 Mensajes de respuesta a los comandos enviados por la PC.

Respuesta	Identificador de respuesta	Parámetros	Bytes de datos	Tamaño
Recibido	0x0101	14 bytes	Variable	Variable
Transmitido	0x0102	2 bytes	-	4 bytes
Sincronizado	0x0103	2 bytes	-	4 bytes
Reinicio	0x0104	-	-	2 bytes

1.2.1 Comando *Recibe*

Este comando se transmite de la PC al módulo y consta únicamente de los 2 bytes correspondientes al identificador de comando, no hay información adicional que preceda al identificador. Mediante este comando se solicita al módulo toda la información almacenada hasta el momento.

Una vez enviado el comando ***Recibe***, el módulo contesta con la respuesta ***Recibido*** la cual incluye información de parámetros del módulo (***estado***, ***reloj***, ***nbd***, etc) y los datos que habían sido almacenados por el módulo y que fueron recuperados. La longitud de esta respuesta varía en función del número de datos almacenados por el módulo al momento de enviar el comando ***Recibe***. En la tabla 1.3 se muestra la descripción del mensaje de respuesta correspondiente al comando ***Recibe***.

Tabla 1.3 Respuesta al comando *Recibe*.

Byte	Campo	Valor	Descripción
0-1	Identificador de respuesta	0x0101	***Recibido***, respuesta al comando ***Recibe***.
2-3	***Estado***	Variable	Estado de la información en el módulo y de la memoria de recepción de información.
4-7	***Reloj***	Variable	Valor actual del contador del "Reloj en milisegundos".
8-11	***nbd***	Variable	Número de bytes de datos enviados a la PC.
12-15	***Msinc***	Variable	"Marca de sincronía" actual en el módulo.
16,17,…	***Datos***	Variable	Mensajes de datos recuperados.

1.2.2 Comando *Transmite.*

Este comando, al igual que ***Recibe***, es enviado desde la PC al módulo sin embargo éste si cuenta con información adicional en los campos *número de bytes de datos* y *bytes de datos* de la trama de mensajes enviados por la PC. Este comando es utilizado para enviar información hacia el equipo NUMAC, su longitud varía pues el tamaño de los mensajes que deben ser enviados a NUMAC es variable.

Tabla 1.4 Comando *Transmite*.

Byte	Campo	Valor	Descripción
0-1	Identificador de comando	0x0002	Comando ***Transmite***.
2-3	Número de bytes de datos	Variable	Número de bytes de datos que deben ser enviados a NUMAC.
4,5,…	Bytes de datos	Variable	Bytes de datos correspondientes a mensajes enviados a NUMAC.

A cada comando ***Trasmite*** el módulo responde con ***Transmitido*** el cual se integra de los 2 bytes de identificador de respuesta e información adicional de parámetros del módulo, en este caso los dos bytes correspondientes al parámetro ***estado***.

Tabla 1.5 Respuesta al comando *Transmite*.

Byte	Campo	Valor	Descripción
0-1	Identificador de respuesta	0x0102	***Transmitido***, respuesta al comando ***Transmite***.
2-3	***Estado***	Variable	Estado de la información en el módulo y de la memoria de recepción de información.

1.2.3 Comando *Sincroniza*

El comando ***Sincroniza***, transmitido de la PC al módulo, se integra del identificador de comando (2 bytes) y 6 bytes de información adicional (cuatro de estos, correspondientes a la hora en milisegundos). La función de este comando es la de actualizar el reloj o contador de tiempo real interno.

Tabla 1.6 Comando *Sincroniza*.

Byte	Campo	Valor	Descripción
0-1	Identificador de comando	0x0003	Comando ***Sincroniza***.
2-3	Número de bytes de datos	0x0004	Número de bytes que se enviarán para actualizar el reloj interno.
4-7	Bytes de datos	Variable	El valor de estos 4 bytes es con que se cargará el reloj interno al enviar el comando ***Sincroniza***.

La respuesta al comando ***Sincroniza*** es muy parecida a la respuesta del comando ***Transmite***, consta de los 2 bytes de identificador de la respuesta e información adicional de ***estado***.

Tabla 1.7 Respuesta al comando *Sincroniza*.

Byte	Campo	Valor	Descripción
0-1	Identificador de respuesta	0x0103	***Sincronizado***, respuesta al comando ***Sincroniza.***
2-3	***estado***	Variable	Estado de la información en el módulo.

1.2.4 Comando *Reset*

El comando ***Reset*** es el más sencillo de los cuatro comandos en cuanto a la trama enviada. Solo se envían dos bytes de identificador de comando y se reciben 2 bytes de identificador de respuesta. La función de este comando es la de poner en condiciones iníciales a la lógica encargada del almacenamiento, el reloj interno y los módulos de codificación y decodificación. Al enviar un comando ***Reset*** las memorias del sistema de almacenamiento regresarán a su estado vacío y el reloj interno se reiniciará.

Tabla 1.8 Comando *Reset*.

Byte	Campo	Valor	Descripción
0-1	Identificador de comando	0x0004	Comando ***Reset***. Solicita el reinicio del módulo.

La respuesta al comando ***Reset*** consta solo de 2 bytes, e indica que el módulo se ha reiniciado.

Tabla 1.9 Respuesta al comando *Reset*.

Byte	Campo	Valor	Descripción
0-1	Identificador de respuesta	0x0104	***Reinicio***, respuesta al comando ***Reset***.

1.3 Sistema de almacenamiento.

El módulo MCUF debe recibir continuamente la información proveniente de equipos NUMAC y almacenarla en memorias internas de al menos 256 KB, una memoria o región separada de memoria por cada dirección en la que se transmiten los datos; PC a equipo NUMAC y viceversa. La figura 1.2 muestra un diagrama básico del sistema de almacenamiento y las posibles direcciones de transferencia. La memoria debe ser escrita y leída como una cola FIFO (First In First Out). En caso de que se escriban datos más allá del límite *ML Llena* y el espacio disponible en la memoria de lectura no sea suficiente para

almacenar un mensaje completo, la memoria será reiniciada y los datos de este mensaje se escribirán en las primeras localidades de la memoria. Los datos previamente almacenados y que no fueron leídos se perderán y no podrán ser recuperados, almacenando así solo la información más actual y desechando toda la información que llenó la memoria y no fue solicitada por la PC.

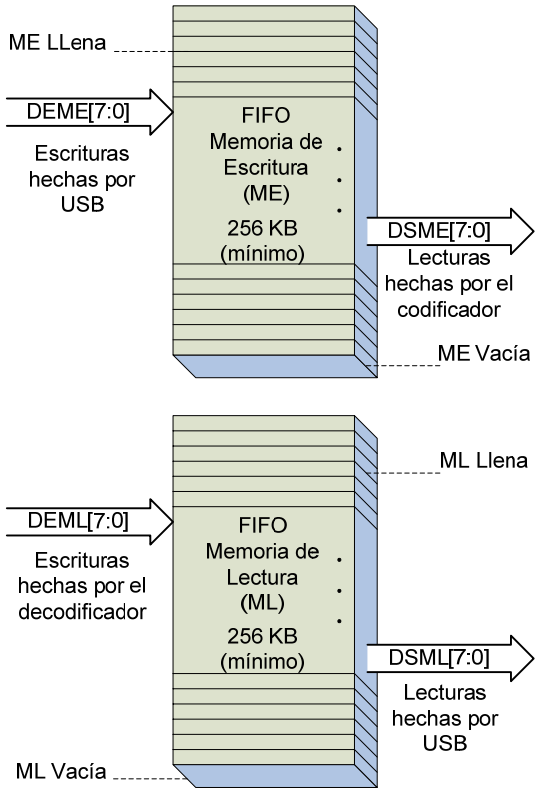


Figura 1.4 Memorias de almacenamiento temporal.

El sistema de almacenamiento de datos es accedido por cuatro clientes cuyas peticiones de escritura o lectura pueden presentarse de forma simultánea y se atienden con base en un nivel de prioridad asignado a cada cliente. Los cuatro posibles clientes son:

- Escrituras hechas por USB, por medio de las cuales se escribe el dato DEME[7:0] en la memoria de escritura (ME).

- Lecturas hechas por el codificador, con las cuales se lee el dato DSME[7:0] de la memoria de escritura.
- Escrituras hechas por el decodificador, con las cuales se escribe el dato DEML[7:0] en la memoria de lectura (ML).
- Lecturas hechas por USB, por medio de las cuales se lee el dato DSML[7:0] de la memoria de lectura.

Cuando se transmite desde la PC al equipo NUMAC, mediante *escrituras USB* se almacenan datos en la *memoria de escritura* (ME) haciendo uso del comando ***Transmite***, las *lecturas del codificador* acceden a los datos almacenados en la *memoria de escritura* (ME) para posteriormente codificarlos y transmitirlos por fibra óptica utilizando el protocolo IBM 3270. En el sentido inverso, un módulo decodificador se encarga de recibir los datos por fibra óptica, decodificarlos y posteriormente escribirlos en la *memoria de lectura* (ML) mediante *escrituras del decodificador*. A cada mensaje recibido se le agregan 4 bytes que representan el valor actual del reloj interno del módulo para así llevar un registro del tiempo específico en el que un mensaje fue almacenado. Haciendo uso del comando ***Recibe*** se accede (*lecturas USB*) a los datos que han sido almacenados en la memoria de lecturas (ML). Una vez que los datos han sido recuperados por la PC, ésta pone los datos a disposición de las entidades correspondientes dentro del sistema integral de información proceso (SIIP) para realizar su análisis.

Como ya se mencionó, el presente trabajo contempla la implementación de una parte del módulo electrónico (MCUF) que remplazará a los Formatters en el sistema de adquisición de datos de la CNLV, concretamente lo relacionado a la implementación de la comunicación vía USB y el desarrollo del sistema de almacenamiento del mismo módulo. En los capítulos posteriores se presenta información referente a la especificación USB y la tecnología de memoria que se utiliza en este trabajo, se presenta también el desarrollo de sistema y posteriormente la integración de éste en el módulo de comunicación USB-Fibra óptica así como algunas pruebas realizadas al módulo.

Capítulo 2 El bus serie universal (USB).

2.1 Introducción

El bus serie universal (USB) es una especificación desarrollada por Compaq, Intel, NEC, y Hewlett Packard entre otras. La idea de desarrollar este bus surge de la necesidad de conectar un gran número de dispositivos periféricos a una computadora personal mientras ésta se encuentra funcionando normalmente y sin tener que reiniciar todo el sistema después de haber agregado el nuevo periférico, los dispositivos son del tipo *hot-pluggable*. Otra de las características de este bus es que el usuario no necesita de un conocimiento profundo acerca del hardware para instalar un nuevo dispositivo, ya que el sistema se encarga de diferenciar cada uno de los dispositivos conectados y de asignar el controlador más adecuado. La velocidad de transferencia del bus varía, y puede estar dentro de tres posibles rangos, a saberse: *Low Speed*, *Full Speed* y *High Speed*.

Tabla 2.1 Rango de velocidades soportadas por el bus USB.

Velocidad	Desempeño	Aplicaciones	Características
Low Speed	10 – 100 kb/s	Teclados, Dispositivos señaladores, Periféricos para juegos y realidad virtual.	Bajo costo (el más bajo). Facilidad de uso. Conexión-Desconexión dinámica. Múltiples periféricos.
Full Speed	500kb/s – 10 Mb/s	Audio, Micrófonos, Teléfono.	Bajo costo. Facilidad de uso. Ancho de banda garantizado. Latencia garantizada. Conexión-Desconexión dinámica. Múltiples periféricos.
High Speed	25 – 400 Mb/s	Video, almacenamiento.	Facilidad de uso. Múltiples periféricos. Ancho de banda garantizado. Conexión-Desconexión dinámica. Mayor ancho de banda.

La especificación del USB más actual es la versión 2.0 y remplaza a las versiones anteriores 1.0 y 1.1. Un dispositivo que cumple con la especificación mas reciente debe también dar soporte para versiones anteriores, así un dispositivo USB 2.0 puede realizar transferencias en el rango de *Full Speed*, sin embargo un dispositivo que cumple con la versión 1.1 de la especificación no es capaz de realizar transferencias en el rango de *High Speed*.

2.2 Interfase eléctrica

La transmisión de datos y alimentación se realiza por medio de 4 conductores, dos de ellos (D+ y D-) son utilizados exclusivamente para la señalización necesaria en las transmisiones de datos, son conductores trenzados en los cuales se utiliza voltaje diferencial para la señalización.

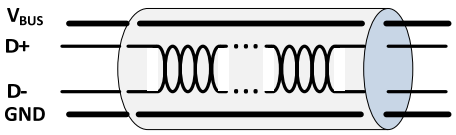


Figura 2.1 Líneas disponibles en el bus USB.

En dispositivos que trabajan a *Low Speed* un '1' lógico es transmitido colocando un voltaje mayor a 2.8V en la línea D+ y un voltaje menor a 0.3V en la línea D-, ambas líneas conectadas a tierra a través de resistores de entre 14.75KΩ a 28.80KΩ. Mientras que un '0' lógico se logra colocando un voltaje mayor a 2.80 V en la línea D- y un voltaje menor a 0.3V en la línea D+ utilizando los mismos resistores. Del lado del receptor si D+ es al menos 200mV mayor que D- esto será interpretado como un '1', pero si D+ es al menos 200mV menor que D- esto representará un '0'.

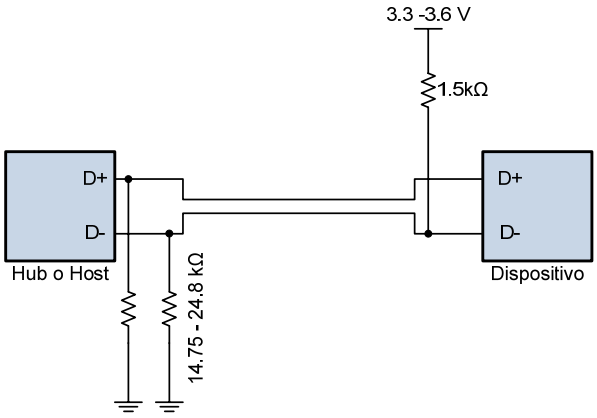


Figura 2.2 Conexión *host*-dispositivo.

La polaridad de estas señales es invertida dependiendo de la velocidad a la que se realizan las transferencias. En *Low Speed* y *Full Speed* un nivel lógico '0' es considerado como el estado J del bus y un '1' representa el estado K del bus, para *high speed* esto funciona a la inversa; un '0' representa el estado K mientras que un '1' el estado J. El resistor de 1.5 KΩ en la figura 2.2 sirve como identificador de la velocidad a la que trabajará el dispositivo; este resistor está conectado a la línea D+ para dispositivos *full speed* o a D- para dispositivos *low speed*. En el caso de dispositivos *high speed* la conexión de este resistor es idéntica a la de *full speed*, de hecho un dispositivo *high speed* siempre se configura inicialmente como *full speed* para después notificar que el dispositivo puede funcionar a una velocidad superior y reconfigurarse automáticamente.

Los conductores restantes V_{BUS} y GND, son una fuente de alimentación de 5 VCC que puede ser utilizada por el dispositivo. Dada esta capacidad del bus, los dispositivos pueden ser alimentados por el bus (*bus powered*) o de forma independiente (*self powered*).

En el caso de dispositivos alimentados por el bus, su consumo de corriente no debe exceder 100 mA. Aunque el host puede suministrar hasta 500 mA y los dispositivos pueden ser configurados para que el host les suministre más de 100 mA, en caso de que se conecten varios dispositivos al bus, es recomendable el uso de una fuente de alimentación común a todos los dispositivos o en su caso una fuente de alimentación independiente por cada dispositivo.

2.3 Terminología

La finalidad de cualquier bus es la de transferir información desde y hacia la PC por medio de algún dispositivo conectado al bus. En el caso de USB, cada dispositivo puede soportar una parte que genera datos y una que recibe datos, de tal forma que varios controladores (software) serian necesarios, uno por cada una de estas dos opciones. Estos controladores así como otros componentes son importantes en el esquema USB para que el objetivo de intercambiar información se lleve a cabo de manera satisfactoria. Algunos de los componentes más importantes en un sistema USB son:

- Host.
- Dispositivo o función.
- Puntos terminales (o *Endpoints*).
- Tuberías (o *Pipes*)

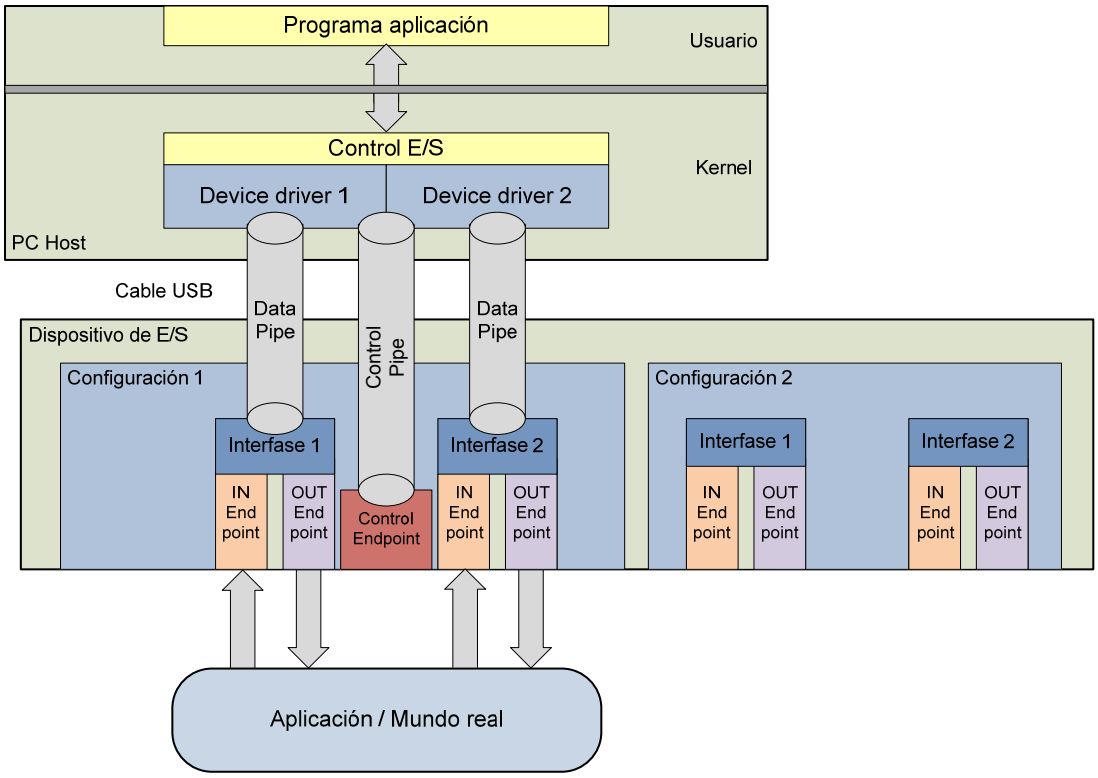


Figura 2.3 Partes principales de un sistema de comunicación USB.

2.3.1 Host

En un sistema USB existe un solo *host* cuya interfase hacia el sistema principal (CPU) es llamado controlador USB. Este controlador USB es implementado como una combinación de hardware, firmware y software. Así, un concentrador raíz es integrado en el sistema principal, por medio del controlador USB, para así proveer varios puntos de conexión o puertos en los cuales se podrán conectar ya sea concentradores (hubs) o dispositivos. Un dispositivo puede ser conectado a un concentrador y éste a otro concentrador y así sucesivamente. Sin embargo solo pueden tenerse 6 capas como máximo.

El *host* es el encargado de iniciar, dirigir y terminar todas las comunicaciones, por tanto no puede haber comunicación entre dispositivos. Un dispositivo no puede iniciar una transferencia si ésta no le es solicitada por el *host*. La única excepción sucede cuando un dispositivo es puesto en un estado conocido como suspendido (estado de bajo consumo), en este estado el dispositivo puede enviar al *host* una señal llamada *remote wakeup*, para así informar al *host* que el dispositivo regresará a su modo de operación normal.

2.3.2 Dispositivos o funciones

Un dispositivo USB puede ser:

- Un concentrador, el cual provee al sistema de puertos USB adicionales.
- Una función, la cual da alguna capacidad al sistema, por ejemplo un control para juegos, un ratón o una tarjeta de adquisición de datos.

Los dispositivos USB son clasificados como tales, si:

- Comprenden el protocolo USB.
- Responden adecuadamente a operaciones USB estándar, tales como configuración o reinicialización.

- Cuentan con la capacidad de proporcionar información descriptiva tales como *VendorID* (VID), *ProductID* (PID) o número de *endpoints* entre otros. Esta información está almacenada en una tabla de descriptores, a los cuales se accede mediante transferencias tipo *Control*.

Los dispositivos USB están divididos en clases, dentro de las cuales se encuentran: hubs, interfase humana (HID), imagen y almacenamiento masivo entre otros. Cada dispositivo USB es accedido usando una dirección específica, la cual es asignada después de que éste es conectado, concretamente en la etapa de enumeración la cual consiste en la detección e identificación del dispositivo. Datos tales como el fabricante (Vendor ID), producto (Product ID), clase de dispositivo, opciones de alimentación y número de *endpoints* son proporcionados por el dispositivo durante la enumeración. La transferencia de dicha información se realiza a través del *endpoint 0*, o *endpoint* de control, el cual debe estar implementado en todo dispositivo USB.

2.3.3 Endpoint

Los datos pueden entrar o salir del sistema, cada uno de los puntos por los cuales la información abandona o entra al sistema es llamado *endpoint*, por tanto existen *endpoints* de entrada (*IN endpoint*) y de salida (*OUT endpoint*). La dirección del flujo de datos es relativa al host. Así, un *endpoint* de entrada es una fuente de datos para el host mientras que uno de salida es un receptor de los datos que el host envía. Físicamente, es una porción de memoria tipo FIFO (*First In First Out*) que se direcciona de manera única dentro de un dispositivo USB en la cual los datos a ser transferidos son almacenados de forma temporal.

2.3.4 Interfases

Una función o dispositivo útil requiere por lo general más de un *endpoint*. Una colección de *endpoints* de entrada y salida conforma una interfase. Un dispositivo puede

tener más de una y si es el caso, el sistema operativo debe contar con un controlador para cada interfase.

2.3.5 *Pipe*s o tuberías

Es una abstracción lógica que el sistema operativo utiliza para comunicar un controlador en el host con una interfase en el dispositivo o función. Concretamente conectan la parte lógica del controlador (software) con los *endpoints* que se encuentran en el dispositivo.

2.3.6 Configuración

Algunos dispositivos necesitan más de una interfase, por ejemplo, un teléfono necesita una interfase para controlar el audio, una más para el teclado y tal vez una más para el despliegue en pantalla. Cada una de estas interfases es manejada por un controlador distinto del sistema operativo. A la colección de varias interfases se le llama configuración. Solo una configuración puede estar activa en todo tiempo, la configuración es lo que define los atributos y características de algún modelo de un dispositivo USB en especial.

2.4 Topología del bus

El bus serie universal conecta dispositivos USB con el *host*, el cual soporta un máximo de 127 dispositivos esclavos, entre concentradores y funciones. Físicamente, la interconexión de dispositivos sigue una topología estrella multicapa, en cuyo centro se encuentra el *host* o maestro mientras que en las siguientes capas se encuentran *hubs* o concentradores. Como se muestra en la figura 2.4, cada segmento del bus es una conexión punto a punto entre un *host* y un *hub* o un dispositivo, estos últimos también llamados funciones.

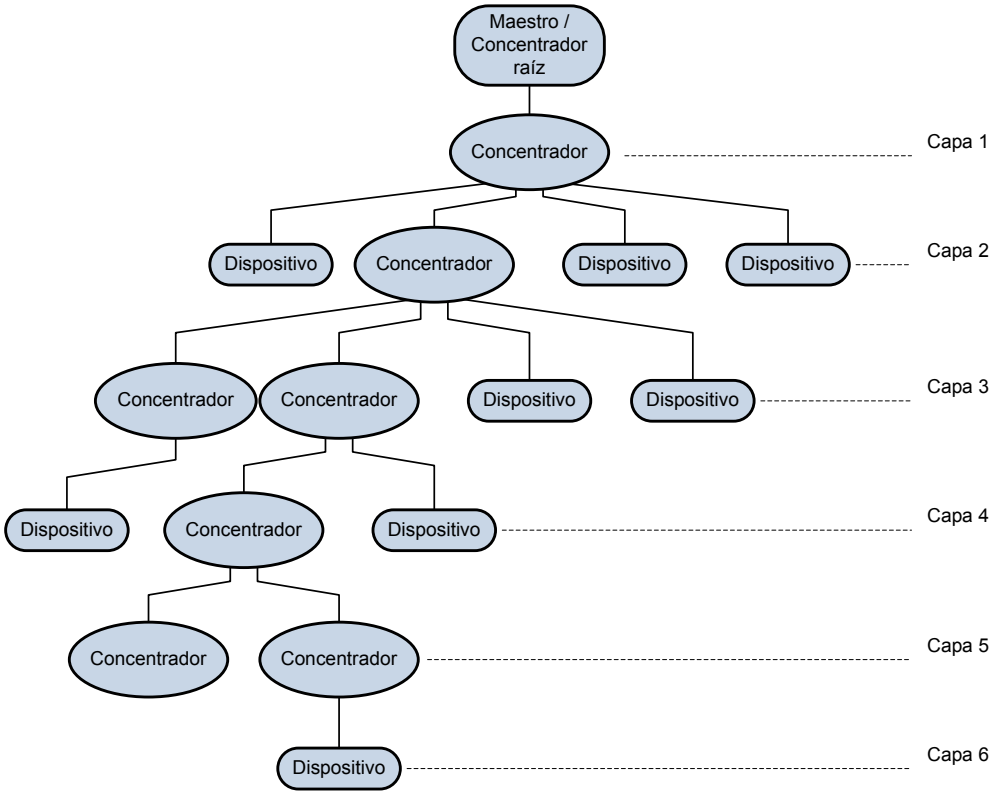


Figura 2.4 Topología del bus.

2.5 Protocolo

En el bus USB todas las transferencias de datos son iniciadas por el *host*. Todas las transacciones comprenden el envío de tres paquetes. Cada transacción inicia con el envío, por el controlador del *host*, de un paquete en el que se describe el tipo y la dirección de la transacción, la dirección del dispositivo y el número de *endpoint* por el que se realizará la transferencia, a este paquete básico se le denomina *token*. Posteriormente la fuente desde la cual se realiza la transferencia envía un paquete de datos, denominado *data*, o notifica en caso de que no haya datos que transferir. Finalmente el destinatario responde

con un paquete *handshake* con el cual se indica si la transferencia fue satisfactoria o no. Cada paquete consta de tres partes; inicio, información y fin.

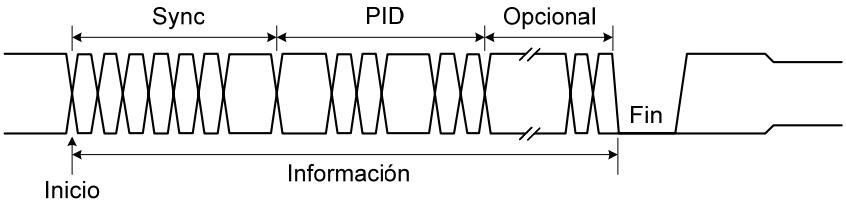


Figura 2.5 Paquete USB.

El inicio de un paquete es marcado con la transición del estado J al estado K. Después de esto, más transiciones tienen lugar para generar una secuencia de sincronización *sync*. Una transición sucede en el bus cada 666.6 ns, 83.3 ns o 2.083 ns dependiendo de si se trata de un bus *low speed*, *full speed* o *high speed* respectivamente. El bus USB no incluye una señal de reloj, por lo que la primera operación a realizar en toda transacción es la sincronización, el receptor utiliza la secuencia *sync* para ajustar su reloj a las transiciones recibidas y así asegurar la correcta recepción de la información precedente. La secuencia *sync* termina con la inclusión de dos estados K. Para un bus *high speed* el host enviará 32 bits en la secuencia *sync* (*KJKJKJ…KJKK*), si el bus es *low speed* o *full speed* solo habrá 8 bits en la secuencia.

La siguiente parte del paquete es el PID (packet identifier), el identificador del paquete. El total de bytes de información en el paquete varía de 1 a 3074 bytes. El primer byte es el identificador de paquete el cual definirá como serán interpretados los datos siguientes. El identificador del paquete está formado por 4 bits y el complemento de estos, esto permite al receptor verificar si existe error en el identificador del tipo de paquete. En la especificación USB existen cuatro tipos de paquetes: *token*, *data*, *handshake* y *special.* Estos son mostrados en la siguiente tabla junto con la codificación para cada tipo.

Tabla 2.2 Tipos de paquetes USB.

Tipo de paquete	Valor PID <3:0>	Descripción
Token	0001	OUT
	1001	IN
	0101	SOF (Start of frame)
	1101	SETUP
Data	0011	DATA0
	1011	DATA1
	0111	DATA2
	1111	MDATA
Handshake	0010	ACK
	1010	NAK
	1110	STALL
	0110	NYET (aún sin respuesta)
Special	1100	PRE (Token)
	1100	ERR (Handshake)
	1000	Split (Token)
	0100	Ping (Token)
	0000	Reservado

Los paquetes tipo *token* establecen las condiciones necesaria para transmitir correctamente un paquete tipo *data*, mientras que los paquetes *handshake* se utilizan para notificación acerca del resultado de la transmisión de un paquete *data*; si ésta fue exitosa o no, es información que está contenida en un paquete *handshake*.

La última parte del paquete es un identificador de final de paquete EOP (*End Of Packet*), el cual varía dependiendo de la velocidad del bus. En transferencias a *high speed* el EOP se representa como la ausencia de transiciones durante el tiempo correspondiente a 40 bits, mientras que para *low speed* y *full speed* las líneas D+ y D- son puestas en bajo durante el tiempo correspondiente a 2 bits para señalizar el EOP.

El bus USB utiliza el esquema de codificación NRZI (Nonreturn to Zero Inverted) o *no retorno a cero invertido* en el cual un '1' se representa al mantener un mismo nivel lógico mientras que un '0' se representa como unta transición entre niveles lógicos. Así, una sucesión de 1's se observa como una señal estable en un mismo nivel, no así una sucesión de 0's la cual se observa como una serie de transiciones consecutivas entre niveles.

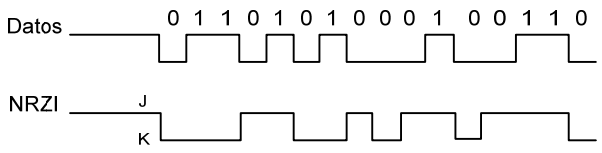


Figura 2.6 Codificación NRZI.

Dado que en el bus USB no existe una línea especial para la transmisión de una señal de reloj, en todo momento de la transmisión esta señal se encuentra embebida en los datos. Por esto, es necesario insertar bits de relleno, los cuales sirven para recuperar la sincronía que se estableció mediante la secuencia *Sync* al inicio del paquete. Estos bits de relleno son insertados siempre que se tenga una secuencia de seis 1's, de esta forma la lógica del receptor observará al menos una transición cada 7 bits. A este procedimiento se le llama *bit stuffing*.

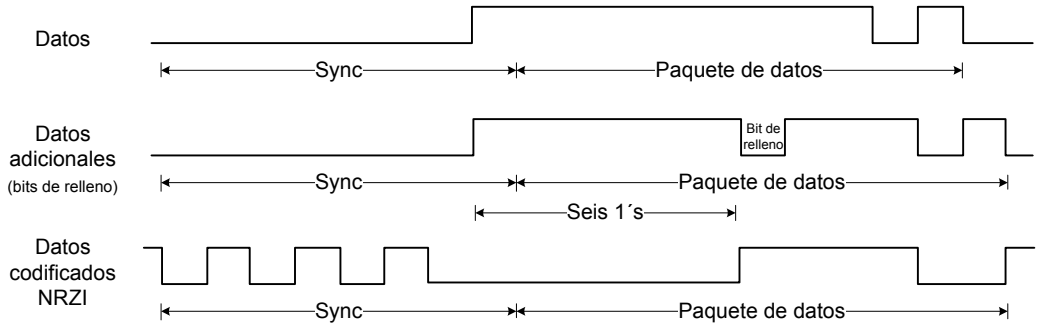


Figura 2.7 Codificación NRZI y bit stuffing.

2.5.1 Paquetes tipo TOKEN

SOF (Start of Frame) o inicio de *Frame*. El host envía un paquete SOF cada 1.0 ms, este intervalo de tiempo entre dos paquetes SOF es llamado *Frame*. En el caso de un host de *high speed*, se envían 8 paquetes SOF adicionales dentro de un *Frame*, el intervalo de tiempo entre estos paquetes es denominado *microframe*. Dentro de un *frame*/*microframe* pueden realizarse varias transacciones.

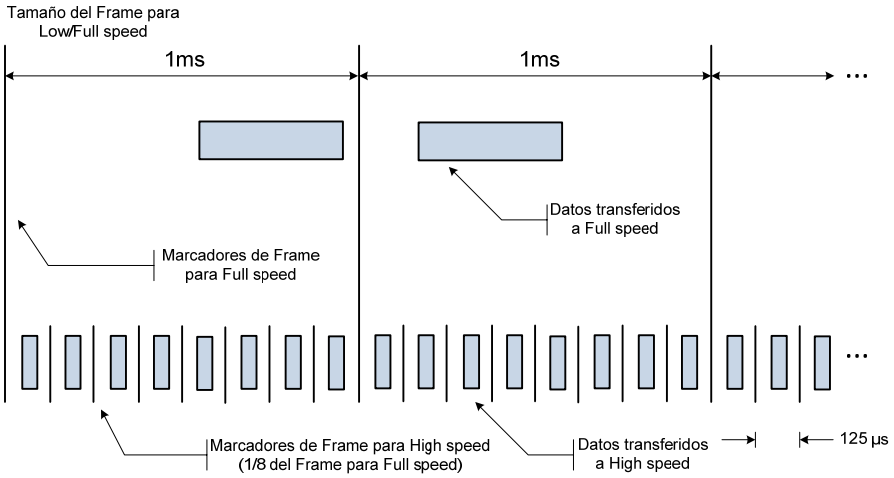


Figura 2.8 Frames y microframes.

La inclusión de *microframes* hace más complejo el control de las transacciones en el bus USB, sin embargo esto también reduce la complejidad de los buffers utilizados. Si no se utilizaran *microframes*, los dispositivos *high speed* deberían ser capaces de almacenar en sus buffers la información contenida en un *frame*, que son alrededor de 60 MB de datos incluidos entre *Frames*. El uso de *microframes* y de paquetes tipo *DATA2* y *MDATA* permiten a los dispositivos *high speed* contar con buffers de solo 40 KB. En una conexión *full speed* pueden transferirse hasta 1500 bytes de datos dentro de un *Frame*, mientras que a *low speed* solo pueden transferirse hasta 150 bytes por *Frame*.

Un paquete SOF contiene 11 bits de datos y 5 de CRC.

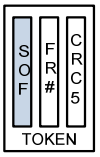


Figura 2.9 Paquete TOKEN: SOF.

Los 11 bits (FR#) dentro de un paquete es el número de *Frame*, puede ser utilizado por algunos dispositivos para sincronizar las transferencias de datos. Los *microframes* dentro de un *Frame* tiene el mismo número de *Frame*. Si un dispositivo *high speed* necesita sincronizar las transferencias de datos, es necesario que éste lleve una cuenta de

los paquetes SOF cada que el número de *Frame* cambie. Los paquetes SOF son los únicos que no cuentan con un campo *ADDR*, son transmitidos a todos los dispositivos conectados al bus.

IN, OUT* y *SETUP. Los paquetes IN establecen transferencias desde un dispositivo hacia el *host*, mientras que los paquetes OUT establecen transferencias desde *host* hacia un dispositivo. Con los campos ADDR y ENDP se puede seleccionar cualquier *endpoint* del dispositivo deseado.

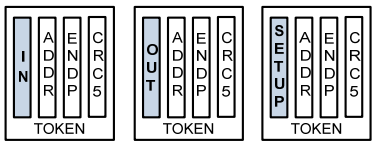


Figura 2.10 Paquetes TOKEN: IN, OUT y SETUP.

Los paquetes *SETUP* son un tipo especial de paquete *OUT* el cual tiene alta prioridad, lo cual significa que el dispositivo debe aceptarlo aún cuando tenga que abortar la transacción en curso, este tipo de paquetes siempre están dirigidos al *endpoint 0* bidireccional.

2.5.2 Paquetes tipo DATA

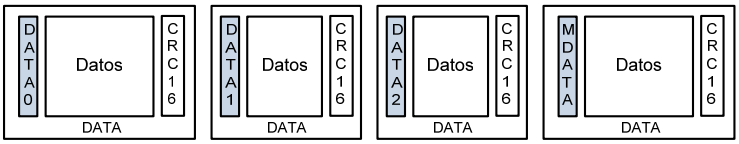


Figura 2.11 Paquete tipo DATA.

Las transferencias de datos establecidas por los paquetes *token* son implementadas mediante los paquetes DATA0, DATA1, DATA2 y MDATA. El número de datos varía entre 0 y 1023 bytes por cada paquete y siempre hay 16 bits de CRC. En la transferencia de datos siempre se alternan los paquetes DATA0 y DATA1 como medio para

verificar errores, si se reciben dos paquetes DATA0, por ejemplo, esto significa que hubo pérdida de información. Los paquetes DATA2 y MDATA son únicamente utilizados en transferencias isócronas para dispositivos *high speed.*

2.5.3 Paquetes tipo HANDSHAKE

Los paquetes de éste tipo son solo utilizados por el receptor para indicar el estado de la recepción de paquetes tipo *token* y *data*. Solo consisten de un identificador de paquete PID y son cuatro distintos:

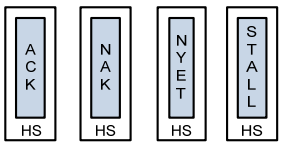


Figura 2.12 Paquetes tipo HANDSHAKE.

- *ACK* (acknowledge), indica la recepción exitosa de paquetes *token* o *data*.
- *NAK* (negative acknowledge), indica que el bus está temporalmente ocupado o que no cuenta con los recursos necesarios para atender las transacciones solicitadas.
- *STALL*, si existe algún problema durante la comunicación, un dispositivo enviará este paquete para indicar al host que se necesita atención especial. Por ejemplo, un dispositivo responderá con *STALL* a toda petición o comando que éste no entiende.
- *NYET* (not yet), tiene la misma función que *ACK*, pero además de indicar una transferencia exitosa indica también si el *endpoint* aún no está listo para recibir otra transferencia de salida. Este tipo de paquete solo se utiliza en transferencias *High Speed*.

2.6 Transferencias USB

La especificación USB contempla cuatro tipos de transferencias, cada una tiene características especiales tales como confiabilidad o tamaño máximo del paquete, esto da la oportunidad de elegir de entre ellas la que mejor se ajuste a las necesidades de la aplicación.

Tabla 2.3 Tipos de transferencias USB.

Tipo	Atributos	Tamaño máximo (Bytes)			Ejemplos
		LS	FS	HS	
Interrupt	Calidad + tiempo	8	64	1024	Teclado, ratón
Bulk	Calidad	-	64	512	Scanner, impresora
Isochronous	Tiempo	-	1023	1024	Audio, video
Control	Calidad + tiempo	8	64	64	Únicamente para control del bus.

2.6.1 Transferencias BULK

Este tipo de transferencia está diseñada para dar soporte a dispositivos que necesitan transferir cantidades relativamente grandes de información en tiempos altamente variables en los que la transferencia puede utilizar cualquier ancho de banda disponible. Este tipo de transferencia se realiza en paquetes de 8, 16, 32 o 64 bytes en *full speed* y 512 bytes en *high speed*. La solicitud de una transferencia tipo *Bulk* tiene las siguientes características:

- Acceso al bus en base al ancho de banda disponible en el momento de la transferencia.
- Reintento de transferencia en caso de errores ocasionales en el bus.
- Se garantiza la operación de transferencia pero no se garantiza que ésta será con un ancho de banda o retardo específico.

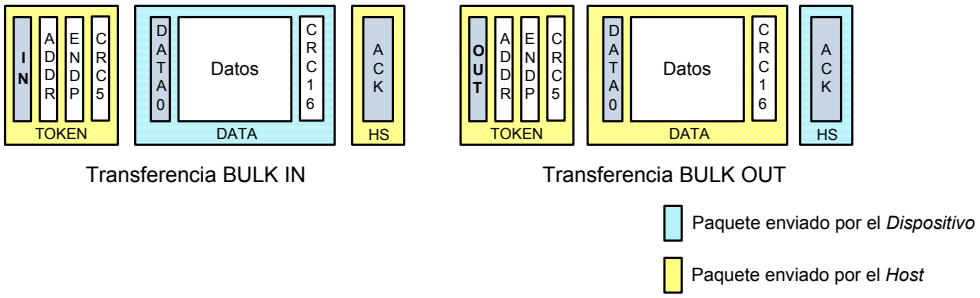


Figura 2.13 Transferencia BULK.

En una transferencia Bulk de entrada (*Bulk IN*) el *host* envía un paquete *token* de entrada (*IN Token*) al cual el dispositivo o función responderá con un paquete *data* o, de no ser posible la transferencia, con algún paquete *handshake* para indicar el estado del *endpoint*. Si la transferencia se realiza exitosamente el *host* enviará un paquete *ACK.*

En transferencias de salida (*Bulk OUT*) el *host* envía un paquete *token* de salida (*OUT Token*) seguido de un paquete *data*, el dispositivo responderá con *ACK* si la transferencia es exitosa o con algún otro paquete *handshake* dependiendo del estado del *endpoint*.

2.6.2 Transferencias INTERRUPT

Son transferencias similares al tipo *Bulk*. Los paquetes de datos pueden ser desde 1 hasta 64 bytes en *full speed* y hasta 1024 bytes en *high speed*. Una característica especial de estas transferencias es que los *endpoints* tienen un intervalo de encuesta (*polling interval*) asociado a ellos, en el cual se envía un *IN token* en el caso de *Interrupt IN* o un *Out token* seguido de un *data token* en el caso de *Interrupt OUT*, de esta forma las transacciones se realizan en una base de tiempo fija mediante encuestas periódicas a los *endpoints*.

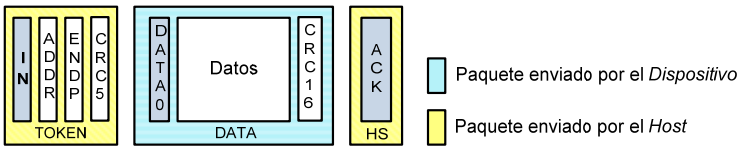


Figura 2.14 Transferencia INTERRUPT.

Si una transferencia tipo *Interrupt* no se completa satisfactoriamente se reintentará en el siguiente periodo de encuesta para asegurar así que la información se transfiera.

2.6.3 Transferencias ISOCHRONOUS

Son transferencias utilizadas principalmente en la transmisión de datos dependientes del tiempo (audio y video en tiempo real, por ejemplo). Los paquetes de datos pueden ser hasta de 1023 bytes en *full speed* o hasta 1024 bytes en *high speed*. Dado que lo mas importante en este tipo de transferencias es el tiempo en que se transfiere la información y no la integridad de ésta, en cada *frame* (o *microframe*) existe un ancho de banda asignado especialmente para transferencias *Isochronous*, los paquetes tipo *handshake* no son utilizados, no hay reintentos de transferencia y la corrección de errores se limita al uso de un CRC16.

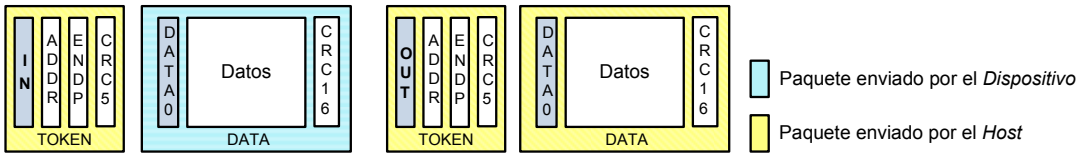


Figura 2.15 Transferencia ISOCHRONOUS.

En modo *full speed* solo una transferencia *Isochronous* por *endpoint* dentro del mismo *frame* puede realizarse, mientras que en modo *high speed* es posible realizar hasta tres transferencias por *endpoint* dentro de un mismo *microframe*. Para transferencias en *full speed* solo se utilizan paquetes DATA0. Los paquetes DATA0, DATA1, DATA2 y MDATA se utilizan en modo *high speed*.

2.6.4 Transferencias CONTROL

Las transferencias tipo control permiten el acceso a diferentes partes de algún dispositivo. Este tipo de transferencias son las que dan soporte a las operaciones de configuración entre el software cliente (en el *host*) y una función (dispositivo). Una transferencia tipo *Control* consta de dos o tres etapas;

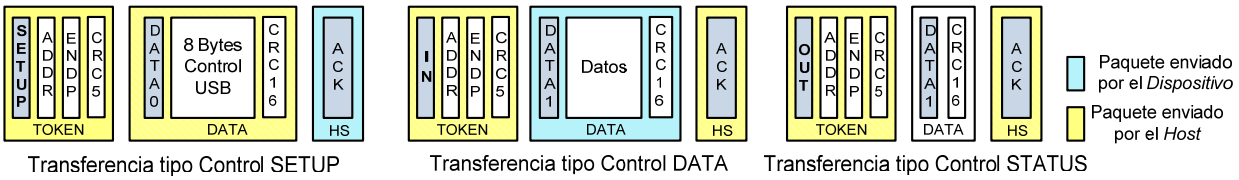
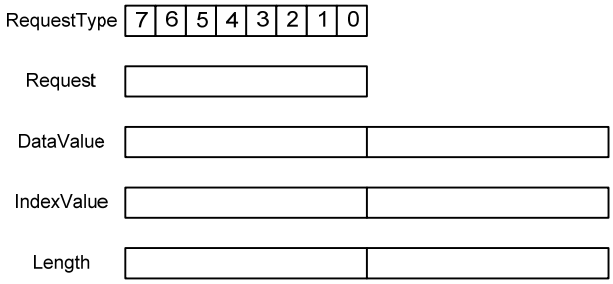


Figura 2.16 Transferencias CONTROL.

La primera etapa se llama *SETUP* y es en la que se establece el tipo de operación o solicitud que se realizará, el *host* envía 8 bytes (*setup packet*) de datos de control del bus. Posteriormente en la etapa *DATA* se transmiten los datos que se especificaron en la etapa *SETUP*. Para finalizar la transferencia se hace uso de la etapa *STATUS* en la que se indica si la transferencia se realizó satisfactoriamente o no. Las transferencias tipo control conforman el mecanismo por el cual el *host* accede a los descriptores de un dispositivo o hace solicitudes para modificar el comportamiento del dispositivo.

2.7 El paquete SETUP

Todos los dispositivos USB son capaces de responder a peticiones especiales hechas por el *host* a mediante transferencias tipo *Control* y a través del *endpoint 0* (*control endpoint*). El *setup packet* (paquete SETUP), paquete de 8 bytes en la primera etapa de una transferencia tipo *control*, es utilizado para indicar las diferentes peticiones que el *host* puede hacer a un dispositivo.



Bits del campo RequestType	Valores posibles
Bit 7	0 = Transferencia del Host al Dispositivo 1 = Transferencia del Dispositivo al Host
Bit 6:5	00 = Standard 01 = Clase 10 = Vendor 11 = Reservado
Bit 4:0	00000 = Dispositivo 00001 = Interfase 00010 = Endpoint 00011 = Otro

Figura 2.17 El paquete SETUP.

El parámetro *RequestType* en el *setup packet* indica la dirección de la transferencia, los bits 5 y 6 indican el tipo de petición; estas pueden ser *Standard*, según la *Clase* de dispositivo o específica del fabricante (*Vendor*). Los bits restantes especifican el destino de la petición, éste puede ser un dispositivo, una interfase o un *endpoint*.

Las peticiones estándar al que todo dispositivo USB debe ser capaz de responder son las siguientes:

Tabla 2.4 Peticiones estandar del host.

Petición	Descripción
Get_Status	Obtiene información acerca del suministro de energía al dispositivo y capacidad de *Remote wakeup*
Clear_Feature	Limpia o deshabilita características especiales como *Remote wake up* o *Test mode*
Set_Feature	Habilita características especiales.
Set_Address	Asigna la dirección del dispositivo durante la enumeración.
Get_Descriptor	Obtiene alguno de los descriptores del dispositivo.
Set_Descriptor	Actualiza o crea nuevos descriptores del dispositivo.
Get_Configuration	Obtiene el valor de la configuración utilizada actualmente.

Tabla 2.4 (continuación) Peticiones estandar del host.

Petición	Descripción
Set_Configuration	Establece la configuración activa del dispositivo.
Get _Interface	Obtiene el valor de la configuración alternativa (*alternate setting*) de la interfase.
Set_Interface	Establece el valor configuración alternativa de la interfase activa.
Sync_Frame	Establece la sincronía entre el dispositivo y el *host*, de tal manera que se puedan realizar transferencias *Isochronous* mediante múltiples *frames*.

La mayoría de las peticiones son para control del bus y solo las realiza el *host* como parte de la etapa de enumeración de cualquier dispositivo USB.

2.8 Enumeración

Al proceso de detección e identificación de cualquier dispositivo al ser conectado al bus, se le llama enumeración. Es en esta etapa en la que se define el comportamiento del dispositivo recientemente conectado, el *host* obtiene información acerca del dispositivo tal como número y tipo de *endpoints*, clase de dispositivo, Vendor ID y Product ID entre otros. Es también en la etapa de enumeración en la que se asigna al dispositivo una dirección fija con la cual será accedido a partir de ese momento y hasta que se desconecte, en la próxima conexión esta dirección puede cambiar. El proceso de enumeración consta de los siguientes pasos:

1. El *host* detecta la conexión de un nuevo dispositivo mediante las resistencias pull-up conectadas a las líneas D+ o D- del dispositivo. Se tiene un tiempo de espera de 100ms antes de continuar, esto es para asegurar que el dispositivo se conecte correctamente y que el voltaje de alimentación del mismo sea estable.

2. El *host* pone el bus en *reset,* esto pone al dispositivo en el estado *default*. El dispositivo responde a todas las transferencias que se hacen hacia la dirección 0.
3. El *host* emite una petición *Get_Descriptor* para obtener el tamaño máximo de datos que pueden enviarse en una transferencia a través del *endpoint 0*.
4. Se pone nuevamente en *reset* al bus.
5. El *host* emite una petición *Set_Address*, para establecer la dirección del dispositivo. Esto pone al dispositivo en el estado *addressed*.
6. El *host* emite una petición *Get_Descriptor/Device*.
7. El *host* emite una petición *Get_Descriptor/Configuration* para obtener los primeros 9 bytes y determinar el tamaño completo del descriptor.
8. El *host* emite una petición *Get_Descriptor/Configuration* para obtener el descriptor completo.
9. Se hace la petición de las cadenas de caracteres (*strings*) que describen al dispositivo. Después de esto, si el dispositivo necesita de un controlador especial el sistema se encargara de solicitarlo, si no es el caso el dispositivo está listo para utilizarse.

2.9 Descriptores

Todos los dispositivos USB hacen uso de descriptores para informar al *host* acerca de sus capacidades y atributos. Los descriptores son tablas de datos que contienen información acerca del dispositivo, el primer campo de un descriptor es siempre un byte que representa la longitud del descriptor completo. Cada dispositivo cuenta con varios descriptores de los cuales se obtiene la información necesaria para el proceso de enumeración. Estos descriptores son *Device Descriptor*, *Device Qualifier Descriptor*, *Configuration Descriptor*, *Other speed Configuration Descriptor, Interface Descriptor*, *Endpoint Descriptor* y *String Descriptor*.

Device Descriptor. Contiene la información general del dispositivo. Solo se permite un descriptor de este tipo para cada dispositivo.

Tabla 2.5 Device descriptor.

Byte	Campo	Tamaño (Byte)	Valor	Descripción
0	*bLength*	1	Numérico	Tamaño del descriptor en bytes.
1	*bDescriptorType*	1	Constante	Descriptor de tipo *Device*.
2	*bcdUSB*	2	BCD	Especificación USB con la que se cumple. Valor expresado en BCD. Por ejemplo, 0x0100 corresponde a la especificación 1.0.
4	*bDeviceClass*	1	Clase	Código de la clase del dispositivo.
5	*bDeviceSubClass*	1	Subclase	Código de subclase.
6	*bDeviceProtocol*	1	Protocolo	Código de protocolo. El protocolo utilizado es dependiente de la Clase del dispositivo.
7	*bMaxPacketSize0*	1	Numérico	Tamaño máximo de paquete para el *endpoint0* (*control endpoint*). Solo son válidos los valores 8, 16, 32 y 64.
8	*idVendor*	2	Identificador	*Vendor ID*. Es el identificador del fabricante del dispositivo.

Tabla 2.5 (continuación) Device descriptor.

Byte	Campo	Tamaño (Byte)	Valor	Descripción
10	*idProduct*	2	Identificador	*Product ID*. Es el identificador de producto.
12	*bcdDevice*	2	BCD	Número de versión del dispositivo.
14	*iManufacturer*	1	Índice	Índice a la cadena de caracteres que contiene el nombre del fabricante.
15	*iProduct*	1	Índice	Índice a la cadena de caracteres que contiene el nombre del producto.
16	*iSerialNumber*	1	Índice	Índice a la cadena de caracteres que contiene el número de serie del dispositivo
17	*bNumConfigurations*	1	Numérico	Número de posibles configuraciones

Datos como *VendorID* y *ClassCode* (clase del dispositivo) son proporcionadas por una institución llamada USB-IF (USB Implementers Forum) que también se encarga de dar soporte y promover tecnologías derivadas de USB tales como USB wireless.

Device Qualifier Descriptor. Este descriptor contiene de información acerca del comportamiento del dispositivo en caso de que éste funcionara a una velocidad diferente a la actual. Por ejemplo, si se tiene un dispositivo funcionando en *full speed* pero es capaz de funcionar a *high speed*, este descriptor contiene la información de los atributos del dispositivo funcionando en *high speed*.

Tabla 2.6 Device qualifier descriptor.

Byte	*Campo*	Tamaño (Byte)	Valor	Descripción.
0	*bLength*	1	Numérico	Tamaño del descriptor.
1	*bDescriptorType*	1	Constante	Tipo de descriptor *Device Qualifier*.
2	*bcdUSB*	2	BCD	Versión de la especificación USB.
4	*bDeviceClass*	1	Clase	Código de la Clase.
5	*bDeviceSubClass*	1	Clase	Código de la Subclase.
6	*bDeviceProtocol*	1	Numérico	Código del protocolo.
7	*bMaxPacketSize*	1	Numérico	Tamaño máximo de paquete para otra velocidad.
8	*bNumConfigurations*	1	Numérico	Número de configuraciones posibles para otra velocidad.
9	*bReserved*	1	-	Reservado para uso futuro.

Configuration Descriptor. Contiene información acerca de una configuración. Existe un descriptor de este tipo por cada configuración posible del dispositivo.

Tabla 2.7 Configuration descriptor

Byte	Campo	Tamaño	Valor	Descripción
0	*bLength*	1	Numérico	Tamaño del descriptor.
1	*bDescriptorType*	1	Constante	Tipo de descriptor Configuration
2	*wTotalLength*	2	Numérico	Longitud total. Esto incluye la longitud todos los descriptores que se utilizan en conjunto con esta configuración (*Configuration*, *Interface*, *Endpoint*).
4	*bNumInterfaces*	1	Numérico	Número de interfases que incluye esta configuración.
5	*bConfigurationValue*	1	Numérico	Valor que debe usarse como parámetro de la petición *Set_Configuration* para seleccionar esta configuración.
6	*iConfiguration*	1	Índice	Índice de la cadena de caracteres que contiene el nombre de esta configuración.
7	*bmAtributtes*	1	Bits	D7: Reservado (poner a '1'). D6: Auto alimentado (self-powered). D5: Remote wake up. D4 – D0: Reservados (poner a '0').
8	*bMaxPower*	1	mA	Máximo consumo de corriente. Valor expresado en unidades de 2 mA, por ejemplo 50 equivale a 100 mA.

Other speed Configuration Descriptor. Contiene la descripción de la configuración de un dispositivo USB capaz de operar en *high speed* como si éste estuviera operando a otra velocidad.

Tabla 2.8 Other speed configuration descriptor.

Byte	Campo	Tamaño	Valor	Descripción
0	*bLength*	1	Numérico	Tamaño del descriptor.
1	*bDescriptorType*	1	Constante	Tipo de descriptor *Configuration*
2	*wTotalLength*	2	Numérico	Longitud total. Esto incluye la longitud todos los descriptores que se utilizan en conjunto con esta configuración (*Configuration*, *Interface*, *Endpoint*).
4	*bNumInterfaces*	1	Numérico	Número de interfases que incluye esta configuración operando a esta velocidad.
5	*bConfigurationValue*	1	Numérico	Valor que debe usarse como parámetro de la petición *Set_Configuration* para seleccionar esta configuración.
6	*iConfiguration*	1	Índice	Índice de la cadena de caracteres que contiene el nombre de esta configuración.
7	*bmAtributtes*	1	Bits	Igual que en el descriptor *Configuration Descriptor*.
8	*bMaxPower*	1	mA	Igual que en el descriptor *Configuration Descriptor*.

Interface Descriptor. Describe una interfase específica dentro de una configuración. Se utilizan las peticiones *Get_Interface* y *Set_Interface* para obtener información acerca de la interfase activa o para establecer otra como activa.

Tabla 2.9 Interface descriptor.

Byte	Campo	Tamaño	Valor	Descripción
0	*bLength*	1	Numérico	Tamaño del descriptor
1	*bDescriptorType*	1	Constante	Descriptor tipo *Interface*
2	*bInterfaceNumber*	1	Numérico	Número de interfase. Las interfases posibles dentro de una configuración se enumeran a partir de 0.
3	*bAlternateSetting*	1	Numérico	Valor usado para seleccionar características alternativas para la interfase seleccionada con *bInterfaceNumber*.
4	*bNumEndpoints*	1	Numérico	Número de *endpoints* utilizados por esta interfase. Si el valor es '0', la interfase solo utiliza el *endpoint* de control (*endpoint 0*).
5	*bInterfaceClass*	1	Clase	Código de la Clase.
6	*bInterfaceSubClass*	1	SubClase	Código de la Subclase
7	*bInterfaceProtocol*	1	Protocolo	Código del Protocolo
8	*iInterface*	1	Índice	Índice de la cadena de caracteres que describe esta interfase.

Endpoint Descriptor. Existe un descriptor de este tipo por cada *endpoint* dentro de una interfase, a excepción del *endpoint 0* que no tiene un descriptor. Contiene la información acerca del tipo y tamaño del *endpoint* entre otras.

Tabla 2.10 Endpoint descriptor.

Byte	Campo	Tamaño	Valor	Descripción
0	*bLength*	1	Numérico	Tamaño del descriptor
1	*bDescriptorType*	1	Constante	Descriptor tipo *Endpoint*
2	*bEndPointAddress*	1	Endpoint	La dirección del *endpoint* en el dispositivo USB. Esta dirección está organizada así: Bit 3 – 0: Número de *endpoint*. Bit 6 – 4: Reservados (poner a '0'). Bit 7: Dirección de las transferencias. Ignorado para *endpoints* de control. 0 = Endpoint de salida. 1 = Endpoint de entrada.
3	*bmAttributes*	1	Bits	Descripción de los atributos del endpoint. Bits 1 – 0: Tipo de transferencias 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt Bits 3 – 2: Tipo de sincronización (solo para *endpoints* tipo *Isochronous*). 00 = No sincronización 01 = Asíncrono 10 = Adaptivo 11 = Síncrono Bits 5 – 4: Uso 00 = *Endpoint* de datos 01 = *Endpoint* de retroalimentación 10 = *Endponit* de retroalimentación implícita de datos 11 = Reservado El resto de los bits son reservados, deben ser puestos a '0'.

Tabla 2.10 (continuación) Endpoint descriptor.

Byte	Campo	Tamaño	Valor	Descripción
4	*wMaxPacketSize*	2	Numérico	Es el tamaño en bytes que el *endpoint* tiene capacidad de transferir. Para todos los endpoints, los bits 10 a 0 especifican el tamaño. Para endpoints tipo *Isochronous* e *Interrupt* en modo *high speed* los bits 12 y 11 especifican el número de oportunidades de transacción adicionales por *microframe*. 00 = Ninguna (solo 1 transacción por *microframe*) 01 = 1 transacción adicional (2 por *microframe*) 10 = 2 transacciones adicionales (2 por *microframe*) 11 = Reservado Los bits 15 a 13 son reservados y deben ser puestos a '0'.
6	*bInterval*	1	Numérico	Intervalo de encuesta para transferencia de datos.

String Descriptor. Este tipo de descriptor es adicional. Contienen una descripción del dispositivo que es visible para el usuario en forma de texto.

El uso de los descriptores es de mucha ayuda para hacer del USB un bus inteligente y de fácil uso pues el usuario solo requiere conectar los dispositivos a la PC, una vez hecho esto la información de descriptores y las demás partes del sistema USB determinan el modo en que el dispositivo operará.

Capítulo 3 Memoria SDRAM.

3.1 Introducción

A diferencia de las memorias SRAM (Static Random Access Memory), en las que la interfase es asíncrona y por tanto los comandos se registran con el cambio de alguna de las señales de control, las memorias SDRAM (Synchronous Dynamic Random Access Memory) tienen una interfase síncrona en la que los comandos se registran con el cambio de una señal de reloj, es decir, el comando a ejecutar es aquel que está presente en las líneas de control cuando la señal de reloj presenta una transición, por lo general la transición positiva. Ésta es una de las diferencias que existen entre estos dos tipos de memoria; la tabla 3.1 presenta algunas de las características más importantes de estas memorias y posteriormente se justifica la elección de una memoria SDRAM para la implementación del sistema de almacenamiento desarrollado en el presente trabajo.

Tabla 3.1 Comparación de las memorias SRAM y SDRAM.

SRAM (Static Random Access Memory)	SDRAM (Synchronous Dynamic Random Access Memory)
Memoria volátil	Memoria volátil
No utiliza interfase síncrona	Utiliza interfase síncrona
Latencia no programable	Latencia programable
Líneas de direcciones no multiplexadas	Líneas de direcciones multiplexadas
No necesita refresco	Necesita refresco
Densidades de hasta 64Mb	Densidades de hasta 512Mb
Estructura de la celda de 4 a 6 transistores	Estructura de la celda de 1 transistor y 1 capacitor
Tiempo de acceso de hasta 8ns	*Frecuencia de trabajo de hasta 133Mhz (tiempo de acceso 5.4ns)*
Organizada en un solo banco	Organizada en 2 ó 4 bancos
Acceso a una sola localidad por operación	Acceso tipo ráfaga de 1, 2, 4 y 8 localidades

En la interfase síncrona de una memoria SDRAM la señal de reloj es también utilizada para manejar una máquina de estados finitos interna, la cual administra y temporiza las operaciones que se solicitan a la memoria, de tal forma que se pueda aceptar una nueva operación aún cuando no se haya terminado de procesar la anterior, a esta característica se le llama *pipelining*, la cual no poseen otras tecnologías de memoria como la RAM estática. Algunas de ventajas que presenta la memoria SDRAM ante las típicas SRAM y razones por las que se eligió una memoria SDRAM para la implementación del sistema de almacenamiento son:

- Debido a que las líneas de dirección son multiplexadas, no se necesitan muchas líneas para direccionar un gran número de localidades.
- La interfase síncrona permite realizar operaciones a la frecuencia de reloj a la que trabaja el sistema completo.
- La organización en varios bancos es muy útil para la implementación de dos memorias independientes, en este caso una de escritura (ME) y una de lectura (ML).
- Las densidades disponibles en SDRAM son de un rango mayor a las de una SRAM, lo que da al diseño la posibilidad de expandir la capacidad del sistema de almacenamiento.

Aunque también existen desventajas en el uso de SDRAM, tales como la necesidad de implementar la atención a peticiones de refresco y la mayor complejidad que tiene la interfase comparada con la de una SRAM, el uso de esta tipo de memoria permite explorar nuevas tecnologías. Cabe mencionar que la memoria SDRAM es utilizada en sistemas complejos, por ejemplo en la memoria RAM principal de computadoras personales y otros sistemas.

La implementación de un controlador de SDRAM es una tarea complicada y es necesario el conocimiento previo de las características principales de este tipo de memoria por lo que a continuación se presentan los conceptos necesarios para poder implementar un controlador de acceso a memoria SDRAM.

3.2 Organización de la SDRAM

Las memorias SDRAM están organizadas internamente en bancos, los cuales a su vez se dividen en filas y columnas (figura 3.1), de tal forma que para direccionar una localidad de la memoria es necesario especificar el banco, la fila y la columna.

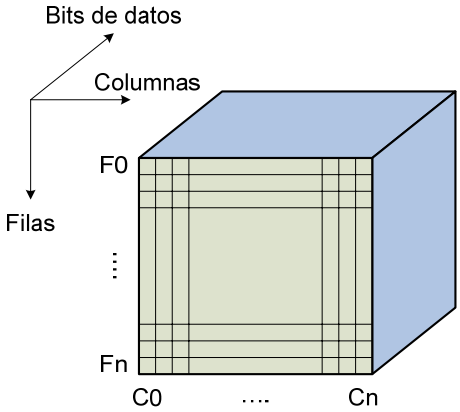


Figura 3.1 Organización en bancos.

Las escrituras y lecturas pueden ser configuradas como modo ráfaga (*Burst*) en el que las operaciones se realizan con el acceso a una determinada localidad de memoria y posteriormente se realiza un número programado de accesos de forma automática (1, 2, 4 u 8) a las localidades subsecuentes en un orden también programado. El acceso a la memoria inicia con la ejecución de un comando ACTIVE seguido ya sea de un comando WRITE o READ según la operación deseada. Para direccionar la localidad objetivo de la operación se utilizan las líneas de direcciones A[12:0], el valor de estas líneas al momento de ejecutar el comando ACTIVE determina la fila, el estado de las líneas BA0 y BA1 en este mismo instante determina el banco. La columna se selecciona mediante las mismas líneas que la fila pero al momento de ejecutar ya sea un comando READ o WRITE. En caso de operación en modo ráfaga, coincidente con el comando de READ o WRITE solo se proporciona la dirección inicial de columna, las direcciones de columna subsecuentes son calculadas automáticamente por la lógica interna de la memoria.

Dependiendo de la capacidad de la memoria, el número de filas y columnas en un banco puede variar, por ejemplo la memoria utilizada en este trabajo (MT48LC16M16A2 – 4 Meg x 16 x 4 banks) tiene una capacidad de 256 Mb (Megabits) organizados en cuatro bancos de 8192 filas, 512 columnas y el tamaño del dato es de 16 bits. Así 4 x 8192 x 512 x 16 = 268435456, lo que es equivalente a 256 Mb. A este tipo de memoria también se le llama SDR SDRAM (SDR, Single Data Rate. La transferencia de datos se efectúa solo en una de las transiciones de la señal de reloj).

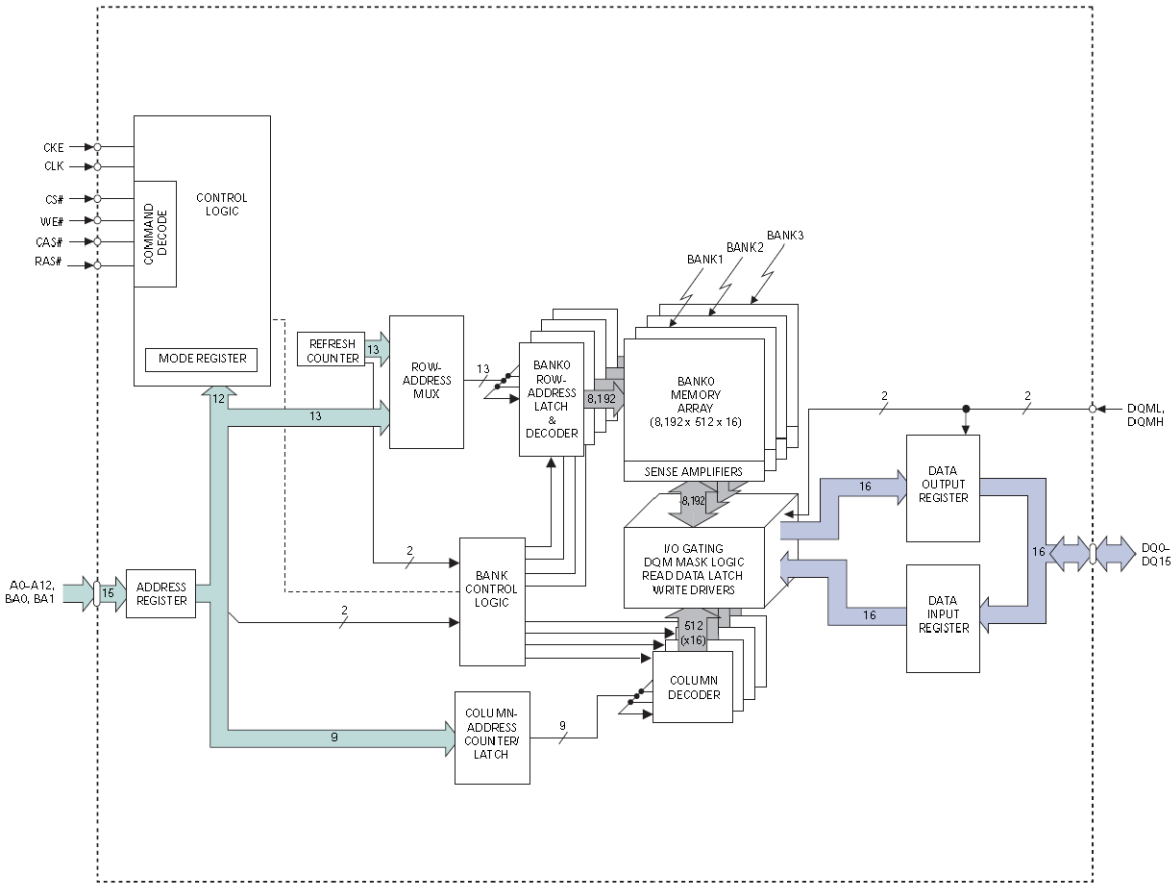


Figura 3.2 Diagrama de bloques de la memoria SDRAM.

3.3 Inicialización

Antes de que la memoria sea accedida ya sea para escribir en, o leer de ella es necesario ejecutar una rutina de inicialización. Ésta incluye la configuración del modo de operación de la memoria; tipo y longitud de *burst*, número de ciclos de latencia para operaciones de lectura. El proceso de inicialización es el siguiente:

1. Poner la línea CKE (clock enable) a nivel bajo.
2. Aplicar señal estable de reloj.
3. Esperar al menos 100 µs.
4. Poner a CKE en alto y ejecutar al menos un comando NOP o DESELECT.
5. Ejecutar comando PRECHARGE ALL (comando PRECHARGE con la línea de direcciones A10 en alto).
6. Esperar al menos un tiempo t_{RP} (20 ns, PRECHARGE command period). Durante este tiempo ejecutar comandos NOP o DESELECT.
7. Ejecutar un comando AUTO REFRESH.
8. Esperar al menos un tiempo t_{RFC} (66ns, AUTO REFRESH command period). Durante este tiempo ejecutar comandos NOP o DESELECT.
9. Ejecutar un comando AUTO REFRESH.
10. Esperar al menos un tiempo t_{RFC} (66ns, AUTO REFRESH command period). Durante este tiempo ejecutar comandos NOP o DESELECT.
11. En este punto la SDRAM está lista para que su registro de modo sea programado. Aplicar un comando LMR (Load Mode Register) con las opciones adecuadas al modo deseado.
12. Esperar al menos un tiempo t_{MRD} (2 t_{CK}, tiempo después de ejecutar LMR y antes de ejecutar AUTOREFRESH o ACTIVE). Durante este tiempo ejecutar comandos NOP o DESELECT.
13. La memoria SDRAM está lista para recibir y ejecutar cualquier comando válido.

3.4 Registro de modo

El registro de modo es utilizado para definir un modo específico de operación de la SDRAM, esto incluye la selección de la longitud y el tipo de la ráfaga de acceso (burst), el número de ciclos de latencia (CAS latency), es decir, el número de ciclos que se demorará un dato válido para estar disponible en el bus después haber aplicado un comando de lectura. Este registro se programa mediante el comando LOAD MODE REGISTER (LMR) y utilizando las líneas de direcciones A[12:0]. Todos los bancos de la memoria deben estar libres en el momento de la configuración del registro y el controlador debe esperar los tiempos necesarios antes de aplicar cualquier comando válido. Omitir la secuencia de inicialización y la configuración de registro de modo resultará en un funcionamiento indefinido y erróneo de la memoria SDRAM.

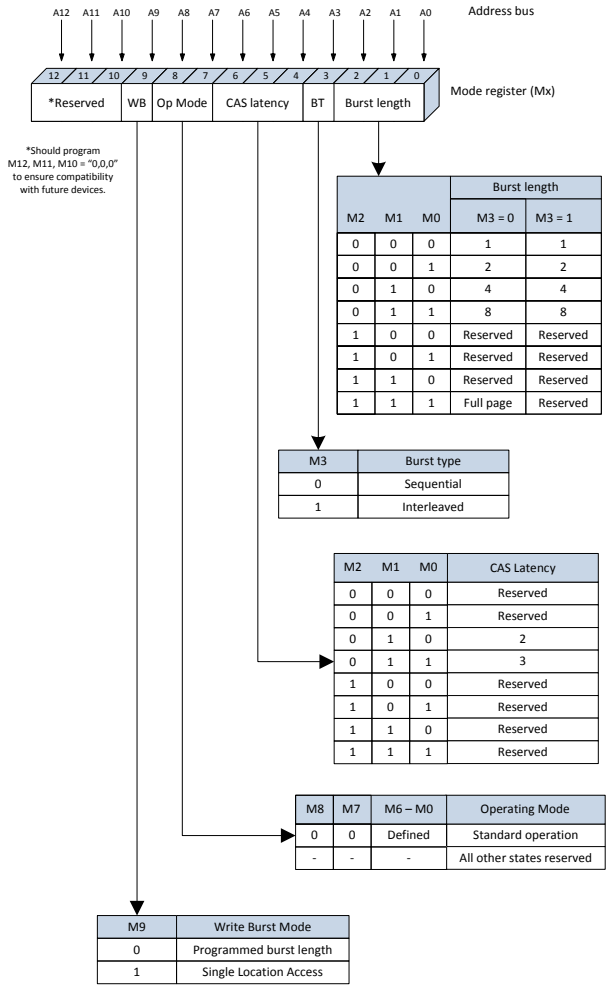


Figura 3.3 El registro de modo y sus diferentes opciones de configuración.

3.4.1 Longitud de la ráfaga (*Burst Length*)

Puesto que las operaciones de escritura y lectura a la memoria SDRAM son de tipo ráfaga, la longitud de esta ráfaga debe ser programada en el registro de modo utilizando los bits M[2:0]. La longitud determina el número máximo de localidades dentro de una fila que puede ser accedido por cada comando READ o WRITE, puede tomar los valores de 1, 2, 4 y 8.

3.4.2 Modo de operación

Los bits M8 y M7 puestos en cero seleccionan el modo de operación normal, de hecho cualquier otra combinación no es válida y está reservada para usos y compatibilidad con futuras versiones de hardware y estándares relacionados con las memorias SDRAM. En el modo normal la longitud de ráfaga aplica tanto para escrituras como para lecturas.

3.4.3 Modo de la ráfaga de escritura

Cuando el bit M9 = '0', la longitud de ráfaga programada mediante M[2:0] aplica para operaciones de lectura y escritura. Si M9 = '1', la longitud de ráfaga solo aplica a operaciones de lectura mientras que las operaciones de escritura son realizadas en una localidad a la vez.

3.4.4 Tipo de ráfaga

El orden de los accesos a la memoria dentro de una ráfaga puede ser programado de dos formas, ya sea secuencial o intercalado. La tabla 3.1 los dos tipos de ráfaga que pueden ser utilizados y cual es seleccionado por medio del bit M3.

Tabla 3.2 Longitud y tipo de ráfaga.

Longitud de ráfaga	Dirección inicial de columna			Orden de acceso dentro de una ráfaga	
				Tipo = Secuencial	Tipo = Intercalado
2			A0		
			0	0-1	0-1
			1	1-0	1-0
4		A1	A0		
		0	0	0-1-2-3	0-1-2-3
		0	1	1-2-3-0	1-0-3-2
		1	0	2-3-0-1	2-3-0-1
		1	1	3-0-1-2	3-2-1-0
8	A2	A1	A0		
	0	0	0	0-1-2-3-4-5-6-7	0-1-2-3-4-5-6-7
	0	0	1	1-2-3-4-5-6-7-0	1-0-3-2-5-4-7-6
	0	1	0	2-3-4-5-6-7-0-1	2-3-0-1-6-7-4-5
	0	1	1	3-4-5-6-7-0-1-2	3-2-1-0-7-6-5-4
	1	0	0	4-5-6-7-0-1-2-3	4-5-6-7-0-1-2-3
	1	0	1	5-6-7-0-1-2-3-4	5-4-7-6-1-0-3-2
	1	1	0	6-7-0-1-2-3-4-5	6-7-4-5-2-3-0-1
	1	1	1	7-0-1-2-3-4-5-6	7-6-5-4-3-21-0

3.4.5 Latencia (CAS Latency, CL)

CL es el retardo, expresado en ciclos de reloj, entre el instante en que se emite un comando READ y el instante en que el primer dato aparece disponible en el bus. Utilizando los bits M[6:4] la latencia puede programarse para 2 ó 3 ciclos de reloj.

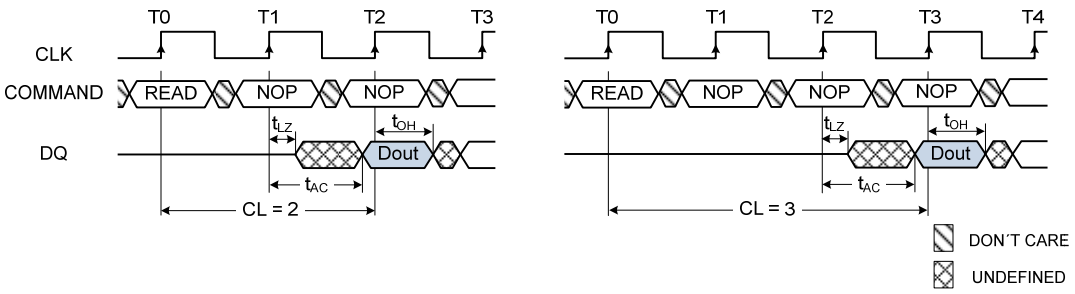


Figura 3.4 Latencia (CAS latency) de 2 y 3 ciclos.

3.5 Comandos

La siguiente tabla muestra los comandos que pueden ser enviados a la memoria utilizando las líneas CS, RAS, CAS y WE, la descripción cada comando se presenta posteriormente.

Tabla 3.3 Comandos válidos para la memoria SDRAM.

Comando	CS	RAS	CAS	WE	DQM	ADDR
COMMAND INHIBIT (NOP)	H	X	X	X	X	X
NO OPERATION (NOP)	L	H	H	H	X	X
ACTIVE (Selección de banco y fila activa)	L	L	H	H	X	Banco / Fila
READ(Selección de banco y columna activa, inicia la ráfaga de lectura)	L	H	L	H	L/H	Banco / Columna
WRITE(Selección de banco y columna activa, inicia la ráfaga de escritura)	L	H	L	L	L/H	Banco / Columna
BURST TERMINATE	L	H	H	L	X	X
PRECHARGE (Desactiva la fila actualmente activa en uno o todos los bancos)	L	L	H	L	X	Código
AUTOREFRESH	L	L	L	H	X	X
LOAD MODE REGISTER	L	L	L	L	X	Código de operación

3.5.1 COMMAND INHIBIT

Evita que cualquier comando nuevo sea ejecutado por la SDRAM independientemente de si la señal CLK está habilitada o no. El chip de memoria es deseleccionado, inhibiendo cualquier nueva operación sin afectar las operaciones en progreso.

3.5.2 NO OPERATION (NOP)

Este comando no realiza operación alguna en la memoria, únicamente es utilizado para prevenir la ejecución de comandos no deseados que pudieran registrarse mientras la memoria se encuentra en estados inactivos o de espera. Las operaciones en progreso no son afectadas por este comando.

3.5.3 LOAD MODE REGISTER

Este comando es utilizado para programar el registro de modo mediante un código de operación enviado a través de las líneas de direcciones A[12:0]. La programación del registro de modo solo puede efectuarse cuando todos los bancos de la memoria están inactivos, y cualquier comando subsecuente no puede ser ejecutado sino hasta que se haya cumplido el periodo t_{MRD} (2 ciclos de reloj).

3.5.4 ACTIVE

El comando ACTIVE es usado para "abrir" o activar una fila dentro de un banco en particular para su posterior acceso. El valor de las líneas BA0 y BA1 que se registra coincidente con el comando ACTIVE selecciona el banco y las líneas A[12:0] seleccionan la fila. La fila "abierta" permanece activa y puede ser accedida mientras no se aplique un comando PRECHARGE.

3.5.5 READ

Este comando es utilizado para iniciar una ráfaga de lectura en la fila activada mediante un comando ACTIVE previo. Las líneas BA0 y BA1 seleccionan el banco y las líneas A[8:0] seleccionan dirección de la columna inicial de acceso. La línea A10 determina si la opción AUTOPRECHARGE está habilitada o no. Si esta opción está habilitada, la fila que está siendo accedida será desactivada al finalizar la operación de lectura; si no está habilitada, la fila permanecerá activa y puede ser accedida para operaciones posteriores. Los datos leídos aparecen en las líneas de datos DQ dependiendo del nivel lógico que las entradas de enmascaramiento DQM presentaban dos ciclos antes de la ejecución del comando READ. Si estas entradas estaban en un nivel alto, las líneas de datos DQ serán puestas en alta impedancia dos ciclos después; si estaban en un nivel bajo, en las líneas de DQ aparecerán los datos leídos.

3.5.6 WRITE

Este comando es utilizado para iniciar una ráfaga de escritura en la fila activada mediante un comando ACTIVE previo. Las líneas BA0 y BA1 seleccionan el banco y las líneas A[8:0] seleccionan dirección de la columna inicial de acceso. La línea A10 determina si la opción AUTOPRECHARGE está habilitada o no. Los datos puestos en las líneas DQ son escritos en la memoria dependiendo del nivel lógico de DQM. Si las entradas DQM están en bajo, los datos son escritos en la memoria; de lo contrario los datos son ignorados y no se escriben en la memoria.

3.5.7 PRECHARGE

La acción de este comando es la de desactivar una fila que previamente ha sido activada, ya sea en un banco en particular o en todos los bancos. Los bancos estarán disponibles para acceder a cualquier fila solo después de que haya transcurrido el periodo t_{RP} después de la ejecución del comando PRECHARGE. La línea A10 determina si la fila se desactiva en uno o todos los bancos; si es solo en un banco (A10 = ‘0’), éste se selecciona mediante BA0 y BA1. El valor de las líneas BA0 y BA1 no importa cuando la operación es en todos los bancos (A10 = ‘1’). Después de la ejecución de PRECHARGE los bancos o el banco seleccionado están inactivos y antes de realizar cualquier operación de lectura o escritura deben ser activados mediante un comando ACTIVE.

3.5.8 BURST TERMINATE

Usado para interrumpir cualquier ráfaga de acceso ya sea de escritura o lectura. Cualquier comando READ o WRITE registrado previo a la ejecución de BURST TERMINATE es interrumpido, la fila a la cual se tenía acceso permanece activa hasta que se ejecute un comando PRECHARGE.

3.5.9 AUTO REFRESH

Para mantener la integridad de los datos almacenados en la memoria es necesario refrescar cada una de las localidades que integran la memoria. Para esto se utiliza el comando AUTOREFRESH, básicamente el proceso de refresco consiste en leer una localidad de memoria y escribir nuevamente en ella para evitar que los niveles de voltaje almacenado en cada celda de la memoria decaigan hasta un nivel en que la información ya no sea válida. En cada ciclo del comando AUTOREFRESH se refresca toda una fila de la memoria. Para ejecutar este comando es necesario que todos los bancos sean desactivados previamente, mediante PRECHARGE, y esperar el periodo t_{RP} (20ns mínimo) posterior a la ejecución de PRECHARGE. La dirección de la fila que se refresca no depende de las líneas de direcciones A[12:0] ya que la memoria cuenta con un controlador interno de refresco que genera automáticamente las direcciones de fila. La memoria SDRAM de 256Mb requiere de 8192 ciclos de AUTOREFRESH cada 64 ms, los cuales pueden distribuirse en todo este periodo (1 cada 7.81µs) o pueden ejecutarse como una ráfaga de 8192 comandos AUTOREFRESH cada 64ms.

3.6 Operaciones de lectura y escritura

Aunque las operaciones de escritura y lectura pueden realizarse en modo ráfaga, en el presente trabajo estas operaciones se realizan una a la vez (longitud de burst = 1), ya que los requerimientos de velocidad de transferencia se satisfacen de forma adecuada utilizando éste modo. A continuación se presentan los diagramas de tiempo correspondientes a cada una de las operaciones y se da una explicación del proceso que se realiza. Cualquiera de las dos operaciones debe iniciarse con un comando ACTIVE para seleccionar alguna fila dentro del banco deseado y posteriormente ejecutando ya sea un comando WRITE o READ según se desee. Los tiempos entre cada comando son visibles en los diagramas. Ambas operaciones se realizan sin la opción AUTOPRECHARGE activada.

Escritura

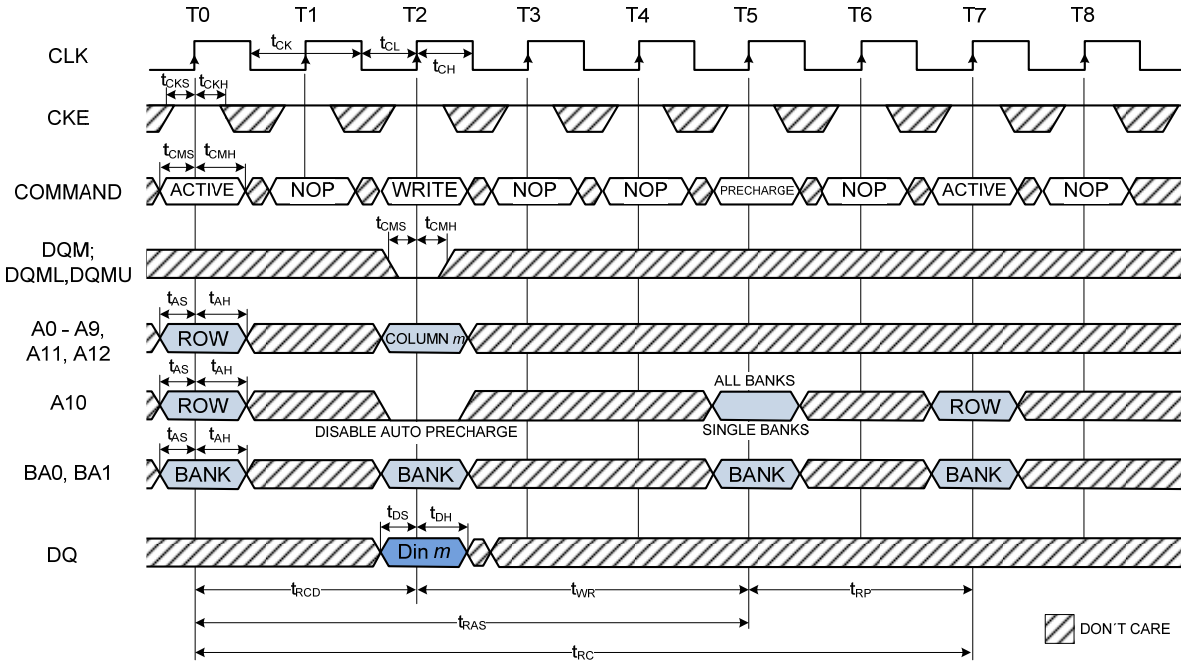


Figura 3.5 Forma de onda para escritura.

En el tiempo T0, se registra un comando ACTIVE y mediante las líneas A[12:0] y BA[1:0] se selecciona el banco y la fila. Posteriormente, antes de emitir el comando WRITE es necesario esperar el periodo t_{RCD} (20ns mínimo) el cual puede ser omitido si el periodo de la señal CLK es mayor que 20ns, lo que permite que la instrucción WRITE se ejecute en la siguiente transición positiva de CLK sin necesidad de un estado de espera. En el momento de emitir el comando WRITE también se selecciona el banco y la columna y en el bus de datos se pone el dato que se desea escribir. Una vez emitido el comando WRITE es necesario esperar el periodo t_{WR} (1 t_{CK} + 7.5ns mínimo) antes de emitir el siguiente comando válido, en este caso PRECHARGE. Al mismo tiempo que se ejecuta el comando PRECHARGE se pone la línea A10 en alto para así deseleccionar la fila actual en todos los bancos. Una vez que haya finalizado el periodo tRP (20ns mínimo) después de la ejecución de PRECHARGE, la memoria está lista para otra operación en cualquier localidad de cualquier banco.

Lectura

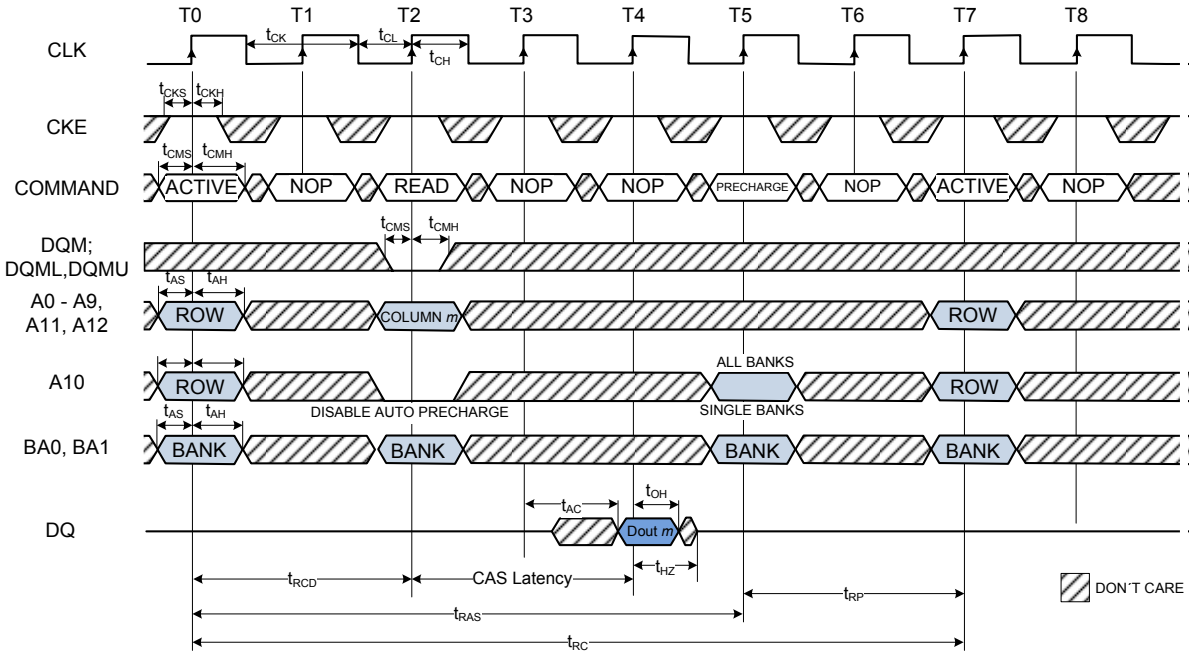


Figura 3.6 Forma de onda para lectura.

Al igual que las operaciones de escritura, las operaciones de lectura se inician con un comando ACTIVE con el que simultáneamente se selecciona la fila y el banco que se desea acceder. El periodo t_{RCD} puede o no ser necesario dependiendo de la duración del periodo t_{CK}. La ejecución del comando READ, posterior al comando ACTIVE, va acompañada de la selección del banco y la columna que se desea acceder. Una vez que sea emitido el comando READ es necesario esperar 2 o 3 ciclos (CL, programado en el registro de modo) antes que el dato que se leyó aparezca en las líneas de datos, la lógica externa puede tomar este dato a partir de este momento. Una vez leído el dato, es necesario emitir un comando PRECHARGE para desactivar la fila seleccionada en todos los bancos y que la memoria esté lista para una nueva operación terminado el periodo t_{RP} posterior a la ejecución de PRECHARGE.

Capítulo 4 Implementación de la comunicación USB.

4.1 El controlador EZ-USB

El EZ-USB FX2LP (CY7C68013A/14A/15A/16A) es un controlador USB fabricado por Cypress® el cual es utilizado para proporcionar conectividad USB a dispositivos que así lo requieran, se trata de un controlador de dispositivo (*device controller*) y no de un controlador de *host* (*host controller*) que proporciona soporte para las especificaciones USB 1.1 y USB 2.0, modos *full speed* y *high speed* respectivamente.

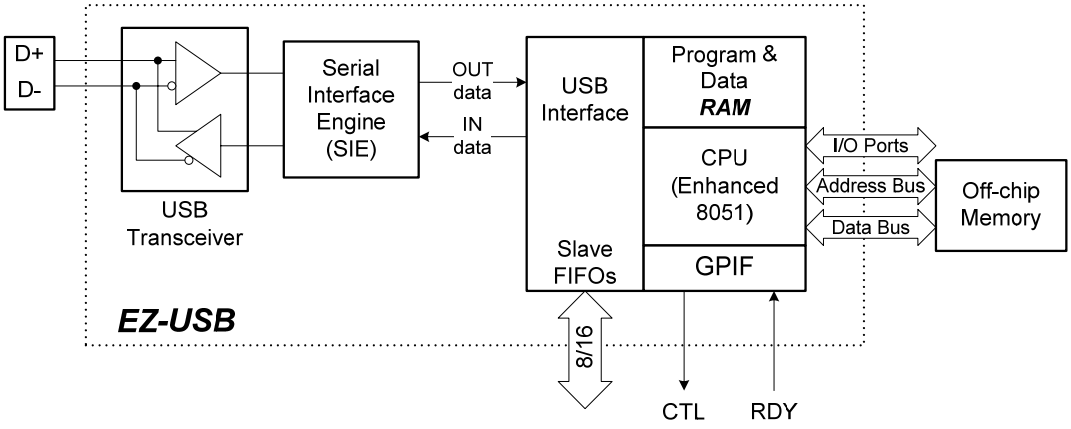


Figura 4.1 Diagrama de bloques del controlador EZ-USB.

La arquitectura interna del chip consta de varias partes las cuales se muestran en la figura 4.1 del diagrama de bloques del chip.

Se cuenta con un transceptor (*transceiver*) al cual están conectadas las líneas D+ y D– del bus, para el EZ-USB FX2LP se tiene un transceptor que trabaja a 480 Mbits/sec. Para el EZ-USB FX1 el transceptor opera a 12Mbits/sec. Ambos dispositivos integran un PLL y el SIE, la capa física completa del protocolo USB. El transceptor conjuntamente con el SIE (*Serial Interface Engine*) se encarga de atender tareas específicas del bus tales como codificación y decodificación de datos serie, identificación de paquetes, *bit stuffing* y demás tareas a nivel de señales eléctricas en el bus. El SIE entrega y recibe datos en forma

paralela hacia y desde el bloque de interfase USB (*USB Interface*). La sección llamada FIFO's esclavos (*slave FIFO's*) integran la memoria FIFO que funciona como *endpoints* para el dispositivo, podrían verse como dos dominios distintos; uno referente a los *endpoints* y otro a los FIFO's esclavos que pueden conectarse con lógica externa específica o un microprocesador, sin embargo se unifican para evitar transferencias internas y así reducir el tiempo de transferencia. Los FIFO's pueden configurarse con doble, triple o cuádruple buffer.

El acceso a los FIFO's esclavos puede hacerse mediante un maestro externo que temporiza las transacciones y genera las señales necesarias para escrituras y lecturas o bien mediante un maestro interno llamado GPIF (General Programable Interface, hardware propietario Cypress®) el cual, mediante su adecuada programación, genera también las señales necesarias para la transferencia de datos utilizando ya sea un oscilador interno de 30 ó 48 MHz o uno externo de 5-48 MHz , el GPIF tiene la facilidad también de generar estados de espera, utilizando las señales RDY, para iniciar, reanudar o finalizar una transferencia.

El CPU es un 8051 mejorado que integra memoria interna de datos y de programa, en un dispositivo USB basado en el EZ-USB la funcionalidad de este CPU es:

- Implementación del protocolo USB a alto nivel atendiendo las peticiones que el *host* hace a través del *endpoint0* (*endpoint* de control).
- Está disponible para uso de propósito general en el sistema.
- En aplicaciones en las que se requiere una velocidad alta de transferencia, el CPU comúnmente está fuera de la ruta de datos, solo participa para configurar el modo de las transferencias y posteriormente ya no tiene ninguna intervención en éstas. En algunas aplicaciones en las que se requiere tratar los datos antes de ser transferidos, el CPU juega un papel importante pues es el que se encarga de este tratamiento de los datos.
- El CPU cuenta hasta con cinco puertos de entrada/salida, dos USARTs, comunicación I2C, tres contadores/temporizadores y un extensivo conjunto de

fuentes de interrupción. Opera una velocidad de hasta 48 MHz y cada instrucción se ejecuta en cuatro ciclos de reloj en contraste con los doce ciclos que utiliza un 8051 estándar. El SIE y la interfase USB implementada en el EZ-USB simplifican la escritura de código ya que gran parte del protocolo USB es manejado por éstos. De hecho el chip puede actuar como un dispositivo USB con todas sus características aún sin tener *firmware* cargado en la memoria interna.

4.1.1 Organización de memoria

La organización de memoria del EZ-USB es similar a la del 8051, la memoria se divide en tres partes principales: memoria interna de datos, memoria externa de datos y memoria externa de programa.

La memoria interna de datos se divide también en tres regiones: 128 bytes inferiores, 128 bytes superiores y región de registros de función específica (SFR). Los 128 bytes inferiores y superiores son de RAM de propósito general mientras que la región de SFR's contiene a los registros de control y estado referentes a la funcionalidad del controlador USB.

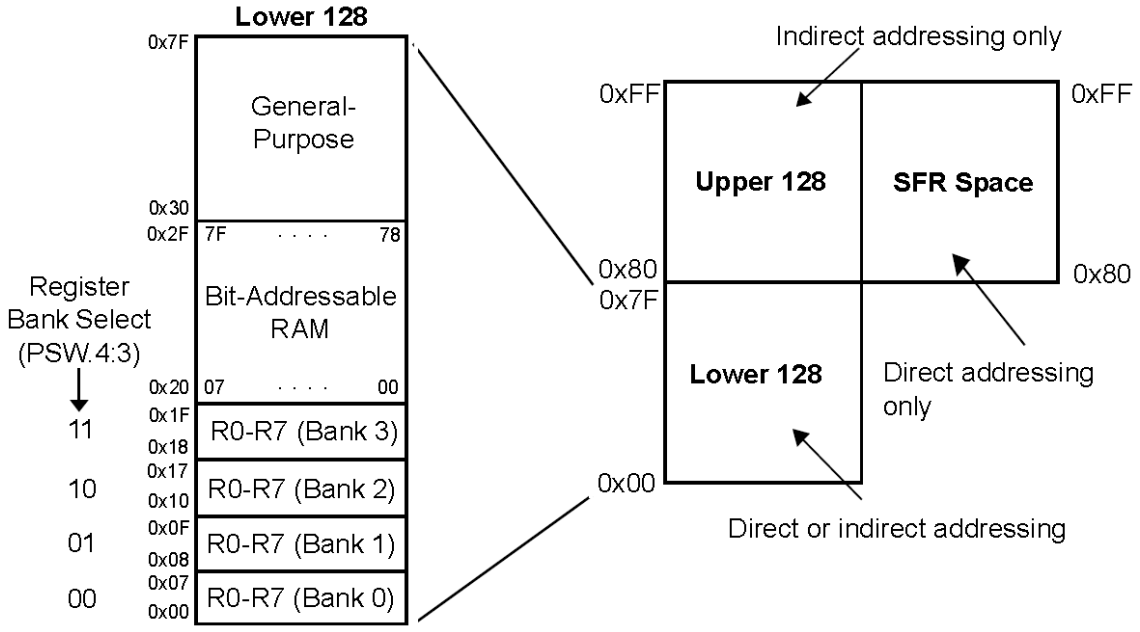


Figura 4.2 Organización de la memoria del EZ-USB.

La memoria externa en el microcontrolador 8051 tiene una arquitectura Harvard, memorias de programa y datos separadas físicamente. En el EZ-USB se utiliza una versión modificada de esta arquitectura en la que la memoria externa de datos y programa están separadas mientras que la memoria interna de datos y programa están unificadas siguiendo la arquitectura Von Neumann, debido a esta característica es posible cargar la RAM interna desde una fuente externa, por ejemplo una EEPROM conectada al bus I2C del chip o incluso por medio del bus USB.

El EZ-USB cuenta con una memoria RAM de 16 KB (0x0000 – 0x3FFF) embebida en el mismo chip y 512 bytes adicionales (0xE000 – 0xE1FF scratch RAM). Esta memoria, a pesar de estar integrada en el chip, es accedida por el *firmware* como si se tratara de una memoria externa. Es en esta memoria en la que se carga el *firmware* necesario para establecer la comunicación USB. Además de este espacio, se reservan también 7.5 KB (0xE200-0xFFFF) para usarse en registros de control/estado y buffers de los *endpoints*.

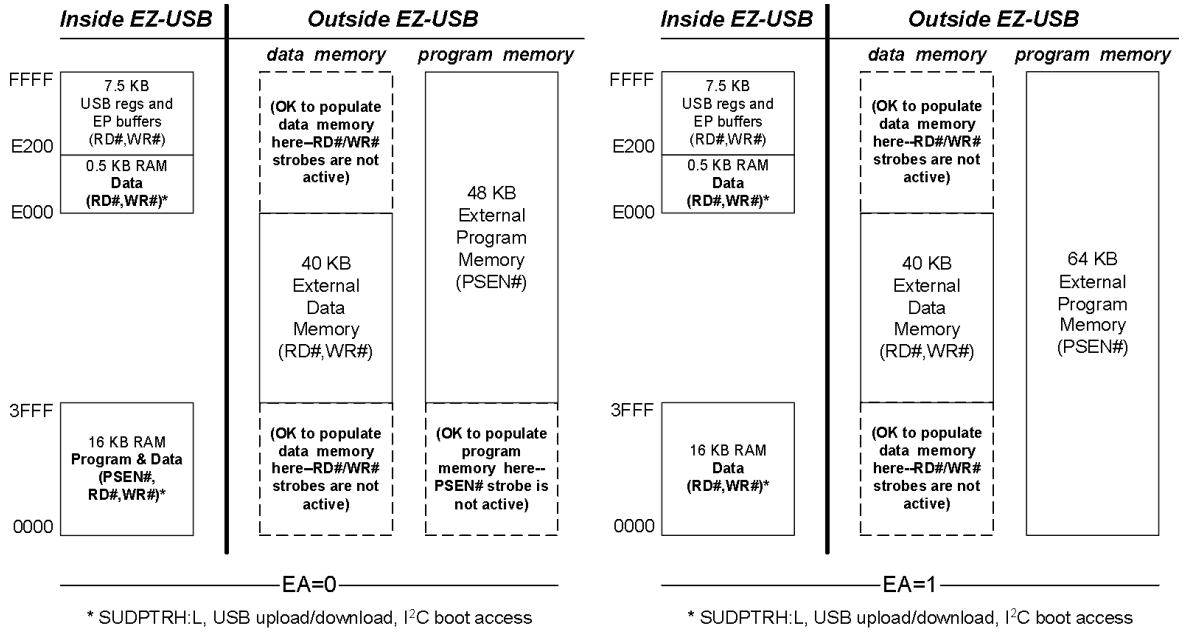


Figura 4.3 Posibles arreglos de memoria del EZ-USB.

Para el caso del encapsulado de 128 pines, al EZ-USB se le puede conectar hasta 64 KB de RAM externa. La figura 4.3 muestra las dos posibles configuraciones cuando se tiene una memoria externa conectada al EZ-USB. La memoria de programa se encuentra en la memoria externa y en los 16KB internos cuando EA = '0', mientras que estos 16 KB son solo de datos y la memoria externa es solo de programa cuando EA = '1'.

4.1.2 Reenumeración

Debido a que la configuración del EZ-USB puede modificarse por medio de *firmware*, el chip puede verse como un dispositivo distinto cada vez. Cuando el dispositivo se conecta al *host*, el EZ-USB se enumera como dispositivo default y descarga el *firmware* y con él las tablas de descriptores, ya sea por medio del bus USB o desde una EEPROM en el bus I2C. Una vez descargado el *firmware*, el dispositivo se enumera por segunda vez pero ahora utilizando la tabla de descriptores obtenida en la primera enumeración. A este proceso se le llama reenumeración.

4.1.3 Dominios en el EZ-USB

Las transferencias de datos se hacen utilizando los FIFO's esclavos. El EZ-USB cuenta con 6 *endpoints* de los cuales 2, los *endpoints* 0 y 1, son bidireccionales y son utilizados para propósitos de control y configuración. Los cuatro *endpoints* restantes (*endpoints* 2, 4, 6 y 8) pueden ser utilizados por el dispositivo para la transferencia de datos mediante alguno de los diferentes tipos de transacciones y en cualquiera de las dos direcciones (entrada o salida). Estos *endpoints* tienen asociado un FIFO de buffer configurable, el cual puede ser doble, triple o cuádruple. La configuración del buffer es determinada por el registro EP*x*FIFOCFG (*x* es el número del *endpoint*). Un buffer doble permite al 8051 la atención de un paquete de datos mientras uno más está siendo atendido por la interfase USB (SIE), triple y cuádruple buffer permite que 2 y 3 paquetes de datos, respectivamente, sean atendidos por el 8051 mientras uno más está siendo atendido por el SIE.

Durante cada transacción, los datos pueden ser controlados por alguno de los tres diferentes dominios del EZ-USB: el dominio USB (Serial Interface Engine), el domino del CPU (8051) y el dominio de la interfase (FIFO's esclavos).

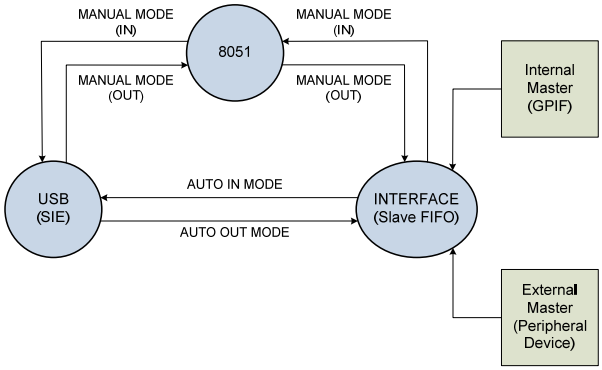


Figura 4.4 Tres diferentes dominios en el EZ-USB.

Existen dos opciones para el modo en que los datos son transferidos una vez que el *endpoint* se ha llenado con datos; estos dos modos son el manual y el automático y se configuran también utilizando el registro EP*x*FIFOCFG.

Por defecto, los *endpoints* 6 y 8 son configurados como *endpoints* de entrada, con doble buffer y transferencias en modo manual, los *endpoints* 2 y 4 son configurados como *endpoints* de salida, de buffer doble y modo manual.

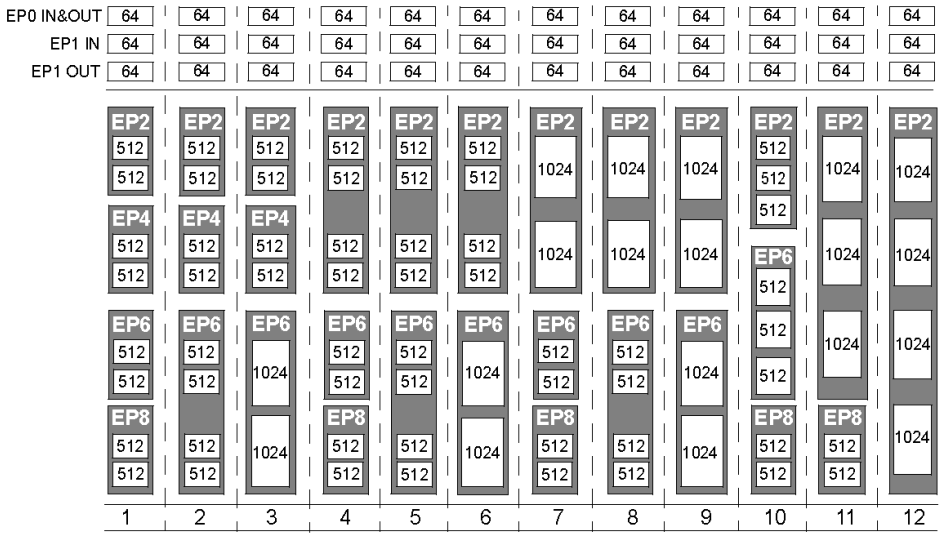


Figura 4.5 Posibles arreglos de la memoria reservada para endpoints.

En el modo manual, el CPU tiene acceso a los datos en el buffer del *endpoint*. Una vez que el *endpoint* está libre y puede recibir datos, los datos enviados por el *host* pueden almacenarse temporalmente en el *endpoint* (dominio USB), una vez almacenados los datos, el 8051 (dominio CPU) tiene acceso a los datos y puede modificarlos si es necesario e incluso puede o no pasarlos al dominio de la interfase (FIFO's esclavos). Una vez que los datos se encuentran en el dominio de la interfase, el CPU no tiene más control sobre ellos. Además de editar los paquetes y transferirlos entre dominios, el CPU también tiene la capacidad de crear paquetes de datos escribiéndolos en los registros EPxFIFOBUF, por ejemplo se puede crear un paquete de dos bytes escribiendo los dos bytes, uno en EP2FIFOBUF[0] y el otro en EP2FIFOBUF[1]; después de haber creado el paquete el CPU debe transferirlo al dominio correspondiente. Edición (*edit packets*), creación (*source packets*) y transferencia (*commit packets*) son las tres acciones que el CPU puede realizar sobre los paquetes cuando se configura en modo manual.

En el modo automático el CPU no tiene acceso a los datos y estos pasan directamente del dominio USB al dominio de la interfase. Esto permite que las transferencias de datos sucedan a una velocidad mucho mayor comparada con el modo manual.

4.2 El GPIF

El GPIF es una interfase paralela altamente flexible de 8 o 16 bits, la cual puede ser utilizada para conectar el EZ-USB directamente con microprocesadores, ASIC's o cualquier dispositivo externo, también puede ser utilizado para implementar protocolos tales como ATAPI o IEEE 1284 (puerto paralelo), entre otros. La interfase GPIF puede ser programada para generar las señales de habilitación y *handshake* para comunicación con el mundo real. Toda la funcionalidad del GPIF gira en torno a una máquina de estados finitos que puede ser programada mediante una interfase gráfica.

El GPIF cuenta con seis salidas programables de control (CTRL [5:0]) las cuales pueden ser utilizadas como líneas de habilitación de escritura/lectura o alguna otra funcionalidad que la interfase específica requiera. También están disponibles 6 líneas de

entrada (RDY [5:0]) las cuales pueden recibir señales provenientes de algún dispositivo periférico para ser muestreadas y así determinar, en base a alguna función lógica, si alguna acción debe ser ejecutada o no. Estas líneas de entrada son muy útiles para incluir estados de espera entre una acción y otra en el GPIF.

La activación y muestreo de señales se hace mediante el uso de formas de onda programables por el usuario. Cada forma de onda consiste de 7 estados programables y un estado ocioso (IDLE). En cada estado es posible insertar un punto de decisión con el cual se verificará el estado de alguna de las señales de entrada (Pines RDY) para así determinar cuál será el siguiente estado de la forma de onda.

La programación de las formas de onda se hace mediante la herramienta GPIFtool.exe proporcionada por Cypress®, con la cual se configura la frecuencia de reloj a la que trabajara el GPIF, si éste es interno o externo, las señales RDY y CTRL disponibles y el tamaño del bus (8 o 16 bits) entre otras características.

4.2.1 Transferencias del GPIF

Los datos en los FIFO's esclavos pueden ser accedidos por el GPIF en las dos direcciones, entrada y salida, y de dos modos distintos: Transferencia tipo simple y transferencia tipo FIFO ráfaga.

Las transferencias del GPIF se hacen entre los FIFO's esclavos y algún dispositivo externo, ya sea un microprocesador, ASIC, FPGA o algún otro dispositivo.

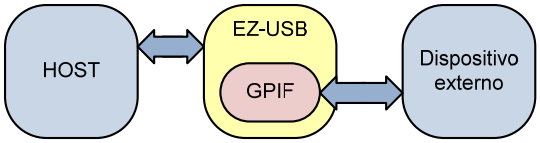


Figura 4.6 Transferencias del GPIF.

Antes de disparar cualquier transferencia del GPIF es importante asegurarse que el GPIF está en estado ocioso (IDLE) y que no se está atendiendo ninguna otra transferencia. Para verificar esta condición se utiliza el bit GPIFDONE del registro GPIFTRIG, este bit debe estar en estado lógico '1' antes de disparar cualquier transferencia. Con excepción del registro GPIFABORT, el cual es utilizado para forzar la terminación de la transferencia actual de manera inmediata, ninguno de los registros asociados con el GPIF deben ser manipulados mientras se está atendiendo una transferencia, esto puede causar que los datos de la transferencia sean erróneos.

4.2.2 Escritura FIFO

Con este tipo de transferencia se escribe un byte/word o más, por cada disparo de la forma de onda correspondiente, al periférico externo mediante el bus de datos del GPIF. El CPU dispara esta transferencia utilizando cualquiera de los siguientes registros:

- Registro GPIFTRIG, escribiendo el número de *endpoint* FIFO (0x02, 0x04, 0x06, 0x08) y el bit R/W de este registro en '0' para indicar que se trata de una operación de escritura.
- Registro EP*x*GPIFTRIG, (*x* es el número del endpoint) escribiendo cualquier valor en este registro se dispara la transferencia.

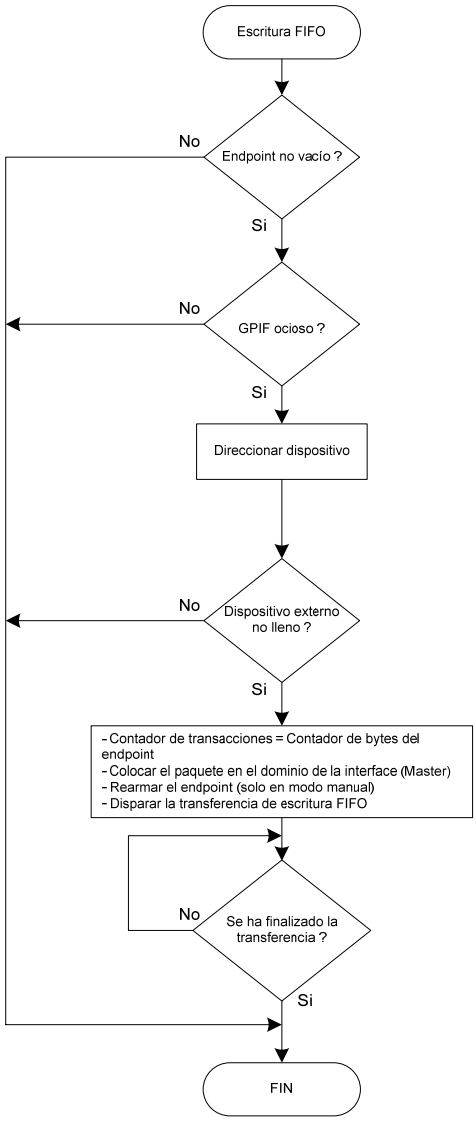


Figura 4.7 Diagrama de flujo para escrituras FIFO hechas por GPIF.

4.2.3 Lectura FIFO

En esta operación se lee uno o más de un byte/word del dispositivo externo por cada disparo de la forma de onda correspondiente. Al igual que en las transferencias de escritura el CPU dispara las transferencias de lectura utilizando cualquiera de los dos registros GPIFTRIG o EP*x*GPIFTRIG.

- GPIFTRIG. La transferencia de lectura se dispara al escribir este registro con el número de *endpoint* (0x02, 0x04, 0x06, 0x08) y el bit R/W puesto a '1'.

- EP*x*GPIFTRIG. La lectura de este registro dispara la transferencia de lectura.

En cualquiera de las dos operaciones pueden ser utilizados los registros GPIFTRIG o EP*x*GPIFTRIG para disparara la transferencia, sin embargo el acceso a GPIFTRIG es más rápido ya que es un registro de función específica y se accede de forma directa.

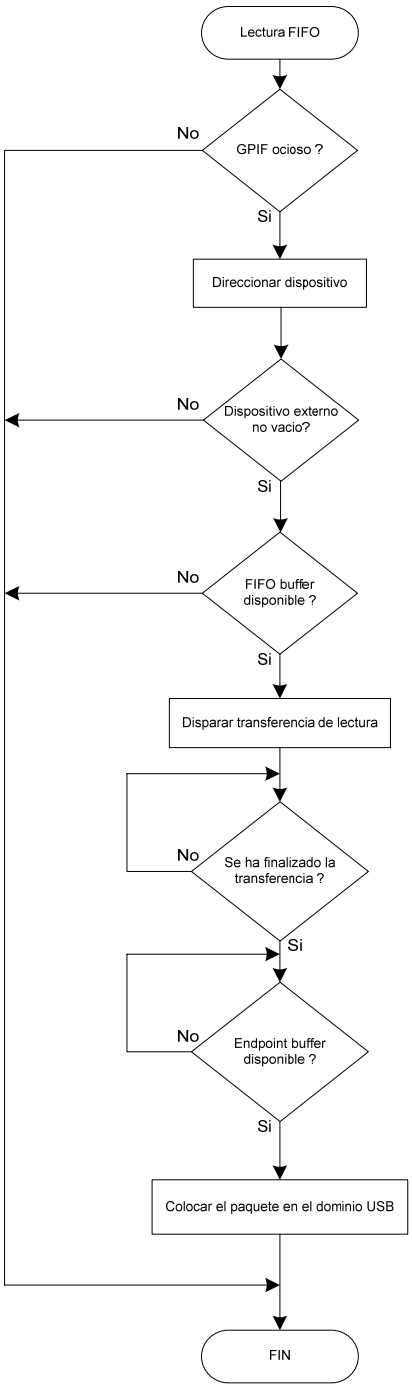


Figura 4.8 Diagrama de flujo para lecturas FIFO hechas por GPIF.

4.2.4 Escritura simple

Una escritura simple es una transferencia hacia el dispositivo externo en la que un byte/word se escribe a través del bus de datos de GPIF por cada disparo de la forma de onda correspondiente, a diferencia de una escritura FIFO, esta transferencia no utiliza los registros GPIFTRIG o EP*x*GPIFTRIG para disparar la transferencia. En lugar de estos registros, se utilizan XGPIFSGLDATLX o XGPIFSGLDATLNOX y XGPIFSGLDATH. Basta con cargar el registro XGPIFSGLDATX o XGPIFSGLDATLNOX con el dato de 8 bits que se quiere escribir por medio del GPIF para disparar la transferencia, si el dato a escribir es de 16 bits es necesario cargar la parte baja en XGPIFSGLDATLX y la parte alta en XGPIFSGLDATH. Si el byte menos significativo es cargado en XGPIFSGLDATLX automáticamente se disparará otra transferencia al finalizar la actual, si se usa XGPIFSGLDATLNOX se realizará una sola transferencia.

4.2.5 Lectura simple

Al igual que una lectura FIFO esta transferencia lee un byte/word del dispositivo externo. La transferencia se dispara al leer cualquiera de los registros XGPIFSGLDATLX o XGPIFSGLDATLNOX con lo cual se obtendrá el byte menos significativo, en caso de una transferencia de 16 bits el byte más significativo es leído de XGPIFSGLDATH. Al igual que en una escritura simple, el uso de el registro XGPIFSGLDATLX o XGPIFSGLDATLNOX determina si se dispara otra transferencia de forma automática o no.

4.3 El dispositivo USB implementado.

Al configurar el EZ-USB en un modo específico de trabajo, haciendo uso de los variados recursos de este controlador, en realidad se está implementando un dispositivo USB a la medida, que sirve a un propósito específico y que gracias a la configuración software del controlador USB utilizado, puede ser fácilmente modificado para

implementar nuevos requerimientos en la aplicación para la que fue diseñado e incluso modificarse también para servir a un propósito totalmente diferente.

El dispositivo USB implementado para esta aplicación hace uso de uno de los recursos más importantes y valiosos del controlador, la interfase de propósito general o GPIF. En la figura 4.9 se muestra un diagrama de bloques del dispositivo USB y los recursos del EZ-USB utilizados.

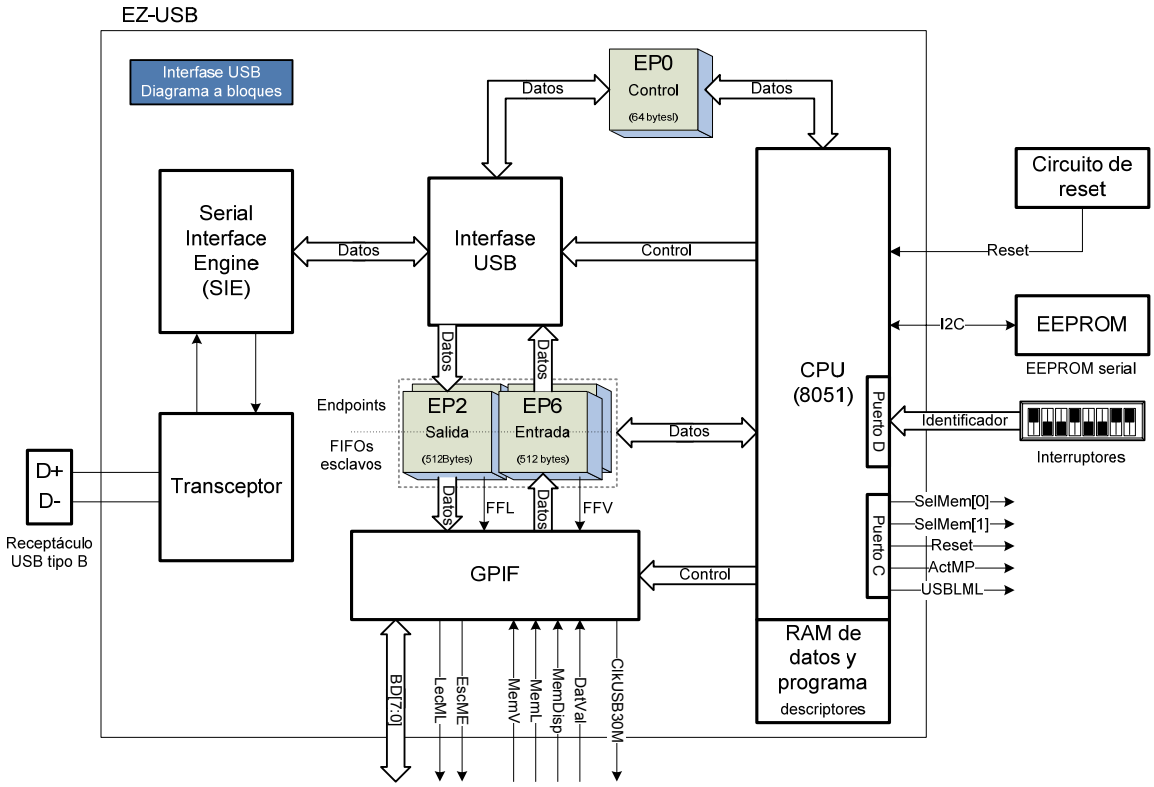


Figura 4.9 Diagrama a bloques del dispositivo USB implementado.

Mediante la interfase GPIF, el dispositivo USB transfiere datos desde y hacia los *endpoints* utilizados en el diseño, esta información puede provenir de la PC o desde la lógica externa con la que se interconecta GPIF. En la figura 4.9 se observa una línea punteada sobre los *endpoints* la cual indica que para el domino USB (SIE e interfase USB)

la región de memoria reservada para EP2 y EP6 es vista como *endpoints* mientras que para el domino de la interfase (GPIF) esta misma región es vista como FIFO's esclavos. El dominio del 8051 o CPU tiene acceso a los datos de los *endpoints* y controla el acceso de los otros dos dominios a estos datos, también se encarga de interpretar la información que la PC envía y la que se recibe mediante GPIF. La configuración por software del EZ-USB se carga desde una memoria EEPROM conectada al bus I2C y se hace uso del Puerto D para asignar un identificador al dispositivo mediante interruptores.

Los bloques mostrados por la figura 4.10 son prácticamente los mismos que se mencionan en la sección 4.1, a continuación se explica aquellos módulos que han sido configurados de manera específica para servir al propósito de este trabajo.

EP2: es el *endpoint* de salida, tiene una capacidad de 512 bytes, cuenta con doble buffer y soporta transferencias tipo Bulk. Los comandos e información a transmitir mediante GPIF hacia el sistema de almacenamiento son enviados a través de este *endpoint*.

EP6: es el *endpoint* de entrada, tiene capacidad de 512 bytes, cuenta también con doble buffer y soporta transferencias tipo Bulk. Toda la información que se lee mediante GPIF desde el sistema de almacenamiento es almacenada temporalmente en este *endpoint*.

CPU: configurado para tener control sobre los otros dos dominios del EZ-USB así como para obtener el identificador del dispositivo mediante el puerto D y enviar señales útiles a la lógica externa utilizando el puerto C. El CPU también carga la configuración del EZ-USB, almacenada en la EEPROM externa, en la memoria RAM interna. Las señales enviadas mediante el puerto C a la lógica externa son:

- ***SelMem[1:0]***, utilizadas por la lógica externa para definir cuál de las memorias o registros (***ME***, ***ML***, ***ML*** o ***RTR***) debe ser seleccionada.
- ***Reset***, utilizada para reiniciar la lógica externa.

- ***ActMP***, utilizada para indicar a la lógica externa que se deben leer todos los datos del registro ***MP*** (memoria de parámetros).
- ***USBLML***, utilizada para indicar a la lógica externa que actualmente la memoria de lectura ***ML*** está siendo leída y por tanto no debe ser restablecida.

GPIF: es la interfase con la lógica externa, programada para atender las operaciones de acceso a las memorias ME y ML del sistema de almacenamiento y los registros ***MP*** y ***RTR***. Para interconectarse con la lógica externa se tienen las siguientes señales:

- ***BD[7:0]***, es el bus de datos bidireccional utilizado para transferir la información hacia y desde el sistema de almacenamiento.
- ***EscME***, es la señal de habilitación de escritura, indica al sistema de almacenamiento una petición de escritura en ME o RTR.
- ***LecML***, es la señal de habilitación de lectura, indica al sistema de almacenamiento una petición de lectura de ML o MP.
- ***DatVal***, señal proveniente del sistema de almacenamiento, indica a la lógica de GPIF que el dato correspondiente a la petición de lectura inmediata anterior ya está presente en el bus.
- ***MemDisp***, señal proveniente del sistema de almacenamiento, indica a GPIF si la memoria o registro seleccionado mediante ***MemSel[1:0]*** está o no disponible para operaciones de lectura o escritura.
- ***MemL***, señal proveniente del sistema de almacenamiento, indica a GPIF si la memoria o registro seleccionado mediante ***MemSel[1:0]*** está o no llena, de ser así no se podrán realizar operaciones de escritura.
- ***MemV***, señal proveniente del sistema de almacenamiento, indica a GPIF si la memoria o registro seleccionado mediante ***MemSel[1:0]*** está o no vacía, de ser así no se podrán realizar operaciones de lectura.

- ***ClkUSB30M***, es la señal de reloj a la que trabaja GPIF, es enviada a la lógica externa (sistema de almacenamiento) con propósitos de sincronización, es una señal de 30 MHz.

Internamente, GPIF también utiliza señales que le indican el estado de los *endpoints*, en el diagrama a bloques estas señales son ***FFL*** y ***FFV*** las cuales indican FIFO lleno y FIFO vacío.

RAM de datos y programa: es la RAM interna en la que se almacena la configuración del EZ-USB una vez que ésta ha sido transferida desde la EEPROM. La información almacenada en esta memoria incluye el firmware que da la funcionalidad deseada al dispositivo, así como los descriptores que identifican al mismo. Estos descriptores se escribieron en un archivo .a51 (ensamblador) y se integran datos tales como el *Vendor ID*, *Product ID*, versión de USB, etc. Algunos de los datos del dispositivo en base a los descriptores se presentan en la siguiente tabla.

Tabla 4.1 Algunos valores de las tablas de descriptores del dispositivo USB.

Campo	Valor	Descripción
bcdUSB	0x0200	Versión de la especificación USB 2.0
idVendor	0x04B4	*Identificador del fabricante
idProduct	0x0410	*Identificador del producto
bNumConfigurations	1	Número de configuraciones
bNumInterfaces	1	Número de interfases
bNumEndpoints	2	Número *endpoints*

* En los campos VendorID y ProductID se utilizan los datos del fabricante (Cypress®) pues no se cuenta con identificadores propios los cuales son asignados por USB.org

En el apéndice C se presenta la lista completa de descriptores del dispositivo en un archivo . a51 el cual forma parte de la implementación del *firmware* del dispositivo.

4.3.1 Desarrollo del *firmware*

El desarrollo del *firmware* para la aplicación se realizó mediante el compilador de C C51 integrado en el ambiente de desarrollo (IDE) Microvision2 de Keil®. El fabricante del chip recomienda este compilador e incluso proporciona librerías y funciones útiles para el desarrollo de aplicaciones utilizando EZ-USB y Microvision2. Ésto facilita un poco el desarrollo del *firmware* para el dispositivo, algunas partes de código proporcionado son útiles en la definición de constantes, tipos de datos, funciones y nombres de registros del EZ-USB. Los principales archivos y los que son de más ayuda son:

- fw.c, es el marco de trabajo (Framework) que sirve como referencia para la implementación del dispositivo USB.
- periph.c, es donde radica la funcionalidad del dispositivo. La mayor parte de la lógica para manejar las transferencias debe ser implementada aquí.
- dscr.a51, este archivo contiene las tablas de descriptores del dispositivo. Puede codificarse a voluntad según las características del dispositivo que se desee implementar.

Aunque estos tres archivos son útiles en la implementación del firmware para el dispositivo USB, estos no resuelven necesidades específicas para un diseño, lo cual deja todavía bastantes tareas por resolver al programador.

4.3.2 Funciones del marco de trabajo (framework)

El marco de trabajo implementa las funciones básicas requeridas por un dispositivo USB. Estas funciones incluyen la inicialización del chip, el manejo de las peticiones estándar de USB, el modo de consumo de potencia (modo suspendido y *wake up*).
Algunas de las funciones implementadas en el marco de trabajo son:

void TD_Init(void)

Esta función es llamada una vez durante la inicialización y previo a ejecutarse la reenumeración. Con esta función se inicializan variables globales, se configuran *endpoints* y se inicializa el GPIF entre otras tareas. En esta aplicación, también se utiliza para configurar los puertos utilizados en la interfase con la lógica externa y los interruptores de identificación.

void TD_Poll(void)

Esta función es llamada repetidas veces mientras el dispositivo está activo, puede decirse que ésta función es la más importante y debe incluir todas las operaciones necesarias para ejecutar las transacciones entre el EZ-USB y la lógica externa.

BOOL TD_Suspend(void)

Esta función es llamada previamente a la entrada al modo suspendido, de hecho éste es su objetivo; poner al dispositivo en un modo de bajo consumo. Regresa un valor de tipo booleano (BOOL), el cual puede ser modificado en el código del usurario; TRUE si se desea entrar a modo suspendido o FALSE si no se desea entrar en este modo. Para el caso del dispositivo a implementar, esta función devuelve un valor FALSE pues no se requiere que el dispositivo entre al modo suspendido.

void TD_Resume(void)

Esta función es llamada para continuar la operación normal del dispositivo después de que éste ha sido puesto en modo suspendido.

bit EEPROMWritePage(WORD addr,BYTE xdata * ptr,BYTE len)

Esta función es utilizada para escribir en la EEPROM conectada al EZ-USB mediante el bus I2C. Recibe como parámetros la dirección inicial a la que se escribirá, un apuntador a BYTE (región donde se encuentran los bytes a escribir) y la longitud (número de bytes)

de los datos que se van a escribir. Regresa un valor de tipo bit; '1' en caso de que la operación se haya realizado satisfactoriamente, de lo contrario '0'.

bit EEPROMRead(WORD addr, BYTE length, BYTE xdata *buf)

Esta función es útil para leer datos almacenados en la EEPROM. Recibe como parámetros la dirección inicial de lectura, la longitud (número de bytes) de los datos que se van a leer y un apuntador a BYTE (región donde almacenarán los bytes leídos). Regresa un valor de tipo bit; '1' en caso de que la operación se haya realizado satisfactoriamente, de lo contrario '0'.

Aunque estas dos últimas funciones no se encuentran en el marco de trabajo (fw.c) sino que están incluidas en los archivos eeprom.h y eeprom.c son también importantes ya que haciendo uso de estas funciones se proporciona al módulo de comunicación USB-Fibra óptica la posibilidad de tener un identificador configurable mediante interruptores. El identificador en los interruptores es leído mediante el Puerto D de propósito general del EZ-USB y posteriormente es escrito en la EEPROM para que la próxima vez que el dispositivo sea puesto en operación tome este nuevo identificador. La lectura de EEPROM se realiza cada vez que es dispositivo es puesto en operación, se lee el identificador almacenado en la EEPROM y se compara con el leído en los interruptores, si son diferentes se escribe en EEPROM el identificador presente en los interruptores.

4.3.3 Descripción de las formas de onda de GPIF

La interfase con la lógica externa implementada mediante GPIF hace uso de un oscilador de 30MHz, ancho de bus de 8 bits, 4 señales RDY (**MemV, MemL, MemDisp y DatVal**) y dos señales de control (CTL); una para habilitación de escritura (**EnWr**) y otra para habilitación de lectura (**EnRd**).

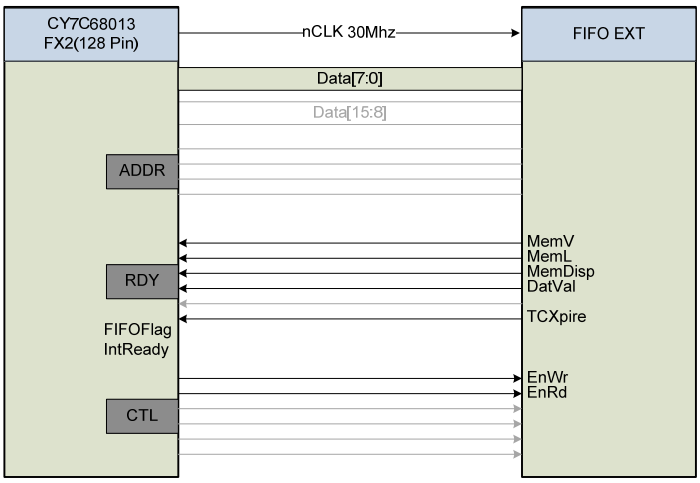


Figura 4.10 Interfase GPIF con la lógica externa.

Aunque en la figura se observa el uso de otra señal RDY marcada como **TCXpire**, ésta es una señal interna que sirve para indicar cuándo se ha completado el número de transacciones previamente especificadas a GPIF. Aunque no existe una conexión física de esta señal con la lógica externa, también es útil para determinar el siguiente estado de GPIF mediante un punto de decisión (♦).

Forma de onda de escritura

La siguiente figura muestra la forma de onda utilizada en operaciones de escritura de la interfase GPIF del dispositivo USB.

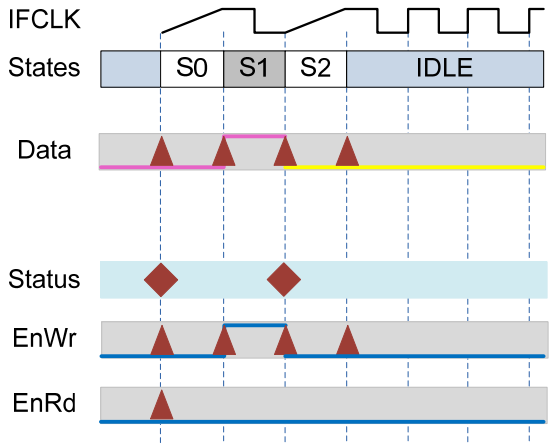


Figura 4.11 Forma de onda de escritura de GPIF.

Las operaciones de escritura se realizan utilizando tres estados:

- Estado S0: Punto de decisión.
- Estado S1: Se pone la habilitación de escritura (***EnWR***) en alto al mismo tiempo que se pone en el bus el dato a escribir. GPIF tiene el control sobre los datos en el FIFO esclavo y por tanto no es necesario especificar cuál es el dato que se va a escribir.
- Estado S2: Punto de decisión.
- Estado IDLE: Si no hay actividad la máquina de estados de GPIF permanece en este estado.

En el primer punto de decisión la lógica que se implementó toma como referencia el valor de la línea RDY llamada **MemDisp** (Disponible) la cual indica si en la lógica externa el sistema de almacenamiento no se encuentra ocupado realizando alguna operación de escritura o lectura. La figura 4.12 muestra la ventana de edición del punto de decisión.

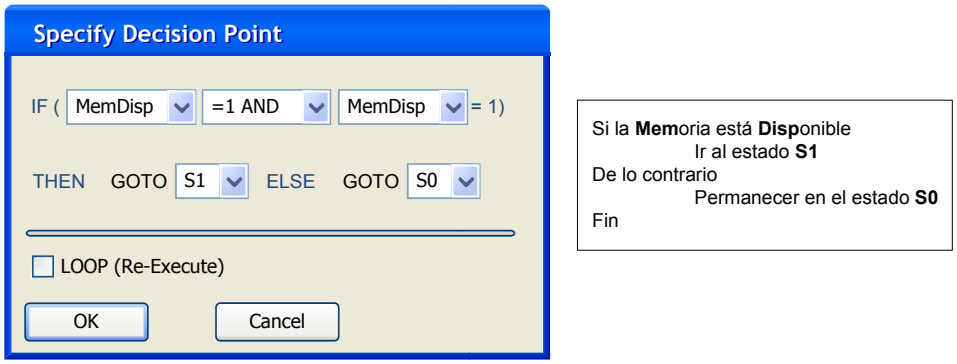


Figura 4.12 Primer punto de decisión de la forma de onda de escritura.

En el segundo punto de decisión la lógica que se implementó es en base a la señal interna TCXpire la cual se pone en alto una vez que el número de transacciones especificadas se ha completado.

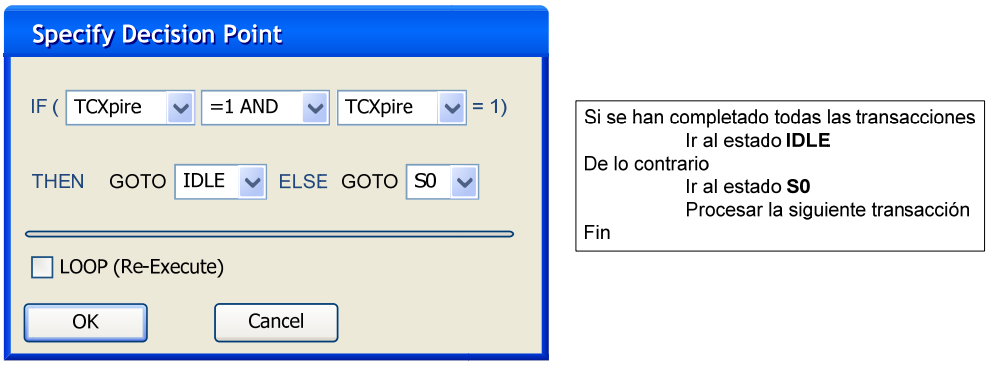


Figura 4.13 Segundo punto de decisión de la forma de onda de escritura.

Forma de onda de lectura

La siguiente figura muestra la forma de onda utilizada en operaciones de lectura de la interfase GPIF del dispositivo USB.

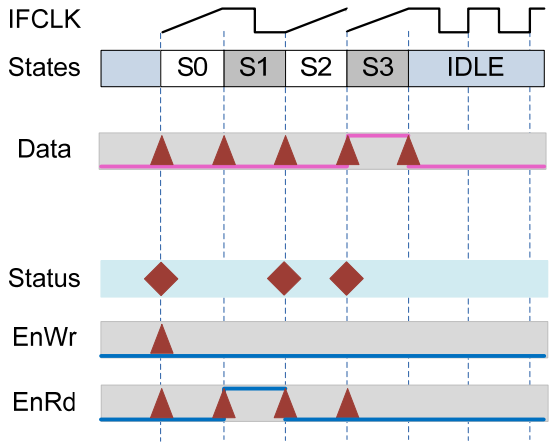


Figura 4.14 Forma de onda de lectura de GPIF.

Las operaciones de escritura se realizan utilizando cuatro estados:

- Estado S0: Punto de decisión.
- Estado S1: Se pone la habilitación de lectura (***EnRd***) en alto.
- Estado S2: Punto de decisión.

- Estado S3: Punto de decisión. También se muestrea el bus de datos para obtener el dato que se lee de la lógica externa.
- Estado IDLE: Si no hay actividad, la máquina de estados de GPIF permanece en este estado.

En el primer punto de decisión la lógica que se implementó es en base a la línea RDY llamada **MemDisp** (Disponible) y es idéntica a la utilizada en la forma de onda de escritura.

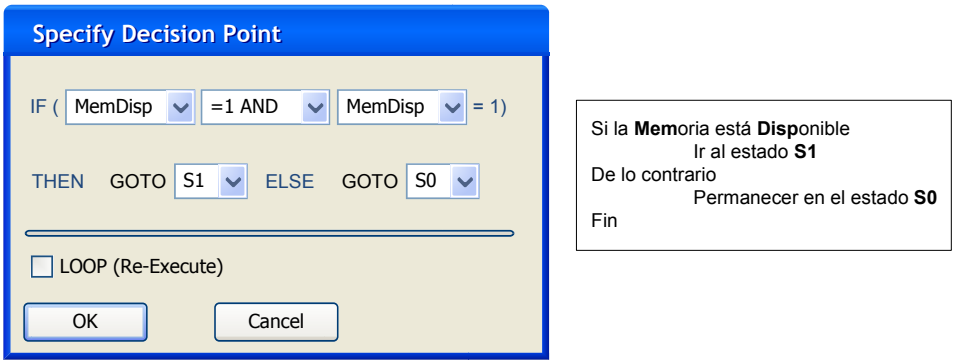


Figura 4.15 Primer punto de decisión de la forma de onda de lectura.

El segundo punto de decisión genera un tiempo de espera para tomar el dato que la lógica externa presenta en el bus después de haber recibido la habilitación de lectura (**EnRd**). Este tiempo se genera utilizando la línea RDY llamada **DatVal**.

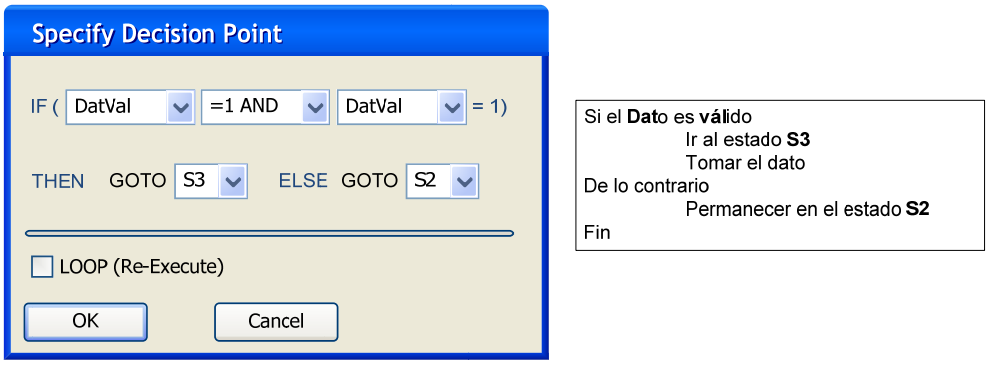


Figura 4.16 Segundo punto de decisión de la forma de onda de lectura.

El tercer punto de decisión es idéntico al segundo de la forma de onda de escritura (Figura 4.13), el cual pone a la máquina en el estado IDLE o S0 dependiendo de si el número de transacciones, en este caso lecturas, se ha completado o no.

La utilidad gpiftool.exe permite generar el archivo gpif.c el cual puede ser exportado a cualquier proyecto en el ambiente integrado de desarrollo MicroVision2 lo que permite la fácil integración del código implementado en la función principal TD_Poll() del marco de trabajo y el diseño de las formas de onda deseadas para lectura y escritura hacia la lógica externa utilizando GPIF.

4.4 Funcionalidad del *firmware* del dispositivo USB implementado

Como se mencionó en la sección 4.3.2, la función TD_Poll() es en la que se implementa la mayor parte de la funcionalidad de un dispositivo USB basado en el chip EZ-USB. Esta función es llamada de forma repetitiva mientras el dispositivo está activo, de hecho se hacen llamados a esta función y a otras más dependiendo de las condiciones del bus. La figura 4.17 presenta la forma en que se hacen las llamadas a TD_Poll() y a otras funciones. Mientras que los llamados a TD_Suspend() y TD_Resume dependen de las condiciones del bus, el llamado a TD_Poll() se hace constantemente dentro de un ciclo infinito.

La función TD_Poll() está compuesta por dos partes principales, una que maneja las operaciones de escritura, es decir, los datos enviados por el host al dispositivo y que requieren ser escritos mediante la forma de onda de escritura de GPIF. Existe también una parte dedicada al manejo de lecturas, es decir, al envío, hacia el host, de datos leídos mediante la forma de onda de lectura de GPIF y que anteriormente fueron solicitados por el host. Recordemos que todas las transacciones son iniciadas por el host.

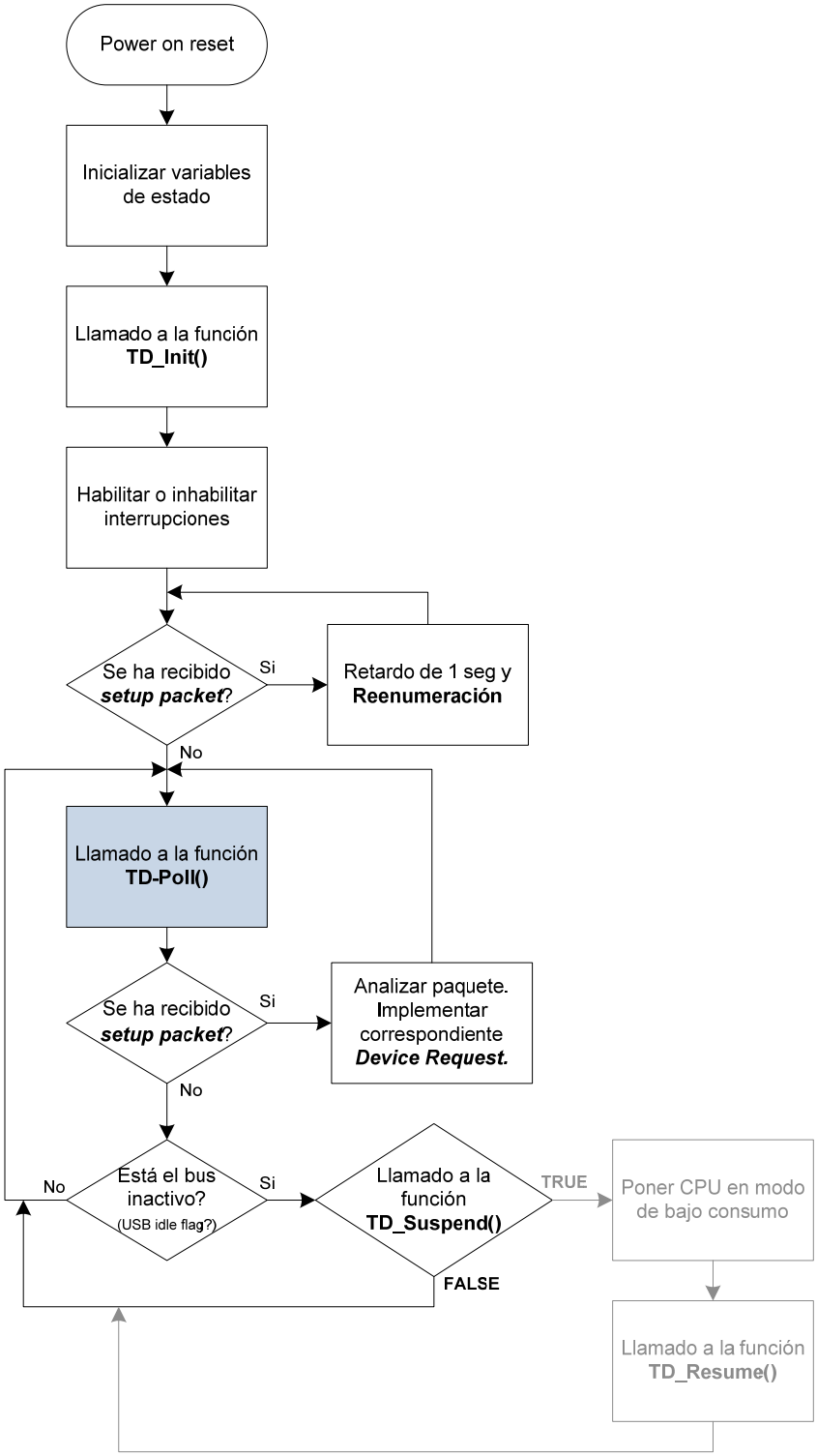


Figura 4.17 Ejecución del *firmware* implementado en un dispositivo basado en el EZ-USB.

Debido a la forma en que TD_Poll() es llamada, las operaciones de escritura y lectura no necesariamente se completan con una sola llamada a esta función. La ejecución de la función TD_Poll() puede incluir la ejecución de la sección de código dedicada a escrituras o a lecturas. Existen variables globales cuyo valor es válido en cada llamado a TD_Poll() y definen qué parte del código se ejecuta; ya sea el dedicado a escrituras o el de lecturas, las variables más importantes se muestran en la siguiente tabla:

Tabla 4.2 Variables importantes en el control de flujo de la función TD_Poll().

Variable	Tipo	Función
auxOutTC	INT	Esta variable almacena el número de datos que GPIF debe transferir hacia la lógica externa (escrituras) después de que se ha hecho la petición por medio de un comando ***Transmite***, enviado por el host, y ésta no se atendió.
Recibidos	DWORD	Almacena el número de bytes que GPIF debe transferir desde la lógica externa (lecturas). Se obtiene al leer la respuesta al comando ***Recibe*** y se decrementa cada vez que GPIF transfiere (lee) un paquete de datos.
Comando	INT	Modificada por la sección de código dedicada a operaciones de escritura. Codifica el tipo de comando que se ha recibido del host y es utilizada por la sección dedicada a lecturas para responder al host según el comando recibido. También se utiliza para indicar que el número de transacciones de lectura que debe hacer GPIF es mayor de 496.
Datos	BOOL	Funciona como una bandera que dispara la sección de código dedicada a lecturas, si es TRUE significa que hay datos que GPIF debe leer o que es necesario generar una respuesta a un comando enviado por el host.

La forma en que la función principal que ejecuta se muestra el siguiente diagrama de flujo.

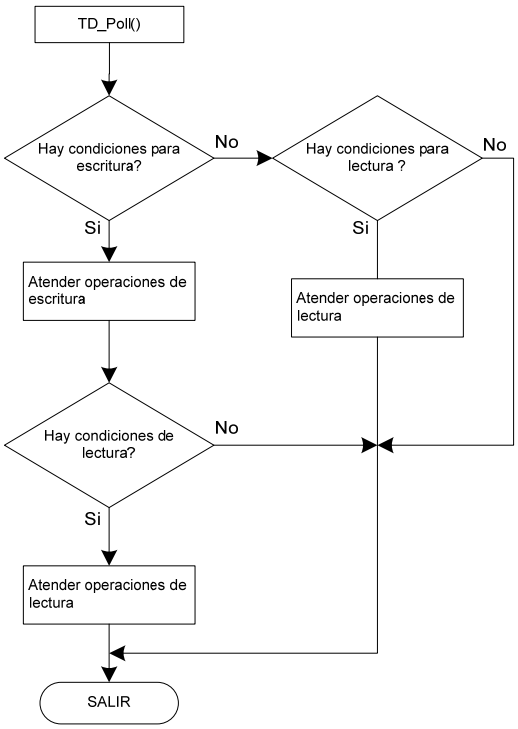


Figura 4.18 Diagrama de flujo simplificado de la función TD_Poll().

El que haya condiciones para escritura implica que el host haya enviado datos que requieran ser escritos y que las condiciones de la lógica externa permiten realizar las operaciones de escritura. Después de haber atendido las operaciones de escritura serán atendidas las operaciones de lectura, ya sea que se trate de lecturas a la lógica externa o de la generación de una respuesta al comando enviado por el host. Una operación de escritura puede no atenderse debido a la ausencia de las condiciones adecuadas. Sin embargo, puede ser atendida en la siguiente ejecución de la función TD_Poll() en la que las condiciones sean las adecuadas. De igual manera las operaciones de lectura pueden atenderse en varias llamadas a la función y dependiendo de las condiciones de las variables antes mencionadas. Para explicar mejor como está implementada la función TD_Poll() se presentan a continuación los diagramas de flujo para atención a operaciones de escritura y lectura. Es importante mencionar que aunque presentan de manera separada, trabajan de manera conjunta y complementaria.

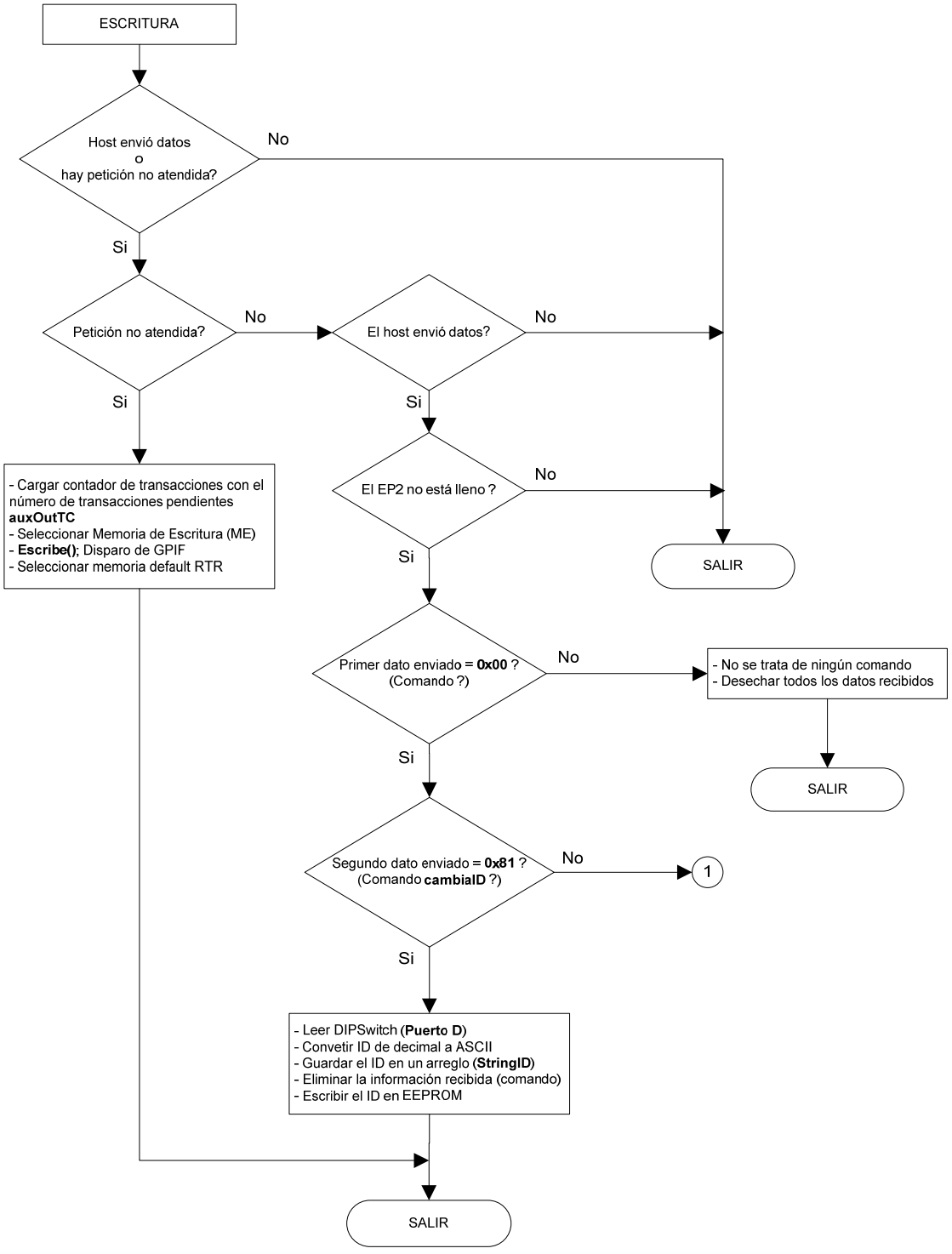


Figura 4.19 Diagrama de flujo para operaciones de escritura (Parte 1).

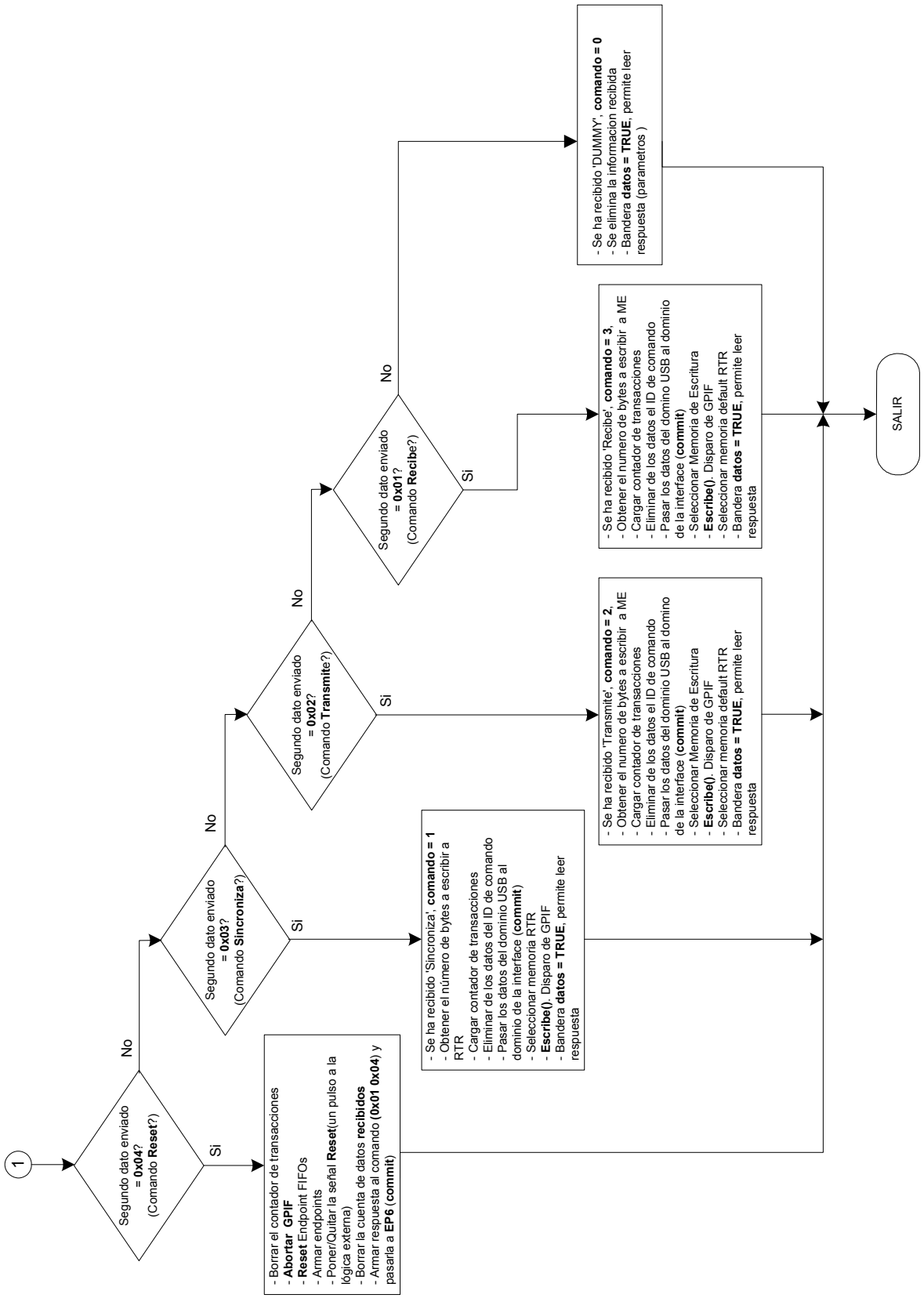


Figura 4.20 Diagrama de flujo para operaciones de escritura (Parte 2).

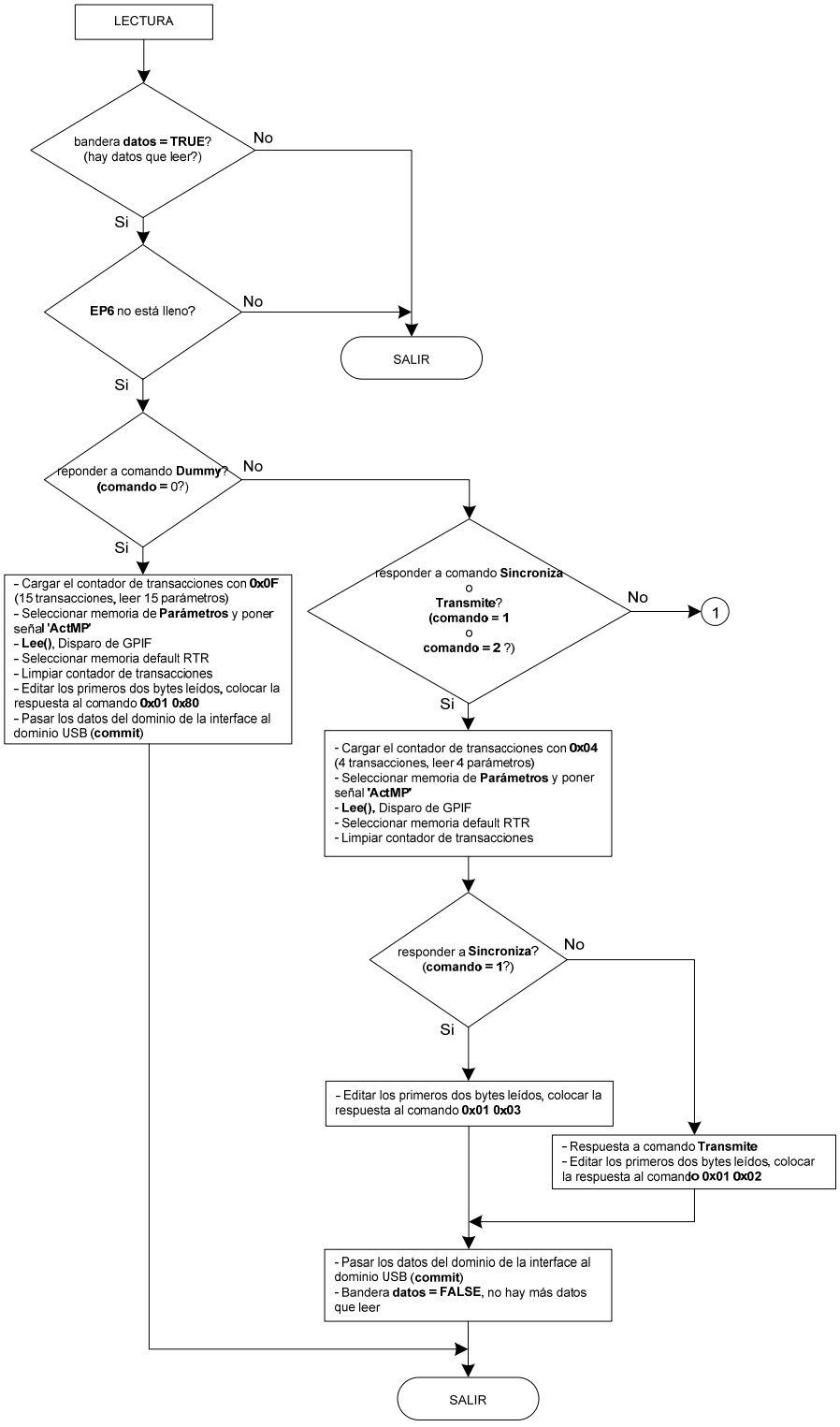


Figura 4.21 Diagrama de flujo para operaciones de lectura (Parte 1).

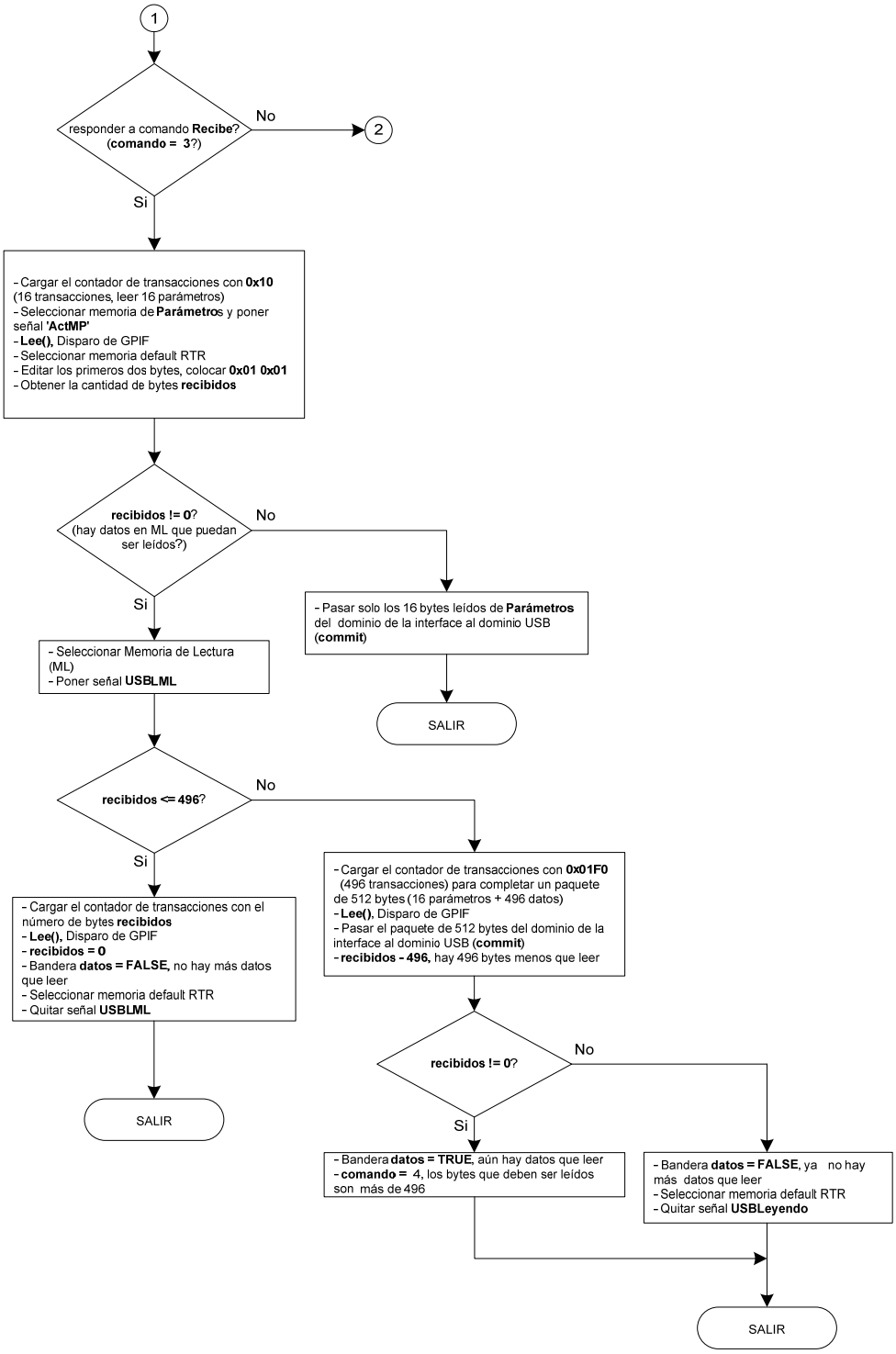


Figura 4.22 Diagrama de flujo para operaciones de lectura (Parte 2).

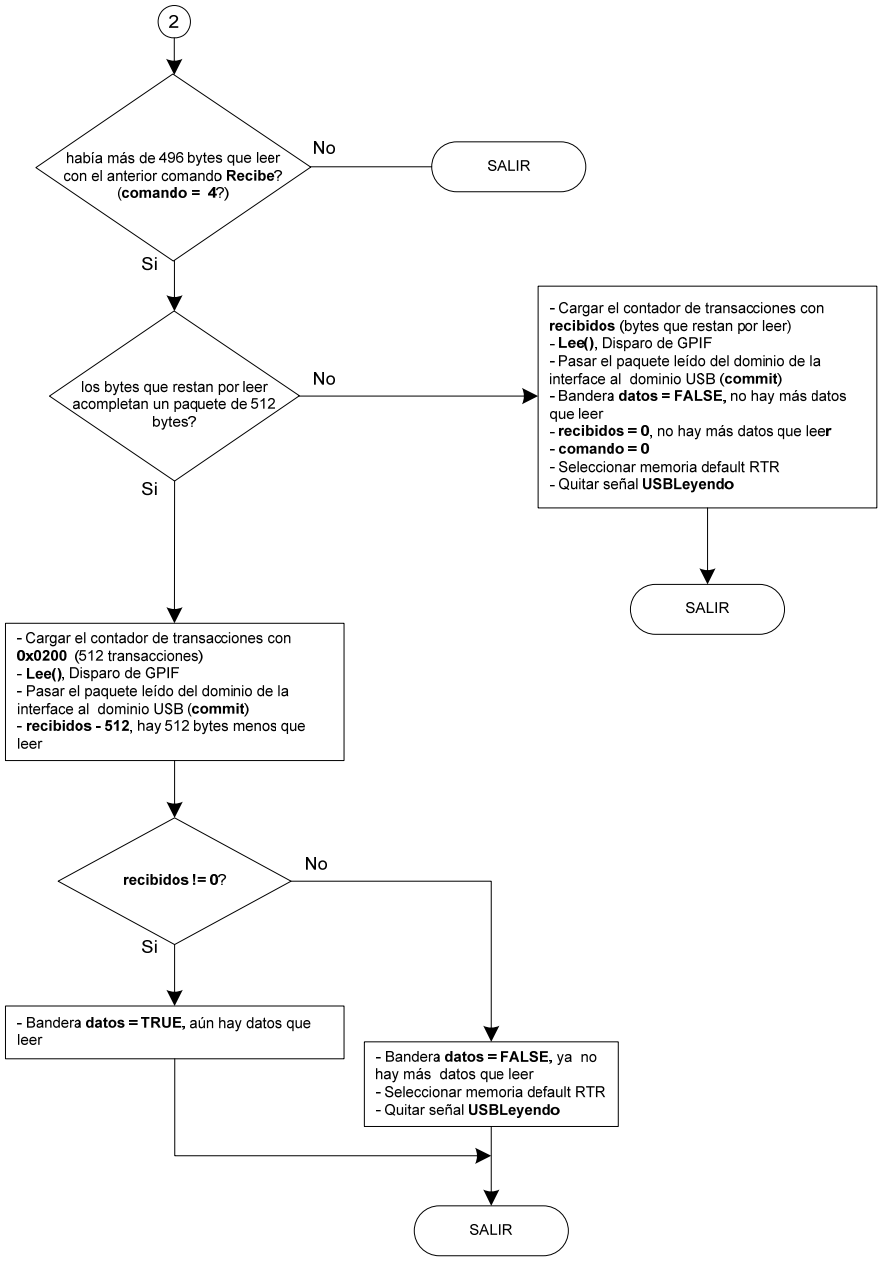


Figura 4.23 Diagrama de flujo para operaciones de lectura (Parte 3).

En los diagramas de flujo anteriormente mostrados existen llamadas a las funciones **Escribe()** y **Lee()** las cuales disparan las transferencias de GPIF escribiendo en el registro GPIFTRIG. La ejecución de estas funciones involucra el muestreo de dos señales RDY de la interfase GPIF; FV y FL, FIFO vacío y FIFO lleno respectivamente. Si se van a realizar operaciones de escritura hacia la lógica externa es necesario garantizar que el dispositivo externo no esté lleno (FL = '0') mientras que si se desea leer es necesario asegurar que el dispositivo externo no esté vacío (FV = '0'). Las funciones Escribe() y Lee() se describen con los siguientes diagramas de flujo.

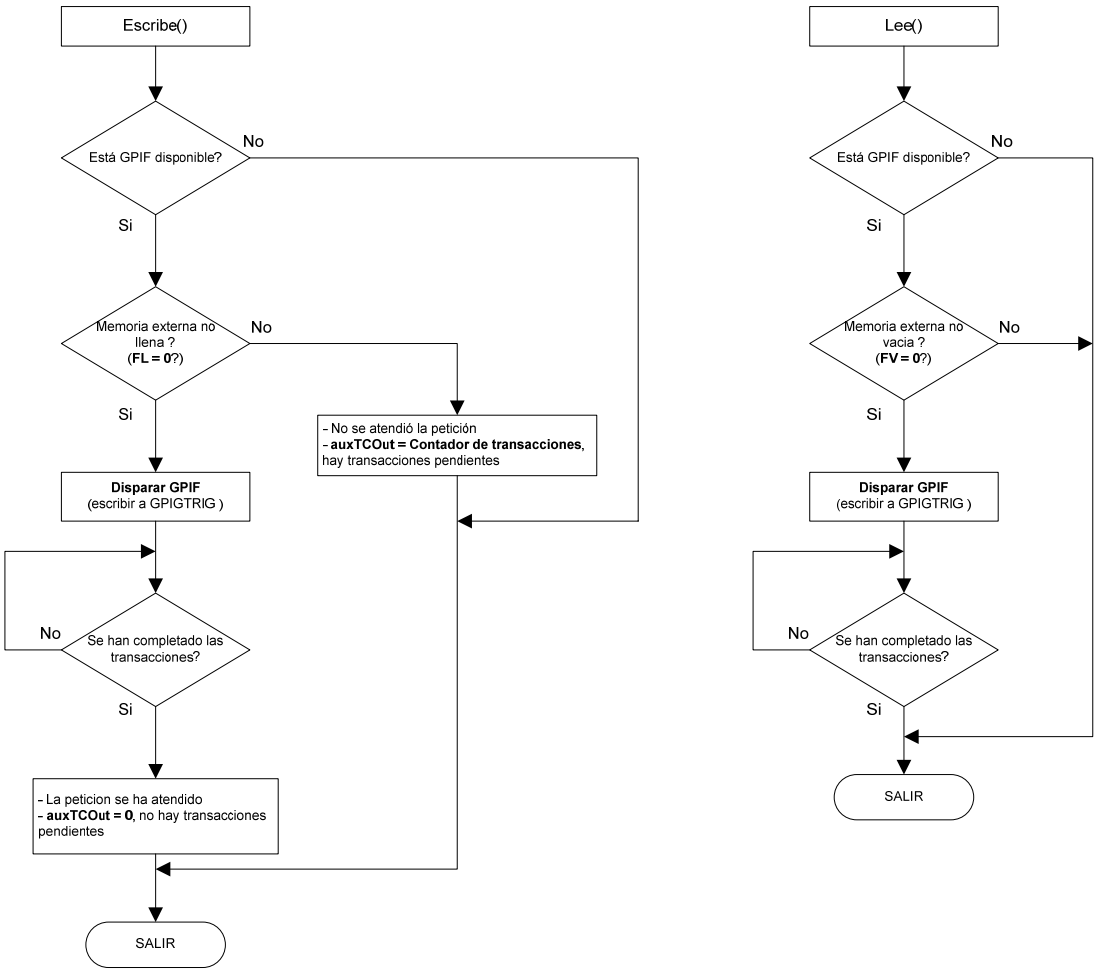


Figura 4.24 Funciones Escribe() y Lee(), estas funciones son las que disparan las transferencias de GPIF.

Capítulo 5 El controlador de RAM

5.1 Introducción

Para la implementación del control de acceso a la memoria SDRAM se utilizó un FPGA (Field Programmable Gate Array), el cual es un dispositivo semiconductor basado en una matriz de bloques lógicos configurables (CLB's, Configurable Logic Blocks) los cuales pueden ser interconectados mediante líneas de conexión programables permitiendo así, implementar casi cualquier sistema en un mismo chip. La configuración de los CLB's y sus interconexiones pueden ser programados una y otra vez (FPGA's basados en SRAM) y su programación se realiza mediante lenguajes descriptivos de hardware (HDL, Hardware Description Language) tales como Verilog o VHDL. La figura 5.1 muestra un diagrama de un FPGA en el que se integran bloques de entrada/salida (IOB's, In/Out Block), administradores digitales de señal de reloj (DCM's, Digital Clock Manager) y bloques lógicos configurables.

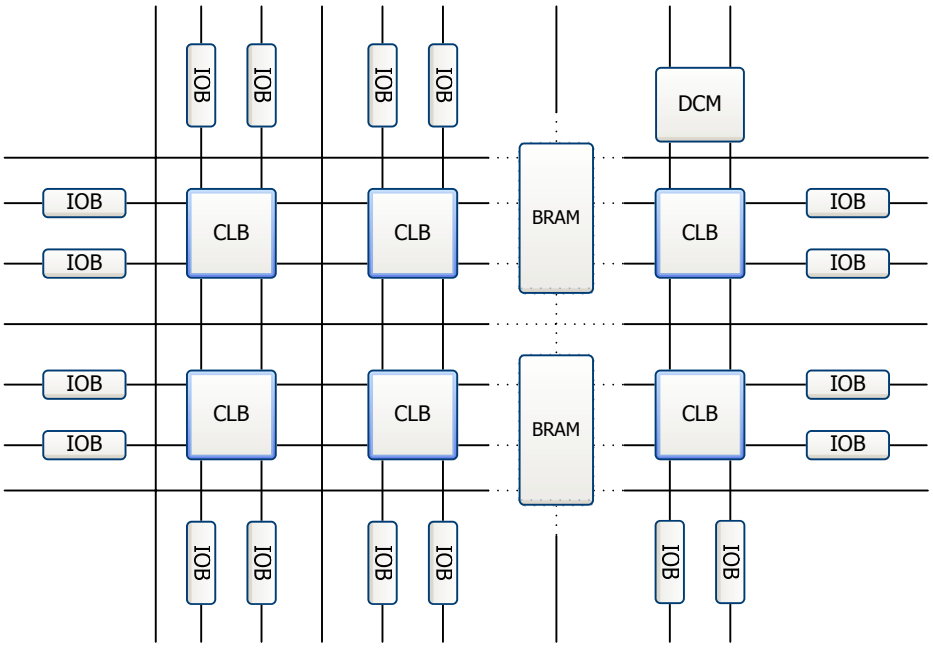


Figura 5.1 Bloques que integran un FPGA.

Los CLB's son la unidad lógica básica al interior del FPGA, los cuales constan de una matriz configurable de interruptores con 4 ó 6 entradas, circuitería para selección (mux, demux) y flip-flops. La matriz de interruptores es flexible y puede configurarse para implementar lógica combinatoria, registros de corrimiento o memoria RAM.

Para la interconexión con el exterior, se hace uso de los bloques de entrada/salida los cuales agrupan pines configurables en cuanto a dirección (I/O) y estándares específicos de voltaje y frecuencia. En caso de que el diseño a desarrollar necesite de memoria, muchos de los FPGAs disponibles cuentan con bloques de RAM (BRAM) internos. En algunos modelos de Xilinx®, se tienen hasta 10 Mb de RAM en bloques de 36kb los cuales soportan modo de acceso bipuerto.

Para efectos de temporización y acondicionamiento de señales de reloj, se cuenta con manejadores digitales de reloj (DCM), los cuales tiene la capacidad de sintetizar señales de reloj a partir de una señal de reloj en su entrada. A la salida de un DCM pueden tenerse señales de frecuencia doble, invertida, desfasada 90, 180 ó 270 grados o multiplicada por algún factor con respecto a la señal de entrada.

El FPGA utilizado en el diseño del controlador de SDRAM del presente trabajo es un Spartan 3 (XC3S1500) de Xilinx®, el cual tiene las siguientes características:

- 1.5 millones de compuertas.
- 3328 CLB's (Configurable Logic Block).
- 4 DCM's (Digital Clock Manager).
- Hasta 481 pines configurables como Entrada/Salida.
- 576kbits de BRAM.

La memoria utilizada en el sistema de almacenamiento es una memoria de 256Mb, específicamente la memoria MT48LC16M16A2 de Micron, las características principales del controlador son las siguientes:

- El acceso a la memoria se hace una localidad a la vez (longitud de ráfaga = 1).
- La latencia es de 2 ciclos de reloj (CL = 2).
- Se utilizan dos de los cuatro bancos disponibles; uno por cada memoria (memoria de escritura y memoria de lectura). Se hace uso solo de una parte del banco, 1MB en cada banco.
- La frecuencia de trabajo es de 30 MHz.
- Se atiende a cinco clientes; escrituras hechas por USB, escrituras hechas por el decodificador, lecturas hechas por USB, lecturas hechas por el codificador y petición de refresco.
- La programación se realizó utilizando VHDL.

El módulo controlador de RAM es un diseño jerárquico que a su vez está integrado por varios módulos. Los módulos incluidos en el diseño jerárquico son básicamente dos máquinas de estados finitos las cuales controlan la secuencia de inicialización de la memoria y la secuencia para operaciones de lectura y escritura así como módulos para generación de direcciones, comandos y banderas utilizadas para señalizar el estado de la memoria. La figura 5.2 muestra un diagrama a bloques del módulo controlador de RAM y la interconexión de los módulos en su interior. Cada uno de los módulos es descrito a continuación. Aunque no se observan conexiones de las líneas ***ClkUSB*** y ***Reset***, éstas están conectadas a cada uno de los módulos; ***ClkUSB*** es el oscilador principal del sistema y ***Reset*** es la señal de reinicio general.

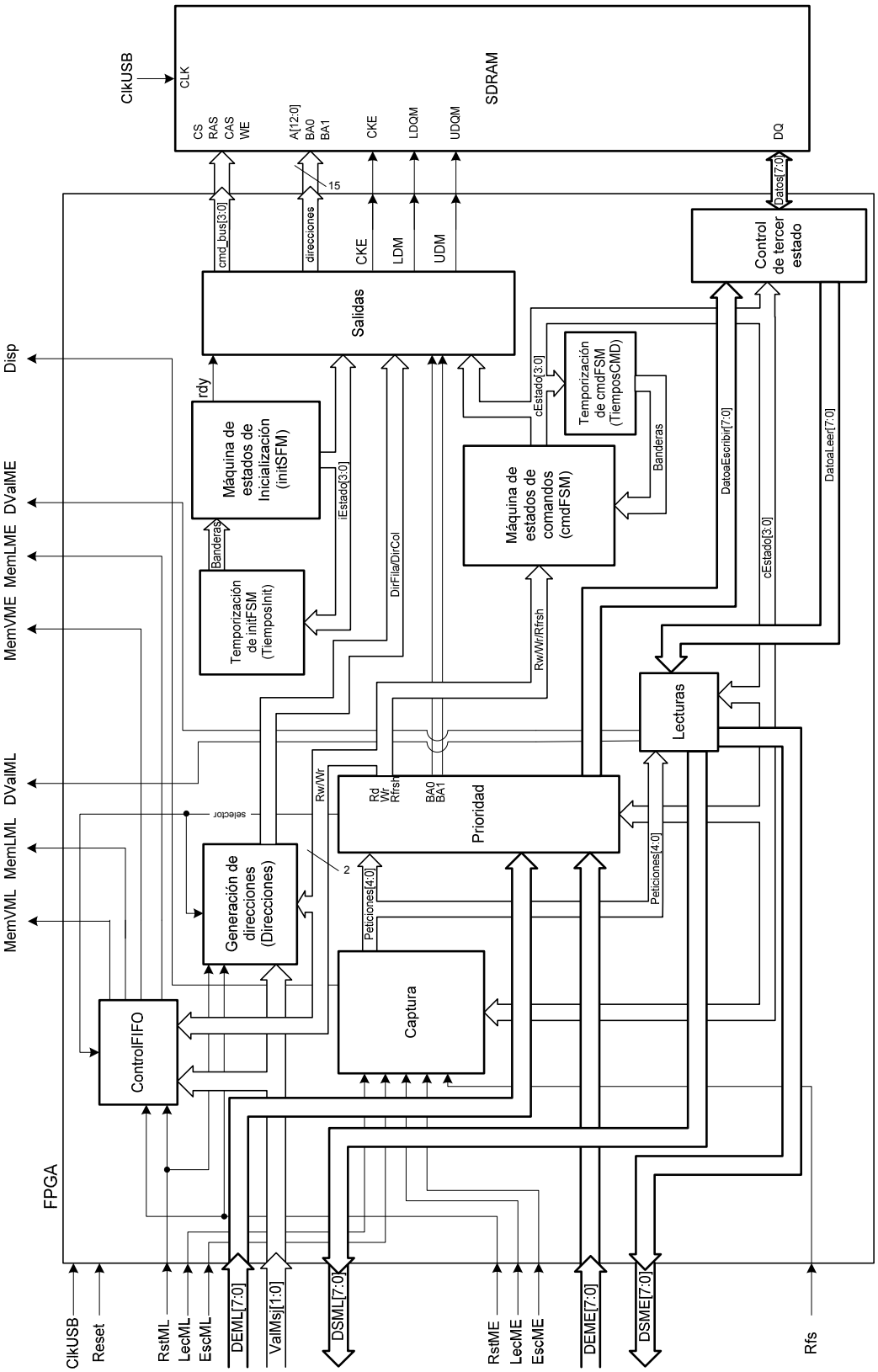


Figura 5.2 Diagrama de bloques del controlador de SDRAM.

5.2 Módulos InitFSM y TiemposInit

Estos módulos son los encargados de realizar la secuencia de inicialización de la memoria. El control de esta secuencia es llevado principalmente por la máquina de estados de inicialización (***initFSM***) cuyo diagrama se muestra a continuación:

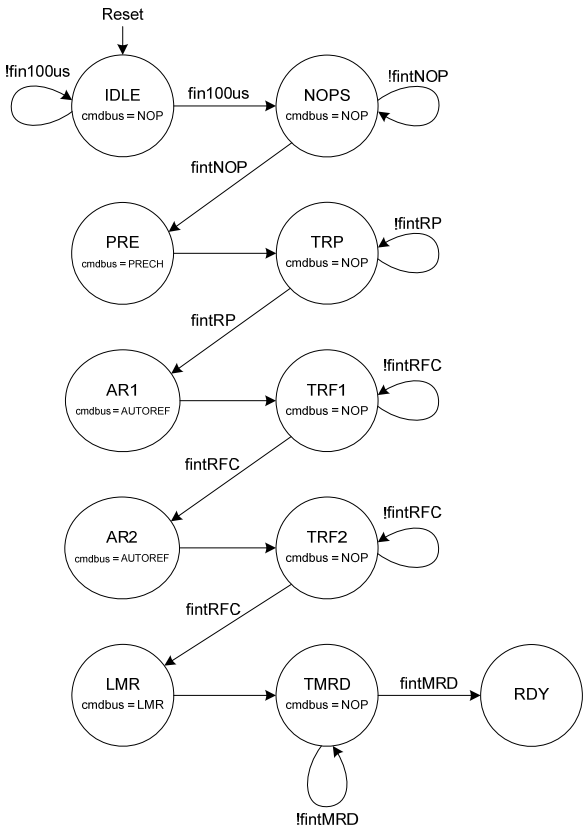


Figura 5.3 Diagrama de estados de la máquina de estados de inicialización (*initFSM*).

Esta máquina gestiona los estados necesarios para la configuración del modo de trabajo de la memoria. Las entradas de esta máquina son las ***banderas*** para temporización y las salidas son un conjunto de señales, de las cuales una es utilizada para indicar que la inicialización se ha realizado (***rdy***) y el resto codifican el estado actual de la máquina de estados mediante 4 bits (***iEstado[3:0]***). Específicamente, el registro de modo se programa en el estado ***LMR***. La memoria está lista para usarse cuando la máquina de estados de inicialización ha llegado al estado ***RDY***.

La generación de tiempos durante estados de espera (IDLE, NOPS, TRP, TRF1, TRF2 y TMRD) se realiza mediante el módulo ***TiemposInit*** el cual está integrado por varios contadores que dependen del estado actual de la máquina de estados y tienen una bandera para indicar cuando una cuenta específica se ha alcanzado mientras la máquina está en un estado de espera; por ejemplo, para la generación de 100µs de espera en estado IDLE antes de pasar al estado NOPS existe un contador que solo cuenta mientras la máquina de estados está en el estado *IDLE* y al llegar a la cuenta con la que se hace un periodo de 100µs, pone la bandera ***fint100us*** para indicar que el tiempo de espera se ha cumplido y que es posible pasar al siguiente estado. Existe un contador como éste por cada tiempo de espera que necesario en la máquina ***initFSM***. El diagrama de flujo de la figura 5.4 muestra la lógica utilizada para cada uno de los contadores utilizados para temporizar los estados de espera, los tiempos de espera pueden ser de uno o varios ciclos de reloj.

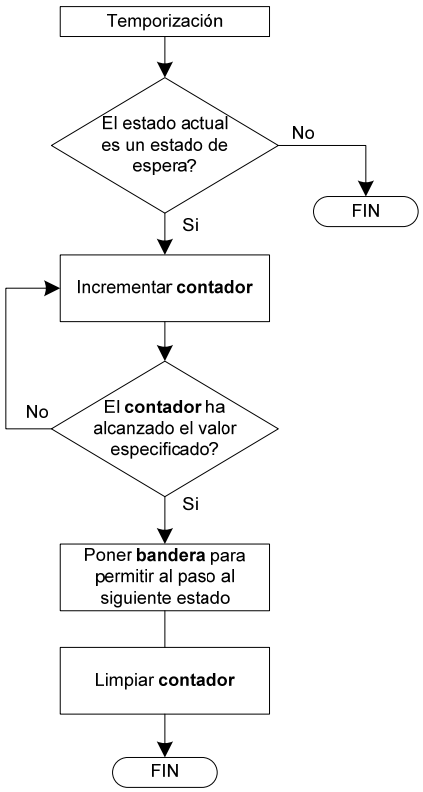


Figura 5.4 Diagrama de flujo de la lógica de los temporizadores.

5.3 Módulos cmdFSM y TiemposCMD

Las peticiones de lectura, escritura y refresco disparan la máquina de estados para comandos ***cmdFSM*** la cual comprende varios estados arreglados en tres lazos (figura 5.5), cada uno lleva el control de una petición. Las entradas a la máquina de estados son las ***banderas*** de temporización y las señales ***Rd***, ***Wr*** y ***Rfrsh*** las cuales disparan a la máquina para realizar la secuencia de estados correspondiente; las salidas son cuatro señales que codifican el estado actual de la máquina (***cEstado[3:0]***). Los tiempos en los estados de espera (tWR, rDATA, tRP, tRP1 y tRFC) son controlados por el módulo ***TiemposCMD***, el cual es similar al módulo ***TiemposInit*** y la lógica de temporización es idéntica a la de la figura 5.4.

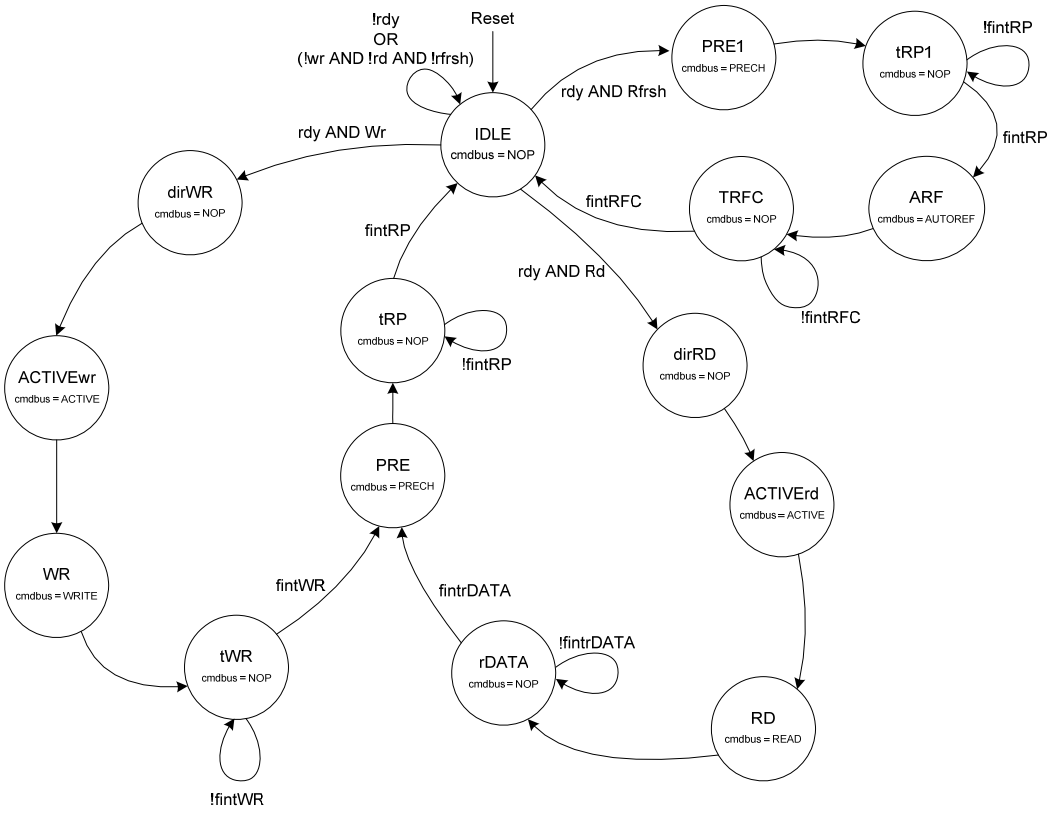


Figura 5.5 Diagrama de estados de la máquina *cmdFSM*.

Cada uno de los lazos en el diagrama de estados de ***cmdFSM*** está dedicado a una petición. Las tres secuencias, correspondientes a las operaciones de escritura, lectura y refresco, se muestran en los siguientes diagramas de tiempo.

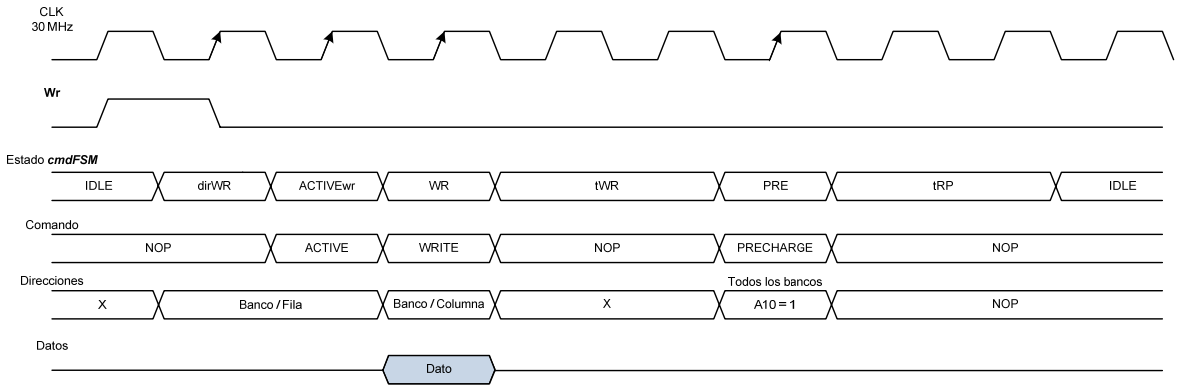


Figura 5.6 Diagrama de tiempo para operaciones de escritura.

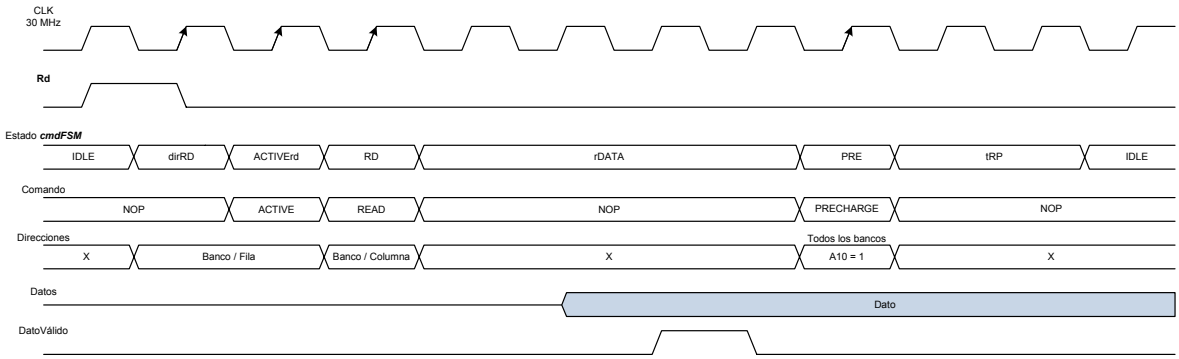


Figura 5.7 Diagrama de tiempo para operaciones de lectura.

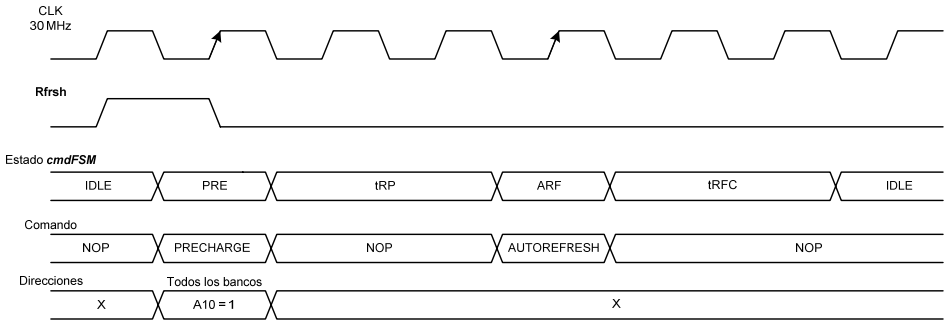


Figura 5.8 Diagrama de tiempo para operaciones de refresco.

5.4 Módulos Captura y Prioridad

Debido a que existen cinco posibles peticiones distintas que atender, es necesario saber cuál o cuáles de las peticiones, en caso de que se registre más de una petición a la vez, debe atenderse. El módulo de ***Captura*** dentro del módulo principal es el encargado de obtener y almacenar las peticiones que se realizan en un determinado instante, este módulo depende del estado actual de la máquina de estados de comandos (***cEstado[3:0]***). Para entender cómo se realiza esto, es útil la figura 5.9 en la que se muestra el modo en que las peticiones se registran.

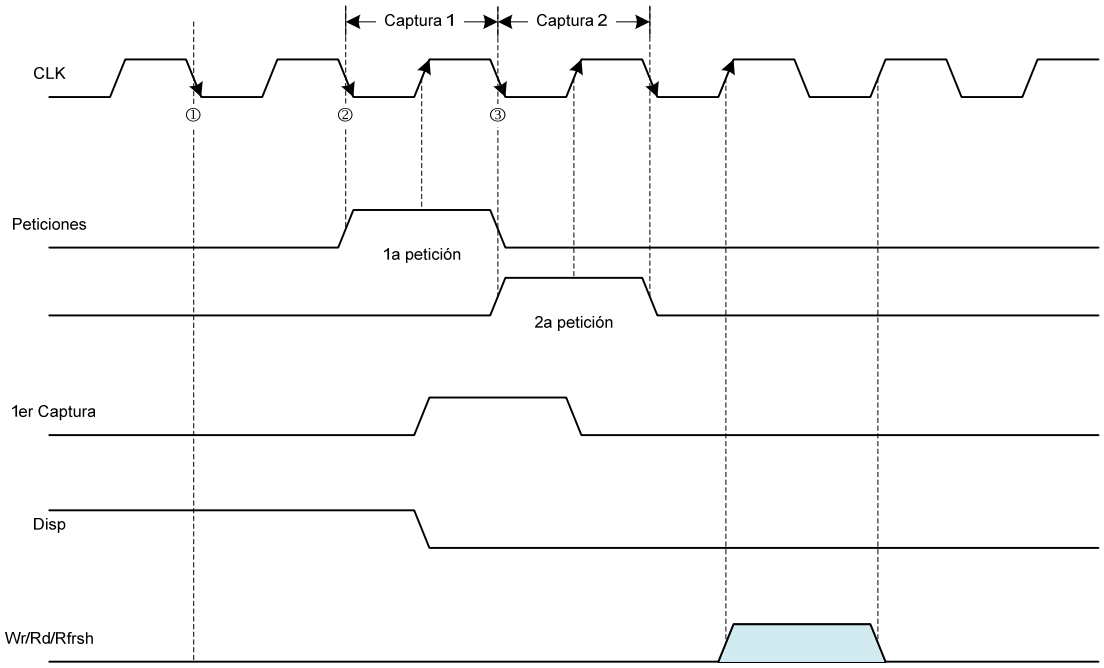


Figura 5.9 Módulo de captura de las peticiones de acceso a memoria.

Como se observa en la figura, todas las peticiones se registran en la transición negativa del reloj del sistema (***CLK***). Toda petición se genera en un módulo externo al módulo controlador de SDRAM y siempre se utiliza un ciclo de reloj para determinar si la memoria se encuentra disponible (***Disp*** en alto) y de ser así en el próximo ciclo de reloj se genera la petición. Por ejemplo, en la transición ①, ***Disp*** está en alto por lo que en la transición ② se genera la petición. Una vez generada esta primera petición, entre las transiciones ② y ③, se realiza la Captura 1, con lo que se almacenan las peticiones hechas

hasta este instante y se pone en bajo la señal ***Disp***. Es necesaria una Captura 2 ya que cuando se registra la transición ② ***Disp*** aún está en alto y si alguno de los clientes muestreó esta señal en la transición ②, podría generarse otra petición en la transición ③. Como se observa, las dos capturas se realizan en transiciones positivas del reloj. Una vez realizadas estas capturas, las peticiones externas registradas se almacenan en un arreglo de cinco bits (***Peticiones[4:0]***) en el que cada bit representa a cada una de las cinco peticiones posibles, el bit más significativo en este arreglo representa la petición con mayor prioridad.

Tabla 5.1 Prioridad de las peticiones externas.

Petición	Prioridad de atención
Lectura por Codificador (***LecME***)	5
Escritura por Decodificador (***EscML***)	4
Lectura por USB (***LecML***)	3
Escritura por USB (***EscME***)	2
Refresco (***Rfs***)	1

En la transición positiva posterior a la captura 2, se genera solo una de tres posibles peticiones que dispararan la máquina de estados de comandos (***cmdFSM***), estas tres posibles peticiones son: ***Wr***, ***Rd*** y ***Rfrsh***.

La generación de estas tres señales es la tarea del módulo llamado ***Prioridad*** en el cual se analiza el arreglo de bits creado en el módulo ***Captura***. Se genera una petición de escritura (***Wr***), lectura (***Rd***) o refresco (***Rfrsh***) hacia la máquina de estados de comandos (***cmdFSM***) con base en la prioridad de las peticiones almacenadas. Por ejemplo, si en el arreglo de bits se almacenó “11010” esto se interpreta así:

- Hubo una petición de **lectura** por parte del codificador.
- Hubo una petición de **escritura** por parte del decodificador.
- No hubo petición de **lectura** por parte de USB.

- Hubo petición de **escritura** por parte de USB.
- No hubo petición de **refresco**.

Tomando en cuenta las prioridades establecidas en la tabla 5.1, se generaría una petición ***Rd*** para atender la petición de lectura del codificador, el cual tiene la mayor prioridad, posteriormente se generaría una petición ***Wr*** para atender la petición de escritura de decodificador y por último se generaría una petición ***Wr*** para atender la escritura por USB. Estas peticiones solo se generan si el estado actual de ***cmdFSD*** es IDLE.

El módulo ***Prioridad*** también se encarga de direccionar los datos que van a ser escritos en la memoria así como de seleccionar el banco en el que se realizará la operación. Debido a que se tienen dos clientes que pueden escribir datos a la memoria, USB y decodificador, al analizar las prioridades es necesario también determinar de dónde provienen los datos que han de ser escritos a la memoria y seleccionar de entre los buses de entrada ***DEME[7:0]*** y ***DEML[7:0]***. Los datos provenientes de alguno de estos buses se colocan en el bus ***DatoaEscribir[7:0]*** el cual a través de un control de tercer estado es conectado al bus bidireccional de la memoria SDRAM.

5.5 Módulo *Lecturas*

Este módulo se encarga de direccionar los datos leídos, es un módulo similar a ***Prioridad***. En este caso se tiene también dos posibles clientes; las lecturas hechas por USB (***DSML[7:0]***) y las lecturas hechas por el codificador (***DSME[7:0]***), a los cuales se les puede asignar los datos provenientes de la memoria mediante el bus ***DatoaLeer[7:0]*** proveniente del control de tercer estado del bus bidireccional de la SDRAM. Además de hacer esta asignación, este módulo también genera las señales ***DValME*** y ***DValML*** las cuales son útiles a la lógica externa a este módulo para saber cuándo el dato que se va a leer es válido. Al igual que los módulos ***Prioridad*** y ***Captura***, este módulo también es dependiente del estado actual de la máquina de estados de comandos (***cEstado[7:0]***).

5.6 Módulos ControlFIFO y Direcciones

Estos dos módulos están relacionados con el manejo de las direcciones y conjuntamente dotan a las memorias de la propiedad de comportarse como colas FIFO, mientras que el módulo ***Direcciones*** se encarga de generar las direcciones de columna (***DirCol***)y fila (***DirFila***) para cualquier operación a la memoria, el módulo ***ControlFIFO*** se encarga de llevar el control del espacio disponible en la memoria así como de poner las banderas de vacío y lleno de las dos memorias del sistema de almacenamiento (***MemVML***, ***MemLML***, ***MemVME*** y ***MemLME***).

Para generar las direcciones de fila y columna se utilizan contadores de 20 bits en los cual los primeros 9 bits sirven para direccionar 512 columnas (2^9 = 512) y los 11 bits restantes direccionan 2048 filas (2^{11} = 2048). Así, el arreglo de 2048 filas por 512 columnas da un total de 1048576 localidades (1MB). Para la memoria de escritura se utilizan dos contadores, uno para generar direcciones de escritura y otro para generar direcciones de lectura. Para la memoria de lectura se utiliza también un contador para generar direcciones de lectura y además se utilizan dos contadores para generar direcciones de escritura; uno de estos contadores es provisional ya que utilizando las señales para validar el mensaje escrito (***ValMsj[1:0]***) se pueden invalidar todas las escrituras correspondientes a un mensaje y por consiguiente estas direcciones deben seguir estando disponibles para escritura. La función principal de este contador provisional es la de generar las direcciones mientras se escriben los datos pero una vez que se valida o invalida el mensaje, pueden suceder dos cosas; si el mensaje es válido, el valor del contador provisional se almacena en un contador permanente el cual llevará el control de los mensaje escritos a la memoria y que fueron válidos, si el mensaje no es válido el contador provisional almacenará el valor del contador permanente, de esta forma las últimas direcciones que se utilizaron para escritura y no se validaron siguen estando disponibles.

Cada uno de los contadores se incrementa al registrarse ya sea una petición de lectura (***Rd***) o escritura (***Wr***), esto determina si el contador que se incrementa es uno relacionado con direcciones de lectura o escritura. Para definir en qué memoria se realiza la operación se utiliza la señal ***Selector*** proveniente del módulo ***Prioridad*** la cual está

directamente relacionada con las señales de dirección de banco ***BA0*** y ***BA1***. Mediante las señales ***RstML***, ***RstME*** y ***Reset*** pueden limpiarse los contadores relacionados con la memoria de lectura, escritura o ambas.

El módulo ***ControlFIFO*** lleva el control del espacio disponible en cada uno de las memorias. Al igual que el módulo ***Direcciones***, hace uso de contadores de 20 bits los cuales direccionan un arreglo de 2048 filas por 512 columnas (1MB) y son utilizados para contar el número de localidades disponibles en la memoria. A diferencia de los contadores utilizados en ***Direcciones***, los cuales solo se incrementan con cada petición de escritura o lectura, los contadores que utiliza este módulo se decrementan con cada escritura y se incrementan con cada lectura. Esto, siguiendo la lógica de que un dato escrito ocupa una localidad en la memoria, mientras que una lectura deja disponible una localidad en la memoria. Hay un contador por cada memoria y la señal ***Selector*** determina cual de los dos contadores será afectado por la operación de lectura o escritura, si ***Selector*** = '1' la operación afecta a los contadores relacionados con la memoria de escritura, de lo contrario los contadores afectados son aquellos relacionados con le memoria de lectura. Para las escrituras a memoria de lectura existe también otro contador auxiliar debido a que los mensajes escritos pueden ser válidos o no, este contador se incrementa con las escrituras en cada mensaje, mientras que el contador del espacio total disponible en la memoria se decrementa. El contador auxiliar solo lleva la cuenta de los datos que integran el mensaje que se escribe en un determinado momento. Al validar el mensaje, el contador auxiliar se limpia y el contador de espacio total conserva el valor alcanzado con la última escritura, si el mensaje no es válido, el contador auxiliar se limpiará solo después de haber sumado su valor al contador de espació total; las localidades que ya no estaban disponibles después de haberse escrito un mensaje vuelven a estar disponibles al registrarse este mensaje como no válido.

5.7 Módulo *Salidas*

Este módulo envía a los pines físicos de memoria las señales necesarias para ejecutar cualquier comando válido. A través del bus ***cmd_bus*** se envían las señales CS,

RAS, CAS y WE. En el bus de ***Direcciones*** están integradas las líneas de direcciones A[12:0] utilizadas para seleccionar fila/columna y las líneas BA0 y BA1 para selección de banco, el módulo también controla las líneas CKE, UDM y LDM. En todo instante el comando enviado y el valor en las líneas de direcciones depende del estado actual de las máquinas de estado ***iEstado[3:0]*** y ***cEstado[3:0]***. La tabla 5.2 muestra el estado de las señales de direcciones y comandos correspondientes a cada estado de la máquina ***initFSM***. La señal ***rdy*** proveniente de la máquina de estados de inicialización se pone en '1' cuando la máquina ha llegado al estado RDY indicando que la inicialización y configuración de la memoria ya se ha realizado y está lista para ejecutar cualquier comando válido. En el estado RDY la máquina de estados ***initFSM*** cede el control a la máquina de estados para comandos ***cmdFSM***. En la tabla 5.2 no se muestra ningún valor para las líneas de comando y direcciones correspondientes al estado RDY, debido a que en este estado el control de estas señales lo tiene la máquina ***cmdFSM***.

Tabla 5.2 Valor de las salidas según el estado actual de las máquinas de estado.

Estado de ***initFSM***	***Direcciones***	***cmd_bus***	***CS***	***RAS***	***CAS***	***WE***	***CKE***	***UDM***	***LDM***
IDLE	-	NOP	L	H	H	H	L	L	L
NOPS	-	NOP	L	H	H	H	H	L	L
PRE	A10 = '1'	PRECHARGE	L	L	H	L	H	L	L
TRP	-	NOP	L	H	H	H	H	L	L
AR1	-	AUTOREFRESH	L	L	L	H	H	L	L
TRF1	-	NOP	L	H	H	H	H	L	L
AR2	-	AUTOREFRESH	L	L	L	H	H	L	L
TRF2	-	NOP	L	H	H	H	H	L	L
LMR	Código	LOAD MODE REGISTER	L	L	L	L	H	L	L
TMRD	-	NOP	L	H	H	H	H	L	L
RDY	-	-	-				H	L	L

Tabla 5.2 (continuación) Valor de las salidas según el estado actual de las máquinas de estado.

Estado de *cmdFSM*	**Direcciones**	***Cmd_bus***	***CS***	***RAS***	***CAS***	***WE***	***CKE***	***UDM***	***LDM***
IDLE	-	NOP	L	H	H	H	L	L	L
dirWR	AddrFila	NOP	L	H	H	H	H	L	L
ACTIVEwr	AddrFila / Banco	ACTIVE	L	L	H	H	H	L	L
WR	AddrCol	WRITE	L	H	L	L	H	L	L
tWR	-	NOP	L	H	H	H	H	L	L
PRE	A10 = ‘1’	PRECHARGE	L	L	H	L	H	L	L
tRP	-	NOP	L	H	H	H	H	L	L
dirRD	AddrFila	NOP	L	H	H	H	H	L	L
ACTIVErd	AddrFila / Banco	ACTIVE	L	L	H	H	H	L	L
RD	AddrCol	READ	L	H	L	H	H	L	L
rDATA	-	NOP	L	H	H	H	H	L	L
PRECH1	A10 = ‘1’	PRECHARGE	L	L	H	L	H	L	L
tRP1	-	NOP	L	H	H	H	H	L	L
ARF	-	AUTOREFRESH	L	L	L	H	H	L	L
tRFC	-	NOP	L	H	H	H	H	L	L

Capítulo 6 Integración y pruebas

6.1 Introducción

La interconexión del dispositivo USB (basado en el controlador EZ-USB) y los módulos implementados en el FPGA es fundamental para el funcionamiento del módulo de comunicación USB-Fibra óptica. Aunque algunos de los módulos implementados en el FPGA no conciernen al presente trabajo, se hace mención de ellos ya que son parte importante también del módulo MCUF como un solo ente funcional, sin la inclusión de estos módulos (Decodificador, Codificador, Selector de memoria, etc), los dos módulos tratados en este trabajo (Dispositivo USB y controlador de SDRAM) aún integrados en un mismo sistema, no tendrían la misma utilidad para solucionar las necesidades específicas en el remplazo del sistema de adquisición de datos de la central de Laguna Verde. Sin embargo, estos dos módulos (Dispositivo USB y Controlador de SDRAM) podrían servir a alguna otra aplicación ya sea trabajando conjuntamente o de forma independiente; cada uno de estos módulos puede integrarse a sistemas que requieran ya sea comunicación USB o acceso a una memoria SDRAM.

6.2 Integración

El diagrama de la figura 6.1 muestra la integración de los módulos funcionales que conforman al MCUF. El módulo correspondiente a la comunicación USB (controlador EZ-USB) es fácilmente distinguible en el diagrama de bloques ya que es un solo circuito integrado. Sin embargo, dado que el módulo controlador de SDRAM está implementado en el mismo FPGA en el que se implementaron los módulos de codificación, decodificación, selección de memoria y otros, no es fácil distinguir dicho módulo. Para facilitar la visualización de este módulo como otro bloque más en el diagrama de la figura 6.1 se hace uso de las señales que están involucradas con el controlador de SDRAM, las cuales son diferenciables porque están en negritas y letra cursiva; por ejemplo ***MemVML***.

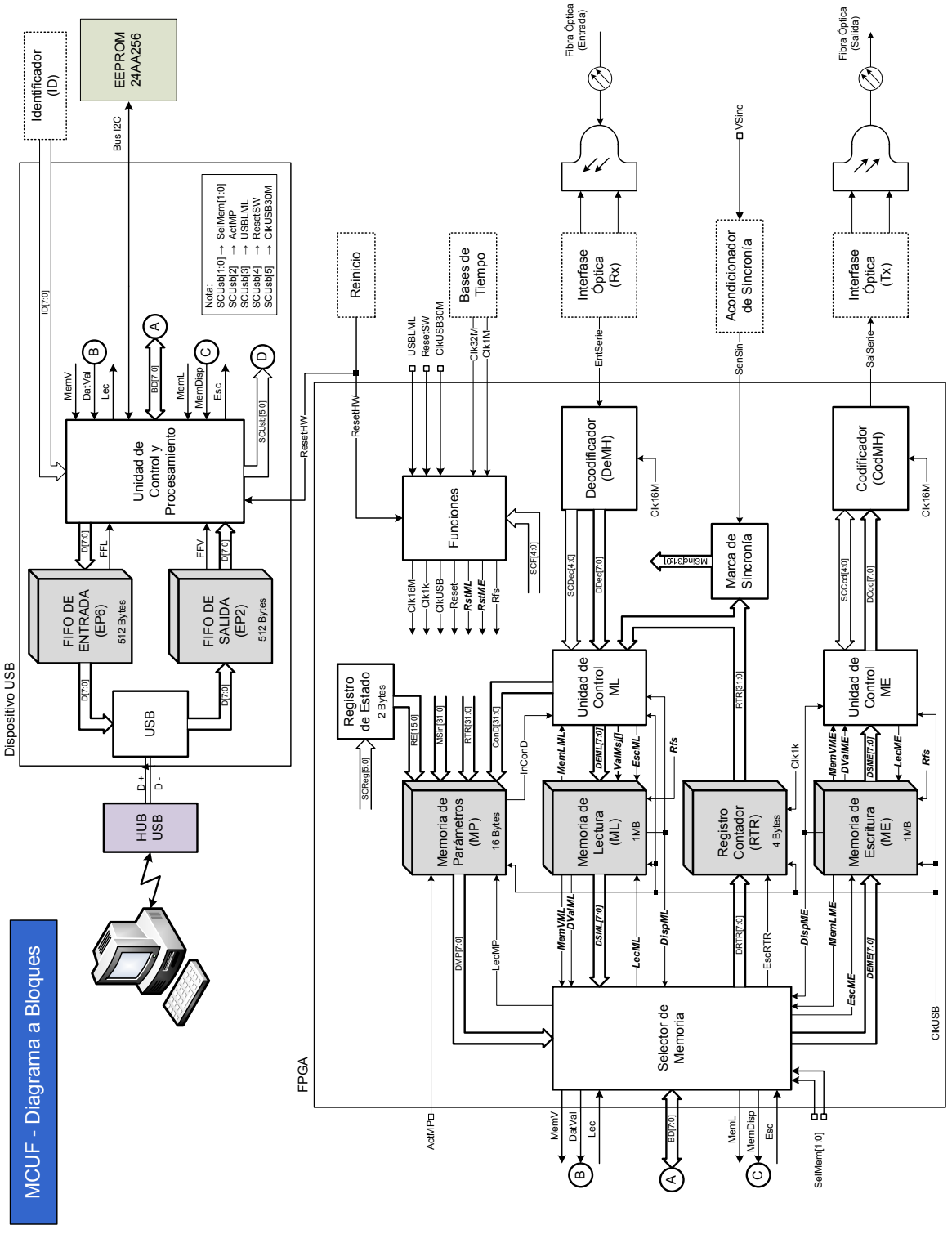


Figura 6.1 Diagrama a bloques del módulo de comunicación USB-Fibra óptica (MCUF).

Para enfatizar la presencia del controlador de SDRAM en la figura 6.1, las mismas señales que se marcan en la figura 6.1 se marcan también en la figura 6.2 que presenta un diagrama simplificado del controlador de SDRAM y en el que también se presenta que aunque se han mencionado dos memorias, Memoria de Escritura (ME) y Memoria de Lectura (ML), estas se encuentran dentro de un mismo chip pues se aprovecha la forma de organización de la memoria SDRAM en bancos. Así, el banco 0 se utiliza para implementar la memoria de escritura y el banco 1 para la memoria de lectura. Cada una de las señales que interconectan al dispositivo USB con la lógica implementada en el FPGA se presenta en la tabla 6.1 especificando la dirección, es decir; si la señal es enviada del EZ-USB al FPGA o en dirección contraria, asi como la función que cumple en todo el sistema.

Tabla 6.1 Señales que interconectan al EZ-USB con la lógica implementada en el FPGA.

Señal	Dirección	Función
MemV	FPGA → EZ-USB	Es la bandera de Vacío de la memoria, si está puesta en '1' la interfase GPIF no puede leer de la memoria actualmente seleccionada.
DatVal	FPGA → EZ-USB	Después de que se ha ejecutado un comando de lectura, esta señal indica el momento en que la interfase GPIF puede tomar el dato.
Lec	EZ-USB → FPGA	Esta señal es la petición de lectura por parte de USB, es uno de los cinco clientes que atiende el controlador de SDRAM.
BD[7:0]	Bidireccional	Éste es el bus de datos de la interfase GPIF.
MemL	FPGA → EZ-USB	Es la bandera de Lleno de la memoria, si está puesta en '1' la interfase GPIF no puede escribir a la memoria actualmente seleccionada.
MemDisp	FPGA → EZ-USB	Esta señal indica si la memoria está disponible para cualquier acceso. Ya sea lectura o escritura por USB, si MemDisp está en '0' la interfase GPIF no puede realizar ninguna operación.
Esc	EZ-USB → FPGA	Esta señal es la petición de escritura por parte de USB, es uno de los cinco clientes que atiende el controlador de SDRAM.

Tabla 6.1 (continuación) Señales que interconectan al EZ-USB con la lógica implementada en el FPGA.

Señal	Dirección	Función
SelMem[1:0]	EZ-USB → FPGA	Estas dos señales sirven para definir en qué memoria se realizará la operación (Escritura/Lectura); hay cuatro posibles valores: 00: Memoria de Lectura (ML). 01: Registro contador (RTR). 10: Memoria de parámetros (MP). 11: Memoria de Escritura (ME).
ActMP	EZ-USB → FPGA	Esta señal se utiliza, al enviar un comando ***Recibe*** y sirve para reiniciar el contador de datos almacenados en ML, este contador está almacenado en Memoria de Parámetros (MP).
USBLML	EZ-USB → FPGA	Esta señal indica a la lógica en el FPGA que los datos de memoria de lectura (ML) están siendo leídos por USB. Impide que la lógica reinicie ML mientras se está leyendo.
ResetSW	EZ-USB → FPGA	Esta señal es enviada a la lógica del FPGA cuando el EZ-USB recibe un comando ***Reset***, reinicializa todos los módulos en el FPGA.
ClkUSB30M	EZ-USB → FPGA	Es la señal de reloj utilizada por algunos de los módulos implementados en el FPGA.

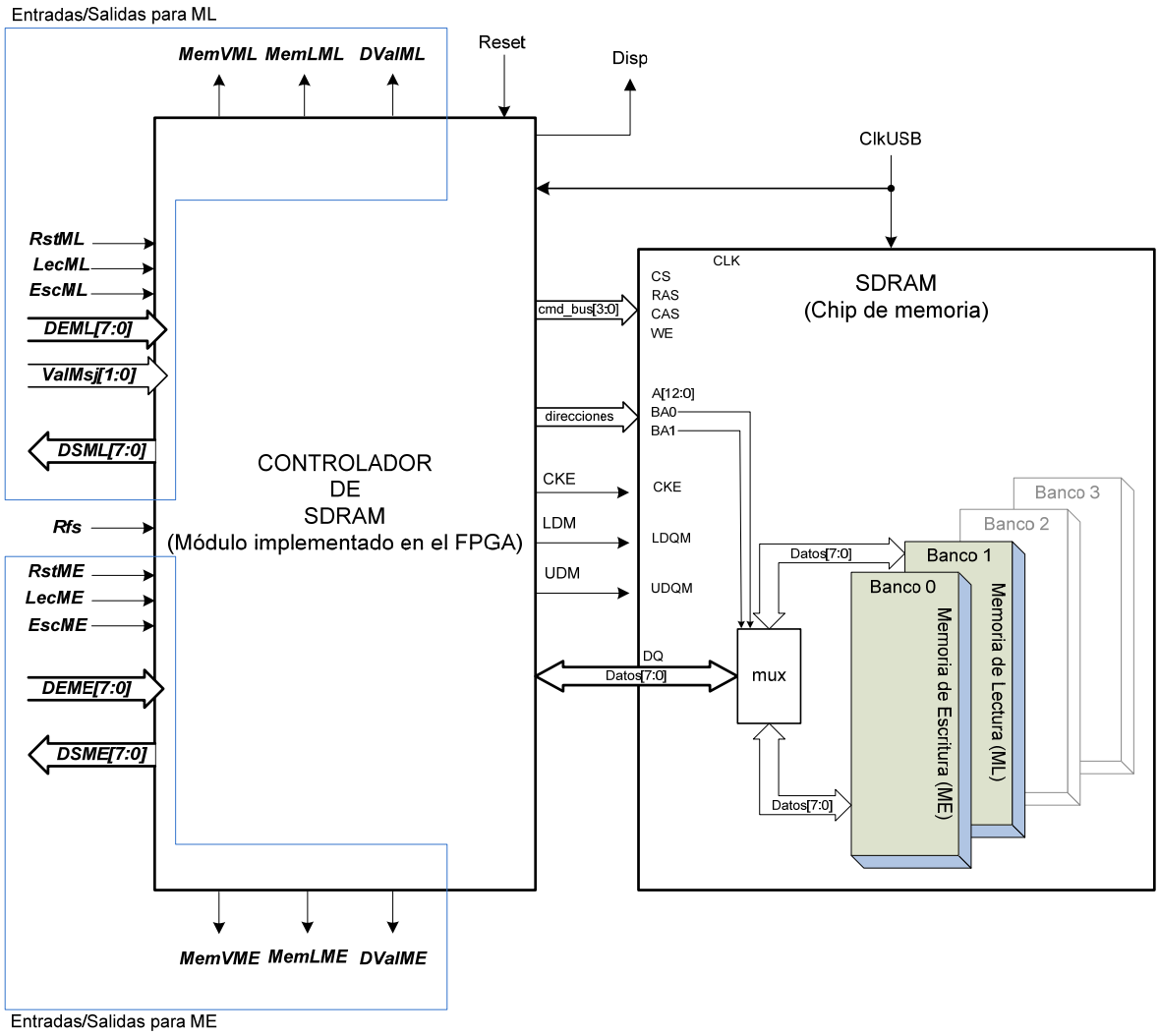


Figura 6.2 Diagrama simplificado del controlador de SDRAM, se muestra también el uso de bancos independientes para cada memoria.

La integración de cada uno de los módulos como se muestran en la figura 6.1 conforman el módulo de comunicación USB-Fibra óptica. Para el desarrollo en hardware se utilizaron dos tarjetas de circuito impreso. En una de estas tarjetas se encuentra el controlador EZ-USB, la memoria EEPROM en la que se almacena el *firmware* del controlador USB, los circuitos necesarios para adecuar la señal de sincronía externa, la interfase electro-óptica, los interruptores para identificación y los conectores para interfase con la lógica implementada en el FPGA. También se hace uso de una tarjeta que

integra al FPGA Spartan3, la memoria SDRAM que se utiliza en el sistema de almacenamiento, la memoria Flash que almacena la configuración de FPGA así como la interfase JTAG por medio de la cual se programa la memoria Flash. Estas dos tarjetas se presentan en las figuras 6.3 y 6.4, el módulo de comunicación USB-Fibra óptica completo se muestra en la figura 6.5.

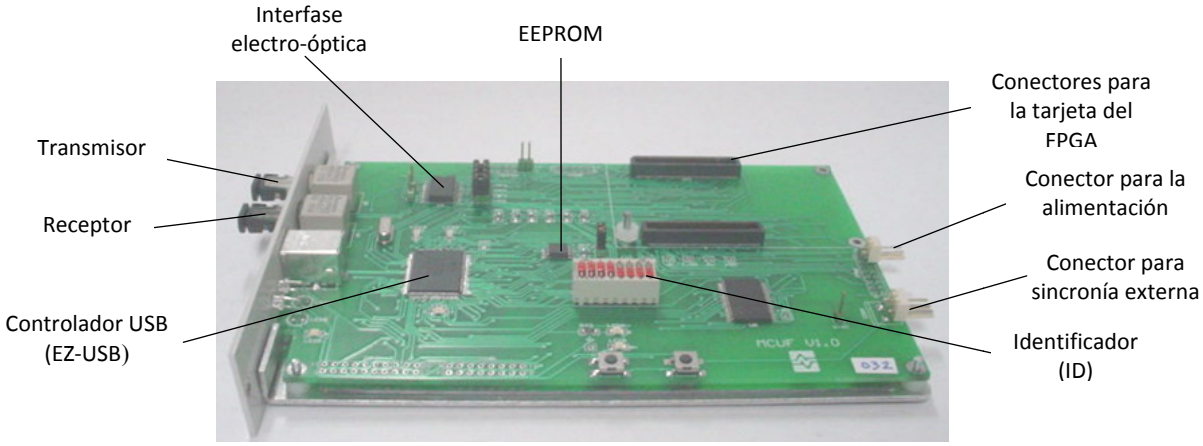


Figura 6.3 Tarjeta de circuito impreso de la interfase USB y electro- óptica.

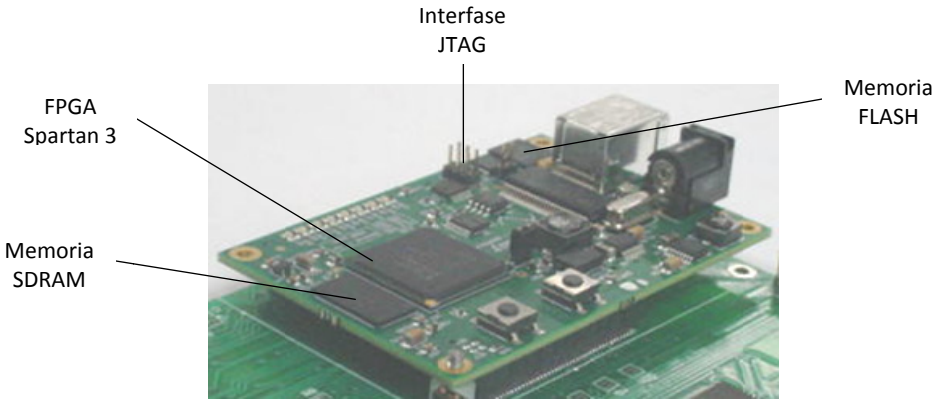


Figura 6.4 Tarjeta utilizada en el módulo de comunicación USB-Fibra óptica, integra al FPGA y la memoria SDRAM.

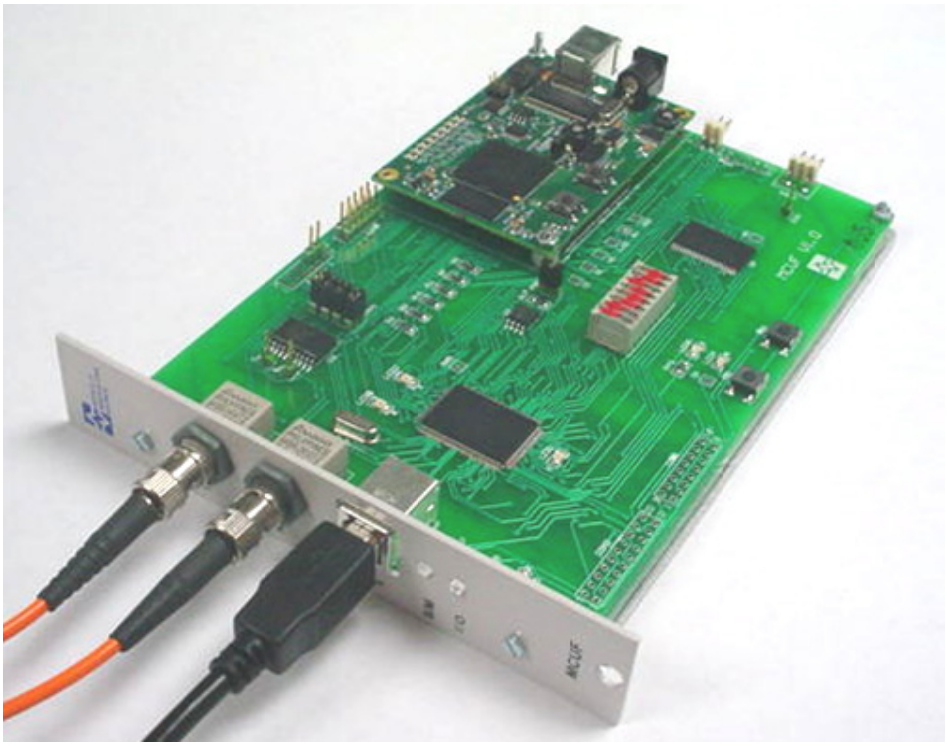


Figura 6.5 Módulo de comunicación USB-Fibra óptica.

6.3 Pruebas realizadas al módulo

Para comprobar el funcionamiento del módulo MCUF se realizaron pruebas de envío y recepción de datos utilizando el software CyConsole proporcionado por el fabricante del controlador USB el cual permite comunicarse con el dispositivo EZ-USB, así como el editor de archivos hexadecimales WinHex para crear los archivos que se enviaron (comandos), estos archivos son:

- Sincroniza.hex: Archivo de 8 bytes correspondientes al comando ***Sincroniza*** (0x*00030004*11223344). Carga el valor 0x11223344 en el registro contador (RTR).

- Transmite128.hex: Archivo de 132 bytes correspondientes al comando ***Transmite*** (0x*00020080*000102…7F) seguido de 128 bytes (valores 0x00 a 0x7F) que serán enviados por medio de fibra óptica.

- Recibe.hex: Archivo de 2 bytes correspondientes al comando ***Recibe*** (0x0001).

En la consola de la interfase con el EZ-USB se pueden seleccionar cualquiera de los dos endpoints disponibles para transmisión y recepción de datos y con el control *File Trans…* se puede seleccionar el comando que se desea enviar o el destino (archivo.hex) de la información que se reciba en caso de un comando ***Recibe***, lo cual es muy útil en las pruebas realizadas ya que se aprovecha esta opción para guardar los datos recibidos en uno o varios archivos .hex para después compararlos con el archivo original enviado. La figura 6.6 muestra la ventana de la aplicación utilizada en las pruebas, se resaltan los controles utilizados.

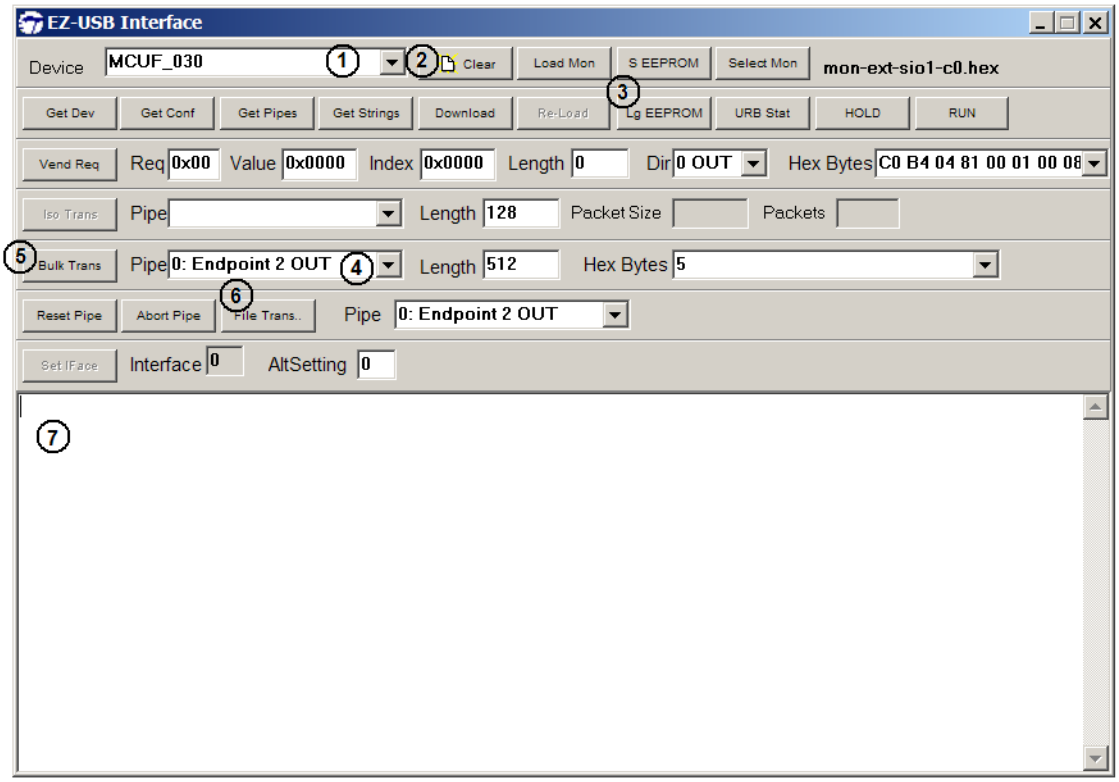


Figura 6.6 Ventana de la aplicación CyConsole utilizada para las pruebas.

① Control de selección del dispositivo al que se desea enviar datos. Éste es utilizado en caso de que se conecten más de módulo a la PC.

② Botón *Clear* utilizado para limpiar el listado de eventos que se registran en la ventana.

③ Botón *Lg EEPROM* utilizado para cargar el firmware del controlador USB en la memoria EEPROM conectada la bus I2C del EZ-USB.

④ Control de selección del endpoint a utilizar.

⑤ Botón *Bulk Trans* utilizado para efectuar transferencias tipo Bulk.

⑥ Botón *File Trans…* utilizado para abrir la ventana de selección del archivo que se desea enviar.

⑦ Zona de visualización de mensajes. Aquí se despliegan las respuestas a los comandos.

6.3.1 Prueba con el comando *Sincroniza*

Esta prueba consistió en enviar el comando ***Sincroniza*** el cual carga el valor 0x11223344 en el registro contador (RTR) del módulo y después se envió un comando ***Recibe*** y al leer su respuesta se verificó la carga del registro contador. La prueba se realizó de la siguiente forma:

1. Se envió el archivo Sincroniza.hex utilizando la aplicación CyConsole.
2. Se leyó la respuesta correspondiente al comando.

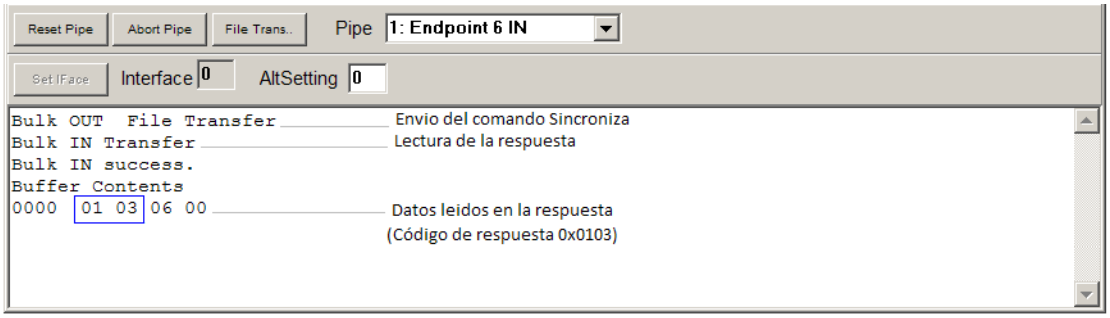


Figura 6.7 Mensajes desplegados al enviar el comando *Sincroniza*.

3. Se envió el comando ***Recibe***. Puesto que no hay ningún dato almacenado en la memoria hasta ahora, solo se leyeron 16 bytes (información de parámetros del módulo) entre los que se encuentra el valor actual del registro contador.

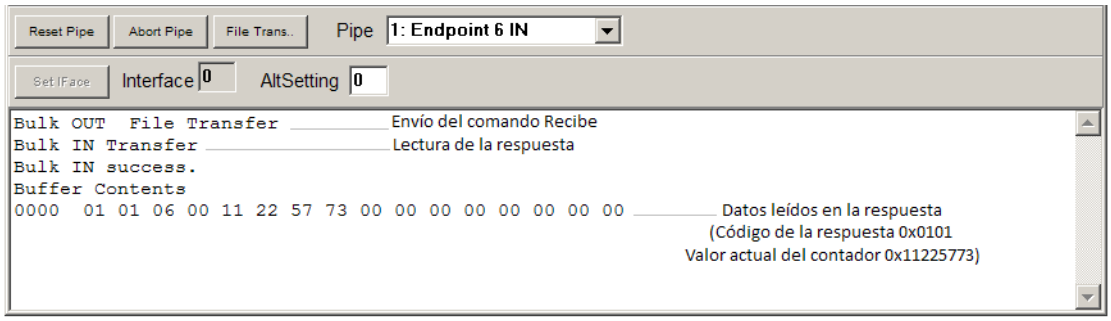


Figura 6.8 Mensajes desplegados al enviar el comando *Recibe*.

El valor del contador leído (0x11225773) al enviar el comando ***Recibe*** no corresponde exactamente al valor cargado (0x11223344) mediante el comando ***Sincroniza*** debido a que transcurrió cierto tiempo entre la carga del contador y la lectura del mismo, dando oportunidad al contador de incrementar su valor.

Mediante el comando ***Sincroniza*** se escriben 4 bytes en el registro contador el cual está implementado en el mismo FPGA en el que se implemento el controlador de SDRAM. Las escrituras al registro contador son realizadas utilizando el GPIF, las señales de habilitación de escritura, y los dos bits menos significativos del dato a escribir (Dato[1] y Dato[0]) se muestran en la figura 6.9, también se muestra la señal de petición de refresco la cual se pone en alto cada 3.76 µs para efectuar un comando AUTOREFRESH en la SDRAM.

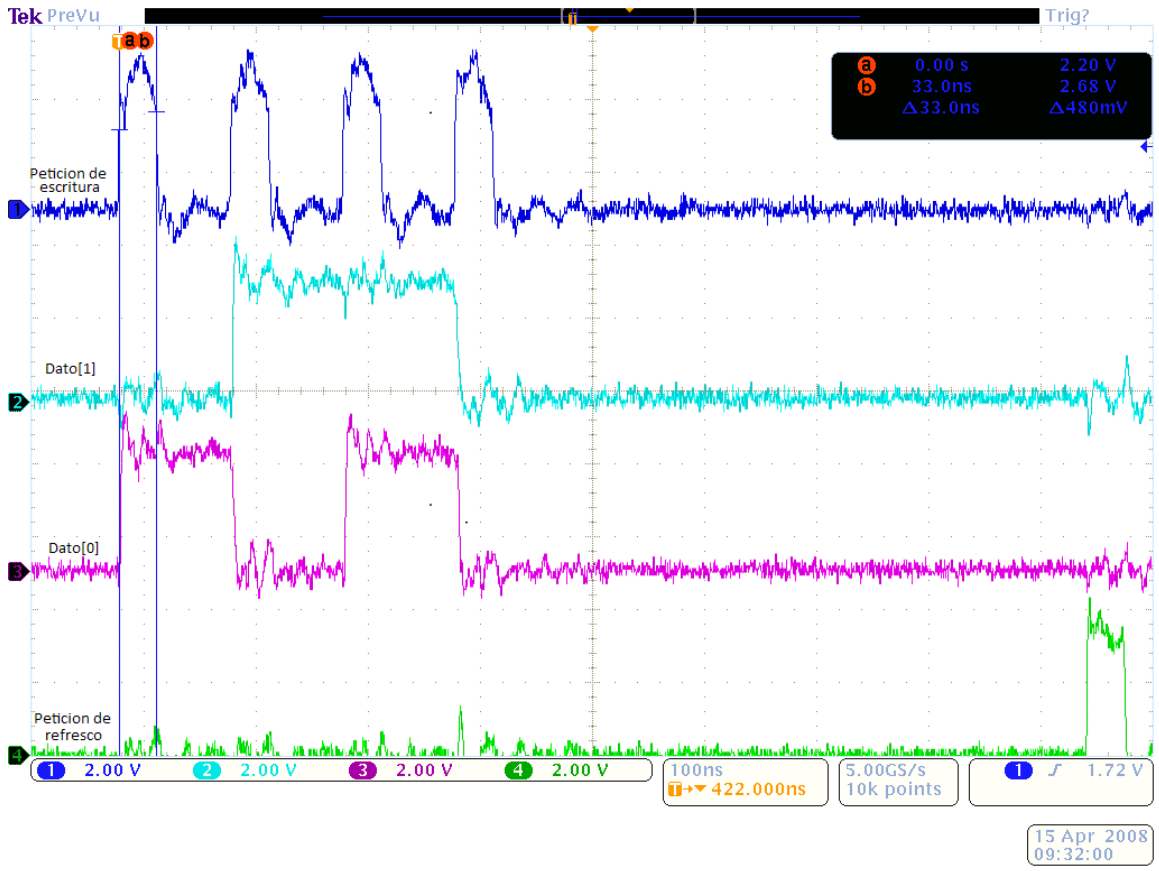


Figura 6.9 Señales observadas en el osciloscopio al enviar el comando *Sincroniza*.

6.3.2 Prueba con los comandos *Transmite/Recibe*

El objeto de esta prueba fue determinar si los datos se escriben y leen correctamente en cada una de las memorias del módulo (ME y ML). Para esto, se realizó la conexión entre los conectores Tx y Rx, de tal forma que los datos enviados por la PC son escritos en ME, posteriormente leídos de la misma memoria por el módulo codificador y enviados por fibra óptica. Mediante la conexión de Tx y Rx los mismos datos son recibidos y escritos por el decodificador en ML. Una vez almacenados en ML pueden ser leídos al enviar un comando ***Recibe*** desde la PC.

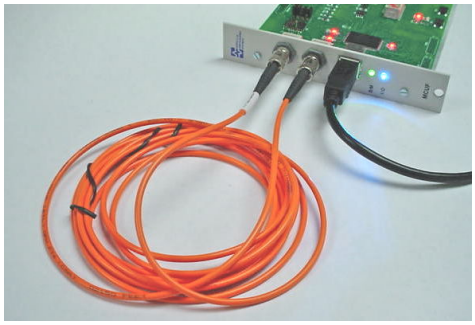


Figura 6.10 Lazo de retroalimentación para realizar pruebas con los comandos *Transmite*/*Recibe*.

Para esta prueba se realizó lo siguiente:

1. Se conecto el cable de fibra óptica a modo de crear el lazo de retroalimentación (conexión Tx-Rx).
2. Se envió el comando ***Transmite*** (archivo Transmite128.hex) al módulo para transmitir 128 bytes (secuencia 00,01…,7F).
3. Se leyó la respuesta del comando.

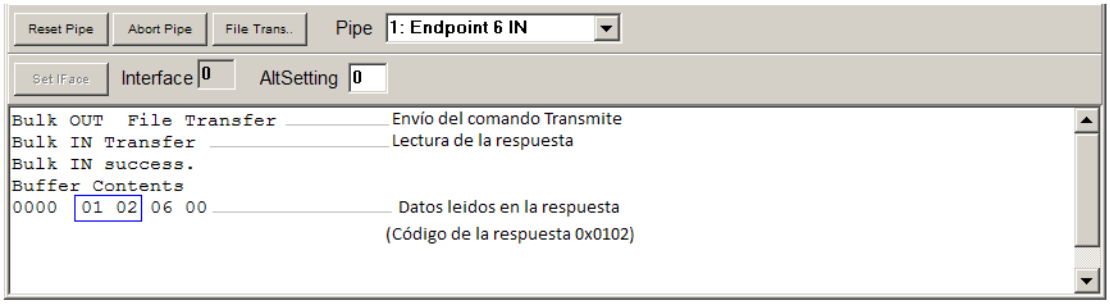


Figura 6.11 Mensajes desplegados al enviar el comando *Transmite*.

4. Se envió el comando ***Recibe*** para recuperar los datos enviados.
5. Se leyó la respuesta la cual incluye los datos almacenados.

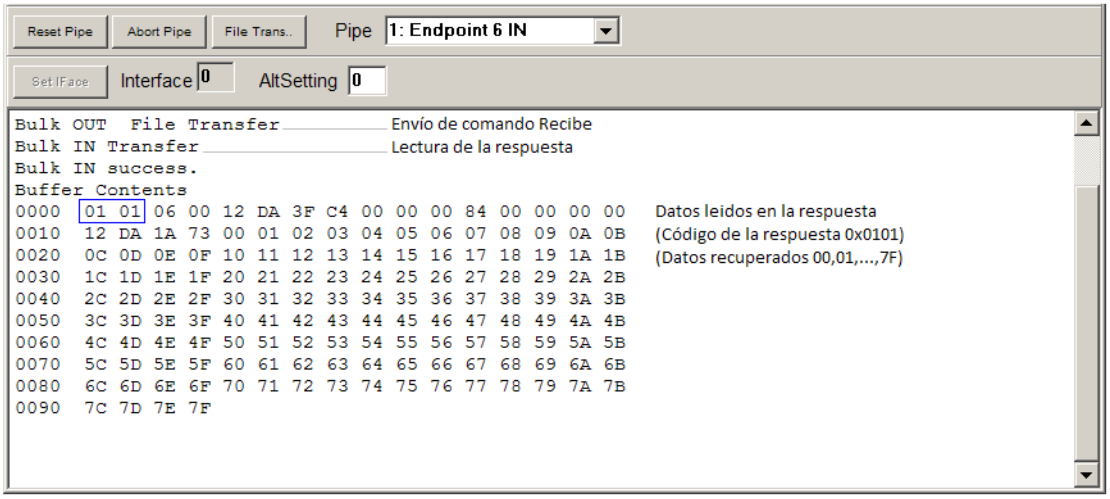


Figura 6.12 Mensajes desplegados al enviar el comando *Recibe*.

Los datos recibidos incluyen el código de respuesta del comando ***Recibe*** (0x0101) y 14 bytes mas de parámetros del módulo los cuales incluyen el número de bytes que estaban almacenados en ML y que han sido leídos (0x00000084 ó 132). Al recibir un mensaje por fibra óptica y ser codificado, siempre se escribe una estampa de tiempo de cuatro bytes antes del mensaje, esto bytes contienen el valor del registro contador al momento de ser recibido el mensaje, por lo que el número de bytes leídos no son 128 sino 132 pues se incluye la estampa de tiempo. En la figura 6.12 se observa esta estampa de tiempo antes del mensaje de 128 bytes recibidos, en este caso el valor es 0x12DA1A73.

Al realizar esta prueba, se realizan operaciones tanto de escritura y lectura en las dos memorias que conforman el sistema de almacenamiento. A continuación se presentan imágenes de señales observadas utilizando un osciloscopio, todas las imágenes son para unas transferencia de 32 datos (secuencia 00,01,…,1F), se eligió este número de datos ya que esto permite la fácil visualización de las señales. Primeramente, los datos son escritos en la memoria de escritura (ME), en la figura 6.13 se muestran las peticiones de escritura que se hacen por medio de la interfase GPIF, ***EscME***, la señal interna del controlador de SDRAM ***Wr*** que dispara la máquina de estados ***cmdFSM***, el estado actual de dicha

máquina y el bit menos significativo del dato que se escribe en la memoria de escritura ***DEME[0]***. La figura 6.14 muestra a detalle estas mismas señales, debido a que el estado actual de ***cmdFSM*** se codifica con 4 bits y solo se cuentan con cuatro canales en el osciloscopio, la señal que representa el estado actual de ***cmdFSM*** se pone en alto solo en los estados ACTIVEWr y tWR, el estado WR intermedio se presenta como un nivel bajo.

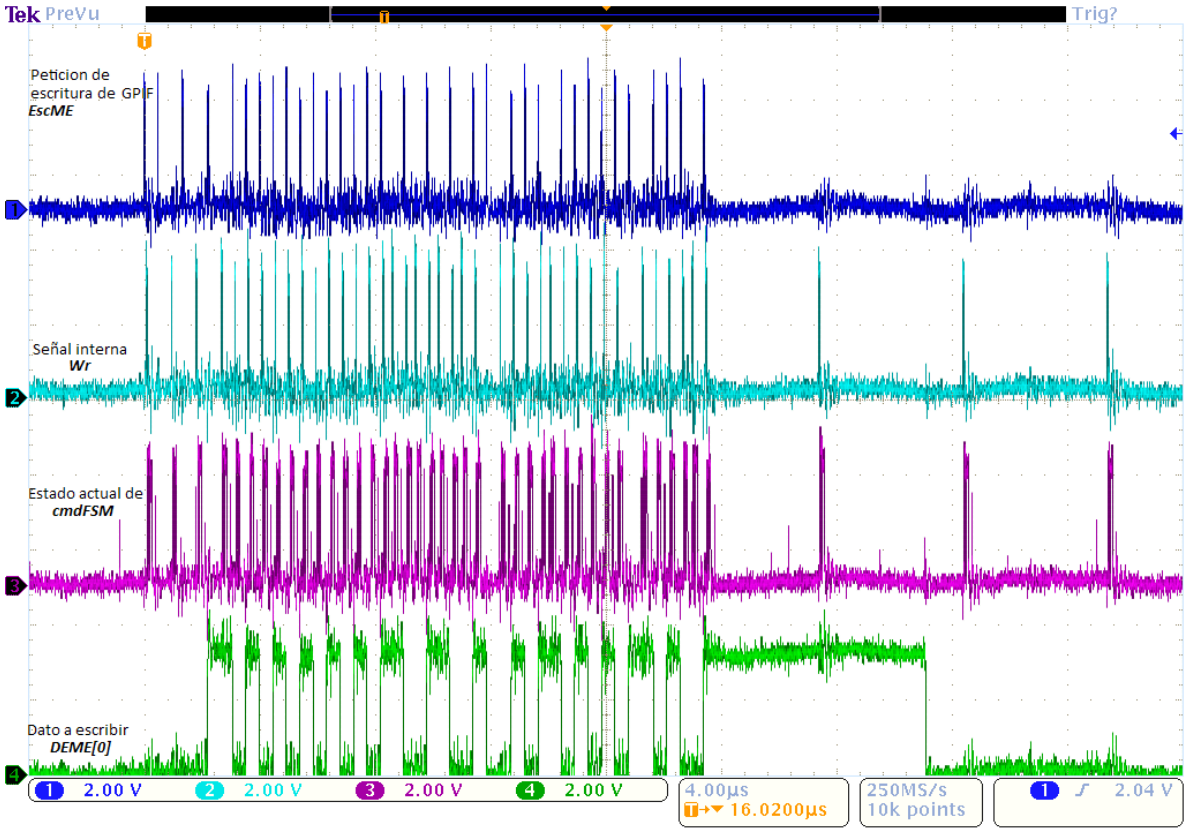


Figura 6.13 Señales observadas al enviar 32 bytes mediante el comando *Transmite*.

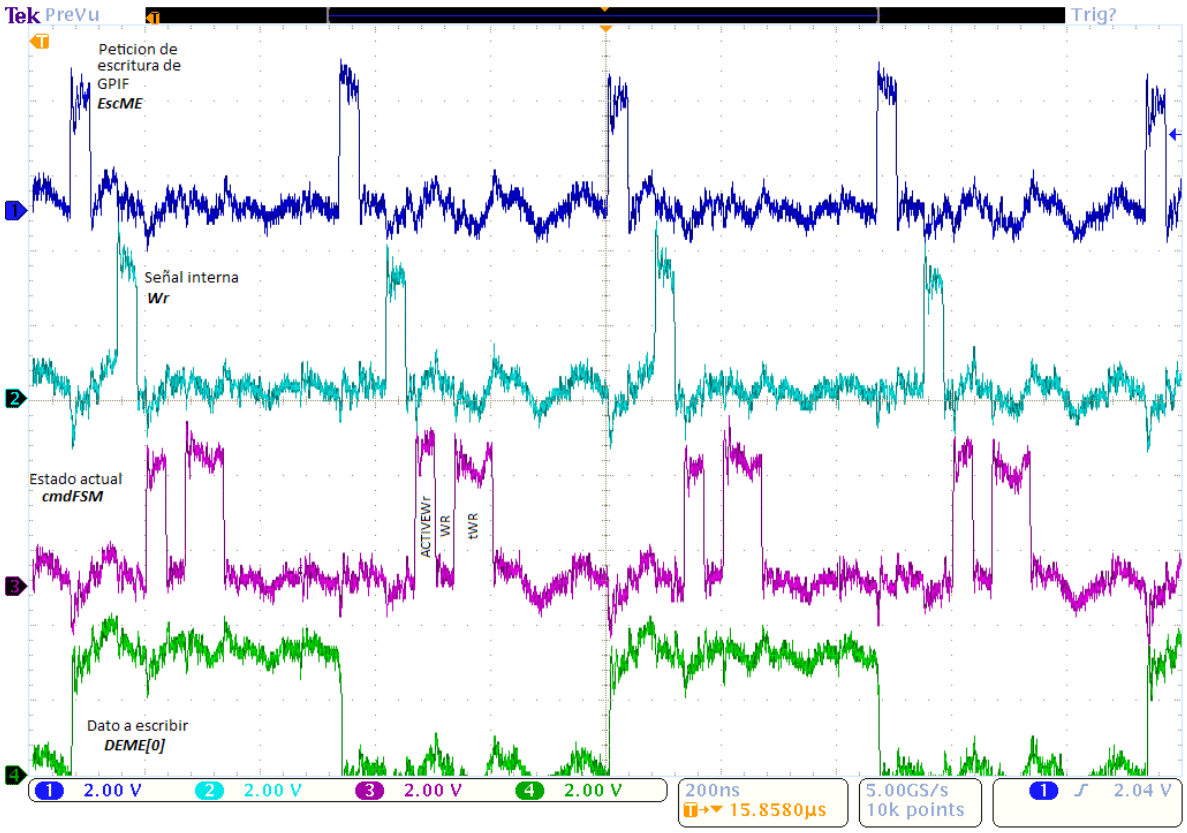


Figura 6.14 Detalle de las escrituras hechas a ME mediante el comando *Transmite*.

Una vez que los datos se han almacenado en la memoria de escritura (ME) son leídos por el codificador para posteriormente ser enviados de forma serial a través de fibra óptica. El envío por fibra óptica de un dato le toma al codificador al rededor de 5µs por lo que también hay un periodo de aproximadamente 5µs entre lecturas realizadas por el codificador. La figura 6.15 muestra las lecturas hechas por el codificador, se observan las señales ***LecME***; que es la petición de lectura del codificador, la señal interna ***RD***; que dispara la operación de lectura en **cdmFSM**, el estado actual de ***cmdFSM*** y el bit menos significativo del dato leído ***DSME[0]***. La figura 6.16 muestra el detalle de una operación de lectura en la que se observa la señal de dato válido ***DValME*** con la que el controlador de SDRAM indica el momento en que el dato se ha leído de la memoria y puede ser tomado por el codificador, en la figura se aprecia el tiempo de retardo CAS Latency.

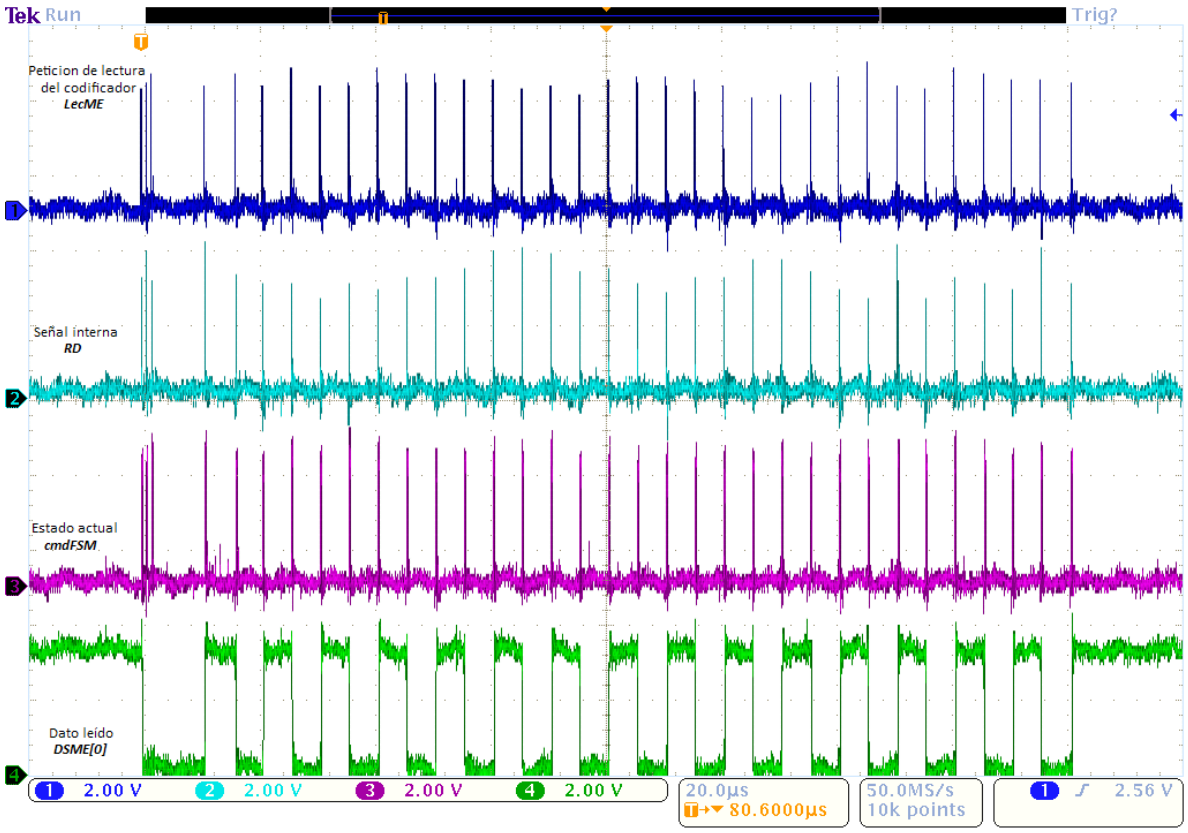


Figura 6.15 Señales observadas de los accesos del Codificador (lecturas) a la memoria de escritura.

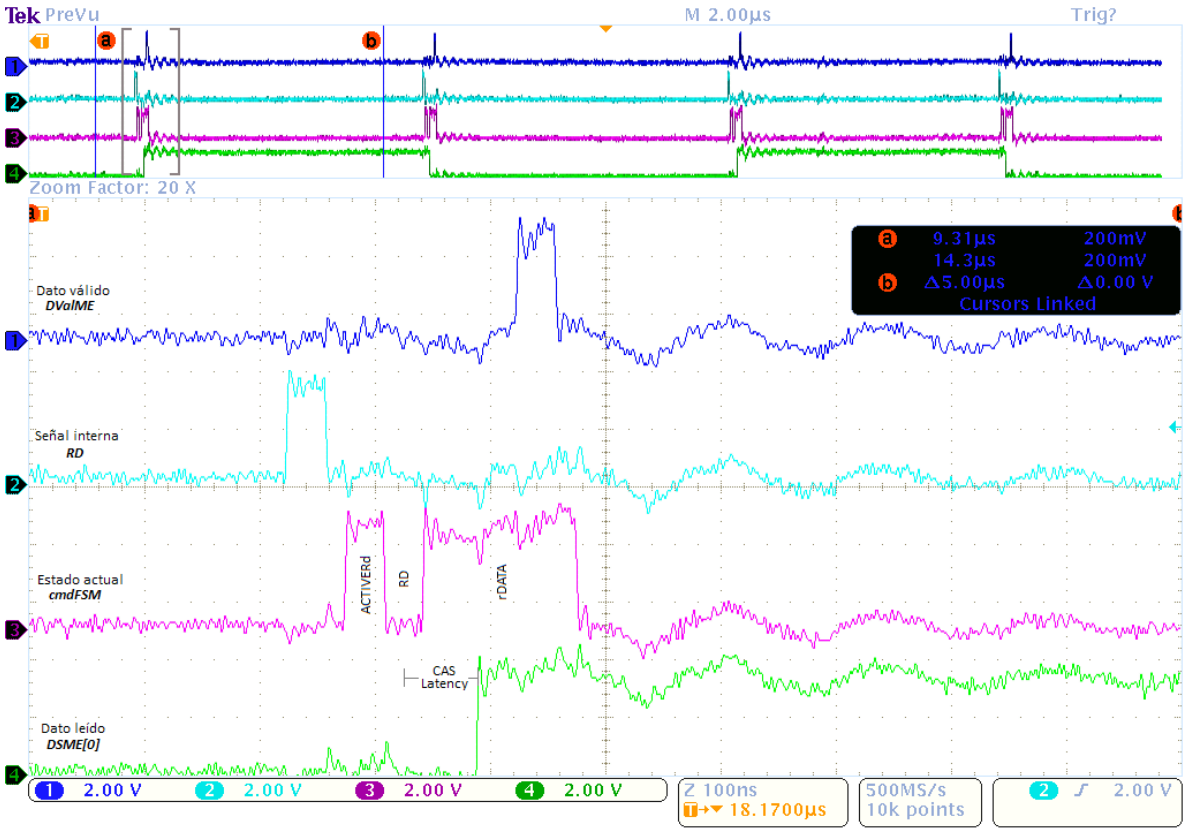


Figura 6.16 Detalle de las lecturas realizadas por el Codificador.

Los datos codificados se envían por fibra óptica para posteriormente ser decodificados y escritos por el decodificador en la memoria de lectura (ML). En la figura 6.17 se muestran estas escrituras hechas por el decodificador, las señales visibles son: la petición de escritura del decodificador ***EscML***, la señal interna ***WR***, el estado actual de ***cmdFSM*** y el bit menos significativo del dato a escribir ***DEML[0]***. Se observan cuatro peticiones iníciales que se realizan de forma más rápida que las restantes, estas corresponden a los cuatro bytes de estampa de tiempo que se añaden a cada mensaje recibido, además de esto se observa también que antes de realizarse cualquier petición de escritura por parte del decodificador, la señal interna ***WR*** y el estado actual de ***cmdFSM*** varían aún sin registrarse peticiones del decodificador, esto es debido a que la señal interna ***WR*** también se genera para atender las peticiones de escritura realizadas

mediante GPIF (***EscME***), estos cambios en ***WR*** y el estado actual de ***cmdFSM*** intercalados con las peticiones del decodificador corresponden a las escrituras realizadas al enviar un comando ***Transmite***.

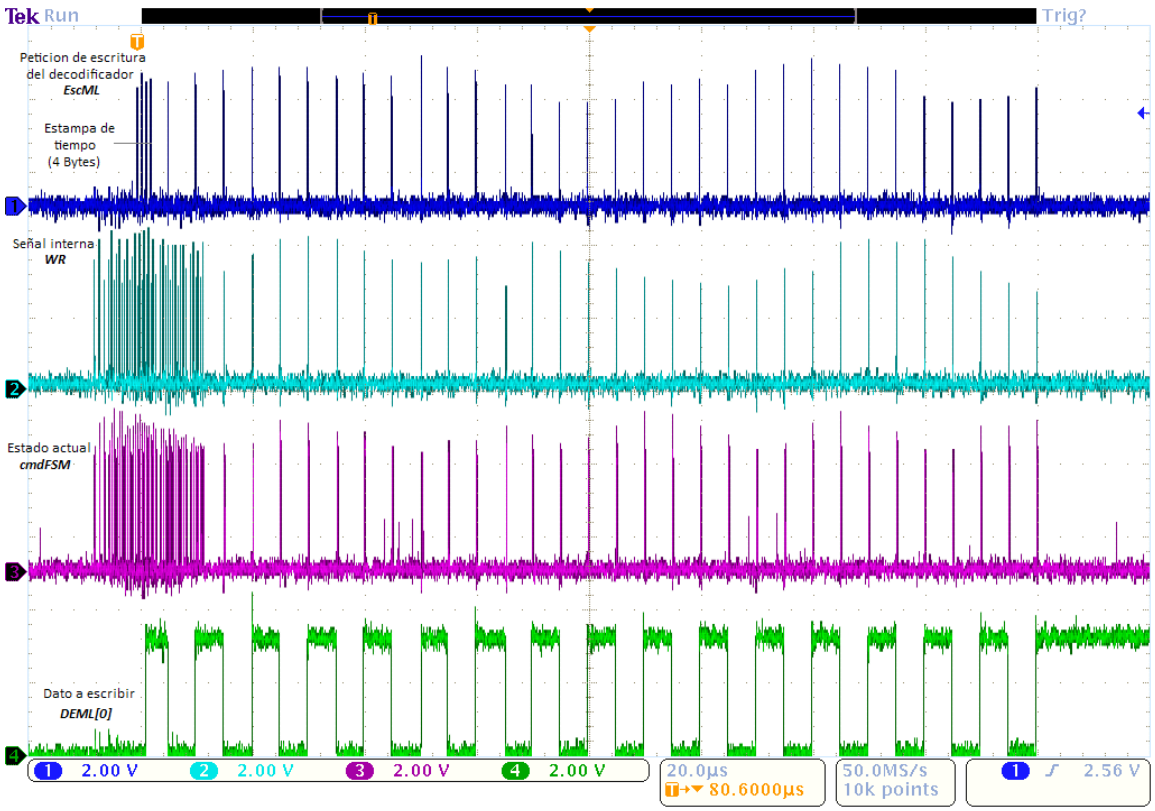


Figura 6.17 Señales observadas de los accesos del Decodificador (escrituras) a la memoria de lectura.

Los datos que se almacenaron en la memoria de lectura (ML) son recuperados al enviar un comando ***Recibe***, con lo cual se dispara la forma de onda de lectura de GPIF. Como de observa en la figura 6.18, antes de acceder a los datos de la memoria ML se leen 16 bytes de parámetros de la memoria de parámetros (MP), estos datos incluyen el número de datos que se deben leer posteriormente de la memoria de lectura. En la figura 6.18 no se observan cambios en las señales ***RD***, estado actual de ***cmdFSM*** y dato leído ***DSML[0]*** correspondientes a las primeras 16 peticiones; esto es debido a que estas peticiones son para leer MP, la cual es un registro implementado en el FPGA y no se

encuentra en la memoria SDRAM, por tanto no hay una petición interna de lectura ***RD*** y no se dispara la máquina de estados ***cmdFSM***. El detalle de una de las peticiones posteriores con las que si se recuperan datos de ML se muestra en la figura 6.19, las señales que se observan son: ***DValML***; que indica a la interfase GPIF el momento en que el dato leído puede ser tomado, la señal interna ***RD***; la cual dispara las operaciones de lectura en la máquina ***cmdFSM***, también se presenta el estado actual de dicha máquina.

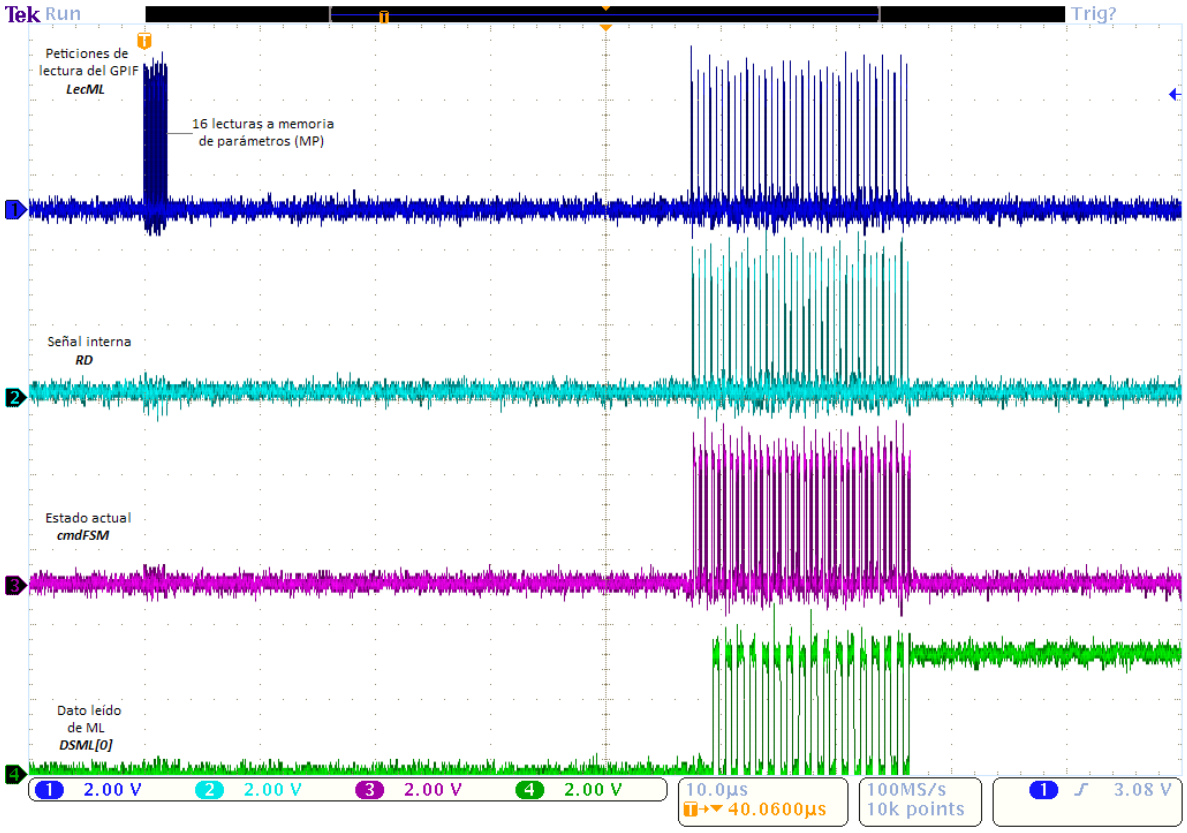


Figura 6.18 Señales observadas al enviar el comando *Recibe*.

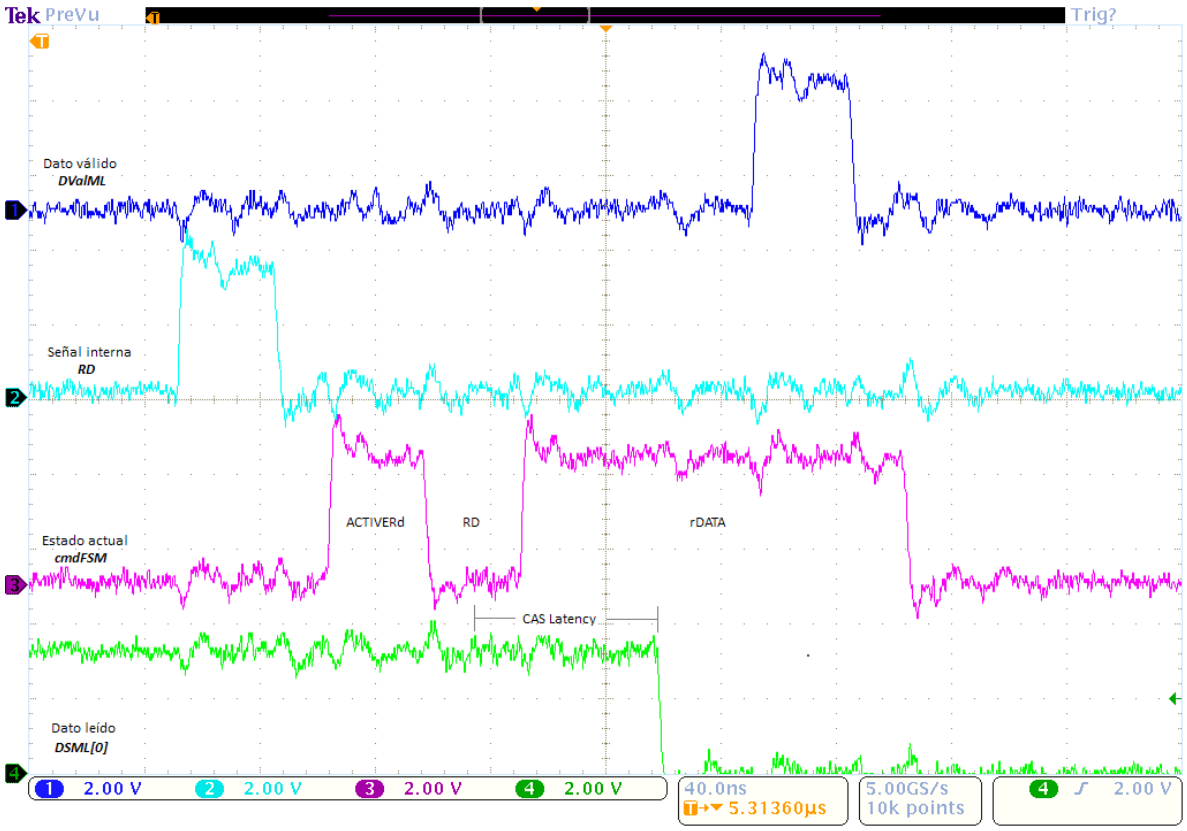


Figura 6.19 Detalles de las lecturas hechas a ML al enviar el comando *Recibe*.

Una prueba más exhaustiva consistió en enviar 60 mensajes de 128 bytes (7680 bytes en total). Debido a que a cada mensaje se le adjunta una estampa de tiempo de 4 bytes al ser recibido, en realidad de recuperan 16 bytes de parámetros del módulo y 7920 bytes de datos al enviar el comando ***Recibe*** después de que los 60 mensajes se han enviado. Al enviar el comando ***Recibe***, los datos leídos se guardaron en archivos .hex (15 archivos de 512 bytes y uno de 240 bytes). Para esta prueba se creó un archivo hexadecimal de 7936 bytes el cual contiene 16 bytes de valor nulo (código de respuesta y parámetros) y 60 mensajes de 128 bytes con su respectiva estampa (valor nulo) cada uno.

Se utilizo la aplicación WinHex para concatenar los archivos de datos recibidos y crear un solo archivo de 7936 bytes el cual se comparó con el archivo de igual tamaño que contiene parámetros y estampas con valores nulos.

Figura 6.20 Comparación entre los datos enviados y los recibidos.

Al comparar los dos archivos solo se encontraron diferencias en los parámetros del módulo leídos y en la estampa de tiempo de cada mensaje, lo cual era esperado pues mientras en un archivo estos datos tienen valores nulos, en el archivo que concatena a todos los archivos leídos con el comando ***Recibe*** estos datos contienen valores reales de los parámetros y las estampas de tiempo. Los datos recibidos correspondientes a mensajes fueron iguales a los datos enviados.

Conclusiones

Durante el desarrollo del presente trabajo se incursionó en dos temas completamente nuevos para mí como pasante de la carrera de Ingeniería en comunicaciones y electrónica, los cuales ahora considero de suma importancia pues debido a la constante evolución de los sistemas electrónicos es necesario mantenerse actualizado en cuanto a las tecnologías que se utilizan en aplicaciones reales. Tal es el caso de la comunicación USB, pues es notoria la casi completa extinción de equipos nuevos que cuenten con las interfases tales como RS-232 o IEEE 1284 (Puerto paralelo) por lo que dichas interfases están siendo remplazadas por el bus USB, además de que este bus se ha vuelto muy popular y está siendo usado en distintos tipos de dispositivos, en parte por la capacidad del bus de conectar varios dispositivos en un mismo puerto y en parte por las altas tasas de transferencia que se pueden alcanzar utilizando USB.

De haberse desarrollado la comunicación con una PC mediante alguno de las interfases antes mencionadas se encontraría con el problema (en muchas ocasiones) de no contar con el equipo necesario para operar el dispositivo, además de que si se utilizara alguna de las interfaces mencionadas, solo podría conectarse un dispositivo por cada puerto. Cabe mencionar también la versatilidad que tiene el controlador USB utilizado (EZ-USB) pues una vez comprendido su funcionamiento y características, es adaptable casi a cualquier dispositivo que se desee dotar de comunicación USB. Por otro lado, el uso de tecnología de lógica programable es también muy importante pues de no haberse implementado el controlador de SDRAM en un dispositivo de lógica programable se hubiera requerido de un dispositivo (un microcontrolador por ejemplo) únicamente dedicado al control de acceso a la memoria, encareciendo el costo del diseño, además de que se tiene la ventaja de que se pueden realizar cualquier cambio en el comportamiento de dicho controlador mediante la actualización de la configuración del FPGA. En el caso de que se utilizara un solo chip para el controlador y éste no pudiera ser actualizado, tendría que desecharse. Simplemente encuentro a los dispositivos de lógica programables casi 'mágicos', pues se puede implementar diversidad de sistemas digitales en estos

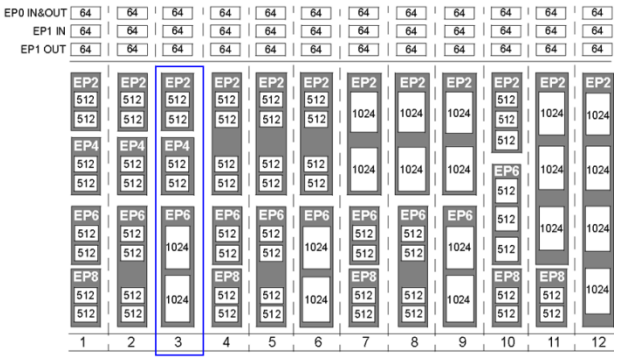
dispositivos, además de que pueden ser actualizados infinidad de veces gracias a que son reconfigurables.

En cuanto al medio de almacenamiento utilizado, aunque no se explotó en su totalidad el potencial que éste posee, considero que abrió el camino para implementar aplicaciones mucho más complejas utilizando este tipo de memoria y además de que, por su parecido con tecnologías aún más nuevas DDR SDRAM y DDR2 SDRAM, el conocimiento adquirido permite implementar aplicaciones con éstas y otras nuevas tecnologías.

Trabajo futuro

El desarrollo del diseño del que trata el presente trabajo, por ser una aplicación real y aplicable, cumple con ciertos requerimientos especificados en documentos internos del IIE. No obstante, como casi cualquier diseño, se le pueden realizarse algunas mejoras entra las que se proponen:

- Implementación de la comunicación USB utilizando endpoints dedicados. En el controlador USB se cuentan con 4 endpoints configurables de los cuales uno puede ser exclusivamente configurado de tamaño 1KB y buffer doble y ser utilizado únicamente para la recepción de los datos leídos de la memoria de lectura. Se utilizarían también dos endpoints de 512 bytes y doble buffer para transferencia de comandos y recepción de respuestas que cuyo tamaño sea pequeño.



Posible arreglo de los endpoints para un diseño mejorado.

- Actualización del controlador de SDRAM para hacer uso del modo ráfaga (BURST) para lectura y escritura con accesos de 2, 4 u 8 localidades por cada disparo de cualquiera de los comandos WRITE o READ. Esta actualización se plantea para un sistema en el que se requiera de mayor velocidad de acceso pues el modo BURST mejora significativamente la velocidad con que se accede a la memoria. Esto se debe principalmente a que a diferencia de las accesos simples en los que todos los periodos de espera tienen que respetarse para cada acceso, en el modo BURST solo el primer acceso es lento, los accesos posteriores son automáticos y solo se tiene que poner o tomar los datos del bus en cada ciclo de reloj subsecuente para realizar las operaciones de escritura o lectura. Para satisfacer los requerimientos en sistemas más veloces también puede incrementarse la frecuencia de reloj a la que trabajan el controlador y la memoria. Para el caso de la memoria utilizada en este trabajo la frecuencia puede llegara a ser de hasta 133 MHz.

- Dotar al controlador de la capacidad para controlar el acceso a memorias con tecnología DDR (Double Data Rate) las cuales son también del tipo SDRAM y permiten los mismos comandos que la memoria utilizada en el presente trabajo. Sin embargo, tiene la peculiaridad de que pueden transferir datos en ambas transiciones de la señal de reloj, haciéndolas dos veces más veloces, de ahí la designación de Double Data Rate o taza doble de transferencia de datos.

Además de las mejoras propuestas es también importante mantenerse al día en cuanto a todos los adelantos relacionados a las tecnologías utilizadas en el presente trabajo. En internet y en la misma página de usb.org ya se habla de una especificación USB 3.0 y los recursos didácticos disponibles relacionados con FPGA's varían de fabricante en fabricante, algunos muestran ejemplos de código desarrollado en Verilog, un lenguaje descriptivo de hardware distinto al utilizado en este trabajo. Esto deja ver que el campo de las tecnologías que se utilizaron es bastante amplio y que se están utilizando en infinidad de aplicaciones. Dejando a la creatividad y el esfuerzo la creación de nuevos y más grandes proyectos utilizando estas y otras tecnologías relacionadas.

APÉNDICE A

[Código C para el controlador USB.]

```
//-----------------------------------------------------------------------------
// File:     mcuf.c
// Contents:  Hooks required to implement USB peripheral function.
// ESCRITURAS Y LECTURAS EN MODO MANUAL Y DECODIFICACION DE COMANDOS
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
#pragma NOIV            // Do not generate interrupt vectors
#include "fx2.h"
#include "fx2regs.h"
#include "fx2sdly.h"
#include "eeprom.h"
/*--- Declarar las bandera de estado de la memoria externa ---*/
#define FIFOLleno   GPIFREADYSTAT & bmBIT1          // Memoria externa llena
#define FIFOVacio   GPIFREADYSTAT & bmBIT0          // Memoria externa vacía
// Identificacion de cada una de la memorias
#define LECTURA        0x00       // Mediante pines del Puerto C (PC[1:0])
#define RTR            0x01       // se selecciona la memoria sobre la cual
#define PARAMETROS  0x02       // se efectuara la operación ya sea de
#define ESCRITURA      0x03       // lectura o escritura

/*--- Habilitación de forma de onda del GPIF--- */
#define GPIFTRIGRD 4
#define GPIF_EP2 0
#define GPIF_EP4 1
#define GPIF_EP6 2
#define GPIF_EP8 3
/*Vendor commands*/
#define     VR_UPLOAD           0xc0
#define VR_DOWNLOAD         0x40
#define VR_ANCHOR_DLD       0xa0       // handled by core
#define VR_EEPROM           0xa2       // loads (uploads) EEPROM
#define     VR_RAM              0xa3       // loads (uploads) external ram
#define VR_SETI2CADDR       0xa4
#define VR_GETI2C_TYPE      0xa5        // 8 or 16 byte address
#define VR_GET_CHIP_REV     0xa6       // Rev A, B = 0, Rev C = 2 // NOTE: New TNG Rev
#define VR_TEST_MEM         0xa7        // runs mem test and returns result
#define VR_RENUM            0xa8       // renum
#define VR_DB_FX            0xa9       // Force use of double byte address EEPROM (for FX)
#define VR_I2C_100          0xaa       // put the i2c bus in 100Khz mode
#define VR_I2C_400          0xab       // put the i2c bus in 400Khz mode
#define VR_NOSDPAUTO        0xac       // test code. does uploads using SUDPTR with manual length override
#define EP0BUFF_SIZE        0x40       //Definicion de tamaño del buffer del EP0
#define GET_CHIP_REV()   REVID
extern BOOL   GotSUD;                  // Received setup data flag
extern BOOL   Sleep;
extern BOOL   Rwuen;
extern BOOL   Selfpwr;
/* ---Seleccionar el modo de operación del dispositivo USB (HS/FS) ---*/
BYTE   Configuration;          // Current configuration
BYTE   AlternateSetting;       // Alternate settings
/*--- Declaración e inicialización de variables del programa ---*/
static WORD enum_pkt_size = 0x0000;
WORD  OutTC;          // Variable utilizada como contador de transacciones
WORD  auxOutTC;     // Var auxiliar en el conteo de transacciones (pendientes)
BOOL datos = FALSE;  // Bandera para indicar que hay datos leer en ML o respuesta al comando.
BOOL atendida = TRUE;          //Bandera que indica si hay peticiones pendientes
int comando = 0;               //Codigo de comando utilizado para asignar la respuesta correspondiente
BYTE        DB_Addr;           // Dual Byte Address stat
BYTE        I2C_Addr;          // I2C address
BYTE        EE_Page_Size;      // EEPROM page size
BYTE xdata stringID[6];        // Variable en XDATA, almacena el ID leido en los DIPSwitch
```

```
//-----------------------------------------------------------------------------
void GpifInit ();          // Inicialización del GPIF
void escribe(void);        // Prototipo de la funcion ESCRIBE
void lee(void);            // Prototipo de la funcion LEE
void lee_id(void);         // Prototipo de la funcion LEE
//-----------------------------------------------RUTINA DE INICIALIZACION ----------------------------------------------
void TD_Init(void)         // Called once at startup
{
 Rwuen = TRUE;             // Enable remote-wakeup
 EE_Page_Size = 0;         // Set the page size to 0, so we know to calculate it on an EEPROM write
                           // Determine I2C boot eeprom device address;addr = 0x0 for 8 bit addr eeproms (24LC00)
 I2C_Addr = SERIAL_ADDR | ((I2CS & 0x10) >> 4);        // addr=0x01 for 16 bit addr eeprom (LC65)
                                                       // Indicate if it is a dual byte address part
 DB_Addr = (BOOL)(I2C_Addr & 0x01);                    // ID1 is 16 bit addr bit - set by rocker sw or jumper
/*--- CONFIGURACION DEL RELOJ INTERNO A 48 MHZ ---*/
  CPUCS = ((CPUCS & ~bmCLKSPD) | bmCLKSPD1);
  SYNCDELAY;
  GpifInit();              // Inicializacion de registros del GPIF
  REVCTL = 0x03;           // El CPU tiene capacidades mejoradas sobre los paquetes:
  SYNCDELAY;               // COMMIT, SKIP & EDIT/SOURCE tanto para paquetes de entrada como de salida
/*--- CONFIGURACION DE ENDPOINTS ---*/
  EP2CFG = 0xA2;           // EP2OUT, BULK, 512, Doble buffer
  SYNCDELAY;
  EP4CFG = 0x00;           // EP4 NO VALIDO!!
  SYNCDELAY;
  EP6CFG = 0xE2;           // EP6IN, BULK, 512, Doble buffer
  SYNCDELAY;
  EP8CFG = 0x00;           // EP8 NO VALIDO!!
  SYNCDELAY;
/* ---RESET DE FIFOS---*/
  FIFORESET = 0x80;        // Set NAKALL bit to NAK all transfers from host
  SYNCDELAY;               // No reconocimiento (NAK) a todas las transacciones pendientes
  FIFORESET = 0x02;        // Reset EP2 FIFO
  SYNCDELAY;
  FIFORESET = 0x06;        // Reset EP6 FIFO
  SYNCDELAY;
  FIFORESET = 0x00;        // Clear NAKALL bit to resume normal operation
  SYNCDELAY;               // Se borra el bit NAKALL, regreso a operacion normal.
/* ---CONFIGURACION DE LOS EP26FIFO--- */
  EP2FIFOCFG = 0x20;               // Modo **MANUAL OUT**, paquetes de tamaño 0 inhabilitados,
  SYNCDELAY;                       // Operaciones con bytes, la bandera de vacío de pone 1 operacion antes.
  EP6FIFOCFG = 0x00;               // Modo **MANUAL IN**, paquetes de tamaño 0 inhabilitados,
  SYNCDELAY;                       // Operaciones con bytes
  EP4FIFOCFG = 0x00;               // Forzar operations con Bytes (bus de tados de 8 bits), si algun EPxFIFOCFG se configura para
  SYNCDELAY;                       // operaciones con palabras (bus de tados de 16 bits) todos se comportaran de esta manera.
  EP8FIFOCFG = 0x00;               // Forzar operaciones con Bytes.
  SYNCDELAY;
/*--- 'ARMADO' DE BUFFERS (2) DEL EP2 ---*/
  OUTPKTEND = 0x82; // Arma el EP2 de salida (doble buffer)
  SYNCDELAY;               // Al 'ARMAR' un buffer este queda disponible al CPU
  OUTPKTEND = 0x82; // para operaciones de lectura y escritura
  SYNCDELAY;
/* ---SELECCION DE LAS BANDERAS DE LOS FIFOS UTILIZADAS POR GPIF ---*/
  SYNCDELAY;
  EP2GPIFFLGSEL = 0x01;            // Para EP2OUT, el GPIF utiliza EF (Empty Flag), bandera de VACÍO.
  SYNCDELAY;
  EP6GPIFFLGSEL = 0x02;            // Para EP6IN, el GPIF utiliza FF (Full Flag), bandera de LLENO
  SYNCDELAY;
/* ---HABILITAR PC3:0 COMO SALIDAS---*/
  PORTCCFG = 0x00;         // El puerto C se configura como I/O de proposito general
  OEC = 0x3F;              // Se habilitan PE[5:0] como salidas
  IOC = 0x10 | RTR;        // RTR es la memoria seleccionada por defecto, USBListo (PC4 ) en '1'

//          PC1                 PC0
//           0                   0          0x00 --> LECTURA     PC3 0x08 --> 0x00 _|_ -> Activa carga
//           0                   1          0x01 --> RTR         PC6 0x40 --> 0x00  _|_ -> RESET, PC5 --> Leyendo ML (USBRecibe)
```

```
//            1                    0          0x02 --> PARAMETROS          **No se utilizan PC[7:6]
//            1                    1          0x03 --> ESCRITURA
 OED = 0x00;            // El Puerto D es de entrada, El DIPSwitch esta conectado a este puerto
}
//-----------------------------------------FIN DE RUTINA DE INICIALIZACION ------------------------------

//-------------------------------------------------RUTINA TD_POLL ------------------------------------------------
void TD_Poll(void)
{
 int  i;       // Variable auxiliar, control de ciclo FOR
 WORD  dir;            // Variable auxiliar, control de ciclo FOR
 DWORD  recibidos;   // Variable Recibidos, es el numero de datos que se leeran despues de enviar RECIBIR.
 DWORD  recibidosH; // Parte alta de la varible Recibidos
 DWORD  recibidosL;  // Parte baja de la varible Recibidos
 WORD  pack512;       // Numero de paquetes de 512 bytes que hay en los datos que van a recibirse.
 BYTE  *apunta = stringID;          // Apuntador a la cadena ASCII almacenada en XDATA (ID leido del DIPSwitch)
 BYTE  byteDIP;                     // Byte leido del DIPSwitch
 BYTE centena,decena,unidad;    // Valores C,D,U del ID leido del DIP (Un ID es MCUF_CDU, ej. MCUF_126)
//------------------------------------------------------ ESCRITURA ------------------------------------------------------
if (!(EP2468STAT & 0x01) || !atendida )      // Si el EP2 no esta vacío(el host ha mandado datos)
            {                                // o hay una peticion pendiente
            if(!atendida)           // si no se atendio la ultima peticion
                        {
                        GPIFTCB3 = 0x00;      // Cargar el contador de transacciones con en el numero de bytes
                        SYNCDELAY;            // de la peticion pendiente (auxOutTC).
                        GPIFTCB2 = 0x00;
                        SYNCDELAY;
                        GPIFTCB1 = MSB(auxOutTC);
                        SYNCDELAY;
                        GPIFTCB0 = LSB(auxOutTC);
                        SYNCDELAY;
                        IOC = ESCRITURA;      // Se selecciona la 'memoria de ESCRITURA
                        escribe();            // Funcion Escribe, Escribe a la memoria externa el numero de bytes
                                              // especificados por GPIFTCB[3:0]
                        IOC = RTR;            // Deselecciona memoria, regresa a la memoria default (RTR)
                        }

                        else if (!(EP2468STAT & 0x01))    // Si el host envio datos y la ultima peticion se atendio
                        {
                                    if(!(EP2FIFOFLGS & 0x01) )        // Si el EP2FIFO no está lleno
                                                {
                                                if((EP2FIFOBUF[0] == 0x00))       //Es el primer byte = 0x00 ? (se trata de algun comando?)
                                                            {
                                                            switch(EP2FIFOBUF[1])//Se checa el segundo byte para identificar el comando
                                                                        {
                                                /*CAMBIA ID*/           case 0x81:              // Segundo byte 0x80
                                                                        byteDIP = IOD;          // Lee los switches
                                                                        byteDIP = ~(byteDIP); // byteDIP = not(byteDIP)
                                                                        centena = byteDIP / 100;          // Se obtienen los valores
                                                                        decena = (byteDIP - 100 * centena)/10;      // CDU del nuevo
                                                                                                                    //ID (MCUF_CDU)
                                                                        unidad = byteDIP - 100 * centena - 10 * decena;
                                                                        stringID[0] = centena + 48;         // Se convierten los valores
                                                                        stringID[1] = 0x00;                  // UDC a caracteres ASCII
                                                                        stringID[2] = decena + 48;          // y se almacenan en stringID
                                                                        stringID[3] = 0x00;                  // (en XDATA)
                                                                        stringID[4] = unidad + 48;
                                                                        stringID[5] = 0x00;
                                                                        OUTPKTEND = 0x82;  // SKIP packet, informacion
                                                                                           // solo util como comando
                                                                        SYNCDELAY;
                                                                        EEPROMWritePage(0x011C, apunta, 0x06); // Se escribe el nuevo
                                                                                                               // ID a la EEPROM

                                                                                                                // al Resetear se cargara
                                                                                                               //este nuevo ID.
```

```
                    break;

/*RESET*/           case 0x04:              // Segundo byte 0x04
                    /* ---RESET de Contador de transacciones y GPIF*---/
                    GPIFABORT = 0xFF;   // Se abortan todas las formas
                                        //de onda de GPIF
                    GPIFTCB3 = 0x00;    // que pudieran estar ejecutandose.
                    SYNCDELAY;          // Se limpia el contador de transacciones de
                    GPIFTCB2 = 0x00;    // GPIF GPIFTCB[3:0].
                    SYNCDELAY;
                    GPIFTCB1 = 0x00;
                    SYNCDELAY;
                    GPIFTCB0 = 0x00;
                    SYNCDELAY;
                    /*---RESET DE FIFOS---*/
                    FIFORESET = 0x80;   // NAK (No Acknowledge)
                                        //a todas la transacciones
                    SYNCDELAY;
                    FIFORESET = 0x02;   // Reset EP2 FIFO
                    SYNCDELAY;
                    FIFORESET = 0x06;   // Reset EP6 FIFO
                    SYNCDELAY;
                    FIFORESET = 0x00;   // NAKALL bit = '0', Operacion normal
                    SYNCDELAY;
                    /*--- 'ARMADO' DE BUFFERS (2) DEL EP2 ---*/
                    OUTPKTEND = 0x82;  // Arma el EP2 de salida (doble buffer)
                    SYNCDELAY;
                    OUTPKTEND = 0x82;
                    SYNCDELAY;
                    /* ---PC2 EN ALTO POR 600ns: Reset FPGA---*/
                    IOC |= 0x04;            // ResetFPGA (PC2) en '1'
                    SYNCDELAY;
                    IOC &= 0XDB;            // ResetFPGA (PC2) y USBRecibe (PC5) en '0',
                                            //el resto no cambia
                    /*---No hay datos que leer---*/
                    recibidos = 0x00000000;         // El numero de datos a leer es cero.
                    recibidosH = 0x0000;
                    recibidosL = 0x0000;
                    atendida = TRUE;
                    auxOutTC = 0x0000;  // 0x0000 Transacciones pendientes
                    datos = FALSE;          //  Bandera datos, no hay datos
                    /*---Respuesta a comando RESET---*/
                    EP6FIFOBUF[0] = 0x01;           //Se edita una respuesta a RESET
                    EP6FIFOBUF[1] = 0x04;           //0x0104, respuesta al RESET
                    SYNCDELAY;
                    EP6BCH = 0x00;          // Source packet
                    SYNCDELAY;
                    EP6BCL = 0x02;          // Tamaño: 2 bytes, COMMIT
                    break;

/*SINCRONIZA*/      case 0x03:              //Segundo byte = 0x03 --> comando 'SINCRONIZA'
                    comando = 1;        // El comando recibido fue SINCRONIZA
                    OutTC = (EP2FIFOBUF[2]<<8) + EP2FIFOBUF[3];
                    GPIFTCB3 = 0x00;    //Se carga el valor del numero de datos en el
                    SYNCDELAY;          //contador de transacciones para posteriormente
                    GPIFTCB2 = 0x00;    //escribirlos al disparar GPIF
                    SYNCDELAY;
                    GPIFTCB1 =MSB(OutTC);
                    SYNCDELAY;
                    GPIFTCB0 =  LSB(OutTC);
                    SYNCDELAY;

                    for(i=0; i<4; i++)      //Se remueven los primeros
                                            //4 bytes correspondientes al
                                            // ID del comando y la longitud
                    {                       //de los datos enviados
```

```
                    EP2FIFOBUF[i] = EP2FIFOBUF[i+4];
                    EP2FIFOBUF[i+4] = 0x00;
                    }
                    EP2BCH = 0x00;          // 'COMMIT' 4 Bytes (RTR) del
                                            // dominio USB (endpoint) al
                    SYNCDELAY;              // dominio del la interface (slave FIFO).
                    EP2BCL = 0x04;
                    SYNCDELAY;
                    IOC = RTR;              // Direcciona la memoria del RTR
                    escribe();              // Funcion Escribe, Escribe a la memoria
                                            //externa el numero
                    datos = TRUE;           // de bytes especificados porGPIFTCB1:0
                    break;

/*TRANSMITE*/       case 0x02:              //Segundo byte = 0x02 --> comando 'TRANSMITE'
                    comando = 2;            // El comando recibido fue TRANSMITE
                    OutTC = (EP2FIFOBUF[2]<<8) + EP2FIFOBUF[3];  // Se guarda el
                                                                 //dato que indica el
                    dir = OutTC + 3;        // numero N de bytes a ser transmitidos.
                                            // 'dir' es la direccion del último
                                            //dato del paquete
                    for(i=0; i <= (dir-2); i++)      // Se remueven los primeros dos bytes
                    {                                // correspondientes al ID de comando
                                                    //'TRANSMITE
                    EP2FIFOBUF[i] = EP2FIFOBUF[i+2]; //[0]<--[2], [1]<--[3], [2]<--[5],..
                    }
                    OutTC +=2;              // A los datos a enviar se le suman dos Bytes
                                            // que contienen el tamaño del paquete
                    EP2BCH = MSB(OutTC);
                    SYNCDELAY;
                    EP2BCL = LSB(OutTC);// 'Commit' N + 2 bytes
                    SYNCDELAY;              // (los bytes a transmitir + NBD)
                    GPIFTCB3 = 0x00;        // Carga el contador de transacciones con
                    SYNCDELAY;              // el numero de bytes a escribir (OutTC)
                    GPIFTCB2 = 0x00;
                    SYNCDELAY;
                    GPIFTCB1 = MSB(OutTC);
                    SYNCDELAY;
                    GPIFTCB0 = LSB(OutTC);
                    SYNCDELAY;
                    IOC = ESCRITURA;        //Direcciona la 'memoria de ESCRITURA
                    escribe();              // Funcion Escribe, Escribe a la memoria externa
                                            //el numero de bytes  especificados por
                                            // GPIFTCB[3:0]
                    IOC = RTR;              // Deselecciona memoria, regresa a la memoria
                                            //default (RTR)
                    datos = TRUE;
                    break;

/*RECIBE*/          case 0x01:              //Segundo byte = 0x01 --> comando 'RECIBE'
                    comando = 3;            // El comano recibido fue RECIBE
                    datos = TRUE;           // Bandera datos, hay datos que leer.
                    OUTPKTEND = 0x82;       // 'SKIP data' No se escribe nada a memoria
                                            //externa
                    SYNCDELAY;
                    break;

/*DUMMY*/           default:    //Segundo Byte diferente a 0x01, 0x02, 0x03, 0x04...
                    OUTPKTEND = 0x82;   // No se trata de ningun comando --> 'SKIP'
                    SYNCDELAY;
                    comando = 0;            // **Agregado para crear un comando
                                            // 'DUMMY'**
                    datos = TRUE;           // datos = TRUE para poder leer los
                                            //PARAMETROS
                    break;
                    }
```

```
                                        }
                                        else
                                        {

                        /*OTROS*/       OUTPKTEND = 0x82;  //No fue un comando 'SINCRONIZA' ni 'TRANSMITE'
                                                           // --> SKIP PACKET
                                        SYNCDELAY;
                                        }
                                }
                        }
}
//-----------------------------------------------FIN ESCRITURA ---------------------------------------------------
//------------------------------------------------ LECTURA ---------------------------------------------------
if(datos)   // datos = TRUE? (Hay datos que leer?)
{
        if(!(EP68FIFOFLGS & 0x01 )) // Si el EP6FIFO no esta lleno
        {
        switch(comando)
                {
                case 0:                                         // Comando 'DUMMY'
                GPIFTCB3 = 0x00;    // Se carga el contador de transacciones con el valor 0x0000000F (15)
                SYNCDELAY;          // Se leen solo 15 bytes de la memoria de parametros
                GPIFTCB2 = 0x00;    // * La respuesta a los comandos SINCRONIZA y TRANSMITE deben ser solo 4 bytes*
                SYNCDELAY;
                GPIFTCB1 = 0x00;
                SYNCDELAY;
                GPIFTCB0 = 0x0F;
                SYNCDELAY;
                IOC = PARAMETROS;           // Direcciona la 'Memoria de PARAMETROS'    __
                IOC = 0x08 | PARAMETROS;    // Señal ActivaCarga                  __| |__
                IOC = PARAMETROS;
                lee();                      // Se lee PARAMETROS
                IOC = RTR;          //Deselecciona memoria, se vuelve a seleccionar la mem default (RTR)
                GPIFTCB3 = 0x00;    // Reset del contador de transacciones de
                SYNCDELAY;           // GPIF GPIFTCB[3:0]
                GPIFTCB2 = 0x00;
                SYNCDELAY;
                GPIFTCB1 = 0x00;
                SYNCDELAY;
                GPIFTCB0 = 0x00;
                SYNCDELAY;
                EP6FIFOBUF[0] = 0x01;       //Se edita el paquete recien llegado colocando
                EP6FIFOBUF[1] = 0x80;       //el ID de la respuesta 'DUMMY' 0x0180
                INPKTEND = 0x06;            //'COMMIT' datos
                SYNCDELAY;
                datos = FALSE;              // Bandera datos, ya no hay mas datos que leer
                break;
                case 1: case 2:             //Comando 'SINCRONIZA' o 'TRANSMITE'
                GPIFTCB3 = 0x00;            // Se carga el contador de transacciones con
                SYNCDELAY;                  // el valor 0x00000004 (4)
                GPIFTCB2 = 0x00;
                SYNCDELAY;
                GPIFTCB1 = 0x00;
                SYNCDELAY;
                GPIFTCB0 = 0x04;
                SYNCDELAY;          // * La respuesta a los comandos SINCRONIZA y TRANSMITE deben ser solo 4 bytes*
                IOC = PARAMETROS;  // Direcciona la 'Memoria de PARAMETROS'    __
                IOC = 0x08 | PARAMETROS;    // Señal activacarga (PC3)          __| |__
                IOC = PARAMETROS;
                lee();              // Se lee PARAMETROS
                IOC = RTR;          //Deselecciona memoria, se vuelve a seleccionar la mem default (RTR)
                GPIFTCB3 = 0x00;    // Reset del contador de transacciones de
                SYNCDELAY;          // GPIF GPIFTCB[3:0]
                GPIFTCB2 = 0x00;
                SYNCDELAY;
                GPIFTCB1 = 0x00;
```

```
SYNCDELAY;
GPIFTCB0 = 0x00;
SYNCDELAY;
if (comando == 1)                       //Si es comando 'SINCRONIZA'...
        {
        EP6FIFOBUF[0] = 0x01;           //Se edita el paquete recien llegado colocando
        EP6FIFOBUF[1] = 0x03;           //el ID de la respuesta 'SINCRONIZADO' 0x0103
        }
else if(comando == 2)                   //Si es comando 'TRANSMITE'...
        {
        EP6FIFOBUF[0] = 0x01;           //Se edita el paquete recien llegado colocando
        EP6FIFOBUF[1] = 0x02;           //el ID de la respuesta 'TRANSMITIDO' 0x0102
        }
INPKTEND = 0x06;    //'COMMIT' datos
SYNCDELAY;
datos = FALSE;      // Bandera datos, ya no hay mas datos que leer
break;

case 3:             //Comando 'Recibe'
GPIFTCB3 = 0x00;
SYNCDELAY;
GPIFTCB2 = 0x00;
SYNCDELAY;
GPIFTCB1 = 0x00;    // Se carga el contador de transacciones con el
SYNCDELAY;          // valor 0x00000010 (16)
GPIFTCB0 = 0x10;    // Las 16 lecturas corresponden a los 16 bytes de la
SYNCDELAY;          // memoria de PARAMETROS
IOC = PARAMETROS;  //Selecciona memoria PARAMETROS     __
IOC = 0x08 | PARAMETROS;    //Señal activa-carga        __| |__
IOC = PARAMETROS;
lee();                      // Se dispara la lectura hacia el EP6FIFO
IOC = RTR;                  //Deselecciona memoria
EP6FIFOBUF[0] = 0x01;       //Se edita el paquete recien llegado colocando
EP6FIFOBUF[1] = 0x01;       //el ID de la respuesta 'RECIBIDO' 0x010
recibidosH = (EP6FIFOBUF[8] << 8) + EP6FIFOBUF[9];          // dato DWORD  0x0000[8][9]
recibidosL = (EP6FIFOBUF[10] << 8) + EP6FIFOBUF[11];        // dato DWORD  0x0000[10][11]
recibidos = (recibidosH << 16) + recibidosL;                // dato DWORD  0x[8][9][10][11]
if(recibidos != 0)
        {
        IOC = LECTURA | 0x20;       // **Pone en alto PC5 para indicar 'USB leyendo'**
        if(recibidos <= 496)        //Si el numero de bytes recibidos es <= 496...
                {
                GPIFTCB3 = EP6FIFOBUF[8];   // Se carga el contador de transacciones
                SYNCDELAY;                  // del GPIF con el numero NBD que se leyo
                GPIFTCB2 = EP6FIFOBUF[9];   // de la memoria de PARAMETROS.
                SYNCDELAY;
                GPIFTCB1 = EP6FIFOBUF[10];
                SYNCDELAY;
                GPIFTCB0 = EP6FIFOBUF[11];
                SYNCDELAY;
                lee();              // Se dispara una sola vez el GPIF para leer un
                                    // solo paquete de 496 bytes o menos
                INPKTEND = 0x06;    // 'COMMIT' datos. Se pasa el paquete al endpoint,
                SYNCDELAY;          // el paquete incluye los 16 bytes de PARAMETROS
                recibidos = 0x00000000;
                datos = FALSE;      // Bandera datos, ya no hay datos que leer.
                IOC = RTR;          // Deselecciona memoria
                }
        Else    // De otra forma, si el numero de bytes recibidos > 496...
                {
                GPIFTCB3 = 0x00;    // Se carga el contador de transacciones
                SYNCDELAY;          // del GPIF con el valor 0x01F0 (496) para
                GPIFTCB2 = 0x00;    // acompletar el primer paquete de 512 bytes
                SYNCDELAY;          // esto es:
                GPIFTCB1 = 0x01;    //16 bytes (de PARAMETROS) + 496 bytes(LECTURA)
                SYNCDELAY;
```

```
                        GPIFTCB0 = 0xF0;
                        SYNCDELAY;
                        lee();                  // Se leen los primeros 496 bytes.
                        INPKTEND = 0x06;        // 'COMMIT' datos. Pasa el primer paquete de 512 bytes
                        SYNCDELAY;              // al endpoint
                        recibidos -= 496;       // Se restan los primeros 496 bytes que pasan al EP6
                                                // junto con los parametros
                        if (recibidos != 0)
                                {
                                datos = TRUE;     //Bandera datos, aun hay datos por leer
                                comando = 4;      // Ya se leyeron los primeros 496   bytes, la proxima vez
                                }                 // que se cheque el comando ya no ejecutara esta primera parte
                        else.
                                {
                                datos = FALSE;    // Bandera datos, ya no hay más datos que leer
                                }
                        }
                }
                else        //no hay datos que leer
                {
                INPKTEND = 0x06;        // 'COMMIT' datos. Se pasa el paquete al endpoint,
                SYNCDELAY;              // el paquete incluye los 16 bytes de PARAMETROS
                datos = FALSE;          //**
                IOC = RTR;
                }
break;

case 4:     /* ACTUALIZAR 'recibidos'*/
recibidosH = MSW(recibidos);    // Palabra mas significativa de 'recibidos'
recibidosL = LSW(recibidos);    // Palabra menos significativa de 'recibidos'
pack512 = recibidos / 512;      // Numero total de transferencias de 512 bytes
if (pack512 == 0)       // Si los bytes no acompletan el siguiente paquete de 512
                {
                GPIFTCB3 = MSB(recibidosH);     // Se carga el contador de transacciones
                SYNCDELAY;
                GPIFTCB2 = LSB(recibidosH);     // para leer los bytes restantes que no acompletan
                SYNCDELAY;                      // un paquete de 512
                GPIFTCB1 = MSB(recibidosL);
                SYNCDELAY;
                GPIFTCB0 = LSB(recibidosL);
                SYNCDELAY;
                lee();                          // Se leen los bytes restantes
                INPKTEND = 0x06;                //'COMMIT' datos
                SYNCDELAY;
                datos = FALSE;                  // Bandera datos, ya no hay datos que leer
                IOC = RTR;
                recibidos = 0x00000000;
                comando = 0;
                }
else                    //De otra forma, si hay al menos un paquete completo de 512
                {
                GPIFTCB3 = 0x00;
                SYNCDELAY;
                GPIFTCB2= 0x00;
                SYNCDELAY;
                GPIFTCB1= 0x02;         // Carga el contador de transacciones con el valor
                SYNCDELAY;              // constante 0x0200 (512)
                GPIFTCB0= 0x00;
                SYNCDELAY;
                lee();                  // Se leen 512 bytes
                INPKTEND = 0x06;        //'COMMIT' datos
                SYNCDELAY;
                recibidos -= 512;       // Se restan los 512 bytes que se acaban de leer
                if (recibidos == 0)                 // Si ya no hay mas datos que leer
                        {
                        datos = FALSE;          // Bandera datos, ya no hay mas datos que leer
```

```
                                        IOC = RTR;              // Seleccion de RTR
                                        }
                                else
                                        {
                                        datos = TRUE;           // Bandera datos, aun hay datos que leer
                                        }
                                }
                        break;

                        default:

                        break;

                        }               //end case
                }
}               //------------------------------------------- FIN LECTURA ----------------------------------------------------
}
//------------------------------------------------ FIN RUTINA TD_POLL ----------------------------------------
//------------------------------------------------DEFINICION FUNCION ESCRIBE ------------------------------
void escribe(void)      // Funcion escribe, llamada cada vez que se recibe un comando valido
{
                if(GPIFTRIG & 0x80)             // Es GPIF disponible?
                        {
                        if (!(FIFOLleno))       // La Mem-ext no esta llena???
                                {
                                GPIFTRIG = GPIF_EP2;            // Dispara GPIF para realizar escrituras
                                SYNCDELAY;
                                while (!(GPIFTRIG & 0x80))
                                        {
                                        ;                       // Espera que el GPIF saque todos los datos
                                        }
                                atendida = TRUE;                // La peticion se atendio
                                datos = TRUE;                   // datos = TRUE, permite leer la respuesta
                                auxOutTC = 0x0000;              // Las transacciones pendientes son 0x0000
                                }
                        else
                                {
                                atendida = FALSE;       // La peticion no se atendio
                                datos = FALSE;          // No se puede leer respuesta
                                auxOutTC = OutTC;       // Hay un numero OutTC de transacciones pendientes
                                }
                        }
}//-------------------------------------------FIN DEFINICION FUNCION ESCRIBE --------------------------
//-------------------------------------------DEFINICION FUNCION LEE ---------------------------------
void lee(void)          // Funcion lee, llamada cada vez que se solicita obtener los datos en alguna de las memorias
{
                if(GPIFTRIG & 0x80)             // Si el GPIF esta en estado ocioso...
                        {
                        if(!(FIFOVacio))        // Si FIFO externo no esta vacío
                                {
                                GPIFTRIG = GPIFTRIGRD | GPIF_EP6;       // Dispara GPIF para realizar lecturas
                                SYNCDELAY;
                                while(!(GPIFTRIG & 0x80))
                                        {;}                             // Espera a que el GPIF lea el # de bytes especificado
                                }
                        }
}
//-------------------------------------------FIN DEFINICION FUNCION LEE---------------------------------
BOOL TD_Suspend(void)       // Called before the device goes into suspend mode
{
  return(FALSE);                //No entrar a modo suspendido
}
BOOL TD_Resume(void)        // Called after the device resumes
{
  return(TRUE);
}
```

APÉNDICE B

[Código VHDL para el controlador de SDRAM.]

```
----------------------------------------------------------------------------------
-- Module Name:      Main Controller
-- Project Name:
-- Target Devices:   Spartan3
-- Description:      Modulo principal del controlador de la SDRAM
-- Tool versions:    ISE 9.2i
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity main is
   Port (   ifclk : in std_logic;      -- Interface clock (USB)
            reset : in  STD_LOGIC;-- Reset General
            resetffwr : in  STD_LOGIC;          -- Reset FIFO de escritura
             resetffrd : in  STD_LOGIC;         -- Reset FIFO de lectura
            llenoffwr : out  STD_LOGIC;         -- Bandera 'LLENO' memoria de Escritura
            vacioffwr : out  STD_LOGIC;         -- Bandera 'VACÍO' memoria de Escritura
            llenoffrd : out  STD_LOGIC;         -- Bandera 'LLENO' memoria de Lectura
            vacioffrd : out  STD_LOGIC;         -- Bandera 'VACÍO' memoria de Lectura
            refresco : in std_logic;            -- Peticion de refresco (5o cliente)
            codRd : in  STD_LOGIC;              -- Senal de habilitacion de lectura (Codificador)
            decWr : in  STD_LOGIC;              -- Senal de habilitacion de escritura (Decodificador)
            USBWr : in  STD_LOGIC;              -- EnWR, señal proveniente de GPIF (transmite)
            USBRd : in  STD_LOGIC;              -- EnRD, señal proveniente de GPIF (recibe)
-------------------------- Puertos para los 4 clientes -------------------------------------
            WriteUSB  : in std_logic_vector(7 downto 0);            --Bus de entrada (USB escribe)
            ReadUSB   : out std_logic_vector(7 downto 0);           --Bus de salida (USB lee)
            Decobus   : in std_logic_vector(7 downto 0);             --Bus de entrada (Decodificador escribe)
            codbus    : out std_logic_vector(7 downto 0);            --Bus de salida (Codificador lee)
----------------------------PINES DE MEMORIA-----------------------------------------
            cmd_bus   : out std_logic_vector(3 downto 0);           -- Lineas de comando de la SDRAM
            CKE       : out  STD_LOGIC;     -- SDRAM Clock enable
            LDM       : out  std_logic;     -- Lower Data Mask
            UDM       : out  std_logic;      -- Upper Data Mask
            BA        : out  STD_LOGIC_vector(1 downto 0);      -- Bank address
            A         : out  std_logic_vector(12 downto 0);     -- Bus de direcciones
            DatoaLeer : in std_logic_vector(7 downto 0);        --Bus de datos, datos provenientes de SDRAM
            DatoaEscribir : out std_logic_vector(7 downto 0);   --Bus de datos, datos hacia SDRAM
            estado    : out std_logic_vector(3 downto 0);       -- Estado actual de la maquina de estados de comandos
            msgval    : in std_logic_vector(1 downto 0);        -- Señal de validacion de mensajes
---------------------------------- DATO VALIDO ------------------------------------------
            datoValidoFE: out std_logic;    -- Bandera de 'Dato válido' de la memoria de escritura
            datoValidoFL : out std_logic;   -- Bandera de 'Dato válido' de la memoria de lectura
            disponible : out std_logic;     -- Bandera de 'Disponible' de la memoria
            tst1 : out std_logic);
end main;

architecture Behavioral of main is
-- Cosntantes utilizadas para las lineas de comandos --
constant cmdDSL : std_logic_vector(3 downto 0):= "1000"; --cmd_bus(3) -> CS
constant cmdNOP : std_logic_vector(3 downto 0):= "0111"; --cmd_bus(2) -> RAS
constant cmdPRE : std_logic_vector(3 downto 0):= "0010"; --cmd_bus(1) -> CAS
constant cmdACT : std_logic_vector(3 downto 0):= "0011"; --cmd_bus(0) -> WE
constant cmdLMR : std_logic_vector(3 downto 0):= "0000";
constant cmdARF : std_logic_vector(3 downto 0):= "0001";
constant cmdWR  : std_logic_vector(3 downto 0):= "0100";
constant cmdRD  : std_logic_vector(3 downto 0):= "0101";
```

```
-- Módulo para generar tiempos de espera entre estados de la maquina de estados de comandos --
COMPONENT tiemposCMD
PORT(
		clk : IN std_logic;
		reset : IN std_logic;
		estado : IN std_logic_vector(3 downto 0);
		contRD : OUT std_logic_vector(2 downto 0);
		contWR : OUT std_logic_vector(2 downto 0);
		contref : OUT std_logic;
		endtRP : OUT std_logic);
END COMPONENT;

-- Módulo de la maquina de estados de comandos --
COMPONENT states
PORT(
		clk : IN std_logic;
		reset : IN std_logic;
		initrdy : IN std_logic;
		ref_req : IN std_logic;
		lee : IN std_logic;
		scribe : IN std_logic;
		contWR : IN std_logic_vector(2 downto 0);
		contRD : IN std_logic_vector(2 downto 0);
		contref : IN std_logic;
		endtRP : IN std_logic;
		cestado : OUT std_logic_vector(3 downto 0));
END COMPONENT;

-- Módulo de inicializacion de la SDRAM, integra una maquina de estados y generacion
-- de tiempos de espera entre estados.
COMPONENT inicializacion
PORT(
		reset : IN std_logic;
		clk : IN std_logic;
		ready : OUT std_logic;
		estado : OUT std_logic_vector(3 downto 0));
END COMPONENT;
-------SIGNALS----------
signal sgready		: std_logic;		-- Señal para indicar si la SDRAM está lista
signal sgiestado	: std_logic_vector(3 downto 0);  -- Señal estado actual de inicializacion
signal sgcestado	: std_logic_vector(3 downto 0);  -- Señal estado actual de comandos
signal sgWrite		: std_logic;		-- Señales de peticion de escritura/lectura, son generadas a partir de
signal sgRead		: std_logic;		-- prioridades y disparan a la maquina de estados de comandos
signal sgrefresh	: std_logic;		-- Señal de peticion de refresco, dispara la maquina de estados de comandos
signal Selector		: std_logic;		-- Selector de memoria (ML/ME)
signal sgWriteUSB,sgWriteUSB1 : std_logic_vector(7 downto 0);  -- Datos escritos por USB y
signal sgdecobus,sgdecobus1	: std_logic_vector(7 downto 0);  -- datos que escribe el decodificador
signal varx,sgvarxi	: std_logic_vector(4 downto 0);		-- Vector que contiene las peticiones actuales
signal sgAddrCol,sgAddrFila	: std_logic_vector(12 downto 0);-- Direcciones de columna y fila
signal peticiones,sgpet		: std_logic_vector(4 downto 0);  -- Vector que contiene la peticiones hechas en un instante
signal apet		: std_logic;					--Bandera, indica si ya se ha hecho la primer captura de peticiones
signal resets		: std_logic_vector(2 downto 0);		-- Vector que contiene a todos los resets
signal sgcontWR,sgcontRD	: std_logic_vector(2 downto 0);  -- Banderas, indican fin de periodo de espera
signal sgcontref	: std_logic;		-- Bandera, indica fin de periodo de espera
signal sgendtRP		: std_logic;		-- Bandera, indica fin de periodo de espera

begin

tst1 <= sgRead;
estado <= sgcestado; -- El estado actual de comando siempre es visible en el puerto 'estado'
LDM <= '0';		-- No se enmascara ningun dato --
UDM <= '0';
resets <= reset & resetffwr & resetffrd;	-- La señal 'resets' incluye al reser general y al de cada memoria
peticiones <= codRd & decWr & USBRd & USBWr & refresco;	-- 'peticiones' incluye a todas la peticiones posibles
```

```
--                                          PETICION                        PRIORIDAD

                                        -- Lectura por codificador (codRd)          5
                                        -- Escritura por decodificador (decWr)     4
                                        -- Lectura por USB (USBRd)                  3
                                        -- Escritura por USB (USBWr)                2
                                        -- Peticion de refresco                     1

captura: process(reset,ifclk)is     -- Proceso para obtener las peticiones actuales
variable varpet,vrpet1,vrpet2 : std_logic_vector(4 downto 0);
        begin
        if reset = '1' then                 -- Al resetear:
                disponible <= '1';      -- La memoria vuelve a estar disponible
                apet <= '0';            --No se ha hecho una primera captura
                sgpet <= "00000";       --No hay peticiones que atender
        elsif ifclk'event and ifclk = '1' then          -- En cada tansicion positiva de ifclk
                if sgcestado = x"0" and apet = '0' and varx = "00000" then          -- Si el estado es IDLE y no hay peticiones que
                                                                                    --atender y no se ah hecho una primera captura
                        if peticiones /= "00000" and peticiones /= "00001" then-- Si hubo peticiones y  no es solo refresco
                                disponible <= '0';      -- La memoria ya no esta disponible!
                                apet <= '1';                                        -- Ya se hizo la primer captura!!
                                vrpet1 := peticiones(4 downto 1) & '0';     -- Se elimina la peticion de refresco
                                if peticiones(4 downto 1) = x"2" or peticiones(4 downto 1) = x"8" -- Si la peticion es de
                                or peticiones(4 downto 1) = x"A"  then      --lectura, ya sea por--codificador, por USB
                                                                            -- o ambas
                                        sgWriteUSB1 <= x"00";           -- No hay datos que escribir por USB
                                        sgdecobus1 <= x"00";            -- ni por el decodificador
                                        elsif peticiones(4 downto 1) = x"1" or peticiones(4 downto 1) = x"3" or-- Pero si
                                        peticiones(4 downto 1) = x"9" or peticiones(4 downto 1) = x"B" then –las
                                                                             --peticiones son de lectura (cualquiera)
                                                                            -- y tambien de escritura por USB
                                                sgWriteUSB1 <= WriteUSB; --Se guardan los datos que USB escribe
                                                                                --para su posterior escritura
                                                sgdecobus1 <= x"00";--No hay datos que escribir por el decodificador
                                elsif peticiones(4 downto 1) = x"4" or peticiones(4 downto 1) = x"6" or
                                peticiones(4 downto 1) = x"C" or peticiones(4 downto 1) = x"E" then
                                                        -- Pero si las peticiones son de lectura (cualquiera)
                                                        --y tambien de escritura por decodificador
                                        sgWriteUSB1 <= x"00";-- No hay datos que escribir por USB
                                        sgdecobus1 <= decobus; -- Se guardan los datos que el
                                                        -- decodificador escribe para su posterior escritura
                                elsif peticiones(4 downto 1) = x"5" or peticiones(4 downto 1) = x"7" or
                                peticiones(4 downto 1) = x"D" or peticiones(4 downto 1) = x"F" then
                                                -- Pero si las peticiones son de escritura/lectura (cualquiera)
                                        sgWriteUSB1 <= WriteUSB;        -- Se guardan los datos que USB escribe para su
                                                                        -- posterior escritura
                                        sgdecobus1 <= decobus;          -- Se guardan los datos que el decodificador
                                                                        -- escribe para su posterior escritura
                                end if;
                        elsif peticiones = "00001" then   -- Pero si 'peticiones' es solo refresco
                                vrpet1 := peticiones;  -- Variable auxiliar vrpet1 = peticiones
                                disponible <= '0';      -- Sa no esta dsiponible!
                                apet <= '1';            -- Se ha hecho la primer captura!
                        else        -- De otra forma
                                vrpet1 := peticiones;  -- Variable auxiliar vrpet1 = peticiones.
                                apet <= '0';            -- no hubo primer captura.
                                if sgpet /= "00000" then            -- Si no hay pqeticiones que atender
                                        disponible <= '0';      -- la memoria esta disponible
                                else                                -- de lo contrario
                                        disponible <= '1';      -- no esta disponible
                                end if;
                        end if;
                elsif sgcestado = x"0" and apet = '1' then     -- Si el estado es IDLE y ya se hizo la primer captura
                        apet <= '0';            -- Se borra la bandera de 'primer captura'
                        vrpet2 := peticiones(4 downto 1) & '0';     -- Variable auxiliar vrpet2 = peticiones, se borra la
```

```
                                                        --peticion de refresco
                        if peticiones(4 downto 1) = x"0" or peticiones(4 downto 1) = x"2" or    -- Esto es una segunda captura
                        peticiones(4 downto 1) = x"8" or peticiones(4 downto 1) = x"A"  then -- de la speticiones actuales
                                sgWriteUSB <= sgWriteUSB1;                                      -- por lo que se realizo lo mismo
                                sgdecobus <= sgdecobus1;                                        -- que en la primer captura
                        elsif peticiones(4 downto 1) = x"1" or peticiones(4 downto 1) = x"3" or
                        peticiones(4 downto 1) = x"9" or peticiones(4 downto 1) = x"B" then
                                sgWriteUSB <= WriteUSB;
                                sgdecobus <= sgdecobus1;
                                elsif peticiones(4 downto 1) = x"4" or peticiones(4 downto 1) = x"6" or
                                peticiones(4 downto 1) = x"C" or peticiones(4 downto 1) = x"E" then
                                        sgWriteUSB <= sgWriteUSB1;
                                        sgdecobus <= decobus;
                                        elsif peticiones(4 downto 1) = x"5" or peticiones(4 downto 1) = x"7" or
                                        peticiones(4 downto 1) = x"D" or peticiones(4 downto 1) = x"F" then
                                                sgWriteUSB <= WriteUSB;
                                                sgdecobus <= decobus;
                                        end if;
                        varpet := vrpet1 or vrpet2;     -- varpet son las peticiones de la primera y segunda captura
                        if varpet = "00001" then        -- Si es solo refreso
                                sgpet <= varpet;        -- la señal sgpet toma el valor de varpet
                        else                            -- pero si no es solo refresco
                                sgpet <= varpet(4 downto 1) & '0';          -- se borra la peticion de refresco.
                        end if;                         --*sgpet* contiene las peticiones a atender
                end if;                 -- despues de las dos capturas
                if sgcestado /= x"0" then       -- Si el estado es diferente de IDLE
                        sgpet <= "00000";       -- se borra la señal sgpet, se deja lista para las proximas capturas
                end if;
        end if;
end process captura;

prioridad: process(reset,ifclk,sgcestado) is -- Proceso para atender las peticiones en base a su prioridad
variable mask,varxi: std_logic_vector(4 downto 0);
variable vrWriteUSB,vrdecobus : std_logic_vector(7 downto 0);

begin
if reset = '1' then     -- Al resetear:
        sgWrite <= '0';         -- No hay peticion de escritura que dispare la maquina de estados
        sgRead <= '0';          --No hay peticion de lectura que dispare la maquina de estados
        sgrefresh <= '0';       -- No hay peticion de refresco que dispare la maquina de estados
        Selector <= '1';        -- Se selecciona la memoria de escritura
        varx <= "00000";        -- Variable auxiliar, peticiones pendientes
        varxi := "00000";       -- No hay peticiones actuales que atender
        mask := "00000";        -- no hay peticiones pendientes que atender
        DatoaEscribir <= x"00";
        sgvarxi <= "00000";
elsif ifclk'event and ifclk = '1' then          -- En cada transicion positiva de ifclk
        if sgcestado = x"0" then        -- Si el estado es IDLE
                if mask = "00000" then          -- Si no hay peticiones pendientes
                        varxi := sgpet;         -- Las periciones actuales son las peticiones de las capturas
                        sgvarxi <= sgpet;       -- sgvarxi se utiliza en el process 'Lecturas'
                else                            -- si hay peticiones pendientes
                        varxi := mask;          -- estas seran las proximas en atender
                        sgvarxi <= mask;        -- sgvarxi se utiliza en el process 'Lecturas'
                end if;
                -- *Se atiende la peticion de mayor prioridad y se borra de 'varxi' una vez atendida* --
                if varxi(4) = '1' then  -- Si la proxima peticion a atender es Lectura a ME
                        sgRead <= '1';          -- Se hace al peticion de lectura que dispara al maquina de estados (CodLee)
                        sgWrite <= '0';         -- No hay peticion de escritura
                        sgrefresh <= '0';       -- No hay peticion de refresco
                        Selector <= '1';        -- Seleccion de Memoria de escritura
                        BA <= "11";
                        mask := '0' & varxi(3 downto 0);  -- Se borra la peticion que se acaba de atender
                        DatoaEscribir <= x"00";         -- No hay datos que escribir
                elsif varxi(4)= '0' and varxi(3)= '1' then      -- Si la proxima peticion a atender es ecritura a ML
                        sgRead <= '0';                  -- No hay peticion de lectura
```

```
                        sgWrite <= '1';         -- Se hace al peticion de lectura que dispara al maquina de estados (DecoEscribe)
                        sgrefresh <= '0';       -- No hay peticion de refresco
                        Selector <= '0';        -- Seleccion Memoria de lectura
                        BA <= "00";
                        mask := "00" & varxi(2 downto 0);           -- Se borra la peticion que se acaba de atender
                        DatoaEscribir <= sgdecobus;                 -- El dato a escribir proviene de decobus (Decodificador)
                elsif varxi(4 downto 3)= "00" and varxi(2)= '1' then    -- Si la peticion a atender es Lectura a ML
                        sgRead <= '1';                              -- Se hace la peticion de lectura que dispara la maquina
                                                                    -- de estados (USBLee)
                        sgWrite <= '0';                 -- No hay peticion de escritura
                        sgrefresh <= '0';               -- no hay peticion de refresco
                        Selector <= '0';                -- Seleccion de Memoria de lectura
                        BA <= "00";
                        mask := "000" & varxi(1 downto 0);          -- Se borra la peticion que se acaba de atender
                        DatoaEscribir <= x"00";                     -- No hay datos que escribir
                elsif varxi(4 downto 2)= "000" and varxi(1)= '1' then   -- Si la peticion a atender es Escritura a ME
                        sgRead <= '0';          -- No hay peticion de lectura
                        sgWrite <= '1';         -- Se hace la peticion de lectura que dispara la maquina de estados (USBEsc)
                        sgrefresh <= '0';       -- No hay peticion de refresco
                        Selector <= '1';        -- Seleccion de Memoria de escritura
                        BA <= "11";
                        mask := "0000" & varxi(0);          -- Se borra la peticion que se acaba de atender
                        DatoaEscribir <= sgWriteUSB;    -- El dato a escribir proviene de WriteUSB (USB)
                elsif varxi(4 downto 1) = "0000" and varxi(0) = '1' then -- Si la peticion a atender es Refresco
                        sgRead <= '0';          -- No hay peticoin de lectura
                        sgWrite <= '0';         -- No hay peticion de escritura
                        sgrefresh <= '1';       -- Se genera la peticion de Refresco que dispara la maquina de estados
                        mask := "00000";        -- Se borra la peticion que se acaab de atender
                        DatoaEscribir <= x"00";             -- No hay datos que escribir
                else
                        sgRead <= '0';          --** NO PETICION **--
                        sgWRite <= '0';
                        sgrefresh <= '0';
                        mask := "00000";
                        DatoaEscribir <= x"00";
                end if;
        else
                sgRead <= '0';
                sgWrite <= '0';
                sgrefresh <= '0';
        end if;
end if;
varx <= mask;           -- La variable auxiliar varx contiene las peticiones pendientes
end process prioridad;


lecturas: process(ifclk,reset) is    -- * Proceso de atencion de lecturas * --
variable contador : std_logic_vector(3 downto 0);       -- Contador para controlar la señal 'dato valido'
variable habcuenta : std_logic;                         -- Habilitador de 'contador'

begin
if reset = '1' then       -- Si se resetea
        codbus <= x"00";        -- No hay datos de salida en el puerto 'codbus'
        ReadUSB <= x"00";       -- No hay datos de salida en el puerto 'ReadUSB'
        datoValidoFE <= '0';    -- No hay 'dato valido' de la memoria de escritura
        datoValidoFL <= '0';    -- No hay 'dato valido' de la memoria de lectura
        contador := "0000";     -- Reset del contador
elsif  ifclk'event and ifclk = '1' then         -- En cada transicion positiva de ifclk...
        if sgcestado = x"A" then        -- Si el estado actual es rDATA
                habcuenta := '1';                       -- Se habilita el contador (inicia cuenta al entrar al estado rDATA)
                if sgvarxi(4 downto 1) = x"2" or sgvarxi(4 downto 1) = x"3" then    -- Si la peticion es lectura por USB
                        ReadUSB <= DatoaLeer;                   -- El puerto de salida ReadUSB muesta el DatoaLeer
                        codbus <= x"00";                        -- No hay dato a ser leidos por el codificador
                        if contador = "0010" then               -- Poner la señal de 'dato valido' de memoria de lectura
                                datoValidoFE <= '0';    -- cuando el contador llegue al valor 2 durante el estado rDATA
                                datoValidoFL <= '1';
```

```
                        end if;
                elsif sgvarxi(4 downto 1) = x"8" or sgvarxi(4 downto 1) = x"9" or -- Si la peticion es lectura por Codificador
                        sgvarxi(4 downto 1) = x"A" or sgvarxi(4 downto 1) = x"B" or
                        sgvarxi(4 downto 1) = x"C" or sgvarxi(4 downto 1) = x"D" or
                        sgvarxi(4 downto 1) = x"E" or sgvarxi(4 downto 1) = x"F" then
                        codbus <= DatoaLeer;            -- El puerto de salida codbus muestra el DatoaLeer
                        ReadUSB <= x"00";               -- No hay datos a ser leidos por USB
                        if contador = "0010" then       -- Poner la señal de 'dato valido' de memoria de escritura
                                datoValidoFE <= '1';    -- cuando el contador llegue al valor 2 durante el estado rDATA
                                datoValidoFL <= '0';
                        end if;
                else                                    -- En caso de que no se trate de una peticion de lectura
                        codbus <= x"00";        -- No hay datos a ser leidos por el codificador
                        ReadUSB <= x"00";       -- No hay datos a ser leidos por el codificador
                        datoValidoFE <= '0';    -- No hay 'dato valido' de ninguna de las dos memorias
                        datoValidoFL <= '0';
                end if;
        end if;
        if habcuenta = '1' then                         -- Si el habilitador de 'contador' esta habilitado
                if contador > "0010" then       -- Si cuenta > 2
                        contador := "0000";             -- Resetear 'contador'
                        habcuenta := '0';               -- Quitar habilitacion de contador
                        datoValidoFE <= '0';            -- Quitar señal de 'dato valido' de las dos memorias
                        datoValidoFL <= '0';            -- ('dato valido' solo se pone durante un ciclo)
                else                                    -- En caso de que cuenta aun no sea mayor que 2
                        contador := contador + 1;       -- incrementar el contador
                end if;
        end if;
end if;
end process lecturas;

--** Proceso para controlar las direcciones de escritura/lectura **--
direcciones: process(ifclk,sgRead,sgWrite,sgvarxi,resets)is
variable avcuentawr_ML : std_logic_vector(19 downto 0);         -- Cuenta (AVANZADA) escrituras a ML
variable recuentawr_ML : std_logic_vector(19 downto 0);         -- Cuenta (REAL)escrituras a ML
variable cuentawr_ME : std_logic_vector(19 downto 0);           -- Cuenta escrituras a ME
variable cuentard_ML : std_logic_vector(19 downto 0);           -- Cuenta lecturas a ML
variable cuentard_ME : std_logic_vector(19 downto 0);           -- Cuenta lecturas a ME
variable vrAddrCol : std_logic_vector(12 downto 0);             -- Direccion de la columna
variable vrAddrFila : std_logic_vector(12 downto 0);            -- Direccion de la fila

begin
if resets /= "000" then                         -- Si se da alguno de los resets
        case resets is                          -- Caso del reset de ME
        when "010" =>   cuentawr_ME := "00000000000000000000"; -- Resetea cuentas de lectura/escritura
                        cuentard_ME := "00000000000000000000";          -- de la Memoria de Escritura
                        -- Caso del reset de ML
        when "001" =>   avcuentawr_ML := "00000000000000000000";        -- Resetea cuentas de lectura/escritura
                        recuentawr_ML := "00000000000000000000";        -- tanto REAL como AVANZADA de la Memoria
                        cuentard_ML := "00000000000000000000";          -- de Lectura
                        -- Caso del reset general
        when others =>  avcuentawr_ML := "00000000000000000000";        -- reset de TODOS los contadores de
                        recuentawr_ML := "00000000000000000000";        -- lectura/escritura de las dos MEMORIAS
                        cuentawr_ME := "00000000000000000000";
                        cuentard_ML := "00000000000000000000";
                        cuentard_ME := "00000000000000000000";
        end case;
elsif ifclk'event and ifclk = '0' then          -- En cada transicion negativa de ifclk checar peticiones
        if sgWrite = '1' then                   -- Si hay peticion de escritura
                if sgvarxi(4) = '0' and sgvarxi(3) = '1' then --Si es escritura del DECO a ML
                        vrAddrCol := "0000" &  avcuentawr_ML(8 downto 0);--La direccion de columna y fila esta dada
                        vrAddrFila :="00" &  avcuentawr_ML(19 downto 9);--por el contador (AVANZADA) de escrituras a ML
                        avcuentawr_ML := avcuentawr_ML + 1;  -- Incremeto al contador (AVANZADA) de escrituras a ML
                        elsif sgvarxi(4 downto 2) = "000" and sgvarxi(1) = '1' then --Si es escritura USB a ME
                                vrAddrCol := "0000" &  cuentawr_ME(8 downto 0);--La direccion de columna y fila esta dada
                                vrAddrFila :="00" &  cuentawr_ME(19 downto 9); -- por el contador de escrituras a ME
```

```
                                        cuentawr_ME := cuentawr_ME + 1;          -- Incremento al contador de escrituras a ME
                                end if;
        elsif sgRead = '1' then             -- Si hay peticion de lectura
                if sgvarxi(4) = '1' then -- Si es lectura de lectura del Codificador a ME
                        vrAddrCol := "0000" &  cuentard_ME(8 downto 0);     -- La direccion de columna y fila esta dada
                        vrAddrFila :="00" &  cuentard_ME(19 downto 9);      -- por el contador de lecturas a ME
                        cuentard_ME := cuentard_ME + 1;-- Incremento al contador de lecturas a ME
                        elsif sgvarxi(4 downto 3) = "00" and sgvarxi(2) = '1' then          -- Si es Lectura USB a ML
                                vrAddrCol := "0000" &  cuentard_ML(8 downto 0);-- La direccion de columna y fila esta dada
                                 vrAddrFila:="00" &  cuentard_ML(19 downto 9);      --por el contador de lecturas a ML
                                cuentard_ML := cuentard_ML + 1;             -- Incremento al contador de lecturas a ML
                        end if;
                end if;
                --* Validacion del mensaje escrito en ML *--
        if msgval = "01" then                               -- Si hay error en el mensaje
                avcuentawr_ML := recuentawr_ML;          -- El contador de AVANZADA regresa al valor REAL
                elsif msgval = "10" then                                -- Pero si el mensaje es bueno
                        recuentawr_ML := avcuentawr_ML;          -- El contador REAL se incrementa al valor de AVANZADA
                end if;
        end if;
sgAddrCol <= vrAddrCol;          -- Asignacion de los valores de direccion de columna y fila
sgAddrFila <= vrAddrFila;        -- a las señales que van hacia el puerto de direcciones.
end process direcciones;

--*Proceso para dotar a la memoria de comportamiento FIFO*--
fifo : process(sgWrite,sgRead,selector,ifclk,resets) is
variable disponibles_ME : std_logic_vector(19 downto 0):= x"FFFFF";--Bytes disponibles (1048575)--CAMBIO DE TAMANO
variable disponibles_ML : std_logic_vector(19 downto 0):= x"FFFFF";--Bytes disponobles (1048575)--CAMBIO DE TAMANO
variable cntdatos : std_logic_vector(15 downto 0);      -- Contador de datos que se han escrito a ML
begin
if resets /= "000" then                     -- Si se da alguno de los resets
        case resets is                      -- Caso del reset de ME
        when "010" =>       disponibles_ME := x"FFFFF";     -- La disponibilidad de Memoria de Escritura es maxima
                            vacioffwr <= '1';               -- La Memoria de Escritura esta vacía
                            llenoffwr <= '0';               -- La memoria no esta llena
                            -- Caso del reset de ME
        when "001" =>       disponibles_ML := x"FFFFF";     -- La disponibilidad de Memoria de Lectura es maxima
                            vacioffrd <= '1';               -- La Memoria de Lectura esta vacía
                            llenoffrd <= '0';               -- La memoria no esta llena
                            cntdatos := x"0000";            -- No se ha escrito ningun dato a ML
                            -- Caso del reset general
        when others =>      disponibles_ME := x"FFFFF";     -- La disponibilidad de las dos memorias
                            disponibles_ML := x"FFFFF";     -- es maxima.
                            vacioffwr <= '1';               -- Ambas memorias estan vacías y no llenas
                            llenoffwr <= '0';
                            vacioffrd <= '1';
                            llenoffrd <= '0';
                            cntdatos := x"0000";            -- No se ha escrito ningun dato a ML
        end case;
elsif ifclk'event and ifclk = '0' then          -- EN cada transicion negativa de ifclk
        if selector = '1' then              --Si se ha seleccionado **Memoria de Escritura**--
                if sgWrite = '1'  then                          -- si hay peticion de escritura
                        disponibles_ME := disponibles_ME - 1;   -- Hay 1 byte menos disponible en ME
                        elsif sgRead = '1'  then                        -- Si hay peticion de lectura
                                disponibles_ME := disponibles_ME + 1;   -- hay 1 byte mas disponible en ME
                        end if;
        else                                -- Si se ha seleccionado **Memoria de Lectura**--
                if sgWrite = '1' then                       -- Si hay peticion de escritura
                        disponibles_ML := disponibles_ML - 1;   -- Hay 1 byte menos disponible en ML
                        cntdatos := cntdatos + 1;               -- Se ha escrito 1 byte mas en ML
                elsif sgRead = '1' then                         -- Si hay peticion de lectura
                        disponibles_ML := disponibles_ML + 1;   -- Hay 1 byte mas disponible en ML
                -- *Validacion de mensaje* --
                elsif msgval = "01" then                        -- Si el mensaje es erroneo
                        disponibles_ML := disponibles_ML + cntdatos; -- Los bytes disponibles en ML se incrementa, los datos
                                                                     -- escritos no fueron validos
```

```
                        cntdatos := x"0000";            -- Se resetea el contador de datos escritos a ML
                elsif msgval = "10" then                -- Si el mensaje es bueno
                        cntdatos := x"0000";            -- Reset de contador de datos escritos a ML
                end if;                                 -- El numero de bytes disponibles en ML no cambia.
        end if;

                                        --**Disponibilidad ML**--
        if disponibles_ML = x"FFFFF" then       -- Si la disponibilidad de ML es maxima
                vacioffrd <= '1';               -- La memoria esta vacía
                llenoffrd <= '0';               -- la memoria no esta llena
        elsif disponibles_ML <= x"00084" then -- Pero si disponibles < 132 bytes
                vacioffrd <= '0';               -- La memoria no esta vacía
                llenoffrd <= '1';               -- La memoria esta llena
        else                                    -- Cualquier otro valor de disponibles:
                vacioffrd <= '0';               -- La memoria no esta llena
                llenoffrd <= '0';               -- ni vacía
        end if;
                                        --**Disponibilidad ME**--
        if disponibles_ME = x"FFFFF" then       -- Si la disponibilidad de ME es maxima
                vacioffwr <= '1';               -- La memoria esta vacía
                llenoffwr <= '0';               -- La memoria no esta llena
        elsif disponibles_ME <= x"00084" then -- Pero si disponibles < 132 bytes
                vacioffwr <= '0';               -- La memoria no esta vacía
                llenoffwr <= '1';               -- La memoria esta llena
        else                                    -- Cualquier otro valor de disponibles:
                vacioffwr <= '0';               -- La memoria no esta llena
                llenoffwr <= '0';               -- ni vacía
        end if;
        end if;
end process fifo;


--* Proceso para generar las señales de salida, principalmente comandos, hacia los pines de la memoria*--
salidas: process(reset,sgiestado,sgcestado,sgAddrFila,sgAddrCol)
begin
        if reset = '1' then             -- Al resetear
                cmd_bus <= cmdDSL; -- Aplicar comando DESELECT
        else
                case sgiestado is               -- Estado de las salidas dependiendo del estado actual de la inicializacion
--*** Secuencia de inicializacion ***
                when x"0" => cmd_bus <= cmdNOP;         -- Estado IDLE -> Comando NOP

                when x"A" => cmd_bus <= cmdNOP;  -- Estado NOPS -> Comando NOP

                when x"1" => cmd_bus <= cmdPRE;  -- Estado PRE       -> Comando PRECHARGE
                          A(10) <= '1';         -- A10 = '1' Para precargar todos los bancos

                when x"2" => cmd_bus <= cmdNOP;         --Estado TRP  -> Comando NOP
                                                        -- Tiempo de espera tRP

                when x"3" => cmd_bus <= cmdARF;-- Estado AR1  -> Comando AUTOREFRESH


                when x"4" => cmd_bus <= cmdNOP;         --Estado TRF1 -> Comando NOP
                                                        -- Tiempo de espera tRFC
                when x"5" => cmd_bus <= cmdARF;--EstadoAR2 -> Comando AUTOREFRESH

                when x"6" => cmd_bus <= cmdNOP;         --Estado TRF2 -> Comando NOP
                                                        -- Tiempo de espera tRFC
                when x"7" => cmd_bus <= cmdLMR;         -- Estado LMR  -> Comando LMR  (Carga registro de modo)
                           A(2 downto 0) <= "000";      -- Burst = 1
                           A(3) <= '0';                  -- Burst Secuencial
                          A(6 downto 4) <= "010";       -- CL = 2
                          A(8 downto 7) <= "00";         -- Modo de operacion standard
                          A(9) <= '1';                  -- '0' Programmed Burst Lenght '1' single access
                          A(12 downto 10) <= "000";     -- 'Reservado'
```

```
                    when x"8" => cmd_bus <= cmdNOP;          -- Estado TMRD        -> Comando NOP
                                                             -- Tiempo de espera tMRD

                    when x"9" => NULL;   -- Estado RDY       -- * Memoria lista para cualquier operacion valida *

                    when others => NULL;

                    end case;
            end if;

--*** Secuencia para Comandos ***
            if  sgiestado = x"9" then           -- Si la memoria ya esta lista (estado RDY de inicializacion)
                    case sgcestado is      -- Generar salidas dependiendo del estado actual de la maquina  de estados de comandos.

                    when x"0" => cmd_bus <= cmdNOP;  -- Estado IDLE     -> Comando NOP
----*** Ciclo de refresco            ***

                    when x"1" => cmd_bus <= cmdARF;          -- Estado ARF->Comando AUTOREFRESH

                    when x"2" => cmd_bus <= cmdNOP;          -- Estado tRFC -> Coamndo NOP
                                                             -- Tiempo de respera tRFC
--          ***     Ciclo de Escritura      ***
                    when x"B" => cmd_bus <= cmdNOP;          -- Estado DirWR -> Comando NOP
                                                                                    -- Tiempo de espera
                    when x"3" => cmd_bus <= cmdACT;          --Estado ACTIVEwr -> Comando ACTIVE
                                 A <= sgAddrFila;            -- Poner direccion de fila

                    when x"4" => cmd_bus <= cmdPRE;          --Estado wrtRCD       -> Comando NOP
                                                             -- Tiempo de espera tRCD
                                 A(10) <= '1';

                    when x"5" => cmd_bus <= cmdWR;           -- Estado WR->        Comando WRITE
                                 A <= sgAddrCol;             -- Poner direccion de columna

                    when x"6" => cmd_bus <= cmdNOP;          --Estado tWR->Comando NOP
                                                             -- Tiempo de espera tWR

--          ***     Ciclo de Lectura        ***
                    when x"C" => cmd_bus <= cmdNOP;          -- Estado dirRD       -> Comando NOP
                                                                                    -- Tiempo de espera
                    when x"7" => cmd_bus <= cmdACT;          -- Estado ACTIVErd -> Comando ACTIVE
                                 A <= sgAddrFila;            -- Poner direccion de fila

                    when x"8" => cmd_bus <= cmdNOP;          -- Estado rdtRCD      -> Comando NOP
                                                             -- Tiempo de espera
                                 A(10) <= '0';
                    when x"9" => cmd_bus <= cmdRD;           -- Estado RD          -> Comando READ
                                 A <= sgAddrCol;             -- Poner direccion de columna

                    when x"A" => cmd_bus <= cmdNOP;          -- Estado rDATA       ->          Comando NOP
                                                             -- Tiempo de espera, incluye CL

--          ***     Precharge & precharge command period    ***
                    when x"D" => cmd_bus <= cmdPRE;          -- Estado PRECH       -> Comando PRECHARGE
                                 A(10) <= '1';               -- A(10) = '1' para precargar todos los bancos

                    when x"E" => cmd_bus <= cmdNOP;          -- Estado tRP         -> Comando NOP
                                A(10) <= '0';                -- Tiempo de espera tRP

                    when others => NULL;

                    end case;
            end if;

end process salidas;
```

```
--*Proceso para habilitar la senal de reloj  medirnte el pin CKE del la SDRAM*--
habCKE: process (reset,sgiestado) is
        begin
        if reset = '1' then       -- Al resetear
                CKE <= '0';                     -- Clock Enable (CKE) de la SDRAM esta en '0'
        else
                if sgiestado /= x"0" then       -- Si el estado de inicializacion es distinto de IDLE
                        CKE <= '1';             -- Pin CKE = '1'
                else
                        CKE <= '0';             -- Pin CKE = '0' solo en el esatdo IDLE
                end if;
        end if;
        end process habCKE;

--**Generacion de periodos de espera para cmdFSM**--
Inst_tiemposCMD: tiemposCMD PORT MAP(
        clk => ifclk,
        reset => reset,
        estado => sgcestado,
        contRD => sgcontRD,
        contWR => sgcontWR,
        contref => sgcontref,
        endtRP => sgendtRP);

--** Maquina de estados CMD **--
Inst_states: states PORT MAP(
                clk => ifclk,
                reset => reset,
                initrdy => sgready,
                ref_req => sgrefresh,
                lee => sgRead,
                scribe => sgWrite,
                contWR => sgcontWR,
                contRD => sgcontRD,
                contref => sgcontref,
                endtRP => sgendtRP,
                cestado => sgcestado);

--**Inicializacion de la SDRAM**--
Inst_inicializacion: inicializacion PORT MAP(
        reset => reset,
        clk => ifclk,
        ready => sgready,
        estado => sgiestado);

end Behavioral;
```

APÉNDICE C

[Listado .a51 de los descriptores del dispositivo]

```
;;-----------------------------------------------------------------------------
;;   File:     dscr.a51
;;   Contents:  This file contains descriptor data tables.
;; $Archive: /USB/Examples/Fx2lp/bulkloop/dscr.a51 $
;; $Date: 9/01/03 8:51p $
;; $Revision: 3 $
;;-----------------------------------------------------------------------------
DSCR_DEVICE   equ   1   ;; Descriptor type: Device
DSCR_CONFIG   equ   2   ;; Descriptor type: Configuration
DSCR_STRING   equ   3   ;; Descriptor type: String
DSCR_INTRFC   equ   4   ;; Descriptor type: Interface
DSCR_ENDPNT   equ   5   ;; Descriptor type: Endpoint
DSCR_DEVQUAL  equ   6   ;; Descriptor type: Device Qualifier

DSCR_DEVICE_LEN   equ   18
DSCR_CONFIG_LEN   equ   9
DSCR_INTRFC_LEN   equ   9
DSCR_ENDPNT_LEN   equ   7
DSCR_DEVQUAL_LEN  equ   10

ET_CONTROL   equ   0   ;; Endpoint type: Control
ET_ISO       equ   1   ;; Endpoint type: Isochronous
ET_BULK      equ   2   ;; Endpoint type: Bulk
ET_INT       equ   3   ;; Endpoint type: Interrupt

public     DeviceDscr, DeviceQualDscr, HighSpeedConfigDscr, FullSpeedConfigDscr, StringDscr, UserDscr
;DSCR   SEGMENT   CODE  PAGE
CSEG         AT 0X0100
DeviceDscr:
    db   DSCR_DEVICE_LEN ;; Descriptor length
    db   DSCR_DEVICE   ;; Decriptor type
    dw   0002H     ;; Specification Version (BCD)
    db   00H       ;; Device class
    db   00H       ;; Device sub-class
    db   00H       ;; Device sub-sub-class
    db   64        ;; Maximum packet size
    dw   0B404H     ;; Vendor ID
    dw   0410H     ;; Product ID (Sample Device)
    dw   0000H     ;; Product version ID
    db   1         ;; Manufacturer string index
    db   2         ;; Product string index
    db   0         ;; Serial number string index
    db   1         ;; Number of configurations

DeviceQualDscr:
    db   DSCR_DEVQUAL_LEN   ;; Descriptor length
    db   DSCR_DEVQUAL   ;; Decriptor type
    dw   0002H     ;; Specification Version (BCD)
    db   00H       ;; Device class
    db   00H       ;; Device sub-class
    db   00H       ;; Device sub-sub-class
    db   64        ;; Maximum packet size
    db   1         ;; Number of configurations
    db   0         ;; Reserved

HighSpeedConfigDscr:
    db   DSCR_CONFIG_LEN            ;; Descriptor length
    db   DSCR_CONFIG                ;; Descriptor type
    db   (HighSpeedConfigDscrEnd-HighSpeedConfigDscr) mod 256 ;; Total Length (LSB)
    db   (HighSpeedConfigDscrEnd-HighSpeedConfigDscr)  /  256 ;; Total Length (MSB)
    db   1     ;; Number of interfaces
```

```
    db  1     ;; Configuration number
    db  0     ;; Configuration string
    db  00100000b  ;; Attributes (b7 - buspwr, b6 - selfpwr, b5 - rwu)
    db  50    ;; Power requirement (div 2 ma)

;; Interface Descriptor
    db  DSCR_INTRFC_LEN    ;; Descriptor length
    db  DSCR_INTRFC        ;; Descriptor type
    db  0             ;; Zero-based index of this interface
    db  0             ;; Alternate setting
    db  2             ;; Number of end points
    db  0ffH          ;; Interface class
    db  00H           ;; Interface sub class
    db  00H           ;; Interface sub sub class
    db  0             ;; Interface descriptor string index

;; Endpoint Descriptor
    db  DSCR_ENDPNT_LEN    ;; Descriptor length
    db  DSCR_ENDPNT        ;; Descriptor type
    db  02H           ;; Endpoint number, and direction
    db  ET_BULK           ;; Endpoint type
    db  00H           ;; Maximun packet size (LSB)
    db  02H           ;; Max packect size (MSB)
    db  00H           ;; Polling interval

;; Endpoint Descriptor
    db  DSCR_ENDPNT_LEN    ;; Descriptor length
    db  DSCR_ENDPNT        ;; Descriptor type
    db  86H           ;; Endpoint number, and direction
    db  ET_BULK           ;; Endpoint type
    db  00H           ;; Maximun packet size (LSB)
    db  02H           ;; Max packect size (MSB)
    db  00H           ;; Polling interval

HighSpeedConfigDscrEnd:

FullSpeedConfigDscr:
    db  DSCR_CONFIG_LEN          ;; Descriptor length
    db  DSCR_CONFIG              ;; Descriptor type
    db  (FullSpeedConfigDscrEnd-FullSpeedConfigDscr) mod 256 ;; Total Length (LSB)
    db  (FullSpeedConfigDscrEnd-FullSpeedConfigDscr)  /  256 ;; Total Length (MSB)
    db  1     ;; Number of interfaces
    db  1     ;; Configuration number
    db  0     ;; Configuration string
    db  10000000b  ;; Attributes (b7 - buspwr, b6 - selfpwr, b5 - rwu)
    db  50    ;; Power requirement (div 2 ma)

;; Interface Descriptor
    db  DSCR_INTRFC_LEN    ;; Descriptor length
    db  DSCR_INTRFC        ;; Descriptor type
    db  0             ;; Zero-based index of this interface
    db  0             ;; Alternate setting
    db  2             ;; Number of end points
    db  0ffH          ;; Interface class
    db  00H           ;; Interface sub class
    db  00H           ;; Interface sub sub class
    db  0             ;; Interface descriptor string index

;; Endpoint Descriptor
    db  DSCR_ENDPNT_LEN    ;; Descriptor length
    db  DSCR_ENDPNT        ;; Descriptor type
    db  02H           ;; Endpoint number, and direction
    db  ET_BULK           ;; Endpoint type
    db  40H           ;; Maximun packet size (LSB)
    db  00H           ;; Max packect size (MSB)
    db  00H           ;; Polling interval
```

```
;; Endpoint Descriptor
    db   DSCR_ENDPNT_LEN     ;; Descriptor length
    db   DSCR_ENDPNT         ;; Descriptor type
    db   86H                 ;; Endpoint number, and direction
    db   ET_BULK             ;; Endpoint type
    db   40H                 ;; Maximun packet size (LSB)
    db   00H                 ;; Max packect size (MSB)
    db   00H                 ;; Polling interval

FullSpeedConfigDscrEnd:
StringDscr:

StringDscr0:
    db   StringDscr0End-StringDscr0     ;; String descriptor length
    db   DSCR_STRING
    db   09H,04H
StringDscr0End:

StringDscr1:
    db   StringDscr1End-StringDscr1     ;; String descriptor length
    db   DSCR_STRING
    db   'I',00
    db   'I',00
    db   'E',00
StringDscr1End:

StringDscr2:
    db   StringDscr2End-StringDscr2     ;; Descriptor length
    db   DSCR_STRING
    db   'M',00
    db   'C',00
    db   'U',00
    db   'F',00
    db   '_',00
    db   'x',00 ;; String por default, posteriormente puede ser cambiada utilizando
    db   'x',00 ;; DIPSwitch y comando o retirando la alimentacion.
    db   'x',00
StringDscr2End:

UserDscr:
    dw   0000H
    end
```

Fuentes bibliográficas

1.- USB Complete
Third edition
Jan Axelson
Lakeview Research LLC

2.- USB Design by example, A practical guide to building I/O devices
Second edition
John Hyde
Intel Press

Fuentes en internet

1.- Introduction to the EZ-USB FX2 GPIF Engine
Cypress Application note AN1115
http://www.cypress.com/design/AN1115
http://download.cypress.com.edgesuite.net/design_resources/application_notes/contents/introduction_to_the_ez_usb_fx2_tm__gpif_engine___an1115_12.pdf

2.- SDR SDRAM CONTROLLER
Lattice Reference design RD1010
http://www.latticesemi.com/products/intellectualproperty/referencedesigns/advancedsdrsdramcontrolle.cfm
http://www.latticesemi.com/dynamic/view_document.cfm?document_id=3467

3.- DDR SDRAM CONTROLLER
Lattice Reference design RD1020
http://www.latticesemi.com/products/intellectualproperty/referencedesigns/ddrsdramcontroller.cfm
http://www.latticesemi.com/dynamic/view_document.cfm?document_id=8492

4.- ANALYZING AND IMPLEMENTING SDRAM AND SGRAM CONTROLLERS
Christian Green
EDN Magazine article
http://www.edn.com/archives/1998/020298/03df_04.htm

5.- OPB Double Data Rate SDRAM controller
Xilinx application note
http://www.xilinx.com/support/documentation/ipembedprocess_memoryinterface_opb-ddr-sdramcontrol.htm
www.xilinx.com/support/documentation/ip_documentation/opb_ddr.pdf

6.- EZ-USB Technical Reference Manual V1.4
Cypress Semiconductor Corporation.
http://www.cypress.com/design/TR10001
http://download.cypress.com.edgesuite.net/design_resources/technical_reference_manuals/contents/ez_usb_r___technical_reference_manual__trm__14.pdf

7.- EZ-USB FX2 GPIF Primer
Cypress Application note AN1197
http://www.cypress.com/design/AN1197
http://download.cypress.com.edgesuite.net/design_resources/application_notes/contents/ez_usb_fx2_gpif_primer___an1197_12.pdf

8.- Especificación USB 2.0
http://www.usb.org/developers/docs/usb_20_040908.zip

10.- Hoja de datos de la Memoria SDRAM
http://download.micron.com/pdf/datasheets/dram/sdram/256MSDRAM.pdf