



**UNIVERSIDAD NACIONAL DE SAN JUAN
FACULTAD DE INGENIERIA**

**“PROCESAMIENTO DIGITAL DE SEÑALES Y VISUALIZACIÓN EN
TIEMPO REAL CON FUNCIONES DE MATLAB ADQUIRIDAS POR FPGA-
USB 3.0”**

**ALUMNO:
GABRIEL FRANCISCO NAVAS NALE**

ASEORES:

**MG. ING. CRISTIAN SISTERNA– Dr. Lic. MARÍA LIZ CRESPO- DR. ING. MARCELO
SEGURA.
AÑO: 2016**

**DEPARTAMENTO DE ELECTRONICA Y AUTOMATICA
AVENIDA SAN MARTIN 1109 OESTE
CP J5400ARL SAN JUAN – ARGENTINA
TEL 0264 4211700 – INTERNO 354 – 360**

Agradecimiento

A mi familia, en especial a mis padres Guillermo y Mabel, mis hermanas Verónica y Bárbara, que siempre me brindaron su fe, fuerza y medios para poder estudiar

A mi novia Melina Leiva que me apoyo incondicionalmente y fue pilar para lograr llegar a este momento tan esperado.

A mi amigo Ing. Eduardo Granero por ayudarme en los momentos de dificultad al inicio de esta tesis.

A mi profesor y director Mg. Ing. Cristian Sisterna por el apoyo, la flexibilidad y ayuda brindada durante la carrera y mi tesis.

A mis compañeros y amigos que compartimos momentos de estudio, angustia y felicidad durante el transcurso de la carrera.

A Dios.

Índice de contenidos

Introducción	viii
Objetivos	ix
Desarrollo	1
Capítulo 1: Componentes utilizados	1
1.1 - Kit de Cypress CYUSB3KIT-001	1
1.2 - Placa de desarrollo ZedBoard	2
1.3 - Placa con conversor A/D Pmod AD5	3
1.4 - Placa de interconexión FMC	4
Capítulo 2: El USB	6
2.1 - Origen y evolución del USB	6
2.2 - Fundamentos del sistema USB 2.0	6
2.3 - USB 3.0: Diferencias y mejoras respecto del USB 2.0	13
Capítulo 3: Desarrollo de la interface USB - Visual C++ - MatLab	20
3.1 Librería CyAPI.lib de Cypress	20
3.2 Creación del programa VS2010	20
3.3 Creación de interface MatLab – VS 2010	28
Capítulo 4: Interface conversor ADC - FPGA – EZ USB FX3 – USB	39
4.1 Interface FIFO Esclavo	39
4.2 Transferencias, hilos y zócalos	43
4.3 Firmware del FX3 y código VHDL del FPGA	44
Capítulo 5: Prueba del sistema y resultados obtenidos	63
Conclusiones	66
Bibliografía	68

Índice de figuras:

Figura 1: Diagrama de bloques conexión de sistema completo.....	1
Figura 2: Kit CYUSB3KIT-001	2
Figura 3: Kit de desarrollo ZedBoard	3
Figura 4: Pmod AD5	4
Figura 5: Conector FMC.....	4
Figura 6: Placa ZedBoard y CYUSB3FX con FMC.....	5
Figura 7 : Paquete de transferencia de salida	9
Figura 8: Transferencias masivas de entrada y salida	11
Figura 9: Transferencia de interrupción.....	11
Figura 10: Transferencia asincrónica	11
Figura 11: Transferencia de control.....	11
Figura 12: Capa de protocolo	14
Figura 13: Estructura de un paquete de gestión de enlace.....	15
Figura 14: Paquete de transacción ACK.....	15
Figura 15: Ejemplo de un paquete de datos.....	16
Figura 16: Estructura de un ITP	16
Figura 17: Creando un nuevo proyecto en VS2010.....	21
Figura 18: Añadir la CyAPI.lib al proyecto	22
Figura 19: Configuración adicional del proyecto VS2010.....	23
Figura 20: Configuración de propiedades del proyecto.....	23
Figura 21: Aplicación Streamer ejemplo	27
Figura 22: Aplicación ejemplo 1 modificada con engine.....	32
Figura 23: Grafica 1 de programa ejemplo 1	32
Figura 24: Grafica 2 de programa ejemplo 1.....	33
Figura 25: Programa Tesis Osciloscopio	37
Figura 26: Diagrama de bloques de interface ADC -FPGA - EZ USB FX3 – USB	39
Figura 27: Diagrama de la Interface FIFO M/E	40
Figura 28: Asignación de pines del FX3 para la FIFO.	42
Figura 29: Secuencia de acceso a la FIFO y diagrama de tiempos.	43
Figura 30: Importar proyecto en Eclipse EZ USB Suite.....	45
Figura 31: Seleccionar proyecto existente al Workspace.....	45
Figura 32: Configuración de jumpers en la placa del FX3.....	46
Figura 33: Maquina de estados del FPGA para la interface FIFO	47
Figura 34: Maquina de estados del FPGA en STREAM IN.....	49
Figura 35: Pines de conexión entre FPGA y FX3	51
Figura 36: Captura de StreamIN mediante aplicación BulkLoop de Cypress.	52
Figura 37: Aplicación típica del código spi_master	53
Figura 38: Diagrama de tiempos y señales de control del SPI Master.	53
Figura 39: Diagrama de estados del bloque CONTROL	54
Figura 40: IPs de CONTROL y SPI Master conectados.	55
Figura 41: Bloques SPI Master - CONTROL - Control_p5mod para probar	55
Figura 42: Sistema completo del FPGA.....	56
Figura 43: Grafica probando el sistema con Pmod AD5 - 1.....	57
Figura 44: Grafica probando el sistema con Pmod AD5 - 2.....	57
Figura 45: Diagrama de bloques del conversor XADC.....	58
Figura 46: Puertos y diagrama de bloques del XADC	59
Figura 47: Descripción de los puertos del XADC - 1	59

Figura 48: Descripción de los puertos del XADC - 2	60
Figura 49: Diagrama de tiempos para conversión continua del XADC.....	61
Figura 50: Sistema completo con conversor XADC	62
Figura 51: Sistema completo conectado	63
Figura 52: Grafica en tiempo real con señal de 100Hz y 500mV div	64
Figura 53: Grafica en tiempo real con señal de 100Hz y 200mV div	64
Figura 54: Grafica en tiempo real con señal de 100Hz y 100mV div	65
Figura 55: Grafica en tiempo real con señal de 1000Hz y 100mV div	65

Introducción

En el presente trabajo se expone el diseño e implementación de un sistema de adquisición, procesamiento y visualización de señales en tiempo real en PC. Este mismo va a ser utilizado en el Instituto de Investigaciones Antisísmicas (IDIA), en donde se cuenta con una mesa vibratoria en la cual se llevan a cabo ensayos de calificación sísmica de distintos sistemas de construcción o sistemas eléctricos. En estos ensayos la cantidad de sensores es elevada y muy variada. Los datos de los sensores deben ser procesados en tiempo real a fin de realizar la correspondiente acción de control sobre los actuadores que mueven la mesa vibratoria. Por lo tanto, en este caso un sistema como el que se plantea en este trabajo final es de suma utilidad. El uso de herramientas tales como MatLab y de hardware como FPGA y USB 3.0 hacen que sea un sistema sumamente portable, actual, de alta velocidad, usando hardware del estado del arte. Por ello este trabajo tiene una finalidad práctica sumamente útil y se aplican técnicas de avanzada.

Para este trabajo, se utilizó el conversor analógico digital Pmod AD5 con interface SPI el cual, transmite los datos convertidos al FPGA incluido en la placa de desarrollo ZedBoard de Xilinx. Esta última los envía de forma continua, haciendo uso de una interface FIFO esclava, al EZ USB FX3 de Cypress, que mediante su interface USB los transmite a la PC. Allí, son adquiridos mediante la librería CyAPI.lib, implementada en un programa de Visual Studio 2010 (VS 2010) bajo el lenguaje Visual C++, en donde los datos recibidos. Mediante el motor Engine de MatLab se grafican los datos, utilizando funciones de este programa desde el VS 2010. Después de varias pruebas se decidió cambiar el conversor Pmod AD5 por el conversor XADC, incluido en el FPGA de la ZedBoard, por problemas de velocidad de conversión que dificultaban el normal funcionamiento de la transmisión de datos desde el FPGA hasta la PC.

La idea de este trabajo fue motivada por un proyecto de la materia optativa de la carrera electrónica Sistemas Digitales Avanzados, dictada por el Director de este proyecto Mg. Cristian Sisterna. En esta materia se estudia y aprende a diseñar, configurar e implementar sistemas digitales basados en tecnología FPGA, en lenguaje de descripción de hardware VHDL.

A lo largo de los capítulos escritos del presente trabajo se desarrollan temas como: la explicación del protocolo USB y su evolución hasta el USB 3.0; el uso de Visual Studio 2010 junto con la librería CyAPI.lib de Cypress que permite la comunicación con dispositivos de la empresa que utilizan esta funcionalidad. A continuación, se detalla cómo implementar el Motor Engine de MatLab en lenguaje Visual C++ para trabajar datos en MatLab desde Visual Studio. Luego, se explica el funcionamiento del Kit EZ USB FX3 para el envío de datos a través de la comunicación USB. Finalmente se detalla la implementación de una interface de alta velocidad FIFO esclava entre el FPGA y el FX3. En dicha interface se transmiten los datos convertidos desde el ADC. Por ultimo en la conclusión, se detallan los resultados obtenidos, los desafíos que fueron encontrándose a medida que se realizaba el trabajo, así como sugerencias para la mejora y continuación del proyecto

Objetivos

Objetivo general

- Adquirir, procesar datos en tiempo real en PC, los cuales provienen de señales de distintos tipos de sensores adquiridas en un sistema basado en FPGA y transmitidas a PC por USB 3.0.
- Visualizar datos adquiridos usando las funciones de MatLab.

Objetivos específicos e hipótesis de trabajo

- Desarrollo del software de comunicación USB 3.0 entre el Kit de microcontrolador Cypress FX3 y el sistema operativo Windows.
- Desarrollo del software de visualización entre el programa de Visual C++ y las funciones de gráfication de MatLab.
- Desarrollo del software de comunicación entre el Kit de microcontrolador Cypress FX3 y la placa de desarrollo FPGA ZedBoard de Xilinx.
 - Desarrollo del software del microprocesador ARM del Cypress FX3
 - Desarrollo del VHDL en el FPGA
- Desarrollo del VHDL de comunicación entre el FPGA y el conversor ADC con protocolo SPI.
- Montaje de todas las placas, conexiones y configuraciones.

Desarrollo

Capítulo 1: Componentes utilizados

A lo largo de este capítulo abordaremos y desenvolveremos los distintos materiales y componentes utilizados para el desarrollo de la tesis, entre los que se encuentran: el kit de Cypress CYUSB3KIT-001; la placa de desarrollo FPGA ZedBoard; la placa con conversor A/D Pmod AD5; y la placa de interconexión FMC.

Estos diversos componentes se conectan del modo detallado en la Figura 1:

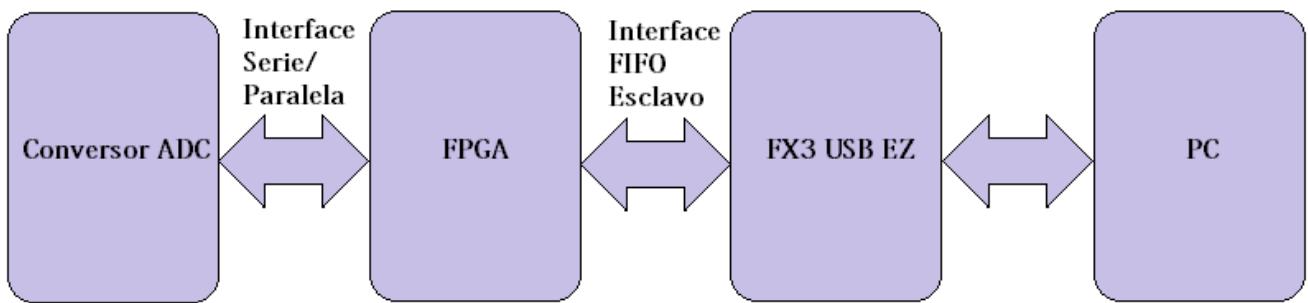


Figura 1: Diagrama de bloques conexión de sistema completo

1.1 - Kit de Cypress CYUSB3KIT-001

El kit de desarrollo utilizado de Cypress cuenta con un microcontrolador EZ-USB FX de 32-bit basado en un procesador ARM926EJ-S, contiene todos los elementos necesarios para aprender y desarrollar aplicaciones con comunicación USB 2.0 y 3.0. Incluye un cd con toda la información pertinente para su utilización (programas ejemplos, documentación del microcontrolador, librería USB y manual de programador de la misma.). Además, incluye una fuente de 5V, el cable USB 3.0, y la placa PCB con el EZ-USB FX. En la Figura 2 se puede observar el contenido del kit.



Figura 2: Kit CYUSB3KIT-001

El microcontrolador que posee este kit cuenta con un Bus de Serie Universal (USB) integrado, el cual es compatible con las especificaciones USB 2.0 y 3.0. Posee, además, una Interface General Programable (GPIF II) de 100MHz que permite la conexión con distintos periféricos que poseen 8/16 y 32 bits de bus de datos y 16 señales de control, conectividad mediante puertos UART, SPI, I2S. Finalmente, el microcontrolador cuenta con una CPU de 32 bits ARM926EJ de 200MHz con 512KB de SRAM embebida. La información detallada puede encontrarse en <http://www.cypress.com/products/ez-usb-fx3-superspeed-usb-30-peripheral-controller>

1.2 - Placa de desarrollo ZedBoard

La placa ZedBoard de la empresa Xilinx, es una placa para el desarrollo de aplicaciones en tecnología FPGA de la familia Zynq-7000 de Xilinx. Esta posee un procesador dual-core ARM Cortex A9 frecuencia de CPU de hasta 1GHz; 32 KB de memoria cache nivel 1 y 512 KB de memoria cache nivel 2; memoria ROM de booteo embebida; 512 KB de RAM; interface de 16 y 32 bits para memorias DDR2, DDR3, DDR3L y LPDDR2; controlador de acceso directo a memoria de 8 canales y varias interfaces de comunicación como dos interfaces Ethernet 10/100/1000 estandar IEEE 802.3; dos interfaces OTG USB; dos controladores de bus CAN 2.0; dos controladores compatibles con SD/SDIO 2.0/MMC3.31, entre otros. La placa incluye diversos conectores de expansión que le

permite conectarse con distintos periféricos tales como micro controladores, placas de adquisición de datos, etc. La placa incluye 512MB de RAM DDR3, una interface de 10/100/1000 Mb Ethernet, un conector USB-JTAG para programación del FPGA. Además, posee múltiples displays y conexiones como HDMI 1080p, 8-Bit VGA, 128x32 Oled, conversor ADC de 1Msps, interface de audio y video, USB-OTG y USB-UART.

El kit de desarrollo (Figura 3) incluye la placa ZedBoard, la fuente de alimentación, el cable USB para programar y una tarjeta SD.



Figura 3: Kit de desarrollo ZedBoard

El FPGA de la familia Zynq-7000 es un SoC (Chip en Sistema según sus siglas en inglés) programable modelo XC7Z020-CLG484-1 el cual posee 85000 celdas lógicas programables, 53200 LUTs, 106400 Flip-Flops, 140 bloques de 36Mb de RAM, 220 bloques DSP y 150 PS-I/O.

1.3 - Placa con conversor A/D Pmod AD5

La placa Pmod AD5 (Figura 4) posee un conversor A/D de 24 bits de resolución con 4 canales de entrada diferenciales u 8 canales pseudo-diferenciales. Posee ganancia programable desde 1 hasta 128. Cuenta con 2 conectores SMA, comunicación SPI de hasta 1 Mhz y conector de salida de 6 pines Pmod compatibles con los conectores Pmod de la placa ZedBoard. La configuración del mismo y la información detallada de funcionamiento se encuentra en https://reference.digilentinc.com/pmod/pmod/ad5/ref_manual.



Figura 4: Pmod AD5

1.4 - Placa de interconexión FMC

La placa de interconexión FMC (Figura 5) permite la conexión de la placa de desarrollo ZedBoard con el kit EZ-USB FX a través de conectores específicos. Estos utilizan los GPIF II del conector FMC del kit de Cypress y el conector FMC de la placa de Xilinx como se ve en la Figura 6.



Figura 5: Conector FMC



Figura 6: Placa ZedBoard y CYUSB3FX con FMC

Capítulo 2: El USB

La comunicación USB es uno de los temas centrales de esta tesis, por lo tanto, se le dedicará un capítulo para la explicación breve de su historia, así como el protocolo, las características, funcionalidades y hardware.

2.1 - Origen y evolución del USB

El Bus Serie Universal (BUS), (en inglés: Universal Serial Bus), más conocido por la sigla USB, es un bus estándar que define los protocolos, conectores y cables usados en un bus para conectar, comunicar y brindar alimentación eléctrica a computadoras, periféricos y distintos dispositivos electrónicos.

El desarrollo del USB nace aproximadamente en el año 1990 en base a la necesidad de un grupo de empresas que buscaban estandarizar la forma de conexión de los periféricos a los equipos que, en aquella época, eran poco compatibles entre sí. Entre las corporaciones que integraban ese grupo se encontraban: Intel, Microsoft, IBM, Compaq, DEC, NEC y Nortel. La primera especificación completa 1.0 se publicó en 1996 por el Foro de Implementadores de BUS, pero no fue hasta 1998 con la especificación 1.1 que comenzó a utilizarse de forma masiva. Actualmente es utilizado como estándar de conexión de periféricos como: teclados, mouses, memorias, joysticks, cámaras, celulares, discos duros, impresoras, placas de sonido, etc. El éxito del USB fue tal que logró desplazar a las conexiones con conectores como el puerto serie, paralelo, PS/2, puerto de juegos, entre otros. Este reemplazo casi masivo se debe a varios factores, entre los principales se encuentran: la velocidad de transmisión, la facilidad de conexión mientras el equipo está encendido o “en caliente”, la robustez física del puerto y la simplicidad de configuración, prácticamente transparente para el usuario final.

Las distintas versiones del USB han evolucionado a través del tiempo y hoy en día comercialmente tenemos el **USB 1.0** (Baja velocidad) 188Kb/s, **USB 1.1** (Velocidad completa) 1,5MB/s, **USB 2.0** (Alta velocidad) 35MB/s y **USB 3.0** (Super alta velocidad) 600MB/s. En este proyecto, debido a las características de la placa de desarrollo de Cypress se desarrolló y utilizó en el USB 2.0 y 3.0, cuya descripción es desplegada en el próximo punto.

2.2 - Fundamentos del sistema USB 2.0

2.2.1 - Director, Dispositivos y Concentradores

Un sistema USB es un diseño de comunicación de serie asincrónica con “host-centrado”, el cual consiste en un solo anfitrión o director y varios dispositivos y concentradores conectados en una topología estrella por niveles. La especificación USB 2.0 soporta tasas de transferencia de datos baja, completa y alta velocidad. Emplea una comunicación con medio compartido de dos hilos con flujo de datos unidireccional con transiciones de la dirección del bus negociadas.

El sistema USB tiene un solo maestro, el director (normalmente una computadora). Los dispositivos implementan funciones específicas y transferencias de información hacia el director y desde el director. El director administra el bus y es el responsable de detectar un dispositivo así como también iniciar y manejar las transferencias entre dispositivos. Los concentradores son terminales que tienen un puerto de subida de flujo y múltiples puertos de bajada de flujo. Lo que permite conectar múltiples dispositivos con el director, creando una topología por niveles. Asociado al director se encuentra el controlador de director, que maneja la comunicación entre este y diversos dispositivos. Cada uno de estos controladores tiene un concentrador raíz asociado a él. Un máximo de 127 dispositivos pueden ser conectados al controlador director con no más de 7 niveles (incluido el concentrador raíz). Como el director es siempre el maestro del bus, la dirección USB de SALIDA u OUT, hace referencia a la dirección desde el director hacia el dispositivo, mientras que una ENTRADA o IN hace referencia a la dirección desde el dispositivo hacia el director.

2.2.2 – Tuberías y puntos de llegada

La transferencia de información USB puede darse entre el software director y una entidad lógica en el dispositivo llamada punto de llegada (End point en inglés) a través de un canal lógico. Un dispositivo USB puede tener hasta 32 canales lógicos activos, 16 para transferencias de salida y 16 de entrada. Una interface es una colección de puntos de llegada que funcionan juntos para implementar una función específica.

2.2.3 – Descriptores

Los dispositivos USB se describen a sí mismos al director usando una cadena de información (bytes) conocidos como descriptores (descriptors en inglés). Los descriptores contienen información, tales como la función que el dispositivo implementa, el fabricante, la cantidad de puntos de llegada e información de clase específica. Los primeros 2 bytes de cualquier descriptor especifican la longitud y el tipo respectivamente. En general, los dispositivos poseen los siguientes descriptores:

- Descriptores de dispositivo
- Descriptores de configuración
- Descriptores de interface
- Descriptores de puntos de llegada
- Descriptores de cadena

Un descriptor de dispositivo especifica el ID del producto (PID) e ID del vendedor (VID), así como también la revisión USB con la cual es compatible. Entre otra información enumerada se encuentran el número de configuraciones y el tamaño máximo del paquete para el punto de llegada 0. El sistema de cargas del director lee el VID y PID para cargar el controlador apropiado para el dispositivo. Un dispositivo USB puede tener sólo un descriptor de dispositivo asociado a él.

A su vez, el descriptor de configuración contiene diferentes datos: la característica de despertado remota del dispositivo, el número de interfaces que pueden existir para la configuración, y la potencia máxima para un uso particular de configuración. Una sola interface de dispositivo puede estar activa a la vez.

Cada función del dispositivo tiene un descriptor de interfaz asociada con él. Un descriptor de interfaz especifica el número de puntos de llegada asociados con la interfaz y otras configuraciones alternativas. Las funciones que se encuentran bajo una categoría predefinida son indicadas usando el código de clase de interfaz y el campo de código de subclase. Esto le permite al director cargar controladores de dispositivos estándar asociados con esa función. Más de una interface puede estar activa en un mismo momento. En este trabajo se detallan las interfaces y descriptores utilizados en el capítulo 4.3.1 Firmware del EZ USB FX3.

2.2.4 – Tipos de transferencias

El USB define cuatro tipos de transferencias a través de las tuberías. Estas coinciden con los requerimientos de diferentes tipos de información que necesitan ser entregadas a través del bus.

2.2.4.1 – Transferencia masiva

Puede denominarse a la información en masa como ‘ráfagas’, que viajan en paquetes de 8, 16, 32, o 64 bytes a velocidad máxima o 512 bytes a velocidad alta. La información en masa garantiza la precisión, a través de un mecanismo de reintento automático para reenviar información errónea. El director programa los paquetes masivos cuando existe tiempo disponible en el bus. Las transferencias masivas son usadas normalmente por impresoras, escáneres, módems, y dispositivos de almacenamiento. La información en masa tiene incorporado un flujo de control provisto por paquetes de intercambio.

2.2.4.2 – Transferencia por interrupción

La transferencia de la información por interrupción es similar al modo en que se envía la información en masa; puede tener paquetes de tamaños desde 1 hasta 64 bytes a velocidad máxima o hasta 1024 bytes a velocidad alta. Los puntos de llegada por interrupciones tienen un intervalo de sondeo asociado, el cual asegura que son revisados (reciben un testigo de entrada (IN TOKEN)) por el director con cierta frecuencia.

2.2.4.3 – Transferencia asincrónica

La información asincrónica es crítica en tiempo y usa un flujo de datos similar al de audio y video. Un paquete asincrónico puede contener hasta 1023 bytes a máxima velocidad o hasta 1024 en alta velocidad. El tiempo de entrega es un requerimiento importante para la información asincrónica. En cada trama USB, una cierta cantidad de

ancho de banda es reservado para las transferencias asincrónicas. Para aliviar las sobrecargas, las transferencias asincrónicas no tienen intercambio ni retransmisión, la corrección de error está limitada a 16-bit CRC.

2.2.4.4 - Transferencia de control

Las transferencias de control configuran y envían comandos a un dispositivo. Como son muy importantes, utilizan el control de error más complejo (en bits). El director reserva una porción de cada trama USB para las transferencias de control.

2.2.5 - Capa de protocolo

La función de la capa de protocolo es interpretar el tipo de transferencia, crear los paquetes IDs y encabezados necesarios, los paquetes de información de longitud, generar los CRCs, y pasar todo a la capa de enlace. Las decisiones de nivel de protocolo similares a los paquetes de retransmisión también son generadas en esta capa.

Todas las comunicaciones sobre el USB suceden en forma de paquetes. Cada paquete USB, consiste en un Paquete ID (PID). Estos PIDs pueden caer en una de cuatro categorías diferentes y son enumeradas a continuación:

- Testigo: IN, OUT, SOF, SETUP
- Información: DATA0, DATA1, DATA2, MDATA
- Intercambio: ACK, NAK, STALL, NYET
- Especial: PRE, ERR, SPLIT, PING

Una transferencia de datos de carga útil (payload) normal, requiere al menos de tres paquetes: Testigo, Información y Ack. La **Figura 7** ilustra una transferencia USB de salida. El paquete 1 es un testigo de salida, indicado por PID de salida. El testigo de salida indica que la información del director va a ser transmitida por el bus. El paquete 2 contiene información como indica el PID DATA1. El paquete 3 es un paquete de intercambio, enviado por el dispositivo usando el PID ACK (acknowledge, reconocimiento) para indicarle al director que el dispositivo recibió la información libre de errores. Continuando con la Figura 7, una segunda transacción comienza con otro Testigo de salida (4), seguido de más información (5), esta vez usando el PID DATA0. Finalmente, el dispositivo indica que las transacciones se llevaron a cabo con éxito mediante la transmisión del PID ACK en un paquete de intercambio (6).

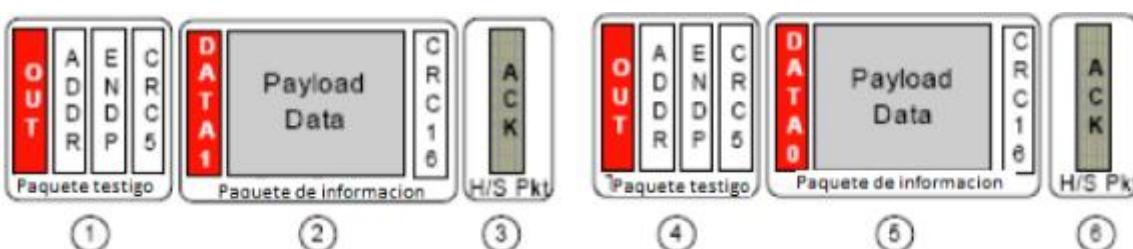


Figura 7 : Paquete de transferencia de salida

Los testigos de configuración (SETUP) se utilizan únicamente para transferencia de control. Tienen un prefacio de 8 bytes de información del cual los periféricos decodifican los pedidos del director. A velocidad máxima, los testigos de comienzo de trama (SOF) ocurren una vez por milisegundo. A velocidad alta, cada trama contiene ocho testigos SOF, que denotan una micro trama de 125 uS.

Cuatro PIDs de intercambio indican el estado de la transferencia USB: ACK (Reconocimiento) significa “éxito”, es decir, que la información se recibió libre de errores; NAK (reconocimiento negativo) significa “ocupado, intentar de vuelta”. Es tentador asumir que el NAK significa “error”, pero no es así. Un dispositivo USB indica error al no responder.

STALL significa que algo está mal (probablemente como resultado de una falta de comunicación o falta de cooperación entre el director y el software del dispositivo). Un dispositivo envía el intercambio STALL para indicar que no entiende el pedido del dispositivo, que algo se salió mal en el extremo de periférico, o que el director trata de acceder a una fuente que no existe. Es similar a HALT, pero con mayores posibilidades, porque el USB provee una forma de recuperarse de un stall. NYET (no aun) tiene el mismo significado que el ACK – que la información se recibió sin errores – pero también indica que el punto de llegada no está listo aun para recibir otra transferencia de salida. Los PIDs NYET suceden solo en el modo de velocidad alta. Un PID PRE (preámbulo) precede a una transmisión USB de velocidad baja (1.5Mbps).

Una característica notable del protocolo USB 2.0 es el mecanismo de alternado de información. Hay dos PIDs DATA (DATA0 y DATA1) en la Figura 7. Como se mencionó anteriormente, el intercambio ACK indica al director que el periférico recibió la información libre de errores (la porción CRC del paquete es usada para detectar errores). Sin embargo, el paquete de intercambio puede quedar ilegible durante la transmisión. Para detectarlo, cada lado (director y dispositivo) mantienen un bit de alternado de información, el cual es intercambiado entre las transferencias de paquetes de información. El estado de este bit de alternado interno es comparado con el PID que llega con la información, ya sea DATA0 o DATA1. Cuando se envía información, el director o el dispositivo envían PIDS DATA0-DATA1 alternados. Mediante la comparación del PID de la información recibida con el estado de su propio bit de alternado interno, el receptor puede detectar un paquete de intercambio corrupto.

El protocolo PING fue introducido en la especificación USB 2.0 para evitar el gasto de ancho de banda del bus bajo ciertas circunstancias. Cuando opera a velocidad máxima, cada transferencia de salida envía la información de salida, aun cuando el dispositivo está ocupado y no puede aceptar información. Tales transferencias masivas repetitivas fracasadas, resultan en un significante desperdicio de ancho de banda del bus. Realizando esto a velocidad alta puede volverse aún peor. Este problema fue remediado mediante el uso del nuevo PID PING. El director envía primero un testigo PING corto hacia un punto de llegada de salida, preguntando si hay lugar para información de salida en el dispositivo periférico. Sólo cuando el PING obtiene como respuesta un ACK, el director envía el testigo de salida e información.

El protocolo para transferencias de: interrupción, masivas, asincrónicas y de control, explicadas anteriormente, son ilustradas en las Figura 8, Figura 9, Figura 10 y Figura 11.

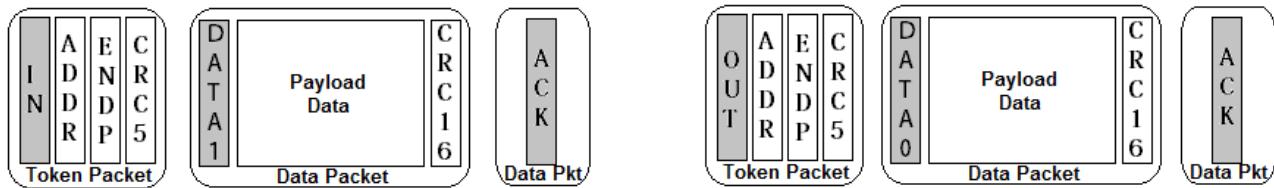


Figura 8: Transferencias masivas de entrada y salida

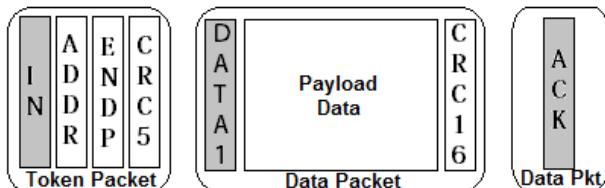


Figura 9: Transferencia de interrupción

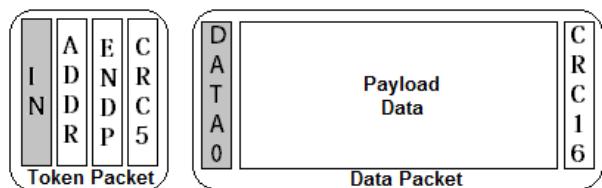


Figura 10: Transferencia asincrónica

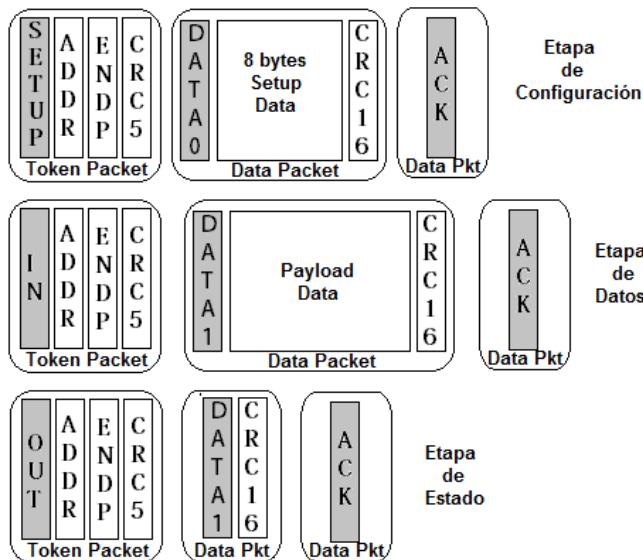


Figura 11: Transferencia de control

2.2.6 – Capa de enlace/física

La capa de enlace lleva a cabo tareas adicionales para incrementar la confiabilidad de la transferencia de información. Esto incluye el ordenamiento de bytes, y el tramado de nivel de líneas, etc.

Más comúnmente conocida como la interface eléctrica del USB 2.0, esta capa consiste en los circuitos para serializar y des-serializar información, circuitos pre y post ecualizadores para manejar y detectar señales diferenciales en las líneas D+ y D-. Todo el

manejo de errores es realizado por la capa de protocolo y no hay capa de datos de bajo nivel discernible para administrar errores.

2.2.6.1 - Detección y enumeración de dispositivos

Una de las ventajas más importantes del USB, sobre otros sistemas de comunicaciones contemporáneos, es su capacidad de plug and play (conexión y uso). Un cambio en el final del puerto USB indica que un dispositivo USB ha sido conectado.

Cuando un dispositivo es conectado por primera vez, el director trata de aprender sobre el dispositivo a través de sus descriptores; este proceso se llama enumeración. El director pasa por la siguiente secuencia de registro:

- 1 *El director envía un pedido de obtención de descriptor de dispositivo a la dirección cero (todos los dispositivos USB deben responder a la dirección cero cuando son conectados la primera vez).*
- 2 *El dispositivo responde al pedido enviando información de identificación de vuelta hacia el director para identificarse a sí mismo.*
- 3 *El director envía un pedido de configuración de dirección, el cual le asigna una dirección única al dispositivo recientemente conectado para que pueda ser distinguido de otros dispositivos conectados al bus.*
- 4 *El director envía más pedidos de obtención de directores, consultando por información adicional del dispositivo. Desde aquí, aprende todo sobre el dispositivo como cantidad de puntos de llegada, requerimientos de energía, ancho de banda requerido, y que controlador cargar.*

Todos los dispositivos de velocidad alta comienzan el proceso de enumeración a velocidad máxima; los dispositivos cambian al modo de operación de velocidad alta solo después de que el director y el dispositivo se ponen de acuerdo en trabajar esta velocidad. El proceso de negociación de velocidad alta ocurre durante el reinicio del USB mediante el protocolo “Chirp” (Chirrido).

Como la configuración del FX3 es adaptable, un solo chip puede tomar las identidades de múltiples y diferentes dispositivos USB. Cuando se conecta en el USB, el FX3 se enumera automáticamente y descarga el firmware, así como las tablas de descriptores USB a través del cable USB. Una desconexión de software es ejecutada, seguida de una nueva enumeración del FX3, esta vez el dispositivo se define por la información descargada. Este proceso de dos pasos patentado, llamado Re-Enumeración TM, sucede instantáneamente cuando el dispositivo es conectado, y no existe indicio alguno de que hubiera ocurrido el primer paso de descarga inicial.

2.2.7 – Administración de energía

La administración de la energía se refiere a la sección de la especificación USB, que explica cómo la energía es asignada a los dispositivos conectados en cascada y cómo diferentes capas de comunicación pueden funcionar para hacer un mejor uso de la energía del bus bajo diferentes circunstancias.

El USB 2.0 soporta dispositivos tanto autoalimentados o alimentados por el bus. El dispositivo indica esto a través de sus descriptores. Los dispositivos, independientemente de sus requisitos y capacidades de potencia, son configurados en modo de baja potencia a menos que el software le ordene al director que configure el dispositivo en el estado de alta potencia. Los dispositivos de baja potencia pueden consumir hasta 100 mA de corriente y dispositivos de alta potencia pueden consumir hasta un máximo de 500 mA.

El director USB puede “suspender” a un dispositivo poniéndolo en un estado de modo apagado. Un estado alto de 3 ms (diferencial ‘1’ indicado por D+ alto D- bajo) en el bus USB desencadena que el director realice un pedido de suspensión y entre al modo de bajo consumo. Los dispositivos USB se les pide que entren a un estado de bajo consumo en respuesta a este pedido.

Cuando es necesario, el dispositivo o el director emiten una reanudación. Una señal de reanudación es iniciada llevando el bus a un estado ‘K’, pidiendo que el director o el dispositivo salga de su estado de bajo consumo “suspendido”. Un dispositivo USB solo puede emitir una señal de resumen si se ha reportado (a través de sus descriptores de configuración) que tiene la capacidad remota de auto activarse, y solo si el director ha permitido esta capacidad de ese dispositivo.

Este mecanismo de suspensión-reanudación minimiza la energía consumida cuando no hay actividad sobre el bus USB.

2.2.8 – Clases de dispositivo.

En un intento de simplificar el desarrollo de nuevos dispositivos, las funciones de dispositivo comúnmente usadas fueron identificadas y controladores nominales fueron desarrollados para soportar estos dispositivos. El director usa la información en el código de clase, código de subclase, código de protocolo del dispositivo y descriptores de interface para identificar si controladores incorporados pueden ser cargados para comunicarse con el dispositivo conectado. La clase dispositivo de interface humana (HID) y clase de almacenamiento masivo (MSC) son algunas de las clases de dispositivos comúnmente usadas.

La clase HID se refiere a dispositivos interactivos tales como el mouse, el teclado y joysticks. Esta interface usa transferencias del tipo control e interrupción para transferir información porque las velocidades de transferencia no son críticas. La información es enviada o recibida usando reportes HID. El dispositivo o el descriptor de interface contiene el código de clase HID.

La clase MSC está destinada principalmente para transferir información a dispositivos de almacenamiento. Esta interface usa principalmente transferencias del tipo masivas para enviar información. Al menos dos puntos de llegada masivos por cada dirección son necesarios. La clase MSC utiliza el conjunto de comandos SCSI transparentes para leer o escribir sectores de datos de la unidad de disco.

2.3 - USB 3.0: Diferencias y mejoras respecto del USB 2.0

2.3.1 - Motivación para el USB 3.0

El USB 3.0 es la siguiente etapa de la tecnología USB. Su meta principal es proveer la misma forma de uso, flexibilidad, y rápida funcionalidad de conexión pero una tasa de datos mucho más alta. Otra meta principal del USB 3.0 es la administración de energía. Esto es importante para las aplicaciones “sincronizar y arrancar” que necesitan hacer concesiones sobre características para la vida de la batería.

La interface USB 3.0 consiste en bus físico SuperSpeed (Super velocidad) además del bus físico USB 2.0. El estándar USB 3.0 define un mecanismo de señalización simple dual a una tasa de 5 Gbits/s.

Inspirado por el PCI Express y la arquitectura OSI de 7 capas, el protocolo USB 3.0 se abstrae en diferentes capas que son ilustradas en las siguientes secciones.

En este documento, el USB 3.0 se refiere implícitamente a la porción de SuperSpeed USB 3.0.

2.3.2 – Capa de protocolo USB

El USB 3.0 SuperSpeed hereda los tipos de transferencia de su predecesor reteniendo el modelo de tuberías, puntos de llegada y paquetes. No obstante, los tipos de paquetes usados y algunos protocolos asociados a las transferencias masivas, de control y asincrónicas han sufrido algunos cambios y mejoras. Estos son discutidos en las secciones que siguen. Se puede observar en la Figura 12 la capa de protocolo.

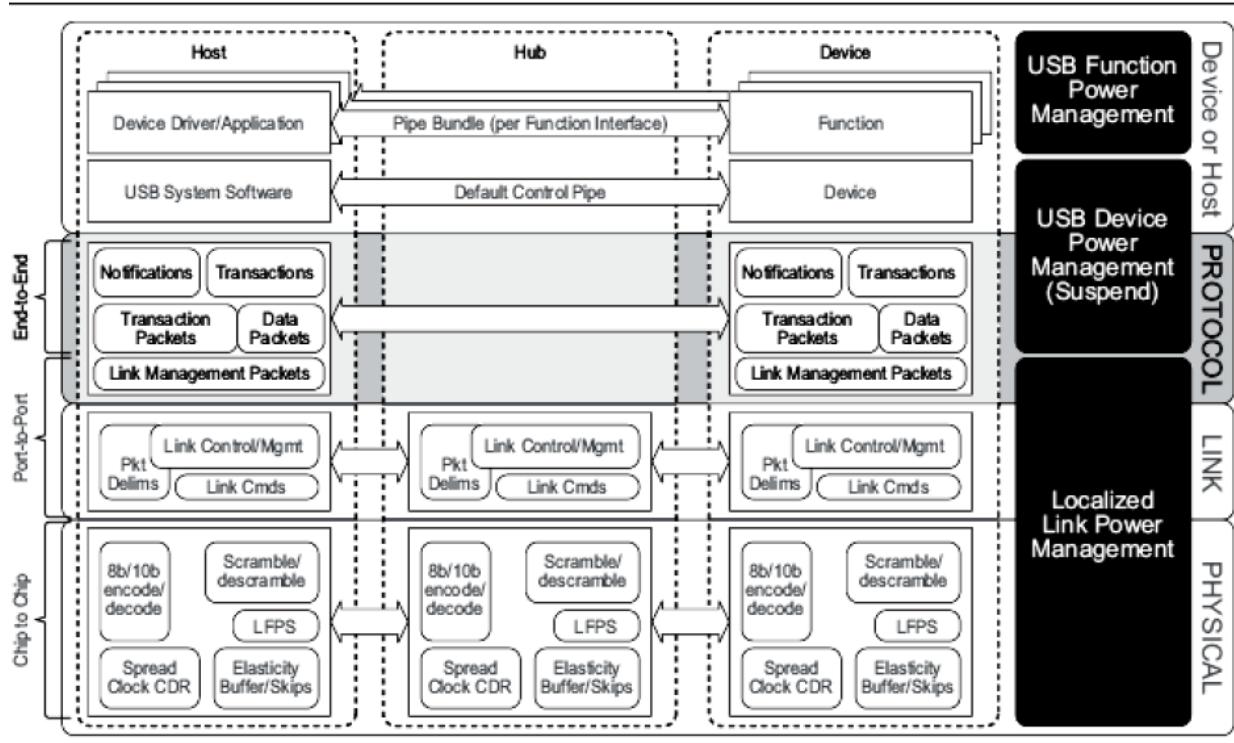


Figura 12: Capa de protocolo

U-135

Los paquetes de gestión de enlaces o Link Management Packets, (LMP por sus siglas en inglés) son enviados entre enlaces para comunicar asuntos de niveles de enlace tales como configuraciones y estados y es por eso que viajan predominantemente entre capas de entre el director y el dispositivo. Por ejemplo, la espera por inactividad U2, LMP es usada para definir el tiempo de espera desde el estado U1 hasta el estado U2. La estructura de un LMP es mostrado a continuación en la Figura 13.

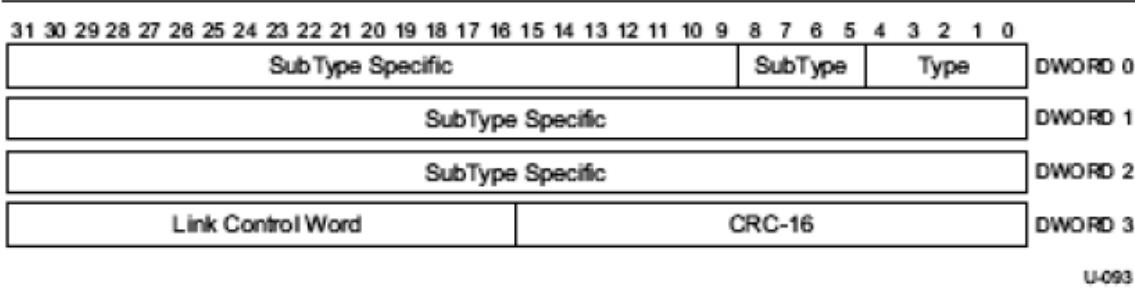


Figura 13: Estructura de un paquete de gestión de enlace

Los paquetes de transacción reproducen la funcionalidad provista por los paquetes testigo e intercambio y viajan entre el director y los puntos de llegada del dispositivo. Ellos no llevan ningún dato si no que forman el núcleo del protocolo. Por ejemplo, el paquete ACK es usado para reconocer el paquete recibido. La estructura de un paquete de transacción es mostrada en la Figura 14.

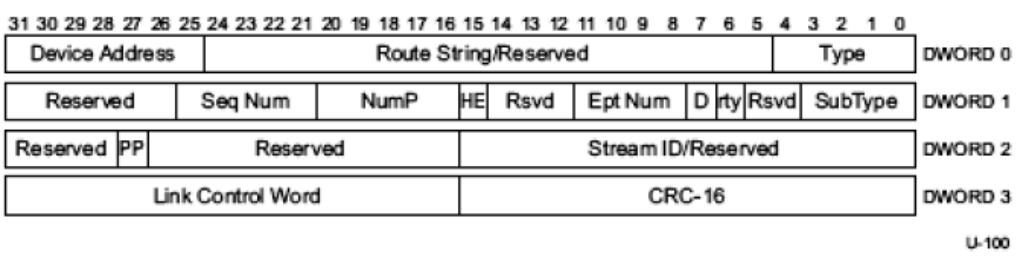


Figura 14: Paquete de transacción ACK

Los paquetes de datos en realidad llevan datos. Se componen de dos partes: Un encabezado de datos y los datos en sí. La estructura de un paquete de datos es mostrada en la derecha en la Figura 15.

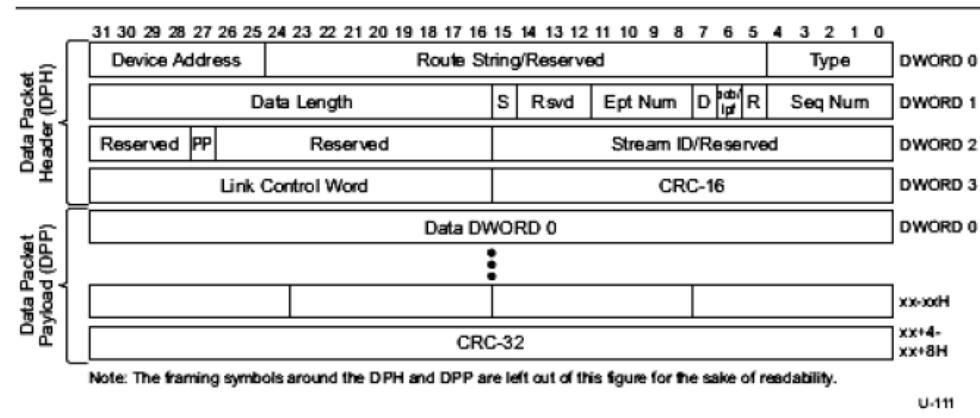


Figura 15: Ejemplo de un paquete de datos.

Los paquetes de marca de tiempo asincrónicos o Isochronous Time Stamp, (ITP por sus siglas en inglés) contienen ranuras de tiempo y se transmiten desde el director hacia todos los dispositivos activos. Los dispositivos usan las marcas de tiempo para sincronizarse con el director. No contienen ninguna información de enruteamiento. La estructura de un ITP es mostrada en la Figura 16.

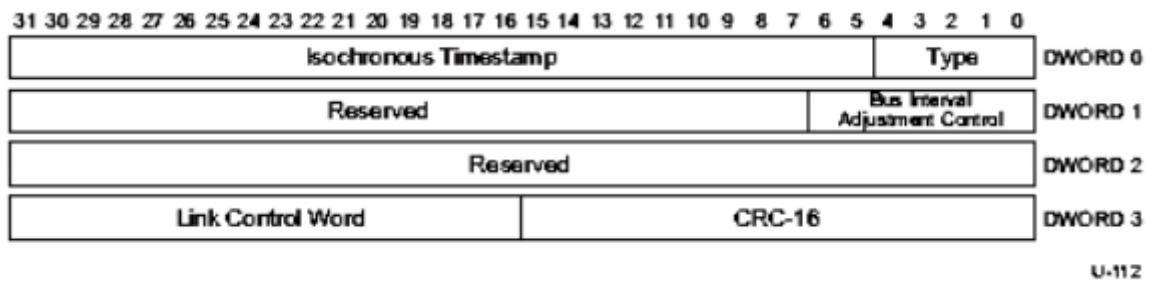


Figura 16: Estructura de un ITP

Las transferencias de salida son iniciadas por el director mediante el envío del paquete de datos sobre el bus en cascada. El paquete de datos contiene la dirección de enruteamiento del dispositivo y el número de punto de llegada. Si la transacción no es asincrónica, entonces, cuando se recibe el paquete de datos, el dispositivo lanza un paquete de reconocimiento, el cual también contiene el siguiente número de paquete en la secuencia. Este proceso continúa hasta que todos los paquetes han sido transmitidos a menos que un punto de llegada responda con un error durante la transacción. Las transferencias de entrada son iniciadas por el director mediante el envío de un paquete de reconocimiento que contiene la dirección del dispositivo y la del punto de llegada y el número de paquetes que el director espera. El dispositivo comienza entonces a enviar los paquetes hacia el director. La respuesta del director reconoce la transferencia anterior mientras que una nueva transferencia se inicia desde el dispositivo.

Una modificación importante en la especificación USB 3.0 es la difusión única en lugar de la difusión masiva. Los paquetes en el USB 2.0 se difundían a todos los dispositivos. Esto necesitaba que cada uno de los dispositivos conectados decodificaran el paquete de dirección para verificar si era para él. Los dispositivos tenían que reactivarse ante cualquier actividad USB independiente de su necesidad en la transferencia. Esto resultaba en un gran desperdicio de energía. Los paquetes USB 3.0 (excepto por el ITP)

son de difusión única hacia el dispositivo. La información necesaria de enrutamiento para los concentradores está incorporada en el paquete.

Otra modificación significativa introducida en el USB 3.0 se relaciona con las transferencias por interrupción. En el USB 2.0 las transferencias por interrupciones eran emitidas por el director en cada intervalo de servicio independientemente de si el dispositivo estaba o no listo para las transferencias. Sin embargo, los puntos de llegada para interrupción SuperSpeed pueden enviar un ERDY/NRDY a respuesta a una interrupción de transferencia o pedido del director. Si el dispositivo contesta un ERDY, el director continúa interrumpiendo al punto de llegada de interrupción del dispositivo en cada intervalo de servicio. Si el dispositivo contesta NRDY, el director detiene los pedidos o transferencias de interrupciones hacia el punto de llegada, hasta que el dispositivo de forma asincrónica (no iniciada por el director) informe un ERDY.

Una de las ventajas más grandes de la arquitectura de bus dual simple con el protocolo USB 3.0 es la capacidad de enviar paquetes múltiples a una sola dirección sin esperar por el paquete de reconocimiento desde el otro lado, lo cual de otra forma en un bus de medio compartido podría causar una contención del bus. Esta capacidad se explota para formar un nuevo protocolo que indica que los paquetes son enviados con un número de paquete, de manera que cualquier pérdida o reconocimiento desfavorable que llegue después de gran retraso pueda ser usado para la retransmisión del paquete perdido, identificado por el número de paquete. El número de paquetes en ráfagas que pueden ser enviados (sin esperar por el reconocimiento) se comunica antes de la transferencia.

Otra característica notable del USB 3.0 es el protocolo de flujo disponible para transferencias masivas. Las transferencias masivas normales (de salida) transfieren un flujo simple de datos a un punto de llegada del dispositivo. Normalmente, cada flujo de datos es obtenido desde un buffer (FIFO) en el transmisor hacia otro buffer (FIFO) en el receptor. El protocolo de flujo permite al transmisor asociar un ID de flujo (de 1 a 65536) con el flujo de pedido o transferencia actual. El receptor del flujo o pedido obtiene o sincroniza la información hacia/desde el buffer FIFO apropiado. Esta multiplexación de flujos logra imitar una tubería que puede cambiar dinámicamente su extremo. El flujo hace posible realizar un modelo de ejecución fuera de orden que es requerido por la cola de comandos. El concepto de flujos permite protocolos de almacenamiento masivos más poderosos. Un enlace de comunicación típico consiste en una tubería comando de salida, una tubería de entrada y salida (con flujos de datos múltiple) y una tubería de estados. El director puede poner en cola comandos, es decir, enviar un nuevo comando sin esperar que se complete uno anterior, etiquetando cada comando con el ID de flujo.

Debido a la forma en que se define la gestión de la energía del USB 3.0, los enlaces no activos (dispositivos concentradores) puede tomar más tiempo ser activados al detectar información en el bus. Las transferencias asincrónicas que activan el enlace toman más tiempo en llegar al destino y puede infringir el requerimiento de intervalo de servicio. El protocolo PING-asincrónico elude este problema. El director envía una transferencia PING antes de una transacción asincrónica. Una respuesta PING indica que todos los enlaces en la ruta están activos (o han sido activados). El director puede enviar o pedir un paquete de datos asincrónico. Los dispositivos asincrónicos USB 2.0 no pueden entrar en un estado de bus de bajo consumo entre intervalos de servicio.

2.3.3 – Capa de enlace

La capa de enlace mantiene la conectividad del enlace y asegura la integridad de la información entre los socios del enlace mediante la implementación de la detección de errores. La capa de enlace asegura la confiabilidad de la entrega de datos mediante enmarcado cabeceras de paquetes al final de la transmisión y detectando errores de nivel de enlace en el extremo de llegada. La capa de enlace también implementa protocolos para el control de flujo y participa en la administración de energía. La capa de enlace provee una interface para la capa de protocolo para el paso a través de mensajes entre capas de protocolo. Los socios del enlace se comunican usando comandos de enlace.

2.3.4 – Capa física

Los dos pares de líneas diferenciales, uno para las transferencias de salida y otro para las transferencias de entrada, definen la conexión física entre un director USB 3.0 SuperSpeed y el dispositivo. La capa física acepta un byte a la vez, codifica los bits (un proceso que es conocido para reducir las emisiones EMI), convierte estos en 10 bits, hace los bits serie, y transmite los datos diferencialmente sobre el par de cables. El circuito de recuperación de datos de reloj ayuda a recuperar los datos en el extremo receptor. El bloque LFPS (señalización periódica de baja frecuencia) se usa para la inicialización y administración de la energía cuando el bus está inactivo.

La detección de dispositivos SuperSpeed se realiza mirando las terminaciones de la línea de forma similar a los dispositivos USB 2.0

2.3.5 Administración de la energía

El USB 3.0 proporciona capacidades mejoradas de administración de energía para hacer frente a las necesidades de aplicaciones portátiles que funcionan con baterías. Dos modos de "IDLE" (Inactivo) (indicada como U1 y U2) se definen además del modo "Suspender" (indicada como U3) de la norma USB 2.0.

El estado U2 ofrece ahorros de energía más altos que U1 al permitir más circuitería analógica (como circuitos de generación de reloj) para ser desactivada temporalmente. Esto se traduce en un tiempo de transición más largo de U2 al estado activo. El estado de suspensión (U3) consume menos poder y nuevamente requiere más tiempo para despertar el sistema.

Los modos de espera pueden ser introducidos debido a la inactividad en un puerto en cascada durante un período de tiempo programable o pueden ser iniciadas por el dispositivo, en base a la programación de la información recibida desde el director. Esta información se indica con el director al dispositivo mediante el paquete banderas pendientes, final de ráfagas, y último paquete. Basado en de estos indicadores, el dispositivo puede decidir entrar en un modo de espera sin tener que esperar la inactividad en el bus. Cuando un enlace está en uno de estos estados de espera, la comunicación puede tener lugar a través de la señalización periódica de baja frecuencia (LFPS), que consume energía significativamente menor que la señalización SuperSpeed. De hecho, el modo de espera se puede salir con una transmisión LFPS, ya sea del director o dispositivo.

El estándar USB 3.0 también introduce la característica de “suspender función”, la cual permite la administración de la energía de funciones individuales de un dispositivo compuesto. Esto provee la flexibilidad de suspender ciertas funciones de un dispositivo compuesto, mientras que otras funciones se mantienen activas.

Se logra ahorro de energía adicional mediante el mecanismo de mensajería de tolerancia de latencia implementado por el USB 3.0. Un dispositivo puede informarle al director el máximo retraso que puede tolerar desde que se reporta un estado ERDY hasta que se recibe una respuesta. El director puede tener en esta tolerancia latencia para administrar la energía del sistema.

Por lo tanto, la eficiencia de energía está incorporada en todos los niveles de un sistema USB 3.0, incluyendo la capa física, la capa de protocolo y la PHY. Un sistema USB 3.0 requiere más potencia mientras está activo. Pero debido a la alta tasa de datos y varias características de eficiencia de energía, se mantiene activo por períodos cortos. Una transferencia de datos SuperSpeed puede gastar hasta un 50 por ciento menos que una transferencia de alta velocidad. Esto es crucial para la vida de batería en dispositivos portátiles de mano como los teléfonos celulares.

Capítulo 3: Desarrollo de la interface USB - Visual C++ - MatLab

3.1 Librería CyAPI.lib de Cypress

El kit de desarrollo CYUSB3KIT-001 ofrece una serie de herramientas para el desarrollo de aplicaciones con comunicación USB. Una ellas, es la librería CyAPI.lib, la cual brinda una simple pero poderosa interface de programación en C++ para dispositivos USB. Más específicamente, es una librería de clase C++ que provee una programación de alto nivel para comunicarse con el driver CyUSB3.sys del dispositivo Cypress.

A diferencia de otras aplicaciones que se comunican mediante llamadas de la API de Windows tales como SteupDiX y DeviceControl, esta librería lo hace mediante simples llamadas a métodos tales como *Open*, *Close* y *XferData*.

Para usar la CyAPI.lib se debe incluir el archivo de cabecera CyAPI.h en el código que se va a acceder a la clase CCyUSBDevice. Además, el archivo estático CyAPI.lib debe ser referenciado dentro del proyecto.

La librería utiliza un modelo de *Dispositivo* y *Punto de llegada*. Para utilizarla, se debe crear una instancia de la clase CCyUSBDevice usando la palabra clave *new*. Un objeto CCyUSBDevice conoce cuantos dispositivos USB están adjuntos al driver CyUSB.sys y puede hacer un abstracto a cualquiera de ellos por vez utilizando el método *Open*. Una instancia del objeto brinda varios métodos y miembros que son específicos del dispositivo tales como *DeviceName*, *DevClass*, *VendorID*, *ProductID* y *SetAltIntfc*.

Cuando un objeto CCyUSBDevice es abierto, su miembro punto de llegada provee una interface para llevar acabo transferencias de datos desde y hacia los puntos de llegada del dispositivo USB. Miembros de datos de puntos de llegada específicos y métodos tales como *MaxPktSize*, *TimeOut*, *bIn*, *Reset* y *XferData*, son accesibles a través de los miembros de puntos de llegada del objeto CCyUSBDevice.

3.2 Creación del programa VS2010

La aplicación de Visual Studio 2010 se desarrolló en Visual C++ en dos formas diferentes. Primero siguiendo las instrucciones del archivo “QuickStartGuide” en formato PDF que sirve como punto de inicio para el desarrollo de aplicaciones que utilizan la librería de Cypress (Cabe aclarar que hay que tener instalado el Visual Studio 2010 el cual tiene una versión de estudiante gratuita y además, todos los programas que vienen incluidos con el Kit). De esta forma se aprenden los conocimientos básicos que permiten utilizar la misma. A continuación se describen los pasos necesarios desde el inicio para crear una aplicación y utilizar la CyAPI.lib.

3.2.1 Creación de un proyecto en visual C++ y configuración de la CyAPI.lib

- 1. Se crea un nuevo proyecto en Visual Studio 2010, seleccionando en la venta principal del mismo: **Visual C++ > Windows Forms Application** y se escribe un nombre que en el ejemplo recomienda *Ejemplo1*. Finalmente se hace clic en *OK*. En la Figura 17 se muestra los pasos descriptos anteriormente.

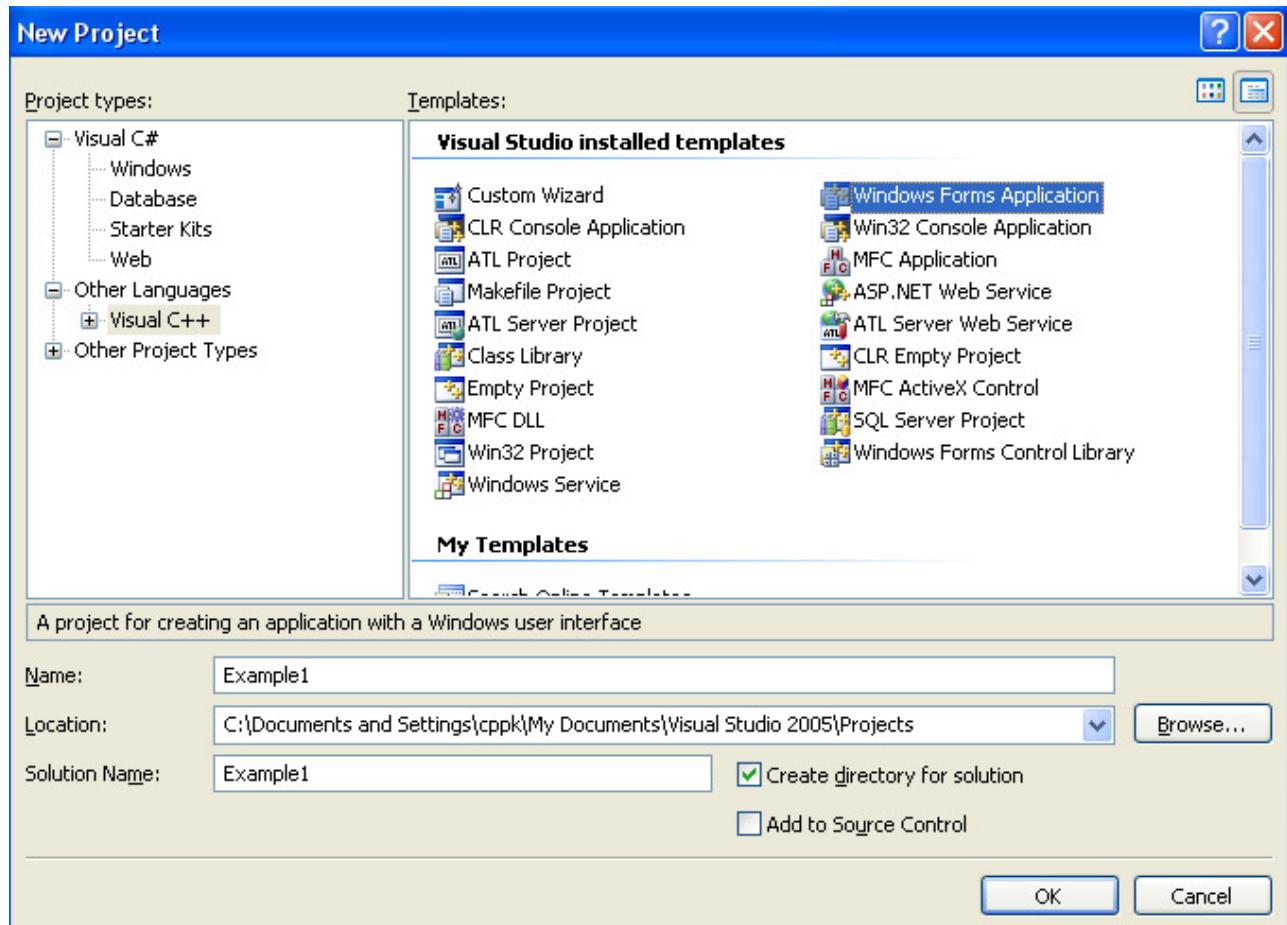


Figura 17: Creando un nuevo proyecto en VS2010

- 2. Después de hacer clic en *OK* se muestra un formulario vacío. Hacer clic en la flecha verde (Ejecutar) y se ejecutara la aplicación en blanco. Ahora podemos ver que la aplicación funciona.
- 3. Seleccionamos con el botón derecho dentro del árbol del proyecto sobre “Sources files”, se desplegará un menú, “*Add > Existing Item*”. Se abrirá una nueva ventana de Windows en donde se buscara dentro del directorio donde se instaló la aplicación USB Suite, de las herramientas brindadas por Cypress, y elegimos el archivo CyAPI.lib. De esta forma se referencia la librería al proyecto. Este paso es mostrado en la Figura 18.

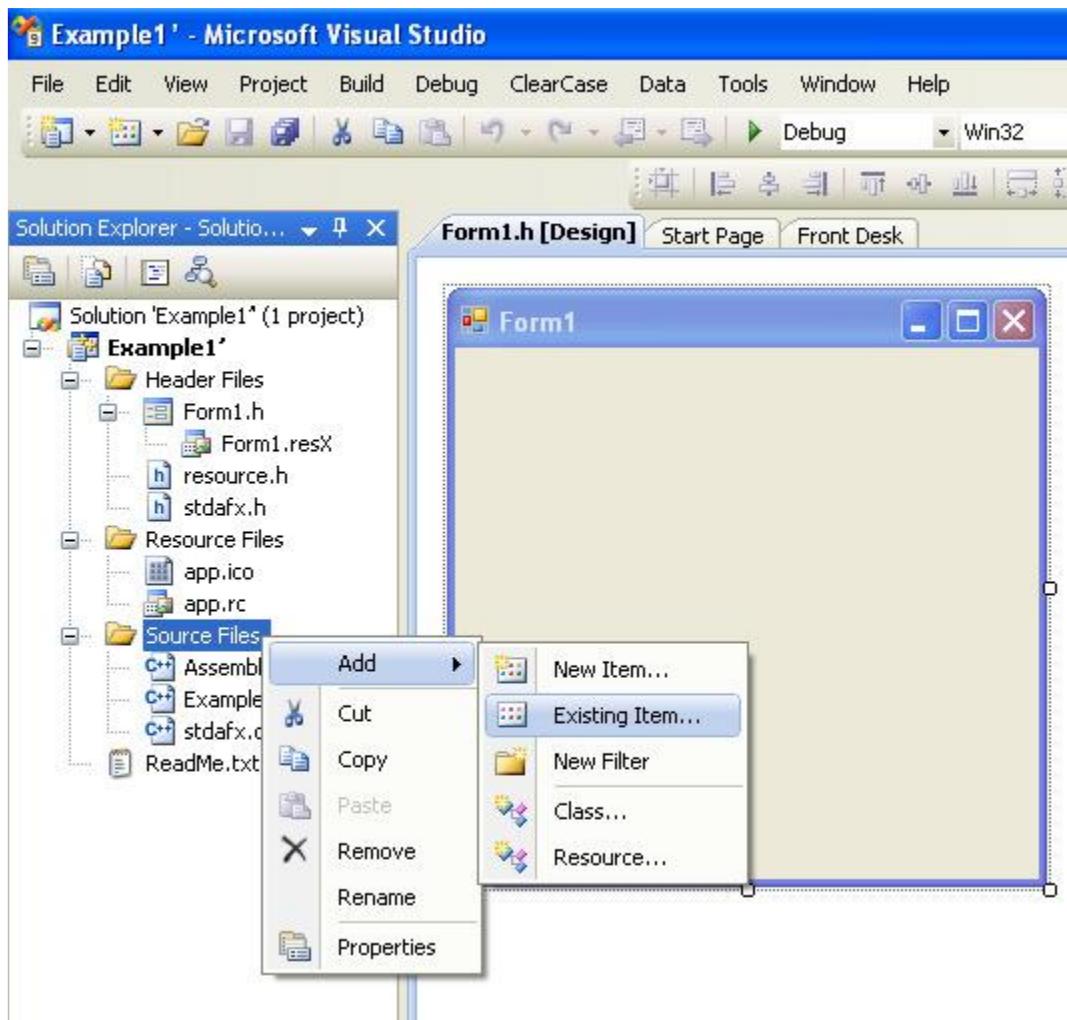


Figura 18: Añadir la CyAPI.lib al proyecto

- 4. Sobre el formulario vacío se hace clic con el botón derecho y se selecciona “View Code”, nos mostrara el código de ejecución del formulario. El programa ya incluye algunas líneas de código para generar el formulario. Al comienzo del código arriba de todo se muestran las directivas “using Namespace”. Se agrega el código las siguientes líneas después de la línea #pragma:

```
#include <wtypes.h>
```

```
#include <dbt.h>
```

Estos dos encabezados son necesarios para los tipos de datos primitivos en CyAPI.h y los eventos PnP respectivos.

- 5. Después de referenciar la librería CyAPI.lib como se describió anteriormente, se debe exponer la interface hacia ella. Esto lo hacemos incluyendo la dirección a la referencia CyAPI.h, lo cual permite tener acceso a la clase. Para esto se hace clic en **Project > Properties**. En el cuadro de dialogo seleccionamos **Configuration Properties > C/C++ > General > AdditionalInclude Directories**. Allí apuntamos el directorio donde se encuentra el archivo CyAPI.h del directorio de instalación del Cypress USB Suite (CyAPI/inc) y hacemos clic en **OK**.

- 6. Nuevamente se hace clic en **Project > Properties**. En el cuadro de dialogo seleccionamos **Configuration Properties > Linker > Input > Additional Dependencies** y escribimos **user32.lib** como es mostrado en la Figura 19.

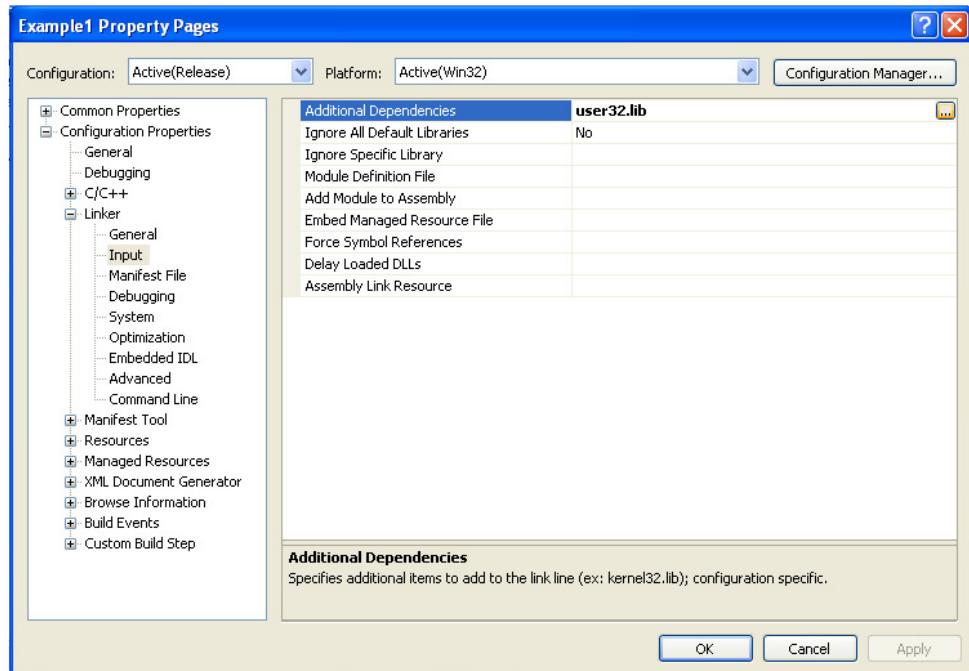


Figura 19: Configuración adicional del proyecto VS2010

- 7. Agregamos el encabezado `#include <CyAPI.h>` junto con los demás al comienzo del proyecto. Luego vamos nuevamente a **Project > Properties**. En el cuadro de dialogo seleccionamos **Configuration Properties > General > Common Language Runtime Support** y seleccionamos del menú desplegable **Common Language Runtime Support (/clr)** como se muestra en la Figura 20.

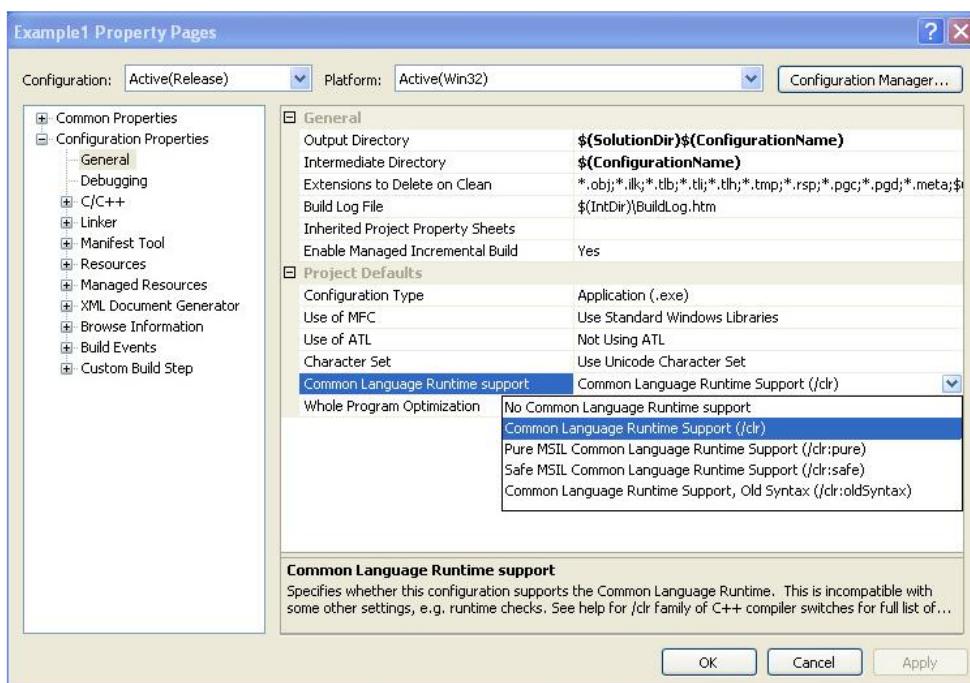


Figura 20: Configuración de propiedades del proyecto.

- 8. Luego insertamos el siguiente código dentro de la ubicación exacta tal cual se describe en el mismo, en el código de la clase Form1. Nótese que *Form1()* y *WndProc* deben ser miembros públicos:

```
1. public ref class Form1 : public
   System::Windows::Forms::Form
2. {
3. public:
4. CCyUSBDevice *USBDevice, *CyStreamdev;
5. int AltInterface;
6. bool bPnP_Arrival;
7. bool bPnP_Removal;
8. bool bPnP_DevNodeChange;
9. Form1(void)
10. {
11. InitializeComponent();
12. USBDevice =new CCyUSBDevice((HANDLE)this-
>Handle,CYUSBDRV_GUID,true);
13. }
14. virtual void WndProc( Message% m ) override
15. {
16. if (m.Msg == WM_DEVICECHANGE)
17. {
18. // Tracks DBT_DEVNODES_CHANGED followed by
DBT_DEVICEREMOVECOMPLETE
19. if (m.WParam == (IntPtr)DBT_DEVNODES_CHANGED)
20. {
21. bPnP_DevNodeChange = true;
22. bPnP_Removal = false;
23. }
24. // Tracks DBT_DEVICEARRIVAL followed by
DBT_DEVNODES_CHANGED
25. if (m.WParam == (IntPtr)DBT_DEVICEARRIVAL)
26. {
27. bPnP_Arrival = true;
28. bPnP_DevNodeChange = false;
29. }
30. if (m.WParam == (IntPtr)DBT_DEVICEREMOVECOMPLETE)
31. bPnP_Removal = true;
32. // If DBT_DEVICEARRIVAL followed by
DBT_DEVNODES_CHANGED
33. if (bPnP_DevNodeChange && bPnP_Removal)
34. {
35. bPnP_Removal = false;
36. bPnP_DevNodeChange = false;
37. GetDevice();
38. }
```

```
39. // If DBT_DEVICEARRIVAL followed by
DBT_DEVNODES_CHANGED
40. if (bPnP_DevNodeChange && bPnP_Arrival)
41. {
42. bPnP_Arrival = false;
43. bPnP_DevNodeChange = false;
44. GetDevice();
45. }
46. }
47. Form::WndProc( m );
48. }
49. void GetDevice()
50. {
51. USBDevice = new CCyUSBDevice((HANDLE)this-
>Handle,CYUSBDRV_GUID,true);
52. AltInterface = 0;
53. if (USBDevice->DeviceCount())
54. {
55. Text = "Device Attached";
56. }
57. else
58. {
59. Text = "No Devices Attached";
60. }
61. }
```

3.2.2 Análisis del código del proyecto ejemplo1

Antes de analizar el código descripto anteriormente se debe tener en cuenta que para encontrar información detallada sobre el uso de la librería CyAPI.h se puede recurrir al archivo que brinda Cypress en la instalación del USB Suite llamado “CyAPI.pdf”.

La clase CCyUSBDevice es el punto de entrada primario a la librería. Toda la funcionalidad de la misma es accedida a través de la instancia de CCyUSBDevice. El objeto creado sirve como interfaz de programación al driver en el cual GUID (Identificador del dispositivo) es pasado como parámetro. El constructor de la clase se muestra en el código en la línea 12.

(HANDLE)this->Handle es un administrador para la aplicación de la ventana principal en el cual la función WndProc procesa los eventos PnP.

CYUSBDRV_GUID es un valor constante único de guid para el driver CyUSB3.sys y está especificado en el archivo inf que es usado para enlazar el dispositivo al driver.

El método WndProc del código main del form1 es usado para supervisar los mensajes PnP. Windows envía todas las ventanas de alto nivel una serie de mensajes por

defecto cuando nuevos dispositivos son agregados y están disponibles, o cuando los dispositivos existentes son desconectados. Estos mensajes son conocidos como “mensajes WM_DEVICECHANGE”. Cada uno de estos tiene asociado un evento el cual describe el cambio. Cuando un dispositivo es conectado o desconectado del sistema, el mismo envía una transmisión general de cambio de dispositivo DBT_DEVNODES_CHANGED usando el mensaje VM_DEVICECHANGE. El sistema operativo envía el mensaje DBT_DEVICEARRIBAL cuando un dispositivo se conecta y un DBT_DEVICEREMOVAL cuando un dispositivo se desconecta. El WndProc captura el mensaje como un argumento y si el mensaje indica una conexión o desconexión de un dispositivo, llama al método GetDevice() para actualizar el estado de los dispositivos USB conectados que estén contemplados dentro del driver de Cypress.

El GetDevice() usa un método DeviceCount() (línea 53 del código), el cual es un miembro de la clase CCyUSBDevice. El mismo, devuelve el número de dispositivos conectados del driver CyUSB3.sys. La declaración IF de la línea 53 a 60 del código realiza una evaluación de la presencia o ausencia de dispositivos USB y cambia el título del formulario a “Device Attached” o “No Device Attached” según existan o no dispositivos conectados respectivamente. Si ejecutamos el código a través de la flecha verde (Ejecutar) y luego conectamos y desconectamos la placa del Kit a través del cable USB 3.0 también incluido en el kit veremos como el mensaje de la aplicación mostrada al comienzo de la ventana cambia según se describió.

Esta es una explicación básica de la aplicación. Como se mencionó anteriormente en el manual “CyAPI.pdf” existe una explicación detallada del uso de la librería.

3.2.3 Uso de la librería a través de código ejemplo Streamer

Una vez comprendido el uso básico de la librería se continuó trabajando para desarrollar una aplicación completa que permita la recepción de los datos que se envían desde el microcontrolador CYUSBFX3.

Se tomó como punto de partida el código ejemplo “Streamer” que brinda Cypress. El mismo utiliza la librería CyAPI.lib y permite leer de forma masiva datos ingresados por el puerto USB, mostrar la velocidad de transferencia, mostrar la cantidad de bytes exitosos y fallados, elegir la cantidad de paquetes a transferir por cada transferencia y mostrar una vista rápida de los datos que se reciben.

Este código ejemplo tiene la librería CyAPI.lib incluida en el proyecto y utiliza algunas de las funciones básicas para desarrollar una comunicación masiva mediante USB. En la Figura 21 se muestra aplicación Streamer en ejecución. En la misma se pueden ver varias opciones que se describen a continuación:

EndPoint permite elegir el punto de llegada de la interface que queramos usar en caso de que la aplicación que desarrollemos tenga varias interfaces. *Packets per Xfer* permite elegir la cantidad de paquetes que vamos a recibir en cada transferencia (Se aclara que para el protocolo USB 3.0 cada paquete consta de 1024 bytes). *Xfer to Queue* esta

opción permite ir cargando una cola de transferencia de la cantidad de paquetes que seleccionemos. *Successes* muestra la cantidad de bytes exitosos que se han recibido. *Failures* indica la cantidad de bytes que fallaron en la transferencia. *Timeout Per Xfer (ms)* elegimos el tiempo de espera máximo que vamos a esperar entre cada transferencia antes declarar perdida una transferencia. Botón *Start* permite comenzar a recibir los datos que ingresan por el puerto USB. *Transfer Rate (KB/s)* Nos muestra un promedio de la tasa de transferencia aproximada en la comunicación. *CPU Utilization (%)* muestra el porcentaje de uso de la CPU por parte de la aplicación. *Show Transferred Data* muestra los datos recibidos constantemente en la caja de texto si la casilla de verificación esta activada.

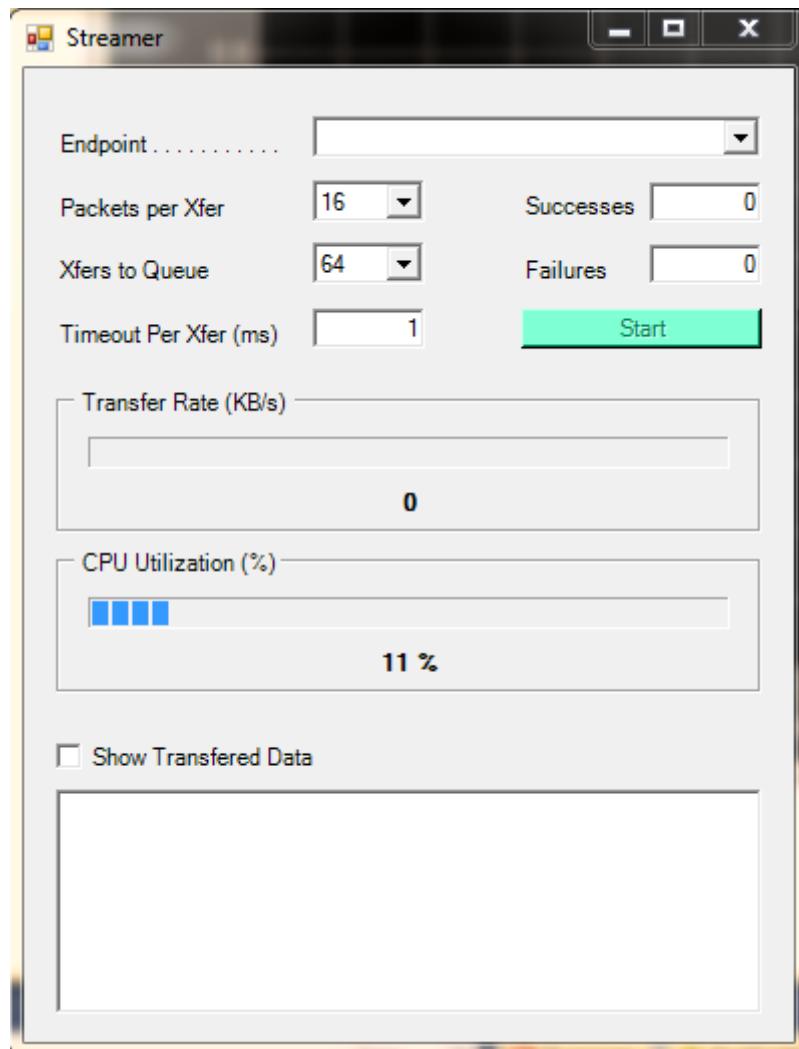


Figura 21: Aplicación Streamer ejemplo

Cuando presionamos el botón Start, el mismo cambia de nombre a Stop y la aplicación crea una serie de punteros que permiten armar la estructura para la recepción de cada paquete y la cola de transferencia. Ese fragmento del código de creación se ve a continuación en el código extraído de la aplicación:

```
// Allocate the arrays needed for queueing
P UCHAR *buffers = new P UCHAR[QueueSize];
CCyIsoPktInfo **isoPktInfos = new CCyIsoPktInfo*[QueueSize];
P UCHAR *contexts = new P UCHAR[QueueSize];
```

```
OVERLAPPED in0vLap[MAX_QUEUE_SZ];
```

Luego la aplicación comienza luego un bucle infinito mientras no se vuelve a presionar el botón *Stop* e inicia la transferencia mediante el uso del método *BeginDataXfer(buff[i],len)* con los parámetros *buff[i]* que es el puntero donde se reciben los datos recibidos y la propiedad *len* el cual indica la longitud de la transferencia. A continuación llama al método *WaitForXfer(timeout)* el cual espera a que la transferencia termine o se llegue al tiempo *timeout* que se pasa como parámetro. Finalmente llama al método *FinishDataXfer(buff[i])* mediante el cual se termina la transferencia y se reciben los datos en el parámetro *buff*. La aplicación toma el tiempo de la PC al momento de iniciar la transferencia y al final de la misma, luego, mediante la cantidad de datos recibidos calcula la tasa de transferencia. El código que realiza dicha operación es simplemente programación de visual C++ y no compete al desarrollo de esta aplicación, por lo tanto no se lo explica en detalle aquí.

Hasta aquí se describe la aplicación ejemplo Streamer que solamente recibe los datos desde el USB. En el siguiente sub capítulo se describirá la modificación del mismo para poder tomar los datos recibidos en cada transferencia y graficarlos a través de MatLab.

3.3 Creación de interface MatLab – VS 2010

La programación en Visual C++ a través de VS2010 permite una forma simple y fluida de comunicación USB mediante la librería CyAPI.lib explicada anteriormente y además, otorga varias herramientas muy potentes para crear una aplicación visual, no obstante la gráfica en este lenguaje y entorno de desarrollo se torna bastante compleja. Es por eso que en este proyecto se decidió utilizar la potencia de gráfica y procesamiento de datos que tiene el programa MatLab. Por lo tanto, de la unión del programa Visual Studio 2010 y MatLab se puede obtener una aplicación de comunicación USB y gráfica muy potente.

3.3.1 Motor Engine de MatLab

MatLab Engine es una herramienta incluida en el programa MatLab que permite, desde aplicaciones desarrolladas en C++, enviar datos a MatLab, procesarlos y graficarlos. Esto, se realiza convirtiendo dentro del código C++ las variables a mxarray que es el tipo de dato que MatLab puede procesar. A continuación se detallan los pasos para incluir el motor Engine en un proyecto de Visual Studio 2010.

Primero vamos al nombre del proyecto y con el botón derecho seleccionamos *Properties*. Se abre una nueva ventana y en ella realizamos los siguientes cambios:

- a) En *C/C++ General*, agregamos el directorio \$MATLABROOT\extern\include en el campo *ADDITIONAL INCLUDE DIRECTORIES*.
- b) Después, dentro de *C/C++ Precompiled Headers*, seleccionamos “*Not Using Precompiled Headers*.”
- c) Dentro de *Linker General* agregamos el directorio: MATLABROOT\extern\lib\win32\microsoft en el campo *ADDITIONAL LIBRARY DIRECTORIES*.

- d) En *Linker Input* agregamos los siguientes nombres: libmx.lib, libmat.lib, libeng.lib en el campo **ADDITIONAL DEPENDENCIES**.
- e) Luego en Windows abrimos una ventana de comandos y ejecutamos los comandos `cd matlabroot\bin\win32. Matlab/regserver`. Finalmente cerramos la ventana de MATLAB que aparece.
- f) Luego debemos incluir en la cabecera del código de VS2010 las siguientes líneas:
`#include <engine.h>`
`#pragma comment (lib, "libmat.lib")`
`#pragma comment (lib, "libmx.lib")`
`#pragma comment (lib, "libmex.lib")`
`#pragma comment (lib, "libeng.lib")`

En el caso del código que se editó del ejemplo Streamer, para general la aplicación de este proyecto, fue necesario realizar una modificación al momento de incluir la línea “`#include <engine.h>`” ya que la misma para algunos tipos de datos (char, int, double) tienen el mismo nombre que en C++ por lo tanto había un problema de espacio de nombres. Para resolver este problema se utilizó la siguiente modificación:

```
Namespace eng {  
#include <engine.h>  
}
```

De esta forma, para utilizar las funciones y declaraciones de engine.h simplemente se debe anteponer “`eng::`” antes de utilizar las mismas.

Una vez finalizados estos pasos ya estamos en condiciones de usar en nuestro proyecto el motor Engine de MatLab.

Para transferir información y utilizarla en MatLab administrándola desde VS2010 se requieren 3 pasos. 1- Abrir MatLab a través del motor engine. 2- Convertir los datos a un formato mxArray. Y 3- Transferir los datos usando la función “`engPutVariable`”.

Para abrir MatLab a través del motor engine se crea un puntero tipo Engine y se llama a la función `engOpen()` de la siguiente forma:

```
Eng::Engine *Ep; // Donde Ep es el puntero tipo Engine por el cual se comunicara  
// toda la información enviada desde Visual y hacia MatLab.  
Ep = eng::engOpen(NULL);
```

Para convertir los datos a un formato mxArray, primero, debemos reservar memoria para un puntero del tipo mxArray y transferir los datos a la memoria reservada. mxArray es la estructura principal de MatLab utilizada para contener información en archivos MEX. Puede contener tipos de datos reales, complejos, matrices, cadenas y otras estructuras de MatLab. El siguiente código describe los pasos descriptos anteriormente:

```
eng::mxArray *Temp; // puntero de tipo mxArray.  
Temp = eng::mxCreateDoubleMatrix(1,3,eng::mxReal)//
```

Aquí en la función “mxCreateDoubleMatrix” el primer parámetro, es la cantidad de filas, y el segundo, la cantidad de columnas, el tercer elemento define el tipo de dato que en este caso es real. Luego para copiar la información de una variable en C++ a la variable creada, utilizamos las funciones “mxGetPtr” y “memcpy” de la siguiente manera:

```
double data[3] = {1.0, 2.0, 3.0};  
memcpy((void*)eng::mxGetPtr(Temp), (void*)data, sizeof(data));
```

De esta forma copiamos la variable data en la memoria de Temp que es el tipo de dato que MatLab “entiende”. Finalmente colocamos la variable en el espacio de trabajo de MatLab de la siguiente forma:

```
eng::engPutVariable(Ep, "NombreDeVariable", Temp);
```

Aquí a través del puntero Ep, colocamos los datos de Temp, en una matriz en MatLab con el nombre que se defina en “NombreDeVariable”.

Además el motor Engine cuenta con otra función que se llama *engEvalString(Ep, "CadenaDeCaracteres")* mediante la cual lo que se escriba dentro de “CadenaDeCaracteres” es igual que si desde la ventana de comandos en MatLab “tipreamos” lo que allí escribiríramos. Por ejemplo, si colocáramos dos matrices de la forma que vimos anteriormente como dos vectores llamados “X” e “Y” en el espacio de trabajo de MatLab podríamos graficar esas dos matrices de la siguiente forma:

```
eng::engEvalString(Ep, "plot(X, Y)");
```

De esta forma obtendríamos una gráfica de X e Y.

3.3.2 Prueba del motor Engine con aplicación ejemplo 1

Con el programa “ejemplo1” que se desarrolló en el capítulo 3.2.1 que ya tiene la librería CyAPI.lib incorporada se probó hacer funcionar el motor engine y graficar llamando funciones de MatLab desde Visual Studio 2010.

Se creó una función llamada “GraficarMatLab” en la cual se implementó un código de prueba que grafica continuamente una onda senoidal que varía en amplitud hasta completar los 1000 datos graficados. Genera un bucle donde carga los datos y los actualiza para ir variando los coeficientes “a” y “b” para modificar la amplitud.

En el formulario en blanco que teníamos originalmente del ejemplo 1, se agregaron dos botones, “Graficar” y “Cerrar grafica”. A estos botones se les asignaron las funciones de llamar a la función “GraficarMatLab” y “cerrarMatLabEngine” respectivamente. Esta última, se encarga de cerrar el puntero Engine abierto, el cual se declaró global para poder utilizarlo en ambas funciones de forma rápida en vez de pasarlo por referencia a cada una de ellas. A continuación se muestra el código de la función “GraficarMatLab”:

```

void GraficarMatLab()
{
    double x[1000];
    double y[1000];
    double z[1000];
    double t = 0;
    const double dt = 0.001;
    int i,j;
    double a,b;

eng::mxArray *z_array = eng::mxCreateDoubleMatrix(1000,1,eng::mxREAL);
eng::mxArray *a_array = eng::mxCreateDoubleMatrix( 1,1,eng::mxREAL);
eng::mxArray *b_array = eng::mxCreateDoubleMatrix( 1,1,eng::mxREAL);

double *pz = eng::mxGetPr(z_array);
double *pa = eng::mxGetPr(a_array);
double *pb = eng::mxGetPr(b_array);

for (i=0;i<1000;i++)
{
    x[i] = cos(2*(3,14)*t);
    y[i] = sin(2*(3,14)*t);
    t+=dt;
}

a = 1;
b = 0;
for (j=0;j<100;j++)
{
    // 
    for(i=0;i<1000;i++)
    {
        z[i] = a*x[i] + b*y[i];
        pz[i] = z[i];
    }
    pa[0] = a;
    pb[0] = b;
    eng::engPutVariable(ep, "z",z_array);
    eng::engPutVariable(ep, "a",a_array);
    eng::engPutVariable(ep, "b",b_array);
    eng::engEvalString(ep,"testPlot");
    a = a - 0.01;
    b = b + 0.01;
}

} //Fin de void GraficarMatLab();

```

En el código de la función “*GraficarMatLab*” se puede observar que después de colocar las variables “*z_array*”, “*a_array*” y “*b_array*” en el espacio de trabajo, se utiliza la función “*engEvalString(ep, "testPlot")*”, la misma, llama a la función *testPlot.m* que está en la carpeta “Work” de MatLab, y cuyo código, simplemente grafica la función y coloca en la gráfica los valores de los coeficientes *a* y *b*. El mismo se puede ver a continuación:

```

plot(z);
axis([0 1000 -1 1]);
grid on;

```

```
title(sprintf('a = %0.3f \t b = %0.3f',a,b));  
pause(0.1);
```

Al ejecutar el código *ejemplo 1* junto con el agregado del motor *Engine*, se obtienen la aplicación mostrada en la Figura 22. Luego al presionar el botón “Graficar” comienza a generar la gráfica. Las Figura 23 y Figura 24 muestran la misma, capturada en diferentes instantes de tiempo:

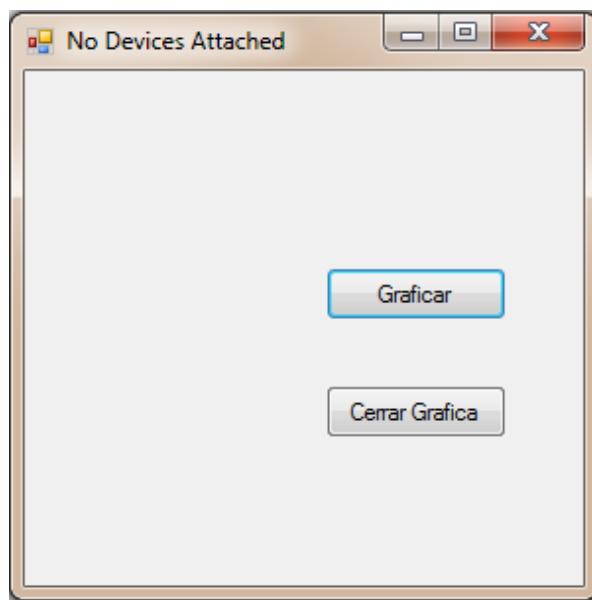


Figura 22: Aplicación ejemplo 1 modificada con engine

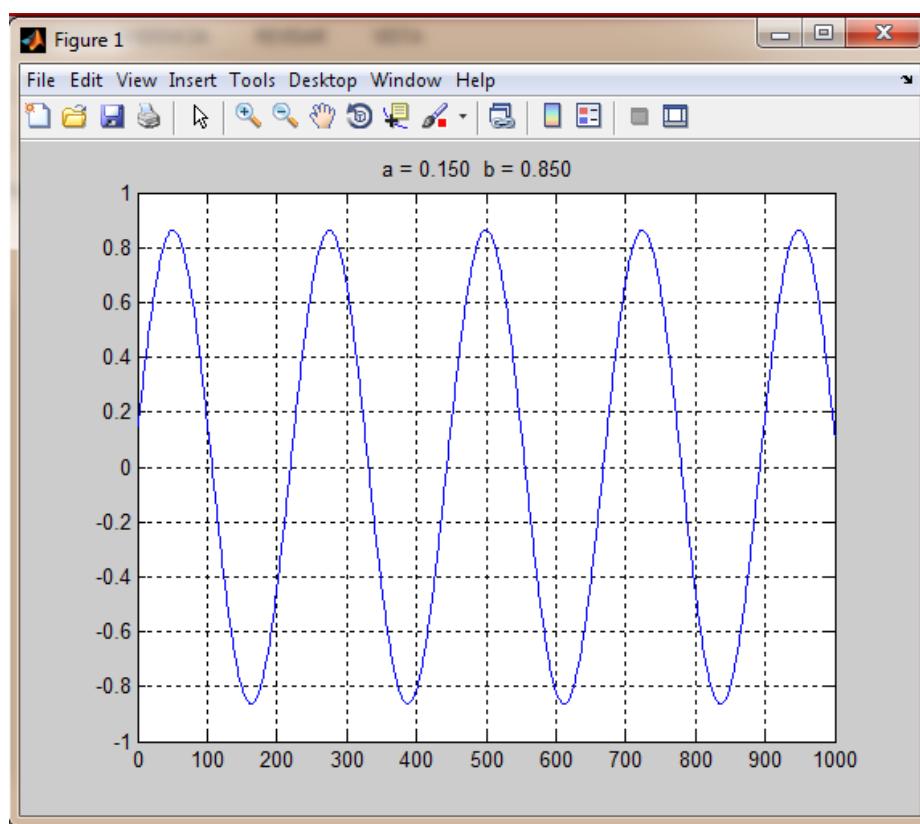


Figura 23: Grafica 1 de programa ejemplo 1

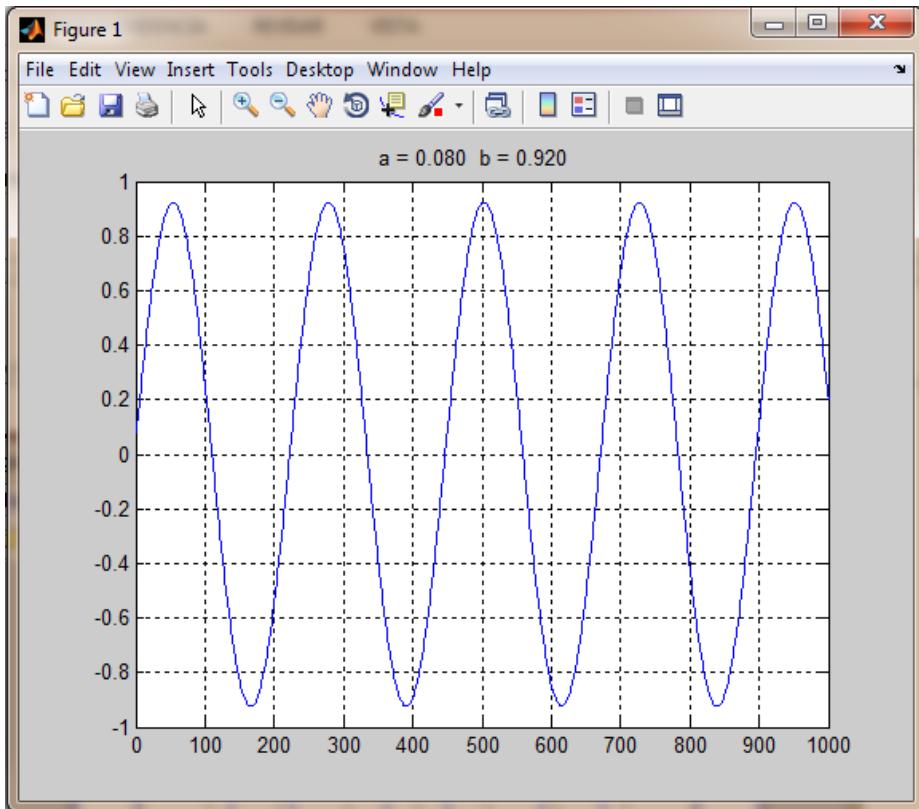


Figura 24: Grafica 2 de programa ejemplo 1

3.3.3 Modificación del programa “Streamer” para graficar los datos

Como se describió en el sub capítulo 3.2.3, la aplicación “Streamer” de Cypress permite recibir, mediante distintas configuraciones, los datos que ingresan a través del puerto USB, de una forma eficiente. A este código, se lo modifica incluyendo el motor *Engine* para poder graficar esos datos. A continuación se describen las modificaciones realizadas para lograr el propósito.

En el código de “Streamer”, el bucle que recibe continuamente los paquetes, tiene una condición que es evaluada al final de cada recepción, la cual, verifica si se ha alcanzado la recepción de todos los paquetes que resultan de la elección de la cantidad de paquetes por transferencia multiplicado por la cantidad de paquetes en cola que se almacenan. Cuando esta condición se cumple, se da por terminada la transferencia actual, se calcula la tasa de transferencia y la cantidad de bytes transmitidos. A prueba y error, se llegó a la conclusión que para poder graficar correctamente los datos, la cantidad de datos que se reciben deben ser 16 paquetes por transferencia y 64 paquetes en cola. De esta forma se obtiene un programa estable entre la recepción de datos y la gráfica. Otras combinaciones para mayor tasa de datos hacían muy lento el programa y no permitía graficar correctamente.

A continuación, se muestra la modificación implementada en el código al final de la condición para obtener los datos recibidos:

```

data_cpy = buffers[i];//AQUI VAN BYTE A BYTE RECIBIDOS PARA CARGALOS EN LA VARIABLE DE
                     //MATLAB*****
i++;

for (int e=0;e<rLen;e+=4)
{
    datos_s[indice_matlab] = data_cpy[e+3]*0 + ((data_cpy[e+2])*0) +
    ((data_cpy[e+1])*256) + ((data_cpy[e])); //Convierte los 4 octetos correspondientes
                                              //con sus pesos a los 32 bits en 1 solo double.
    datos_s[indice_matlab] = (((datos_s[indice_matlab])*(0.99))/(4095)); //Realiza la
                                              cuenta para convertir a volts los 12 bits
    time[indice_matlab]= indice_matlab;
    indice_matlab++;
}

if (i == QueueSize) //Only update the display once each time through the Queue
{
    i=0;
    indice_matlab =0; //Volvemos a inicializar el indice de para la cantidad de datos en
                      //MatLab.
    ShowStats(t1, BytesXferred, Successes, Failures);
}

} // End of the infinite loop

```

La primera línea de este fragmento de código (“`data_cpy = buffer[i];`”) va transfiriendo desde el puntero tipo UCHAR “`buffers`”, que es el que se pasa por referencia en la función de “`FinishDataXfer`” (explicada en el sub capítulo 3.2.3) y recibe cada paquete de 1024 bytes multiplicado por 16 paquetes en este caso, al puntero tipo UCHAR “`data_cpy`” que por cada transferencia almacena 16384 bytes (1024 x 16). A continuación se realiza una conversión por pesos decimales, mediante un bucle “`For`” con paso 4, hasta alcanzar el total de bytes recibidos (“`rLen`” = 16384) que por cada 4 bytes recibidos, genera 1 dato tipo double (tipo de dato que utiliza MatLab de forma más simple) en la variable “`datos_s[i]`”. Esto se realiza de esta forma porque como se explicara en los siguientes capítulos, la interface de la placa ZedBoard envía un dato de 32 bits, de forma que al recibirlos se deben contar 4 bytes consecutivos y sumarlos con sus pesos decimales respectivos, para formar uno nuevamente de 32 bits. Además, el conversor analógico-digital, es de 12 bits con rango de entrada de 0 a 1Volt. Por lo tanto los primeros dos bytes no se tienen en cuenta, y de ahí que sus pesos decimales, se multipliquen por 0. Luego para convertir el dato double a la representación en unidad de Volts, es que por regla simple de 3, se multiplica el valor actual por 0,99 (1V) y se divide en el peso que representan 12 bits en representación decimal (4095).

Se puede observar luego, que se va asignando un índice “`índice_matlab`” a otro doblé llamado “`time`” que va generando los índices correspondiente por cada dato convertido a Volts, para luego, poder graficar en MatLab los datos convertidos con su respectivos índices.

Continuando con el código, se observa que al alcanzar la condición del tamaño de cola (“QueueSize”) de a 16 paquetes recibidos por cada vuelta, se inicializan nuevamente en cero los índices “*i*” e “índice_matlab”. Esto, porque ya se alcanzó la transferencia completa y se va a iniciar una nueva transferencia de 16 paquetes por 64 en cola. Finalmente se ve que se llama a la función “ShowStats(*t1, bytesXferred, successes, failures*)” que es la encargada de mostrarlos resultados tal cual se describió en el sub capitulo 3.2.3. Dentro de esta función se llama a la función creada “GraficarMatLab”. En esta última, se realizan los pasos explicados en el sub capítulo 3.3.1 para utilizar el motor engine de MatLab. Por lo tanto, se crean las variables “mxArray” para los datos, se copian los datos a las variables creadas, y por último se colocan en el espacio de trabajo de MatLab. Se muestra a continuación el código de la función correspondiente a modo de aclaración:

```
static void GraficarMatLab()
{
    //eng::Engine *ep; //Se declaro global arriba después de los include;

    eng::mxArray *baseTiempo = NULL,*valores = NULL;
    eng::mxArray *TimeDiviMat = NULL, *VoltDiviMat = NULL;

    TimeDiviMat = eng::mxCreateDoubleMatrix(1,1,eng::mxREAL);
    VoltDiviMat = eng::mxCreateDoubleMatrix(1,1,eng::mxREAL);
    memcpy((void *)eng::mxGetPr(TimeDiviMat),(void *)TimeDivi,sizeof(TimeDivi));
    memcpy((void *)eng::mxGetPr(VoltDiviMat),(void *)VoltDivi,sizeof(VoltDivi));
    eng::engPutVariable(ep, "TimeDiviMat", TimeDiviMat);
    eng::engPutVariable(ep, "VoltDiviMat", VoltDiviMat);
    /*
     * Creamos una variable para nuestros datos base de tiempo
     */
    baseTiempo = eng::mxCreateDoubleMatrix(cantidad, 1, eng::mxREAL);

    /*
     * Creamos una variable para nuestros datos recibidos
     */
    valores = eng::mxCreateDoubleMatrix(cantidad, 1, eng::mxREAL);

    //Copiamos los datos a las variables del tipo mxArray
    memcpy((void *)eng::mxGetPr(baseTiempo),(void *)time,sizeof(time));

    //Colocamos la variable baseTiempo en el workspace de MatLab
    eng::engPutVariable(ep, "baseTiempo", baseTiempo);

    //Copiamos los datos a las variables del tipo mxArray
    memcpy((void *)eng::mxGetPr(valores),(void *)datos,sizeof(datos));

    //Colocamos la variable valores en el workspace de MatLab
    eng::engPutVariable(ep, "valores", valores);

    //Limpiamos cualquier grafica anterior
    eng::engEvalString(ep, "delete(linea)");
}
```

```

//eng::engEvalString(ep, "title('')");

//Graficamos en MatLab
eng::engEvalString(ep, "linea=plot(baseTiempo,valores,'g');");
//eng::engEvalString(ep,"grid");
eng::engEvalString(ep, "maximo = max(max(valores));");
eng::engEvalString(ep, "minimo = min(min(valores));");
eng::engEvalString(ep,"title(['Osciloscopio    Maximo:' num2str(maximo) 'V Minimo: ' num2str(minimo) 'V'])");

eng::engEvalString(ep,"VentanaDeTiempo=TimeDiviMat*1000;");
eng::engEvalString(ep,"xlim([1 VentanaDeTiempo])");
eng::engEvalString(ep, "xlabel(['Ventana de Tiempo (mS)' num2str(TimeDiviMat)])");
eng::engEvalString(ep, "Div = VoltDiviMat/100;");
eng::engEvalString(ep,"ylim([0 Div])");

} //Fin de GraficarMatLab ();

```

En el código que se muestra anteriormente, se observa que se crean primero las variables tipo *mxArray* en las cuales se van a cargar los datos provenientes de la recepción de datos por USB (*valores*) y sus correspondientes índices (*baseTiempo*). Luego, se copian los datos de las variables *double*, a las variables tipo *mxArray*. A continuación, se colocan las variables en el espacio de trabajo de MatLab y desde ahí se utilizan mediante la función *engEvalString*. Además se buscan los máximos, mínimos mediante las instrucciones de MatLab “*max* y *min*” y se establecen los límites de la gráfica con los valores que se eligen a través de la ventana grafica del programa de VS2010. Esta última selección, se realiza mediante los *comboBox* “*Volts Division*” y “*Ventana de tiempo*” que se explicaran unos párrafos más adelante.

Para mejorar el rendimiento en cuanto a velocidad del programa, se declaró el puntero tipo Engine, “*ep*” mediante el cual se pasa la información a MatLab y las variables donde se cargan los datos recibidos, globales. De esta forma, son accedidas (con precaución) desde todas las funciones que las necesiten en vez de ser pasadas por referencia. También, se generó una función llamada “*AbrirGraficaMatLab()*” la cual inicializa las condiciones necesarias para la gráfica, y abre el puntero “*ep*”. El código la misma, que no tiene nada en particular, solo a modo de aclaración, es mostrado a continuación:

```

void AbrirGraficaMatLab()
{
    eng::mxArray *TimeDiviMat = NULL, *VoltDiviMat = NULL;
    TimeDiviMat = eng::mxCreateDoubleMatrix(1,1,eng::mxREAL);
    VoltDiviMat = eng::mxCreateDoubleMatrix(1,1,eng::mxREAL);
    memcpy((void *)eng::mxGetPr(TimeDiviMat),(void *)TimeDivi,sizeof(TimeDivi));
    memcpy((void *)eng::mxGetPr(VoltDiviMat),(void *)VoltDivi,sizeof(VoltDivi));
    eng::engPutVariable(ep, "TimeDiviMat", TimeDiviMat);
    eng::engPutVariable(ep, "VoltDiviMat", VoltDiviMat);
    eng::engEvalString(ep, "whitebg;");
    eng::engEvalString(ep, "grid");
    eng::engEvalString(ep, "title('Osciloscopio');");
    eng::engEvalString(ep, "xlabel('Datos tomados (unidades)');");
}

```

```

eng::engEvalString(ep, "ylabel('Volts');");
eng::engEvalString(ep, "hold;");
eng::engEvalString(ep, "set(gca,'Color','black');");
eng::engEvalString(ep, "linea = plot(0,0);");
eng::engEvalString(ep, "Div = VoltDivMat/100;");
eng::engEvalString(ep, "ylim([0 Div]);");
} //Fin de AbrirGraficaMatLab ();

```

El programa “Streamer” también se modificó visualmente para que sea útil a la aplicación necesaria en este proyecto. En la Figura 25: Programa Tesis Osciloscopio se observa cómo queda finalmente el mismo modificado y cuyo nombre se cambió, a “Tesis Osciloscopio”:

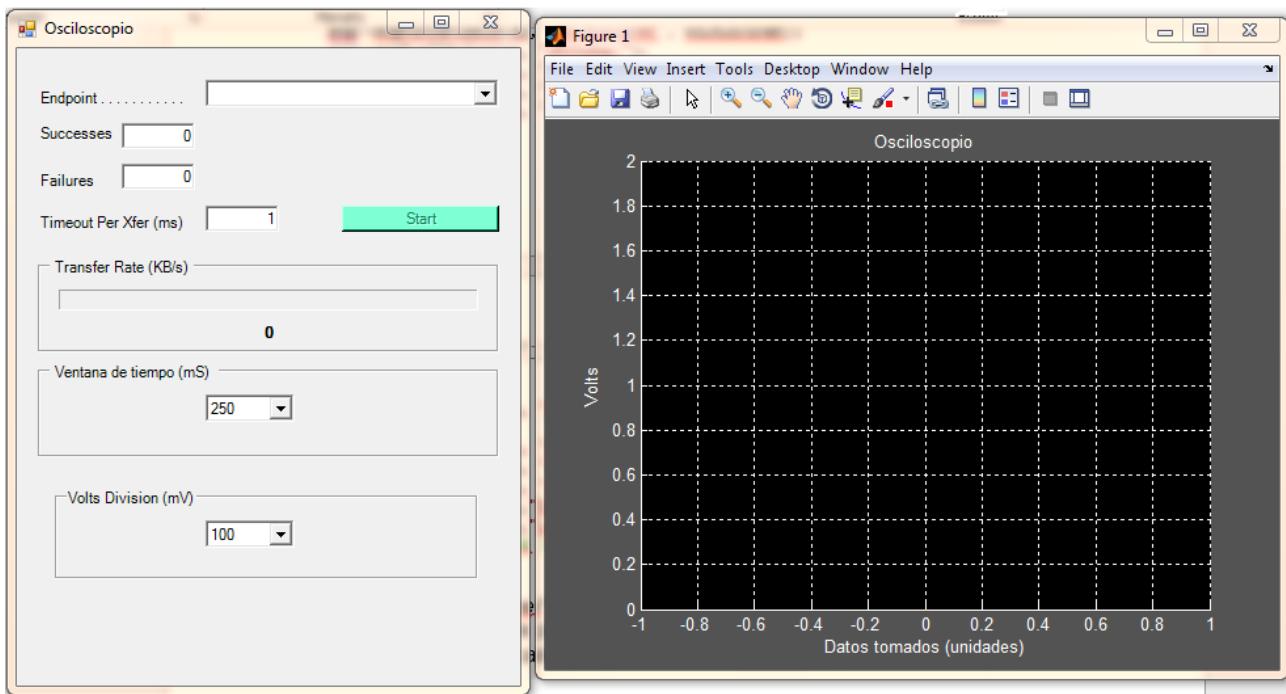


Figura 25: Programa Tesis Osciloscopio

Se puede observar en la Figura 25, que entre los cambios que se realizaron con el programa “Streamer” que se mostró en la Figura 21, se añadieron las selecciones llamadas en VS 2010 “comboBox”, “Ventana de tiempo (mS)” y “Volts Division (mV)”. Los mismos, permiten seleccionar entre un par de valores delimitados, tal cual un osciloscopio, los cuadros de división por volt y la ventana de tiempo que se quiere observar de la gráfica. Esta última, si bien no es igual que en un osciloscopio, donde se elige también la división de tiempo por cuadro, por practicidad para no retardar analizando datos la gráfica continua, se define en cambio, la ventana de tiempo que se quiere observar. Se dejó del programa original la barra de tasa de transferencia, así como también la cantidad de paquetes que se recibieron exitosamente y los que no. Además, se dejó el cuadro que muestra el punto de llegada (end point) para controlar que el dispositivo que está conectado sea el correcto.

De esta forma se ha mostrado hasta aquí, la interface gráfica creada mediante el uso de Visual Studio en lenguaje Visual C++ para la comunicación USB, y también la

transferencia y grafica de los datos a MatLab mediante el uso de la herramienta Engine. En las secciones posteriores se mostrará esta aplicación funcionando en conjunto con todo el proyecto.

Capítulo 4: Interface conversor ADC - FPGA – EZ USB FX3 – USB

En la Figura 26 ¡Error! No se encuentra el origen de la referencia.se observa el diagrama de bloques de la interface que se explica en este capítulo.

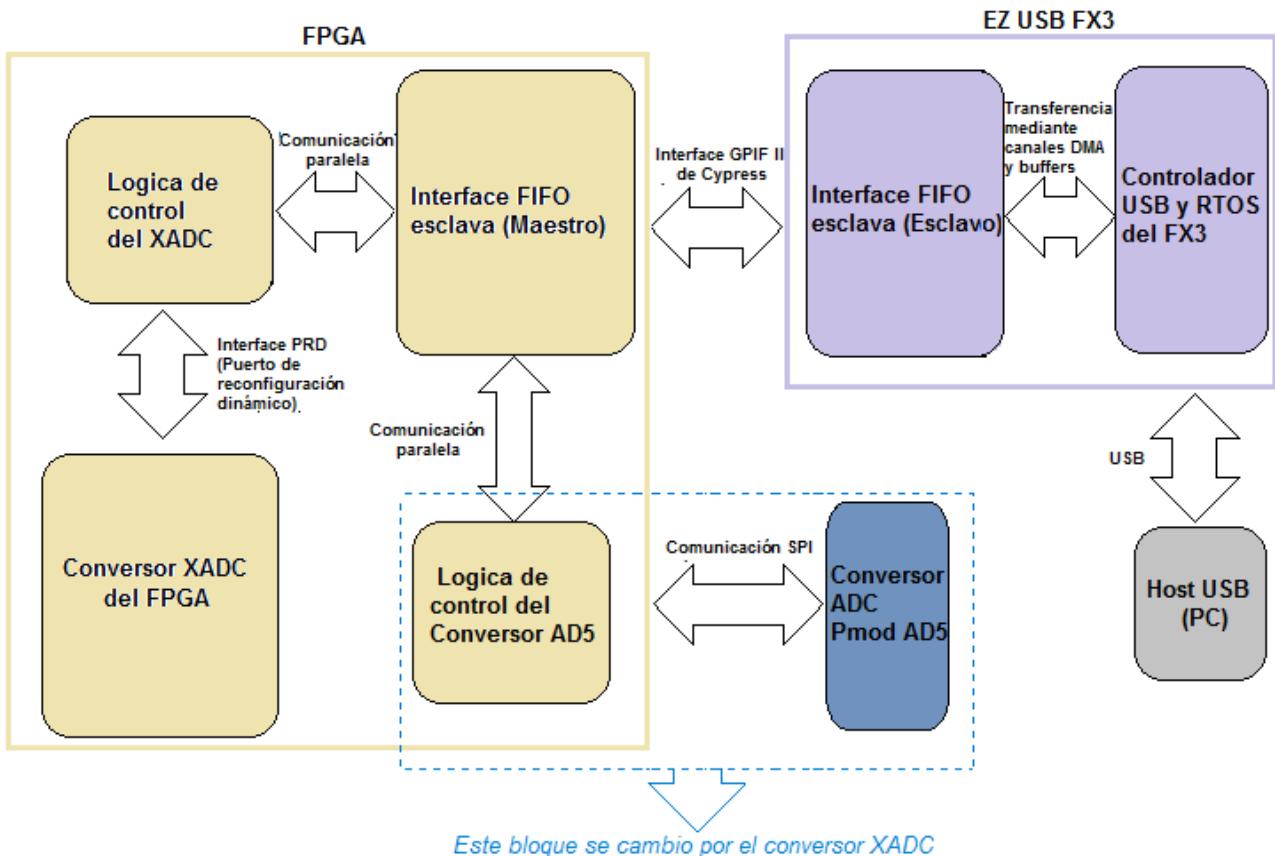


Figura 26: Diagrama de bloques de interface ADC -FPGA - EZ USB FX3 – USB

4.1 Interface FIFO Esclavo

Para realizar la comunicación del microcontrolador con el FPGA, el cual enviará los datos que tome desde el conversor, se partió de una interface típicamente usada para estas aplicaciones propietaria de Cypress, en donde se debe implementar una comunicación Micro-FPGA.

Cypress en la nota de aplicación AN 65974, que se puede buscar en la página www.cypress.com, muestra cómo implementar la misma y adaptarla para diferentes aplicaciones. Se basa este capítulo en la modificación y adaptación de la nota de aplicación a las necesidades del proyecto, así como también la creación de código VHDL para la toma de datos mediante el conversor ADC y la configuración y edición del código para el EZ USB FX3.

La interface FIFO Esclavo, (FIFO) de aquí en adelante, es una comunicación sincrónica paralela que permite transferencias de hasta 100Mhz. La misma, usa una de las características principales del EZ USB FX3, una Interface Paralela Programable (GPIF) la cual es una máquina de estados programable, que permite una configuración flexible de entradas salidas con bus de datos paralelo de 16 y 32 bits, 14 señales de control como banderas, dirección de envío de entrada y salida, soporta el perfil maestro o esclavo y hasta 256 estados programables.

Las transiciones de estado de la GPIF dependen de las señales de control de entrada. Las señales de salida de control son administradas por las transiciones de la GPIF. El comportamiento de la máquina de estado, está definida por un descriptor, el cual es diseñado para coincidir con las especificaciones de la interface. El descriptor, es en esencia, una serie de registros de configuraciones programables. El espacio de registros del EZ USB FX3, dedica 8kB a la configuración de la GPIF. Esta configuración tan flexible permite acceder los buffers internos del FX3 para realizar comunicaciones USB y soportar altas tasas de transferencias. Se muestra en la Figura 27, el diagrama de la interface entre el EZ USB FX3 y el FPGA.

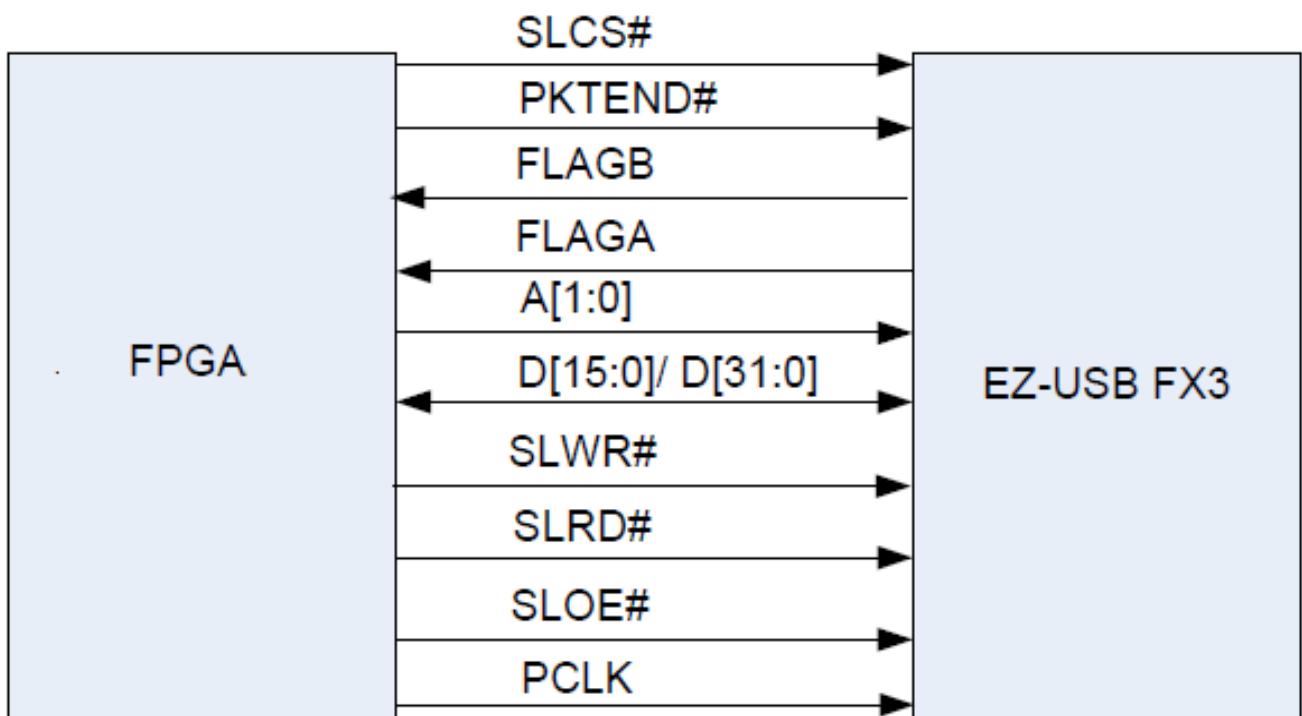


Figura 27: Diagrama de la Interface FIFO M/E

A continuación se detallan las señales de la FIFO:

- SLCS#: Esta es la señal de selección de la interface (Comúnmente llamada Chip Select). Debe ser activada para poder acceder a la FIFO M/E.
- SLWR#: Esta es la señal para escribir en la FIFO M/E. Debe ser activada para poder realizar transferencias de escritura en el interface.

- SLRD#: Esta es la señal de lectura de la FIFO M/E. Debe seleccionarse para poder realizar transferencias de lectura desde la interface.
- SLOE#: Esta es la señal de habilitación de salida. Esta permite que el bus de la interface sea administrado por el FX3. Debe ser activado para realizar transferencias de lectura desde la FIFO.
- FLAGA/FLAGB: Estas son las banderas de salida del FX3. Las banderas indican la disponibilidad de un zócalo del FX3.
- A[1:0]: Estas son la dirección de 2 bit del bus de la FIFO.
- PKTEND#: Esta señal se activa para escribir un paquete corto, o un paquete de longitud cero a la FIFO.
- PCLK: Este es el reloj de la interface.

La Figura 28 muestra la asignación por defecto usada para la FIFO en el FX3, tanto para bus de 16 bits o de 32 bits. En la misma también se muestran los pines de uso general (GPIO) y otras interfaces seriales (UART/SPI/I2S), que están disponibles cuando la GPIF está configurada para la Interface FIFO esclavo.

EZ-USB FX3 Pin	Synchronous Slave FIFO Interface with 16-bit Data Bus	Synchronous Slave FIFO Interface with 32-bit Data Bus
GPIO[17]	SLCS#	SLCS#
GPIO[18]	SLWR#	SLWR#
GPIO[19]	SLOE#	SLOE#
GPIO[20]	SLRD#	SLRD#
GPIO[21]	FLAGA	FLAGA
GPIO[22]	FLAGB	FLAGB
GPIO[23]	FLAGC	FLAGC
GPIO[24]	PKTEND#	PKTEND#
GPIO[25]	FLAGD	FLAGD
GPIO[28]	A1	A1
GPIO[29]	A0	A0
GPIO[0:15]	DQ[0:15]	DQ[0:15]
GPIO[16]	PCLK	PCLK
GPIO[33:44]	Available as GPIOs	DQ[16:27]
GPIO[45]	GPIO	GPIO
GPIO[46]	GPIO/UART_RTS	DQ28
GPIO[47]	GPIO/UART_CTS	DQ29
GPIO[48]	GPIO/UART_TX	DQ30
GPIO[49]	GPIO/UART_RX	DQ31
GPIO[50]	GPIO/I2S_CLK	GPIO/I2S_CLK
GPIO[51]	GPIO/I2S_SD	GPIO/I2S_SD
GPIO[52]	GPIO/I2S_WS	GPIO/I2S_WS
GPIO[53]	GPIO/SPI_SCK /UART_RTS	GPIO/UART_RTS
GPIO[54]	GPIO/SPI_SSNI/UART_CTS	GPIO/UART_CTS
GPIO[55]	GPIO/SPI_MISO/UART_TX	GPIO/UART_TX
GPIO[56]	GPIO/SPI_MOSI/UART_RX	GPIO/UART_RX
GPIO[57]	GPIO/I2S_MCLK	GPIO/I2S_MCLK

Figura 28: Asignación de pines del FX3 para la FIFO.

Un procesador externo, o un dispositivo como un FPGA (funcionando como maestro), puede realizar una transferencia de un solo ciclo o masiva a los buffers internos del EZ USB FX3 de la FIFO. Este dispositivo maestro, administra los dos bits de dirección sobre la línea ADDR y selecciona la señal de lectura o escritura. El FX3 administra las señales FLAGs para indicar una condición de buffer lleno o vacío. A continuación, en la Figura 29 se muestra una secuencia de acceso a la FIFO y el diagrama de tiempo:

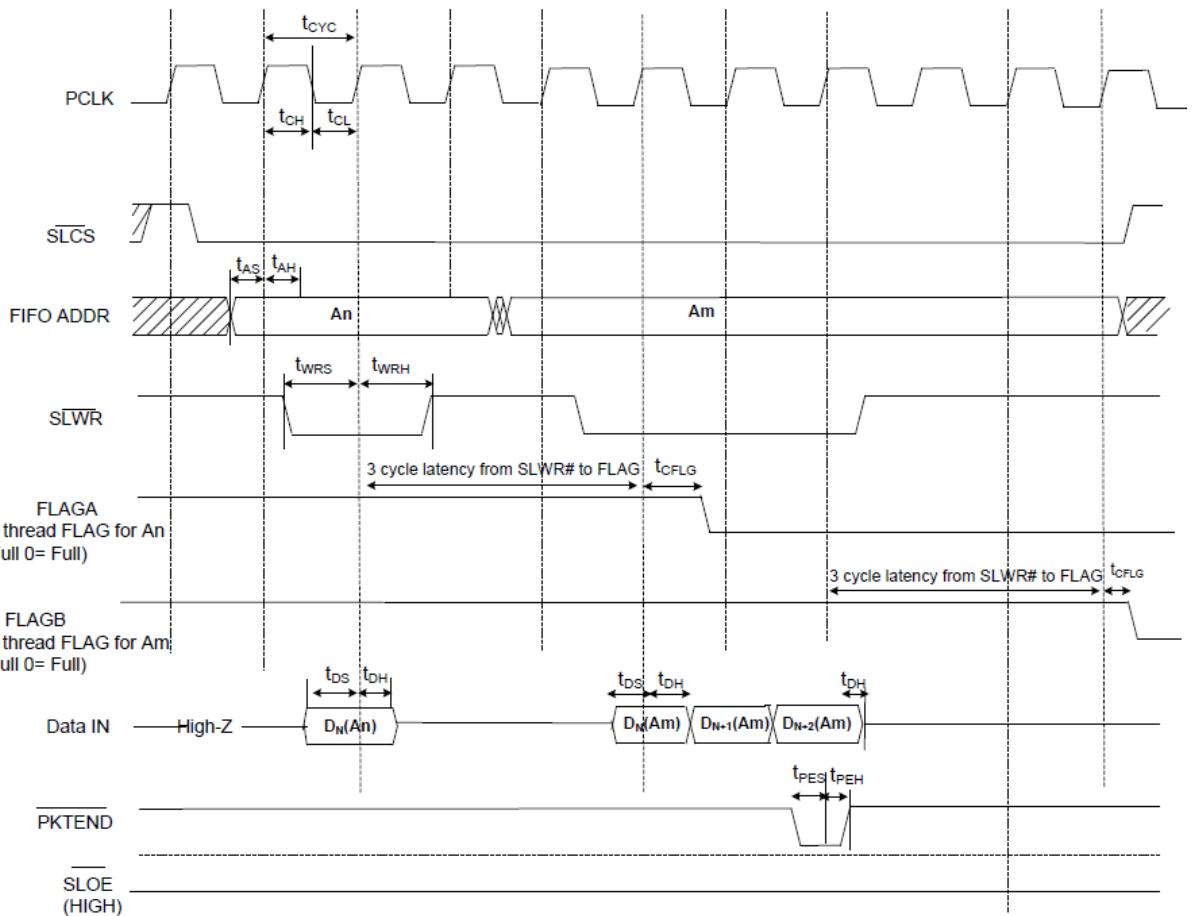


Figura 29: Secuencia de acceso a la FIFO y diagrama de tiempos.

En este proyecto, como se realiza un envío continuo hacia la FIFO, se explica la secuencia de escritura del diagrama de tiempo mostrado en la Figura 29.

Primero, mientras la dirección de la FIFO es estable, la señal SCLS# se activa. Luego, el maestro coloca la información o dato en el bus de datos e inmediatamente después la señal SLWR# se activa. Mientras la señal esté activa, el dato es escrito a la FIFO, en el flanco ascendente de PCLK y el puntero del buffer la misma es incrementado. La bandera de la FIFO es actualizada después de un retardo de tCFLG con respecto al flanco ascendente del reloj. Esta misma secuencia se utiliza en el envío masivo, con la diferencia que en este último tipo, las señales SLWR y SCLS se dejan activadas. En este caso, el valor en el bus de datos, es transferido a la FIFO en cada flanco ascendente de PCLK, y es actualizado el puntero del buffer.

4.2 Transferencias, hilos y zócalos

En esta sección se explica brevemente los conceptos necesarios para las transferencias desde y hacia el FX3.

Zócalos, Descriptores DMA, Buffer DMA e Hilo GPIF:

Un zócalo, es un punto de conexión entre un bloque de hardware de un periférico y la memoria RAM del FX3. Cada bloque de hardware de los periféricos en el FX3, tales como USB, GPIF, UART y SPI, tiene un número fijo de zócalos asociados a cada uno de ellos. El número de datos independientes que fluyen a través de un periférico, es igual al número de zócalos que posee. La implementación de un zócalo, incluye una serie de registros que apuntan al descriptor DMA activo y habilitan la bandera de interrupción asociada al zócalo.

Un descriptor DMA, es una serie de registros reservados en la RAM del FX3. Estos mantienen la información sobre la dirección y tamaño del buffer DMA, así como punteros al siguiente descriptor DMA. Estos punteros generan una cadena de descriptores DMA.

Un buffer DMA, es una sección de RAM utilizada para almacenamiento parcial de datos transferidos a través del FX3. Los buffers DMA son asignados desde la RAM por el FX3 y sus direcciones se almacenan como parte de los descriptores DMA.

Un hilo GPIF es un camino de datos dedicado en el bloque GPIF que conecta los pines de información externos con un zócalo. Los zócalos pueden señalizarse entre ellos a través de eventos o pueden comunicarse con la CPU del FX3 través de interrupciones. El programa configura esta señalización. A modo de ejemplo, en un envío de datos desde el bloque GPIF hacia el bloque USB. El zócalo GPIF puede comunicarle al zócalo USB que ha completado un buffer DMA, y luego, el zócalo USB puede indicarle al zócalo GPIF que el buffer DMA está vacío. Esta implementación se llama canal DMA automático. Este, es usado cuando la CPU del FX3 no tiene que modificar ningún dato en el flujo de datos (Como sucede en este proyecto en donde el envío es directo al USB sin ser utilizado por la CPU del FX3).

4.3 Firmware del FX3 y código VHDL del FPGA

Como se mencionó al comienzo de este capítulo, se usó como base para esta interface, la nota de aplicación de Cypress. La misma, incluye el firmware para el FX3 y un código VHDL para un FPGA que permite la comunicación de la interface FIFO esclavo. Para poner en marcha la misma, se deben configurar algunas opciones que se explicarán a continuación.

4.3.1 Firmware del EZ USB FX3

Primero, teniendo instalado el software Eclipse EZ USB Suite, que viene incluido con el kit del EZ USB FX3, debemos importar el proyecto que viene en la nota de aplicación. Para realizar esto una vez abierto el entorno de desarrollo nos dirigimos a: "*File>Import*" se abrirá una ventana nueva en donde seleccionamos: "*Project>Existing Project into Workspace*". Se nos presenta una nueva ventana en donde buscamos la dirección donde tengamos instalado el programa:

"C:\Cypress\FX3\SDK\1.2\firmware\slavfifo_examples\slfifoSync."

Con el nombre “SlaveFifoSync” y lo elegimos (Para no editar el proyecto original, debemos seleccionar en esta última ventana, la casilla: “Copy project into the actual Workspace”). En las Figura 30 y Figura 31 se puede observar los pasos indicados anteriormente.

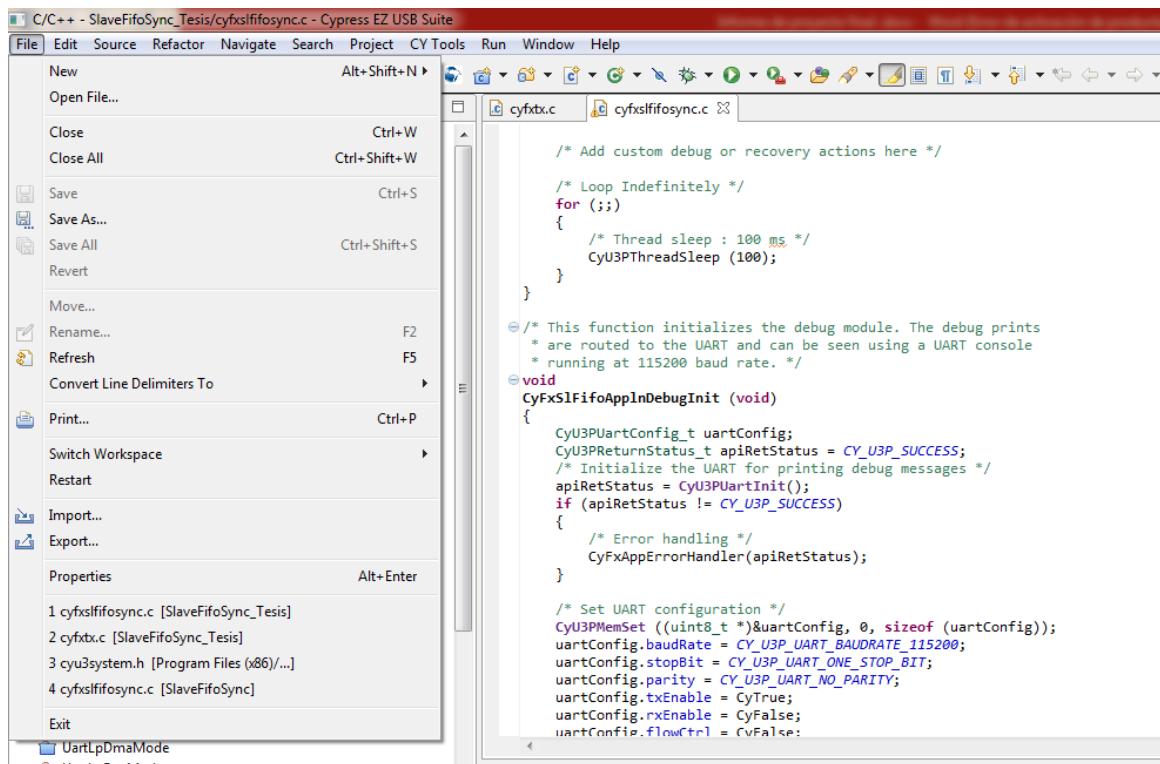


Figura 30: Importar proyecto en Eclipse EZ USB Suite.

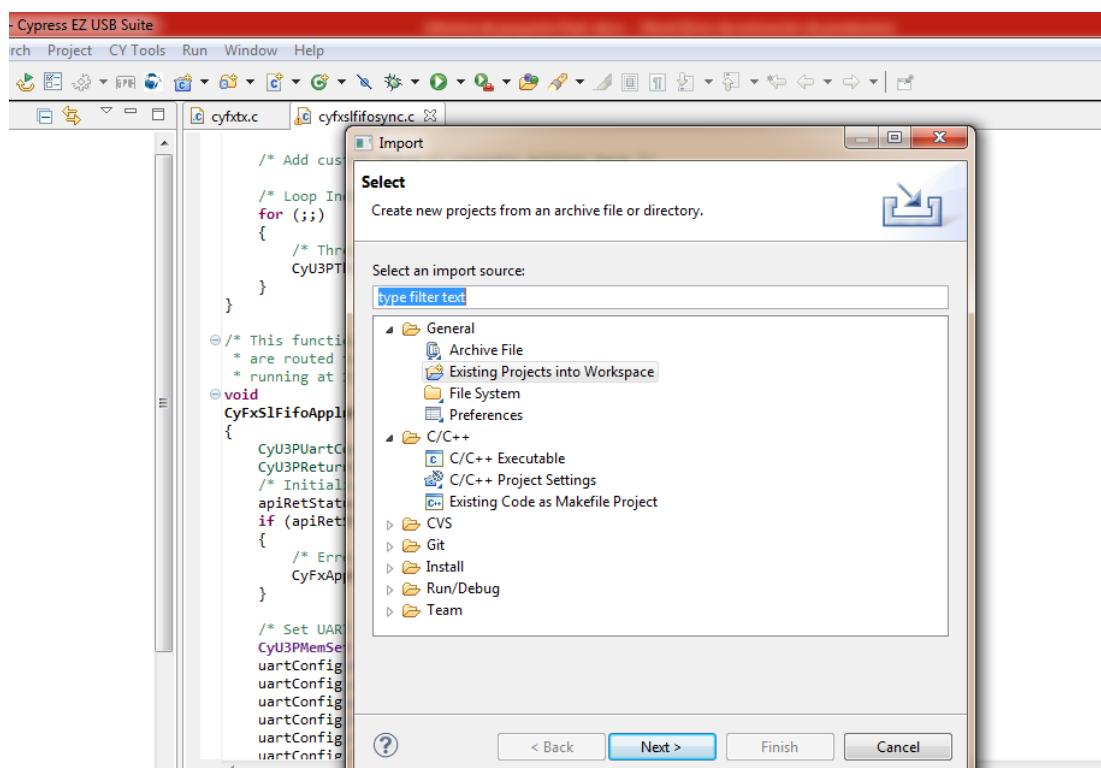


Figura 31: Seleccionar proyecto existente al Workspace

Una vez abierto el proyecto. Buscamos el archivo de cabecera “yfxslfifosync.h” y dentro de este seleccionamos la configuración de bus de datos de 32 bits. Para esto debemos asignar un “1” a la constante **CY_FX_SLFIFO_GPIF_16_32BIT_CONF_SELECT**. Luego, para poder trabajar con la interface a 100Mhz, debemos pasar como parámetro **setSysClk400** en la función **CyU3PdeviceInit()**. De esta forma, configuraremos el PLL a una frecuencia de 400Mhz porque el FX3 divide la frecuencia del PLL en 4.

Este firmware incluye configurados los descriptores para las interfaces BULK IN y BULK e ISOC IN e ISOC OUT con sus respectivos PID = 0x00F1 y VID = 0x04B4, los cuales son los que el programa de VS 2010 busca para conectarse. Estos enlazan perfectamente con los drivers que se instalan al conectar la placa del kit EZ USB FX3, los cuales están diseñados para estos firmwares.

Por último, en la placa del EZ USB FX3 debemos configurar algunos jumpers para que la aplicación pueda funcionar correctamente. En la Figura 32 se muestran tal cual la nota de aplicación de Cypress, como deben colocarse los mismos:

Sl. No.	Jumper/Switch	Pins to be Shorted using Jumpers	Function
1	J100	1 and 2	GPIO[21]/CTL[4] – configured as FLAGA
2	J136	3 and 4	VIO1(3.3V)
3	J144	3 and 4	VIO2(3.3V)
4	J145	3 and 4	VIO3(3.3V)
5	J146	3 and 4	VIO4(3.3V)
6	J134	4 and 5	VIO5(3.3V)
7	J135	2 and 3	CVDDQ(3.3V)
8	J143	3 and 4	VBATT(3.3V)
9	J101	1 and 2	GPIO[46] = UART_RTS
10	J102	1 and 2	GPIO[47] = UART_CTS
11	J103	1 and 2	GPIO[48] = UART_TX
12	J104	1 and 2	GPIO[49] = UART_RX
13	J96 and SW25	2 and 3	PMODE0 pin state (ON/OFF) selection using SW25. SW25.1 should be OFF
14	J97 and SW25	2 and 3	PMODE1 pin state (ON/OFF) selection using SW25. SW25.1 should be OFF
15	J98	1 and 2	PMODE2 pin floating
16	J72	1 and 2	RESET
17	J53	1 and 2	Bus powered
18	SW9	The switch should point to the direction labeled VBUS_IN	Bus powered
19	J156	Place a jumper to short	Powers Samtec connector
20	J45	2 and 3	GPIO[59] - Reset to the FPGA from FX3

Figura 32: Configuración de jumpers en la placa del FX3

Con estas configuraciones ya tenemos el FX3 configurado para transmitir a la PC mediante USB, los datos que reciba a través del maestro por medio de la FIFO.

4.3.2: Código VHDL de la FIFO para el FPGA

El código VHDL desarrollado por Cypress en la nota de aplicación para la interface FIFO, posee 4 tipos transferencias que pueden configurarse entre el FPGA y el FX3:

- *Loopback*: En esta configuración, el FPGA primero lee un buffer completo desde el FX3 y luego lo escribe de vuelta hacia el FX3. El controlador USB debe administrar transferencias de entrada y salida para poder transmitir y recibir los datos.
- *Short packet*: Este modo, permite que el FPGA envíe un paquete completo seguido de un paquete corto hacia el FX3. El controlador USB debe administrar transferencias de entrada solamente para recibir los datos.
- *Zero length packet*: Permite transferir desde el FPGA un paquete completo seguido de un paquete sin datos hacia el FX3. El controlador USB debe administrar transferencias de entrada solamente para recibir los datos.
- *Streaming (IN)*: El FPGA realiza transferencias unidireccionales de salida, esto es, que continuamente escribe datos hacia el FX3 mediante la FIFO. El USB solamente debe administrar paquetes de entrada para recibir los datos.
- *Streaming (OUT)*: El FPGA realiza transferencias unidireccionales de entrada, esto es que continuamente, recibe datos desde el FX3 a través de la FIFO. El USB debe administrar transferencias de salida solamente.

El código del FPGA se basa en una máquina de estados la cual presenta 6 estados principales según el tipo de transferencia a realizar. Se muestran los mismos a continuación en la Figura 33:

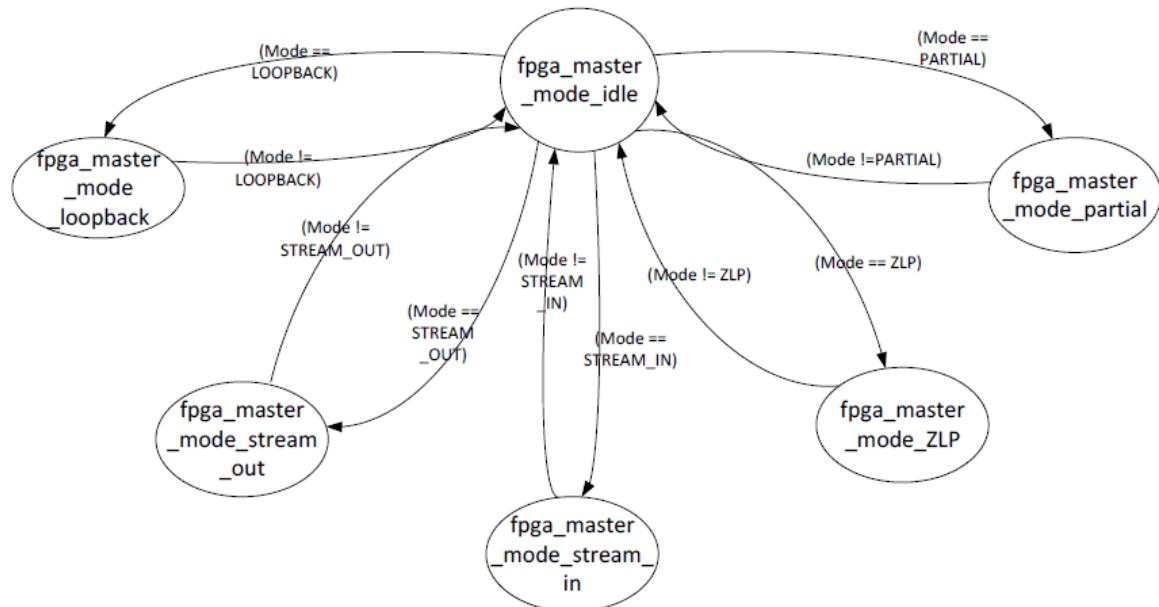


Figura 33: Maquina de estados del FPGA para la interface FIFO

Cada uno de estos estados, también, es a su vez otra máquina de estados. En el caso que compete a este proyecto, vamos a utilizar la transferencia “STREAM IN” por lo que solo se detallará la misma pero las demás estados funcionan de forma similar.

La transferencia “STREAM IN” posee 4 estados:

- *stream_in_idle*: Este estado es el inicial para este tipo de transferencia. El mismo, inicializa todos los registros y señales usados en la máquina de estados y espera por que la señal *flaga_d* sea igual a “1”. Las señales de control en este estado son: PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] = 0; Cuando *flaga_d* es ‘1’, cambia el estado actual a *stream_in_wait_flagb*.
- *stream_in_wait_flagb*: En este estado espera a que la señal *flagb_d* sea igual a “1” y en ese momento, cambia el estado actual a *stream_in_write*.
- *stream_in_write*: En cualquier momento que la señal *flagb_d* = 1, la máquina de estados entrará en este modo y comenzará a escribir hacia la interface FIFO. El estado de las señales de control son: PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 0; A[1:0] = 0;
- *stream_in_write_delay*: En cualquier momento que la señal *flagb_d* = ‘0’ la máquina de estados entrará en este estado. El estado de las señales de control serán: PKTEND# = 1; SLOE# = 1; SLRD# = 1; SLCS# = 0; SLWR# = 1; A[1:0] = 0; Luego de un ciclo de reloj, el estado cambiará a *stream_in_idle*.

En la imagen Figura 34 se puede observar los estados de STREAM IN explicados anteriormente:

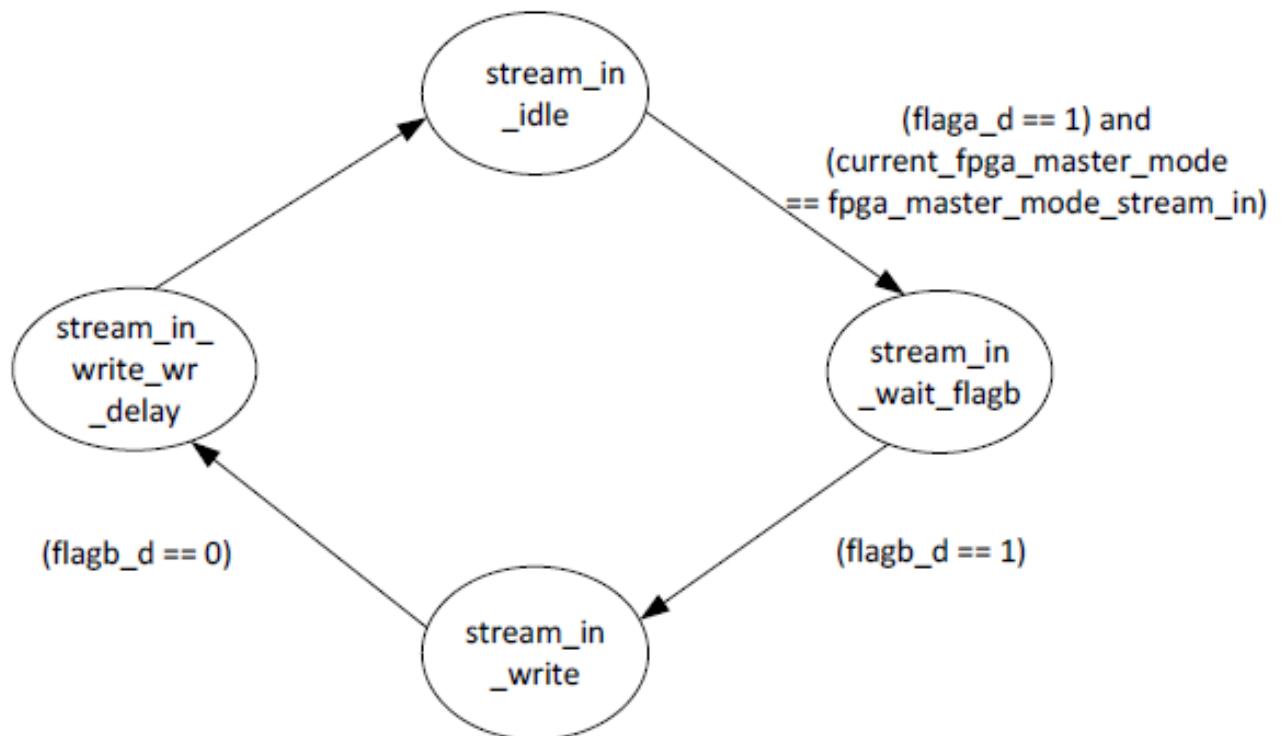


Figura 34: Maquina de estados del FPGA en STREAM IN

El código del VHDL para el FPGA, debe ser cargado en el programa Vivado Suite de Xilinx para poder programar la placa ZedBoard. Este software tiene una versión de licencia gratuita para estudiantes llamada *WebPack edition*. La misma, es limitada con respecto a los dispositivos de Xilinx con los que puede trabajar, pero entre los kits que si permite, se encuentra la placa ZedBoard, por lo tanto es suficiente para desarrollar el código necesario.

Una vez instalado el Vivado Suite (para este proyecto se utilizó la versión 2016 pero versiones anteriores también son compatibles con el código.) cargamos el código de la FIFO para el FPGA, el cual viene incluido en la nota de aplicación como se mencionó anteriormente. El mismo, cuenta con 7 fuentes en vhdl: *slaveFIFO2b_fpga_top.vhd*, *slaveFIFO2b_loopback.vhd*, *slaveFIFO2b_partial.vhd*, *slaveFIFO2b_streamIN.vhd*, *slaveFIFO2b_streamOUT.vhd*, *clk_wiz_v3_6_2.vhd* y *slaveFIFO2b_fp.vhd*. Estas ejecutan de la misma forma que se explicó la máquina de estados para el EZ USB FX3 y se mostró en la Figura 33 y Figura 34.

Las distintas fuentes mencionadas anteriormente, son respectivamente los 5 tipos de transferencias posibles, más el código de un reloj (*clk_wiz_v3_6_2.vhd*) el cual a partir de un reloj de 27 Mhz genera uno de 100 Mhz. Este reloj está incluido entre las fuentes porque la nota de aplicación estaba como ejemplo para una placa Xilinx Spartan 6, la cual, tiene un reloj disponible de 27Mhz. El código *slaveFIFO2b_fpga_top.vhd* es comúnmente llamado el *top hierarchy level* (nivel de jerarquía más alto). Este, instancia a todos los demás códigos y crea una sola estructura que incorpora a las demás dentro de él.

Para poder probar la interface funcionando completamente hasta aquí, se deben buscar en el esquemático de la placa ZedBoard, el esquemático de la placa de

interconexión FMC y los pines asignados del FX3 como se mostró en la Figura 28, para poder asignar los pines en Vivado para la placa ZedBoard. En la Figura 35 se muestra la correspondencia pin a pin desde el FX3 hasta la ZedBoard. Una vez realizada la declaración de cada señal en Vivado, se puede probar con un programa que brinda Cypress y que se instala cuando se instala el Eclipse USB Suite, llamado BulkLoop. Este, permite enviar en bucle, saliendo desde el host USB, pasando por el FX3, hasta llegar al FPGA y nuevamente recorrer el camino de vuelta hasta el host USB. En la nota de aplicación que se puede descargar desde <http://www.cypress.com/documentation/application-notes/an65974-designing-ez-usb-fx3-slave-fifo-interface> se muestran los detalles para ejecutar la aplicación, enviar, recibir datos y controlar el funcionamiento de la interface. Se muestra en la Figura 36, la recepción de datos mediante una transferencia *BulkIN*, luego de haber enviado los mismos desde el un envío *StreamOUT*.

Una vez probado el funcionamiento de la FIFO, se procedió a crear la comunicación entre el código VHDL de la FIFO y el conversor ADC para poder enviar los a través de ella.

EZ USB FX3			Conector FMC				FPGA	
GPIF II	Slave FIFO Interface	Signal Slave FIFO Name	Con. PIN J77	Señales de FX3	GPIO	Grupo de pines del FMC	Signal Connector Name	NOMBRE DEL PIN
GPIO[0]	DQ0	DQ0	68	D0	GPIO[0]	H4	FMC_CLK0_P	L18
GPIO[1]	DQ1	DQ1	74	D1	GPIO[1]	H7	FMC_LA02_P	P17
GPIO[2]	DQ2	DQ2	76	D2	GPIO[2]	H8	FMC_LA02_N	P18
GPIO[3]	DQ3	DQ3	70	D3	GPIO[3]	H10	FMC_LA04_P	M21
GPIO[4]	DQ4	DQ4	50	D4	GPIO[4]	H11	FMC_LA04_N	M22
GPIO[5]	DQ5	DQ5	80	D5	GPIO[5]	H13	FMC_LA07_P	T16
GPIO[6]	DQ6	DQ6	32	D6	GPIO[6]	H14	FMC_LA07_N	T17
GPIO[7]	DQ7	DQ7	42	D7	GPIO[7]	H16	FMC_LA11_P	N17
GPIO[8]	DQ8	DQ8	66	D8	GPIO[8]	H17	FMC_LA11_N	N18
GPIO[9]	DQ9	DQ9	89	D9	GPIO[9]	H19	FMC_LA15_P	J16
GPIO[10]	DQ10	DQ10	61	D10	GPIO[10]	H20	FMC_LA15_N	J17
GPIO[11]	DQ11	DQ11	79	D11	GPIO[11]	H22	FMC_LA19_P	G15
GPIO[12]	DQ12	DQ12	71	D12	GPIO[12]	H23	FMC_LA19_N	G16
GPIO[13]	DQ13	DQ13	81	D13	GPIO[13]	H25	FMC_LA21_P	E19
GPIO[14]	DQ14	DQ14	65	D14	GPIO[14]	H26	FMC_LA21_N	E20
GPIO[15]	DQ15	DQ15	83	D15	GPIO[15]	H28	FMC_LA24_P	A18
GPIO[16]	PCLK	CLK	44	CLK	GPIO[16]	G6	FMC_LA00_CC_P	M19
GPIO[17]	CTL[0]	SLCS#	33	CTL0	GPIO[17]	G19	FMC_LA16_N	K21
GPIO[18]	CTL[1]	SLWR#	51	CTL1	GPIO[18]	G21	FMC_LA20_P	G20
GPIO[19]	CTL[2]	SLOE#	55	CTL2	GPIO[19]	G22	FMC_LA20_N	G21
GPIO[20]	CTL[3]	SLRD#	31	CTL3	GPIO[20]	G24	FMC_LA22_P	G19
GPIO[21]	CTL[4]	FLAGA	60	CTL4	GPIO[21]	G25	FMC_LA22_N	F19
GPIO[22]	CTL[5]	FLAGB	39	CTL5	GPIO[22]	G27	FMC_LA25_P	D22
GPIO[23]	CTL[6]	GPIO	45	CTL6	GPIO[23]	G28	FMC_LA25_N	C22
GPIO[24]	CTL[7]	PKTEND#	37	CTL7	GPIO[24]	G30	FMC_LA29_P	C17
GPIO[25]	CTL[8]	GPIO	49	CTL8	GPIO[25]	G31	FMC_LA29_N	C18
GPIO[26]	CTL[9]	GPIO	41	CTL9	GPIO[26]	G33	FMC_LA31_P	B16
GPIO[27]	CTL[10]	GPIO	43	CTL10	GPIO[27]	G34	FMC_LA31_N	B17
GPIO[28]	CTL[11]	A1	54	CTL11	GPIO[28]	G36	FMC_LA33_P	B21
GPIO[29]	CTL[12]	A0	57	CTL12	GPIO[29]	G37	FMC_LA33_N	B22
GPIO[30]	PMODE[0]	PMODE[0]	36	PMODE0	GPIO[30]	D11	FMC_LA05_P	J18
GPIO[31]	PMODE[1]	PMODE[1]	46	PMODE1	GPIO[31]	D12	FMC_LA05_N	K18
GPIO[32]	PMODE[2]	PMODE[2]	30	PMODE2	GPIO[32]	D14	FMC_LA09_P	R20
INT#	INT#/CTL[15]	CTL[15]	52	INT	int#	D15	FMC_LA09_N	R21
RESET#	RESET#	RESET#	59	RESET	reset#	D17	FMC_LA13_P	L17
GPIO[33]	DQ16	DQ16	56	D16	GPIO[33]	H29	FMC_LA24_N	A19
GPIO[34]	DQ17	DQ17	85	D17	GPIO[34]	H31	FMC_LA28_P	A16
GPIO[35]	DQ18	DQ18	73	D18	GPIO[35]	H32	FMC_LA28_N	A17
GPIO[36]	DQ19	DQ19	87	D19	GPIO[36]	H34	FMC_LA30_P	C15
GPIO[37]	DQ20	DQ20	77	D20	GPIO[37]	H35	FMC_LA30_N	B15
GPIO[38]	DQ21	DQ21	75	D21	GPIO[38]	H37	FMC_LA32_P	A21
GPIO[39]	DQ22	DQ22	69	D22	GPIO[39]	H38	FMC_LA32_N	A22
GPIO[40]	DQ23	DQ23	67	D23	GPIO[40]	G2	FMC_CLK1_P	D18
GPIO[41]	DQ24	DQ24	78	D24	GPIO[41]	G3	FMC_CLK1_N	C19
GPIO[42]	DQ25	DQ25	72	D25	GPIO[42]	G9	FMC_LA03_P	N22
GPIO[43]	DQ26	DQ26	63	D26	GPIO[43]	G10	FMC_LA03_N	P22
GPIO[44]	DQ27	DQ27	58	D27	GPIO[44]	G12	FMC_LA08_P	J21
GPIO[45]	DQ28	DQ28	38	D28_UART-RTS	GPIO[46]	G13	FMC_LA08_N	J22
GPIO[46]	DQ29	DQ29	62	D29_UART-CTS	GPIO[47]	G15	FMC_LA12_P	P20
GPIO[47]	DQ30	DQ30	40	D30_UART-TX	GPIO[48]	G16	FMC_LA12_N	P21
GPIO[48]	DQ31	DQ31	64	D31_UART-RX	GPIO[49]	G18	FMC_LA16_P	J20

Figura 35: Pines de conexión entre FPGA y FX3

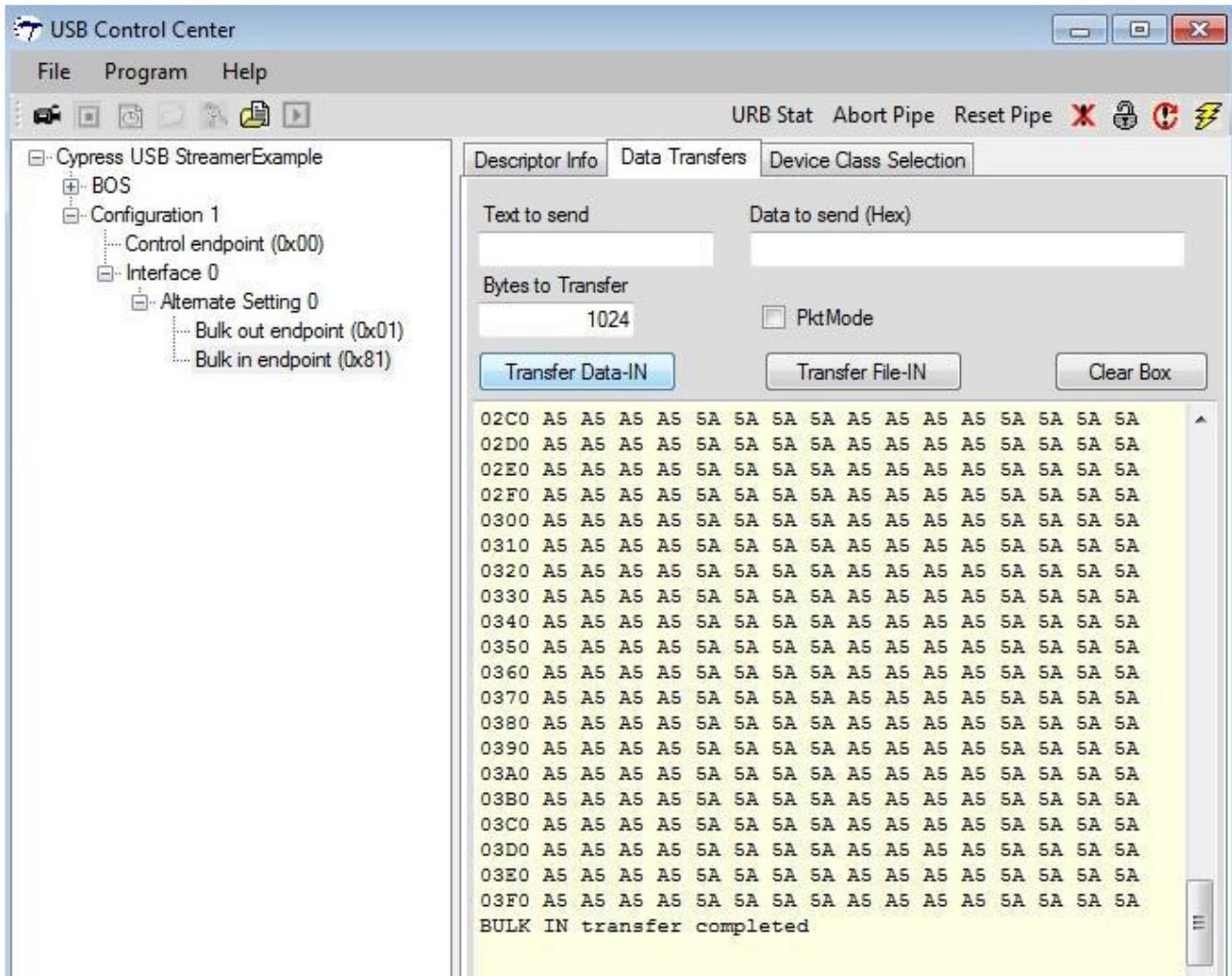


Figura 36: Captura de StreamIN mediante aplicación BulkLoop de Cypress.

4.3.3: Comunicación entre FIFO de FPGA y conversor Pmod AD5

Para poder comunicarse con el conversor de PMOD AD5, se requiere una comunicación con protocolo tipo SPI. Para esto, se debe crear un bloque dentro del FPGA que realice la configuración del conversor, reciba los datos del mismo y los envíe a la FIFO. Se usó como base el código VHDL *spi_master* de la página eeWiki, que es una página donde se comparte código abierto entre desarrolladores de distintas plataformas como (microcontroladores, FPGA, Linux, etc.). El link del código respectivo es: <https://eewiki.net/pages/viewpage.action?pageId=4096096>. En este, se describe el funcionamiento y las señales para su implementación en un FPGA. A continuación se muestra en la Figura 37 una aplicación típica de este código:

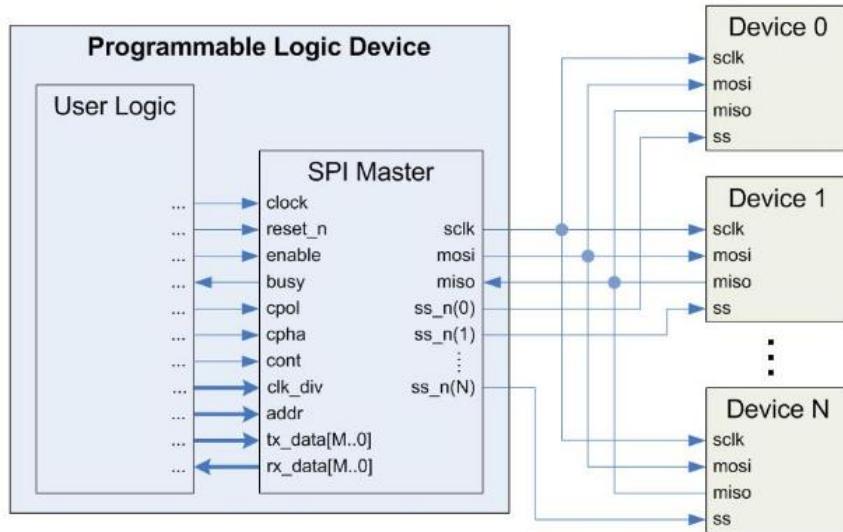


Figura 37: Aplicación típica del código *spi_master*

Para poder controlar el *SPI Master* tal cual se muestra en la Figura 37, se diseñó una unidad de control en Vivado que se denominó *CONTROL*, el cual se encarga de configurar las señales de control *SPI Master*, establecer la dirección del esclavo, detectar mediante la señal *busy* si el modulo todavía está enviando o si está disponible y finalmente, generar un pulso en la señal *enable* cada vez en se deseé enviar un dato. En la Figura 38 se muestra el diagrama de tiempos y señales para enviar datos de forma continua a través del *SPI Master*. En la Figura 39 se puede observar el diagrama de estados del bloque *CONTROL*.

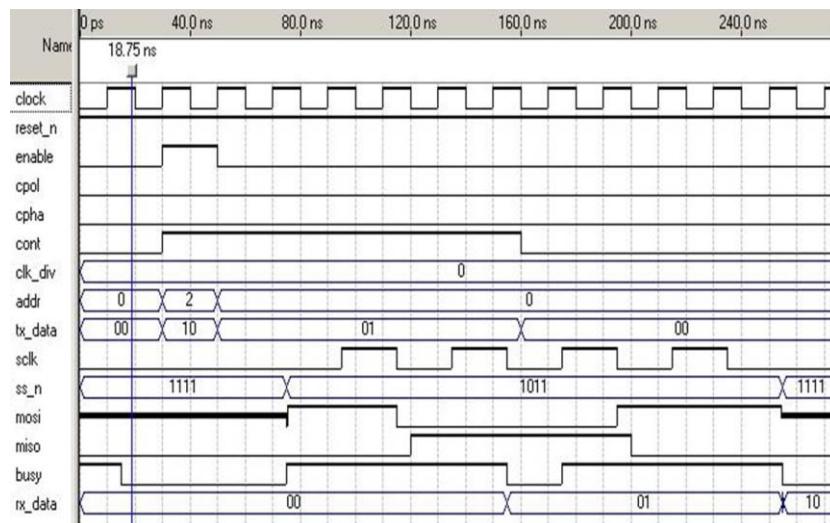


Figura 38: Diagrama de tiempos y señales de control del *SPI Master*.

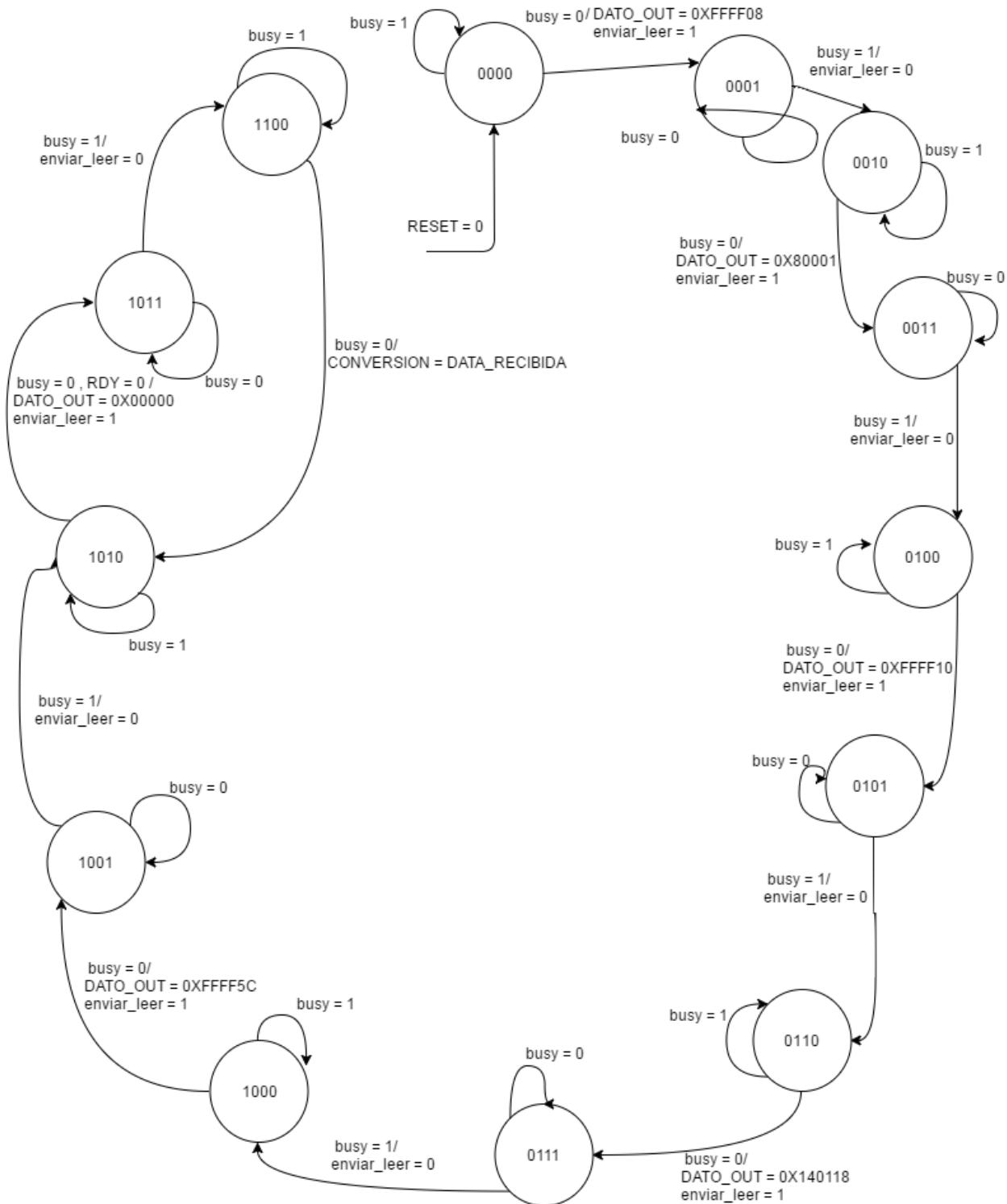


Figura 39: Diagrama de estados del bloque CONTROL

El bloque de control creado, se “empaquetó” en un IP. Un IP en el programa Viivado, permite insertar dentro de cualquier proyecto el código del bloque en cuestión en una forma más abstracta. Es una forma muy simple de reutilizar código, reduciendo tiempo de integración entre sistema, ya que permite realizar la instanciación de bloques, simplemente interconectando las señales de entradas y salidas entre dos o más IPs, gráficamente. Se muestra en la Figura 40 el bloque del código CONTROL, y también, el bloque SPI Master

en dos IPs conectados. Se puede observar que entre las señales de control que tiene CONTROL se encuentra la señal la señal *envio*, esta se implementó, porque luego se creó otro bloque llamado CONTROL_P5MOD el cual es ejecuta la configuración del conversor Pmod AD5, y cada vez detecta que la señal *busy* es ‘0’, genera una señal de envío para que CONTROL actué sobre SPI Master, y además coloca el dato necesario para enviar al conversor y obtener la conversión a través de la señal *rx_data* (24 bits del conversor). Finalmente convierte el dato de 24 bits en uno de 32 bits, agregando 8 bits nulos, en los más significativos de los 32. Esto es para que coincida con el tamaño de dato requerido por la FIFO.

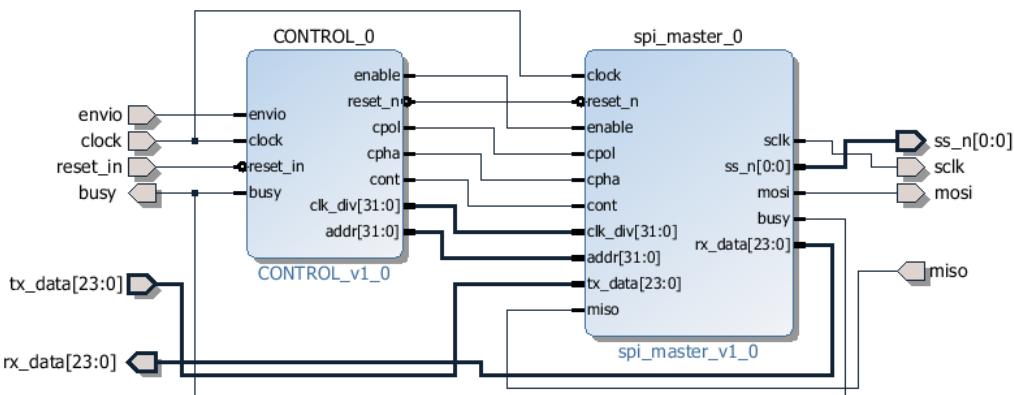


Figura 40: IPs de CONTROL y SPI Master conectados.

El bloque CONTROL_P5MOD es mostrado en la Figura 41. Se observa además un bloque llamado GATE_IN_32_OUT_8. Este, se creó para probar el funcionamiento de esta parte del sistema que se integraría con la FIFO. El mismo, toma el dato de 32 bits y lo divide en 8 compuertas OR de 4 entradas y 1 salida cada una, para que el dato sea mostrado en los 8 leds que tiene la ZedBoard. De esta forma haciendo una especie de “Boometro”, se verifico cuando el Pmod AD5 convierte datos muestrados de un potenciómetro que se colocó en la entrada para generar una señal analógica.

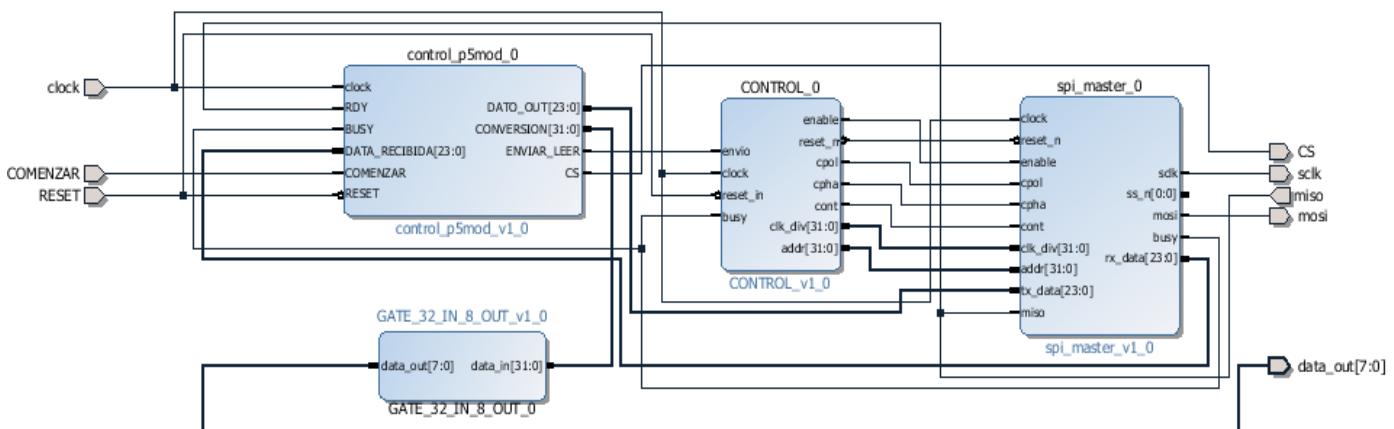


Figura 41: Bloques SPI Master - CONTROL - Control_p5mod para probar

Como se mencionó en el capítulo 1, el Pmod AD5 es un conversor ADC de 24bits con salida SPI el cual requiere ser configurado cada vez que es alimentado para poder comenzar a convertir. Entre las posibilidades de configuración, se encuentran la selección de los canales de conversión como una sola entrada, o varias de forma multiplexada. Además, se puede seleccionar diferentes ganancias de amplificación, filtrado y conversión continua o única. Los registros de configuración y opciones, se encuentran en la página del fabricante Digilent: https://reference.digilentinc.com/pmod:pmod:ad5:ref_manual, así como también la hoja de datos del integrado para descargarla desde el mismo sitio.

En un principio no se lograba obtener ninguna conversión y finalmente se descubrió que aunque en las especificaciones, indica que el reloj del SPI puede trabajar en 1Mhz, y que se configuro en un primer momento en esa velocidad, luego se disminuyó para probar a 100Khz, y de esa forma se logró que el dispositivo comenzara a convertir datos.

Una vez logrado el paso anterior, se procedió a integrar todo el sistema del FPGA. Para este fin, se formó un solo bloque llamado SPI_CONF_MASTER_CONTROL formado por los antes mencionados: CONTROL, SPI Master y Control_p5mod. En la Figura 42 se muestra todo el sistema integrado junto con la FIFO que se llamó tal cual el código que instanciaba todos sus códigos internos: “sistema_slaveFIFO2b_fpga_top”.

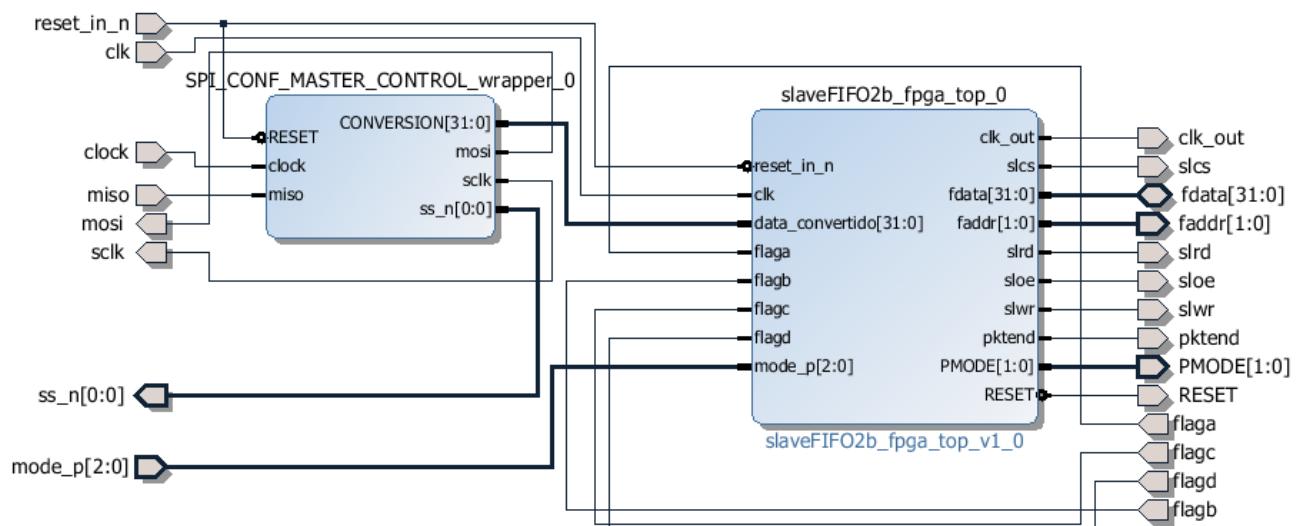


Figura 42: Sistema completo del FPGA.

Este sistema completo, se programó en la placa ZedBoard y se probó junto con todo el proyecto incluyendo el EZ USB FX3, el programa de VS2010 y MatLab. Todo el conjunto se probó con un generador de señales configurado para una señal de 100hz. Si bien el sistema transmitía los datos hasta el programa de VS2010 y se generaba una gráfica en MatLab, la misma, era una línea plana que cambiaba continuamente de nivel. En la Figura 43 y Figura 44 se observan 2 capturas de pantalla de la gráfica de la onda muestreada en MatLab.

Después de analizar el problema, se descubrió que la interface FIFO de Cypress está preparada para transmitir a 100Mhz, pero controlando la hoja de datos del conversor del Pmod AD5, se encontró que en el mismo, la frecuencia de muestreo es de 4,7Khz. Por

lo tanto, como hasta que se realiza una nueva conversión, el dato a la entrada del bloque FIFO en el FPGA no cambia, esta, envía aproximadamente 10 millones de veces el mismo dato hasta que es actualizado. Para buscar solucionar este problema, se probó modificar el reloj de la FIFO tanto en el FX3 como en el FPGA a una frecuencia más baja y cercana a la del conversor, se comenzó por 10Khz pero transmitía ningún dato y solo se veía en el *boxText Failures* del programa *TesisOsciloscopio* los paquetes fallados o perdidos. Luego se fue subiendo a 100Khz, 1Mhz y continuaba sin transmitir datos.

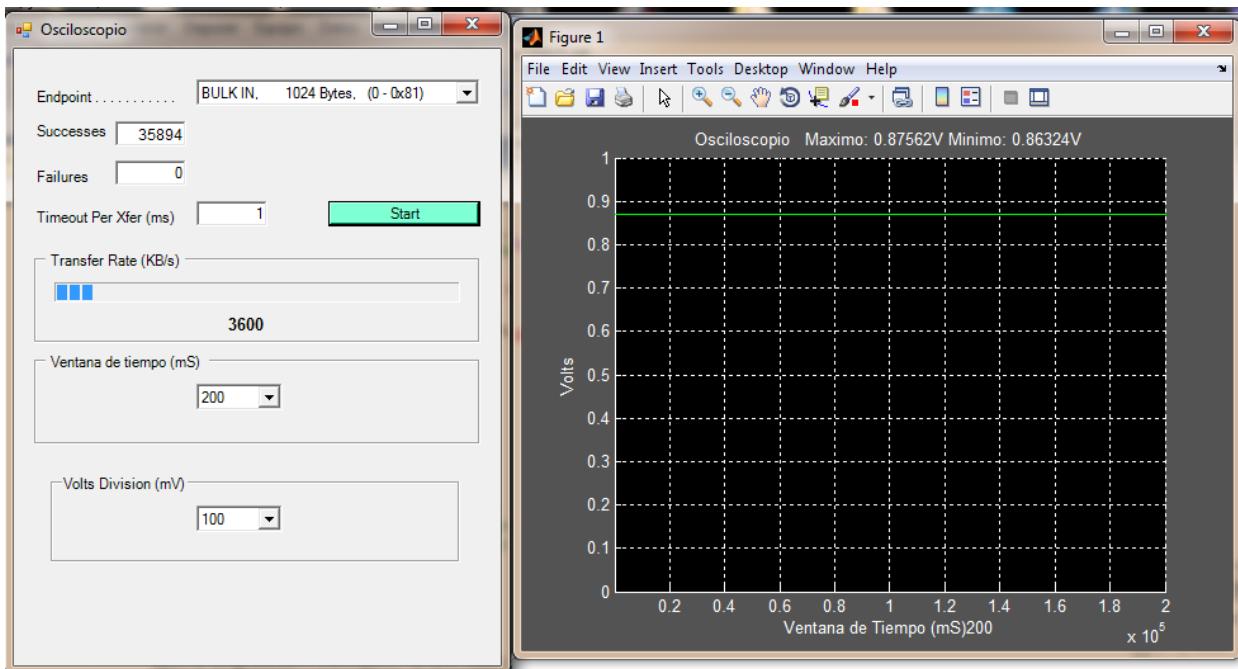


Figura 43: Grafica probando el sistema con Pmod AD5 - 1

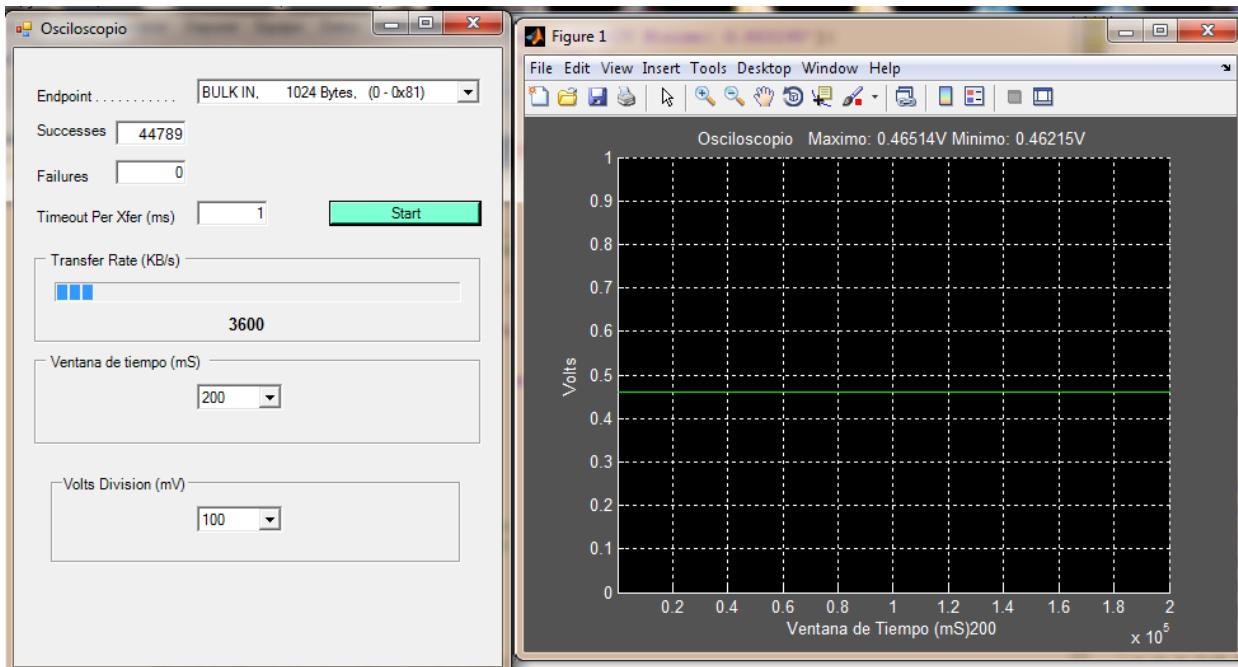


Figura 44: Grafica probando el sistema con Pmod AD5 - 2

Para salvar este problema y lograr ver la onda de forma coherente, se resolvió utilizar el conversor que tiene incorporado la placa ZedBoard que aunque es de 12 bits en vez de 24 bits como el original, la tasa de conversión puede alcanzar 1Msps, lo cual se aproxima un poco más a la velocidad de la FIFO. Se podría haber optado por buscar otro conversor pero ante la posible demora en la llegada del nuevo equipo, sumada a la modificación y adaptación que había que realizar, podía comprometer la conclusión de este proyecto a tiempo. Por lo tanto en el siguiente sub capítulo se explica la modificación realizada.

4.3.4: Modificación del código utilizando el XADC de la ZedBoard:

El conversor incorporado en el FPGA de la placa ZedBoard, es un conversor de 12 bits con tasa de conversión de 1Msps máximo. El mismo se puede instanciar dentro del programa Vivado como un IP, el cual se configura mediante una herramienta llamada “Wizard XADC”. En ella se puede configurar la velocidad de conversión, un filtro promediador, capturar secuencialmente los canales o simplemente trabajar sobre uno solo, realizar conversión continua o única, entre otras opciones. Se muestra a continuación, en la **Figura 45**, el diagrama de bloques del conversor XADC.

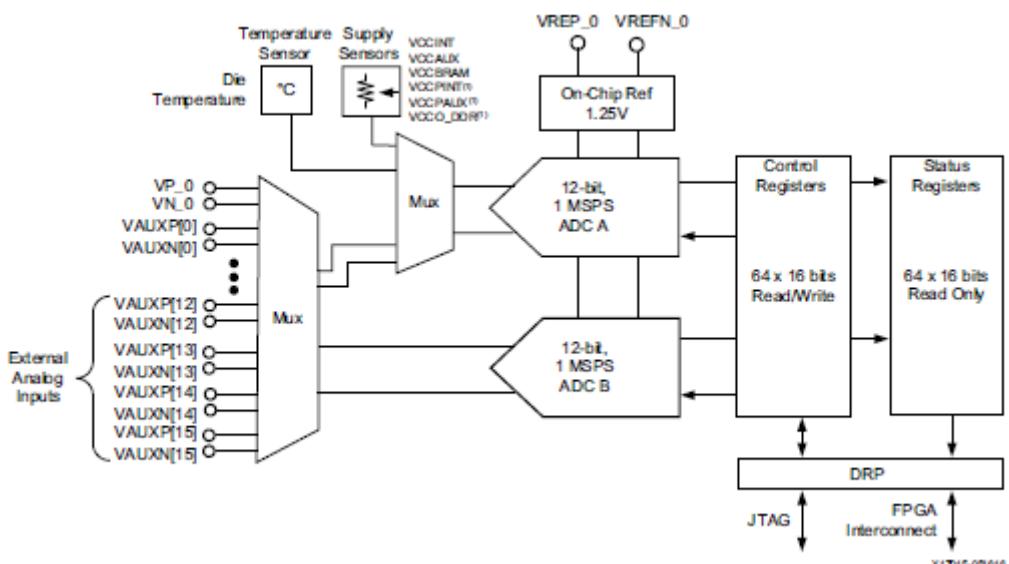


Figura 45: Diagrama de bloques del conversor XADC

En este proyecto, se configuró el conversor en modo de conversión continua, un solo canal (se usaron las entradas dedicadas por defecto V_p y V_n), tasa de conversión de 1Msps y la interface de comunicación llamada DRP (Dynamic Reconfiguration Port o Puerto de reconfiguración dinámico). Esta última opción permite comunicarse directamente con los registros del conversor, detectar cuando un dato ha sido convertido, leerlo directamente desde el registro donde se guarda el valor de conversión y colocarlo en el bus de datos. En la Figura 46 se muestra otro diagrama de bloques del conversor XADC con sus puertos en forma más específica.

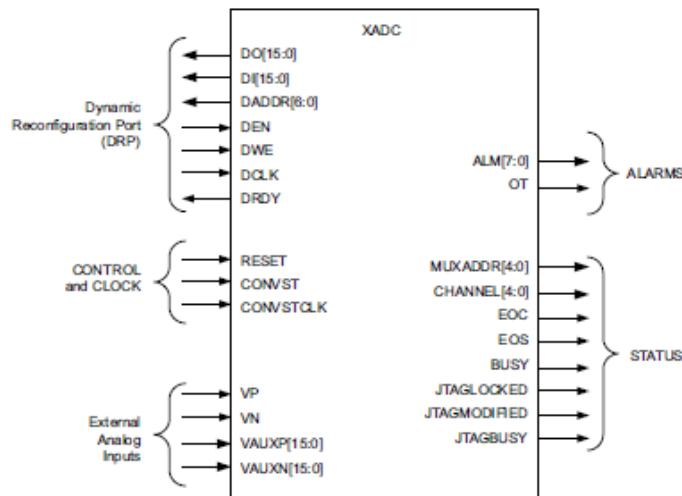


Figura 46: Puertos y diagrama de bloques del XADC

También, se muestra en las Figura 47 y Figura 48 la descripción de los puertos antes mostrados:

Port	I/O	Description
DI[15:0]	Inputs	Input data bus for the DRP. ⁽¹⁾
DO[15:0]	Outputs	Output data bus for the DRP. ⁽¹⁾
DADDR[6:0]	Input	Address bus for the DRP. ⁽¹⁾
DEN ⁽²⁾	Input	Enable signal for the DRP. ⁽¹⁾
DWE ⁽²⁾	Input	Write enable for the DRP. ⁽¹⁾
DCLK	Input	Clock input for the DRP. ⁽¹⁾
DRDY ⁽²⁾	Output	Data ready signal for the DRP. ⁽¹⁾
RESET ⁽²⁾	Input	Asynchronous reset signal for the XADC control logic. RESET will be deasserted synchronously to DCLK or the internal configuration clock when DCLK is stopped.
CONVST ⁽³⁾	Input	Convert start input. This input controls the sampling instant on the ADC(s) inputs and is only used in event mode timing (see Event-Driven Sampling, page 73). This input comes from the general-purpose interconnect in the FPGA logic.
CONVSTCLK ⁽³⁾	Input	Convert start clock input. This input is connected to a clock net. Like CONVST, this input controls the sampling instant on the ADC(s) inputs and is only used in event mode timing. This input comes from the local clock distribution network in the FPGA logic. Thus, for the best control over the sampling instant (delay and jitter), a global clock input can be used as the CONVST source.
V _P , V _N	Input	One dedicated analog input pair. The XADC has one pair of dedicated analog input pins that provide a differential analog input. When designing with the XADC feature but not using the dedicated external channel of V _P and V _N , you should connect both V _P and V _N to analog ground.
VAUXP[15:0], VAUXN[15:0]	Inputs	Sixteen auxiliary analog input pairs. In addition to the dedicated differential analog input, the XADC can access 16 differential analog inputs by configuring digital I/O as analog inputs. These inputs can also be enabled pre-configuration through the JTAG port (see DRP JTAG Interface, page 47).
ALM[0] ⁽²⁾	Output	Temperature sensor alarm output.
ALM[1] ⁽²⁾	Output	V _{CCINT} sensor alarm output.
ALM[2] ⁽²⁾	Output	V _{CCAUX} sensor alarm output.
ALM[3] ⁽²⁾	Output	V _{CCBRAM} sensor alarm output.
ALM[4] ⁽⁴⁾	Output	V _{CCPINT} sensor alarm output.
ALM[5] ⁽⁴⁾	Output	V _{CCPAUX} sensor alarm output.
ALM[6] ⁽⁴⁾	Output	V _{CCO_DDR} sensor alarm output.

Figura 47: Descripción de los puertos del XADC - 1

Port	I/O	Description
ALM[7] ⁽²⁾	Output	Logic OR of bus ALM[6:0]. Can be used to flag the occurrence of any alarm.
OT ⁽²⁾	Output	Over-Temperature alarm output.
MUXADDR[4:0]	Outputs	These outputs are used in external multiplexer mode. They indicate the address of the next channel in a sequence to be converted. They provide the channel address for an external multiplexer (see External Multiplexer Mode, page 63).
CHANNEL[4:0]	Outputs	Channel selection outputs. The ADC input MUX channel selection for the current ADC conversion is placed on these outputs at the end of an ADC conversion.
EOC ⁽²⁾	Output	End of conversion signal. This signal transitions to active-High at the end of an ADC conversion when the measurement is written to the status registers (see Chapter 5, XADC Timing).
EOS ⁽²⁾	Output	End of sequence. This signal transitions to active-High when the measurement data from the last channel in an automatic channel sequence is written to the status registers (see Chapter 5, XADC Timing).
BUSY ⁽²⁾	Output	ADC busy signal. This signal transitions High during an ADC conversion. This signal also transitions High for an extended period during an ADC or sensor calibration.
JTAGLOCKED ⁽²⁾	Output	Indicates that a DRP port lock request has been made by the JTAG interface (see DRP JTAG Interface, page 47). This signal is also used to indicate that the DRP is ready for access (when Low).
JTAGMODIFIED ⁽²⁾	Output	Used to indicate that a JTAG write to the DRP has occurred.
JTAGBUSY ⁽²⁾	Output	Used to indicate that a JTAG DRP transaction is in progress.

Figura 48: Descripción de los puertos del XADC - 2

Para usar y comunicarse con el conversor XADC, se desarrolló el código VHDL llamado CONTROL_XADC, y se incorporó, al igual que el conversor XADC como IP. En el mismo se utilizó una máquina de estados que espera por la señal *EOC*, e inmediatamente después, escribe en el bus de datos del DRP que va a leer el dato convertido. En ese momento se queda esperando a la señal *DRDY*, la cual indica que el dato solicitado se encuentra en el bus de datos. Para una mejor compresión del sistema se muestra en la Figura 49 el diagrama de estados de CONTROL_XADC y en la Figura 50 se observa el diagrama de tiempos para la conversión continua, y la lectura del dato convertido. El manual completo del XADC detalla lo aquí explicado y contiene toda la información necesaria para la configuración y utilización del mismo. Se puede obtener desde la página de Xilinx: http://www.xilinx.com/products/intellectual-property/axi_xadc.html.

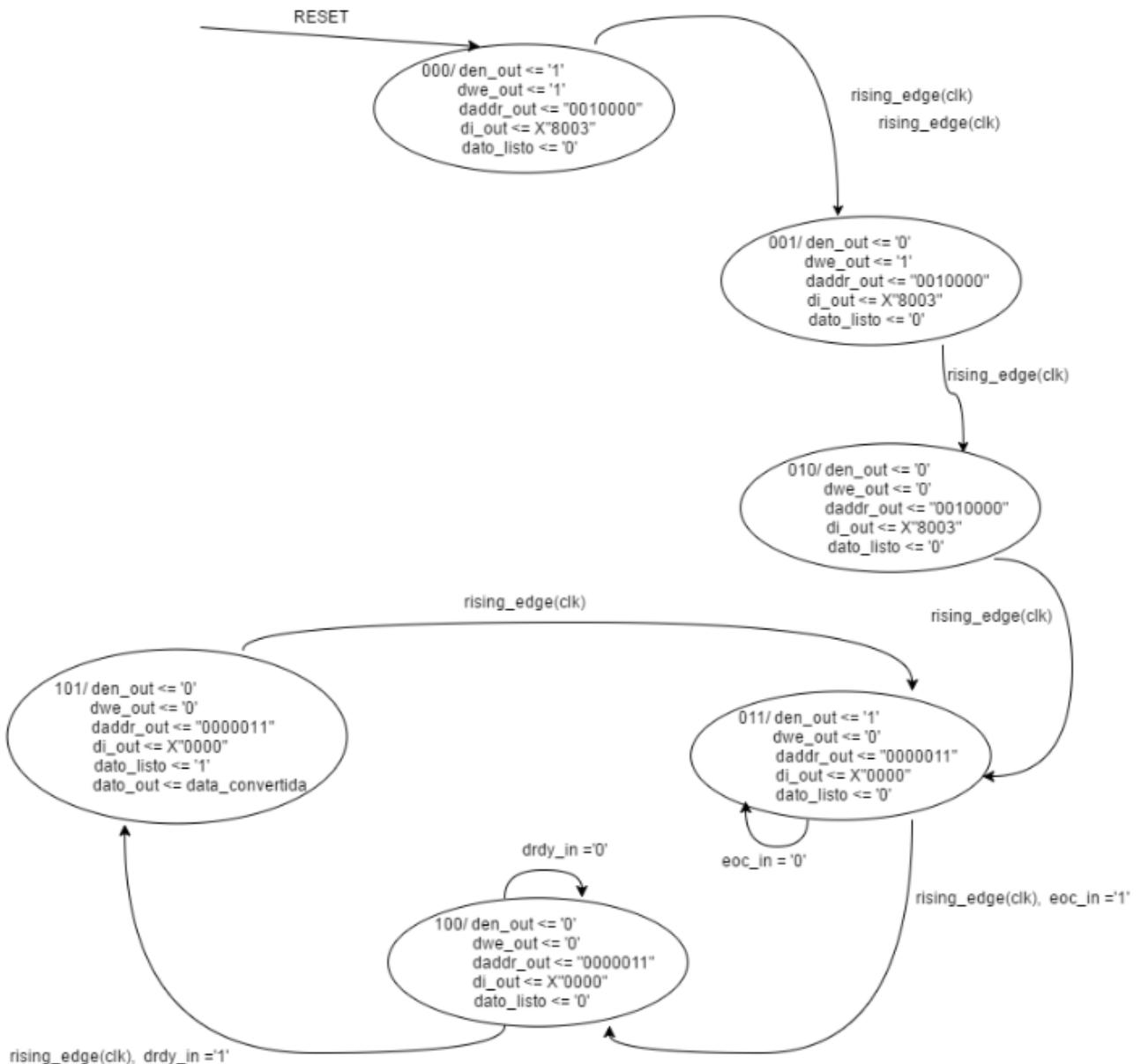


Figura 49: Diagrama de estados de Control XADC

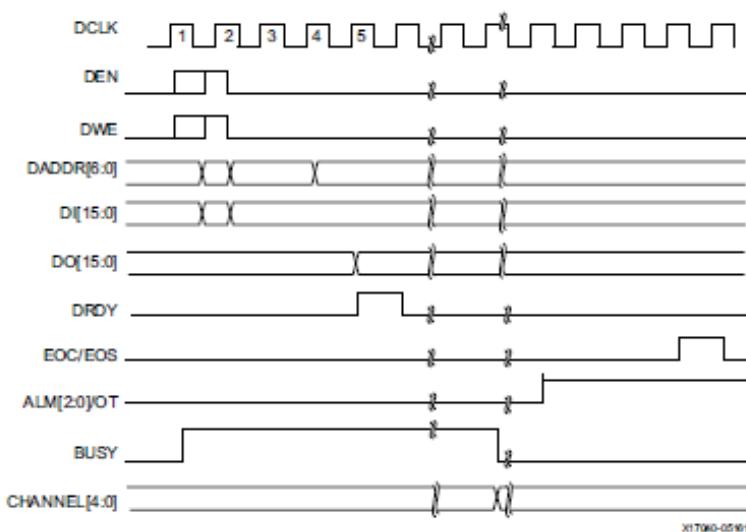


Figura 50: Diagrama de tiempos para conversión continua del XADC

Con el conversor XADC integrado junto con el sistema se obtiene el diagrama en bloques de la Figura 51. En el siguiente capítulo se muestran las pruebas del sistema y los resultados obtenidos.

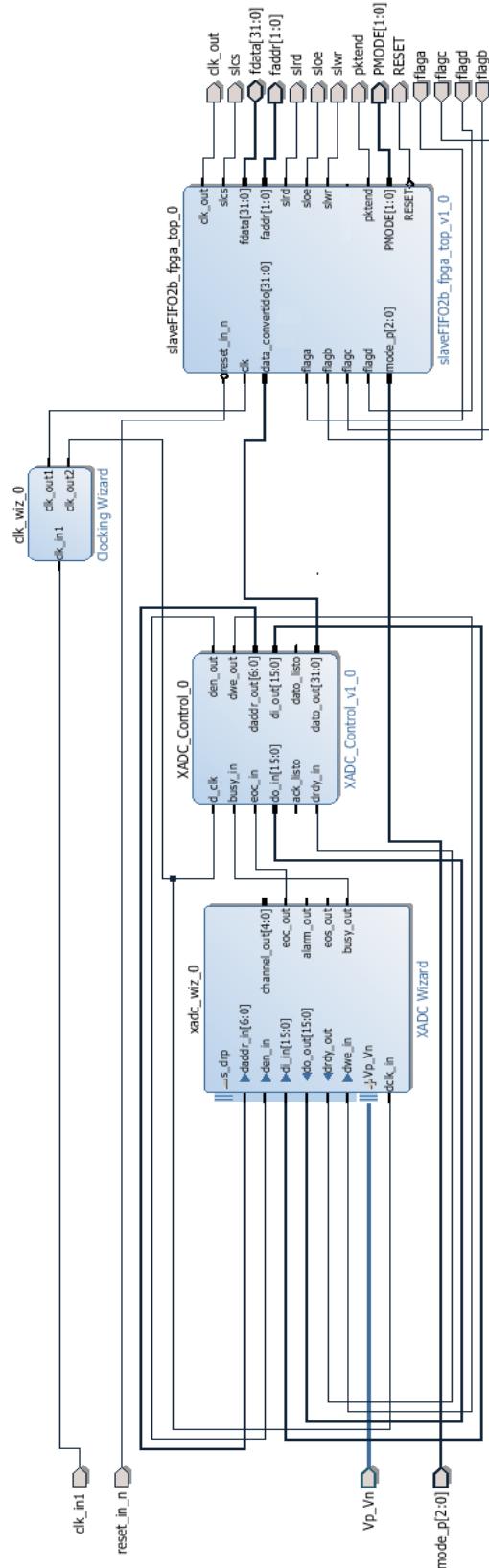


Figura 51: Sistema completo con conversor XADC

Capítulo 5: Prueba del sistema y resultados obtenidos

Una vez obtenido el sistema completo como se mostró en el capítulo anterior, se programó la placa ZedBoard con el conversor XADC. Se vinculó la placa con el kit EZ USB FX3 mediante la FMC, se conectó el cable USB 3.0 a la PC y se probó todo el sistema completo como se muestra en la Figura 52.

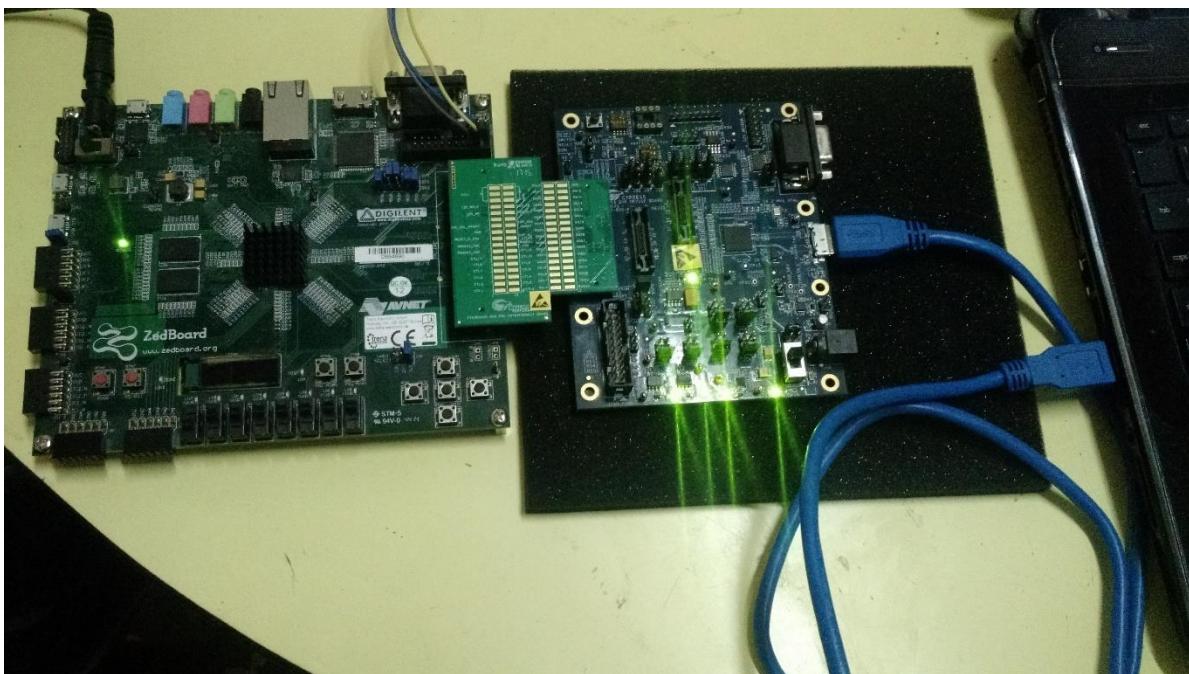


Figura 52: Sistema completo conectado

Al ejecutar el programa de VS2010 “Tesis Osciloscopio”, utilizando como entrada al conversor una onda senoidal de 100Hz de 1Vpp, se logró obtener la gráfica mostrada en la Figura 53. Modificando las selecciones de “Ventana de Tiempo” y “Volts Division” se fueron obteniendo la Figura 54 y Figura 55. En las mismas se puede observar que si bien la gráfica se obtiene bastante fiel a la onda real, existe una forma deformaciones a lo largo de la misma, y si aumentamos la frecuencia de la señal de entrada de 100Hz a 1Khz se obtiene la Figura 56. Como se observa en la misma, la forma de onda graficada se distorsiona. Buscando el problema se encontró que el conversor ADC en todos los ejemplos de Xilinx, apunta a controlar temperaturas y tensiones tanto de sus sensores internos, como de señales externas, es decir señales “lentas”. Es por eso que se asume que al aumentar la velocidad de las señales, se generan problemas de distorsión. Se intentó solucionar el problema implementando un promediador que el mismo XADC incluye pero no se obtuvo mejores resultados. Como mejora a futuro del proyecto, se recomienda cambiar el conversor por uno que pueda tener un mejor rendimiento.

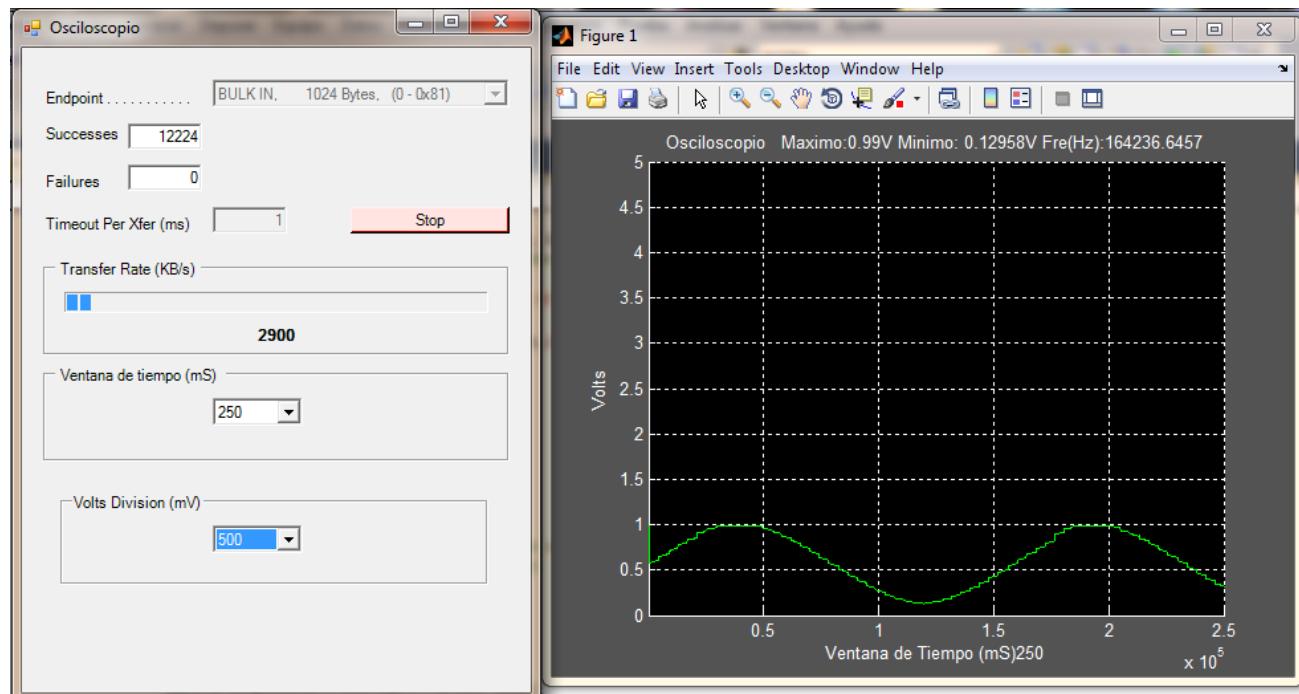


Figura 53: Grafica en tiempo real con señal de 100Hz y 500mV div

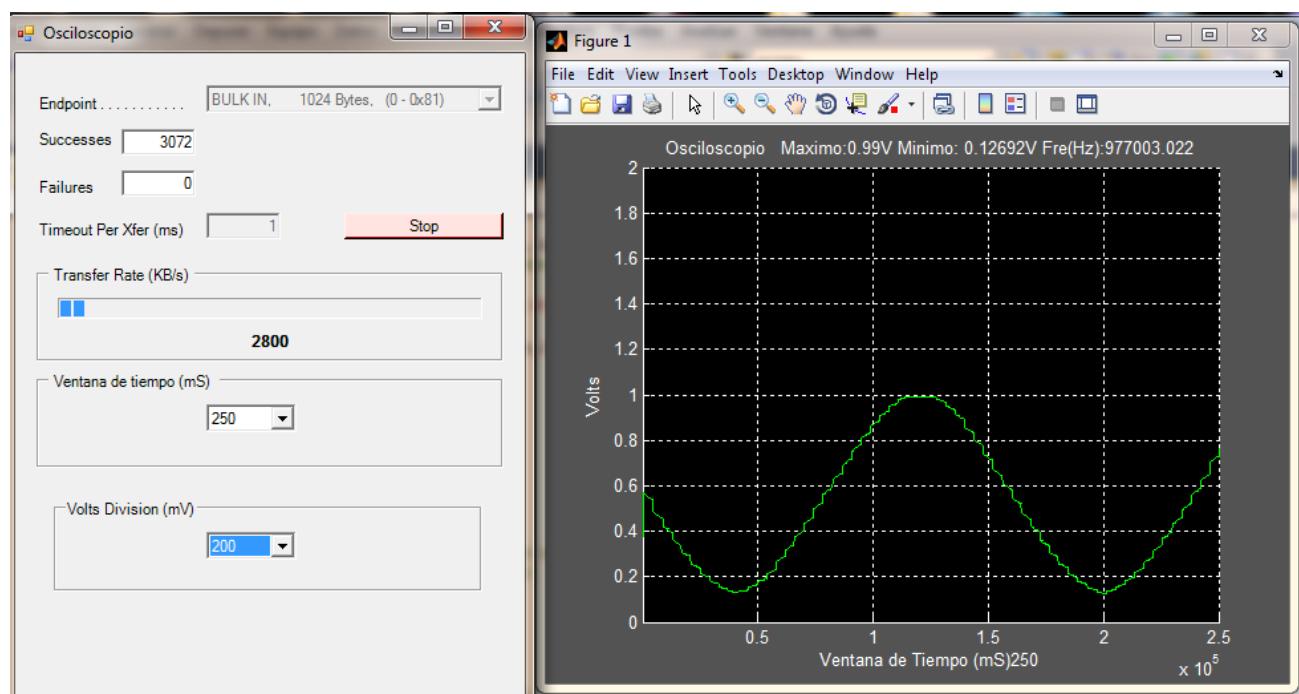


Figura 54: Grafica en tiempo real con señal de 100Hz y 200mV div

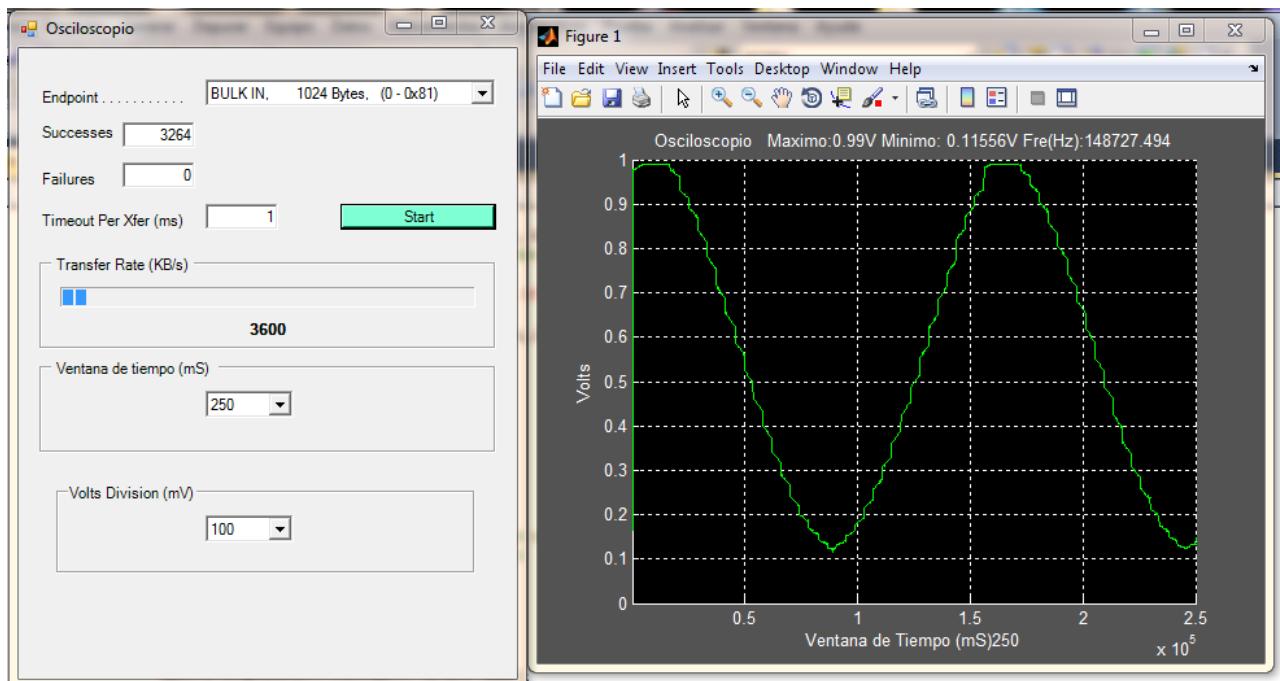


Figura 55: Grafica en tiempo real con señal de 100Hz y 100mV div

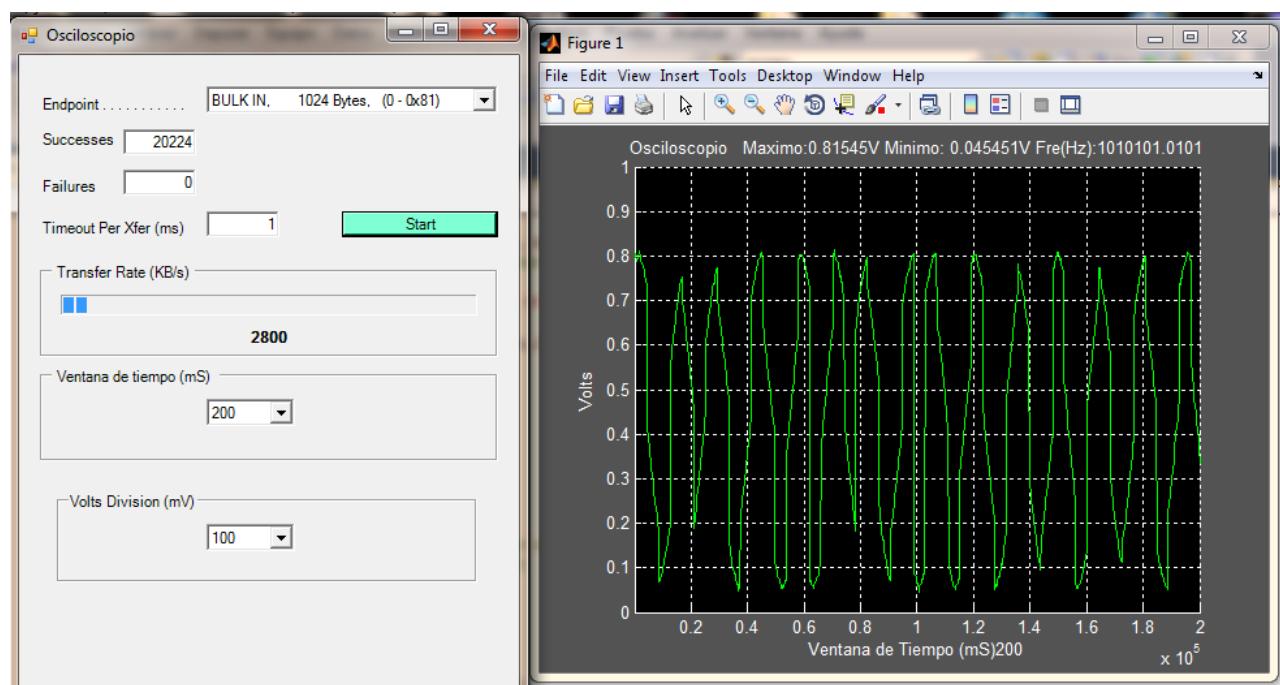


Figura 56: Grafica en tiempo real con señal de 1000Hz y 100mV div

Conclusiones.

A lo largo de este proyecto se encontraron varios desafíos que pudieron resolverse de forma exitosa. Entre ellos, y el principal, particularmente para este tesista, fue el encontrar la forma de graficar en tiempo real los datos que iban a recibirse a través del puerto USB.

Como se trabajaba sobre el Kit EZ USB FX3 el cual brinda la librería CyAPI.lib, se sabía que se podían recibir los datos desde el FX3 utilizando las funciones de la misma. Graficar esos datos era distinto, ya que implementar gráficas a través de Visual Studio 2010 es bastante complejo. Se recurrió a buscar otras alternativas antes de encarar el proyecto para poder definir las tareas del mismo. Se encontró la alternativa de enviar los datos desde VS2010 a MatLab a través del motor Engine del mismo, y con esa solución se encaró el proyecto. Luego, al comenzar a conocer esta herramienta, se descubrió que la misma es potente ya que permite utilizar casi cualquier opción de MatLab pero ejecutándola desde VS2010. Un problema que se encontró al implementar el motor Engine en Visual C++ fue el espacio de nombres, esto en un principio, se vio como una gran dificultad, pero luego consultando la página de MatLab y sus foros, se encontró la solución de forma relativamente simple como se describió en el capítulo 3.

Otro de los problemas que se planteó ya iniciado el proyecto fue el de la velocidad de conversión del Pmod AD5. Afortunadamente, la placa ZedBoard contaba con un conversor incorporado, que aunque no tenía la misma resolución que el primero, podía convertir a velocidades muy superiores. Este problema se tuvo que resolver de forma rápida para poder terminar a tiempo la tesis. Si bien se logró cumplir con los plazos, no se consiguió obtener la mejor respuesta del XADC debido a que presenta problemas para ciertas señales y velocidades. Sería muy interesante como mejora de este proyecto cambiar el conversor y colocar uno dedicado que pueda solucionar los problemas antes mencionados.

Antes de comenzar la tesis y conocer los elementos con los cuales se iba a trabajar, el uso de la comunicación USB parecía un desafío muy grande. Esto se debe a que cuando se necesita utilizar comunicación de un circuito electrónico con una PC, se recurre normalmente a conversores Serie-USB debido a que la comunicación serie, es mucho más fácil de implementar. Una vez que se comenzó a estudiar el Kit del FX3 se encontró que los fabricantes de los dispositivos desarrollan herramientas como la librería CyAPI, las cuales fomentan y simplifican el uso de la comunicación USB. Hoy en día la implementación de un sistema que se comunique de esta forma, es muy sencilla ya que las soluciones de los fabricantes hacen que todo el proceso complejo de comunicación con los drivers de los dispositivos sea totalmente transparente, y simplemente mediante el uso de funciones se puede conseguir transmitir datos con velocidades muy superiores a las obtenidas con una comunicación serie.

Como resultado de esta tesis, se tiene un sistema de adquisición de señales en tiempo real de muy buen ancho de banda, con características propias tales como la velocidad de la comunicación USB 3.0. Está integrado con un FPGA con toda la potencialidad que un sistema de esta tecnología comprende: abarcando desde la velocidad de trabajo hasta la adaptación a cualquier sistema de conversión de señales con múltiples entradas y tipos de comunicaciones mediante la reconfiguración de hardware; incluyendo la alta capacidad de procesamiento de MatLab en conjunto con la facilidad para adquirir datos que otorga la librería CyAPI.lib para Visual Studio 2010. Todas estas características, permiten que este proyecto pueda utilizarse en diversos entornos, principalmente, en el sistema de mesa vibratoria que posee el IDIA, así como también, en cualquier sistema en el cual se necesite adquirir señales en tiempo real con alta velocidad de transferencia.

Bibliografía

- “VHDL FOR LOGIC SYNTHESIS”. Andrew Rushton. John Wiley & Sons, Third Edition, 2011.
- “The Designer’s Guide to VHDL”. P. Ashenden. Morgan Kaufman, Third Edition. 2008.
- “FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version”. Pong Chu. Wiley, Interscience, 2008.
- “Advanced FPGA Design Architecture, Implementation and Optimization”. Steve Kilts; John Wiley & Sons Inc., 2007.
- “RTL HARDWARE DESIGN USING VHDL, Coding for Efficiency, Portability, and Scalability”. Pong Chu; John Wiley & Sons Inc., 2006.
- “Digital Design. Principles and Practices”. J. Wakerly. Prentice Hall. 2004.
- “Manual básico de MatLab” - Mª Cristina Casado Fernández. Disponible en http://www.sisoft.ucm.es/Manuales/MATLAB_r2006b.pdf
- “Introduction to Matlab Engine - Getting the best out of C++ and Matlab.” – Jai Pillai. Disponible en <http://www.umiacs.umd.edu/~jsp/Downloads/MatlabEngine/MatlabEngine.pdf>

Recursos Web

- http://www.mathworks.com/help/matlab/matlab_external/introducing-matlab-engine.html
- https://reference.digilentinc.com/pmod:pmod:ad5:ref_manual#pinout_description_table
- <https://eewiki.net/pages/viewpage.action?pageId=4096096>
- <http://www.cypress.com/products/ez-usb-fx3-superspeed-usb-30-peripheral-controller>
- <http://www.cypress.com/documentation/application-notes/an65974-designing-ez-usb-fx3-slave-fifo-interface>
- http://www.xilinx.com/support/documentation/user_guides/ug480_7Series_XADC.pdf