TRABAJO ESPECIAL DE GRADO

DISEÑO DE UN SISTEMA DE COMUNICACIÓN ENTRE UN COMPUTADOR PERSONAL Y UN NANOBOARD 3000 DE MODO ISÓCRONO CON BASE EN EL USB

Presentado ante la Ilustre
Universidad Central de Venezuela
Por el Br. Borges F., Gabriel A.
Para optar al Título de
Ingeniero Electricista

Caracas, 2017

TRABAJO ESPECIAL DE GRADO

DISEÑO DE UN SISTEMA DE COMUNICACIÓN ENTRE UN COMPUTADOR PERSONAL Y UN NANOBOARD 3000 DE MODO ISÓCRONO CON BASE EN EL USB

Profesor Guía: Prof. Ebert Brea
Tutor Industrial: Ing. Carlelines Gavidia

Presentado ante la Ilustre
Universidad Central de Venezuela
Por el Br. Borges F., Gabriel A.
Para optar al Título de
Ingeniero Electricista

Caracas, 2017

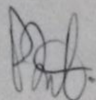
CONSTANCIA DE APROBACIÓN

Caracas, 08 de noviembre de 2017

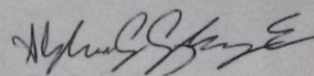
Los abajo firmantes, miembros del Jurado designado por el Consejo de Escuela de Ingeniería Eléctrica, para evaluar el Trabajo Especial de Grado presentado por el Br. Gabriel A. Borges F., titulado:

"DISEÑO DE UN SISTEMA DE COMUNICACIÓN ENTRE UN COMPUTADOR PERSONAL Y UN NANOBOARD 3000 DE MODO ISOCRONO CON BASE EN EL USB"

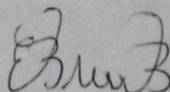
Consideran que el mismo cumple con los requisitos exigidos por el plan de estudios conducente al Título de Ingeniero Electricista en la opción de Electrónica y Control, y sin que ello signifique que se hacen solidarios con las ideas expuestas por el autor, lo declaran APROBADO.



Prof Panayotis Tremante
Jurado



Prof. Alejandro González
Jurado



Prof. Ebert Brea
Prof. Guía

A Dios, mi madre, mis hermanas y mi futura esposa.

RECONOCIMIENTOS Y AGRADECIMIENTOS

Primeramente a Dios, por regalarme cada instante y permitirme disfrutar del milagro de la vida.

A mi familia, en especial a mi Mamá por traerme al mundo, por su cariño y por inculcarme excelentes valores que me han acompañado a lo largo de este arduo camino.

A mis hermanas Gabriela y Michelle por llenar cada momento de alegría, locuras y mucho cariño.

A Adriana Amaral, por brindarme su compañía, su amor, un apoyo incondicional y valiosos consejos aún estando separados miles de kilómetros. No lo hubiese logrado sin ti.

A los profesores que con sus consejos y conocimientos brindados me ayudaron a expandir mis horizontes y explotar adecuadamente mis habilidades, especial mención al profesor Servando Álvarez por brindarme su ayuda en los momentos más difíciles y al profesor Ebert Brea por brindarme su apoyo en la realización de este trabajo.

A mis amigos y compañeros de estudio que más de una vez me enseñaron el valor de una verdadera amistad y con los que pasé tan buenos momentos, sólo ustedes entienden perfectamente las dificultades a las que nos enfrentamos al recorrer este camino. Especial mención a Víctor, Edgar, Alexandra, Rubén, Kati, Arroyo, Cen, Paola, Anderson, Claris y Félix.

A todo el personal del Cendit por permitirme realizar el Trabajo de Grado en sus instalaciones y especialmente a Litza Pinto y Carlelines Gavidia por brindarme su apoyo y asesoría en la realización de este trabajo.

RESUMEN

Gabriel Adolfo Borges Figuera

DISEÑO DE UN SISTEMA DE COMUNICACIÓN ENTRE UN COMPUTADOR PERSONAL Y UN NANOBOARD 3000 DE MODO ISÓCRONO CON BASE EN EL USB.

Tutor Industrial: Ing. Carlelines Gavidia. Profesor Guía: Prof. Ebert Brea. Tesis. Caracas. Universidad Central de Venezuela. Facultad de Ingeniería. Escuela de Ingeniería Eléctrica. Mención: Electrónica. Año 2017.

Palabras Clave: USB, FPGA, VHDL, Altium, NanoBoard 3000, C#, Visual Studio, Cypress, FX2LP, Televisión Digital Abierta.

Resumen.- En el presente trabajo se diseña un sistema de comunicación mediante el USB entre un computador personal y una NanoBoard 3000 para proveer una interfaz de salida al Modulador de baja potencia para Televisión Digital Abierta implementado en software bajo el estándar ISDB-Tb desarrollado actualmente en el Cendit. Una aplicación de control de datos diseñada en Visual Studio y escrita en lenguaje C# se encarga de establecer la conexión USB entre el computador personal y el dispositivo Cypress que recibirá los datos, además prepara y envía los datos de modo isócrono a dicho dispositivo. Para el correcto funcionamiento del dispositivo Cypress se diseñó un firmware en lenguaje C y una tabla de descriptores en *assembler* que permiten al computador identificar al dispositivo y configurar la transmisión de datos USB de modo isócrono a la mayor velocidad posible. Finalmente los datos son recibidos en los puertos de entrada de la NanoBoard 3000 y luego son llevados a los puertos de salida, esto es posible mediante el diseño de un *core* FPGA en Altium que tiene como elemento central el *soft processor* TSK3000.

ÍNDICE GENERAL

CONSTANCIA DE APROBACIÓN ------------------------ ERROR! BOOKMARK NOT DEFINED.

RECONOCIMIENTOS Y AGRADECIMIENTOS -- V

RESUMEN -- VI

ÍNDICE GENERAL -- VII

LISTA DE FIGURAS -- IX

LISTA DE TABLAS -- XI

LISTA DE ECUACIONES -- XII

LISTA DE ACRÓNIMOS -- XIII

INTRODUCCIÓN --- 1

CAPÍTULO I --- 2

PLANTEAMIENTO DEL PROBLEMA --- 2
OBJETIVOS --- 3
1.2.1. Objetivo General --- 3
1.2.2. Objetivos Específicos --- 3

CAPÍTULO II -- 4

MARCO TEÓRICO -- 4
2.1. Televisión Digital Terrestre -- 4
2.2. USB --- 6
2.2.1. Componentes del bus --- 6
2.2.2. *Endpoints* --- 8
2.2.3. Pipes -- 8
2.2.4. Transferencias de datos --- 9
2.2.5. Detalles de las capas de protocolos del USB -- 13
2.2.6. Enumeración del dispositivo esclavo --- 17
2.2.7. Descriptores --- 19
2.2.8. Solicitudes del *Host*. --- 25
2.2.9. EZ-USB® FX2LP™ USB *Microcontroller High-Speed* USB Peripheral *Controller*. --------- 26
2.2.10. EZ-USB® SX2™ *High-Speed* USB *Interface Device*. ------------------------------ 31
2.2.11. Hardware USB -- 32
2.3. FPGA y Tarjeta de desarrollo NanoBoard 3000AL de Altium ------------------------ 33
2.3.1. Sistema Controlador NanoTalk -- 35
2.3.2. Altium Software DXP -- 36
2.3.3. Sistemas embebidos --- 37
2.3.4. Microprocesador Embebido TSK3000 --- 38
2.3.5 *Device View* --- 39
2.3.5. VHDL --- 41
2.3.6. Active-HDL -- 42
2.4. Visual Studio -- 43

2.4.1. .NET Framework ---------- 43
2.4.2. Visual C# .NET ---------- 45

CAPÍTULO III ---------- 46

MARCO METODOLÓGICO ---------- 46
3.1. Estudio de los protocolos y fundamentos teóricos ---------- 46
3.2. Estudio de las herramientas computacionales y dispositivos disponibles para implementar la interfaz. ---------- 47
3.2.1. Computador Personal tipo Laptop ---------- 47
3.2.2. Estudio de VHDL y Active HDL ---------- 48
3.2.3. Estudio de las herramientas disponibles en Altium DXP y la NanoBoard 3000 ---------- 48
3.2.4. Estudio de la tarjeta de desarrollo Cypress CY7C68013A EZ-USB FX2LP ---------- 49
3.2.5. Estudio de C#, librería CyUSB.dll y Visual Studio 2013 ---------- 50
3.3. Esquema de diseño para realizar la comunicación Computador Personal -NanoBoard 3000 ---------- 50
3.4. Pruebas de funcionamiento ---------- 55
3.5.1. Pruebas realizadas a la aplicación de control de transferencias ---------- 55
3.5.2. Pruebas realizadas a la tarjeta EZ-USB FX2LP ---------- 56
3.5.3. Pruebas realizadas a la NanoBoard 3000. ---------- 56

CAPÍTULO IV ---------- 57

DESARROLLO DE LA INTERFAZ ---------- 57
4.1. Etapa 1: Aplicación para control de datos en el computador personal ---------- 57
4.1.1. Interfaz Gráfica ---------- 58
4.1.2. Código de la Aplicación ---------- 61
4.2. Interfaz USB implementada en la placa de desarrollo EZ-USB FX2LP ---------- 70
4.2.1. Tabla de Descriptores ---------- 71
4.2.3. Manejo de solicitudes de control ---------- 74
4.3. Tarjeta de desarrollo NanoBoard 3000 ---------- 78
4.3.1. *OpenBus* ---------- 78
4.3.2. *Schematic* ---------- 86
4.3.3. Proyecto Embebido ---------- 92
4.3.3.1 *Software Platform* ---------- 92
4.3.3.2 Código Fuente ---------- 94
4.3.4 Proceso de lectura de datos en el FIFO esclavo de modo asíncrono ---------- 94

CAPÍTULO V ---------- 98

ANÁLISIS DE RESULTADOS ---------- 98
5.1 Aplicación de control de datos ---------- 98
5.2 *Firmware* de la tarjeta de desarrollo FX2LP ---------- 103
5.3 *Core* FPGA NanoBoard 3000 ---------- 107

CONCLUSIONES ---------- 118

RECOMENDACIONES ---------- 119

REFERENCIAS BIBLIOGRÁFICAS ---------- 120

GLOSARIO ---------- 123

ANEXOS ---------- ERROR! BOOKMARK NOT DEFINED.

LISTA DE FIGURAS

Figura 2.1: Trasnmisión jerárquica en tres capas [1]. 5
Figura 2.2: Sistema de transmisión ISDB-Tb [1]... 5
Figura 2.3: Topología típica del USB [4]. 7
Figura 2.4: Conexión de *endpoints* y búferes mediante *pipes* [4]................................. 9
Figura 2.5: Transacciones contenidas en los *frames* y *microframes* [4]. 10
Figura 2.6: Componentes de las transferencias USB [2]. 13
Figura 2.7: Diagrama de bloques del FX2LP [8]. 26
Figura 2.8: Bloques del FX2LP que permiten compartir datos entre *host* y el dispositivo esclavo mediante Slave FIFO [8]. 28
Figura 2.9: Bloques del FX2LP que permiten compartir datos entre *host* y el dispositivo esclavo mediante el GPIF [8]. 28
Figura 2.10: Pines del FX2LP de acuerdo al tamaño del encapsulado [8]. 29
Figura 2.11: Placa de desarrollo FX2LP [12]. 30
Figura 2.12: Diagrama de bloques del SX2 [13].. 31
Figura 2.13.1: Conectores USB tipo A y B [11]. 32
Figura 2.13.2: Pines del conector USB tipo C [11]... 33
Figura 2.14: Módulos de la NanoBoard 3000 [11]. 34
Figura 2.15: NanoBoard 3000AL [14]. 35
Figura 2.16: Esquema de comunicación JTAG de la NanoBoard 3000 [14]. 36
Figura 2.17: Diagrama de bloques del microprocesador TSK3000 [16]. 38
Figura 2.18: Ventana *Device View* de Altium. 40
Figura 2.19: Flujo de diseño en FPGA [19]. 42
Figura 3.1: Esquema de Diseño. 51
Figura 3.2: Pines disponibles en el FX2LP. 52
Figura 3.3: Numeración de los user headers disponibles en la NanoBoard 3000. 53
Figura 3.4: Esquema de conexiones entre el bus FIFO del FX2LP y los user headers de la NanoBoard3000. 54
Figura 3.5: LEDs conectados a los pines de salida de la NanoBoard 3000. 55
Figura 4.1.1: Propiedades de la referencia CyUSB en Visual Studio. 58
Figura 4.1.2: Capturas de la aplicación sin dispositivo conectado. 59
Figura 5.1.1: Aplicación control datos 1. 99
Figura 5.1.2: Aplicación control datos 2. 99
Figura 5.1.3: Aplicación control datos 3. 99
Figura 5.1.4: Aplicación control datos 4. 99

Figura 5.1.5: Aplicación control datos 5. 101
Figura 5.1.6: Aplicación control datos 6. 101
Figura 5.1.7: Aplicación control datos 7. 101
Figura 5.1.8: Aplicación control datos 8. 101
Figura 5.1.9: Aplicación control datos 9. 102
Figura 5.1.10: Aplicación control datos. 10. 102
Figura 5.2.1: Cargando archivo .hex del firmware. 104
Figura 5.2.2: Información del dispositivo. 104
Figura 5.2.3: Conexión computador-placa FX2LP a través de cable USB. 105
Figura 5.2.4: Conexiones FIFO necesarias para la recepción de datos. 106
Figura 5.2.5: Datos FIFO recibidos. 106
Figura 5.3.1: Conexiones FX2LP – NanoBoard 3000. 107
Figura 5.3.2: Montaje PC - FX2LP - NanoBoard 3000. 108
Figura 5.3.3: Resultado de la compilación. 109
Figura 5.3.4: Resultado de la síntesis. 109
Figura 5.3.5: Resultado de la construcción. 109
Figura 5.3.6: Resumen del proceso de construcción. 110
Figura 5.3.7: Página de vista de dispositivo con la FPGA ya programada. 111
Figura 5.3.8: Instrumento controlador de frecuencia. 111
Figura 5.3.9: Interfaz del analizador lógico. 111
Figura 5.3.10: Ventana de configuración del analizador lógico. 112
Figura 5.3.11: Señales de salida a través del *user header* B. 113
Figura 5.3.12: Zoom de señales de salida del *user header* B. 114
Figura 5.3.13: Tiempo entre flancos de subida de la señales SLOE y SLRD. 115
Figura 5.3.14: Tiempo entre flanco de subida y bajada de la señales SLOE y SLRD. 115

LISTA DE TABLAS

Tabla 2.1: Velocidad de transferencia de datos de las especificaciones USB [4]......... 7
Tabla 2.2: Tamaño máximo de carga útil para cada tipo de transferencia [2]. 11
Tabla 2.3: Comparativa de tipos de transferencias USB [2]. 12
Tabla 2.4: Paquetes de datos según la cantidad de transacciones en transferencias *IN* de tipo Isócrono [2]. 14
Tabla 2.5: Paquetes de datos según la cantidad de transacciones en transferencias *OUT* de tipo Isócrono [2]. 14
Tabla 2.6: Tipos de paquetes en las transacciones USB [2]. 16
Tabla 2.7: Campo bDescriptorType [2]. 20
Tabla 2.8: Estructura del *device descriptor*.. 21
Tabla 2.9: Valores específicos que puede tomar el campo bDeviceClass del *device descriptor*. 21
Tabla 2.10: Estructura del *device qualifier descriptor*. 22
Tabla 2.11: Estructura del *configuration descriptor*. 22
Tabla 2.12: Estructura del *interface descriptor*... 23
Tabla 2.13: El campo bInterfaceClass indica el tipo de dispositivo. 23
Tabla 2.14: Estructura del *endpoint descriptor*. 24
Tabla 2.15: Estructura del *string descriptor*. 24
Tabla 2.16: Solicitudes de control definidas por el estándar USB [2]. 25
Tabla 3.1: Pines FIFO. 52
Tabla 4.1.1: Funciones de la aplicación de control de datos. 62
Tabla 4.2.1: Funciones presentes en el archivo CYStream.c. 72
Tabla 4.2.2: Funciones presentes en el archivo fw.c.. 75
Tabla 4.3.1: Puertos creados en el Port IO. 84
Tabla 4.3.2: Conexiones entre el FX2LP y la Nanoboard 3000. 95
Tabla 5.1.1: Resultados de las pruebas de transferencia de datos. 102
Tabla 5.3.1: Bytes recibidos visibles en el analizador lógico. 112
Tabla 5.3.2: Resultado de las mediciones con el analizador lógico y el osciloscopio. 116
Tabla 5.3.3: Velocidades de transmisión con 8, 16 y 32 bits. 116

LISTA DE ECUACIONES

Ecuación 1: Frecuencia en función del período. 113

Ecuación 2: Tasa de transferencia de bits. 116

LISTA DE ACRÓNIMOS

ASIC: Application Specific Integrated Circuit.

CRC: Cyclic Redundancy Check.

DSP: Digital Signal Processor.

FIFO: First In, First Out.

FPGA: Field Programmable Gate Array.

GPIF: General Programmable Interface.

GPIO: General Purpose Input Output.

ISDB-T: Integrated Services for Digital Broadcasting – Terrestrial.

ISDB-Tb: Integrated Services for Digital Broadcasting – Terrestrial Built-in.

OFDM: Orthogonal Frequency Division Multiplexing.

PHY: Physical Bus Interface.

PID: Packet Identifier.

RISC: Reduced Instruction Set Computer.

SIE: Serial Interface Engine.

TDA: Televisión Digital Abierta.

TVD-T: Televisión Digital Terrestre.

USB: Universal Serial Bus.

VHDL: Very High speed integrated circuits Hardware description Language.

INTRODUCCIÓN

El estándar ISDB-T se desarrolló en Japón y luego fue mejorado en Brasil adoptando el nombre de ISDB-Tb (ISDB-T *Built-in*) o también llamado ISDB-T internacional, dicho estándar ya ha sido aceptado en Argentina, Bolivia, Chile, Costa Rica, Ecuador, Filipinas, Nicaragua, Paraguay, Perú, Uruguay y Venezuela.

El desarrollo de la televisión digital abierta (TDA) en Venezuela se inició en el año 2009 cuando se adoptó oficialmente el estándar ISDB-Tb a partir de ese momento, el Ministerio del Poder Popular para la Comunicación y la Información, la Fundación Centro Nacional de Desarrollo e Investigación en Telecomunicaciones (CENDIT), la Comisión Nacional de Telecomunicaciones (CONATEL), y algunas universidades venezolanas han trabajado en conjunto para capacitar personal y desarrollar tecnologías que sustenten la implementación de la TDA en el territorio nacional.

La Fundación CENDIT contribuye en el diseño y construcción del transmisor de baja potencia trabajando en el desarrollo del Circulador y Sistema Radiante (Dirección de Propagación y Antenas) y del Modulador/Excitador (Dirección de Electrónica de Comunicaciones).

El presente trabajo constituye la interfaz de salida de un prototipo industrializable del modulador de baja potencia para TDA implementado en software bajo el estándar ISDB-Tb. Esto se realiza con el fin de comunicar el Modulador/Excitador con el Circulador y posteriormente trasmitir los datos a través del Sistema Radiante.

CAPÍTULO I

PLANTEAMIENTO DEL PROBLEMA

El desarrollo de la interfaz de salida de un prototipo industrializable de un modulador de transmisión de baja potencia para TDA implementado en software bajo el estándar ISDB-Tb constituye la segunda fase del diseño del bloque Modulador/Excitador llevado a cabo por la Dirección de Electrónica de Comunicaciones del CENDIT.

Para desarrollar dicha interfaz se deben cumplir los criterios de velocidad de transmisión de datos entre los componentes del sistema según el estándar ISDB-Tb. Es posible implementar la primera etapa de comunicación bajo diversos protocolos de comunicación, sin embargo, se determinó por estudios previos que el modo isócrono del estándar USB cumple la tasa de transmisión sugerida en el estándar ISDB-Tb.

Para la implementación de la interfaz de salida del modulador de transmisión de baja potencia se plantea el uso de una tarjeta de desarrollo de FPGA, ya que permite reprogramar la circuitería *in situ* haciendo más fácil la tarea de diseño y el posterior proceso de industrialización además de poseer puertos de entrada y salida que permiten integrar en el diseño otros dispositivos requeridos en otras fases como pudiera ser un DAC requerido en la fase 1 del diseño del bloque Modulador/Excitador.

OBJETIVOS

1.2.1. Objetivo General

Diseñar un sistema de comunicación entre un computador personal y un NanoBoard 3000 de modo isócrono con base en el USB.

1.2.2. Objetivos Específicos

1. Evaluar los aspectos teóricos de la comunicación computador personal-NanoBoard 3000 a través del USB.
2. Diseñar un IP Core USB para:
    - Manejar USB Cypress desde microcontrolador embebido en FPGA.
    - Configurar el sistema de transmisión isócrona del USB desde el computador personal hasta los puertos de salida de la NanoBoard 3000 de Altium.
3. Diseñar una interfaz de control de comunicación en el computador personal (basada en Java o C#).
4. Establecer la comunicación entre el computador personal y la NanoBoard 3000.

CAPÍTULO II

MARCO TEÓRICO

2.1. Televisión Digital Terrestre

La Televisión Digital Terrestre (TVD-T), que engloba también la televisión digital abierta (TDA) consiste en una transmisión a través del espectro radioeléctrico de imágenes y sonidos digitalizados que en conjunto forman videos que pueden estar acompañados de una serie de datos que luego de ser recibidos mediante equipos adecuados pueden presentarse al usuario en las pantallas como guías de programación, mensajes u otra información relevante.

Existen diversos estándares de TVD-T cuyas diferencias radican, entre otros aspectos, en la cantidad de portadoras y el tipo de modulación que utilizan. El sistema ISDB-Tb también llamado Sistema Brasileño de Televisión Digital (SBTVD) se basa en el sistema japonés (ISDB-T) al cual se le realizaron algunas modificaciones en Brasil con la estrecha colaboración de Japón durante el año 2008, entre las que destaca la posibilidad de emplear MPEG-4 para la compresión de datos [1].

El sistema de transmisión ISDB-Tb se organiza en 3 capas jerárquicas que son conformadas por uno o más segmentos OFDM, esta organización se aprecia en la Figura 2.1, el sistema de transmisión completo se observa en la Figura 2.2 donde se distinguen 4 secciones: entrada, bloques de codificación de canal, bloques de modulación y etapa final de conversión (elevación) de frecuencia, ampliación de frecuencia y filtrado [1].

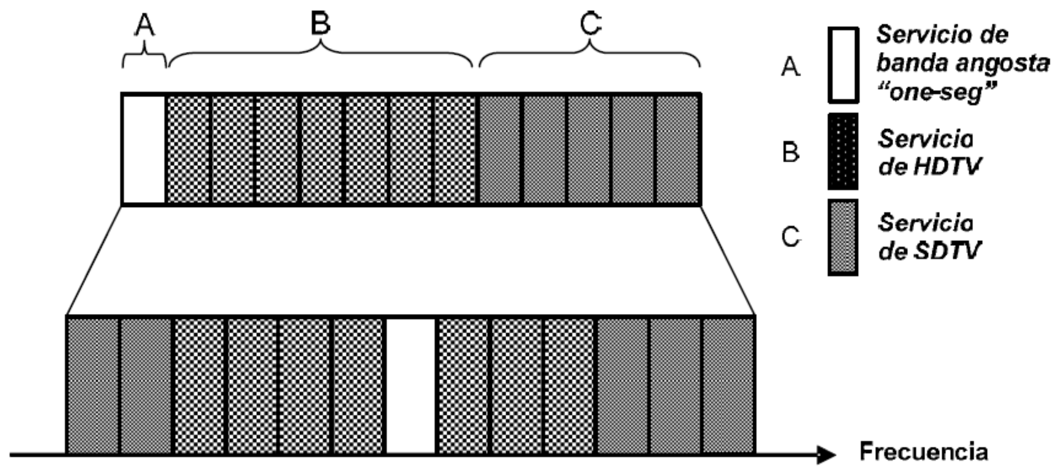


Figura 2.1: Trasnmisión jerárquica en tres capas [1].

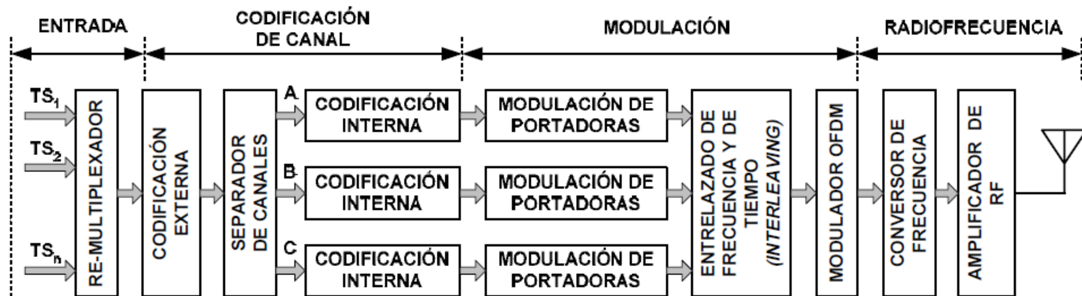


Figura 2.2: Sistema de transmisión ISDB-Tb [1].

2.2. USB

El USB es un bus de datos resultado del trabajo realizado por un comité fundador de 7 grandes empresas de la industria (Compaq, Hewlett-Packard, Lucent, Philips, Intel, Microsoft y NEC) publicado en el año 2000 para estandarizar las comunicaciones entre computadores personales y todo tipo de periféricos, ya que para ese entonces existían multitud de interfaces y esto constituía un problema. Desde aquel entonces el estándar USB aumentó su presencia en computadores y diversos dispositivos hasta ser hoy en día, indispensable para las comunicaciones [2].

2.2.1. Componentes del bus

El USB se compone de tres dispositivos: esclavo, *hub* y maestro (*host*) [3]:

El *host* es por lo general un computador que contiene un controlador para el USB y un *hub* raíz (*root*), estos dos componentes trabajan en conjunto para permitir al sistema operativo comunicarse con los dispositivos conectados al bus. Sus funciones son: gobernar todas las conexiones, comunicarse con los esclavos, asignar ancho de banda a los dispositivos conectados al bus, manejar el flujo de datos, detección de errores y proveer alimentación.

El *hub* raíz tiene como función detectar dispositivos esclavos conectados al bus, removerlos y pasar información entre estos dispositivos y el controlador del *host*. Existen *hubs* externos al maestro que permiten conectar varios periféricos a un mismo puerto compartiendo el ancho de banda asignado a dicho puerto.

Los dispositivos esclavos deben tener circuitos que les permitan comunicarse con el *host*, esto es: responder a las solicitudes estándar, describir sus características al *host*, intercambiar datos con el *host* y manejar la energía recibida desde el *host*. En

la Figura 2.3 se muestra la topología típica del USB donde se aprecian los componentes antes descritos.

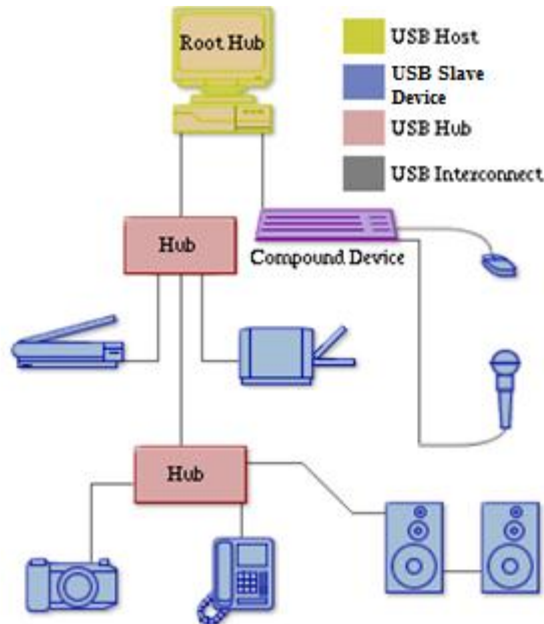


Figura 2.3: Topología típica del USB [4].

Hay tres clases de dispositivos esclavos de acuerdo a su velocidad de transferencia según el estándar USB (ver tabla 2.1):

Tabla 2.1: Velocidad de transferencia de datos de las especificaciones USB [4].

Tipo	Velocidad Máxima	Especificación
Super velocidad *(super-speed)*	5 Gb/s	(USB 3.0)
Alta velocidad *(high-speed)*	480 Mb/s	(USB 2.0)
Velocidad completa *(full-speed)*	12 Mb/s	(USB 1.0)
Baja velocidad *(low-speed)*	1.5 Mb/s	

Las velocidades *low-speed* y *full-speed* son utilizadas generalmente por dispositivos de interfaz humana en los cuales las velocidades de transferencia de

datos son bajas. Las velocidades 2.0 y 3.0 son más adecuadas para dispositivos de almacenamiento masivo y multimedia [3].

2.2.2. *Endpoints*

Los *endpoints* son búferes que almacenan los bytes que serán transmitidos al *host* o los que son recibidos desde el mismo. Para que exista comunicación entre *host* y esclavo, debe existir obligatoriamente al menos un *endpoint* en el esclavo, cada *endpoint* es unidireccional, es decir, solo puede enviar paquetes de datos hacia el *host* (*endpoint* tipo *IN*) o recibirlos (*endpoint* tipo *OUT*) pero se puede configurar el tipo de transmisión que soportará así como el tamaño del búfer. La nomenclatura *OUT-IN* es desde el punto de vista del *host*, aunque los *endpoints* se configuran y solo están presentes en el dispositivo esclavo. Existe un tipo de *endpoint* especial que debe tener cada dispositivo esclavo y se cataloga como un *endpoint* de control, es llamado *endpoint* 0. Éste *endpoint* es el único que puede tener un flujo de datos bidireccional, se encarga de la configuración del dispositivo y de manejar las solicitudes del *host*. Es también el único que debe estar definido en el dispositivo incluso después de un *reset* y así permitir las transferencias de control. Al momento de conectar un dispositivo al *host*, éste último asigna un identificador único a cada *endpoint* llamado *endpoint number*, éste número puede ir desde 0 hasta 15 [2].

2.2.3. Pipes

Un *pipe* es una asociación entre un *endpoint* del dispositivo y un búfer presente en el software controlador del *host*. Antes de que una transferencia ocurra, el *host* y el esclavo deben establecer un *pipe* [2].

Para permitir la transferencia de datos a través del *pipe* 0, al igual que con el *endpoint* 0, se debe establecer de forma automática un *pipe* 0 entre el *host* y el esclavo incluso después de un *reset*.

La Figura 2.4 muestra la conexión entre los *endpoints* del dispositivo y los búferes presentes en el software controlador del *host* a través de *pipes* que establecen el flujo de la comunicación.

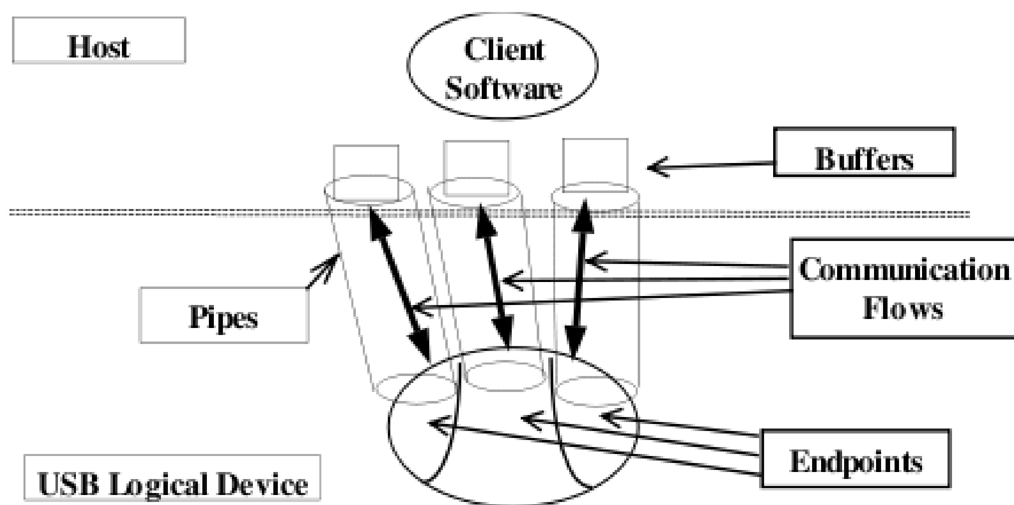


Figura 2.4: Conexión de *endpoints* y búferes mediante *pipes* [4].

2.2.4. Transferencias de datos

El estándar USB permite conectar gran variedad de periféricos a un *host* determinado. Para dar soporte a cada tipo de dispositivo según sus requerimientos de tasas de transmisión, tiempos de respuesta y corrección de errores se tienen cuatro tipos de transferencias: *Control*, *Bulk*, *Interrupt* e *Isochronous* (isócrona).

La transferencia de datos se lleva a cabo mediante transacciones, que a su vez están compuestas por tres paquetes: *token packet*, *data packet* y *handshake packet*:

- ***Token packet*:** indica el tipo y la dirección de la transacción, la dirección del dispositivo y el número de *endpoint*.
- ***Data packet*:** la información contenida en la transacción, puede venir del *host* o del dispositivo esclavo según el tipo de *endpoint* (*OUT* o *IN*).
- ***Handshake Packet*:** El dispositivo que recibe los datos (*host* o esclavo) responde con éste paquete para indicar si la transferencia fue correcta.

Las transacciones se llevan a cabo dividiendo el tiempo de acceso al bus para garantizar la sincronización entre el *host* y el dispositivo esclavo. Estas divisiones de tiempo están fijadas a 1ms en *low-speed* y *full-speed*, son llamadas ***frames***. Para los buses *high-speed* (debido a las altas tasas de transferencia de datos requeridas) se divide el tiempo de acceso al bus en fragmentos de 125 μs llamadas ***microframes***.

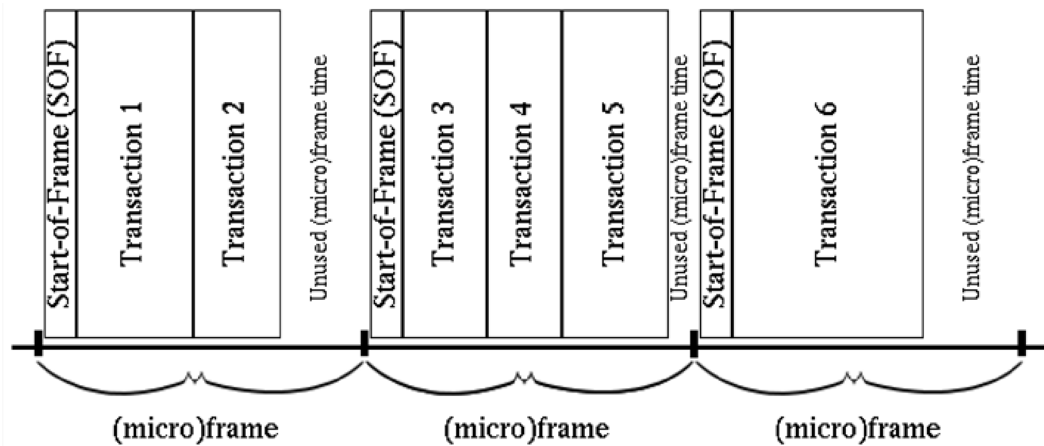


Figura 2.5: Transacciones contenidas en los *frames* y *microframes* [4].

El dispositivo que envía los datos coloca un paquete llamado ***start of frame* (SOF)** al comienzo de cada *frame* o *microframe* para garantizar la sincronización de

la comunicación, el resto del *frame* o *microframe* se utiliza para las transacciones como se observa en la Figura 2.5 [4]

Cada tipo de transferencia tiene un tamaño máximo de carga útil de datos que puede enviar a un determinado *endpoint* para su posterior transferencia, este tamaño también varía de acuerdo a la velocidad con la cual se trabaje como se muestra en la tabla 2.2.

Tabla 2.2: Tamaño máximo de carga útil para cada tipo de transferencia [2].

	Tamaño máximo de carga útil (bytes)		
Tipo de Transferencia	***High-speed***	***Full-speed***	***Low-speed***
Control	64	64	8
Bulk	512	64	No aplica
Interrupt	1024	64	8
Isócrona	1024	1023	No aplica

Cada tipo de transferencia está ideada para aplicaciones específicas y tiene características propias en el manejo de los datos a transmitir, dichas características se pueden observar y comparar en la tabla 2.3.

Las transferencias de tipo *Bulk* pueden llegar a velocidades de transmisión de datos que ningún otro tipo de transferencia es capaz de alcanzar, pero no se utilizan comúnmente en transmisión de audio y video debido a que no garantizan un flujo constante de datos, pudiéndose provocar cortes momentáneos en la transmisión, especialmente cuando ocurren errores en la transmisión de algún paquete, en estos aspectos las transferencias isócronas aventajan a los otros tipos de transferencia.

Tabla 2.3: Comparativa de tipos de transferencias USB [2].

Tipo de Transferencia	**Control**	***Bulk***	***Interrupt***	**Isócrona**
Uso Típico	Identificación y configuración	Impresora, escáner, disco duro	Mouse, teclado	Transmisión de audio y video (*streaming*)
Permite *low-speed*	Si	No	Si	No
Bytes de datos/ms por transferencia, máximo posible por pipe *(high-speed)*	15.872	53.248	24.572	24.576
Bytes de datos/ms por transferencia, máximo posible por *pipe* (*full-speed*)	832	1.216	64	1.023
Bytes de datos/ms por transferencia, máximo posible por *pipe* (*full-speed*)	24	No está permitido	0,8	No está permitido
Dirección de flujo de datos	*IN* y *OUT*	*IN* o *OUT*	*IN o OUT* (USB 1.0 solo soporta IN)	*IN o OUT*
Tipo de *Pipe*	*Message* (bidireccional)	*Stream* (Unidireccional)		
Ancho de banda reservado para todas las transferencias	10% en *low/full speed* 20% en *high speed.* (mínimo)	Ninguno	90% en *low/full speed*, 80% en *high speed* (máximo)	
Corrección de errores	Si	Si	Si	No
Tasa de transferencia asegurada	No	No	No	Si
Latencia asegurada (tiempo máximo entre transferencias)	No	No	Si	Si

2.2.5. Detalles de las capas de protocolos del USB

Los paquetes mencionados anteriormente: *Token Packet, Data Packet* y *Handshake Packet* contienen un PID (identificador de paquete), además de esto pueden contener diversos datos y bits de verificación de datos por redundancia cíclica (CRC), esto se observa en la Figura 2.6.

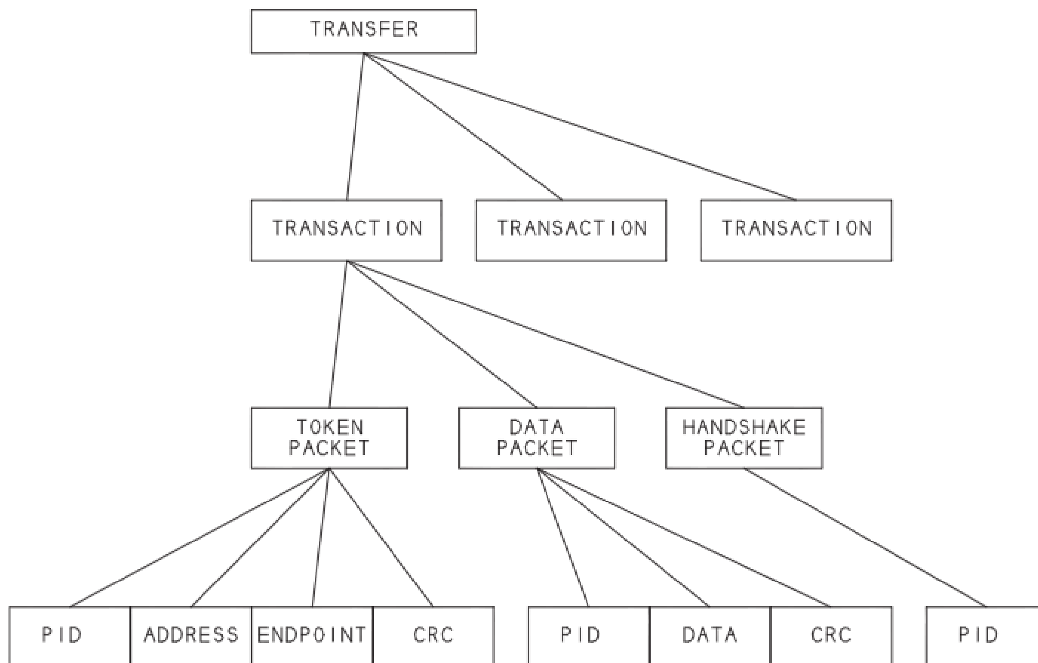


Figura 2.6: Componentes de las transferencias USB [2].

El código de detección de errores CRC permite detectar cambios accidentales en los datos (máximo 2 bits) y corregirlos, a excepción del modo isócrono, en el cual se pueden detectar errores pero no corregirlos. El PID permite identificar el tipo de paquete que se está enviando para que el destinatario pueda procesarlo de forma correcta. El USB fija un número binario entre 0001 y 1111 a cada tipo de paquete para su correcta identificación. El número 0000 está reservado y no se utiliza.

En las transferencias que requieren múltiples transacciones se utiliza un valor llamado *Data Toggle* para asegurar la sincronización y evitar la pérdida de datos, está incluido en el campo PID del *Token Packet*. Cada *endpoint* tiene su propio *Data Toggle*. Los *host* manejan el *Data Toggle* de forma automática sin requerir programación por parte del desarrollador y la mayoría de los chips controladores del USB hacen un manejo automático sin requerir código en el *firmware* para dicha tarea.

Las transferencias de tipo isócrono no pueden utilizar el *Data Toggle* para corregir errores debido a que no poseen *handshake*, en lugar de esto, las transferencias isócronas de alta velocidad y alto ancho de banda utilizan un orden e identificadores específicos para el envío de datos y mantener la sincronización [2].

Dichas transferencias pueden tener de una a tres transacciones por *microframe*, para ordenar los datos se utilizan identificadores de paquetes (PID) como se muestra en las tablas 2.4 y 2.5.

Tabla 2.4: Paquetes de datos según la cantidad de transacciones en transferencias *IN* de tipo Isócrono [2].

Cantidad de transacciones *IN* en el *microframe*	PID		
	Primera transacción	Segunda Transacción	Tercera Transacción
1	DATA0	-	-
2	DATA1	DATA0	-
3	DATA2	DATA1	DATA0

Tabla 2.5: Paquetes de datos según la cantidad de transacciones en transferencias *OUT* de tipo Isócrono [2].

Cantidad de transacciones *OUT* en el *microframe*	PID		
	Primera transacción	Segunda Transacción	Tercera Transacción
1	DATA0	-	-
2	MDATA	DATA1	-
3	MDATA	MDATA	DATA2

Los paquetes de *handshake* sirven para conocer el estatus de la información enviada o recibida, y para sincronizar las solicitudes de control, estos son:

- **ACK (*Acknowledge*):** indica que el *host* o el dispositivo esclavo ha recibido los datos sin error.

- **NAK (*Negative acknowledge*):** indica que el dispositivo está ocupado o no tiene datos que enviar. Todos los dispositivos esclavos deben responder con NAK cuando están siendo configurados mediante transacciones de control. El *host* nunca puede enviar un NAK.

- **STALL:** Se puede dar en tres casos:

    - Cuando se recibe una solicitud de control no soportada.
    - Cuando se recibe una solicitud de control soportada pero que el dispositivo no puede procesar por alguna razón.
    - Cuando el *endpoint* solicitado no está preparado para enviar o recibir datos.

  Cuando se recibe un STALL, el *host* suspende las solicitudes pendientes y no prosigue con la comunicación hasta que el *host* envíe una solicitud satisfactoria para borrar el STALL. El *host* nunca puede enviar un STALL al dispositivo periférico.

- **NYET (*Not yet*):** Sólo es utilizada por dispositivos *high-speed*, indica que el dispositivo recibió el paquete pero aún no está preparado para procesarlo, esto se debe principalmente a que hay otro paquete en el búfer del *endpoint*.

- **ERR:** Es utilizada solamente por los *hubs* de *high-speed* e indica al *host* que no se recibió el *handshake* esperado por parte del dispositivo para realizar la transacción.

En la Tabla 2.6 se muestran los diferentes tipos de paquetes con sus principales características incluyendo el PID.

Tabla 2.6: Tipos de paquetes en las transacciones USB [2].

Tipo de paquete	**PID**	**Valor**	**Usado en transferencias de tipo**	**Fuente**	**Vel.**	**Descripción**
Token	OUT	0001	Todas	Host	Todas	Dirección de *endpoint* y *device* para transferencias OUT.
	IN	1001	Todas	Host	Todas	Dirección de *endpoint* y *device* para transferencias IN.
	SOF	0101	Star-of-Frame	Host	Todas	Marcador Start-of-Frame y número de *frame.*
	SETUP	1101	Control	Host	Todas	Dirección de *endpoint* y *device* para transferencias SETUP.
Data	DATA0	0011	Todas	Host, device	Todas	Data toggle, secuencia de PID de datos.
	DATA1	1011	Todas	Host, device	Todas	Data toggle, secuencia de PID de datos.
	DATA2	0111	Isócrona	Host, device	High	Secuencia de PID de datos.
	MDATA	1111	Isócrona,	Host, device	High	Secuencia de PID de datos.
Handshake	ACK	0010	Todas	Host, device	Todas	Receptor acepta paquete de datos libre de errores.
	NAK	1010	Control, Bulk,Interrupt	Device	Todas	Receptor no puede aceptar datos o el remitente no puede enviar datos o no tiene datos para transmitir.
	STALL	1110	Control, Bulk,Interrupt	Device	Todas	Una solicitud de control no es soportada o el endpoint ha sido detenido.
	NYET	0110	Escritura de control, *Bulk* OUT	Device	High	El dispositivo recibió el paquete sin errores pero no está listo para procesarlo.

Especial	PRE	1100	control, interrupt	Host	Full	Preámbulo usado por el host, indica próximo paquete es low-speed.
	ERR	1100	Todas	Host	High	Regresado por el HUB para informar de un error en transferencias low-speed o full-speed.
	SPLIT	1000	Todas	Host	High	Precede un token packet para indicar una transacción *split.*
	PING	0100	Escritura de control, Bulk OUT	Host	High	Chequea si el receptor está listo luego de haber recibido un NYET.
	reservado	0000	-	-	--	Para usos futuros.

2.2.6. Enumeración del dispositivo esclavo

Cuando se conecta un dispositivo esclavo a un puerto USB se llevan a cabo una serie de procedimientos que garantizan al *host* el conocimiento de todas las características del dispositivo y así proporcionarle entre otras cosas, energía y ancho de banda adecuado para el correcto intercambio de información. Los procesos que se llevan a cabo durante la enumeración son los siguientes [4]:

1. El *hub* al cual fue conectado el dispositivo indica al *host* que hubo un cambio en el pipe. El dispositivo entra en el estado energizado en éste punto pero el puerto al cual fue conectado está desactivado.
2. El *host* requiere más información del *hub* para determinar que un dispositivo fue conectado y a qué puerto.
3. El *host* debe esperar al menos 100ms para permitir al dispositivo completar el proceso de inserción y estabilización de energía. Luego de esto, el *host* activa el puerto y emite una señal de *reset* al dispositivo por al menos 50 ms.
4. El *hub* realiza cualquier proceso de *reset* requerido. Después de que la señal de *reset* ha sido enviada, el puerto es activado y el dispositivo entra en el estado por defecto.

5. El *host* asigna una dirección única al dispositivo, esto lleva al dispositivo al estado de direccionamiento.
6. El *host* solicita el descriptor del dispositivo a través del pipe de control para determinar el tamaño máximo de carga útil que puede procesar el pipe de control del dispositivo. Éste proceso puede ocurrir luego de que el *host* asigne la dirección.
7. El *host* lee todas las opciones posibles de configuración del dispositivo.
8. El *host* selecciona una configuración de la lista de configuraciones soportadas por el dispositivo y ordena al dispositivo usar la misma configuración. Opcionalmente, el *host* puede seleccionar también una interfaz alternativa de una determinada configuración. Todos los *endpoints* son inicializados como se describe en la configuración seleccionada y el dispositivo está listo para usarse.

Los estados que puede tener un dispositivo esclavo son los siguientes:

- **Conectado**: Inmediatamente después de conectado al bus, no está contemplado en el estándar USB.
- **Energizado:** Aún no está preparado para compartir información con el *host* pero toma energía de éste o de alguna fuente externa.
- **Por defecto:** Sucede al recibir el *reset* desde el *hub*. Está preparado para recibir la asignación de una dirección.
- **Direccionamiento:** Recibida la dirección permanece en este estado hasta que se configure totalmente.
- **Configurado:** En éste estado examina las configuraciones y selecciona una. Todos los *endpoints* son configurados para la transmisión de datos.
- **Suspendido:** Cuando no hay tráfico de datos por 1ms el dispositivo entra en éste estado caracterizado por bajo consumo de energía. Se mantienen las configuraciones. El dispositivo se activa cuando detecta actividad en el bus pero el *host* le debe dar 10ms para que responda a las solicitudes.

2.2.7. Descriptores

Los descriptores USB son estructuras de datos o bloques de información que permiten al *host* “aprender” u obtener las características del dispositivo esclavo. Cada descriptor contiene información acerca del dispositivo como un todo o de cada elemento del mismo. Todos los dispositivos esclavos deben guardar la información de los descriptores y responder a la solicitud estándar de descriptores USB [3].

Tipos de descriptores:

- ***Device descriptor*****:** Cada dispositivo tiene un *device descriptor* que contiene información acerca del dispositivo completo y especifica el número de configuraciones que el dispositivo soporta.

- ***Configuration descriptors*****:** Cada dispositivo puede tener uno o más descriptores de configuración que contiene información acerca del consumo de energía y del número de interfaces que soporta dicha configuración.

- ***Interface descriptors*****:** Definen cero o más *endpoints descriptors* que contienen la información necesaria para realizar la comunicación con un *endpoint*.

- ***Endpoint descriptors*****:** Contienen información acerca de cómo los *endpoints* transfieren los datos. Una interfaz que no contiene *endpoint descriptors* debe usar el *endpoint* de control para realizar la comunicación.

- ***String descriptors*****:** Pueden contener texto como el nombre del dispositivo y del fabricante. Otros descriptores pueden tener valores indexados que apunten a los descriptores string y el *host* puede leer estos descriptores usando las solicitudes Get_Descriptor.

Cada descriptor consiste en una serie de campos y contiene un valor que identifica el tipo de descriptor, la mayoría de los nombres de esos campos utilizan prefijos para indicar el formato del contenido de dicho campo: b=byte (8 bits), w=Word (16bits), bm=*bitmap*, bcd=código binario decimal, i=index, id= identificador.

Tabla 2.7: Campo bDescriptorType [2].

bDescriptorType	Tipo de Descriptor	Obligatorio
01h	*Device*	Si
02h	*Configuration*	Si
03h	*String*	No. Texto opcional
04h	*Interface*	Si
05h	*Endpoint*	No. Se puede utilizar solo el *endpoint* 0
06h	*Device_qualifier*	Si, para los dispositivos que soportan *full y high-speed*. No está permitido para otros dispositivos.
07h	*Other_speed _configuration*	Si, para los dispositivos que soportan *full y high-speed*. No está permitido para otros dispositivos.
08h	*Interface_power*	No.
09h	*OTG*	Sólo es obligatorio para dispositivos On-The-Go.
0Ah	*Debug*	No.
0Bh	*Interface_association*	Sólo para dispositivos compuestos.

A continuación se presentan una serie de tablas que detallan la estructura de cada tipo de descriptor, donde se presenta cada campo con su longitud en bytes y una breve descripción [2].

Tabla 2.8: Estructura del *device descriptor*.

Posición (decimal)	**Campo**	**Tamaño (bytes)**	**Descripción**
0	bLength	1	Tamaño del descriptor en bytes.
1	bDescriptorType	1	Constante del *device descriptor* (01h)
2	bcdUSB	2	Numero de versión de especificación USB (BCD).
4	dDeviceClass	1	Código de clase.
5	dDeviceSubclass	1	Código de subclase.
6	bDeviceProtocol	1	Código de protocolo.
7	bMaxPacketSize0	1	Tamaño máximo de paquete para el *endpoint* 0.
8	idVendor	2	*Vendor* ID.
10	idProduct	2	*Product* ID.
12	bcdDevice	2	Número de versión del dispositivo (BCD).
14	iManufacturer	1	Índice del *string descriptor* del fabricante.
15	iProduct	1	Índice del *string descriptor* del producto.
16	iSerialNumber	1	Índice del *string descriptor* del número de serie.
17	bNumConfiguration	1	Número de configuraciones posibles.

Tabla 2.9: Valores específicos que puede tomar el campo bDeviceClass del *device descriptor*.

bDeviceClass	**Descripción**
00h	El interface descriptor nombra la clase.
02h	Comunicaciones.
09h	Hub.
DCh	Dispositivo de diagnóstico bDeviceSubClass=1 para dispositivo de diagnóstico reprogramable con bDeviceProtocol=1 para dispositivo de conformidad USB 2.0
E0h	Controlador inalámbrico bDeviceSubClass=1 para controlador de radio frecuencia con bDeviceProtocol=1 para dispositivo con interfaz de programación bluetooth.
EFh	Diversos dispositivos bDeviceSubClass=2 para clases comunes con bDeviceProtocol=1 para descriptor de asociación de interfaces.
FFh	Específico del fabricante.

Tabla 2.10: Estructura del *device qualifier descriptor*.

Posición (decimal)	**Campo**	**Tamaño (bytes)**	**Descripción**
0	bLength	1	Tamaño del descriptor en bytes.
1	bDescriptorType	1	Constante del *device qualifier* descriptor (06h).
2	bcdUSB	2	Numero de versión de especificación USB (BCD).
4	dDeviceClass	1	Código de clase.
5	dDeviceSubclass	1	Código de subclase.
6	bDeviceProtocol	1	Código de protocolo.
7	bMaxPacketSize0	1	Tamaño máximo de paquete para el *endpoint* 0.
8	bNumConfigurations	1	Número de configuraciones posibles.
9	Reserved	1	Para uso futuro.

Tabla 2.11: Estructura del *configuration descriptor*.

Posición (decimal)	**Campo**	**Tamaño (bytes)**	**Descripción**
0	bLength	1	Tamaño del descriptor en bytes.
1	bDescriptorType	1	Constante del *configuration descriptor* (02h).
2	wTotalLength	2	Número total de bytes en el *configuration descriptor* y en todos sus descriptores subordinados.
4	bNumInterfaces	1	Número de interfaces en la configuración.
5	dConfiguration Value	1	Identificador para las solicitudes Set_Configuration y Get_Configuration.
6	iConfiguration	1	Índice del *string descriptor* para la configuración.
7	bmAttributes	1	Ajustes de alimentación y *wakeup* remoto.
8	bMaxPower	1	Energía máxima requerida por el dispositivo en unidades de 2 mA.

El descriptor *other_speed _configuration* tiene exactamente los mismos campos que el *configuration descriptor*.

Tabla 2.12: Estructura del *interface descriptor*.

Posición (decimal)	**Campo**	**Tamaño (bytes)**	**Descripción**
0	bLength	1	Tamaño del descriptor en bytes.
1	bDescriptorType	1	Constante del *interface descriptor* (04h).
2	bInterfaceNumber	1	Número identificador de esta interfaz.
3	bAlternateSettting	1	Identificador del *Alternate Setting*.
4	dNumEndpoints	1	Número de *endpoints* definidos
5	dDeviceClass	1	Código de clase.
6	dDeviceSubclass	1	Código de subclase.
7	bDeviceProtocol	1	Código de protocolo.
8	iInterface	1	Índice del *string descriptor* de la interfaz.

Tabla 2.13: El campo bInterfaceClass indica el tipo de dispositivo.

Código de Clase (hex)	**Descripción**
01	Audio.
02	Dispositivo de comunicaciones (interfaz de comunicación).
03	Dispositivo de interfaz humana.
05	Físico.
06	Imagen.
07	Impresora.
08	Almacenamiento masivo.
09	Hub.
0A	Dispositivo de comunicaciones (interfaz de datos).
0B	Tarjeta inteligente.
0D	Contenido de seguridad.
0E	Video.
DC	Dispositivo de diagnóstico. bDeviceSubClass=1 para dispositivo de diagnóstico reprogramable con bDeviceProtocol=1 para dispositivo de conformidad USB 2.0
E0	Controlador inalámbrico. bDeviceSubClass=1 para controlador de radio frecuencia con bDeviceProtocol=1 para dispositivo con interfaz de programación bluetooth.
FE	Aplicación específica. bInterfaceSubClass=1, para actualización de *firmware* del dispositivo. bInterfaceSubClass=2, para puente IrDA. bInterfaceSubClass=3, para pruebas y mediciones.
FF	Específico del fabricante.

Tabla 2.14: Estructura del *endpoint descriptor*.

Posición (decimal)	**Campo**	**Tamaño (bytes)**	**Descripción**
0	bLength	1	Tamaño del descriptor en bytes.
1	bDescriptorType	1	Constante del *interface descriptor* (05h).
2	bEndpointAddress	1	Número y dirección del *endpoint*.
3	bmAttributes	1	Tipo de transferencia soportada.
4	wMaxPacketSize	2	Tamaño máximo de paquete soportado.
5	bInterval	1	Tasa máxima de latencia/intervalo de solicitudes/NAK.

Tabla 2.15: Estructura del *string descriptor*.

Posición (decimal)	**Campo**	**Tamaño (bytes)**	**Descripción**
0	bLength	1	Tamaño del descriptor en bytes.
1	bDescriptorType	1	Constante del *interface descriptor* (03h).
2	bSTRING o wLANGID	varios	Arreglo de 1 o más códigos de identificación de lenguaje para el *string descriptor* 0. Para los otros *string descriptor* es una cadena Unicode.

2.2.8. Solicitudes del *Host*.

Las transferencias de control son las únicas que tienen funciones definidas por el estándar USB, todos los dispositivos conectados a un puerto deben responder a dichas solicitudes realizadas por el *host* si aún no se le ha asignado una dirección o si el dispositivo no ha sido configurado. Para realizar una solicitud, el software del *host* crea la etapa de datos *Setup* usando el código de solicitud apropiado. En la Tabla 2.16 se observan las solicitudes que puede realizar el host a los dispositivos esclavos.

Tabla 2.16: Solicitudes de control definidas por el estándar USB [2].

Número solicitud	Solicitud	Fuente datos	Receptor	Tamaño datos (bytes)	Dato
00h	Get_Status	*Device*	*Device* interfaz, *endpoint*	2	Status
01h	Clear_Feature	Host	*Device*	-	-
03h	Set_Feature	Host	*Device*	-	-
05h	Set_Address	Host	*Device*	-	-
06h	Get_ Descriptor	*Device*	*Device*	Tamaño del descriptor	Descriptor
07h	Set_ Descriptor	Host	*Device*	Tamaño del descriptor	Descriptor
08h	Get_ Configuration	*Device*	*Device*	1	Descriptor
09h	Set_ Configuration	Host	*Device*	-	-
0Ah	Get_Interface	*Device*	Interfaz	1	*Alternate Setting*
0Bh	Set_Interface	Host	Interfaz	-	-
0Ch	Synch_Frame	*Device*	*Endpoint*	2	Número de *frame*

2.2.9. EZ-USB® FX2LP™ USB *Microcontroller High-Speed* USB Peripheral *Controller.*

El dispositivo EZ-USB® FX2LP™ (CY7C68013A) de Cypress, es un microcontrolador con USB 2.0 altamente integrado y de bajo consumo. Integra un transceptor USB 2.0, un motor de interfaz serie (SIE), una versión mejorada del microcontrolador 8051, búferes de datos y una interfaz programable (GPIF) en un solo chip [8], ver Figura 2.7.

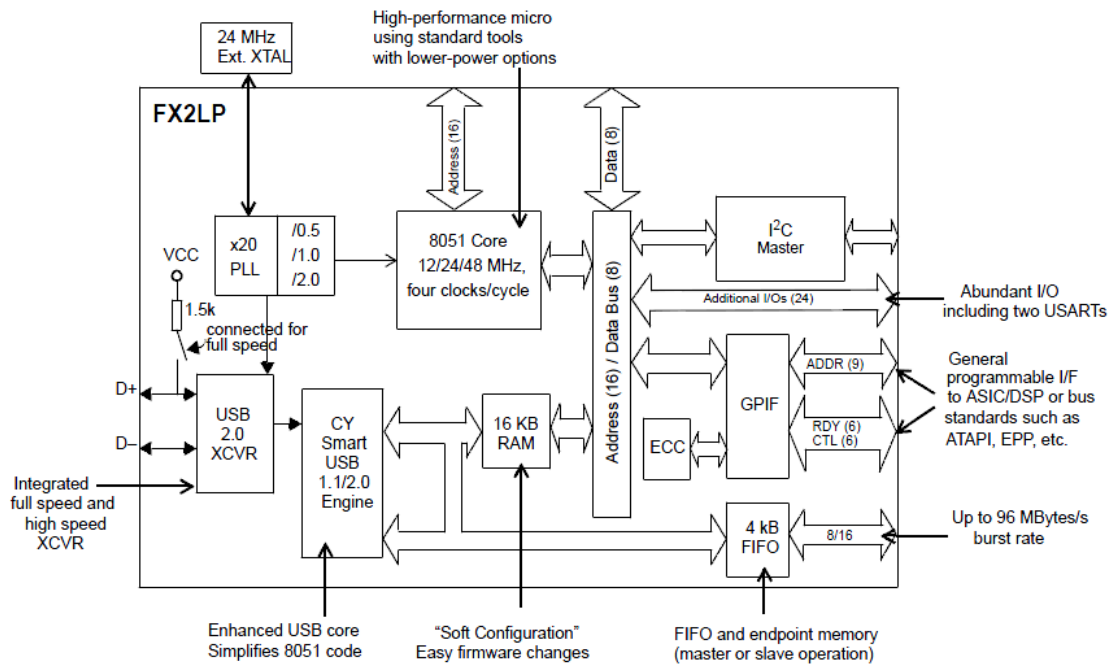


Figura 2.7: Diagrama de bloques del FX2LP [8].

La función del FX2LP es transferir datos entre un *host* y un dispositivo periférico, el FX2LP provee esta conectividad a través de interfaces serie y paralelo. En sistemas en los cuales el CPU del FX2LP no es requerido para modificar los datos antes de enviarlos al *host* o al dispositivo esclavo, el *firmware* solo necesita inicializar las unidades de transferencia, permitiendo a las transferencias USB proceder sin la intervención del CPU.

Para conectarse al *host*, el FX2LP incluye una interfaz de bus física (PHY) que se conecta directamente al USB mediante un cable y contiene un transceptor USB 2.0 que soporta transferencias USB *high-speed* y *full-speed*.

Conectado a dicha interfaz, se encuentra el motor de interfaz serie inteligente que convierte en bytes la señal proveniente del PHY. Éste motor también se encarga de manejar algunos detalles de bajo nivel del USB como corrección de errores y sincronización de los identificadores de paquetes (PID). También contiene la lógica que permite enumerar al FX2LP como un dispositivo esclavo y es capaz de almacenar código en su RAM interna. El SIE entrega los bytes a las memorias FIFO (*endpoint* búferes) y éstas conducen a través de los pipes a las interfaces paralelas.

El FX2LP contiene **dos interfaces paralelas**: *Slave* FIFO y GPIF. Estas interfaces permiten conectar el FX2LP al dispositivo esclavo.

2.2.9.1. Slave FIFO

Esta interfaz es adecuada para ser conectada a dispositivos externos que poseen controladores FIFO como un microcontrolador, una FPGA o un ASIC. Provee un bus de datos de 8 o 16 bits, puede operar de forma síncrona o asíncrona, puede operar con el *clock* interno o uno externo, tiene banderas de salida que indican el estado de los FIFO como lleno o vacío, tiene entradas de control, permite seleccionar el FIFO mediante líneas externas y acciona las transferencias USB cuando el FIFO está lleno o vacío. Estas características hacen fácil su manejo desde un maestro externo. En la Figura 2.8 se aprecia el camino que siguen los datos desde la interfaz de bus física hasta llegar al bus FIFO.

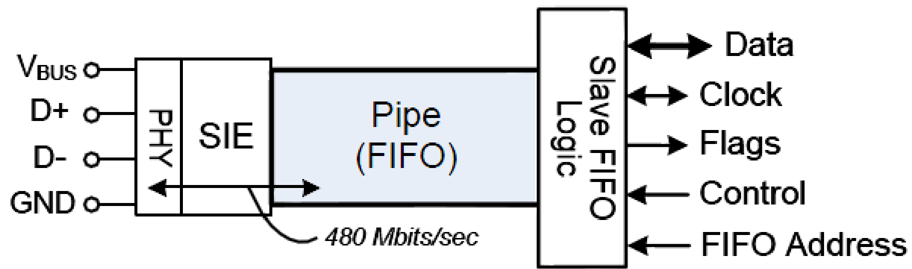


Figura 2.8: Bloques del FX2LP que permiten compartir datos entre *host* y el dispositivo esclavo mediante Slave FIFO [8].

2.2.9.2. GPIF

Esta interfaz es adecuada para ser conectada a dispositivos externos que no poseen controladores FIFO. Para utilizar la interfaz GPIF se requiere el diseño de una máquina de estados que controla el proceso de entrada o salida de datos mediante formas de onda, esto se hace con una herramienta que proporciona Cypress de forma gratuita llamada GPIF Designer. La interfaz GPIF proporciona salida de datos de 8 o 16 bits, trabaja de forma síncrona, acepta *clock* externo o interno, provee nueve salidas de dirección así como seis señales de control incluyendo señal *ready* [7]. En la Figura 2.9 se observa el camino que siguen los datos desde la interfaz de bus física hasta llegar al GPIF.

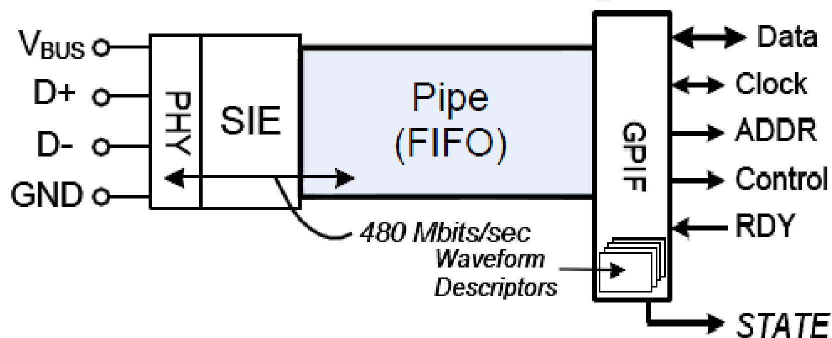


Figura 2.9: Bloques del FX2LP que permiten compartir datos entre *host* y el dispositivo esclavo mediante el GPIF [8].

Además de las interfaces paralelas mencionadas anteriormente, el FX2LP posee interfaces seriales para conectar diversos dispositivos [7]:

- I^2C maestro que opera a 100 o 400kHz.
- Dos UART estándar provenientes del 8051.
- Hasta 40 pines de entradasy salidas para propósitos generales (GPIO).
- La cantidad de pines disponibles varía de acuerdo al chip seleccionado. Hay tres opciones: 56, 100 o 128 pines.

El firmware del FX2 puede ser cargado en la RAM interna o en una EEPROM externa. En la Figura 2.10 se aprecia la distribución de los pines del FX2LP de acuerdo a la versión (56, 100 o 128 pines).

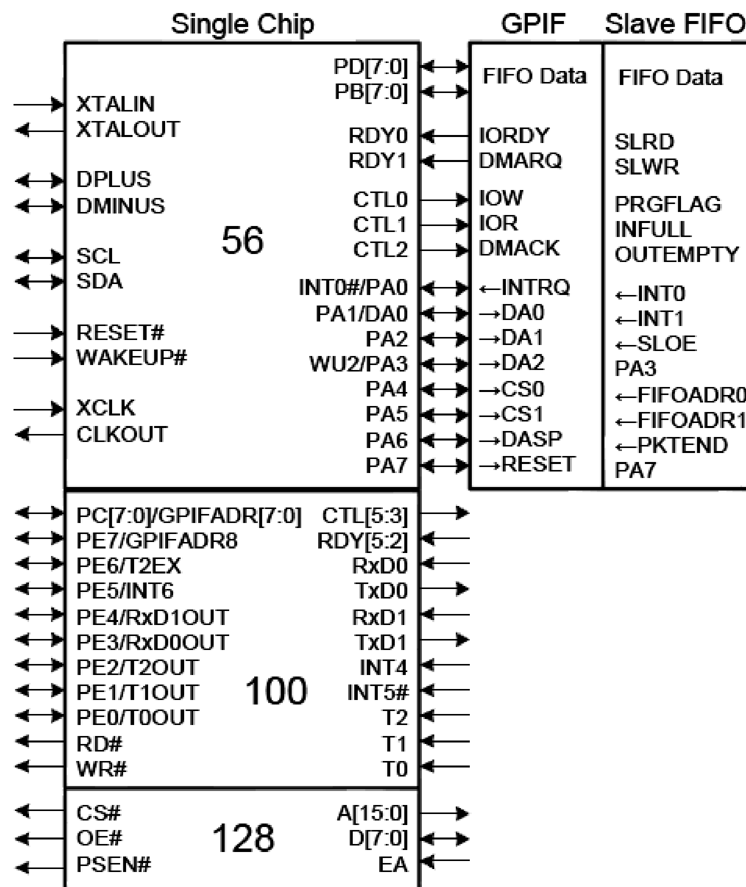


Figura 2.10: Pines del FX2LP de acuerdo al tamaño del encapsulado [8].

2.2.9.3. Placa de desarrollo FX2LP

La placa de desarrollo FX2LP (ver Figura 2.11) incorpora un chip CY7C68013A-56PVXC cuyo encapsulado es de 56 pines que tiene las características anteriormente descritas. La placa posee los elementos necesarios para realizar desarrollos con el chip, entre estos se encuentran:

- Conector USB.
- Botón de *reset*.
- *Switch* de encendido/apagado.
- Regulador de tensión a 3.3V.
- Oscilador de 24 MHz.
- Memoria EEPROM de 64kB.
- *Jumper* que controla el acceso a la RAM interna o a la EEPROM para la carga del *firmware*.

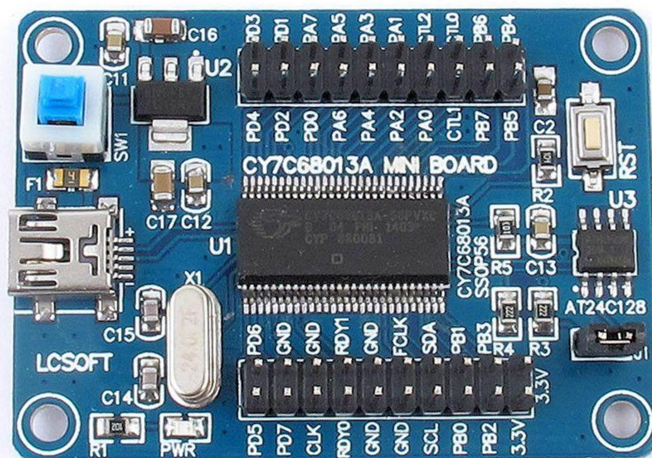


Figura 2.11: Placa de desarrollo FX2LP [12].

2.2.10. EZ-USB® SX2™ *High-Speed* USB *Interface Device*.

El SX2 es un dispositivo diseñado para proveer una interfaz USB 2.0 a microprocesadores, DSPs, ASICs y FPGAs dando soporte a cualquier diseño de periférico. Posee un transceptor, un motor de interfaz serial (SIE), 4kB reservados para los FIFO al igual que el FX2LP pero carece del núcleo 8051 (ver diagrama Figura 2.12), esto implica que los descriptores deben ser cargados en una EEPROM. El SX2 posee registros internos que el maestro debe escribir al iniciar el dispositivo y de ésta manera configurar todas las características de acuerdo a los descriptores. En el resto de características es prácticamente idéntico al FX2 salvo algunos registros específicos [13].

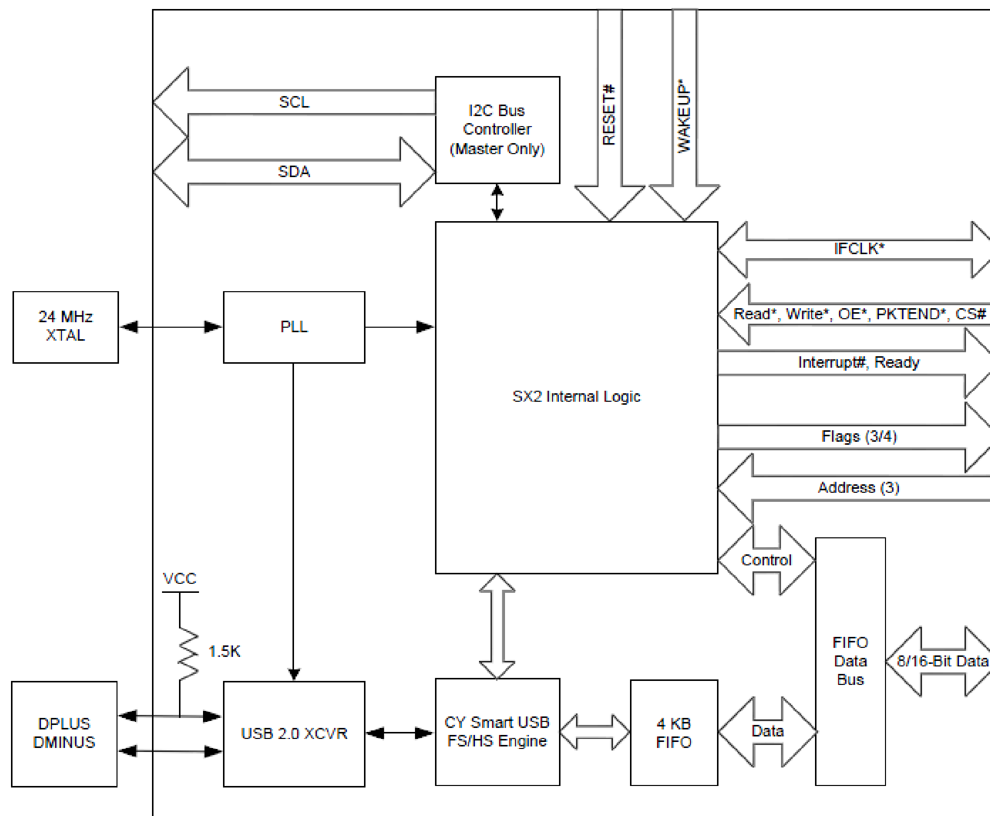


Figura 2.12: Diagrama de bloques del SX2 [13].

2.2.11. Hardware USB

Existen tres tipos de puertos y conectores USB: tipo A, B y C. El tipo C es el más reciente pero la mayoría de dispositivos aún utiliza las interfaces A y B. Los puertos USB tipo A se encuentran principalmente en el *host* o el dispositivo que maneja la conexión, son de forma rectangular y sólo se pueden conectar por una cara. Los conectores tipo B por lo general se encuentran en dispositivos periféricos. Los conectores tipo C presentan la misma forma tanto en el *host* como en el dispositivo esclavo, pueden ser conectados por ambas caras debido a su simetría y permiten mayor transferencia de energía y velocidad de datos para adaptarse a las especificaciones del USB 3.0 y 3.1 [10].

En la Figura 2.13 se aprecian los distintos conectores USB tipo A y B para el estándar USB 2.0.

Estandar USB A y B descripción de pines

Pin	Nombre	Color	Función
1	Vcc	rojo	+5 voltaje de alimentación
2	D-	blanco	Dato - línea de señal negativa
3	D+	verde	Dato + línea de señal positiva
4	GND	negro	Tierra o masa

Tipo A: usado en dispositivos que proveen alimentación admite usb 1.0; 2.0 y recientemente existe una modificación para 3.0.
Tipo B: usado en dispositivos que reciben alimentación (mayoría de dispositivos periféricos). Solo usb 1.0

Mini USB A y B descripción de pines

Pin	Nombre	Color	Función
1	Vcc	rojo	+5 voltaje de alimentación
2	D-	blanco	Dato - línea de señal negativa
3	D+	verde	Dato + línea de señal positiva
4	ID	-	No conectado
5	GND	negro	Tierra o masa

Tipo A: no usado.
Tipo B: corresponde a una actualización de la especificación de velocidad a USB 2.0

Micro USB AB y B descripción de pines

Pin	Nombre	Color	Función
1	Vcc	rojo	+5 voltaje de alimentación
2	D-	blanco	Dato - línea de señal negativa
3	D+	verde	Dato + línea de señal positiva
4	ID	-	No conectado: trabaja como conector B Conectado a GND: trabaja como conector A
5	GND	negro	Tierra o masa

Tipo AB: usado en dispositivos que funcionen tanto en modo esclavo como maestro.
Tipo B: miniaturización de los conectores mini B

Figura 2.13.1: Conectores USB tipo A y B [11].

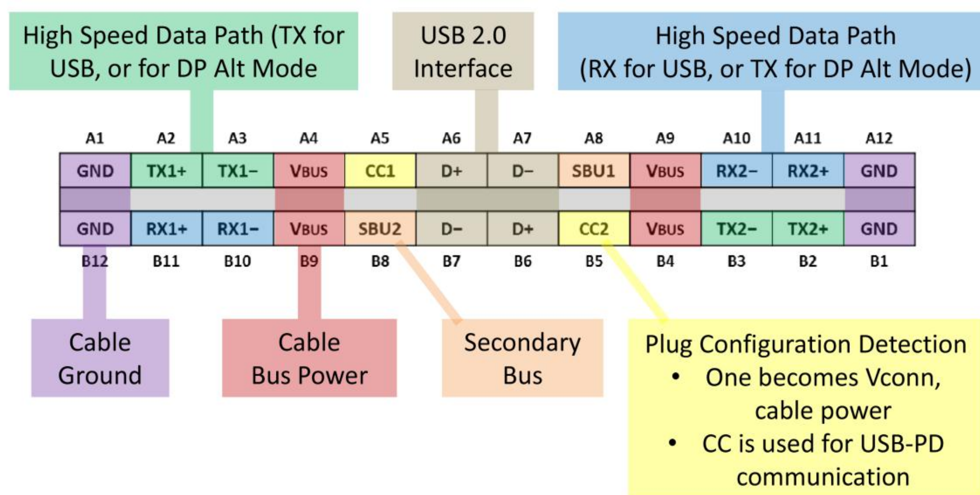


Figura 2.14.2: Pines del conector USB tipo C [10].

2.3. FPGA y Tarjeta de desarrollo NanoBoard 3000AL de Altium

Un FPGA (del inglés *Field Programmable Gate Array*) es un chip de silicio reprogramable que utiliza bloques de lógica pre-construidos y recursos para ruteo reprogramables. Su principal ventaja es que permite implementar funcionalidades en hardware sin tener que utilizar un *protoboard* o un cautín. Para llevar a cabo la implementación sólo es necesario desarrollar tareas de cómputo digital en software y compilarlas en un archivo de configuración o *bitstream* que contenga información de cómo deben conectarse los componentes. A diferencia de los procesadores, los FPGAs llevan a cabo diferentes operaciones de manera paralela, por lo que éstos no necesitan competir por los mismos recursos. Cada tarea de procesos independientes se asigna a una sección dedicada del chip, y puede ejecutarse de manera autónoma sin ser afectada por otros bloques de lógica. [13]

La tarjeta de desarrollo Nanoboard 3000AL es una herramienta que permite realizar proyectos para FPGA, específicamente para la FPGA Cyclone III EP3C10F256C8N que pertenece a la familia Altera. La NanoBoard 3000 incorpora toda la circuitería y los periféricos necesarios para hacer funcionar la FPGA y

brindarle conectividad a través de los principales buses de datos disponibles en la industria incluyendo además distintos módulos de hardware que son de utilidad para el desarrollador.

Adicional a estos recursos de hardware, la arquitectura de la FPGA permite crear componentes virtuales que agrupan bloques lógicos formando así los llamados Módulos IP (*Intelectual Property*) que pueden ser diseñados, sintetizados y reutilizados en la FPGA. Estos módulos consisten en descripciones de hardware realizadas en lenguajes HDL (*Hardware Description Languaje*). La NanoBoard 3000 posee también puertos y controladores dispuestos para el manejo de señales analógicas y digitales, así como un reloj programable que funciona en un rango de 6Hz a 200MHz y otro fijo referencial de 20MHz [11]. (Ver Figura 2.14)

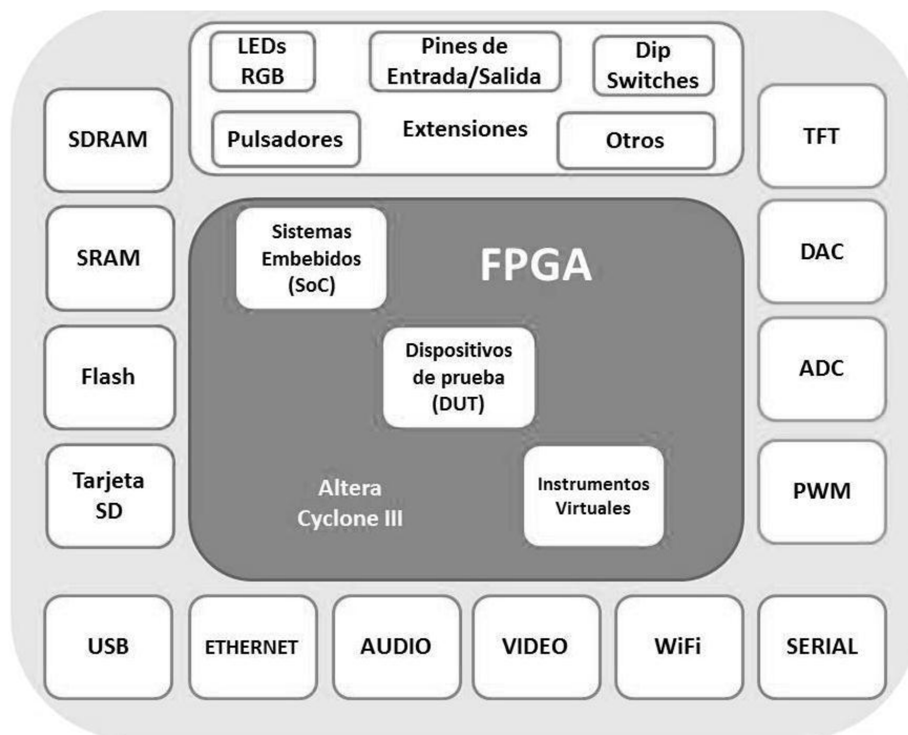


Figura 2.15: Módulos de la NanoBoard 3000 [11].

Figura 2.16: NanoBoard 3000AL [14].

2.3.1. Sistema Controlador NanoTalk

La NanoBoard 3000 utiliza una FPGA como sistema controlador para la tarjeta, este dispositivo es comúnmente llamado *host* FPGA o controlador NanoTalk. Es el dispositivo maestro responsable de la comunicación y configuración a través del JTAG multiplexando las cadenas de comunicación con JTAG Software, JTAG Hardware y los recursos locales de la NanoBoard 3000, tales como:

- Interfaz USB-PC: a través de un sistema JTAG virtual implementado en un puerto USB 2.0
- Sistema de pines JTAG: para configurar desde otros sistemas compatibles con FPGA Altera
- FPGA del usuario: Desde Altium DXP

- Tarjetas periféricas y panel TFT.
- Dispositivos basados en el protocolo SPI.

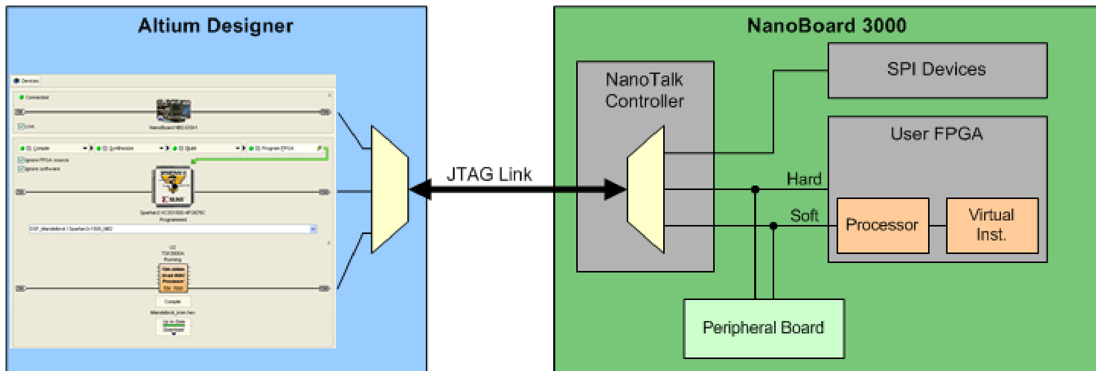


Figura 2.17: Esquema de comunicación JTAG de la NanoBoard 3000 [14].

2.3.2. Altium Software DXP

El software DXP de Altium es una poderosa plataforma de ingeniería que combina el diseño de circuitos PCB, desarrollo de FPGA y su posible simulación e interconexión en un solo programa. Para realizar esto, Altium DXP permite crear varios tipos de proyectos [15]:

- **Diseño de PCB (*.PRJPCB):** Es el conjunto de documentos requeridos para fabricar una placa de circuito impreso. El circuito electrónico es capturado como un esquemático, y los componentes son cargados desde librerías estándar o creadas por el usuario. Luego de esto se genera el patrón o huella de impresión que da lugar a las líneas de conexión y capas físicas de la placa. Finalmente se generan archivos de formato estándar que pueden ser extraídos para llevar a la línea de fabricación de prototipos.

- **Proyecto de FPGA (*.PRJFPG):** Es el conjunto de documentos que deben ser procesados para programar una FPGA. En este tipo de proyecto se parte de un esquemático en el cual se colocan todos los componentes necesarios para realizar el diseño, estos componentes pueden ser los propios de Altium y de la NanoBoard 3000 así como pueden ser diseñados por el usuario mediante el uso de un lenguaje de descripción de hardware como VHDL o Verilog. Adicional a los esquemáticos están los diagramas *OpenBus* en los cuales las conexiones entre elementos se simplifica considerablemente pero se trata de los mismos componentes, el *OpenBus* se puede incluir como una parte del esquemático. En este proyecto también se deben establecer las restricciones de control de funciones o los requerimientos del diseño para ser simulado o sintetizado posteriormente.

- **Proyecto Embebido (*.PRJEMB):** Es el conjunto de archivos necesarios para producir una aplicación de software que corre en un procesador embebido en la FPGA. Este proyecto se enlaza con el proyecto de FPGA pero en dicho proyecto debe existir al menos un microprocesador embebido. Consiste en un *Software Platform* que identifica los dispositivos de hardware presentes en el proyecto de FPGA y brinda los códigos de inicialización para que dichos elementos funcionen adecuadamente, en este proyecto también se debe colocar el código fuente que ejecutará dicho microprocesador.

2.3.3. Sistemas embebidos

Los sistemas embebidos hacen referencia a dispositivos diseñados para realizar una función específica, a diferencia de las computadoras de propósito general que corren programas sobre un sistema operativo y pueden ser empleadas en diversas tareas, los sistemas embebidos ejecutan un *firmware* que no requiere sistema operativo alguno y trabajan en tiempo real. El *firmware* por lo general está diseñado en lenguaje ensamblador, C o C++ dependiendo de su aplicación.

Los sistemas embebidos abarcan dispositivos como: microprocesadores, microcontroladores, memorias, módulos de comunicación, módulos de relojes o módulos de alimentación del sistema.

2.3.4. Microprocesador Embebido TSK3000

El TSK3000 es un *soft processor* o núcleo pre-verificado y pre-sintetizado de un procesador de 32 bits de arquitectura RISC compatible con el bus de datos *Whisbone* que puede ser implementado en un proyecto de FPGA. En la Figura 2.15 se muestra el diagrama de bloques del TSK3000 [16].

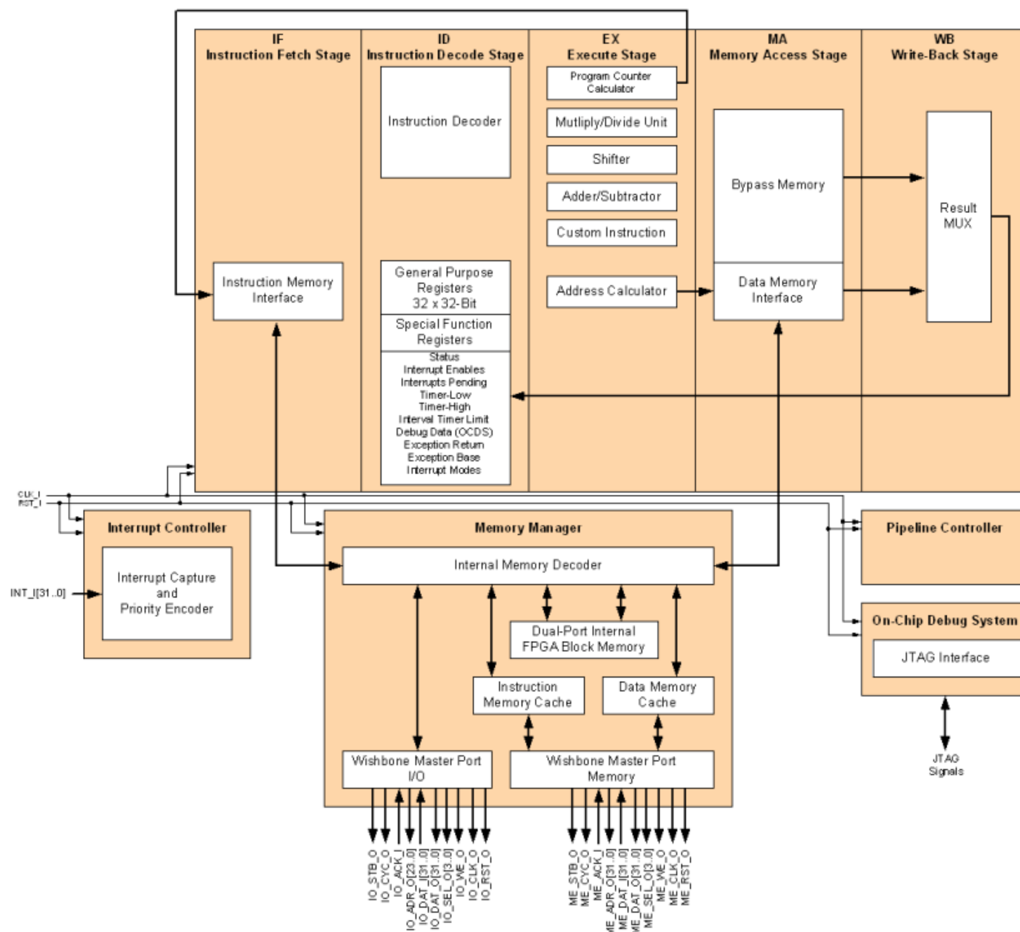


Figura 2.18: Diagrama de bloques del microprocesador TSK3000 [16].

Principales características del TSK3000:

- Instrucciones de 32 bits que se ejecutan en un ciclo de reloj.
- Entradas y salidas a través de bus *Whisbone*.
- Bloque de memoria RAM de doble acceso real que va desde 1kB hasta 1MB.
- 4GB de espacio de direcciones.
- Arquitectura interna tipo Harvard con acceso a memoria externa simplificado.
- Compatible con compiladores C, C++ y *assembler*.
- 32 interrupciones de entrada.
- Multiplicador de hardware 32x32 hasta 64 bits con y sin signo.
- Divisor de 32x32.

2.3.5 *Device View*

El *Device View* (ver Figura 2.18) es una ventana disponible en Altium en la cual se interactúa con la NanoBoard conectada al computador, permite configurar un determinado proyecto en la FPGA, y programar la FPGA pasando por las etapas de compilado, sintetizado, construcción y progamación. Además permite interactuar con los instrumentos virtuales una vez que la FPGA se encuentra programada. A continuación se describen las etapas que se deben realizar secuencialmente para programar la FPGA:

- **Compilado:** Se busca cualquier error de código o eléctrico en cada documento que compone el proyecto.

- **Sintetizado:** Los documentos fuente del proyecto son traducidos a VHDL y se verifica su posibilidad de implementación en la FPGA ya que se puede presentar casos en los cuales se tengan instrucciones o dispositivos que compilan pero no son sintetizables en la FPGA.

- **Construcción:** Se genera una llamada al software Quartus que es propio de Altera (fabrican de de la FPGA de la NanoBoard) y este se encarga de establecer y fijar las rutas físicas que se deben configurar en los componentes lógicos de la FPGA para que se obtenga el comportamiento definido en los documentos fuente.

- **Programación:** Es el paso final y una vez completado los tres anteriores, se tiene la certeza de que el diseño funcionará y por lo tanto es programada la FPGA.

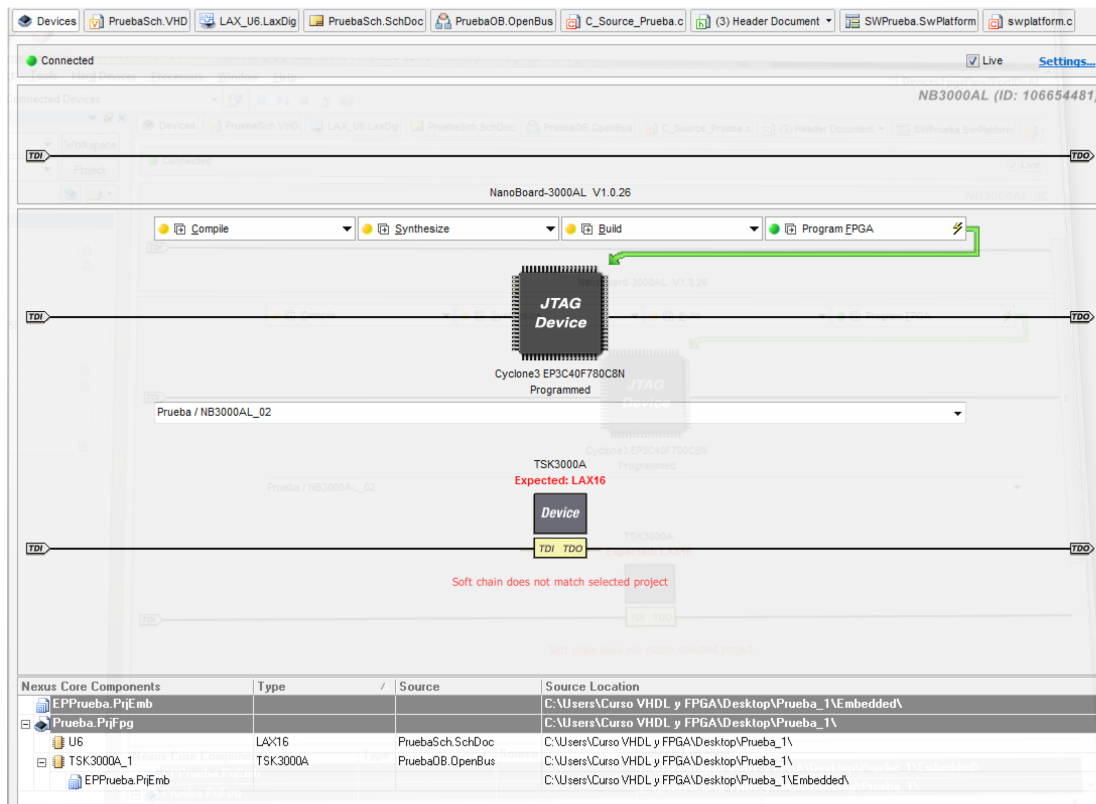


Figura 2.19: Ventana *Device View* de Altium.

2.3.5. VHDL

VHDL es un lenguaje de descripción de circuitos electrónicos digitales que utiliza distintos niveles de abstracción. Como su nombre lo indica, permite acelerar el proceso de diseño. Es importante destacar que VHDL no es un lenguaje de programación, solo permite describir circuitos síncronos y asíncronos.

En la actualidad se rige por un estándar aprobado por la IEEE y es útil para la realización de simulaciones, además existen herramientas que transforman una descripción en VHDL en un circuito real (procedimiento llamado síntesis), como es el caso de una FPGA. Una característica particular de VHDL es que permite realizar una descripción de la estructura del circuito como la especificación de la funcionalidad de un circuito utilizando formas familiares a los lenguajes de programación.

La estructura de un archivo fuente en VHDL está dividida en dos partes: la entidad y la arquitectura, en la arquitectura es donde se describen las sentencias que indican e comportamiento del circuito, a este modelo de programación se le conoce como *behavioral.* Existe otro tipo de descripción llamado estructural en el cual se interconectan los diferentes componentes entre sí mediante sentencias adecuadas, cada componente debe estar previamente definido para llevar a cabo una función específica [17].

2.3.6. Active-HDL

Active-HDL es un entorno de diseño basado en Windows que permite realizar diseños en lenguaje de descripción de hardware para FPGA, simulación y en algunas casos implementación siempre que la FPGA sea soportada (Altera, Atmel, Lattice, Microsemi, Quicklogic, Xilinx). El IDE de Active-HDL incluye herramientas gráficas y una suite completa que hace la tarea al desarrollador mucho más intuitiva y es ideal para el desarrollo en equipo. Las herramientas de Active-HD permiten seguir los pasos lógicos para realizar un diseño en FPGA: diseño, síntesis y la implementación con las rutas definidas [19].

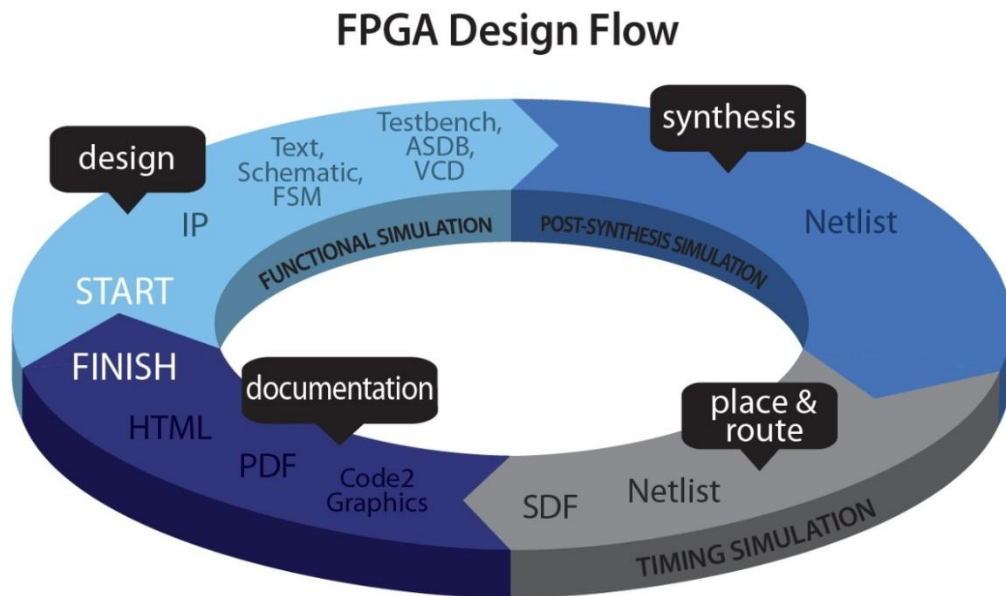


Figura 2.20: Flujo de diseño en FPGA [19].

2.4. Visual Studio

Visual Studio es un conjunto de herramientas de desarrollo para la construcción de aplicaciones web, de escritorio y aplicaciones móviles para dispositivos Android, iOS y Windows. Esto último es posible gracias a una herramienta llamada *cross-platform.* Visual Basic.NET, Visual C++ .NET, Visual C# .NET y Visual J# .NET son los lenguajes de programación de la plataforma .NET Framework propia de Visual Studio y todos comparten el mismo IDE (Entorno de Desarrollo Integrado), esto les permite compartir herramientas y facilita la creación de soluciones en varios lenguajes [20].

2.4.1. .NET Framework

.NET Framework es un entorno de ejecución administrado que proporciona diversos servicios a las aplicaciones en ejecución. Está compuesto por un motor de ejecución CLR (*Common Language Runtime*) que controla las aplicaciones en ejecución, y la biblioteca de clases de .NET Framework que proporciona funciones de código probado que pueden llamar los desarrolladores desde sus propias aplicaciones. .NET Framework ofrece los siguientes servicios a las aplicaciones en ejecución [21]:

- **Administración de la memoria:** El CLR libera a la aplicación de ésta responsabilidad.

- **Sistema de tipos comunes:** Los tipos básicos los define el sistema de tipos de .NET y no el compilador, por lo que los tipos son comunes a todos los lenguajes.

- **Biblioteca de clases extensa:** No es necesario escribir código para controlar operaciones comunes de bajo nivel, la biblioteca de clases de .NET Framework nos proporciona estas funciones accesibles en todo momento.

- **Marcos y tecnologías de desarrollo:** Ofrece ASP.NET para aplicaciones web, ADO.NET para el acceso a los datos y Windows *Communication Foundation* para las aplicaciones orientadas a servicios.

- **Interoperabilidad de lenguajes:** Los compiladores de los lenguajes de la plataforma .NET Framework generan un Lenguaje Intermedio Común (CIL) que se compila en tiempo de ejecución a través de *Common Language Runtime*.

- **Compatibilidad de versiones:** La gran mayoría de aplicaciones que se desarrollan con una versión determinada de .NET Framework se ejecutan sin problemas en versiones anteriores.

- **Ejecución en paralelo:** .NET Framework permite que varias versiones de CLR coexistan en el mismo equipo, esto quiere decir que también pueden coexistir varias versiones de la misma aplicación.

- **Compatibilidad con múltiples versiones (*multi-targeting*):** La biblioteca portable de .NET Framework permite crear ensamblados que funcionen en las siguientes plataformas: Windows 7, Windows 8, Windows 8.1, Windows 10, Windows pone y Xbox 360.

2.4.2. Visual C# .NET

Visual C# es un lenguaje orientado a objetos simple, elegante y con seguridad en el tratamiento de tipos, basa su sintaxis en C/C++ de modo que permite a los desarrolladores de estos lenguajes hacer desarrollos rápidos en la plataforma .NET sin sacrificar la potencia y el control que constituyen el sello de C y C++. C# permite desarrollar aplicaciones en la plataforma .NET incluyendo servicios Web y aplicaciones ASP.NET de forma rápida y fácil, además, está completamente integrado con .NET Framework y *Common Language Runtime*, que conjuntamente proporcionan interoperabilidad del lenguaje, recolección de elementos no utilizados, seguridad ampliada y compatibilidad de versiones mejorada. C# admite herencia única y crea lenguaje intermedio de Microsoft (MSIL) como entrada de código nativo, también proporciona acceso a los tipos de API más comunes: .NET Framework, COM, Automatización y estilo C. Asimismo, admite el modo *unsafe*, en el que se pueden utilizar punteros para manipular memoria que no se encuentra bajo el control del recolector de elementos no utilizados.

C# simplifica y moderniza algunos de los aspectos más complejos de C y C++, como los espacios de nombres, las clases, las enumeraciones, la sobrecarga y el control estructurado de excepciones, también elimina ciertas características de C y C++ como macros, herencia múltiple y clases base virtuales [22].

CAPÍTULO III

MARCO METODOLÓGICO

En este capítulo se describe la metodología empleada para la conceptualización y el desarrollo del proyecto. La comunicación Computador Personal - NanoBoard 3000 requiere una serie de conocimientos que van desde lenguajes de programación de alto nivel, pasando por el USB hasta llegar a la implementación de hardware en una FPGA, por este motivo, en primer lugar se realizó un estudio profundo de los aspectos teóricos involucrados y de las herramientas disponibles, estos conocimientos permitieron plantear un propuesta de diseño y desarrollarla tomando las decisiones adecuadas para llevar a cabo de manera satisfactoria la comunicación Computador Personal - NanoBoard 3000.

3.1. Estudio de los protocolos y fundamentos teóricos

Para el diseño de la interfaz, en primer lugar se procedió a estudiar el estándar técnico de televisión digital terreste ISDB-Tb para entender el papel que cumple la interfaz a desarrollar dentro del esquema de una planta de transmisión de televisión digital y su relación con otros bloques del sistema.

En segundo lugar se estudió en profundidad el USB, haciendo énfasis en el modo de transferencia isócrono ya que constituye la base de la interfaz de comunicación a diseñar.

Luego se estudiaron los aspectos básicos de una FPGA para entender su funcionamiento y se realizó un estudio del lenguaje descriptivo VHDL para poder realizar implementaciones de hardware en la FPGA con dicho lenguaje.

Para finalizar, se hizo un estudio del lenguaje de programación C# para poder llevar a cabo el diseño de la interfaz de control de transmisión de datos utilizando el USB desde la PC. Esta etapa se extendió por un período aproximado de dos meses debido a la extensión de los tópicos estudiados.

3.2. Estudio de las herramientas computacionales y dispositivos disponibles para implementar la interfaz.

Tener un amplio dominio de las herramientas disponibles fue un punto indispensable para llevar a cabo el proyecto, por éste motivo se realizaron prácticas que permitieron adquirir los conocimientos y destrezas necesarias para el manejo correcto de las interfaces de diseño, librerías, configuraciones y manejo de datos presentes en dichas herramientas de diseño, de esta forma se optimizó el tiempo de desarrollo.

3.2.1. Computador Personal tipo Laptop

Se utilizó un computador con sistema operativo Windows 7 x64 Intel Core i5 con memoria RAM de 6GB como base para el desarrollo del proyecto. En este computador están instaladas las herramientas de software que necesarias y que se especifican en los apartados subsecuentes. El sistema operativo utilizado es Windows 7 de arquitectura x64 debido a que el software Altium Designer DXP y la suite de Cypress sobre la cual se realiza el desarrollo de *firmware* para el SX2 y el FX2 funciona únicamente sobre Windows.

3.2.2. Estudio de VHDL y Active HDL

Se realizó un entrenamiento intensivo del lenguaje VHDL utilizando el software Active HDL para diseñar componentes lógicos en dicho lenguaje y probar su funcionamiento con simulaciones y bancos de pruebas (*test benchs*).

3.2.3. Estudio de las herramientas disponibles en Altium DXP y la NanoBoard 3000

El siguiente paso fue empezar a trabajar con Altium DXP y la NanoBoard 3000, se implementaron componentes definidos anteriormente mediante VHDL (con Active HDL) en la FPGA y se verificó su funcionamiento, también se implementaron distintos proyectos a modo de práctica para poner en funcionamiento distintos componentes de la NanoBoard 3000 entre los cuales destacan: LEDS, botones, *user headers*, bloques de memoria RAM, componentes lógicos, relojes del sistema, pantalla TFT, puertos USB tipo A (*host*) y tipo mini-B (esclavo), microprocesador embebido TSK3000 y el analizador lógico. Esto se realizó mediante la creación de proyectos que involucran las interfaces de diseño: *Schematic, OpenBus, Software Platform* y código fuente en leguaje C para el TSK3000.

Para proceder con la habilitación del puerto USB tipo mini-B incorporado en la NanoBoard 3000 se estudió el dispositivo EZ-USB SX2. También se estudió el dispositivo EZ-USB FX2 que es más reciente y posee mayor documentación que puede ser utilizada a modo de referencia para configurar el SX2.

Durante esta etapa se detectó un error en la definición de la función usb_open() asociada al *Software Platform* (Interfaz de software de proyecto embebido de Altium) ésta función inicializa el dispositivo EZ-USB SX2 asociado al puerto USB disponible en la NanoBoard 3000, dicha falla no permite habilitar el puerto USB tipo mini-B disponible en la NanoBoard 3000. Mediante una revisión de

los documentos oficiales de Altium [23], se pudo verificar que esta falla se presenta en la versión de Altium de la cual se posee licencia (Build 10.391.22084), en versiones posteriores (a partir de la versión 10.1051.23878 – 27/04/2012) fue corregida pero no se tiene acceso a dicho software ya que está sujeto a la adquisición de una licencia actualizada, motivo por el cual se decidió hacer uso de la tarjeta de desarrollo Cypress CY7C68013A EZ-USB FX2LP.

3.2.4. Estudio de la tarjeta de desarrollo Cypress CY7C68013A EZ-USB FX2LP

El dispositivo EZ-USB FX2 es muy similar al EZ-USB SX2, su principal diferencia es que incorpora un CORE actualizado del microcontrolador 8051 que actúa como maestro en determinadas circunstancias y permite realizar la configuración inicial del dispositivo e incluso manejar los datos provenientes del USB, en el SX2 el maestro es el TSK3000 tanto para configuración como para el flujo de datos.

Para que el desarrollo sobre éste dispositivo se pueda aplicar posteriormente al SX2 se planteó utilizar el 8051 solamente para realizar la configuración inicial y la carga de descriptores, ya que el FX2 puede pasar los datos de entrada automáticamente a la salida sin intervención alguna del 8051 permitiendo de ésta forma, mediante las conexiones adecuadas, que el TSK3000 sea el maestro del FX2 una vez configurado y se encargue de manejar el flujo de datos proveniente del computador personal.

Estos criterios de diseño se llevan a cabo teniendo en cuenta que cuando se adquiera la licencia de una versión actualizada de Altium Designer, se podrá migrar del FX2 al SX2 fácilmente.

3.2.5. Estudio de C#, librería CyUSB.dll y Visual Studio 2013

Se realizó una investigación y una serie de tutoriales para adquirir conocimientos sobre el manejo del entorno de desarrollo de software Visual Studio y la plataforma .NET empleando el lenguaje de programación orientado a objetos C#. Esto abarca la creación de proyectos, soluciones, formularios, funciones, creación de eventos, objetos, manejo de clases, uso de librerías, comunicación con puertos COM y USB a través del *driver* CyUSB3.sys. La plataforma escogida es Visual Studio debido a que Cypress pone a disposición las librerías CyAPI.lib y CyUSB.dll e indica que están especialmente desarrolladas para ser utilizadas con Visual Studio. Dichas librerías contienen las funciones necesarias para establecer comunicación bidireccional vía USB con el FX2 y el SX2.

3.3. Esquema de diseño para realizar la comunicación Computador Personal - NanoBoard 3000

Con el conocimiento adquirido a través del estudio de los protocolos y fundamentos teóricos así como también de las herramientas y equipos disponibles, se realizó un esquema de diseño que permitió desarrollar la comunicación Computador Personal-NanoBoard 3000. Dicho esquema es presentado en la Figura 3.1 se aprecian 3 bloques interconectados entre sí que se describen a continuación en orden descendente:

- **Computador Personal:** El computador se utiliza para ejecutar la aplicación de control de transmisión de datos desarrollada en C# que se encarga de enviar datos a través de un puerto USB tipo A para su recepción en el dispositivo USB esclavo, esto es posible gracias a funciones predefinidas incluidas en la librería CyUSB.dll de Cypress. Además del envío de datos, también tiene como función enviar solicitudes definidas en el estándar USB al

dispositivo esclavo, recibir los datos asociados a dichas solicitudes, asignar un ancho de banda específico y alimentar al dispositivo esclavo.

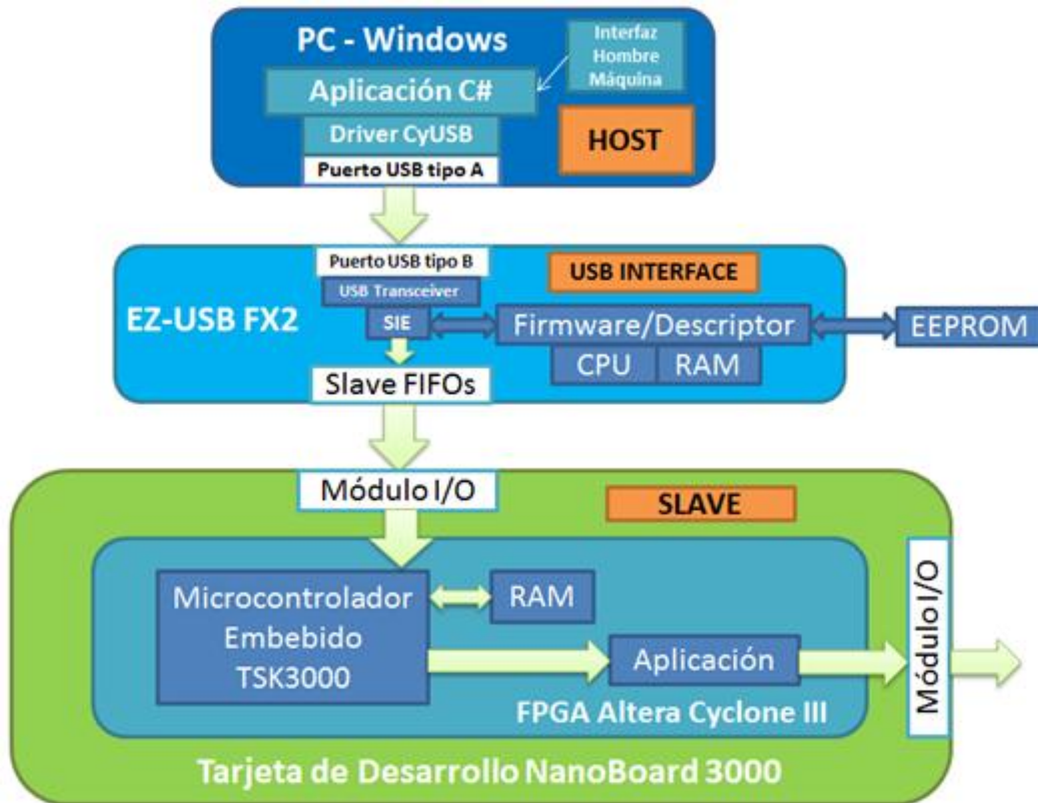


Figura 3.1: Esquema de Diseño.

- **Tarjeta de desarrollo EZ-USB FX2LP:** Tiene como función manejar el USB: cargar descriptores y procesar las solicitudes realizadas por el *host* vía *endpoint* cero. También se encarga de la recepción de datos a través del puerto USB tipo mini-B y de la adaptación de dichos datos para su salida a través de los buses FIFO en formato paralelo de 8 bits, los FIFOs están disponibles en los *pin headers* del dispositivo, específicamente en los puertos B y D (este último solamente se utiliza en la configuración de 16 bits). Para llevar a cabo dichas tareas se procede con el diseño de un *firmware* específico para el FX2LP teniendo en cuenta que se requiere el modo de transmisión isócrono y

así obtener la mayor velocidad de transmisión posible. La distribución física de los pines del FX2LP se muestra en la Figura 3.2.

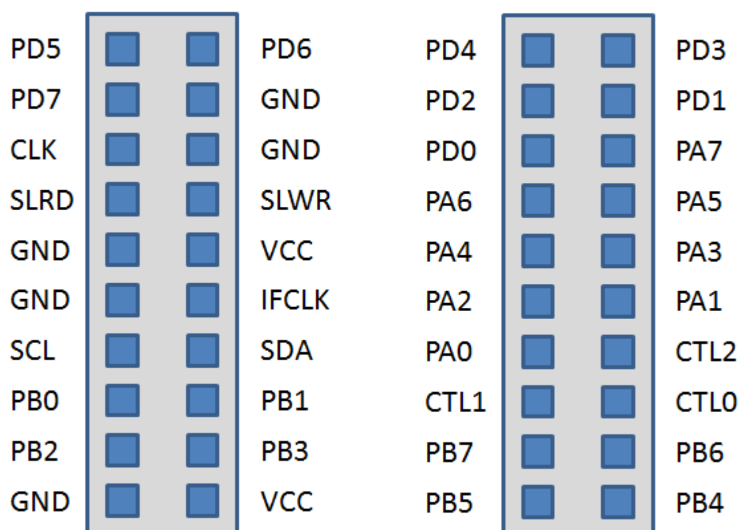


Figura 3.2: Pines disponibles en el FX2LP.

Tabla 3.1: Pines FIFO.

Pines Físicos	Pines FIFO	Descripción
PB[7:0]	Datos	Bus de datos 8bits
SLRD	SLRD	Señal de lectura esclavo
SLWR	SLWR	Señal de escritura esclavo
CTL0	PG	Bandera programable
CTL1	FF	Bandera búfer lleno
CTL2	EF	Bandera búfer vacío
PA2	FIFOADR0	Bits selectores del FIFO
PA4	FIFOADR1	
PA6	PKTEND	Fin del paquete
PA7	SLCS#	Habilitador

- **Tarjeta de desarrollo NanoBoard 3000:** Su función es ejercer de maestro del FX2LP, se encarga de proveer las señales que controlan la salida de datos hacia los FIFO del FX2LP, estos datos provenientes del FIFO del FX2LP son leídos a través de los puertos de entrada del TSK3000 mediante la adecuada conexión entre los pines de salida del EZ-USB FX2LP y el *user header* A disponible en la NanoBoard 3000. Posteriormente estos datos se transmiten a través de los puertos de salida del TSK3000 para estar disponibles como salida en el *user header* B de la NanoBoard3000.

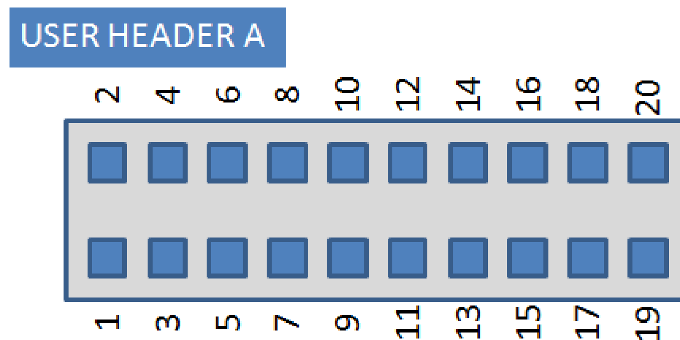


Figura 3.3: Numeración de los *user headers* disponibles en la NanoBoard 3000.

Las conexiones necesarias para la correcta transmisión de datos entre la placa FX2LP y la NanoBoard 3000 se muestran en la Figura 3.4, los pines horizontales representan los *user headers* de la NanoBoard 3000 y los verticales los *pin headers* de la placa FX2LP. La comunicación se implementa con un bus de 8 bits ya que no hay suficientes pines disponibles para implementar uno de 16 bits. El *user header* A se utiliza solo para procesar entradas y el *user header* B para las salidas.

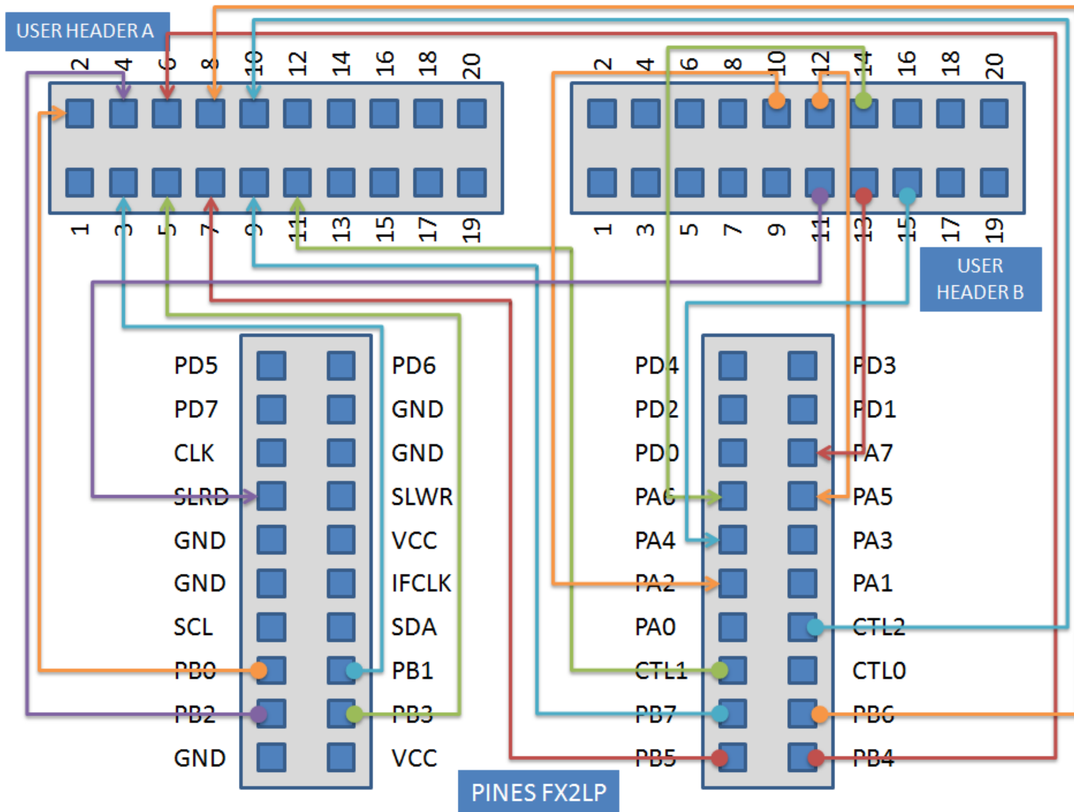


Figura 3.4: Esquema de conexiones entre el bus FIFO del FX2LP y los *user headers* de la NanoBoard3000.

La Figura 3.5 muestra las salidas de la NanoBoard 3000 conectadas a unos diodos LED a modo de montaje experimental para observar las salidas, posteriormente estas salidas irán conectadas a circuitos amplificadores y adecuadores de señal para su transmisión a través de una antena.

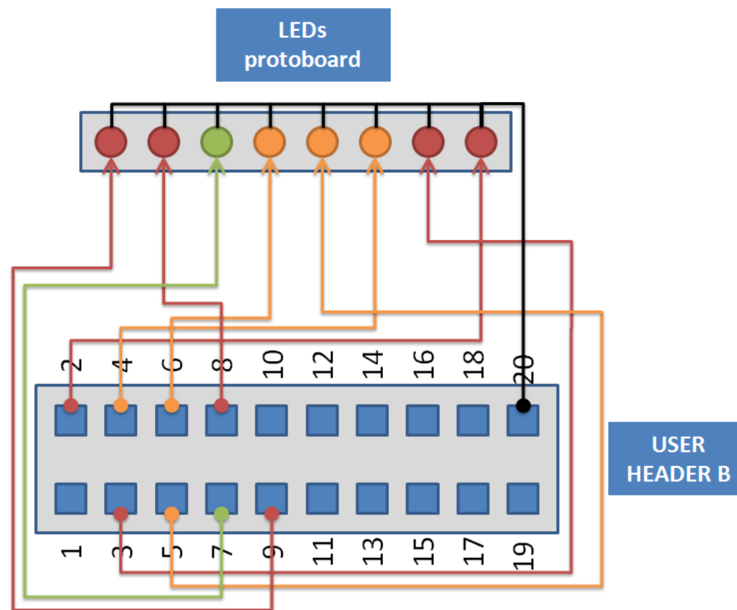


Figura 3.5: LEDs conectados a los pines de salida de la NanoBoard 3000.

3.4. Pruebas de funcionamiento

Para comprobar el funcionamiento de las diferentes etapas que conforman el diseño fue necesario realizar las pruebas que se describen a continuación:

3.5.1. Pruebas realizadas a la aplicación de control de transferencias

1. Se probó la velocidad de salida de datos por software, implementando un indicador de velocidad de transferencia de datos en la aplicación.
2. Se comprobó el reconocimiento del dispositivo FX2LP en la aplicación de acuerdo a los descriptores cargados en el *firmware* del mismo al conectarlo a los puertos USB del computador.

3. Una vez conectado y reconocido por la aplicación, se probó de forma satisfactoria la transferencia de datos hacia el dispositivo esclavo a velocidades adecuadas.

3.5.2. Pruebas realizadas a la tarjeta EZ-USB FX2LP

1. Se probó satisfactoriamente la entrada de datos por medio del USB proveniente de la aplicación de control de transferencias.
2. Se probó satisfactoriamente la transferencia de datos desde el puerto USB hacia los FIFOs.
3. Se probó satisfactoriamente el funcionamiento de las banderas que indican el estado de los FIFOs: vacío y lleno.
4. Se probó satisfactoriamente el funcionamiento de las señales de control para la salida de datos en los FIFOs.

3.5.3. Pruebas realizadas a la NanoBoard 3000.

1. Se Probó satisfactoriamente la recepción de datos en los puertos de entrada del TSK3000.
2. Se Probó con éxito la recepción de datos en los *user headers.*
3. Se Probó satisfactoriamente la salida de datos en los puertos de salida del TSK3000.
4. Se Probó satisfactoriamente la salida de datos y señales de control en los *user headers.*

CAPÍTULO IV

DESARROLLO DE LA INTERFAZ

4.1. Etapa 1: Aplicación para control de datos en el computador personal

La aplicación para el control de datos fue diseñada en Visual Studio empleando el lenguaje C#, esta aplicación se encarga de reconocer el dispositivo FX2LP, también se encarga de procesar las solicitudes necesarias a través del *endpoint* cero mediante transacciones de control para cargar descriptores y toda la información necesaria para el correcto funcionamiento del FX2LP, además filtra los dispositivos conectados mediante el *Product* ID y el *Vendor* ID de modo que solo se comunique con el dispositivo de pruebas, una vez logrado esto, se puede seleccionar el *endpoint* al cual se transmitirán los datos, los datos a transmitir, la cantidad de transferencias en cola y la cantidad de paquetes por transferencia. La aplicación muestra en pantalla la velocidad de transmisión en kilobytes por segundo (kB/s) y también el uso del CPU para que el usuario lo tenga presente a cada momento ya que las transmisiones USB a gran velocidad exigen al procesador y pueden llevar al equipo a uso excesivo del CPU aumentando la temperatura del mismo y mermando el rendimiento. El *driver* CyUSB.sys permite controlar el dispositivo de Cypress, para esto se instaló dicho *driver* en el computador y se añadió la librería CyUSB.dll a las referencias del proyecto en Visual Studio, dicha librería constituye la interfaz entre el driver y la plataforma .NET como se aprecia en la Figura 4.1.1.

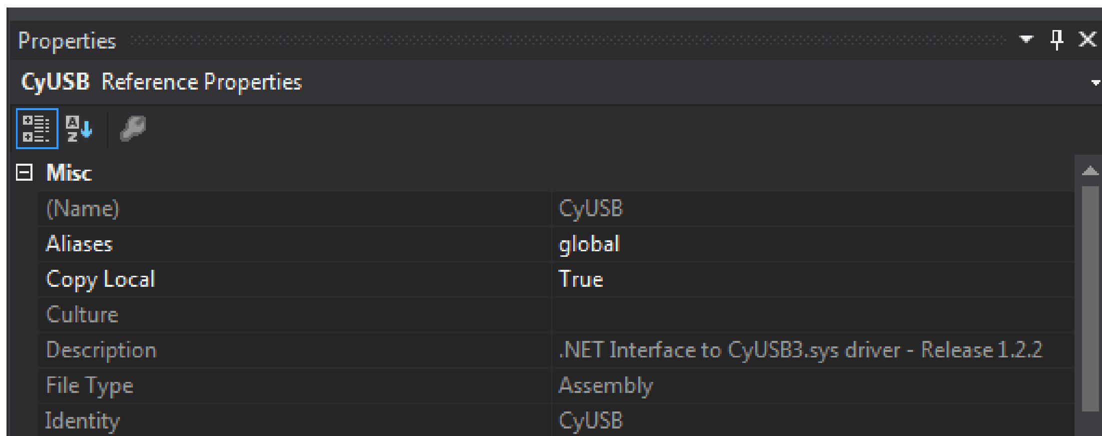


Figura 4.1.1: Propiedades de la referencia CyUSB en Visual Studio.

4.1.1. Interfaz Gráfica

En la Figura 4.1.2 se observan dos capturas de pantalla de la aplicación: la imagen de la izquierda muestra la apariencia normal de la aplicación y en la imagen de la derecha hay un número asociado a cada componente de la interfaz gráfica. Los componentes del 1 al 4 son ComboBox, los 5, 6 y 9 son TextBox, 7 y 8 son RadioButton, el 10 es un botón y los 11 y 12 son barras de progreso. A continuación se describe cada uno de estos componentes y sus funciones.

El ComboBox es un componente gráfico disponible en Visual Studio y permite seleccionar un elemento de una lista haciendo clic sobre él. Los ComboBox utilizados son:

1. **DevicesComboBox:** A su izquierda muestra la etiqueta: Dispositivo USB, muestra los dispositivos conectados al computador y permite seleccionar uno de ellos.
2. **EndpointsComboBox:** A su izquierda muestra la etiqueta: "*Endpoint*…….", este elemento muestra los *endpoints*, el *alternate setting* al cual pertenecen y permite seleccionar uno de ellos. Estos *endpoints* pertenecen al dispositivo seleccionado en el DevicesComboBox.

Figura 4.1.2: Capturas de la aplicación sin dispositivo conectado.

3. **PpxBox:** A su izquierda muestra la etiqueta: Paquetes por Xfer, esto indica los paquetes de datos que se pueden enviar por cada transferencia, este ComboBox permite seleccionar dicha cantidad de paquetes.
4. **QueueBox:** A su izquierda muestra la etiqueta: Xfers en cola, esto indica la cantidad de transferencias que se pueden colocar en cola para agilizar la transferencia de datos, este ComboBox permite seleccionar dicha cantidad de transferencias.

El TextBox es un componente gráfico disponible en Visual Studio, permite mostrar cadenas de texto y también leer datos alfanuméricos. Los TextBox utilizados son:

5. **SuccessBox:** A su izquierda muestra la etiqueta: Xfers Exitosas, esto indica la cantidad de transferencias que se han realizado de forma correcta. Este

TextBox está bloqueado al usuario, no permite modificar sus valores, solo los muestra.

6. **FailuresBox:** A su izquierda muestra la etiqueta: Xfers Erróneas, esto indica la cantidad de transferencias que se han realizado de forma incorrecta. Este TextBox está bloqueado al usuario, no permite modificar sus valores, solo los muestra.
9. **DatoBox:** Este TextBox se habilita al seleccionar la opción que se encuentra a su izquierda: Byte fijo. Permite enviar un byte constante al dispositivo USB, dicho byte debe ser introducido por el usuario en el DatoBox, no debe ser mayor a 255 (8 bits).

El RadioButton es un componente gráfico disponible en Visual Studio, su función es seleccionar una opción entre varias opciones mutuamente excluyentes que estén disponibles, para esto se emplean dos o más RadioButton y solamente puede estar activo uno en un instante determinado. Los RadioButton utilizados son:

7. **RadioButton1:** A su derecha muestra la etiqueta: Secuencia 1-255-1, esto indica que al seleccionar este RadioButton los datos a transferir hacia el dispositivo USB oscilan entre 1 y 255. Esto permite verificar en el dispositivo si la secuencia se recibe de forma adecuada y permite registrar una eventual pérdida de datos.

8. **RadioButton2:** A su derecha muestra la etiqueta: Byte Fijo, esto indica que al seleccionar este RadioButton los datos a transferir hacia el dispositivo USB son constantes, siempre se transmite el mismo valor. Dicho valor es ingresado por el usuario en el DatoBox.

En la aplicación se encuentra un botón (**StartBtn**), está señalado por el número 10 en la Figura 4.1.2, estos permiten ejecutar acciones definidas en

segmentos específicos de código al realizar clic sobre ellos. El StartBtn tiene como tarea empezar la transmisión de datos e interrumpirla cuando el usuario lo desee. Al no haber iniciado la transmisión, es de color verde y tiene el texto “Start” sobre él, cuando se inicia la transmisión pasa a ser de color rojo y muestra la palabra “Stop” (Ver Figura 4.1.2).

Un ProgressBar es un componente gráfico disponible en Visual Studio y permite visualizar una determinada magnitud de forma dinámica, la magnitud visualizada es directamente proporcional al tamaño de la barra. Los ProgressBar utilizados son:

11. **ProgressBar:** Permite visualizar la velocidad de transmisión de datos vía USB, el 100% de ésta barra está un 30% por encima de los 480Mbps (60000kB/s) máximos que permite el estándar USB 2.0.

12. **CpuBar:** Permite visualizar el porcentaje de uso del CPU del computador.

4.1.2. Código de la Aplicación

El código de la aplicación desarrollada fue escrito en C# y posee la estructura de un programa típico de este lenguaje:

- Inclusión de librerías: Destaca la librería CyUSB.dll de Cypress.
- Inicialización de variables y componentes gráficos
- Inicialización de herramientas
- Función Main
- Funciones secundarias

En la Tabla 4.1.1 se presentan y describen las funciones que componen la aplicación de control de datos.

Tabla 4.1.1: Funciones de la aplicación de control de datos.

Función	**Tipo**	**Descripción**
Main	Void	El *namespace* de la aplicación se llama Streamer, en este *namespace* solo hay un formulario o ventana: Form1, que contiene todo el código del programa a excepción de la inclusión de librerías. La función Main es de tipo **static void** y consiste en un bloque Try/Catch que funciona como un bucle infinito, el bloque Try ordena al Form1 que se ejecute de forma continua mientras que el bloque Catch detecta y maneja cualquier excepción que genere bloque Try. Esto evita el cierre inesperado de la aplicación, en su lugar se muestra un mensaje indicando la excepción ocurrida y su fuente.
SetDevice	Bool	Esta función maneja los dispositivos listados en el DevicesComboBox, asigna los dispositivos conectados a una clase llamada **USBDevice** propia de la librería CyUSB.dll de Cypress, esto permite procesar solicitudes de control entre los dispositivos y el computador. Luego de esto, solicita a los dispositivos el *Vendor* ID (VID) y el *Product* ID (PID) para mostrarlos en el DevicesComboBox. Una vez realizado ese proceso, la función se encarga de crear un objeto de la clase **USBDevice** para controlar el dispositivo que seleccione el usuario y envía una solicitud de *endpoints*, si hay al menos un *endpoint* la función habilita el botón **Start**. En caso de no haber dispositivo alguno conectado se muestra el texto “USB Streamer Cendit – conecte dispositivo” sobre el marco superior de la ventana.
GetEndpointsOf Node	Void	Esta función obtiene los *endpoints* de cada *alternate setting* del dispositivo, luego los añade al EndpointsComboBox y los representa como nodos (estructura ramificada).

usbDevices_Device Removed	Void	Cuando se desconecta el dispositivo y se estaban enviando datos esta función bloquea y finaliza el subproceso que se encarga de enviar datos (tListen) y restablece los componentes del Form1.
usbDevices_Device Attached	Void	Al detectar que se conecta un nuevo dispositivo invoca la función SetDevice.
Dispose	Void	Limpia todos los recursos utilizados por la aplicación cuando ésta se cierra.
Initialize Perfomance Monitor	Void	Inicia los componentes del monitor de rendimiento del CPU y comprueba la versión de Windows, en caso de ser Windows Vista no inicia el monitor debido a que no está soportado.
AboutItem_Click	Void	Función que permite desplegar y mostrar la lista de opciones al presionar el botón de ayuda.
ExitItem_Click	Void	Función que cierra la aplicación.
Form1_Form Closing	Void	Al cerrar el programa esta función bloquea y finaliza el subproceso que se encarga de enviar datos (tListen) y finaliza la función de los dispositivos USB utilizados.
PpxBox_Selected IndexChanged	Void	Responde a la selección de la cantidad de paquetes por transferencia en el PpxBox, calcula el tamaño de los datos a transferir como el *product*o del tamaño máximo de paquete que soporta el *endpoint* y la cantidad de paquetes por transferencia, si este cálculo arroja una cantidad menor o mayor a la permitida según el tipo de transferencia, muestra un mensaje de error y establece el número de paquetes a una cantidad permitida.
EndpointsCombo Box_SelectedIndex Changed	Void	Cada vez que se selecciona un *endpoint* nuevo en el EndpointsComboBox esta función actualiza la interfaz alternativa y los *endpoints* del dispositivo, también asegura la cantidad de paquetes por transferencia llamando a la función PpxBox_SelectedIndexChanged.
DeviceComboBox_	Void	Responde a la selección de dispositivo en el

SelectedIndex Changed		DevicesComboBox, elimina la clase del dispositivo y los *endpoints* para luego invocar la función SetDevice y configurar el nuevo dispositivo seleccionado. Si el dispositivo no posee *endpoints* esta función no hace nada.
StartBtn_Click	Void	Responde al hacer clic sobre el botón Start, si al hacer clic el botón está verde, desactiva todas las herramientas gráficas del formulario excepto el botón que se presionó pero cambia su color a rojo y el texto sobre el cambia a "Stop", una vez hecho esto se comprueba el RadioButto2 para saber que dato enviar al dispositivo, luego calcula el tamaño del búfer necesario y lo asigna al tamaño del *endpoint* por el cual viajarán los datos y activa el subproceso que genera las transferencias a través de la función **XferThread**, si se hace clic de nuevo sobre el botón se activan de nuevo las herramientas y el botón recupera su apariencia original, se cancela la transferencia y el subproceso que permite dicha transferencia.
XferThread	Unsafe Void	Esta función constituye el inicio del hilo de transferencia de datos, mediante un bloque try-catch para manejar la posible excepción que ocurre al estar desconectar el dispositivo en medio de una transferencia de datos. Llama continuamente a la función LockNLoad. Además declara los búferes necesarios para la transmisión de datos a través de los *endpoints* de datos y el *endpoint* de control (cBufs, xBufs, oLaps y pktsInfo), y una variable de tipo entero necesaria para levar la secuencia de los datos, estos parámetros son las entradas de la función LockNLoad. Se utiliza la estructura **ISO_PKT_INFO** para generar el búfer pktsInfo (propia de CyUSB.dll) que contiene un arreglo de datos y el estatus del mismo, esto permite

		determinar si el búfer está preparado o no para enviar los datos, esto permite contabilizar los errores en las transferencias de datos. Los otros búferes son arreglos de tipo Byte.
LockNLoad	Unsafe Void	Constituye una rutina recursiva que se encarga de llenar los búferes de datos (cBufs, xBuf, oLaps y pktsInfo) y referenciarlos mediante un apuntador para transferir dichos datos, además gestiona el llenado de los búferes de acuerdo al tamaño de la cola de transmisiones configurada por el usuario en el QueueBox. Se utiliza una estructura especializada llamada **OVERLAPPED** (estructura de datos propia de CyUSB.dll) que proporciona una asignación estructurada en la estructura de señalización de eventos del sistema operativo, de esta manera se facilita el establecimiento del contenido de los búferes. Cuando los búferes están preparados para iniciar la transmisión, se genera una llamada a la función XferData.
XferData	Unsafe Void	Esta función tiene como entrada los búferes de datos, al entrar por primera vez se resetean los contadores de transferencias erróneas y correctas así como el número de transferencias totales. Luego de esto se inicia la transmisión de datos, mediante un ciclo *for* se va transfiriendo el byte contenido en cada casilla del búfer, para hacer esto se comprueba antes el *status* de dicho búfer, si es adecuado se transfiere el dato y se aumenta en una unidad el contador de transferencias correctas, en caso contrario se salta la rutina de transmisión y se aumenta un una unidad el contador de transmisiones erróneas. Cuando se ha transmitido la totalidad del búfer, se comprueba la cantidad de transferencias correctas y se toma la hora del

		sistema, la diferencia de este instante con el instante en el cual se inicia la transmisión (que es tomado al inicio) y la cantidad de datos transmitidos se utiliza para calcular la tasa de transferencia de datos.
StatusUpdate	Void	En primer lugar, esta función comprueba que el hilo de transferencia esté activo para realizar su función, en caso contrario no hace nada. Luego de esto, maneja los datos que se muestran en la ProgressBar que indica la velocidad de transferencia y también la SuccessBox y la FailuresBox.
ThreadException	Void	Rutina que maneja excepciones del hilo de transmisión, restablece el botón de inicio y la variable que indica el estado del hilo de transmisión.
PerfTimer_tick	Void	Comprueba que no se esté corriendo la aplicación en Windows Vista ya que este no permite el monitor de rendimiento, en caso de estar en Windows Vista, esta función no realiza acción alguna. Se encarga de llamar al CpuCounter del sistema y actualiza dichos valores para ser luego representados en la ProgressBar.
radioButton1_ CheckedChanged	Void	Solamente se encarga de desactivar el DatoBox cuando el RadioButton1 está seleccionado.
radioButton2_ CheckedChanged	Void	Mediante la comprobación de paridad en una variable relacionada con la selección de los RadioButton1 y 2, se encarga de habilitar el DatoBox cuando está seleccionado el RadioButton2. Además, muestra un mensaje indicando al usuario que introduzca el dato en el DatoBox.

En la Tabla 4.1.1 hay tres funciones específicas que se encargan de la transferencia de datos; **XferThread, LockNLoad y XferData.** Debido a su naturaleza de funcionamiento y para tener un control estricto en la transferencia de datos es requerido el uso de punteros, esta herramienta es raramente requerida en C# pero este es uno de los casos en los cuales es necesario su uso. Para hacer uso de los **punteros** se requiere la utilización de un **contexto no seguro**, esto se realiza agregando al tipo de las funciones el prefijo ***unsafe*** [24].

El diagrama de flujo que muestra el proceso realizado por la aplicación desarrollada se presenta en la Figura 4.1.3, en esta figura existe un bloque de inicio mas no uno de fin ya que la aplicación finaliza su ejecución cuando el usuario presiona el botón de cerrar aplicación y esto puede suceder en cualquier instante, por lo que, se trata como una interrupción y se presentan los bloques asociados a esta acción de forma separada en la Figura 4.1.5, con el botón de ayuda sucede lo mismo y su diagrama se presenta en la Figura 4.1.4.

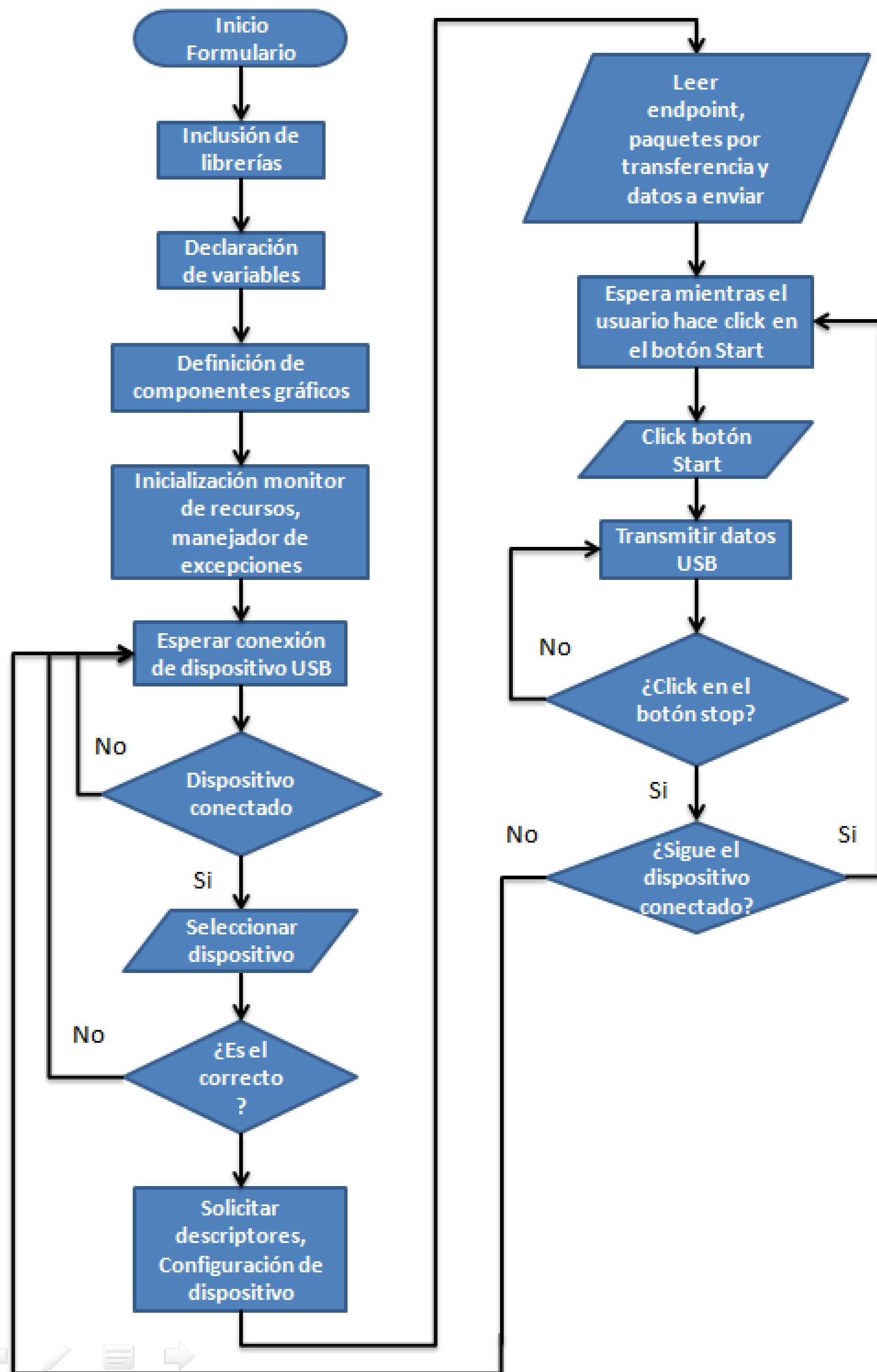


Figura 4.1.3: Procesos de la aplicación de control de datos.

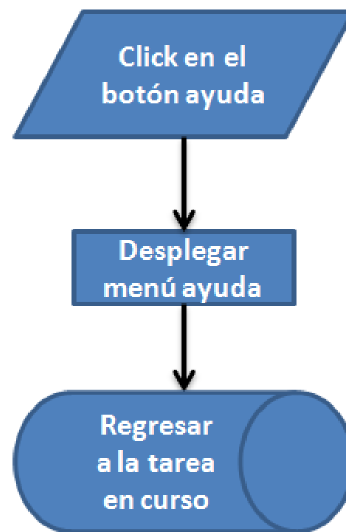


Figura 4.1.4: Acciones al presionar el botón de ayuda.

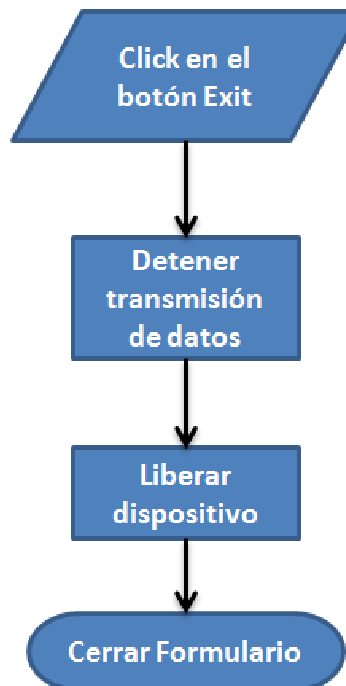


Figura 4.1.5: Acciones al presionar el botón cerrar.

4.2. Interfaz USB implementada en la placa de desarrollo EZ-USB FX2LP

La interfaz USB desarrollada en la placa FX2LP consiste en un *firmware* específico que incluye la tabla de descriptores y los archivos de código fuente necesarios para gestionar las solicitudes de control y la transferencia de datos. Una vez realizada la carga de descriptores, la configuración inicial e iniciada la transferencia de datos, el *firmware* pasa a un segundo plano, siendo el hardware dedicado del FX2LP el encargado del paso de datos del puerto USB hasta el bus FIFO. Para desarrollar dicho *firmware* se utilizó el entorno de desarrollo Keil, este programa se encarga de compilar el *firmware* y generar el código máquina. Los descriptores se realizaron en *assembler* (.a51) y las rutinas de comunicación y configuración se realizaron en C debido a que Cypress proporciona la librería EZUSB.lib en su suite de herramientas, esta librería facilita el desarrollo el *firmware* para el FX2LP, de esta librería se utilizaron los siguientes archivos de cabecera:

- **fx2.h:** Contiene todos los mapas de bits, definiciones, estructuras y macros que definen la estructura del FX2LP incluyendo la estructura de los descriptores.
- **fx2regs.h:** Contiene todas las direcciones de memoria de los registros del FX2LP, a cada dirección le asigna el nombre y el tipo de dato acorde al *datasheet* del dispositivo. También define los pines de cada puerto y los bits individuales de los principales registros mediante mapas de bits.
- **fxsdly.h:** Define retardos de sincronización de diferente duración necesarios en algunos segmentos de código para asegurar el cumplimiento de los tiempos de acceso a determinados registros.
- **intrins.h:** En este archivo de cabecera están los prototipos de algunas funciones utilizadas en los otros archivos de cabecera.

4.2.1. Tabla de Descriptores

La tabla de descriptores se realizó en *assembler* siguiendo la estructura expuesta en el Capítulo II del presente trabajo, pudiendo destacar los siguientes aspectos en cada descriptor:

- ***Device descriptor*****:** Se utilizó el VID y el PID de Cypress para el dispositivo de pruebas FX2LP. Esto se puede cambiar por el VID de la empresa y el PID específico del *product*o.

- ***Device qualifier descriptor*****:** Se fijó el tamaño máximo de los paquetes en 64kB (máximo permitido) y el número de configuraciones en 1.

- ***High-speed configuration descriptor*****:** Se fijó solo una interfaz debido a que con una se pueden abarcar todas las configuraciones de los FIFO necesarias para transmitir de modo isócrono de forma adecuada, cada configuración se define en un campo llamado *Alternate Interface*. El *Alternate Interface* 0 no posee *endpoints* asociados, esto está contemplado en el estándar USB para *endpoints* isócronos. El *Alternate Interface* 1 define un *endpoint* del mayor tamaño posible (3x1024 bytes) y dirección *OUT* para agilizar la transferencia de datos. En este descriptor también se indica que el dispositivo es alimentado por el bus y el consumo de energía máximo se fija en 100mA.

- ***Full-speed configuration descriptor*****:** Se incluye debido a que los dispositivos *high-speed* deben ser capaces de realizar el proceso de enumeración a *full-speed*, indica los mismos datos de energía que en el caso de *high-speed*. En cuanto a *endpoints* el tamaño máximo pasa a ser de 1023 bytes pero sigue siendo isócrono y dirección *OUT*.

4.2.2. Configuración

La configuración del dispositivo se realiza en el archivo de código fuente llamado CYStream.c, en dicho archivo están presentes funciones que son utilizadas por el archivo de código fuente que maneja las solicitudes de control del *host*, en la Tabla 4.2.1 se presentan y describen dichas funciones.

Tabla 4.2.1: Funciones presentes en el archivo CYStream.c.

Función	**Tipo**	**Descripción**
TD_Init	Void	Es la función principal donde se realizan las principales tareas de inicialización de componentes y configuración siguiendo los procedimientos y recomendaciones indicados en el *datasheet* [6] y el manual de referencia técnico [7] del dispositivo FX2LP. Sus tareas son: • Fijar el tipo y frecuencia de funcionamiento del reloj del sistema. En este caso se utiliza el reloj interno a 48MHz. • Configurar el *endpoint* 2 de acuerdo a lo mencionado en los descriptores, esto es: tamaño de 1024 bytes con búfer cuádruple (4Kb en total, ver Figura 4.2.1), esta configuración asegura la transferencia de datos más rápida. • Armar búferes para "cebar la bomba" (acción así descrita en al manual de referencia técnico), de otra manera no se puede iniciar transmisión o recepción alguna. • Configurar la ruta de datos en AUTOOUT, esto permite que los datos que ingresen al FX2LP por el puerto USB estén disponibles en la interfaz

		paralela FIFO de manera instantánea y sin intervención del 8051. • Habilitación de las banderas de llenado y vaciado de los FIFO. • Configuración de la polaridad de los pines que van conectados al maestro externo (SLOE, SLRD, PKTEND, EF, FF, SLCS).
DR_SetConfiguration	Bool	Es llamada cuando se recibe el comando set_configuration, guarda el contenido del bit 2 del registro SETUPDAT en la variable configuration y devuelve TRUE.
DR_GetConfiguration	Bool	Es llamada cuando se recibe el comando get_configuration, envía el contenido de configuration al *endpoint* 0 y devuelve TRUE.
DR_SetInterface	Bool	Es llamada cuando se recibe el comando set_interface, guarda el contenido del bit 2 del registro SETUPDAT en la variable Alternate setting y devuelve TRUE.
DR_GetInterface	Bool	Es llamada cuando se recibe el comando get_interface, envía el contenido de AlternateSetting al *endpoint* 0 y devuelve TRUE.
Interrupciones		
En cada rutina interrupción se limpia el registro de interrupciones mediante la instrucción EZUSB_IRQ_CLEAR() y se limpia la bandera de cada interrupción.		
ISR_Sudav	Void	Indica cuando se ha recibido un paquete de *handshake*.
ISR_Sutok	Void	Indica cuando se ha recibido un *Token Packet*.
ISR_Sof	Void	Interrupción que indica cuando se recibe el paquete SOF (*Start of Frame*).
ISR_Ures	Void	Se activa luego de un *reset*, y luego configura el dispositivo como *high-speed* o *full-speed* según sea el caso y carga los descriptores correspondientes, esto depende del *host*.
ISR_Susp	Void	Se ejecuta cuando el dispositivo entra en estado de ahorro

		de energía, coloca la variable *sleep* en *true*.
ISR_HighSpeed	Void	Esta función configura el dispositivo como *high-speed* o *full-speed* según sea el caso y carga los descriptores correspondientes. Esto sucede cuando el *host* asigna el ancho de banda al dispositivo.

4.2.3. Manejo de solicitudes de control

El manejo de las solicitudes de control se realiza en el archivo de código fuente llamado **fw.c**, este es el archivo principal ya que en él se encuentra la función main. Siguiendo la estructura de un programa en C, este archivo contiene las funciones prototipo de las funciones creadas en CYStream.c y define los punteros de los descriptores que permiten enviarlos al *host* por el *endpoint* cero. Las funciones presentes en fw.c se presentan y describen en la Tabla 4.2.2.

Tabla 4.2.2: Funciones presentes en el archivo fw.c.

Función	Tipo	Descripción
Main	Void	En esta función se llama a la función TD_Init definida en CYStream para inicializar el dispositivo, de ser necesario relocaliza la tabla de descriptores a la memoria interna, activa las interrupciones del USB y del 8051, y decide si el dispositivo debe reenumerar o no, de hacerlo, se encarga de reconectarlo una vez finalizado dicho proceso. Los procesos anteriormente descritos se llevan a cabo en una etapa transitoria que se presenta al conectar el dispositivo y configurarlo, luego de esto la función main entra en un ciclo infinito que espera por la interrupción Sudav para llamar a la función SetupCommand y de esta manera responder adecuadamente a las solicitudes de control del *host*. Dentro de la función main también se encuentra la rutina responsable de colocar al dispositivo en estado de suspensión (bajo consumo de energía) al recibir la bandera que indica esto. Una vez realizado todo el proceso, el 8051 no interviene en la transferencia de datos desde el computador hasta la NanoBoard3000, de eso se encargan los circuitos electrónicos especializados que fueron presentados en el Capítulo II.
Setup Command	Void	Mediante dos *switch-case* del registro SETUPDAT esta función es capaz de identificar y responder a todas las solicitudes de control que el *host* puede realizar al dispositivo esclavo en tiempo real asignando a los registros SUDPTRH y SUDPTRL los datos señalados por los punteros de los descriptores, para las otras solicitudes basta con invocar las funciones creadas para cada solicitud en específico en el archivo CYStream.c.

En la Figura 4.2.1 se aprecian las posibles configuraciones de memoria de los *endpoints* que se pueden realizar al FX2, la configuración utilizada es la número 12 debido a que la transferencia de datos es unidireccional y con un solo *endpoint* se cumple esta tarea además, al tener mayor memoria asignada se agiliza la carga de datos aumentando la velocidad de transferencia.

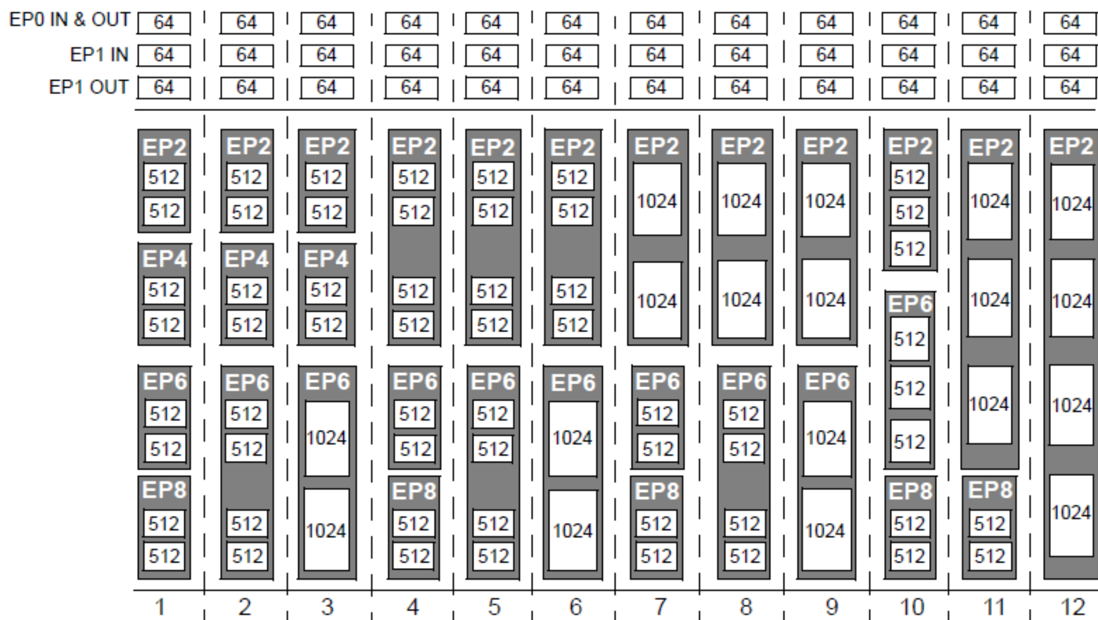


Figura 4.2.1: Configuraciones de *endpoints* admitidas por el FX2LP.

En la Figura 4.2.2 puede observar el diagrama de flujo donde se representa la lógica de funcionamiento del firmware diseñado para el FX2LP.

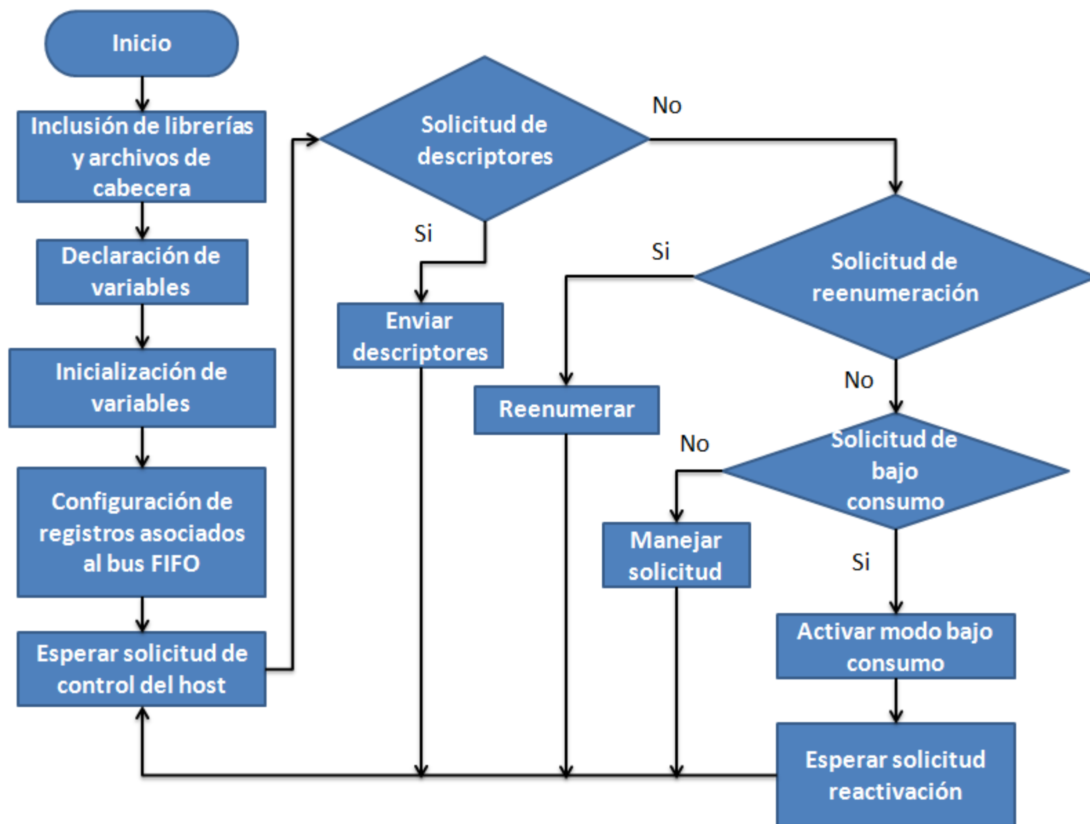


Figura 4.2.2: Funcionamiento del firmware del FX2LP.

4.3. Tarjeta de desarrollo NanoBoard 3000

El desarrollo del *core* FPGA para la NanoBoard3000 se realizó en Altium Designer y está conformado por archivos del sistema *OpenBus*, *Schematic*, y un proyecto embebido para el TSK3000 que a su vez se sustenta con el *Software Platform* y el código fuente en lenguaje C.

4.3.1. *OpenBus*

El TSK3000 constituye el elemento central del diseño del *core* FPGA, este microprocesador embebido puede ser implementado en el esquemático y en el *OpenBus* de Altium Designer pero se decidió implementarlo en el *OpenBus* ya que este permite implementar diseños que en el esquemático pueden llegar a ser muy extensos debido a que en él se representan todas las conexiones físicas entre los distintos dispositivos de forma detallada, el *OpenBus* en cambio, representa dichas conexiones de una manera muy sencilla: mediante flechas. Cada flecha del *OpenBus* representa el conjunto de buses y conexiones individuales necesarias para que el sistema funcione adecuadamente. Los elementos que engloban buses y conductores inidividuales reciben el nombre de *harness.*

El TSK3000 permite varias configuraciones de memoria que van desde 4kB hasta 32kB, en nuestro caso se utilizó la de 32kB ya que las funciones definidas en el *Software Platform* de Altium generan código de gran tamaño aún tratándose de menos de una decena de lineas de código. La ventana de configuración del TSK3000 se muestra en la Figura 4.3.1, se incluyó sistema de *debug* JTAG y una unidad física de mutliplicación y división.

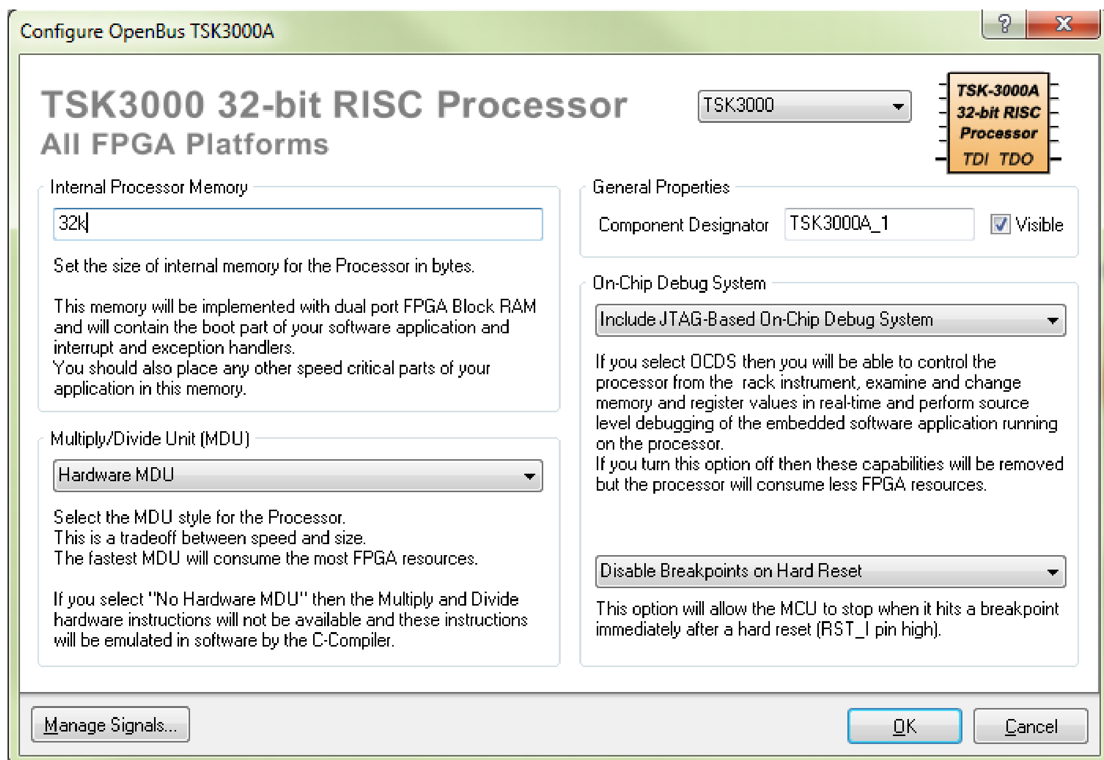


Figura 4.3.1: Configuración del TSK3000 en el *OpenBus.*

En la Figura 4.3.2 se muestran los componentes y conexiones realizadas en el *OpenBus,* el TSK3000 tiene dos componentes *Interconnect* identificados como INTERCON_1 y 2 conectados a sus puertos de memoria y de entradas y salidas, estos elementos permiten conectar toda clase de dispositivos externos al TSK3000 por medio del bus *Whisbone.*

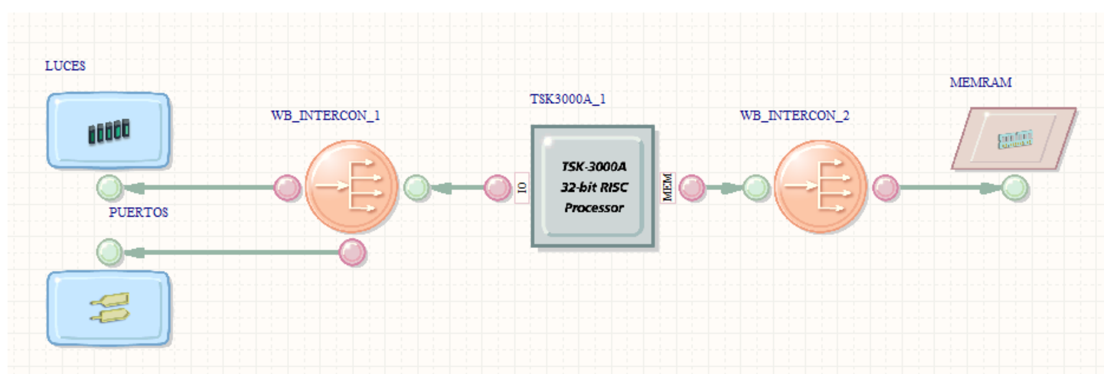


Figura 4.3.2: Diagrama de conexiones del *OpenBus*.

En el bus de memoria del TSK3000 se encuentra la memoria RAM externa identificada como MEMRAM mientras que en el bus de entradas y salidas se encuentra el *driver* de los leds RGB de la NanoBoard 3000 identificado como LUCES y el *driver* de los puertos de entradas y salidas de propósito general del TSK3000 identificado como PUERTOS. El *interconnect* 1 se configura de forma tal que permite que ambos componentes estén conectados a un mismo puerto mediante la multiplexación de los datos, además es capaz de definir asignaciones específicas en el espacio de direcciones del procesador (ver **Figura 4.3.3**) y de controlar el ancho de la direción de decodificación con hardware comparador generado automáticamente.

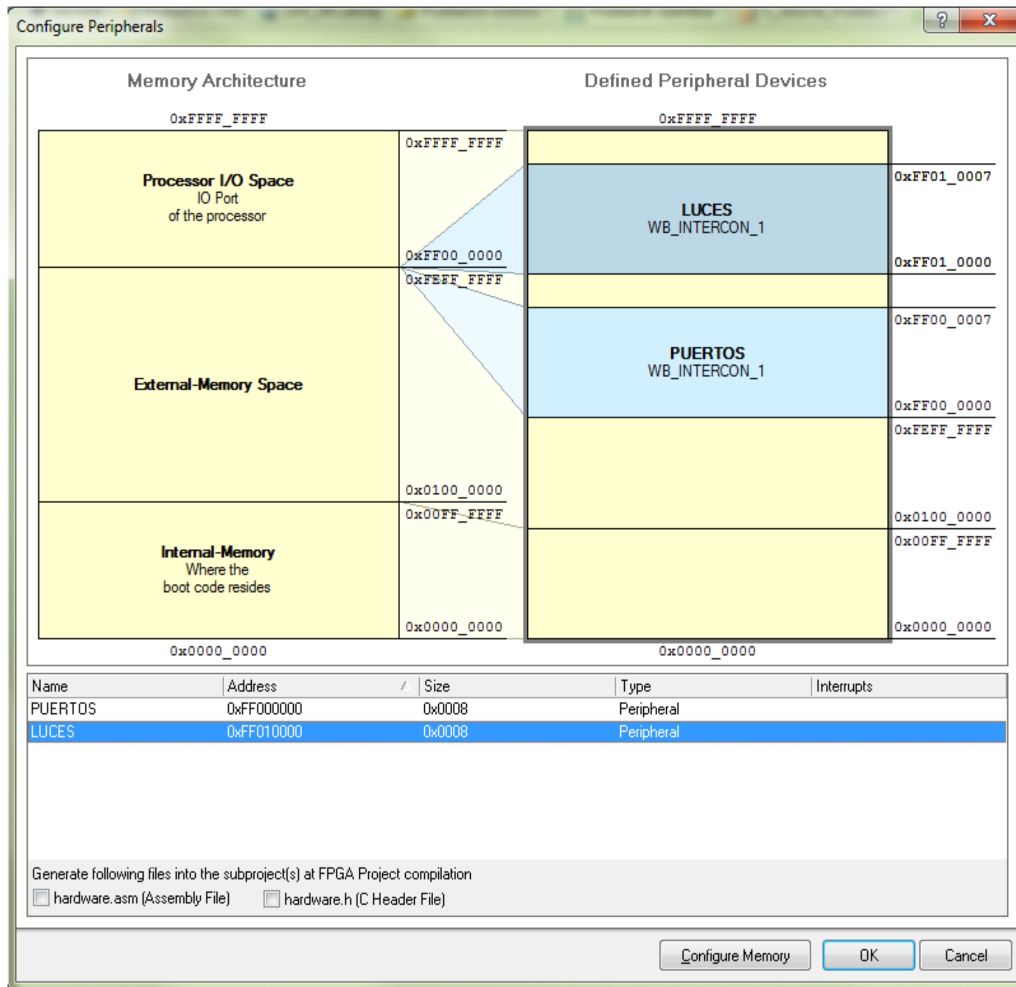


Figura 4.3.3: Direcciones de memoria asignadas a los periféricos en el TSK3000.

El *interconnect* 2 maneja la conexión entre la RAM externa y el TSK3000, la RAM externa puede albergar variables y datos que no quepan en la memoria interna y además funcionar como respaldo de dichas variables si en la memoria interna son reemplazadas por otros datos, en éste caso solamente tiene esa finalidad pero puede almacenar inlcuso el código de la aplicación. En la Figura 4.3.4 se muestran las direcciones de memoria asignadas al TSK3000 y en la Figura 4.3.5 se aprecia la configuración de localización de datos de ambas memorias y el tamaño en bytes de cada segmento de código.

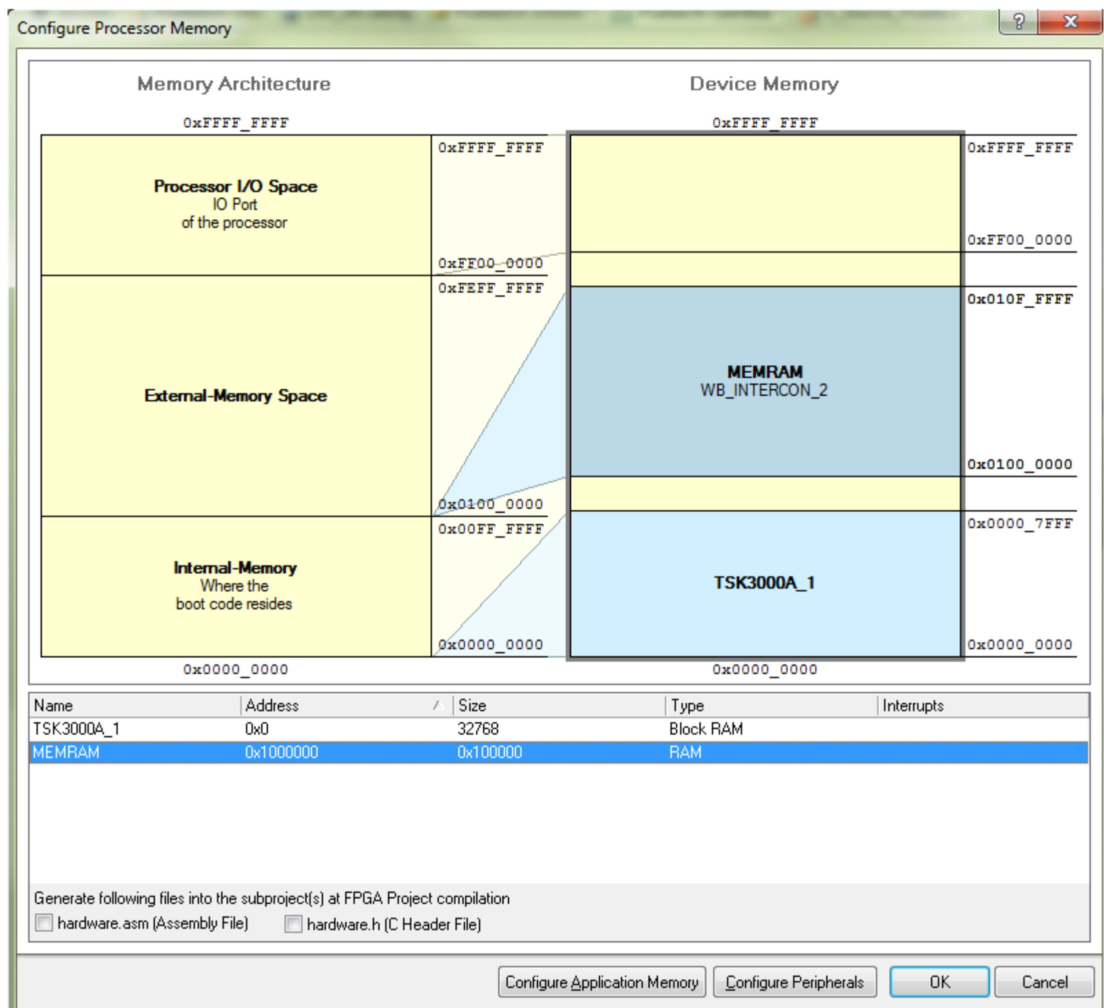


Figura 4.3.4: Direcciones de memoria del TSK3000.

Options for Embedded Project EPPrueba.PrjEmb

Compiler Options | Files With Options | Parameters | Device Sheets | Configure Memory | Locate Options | Sections/Reserved Areas

Sections			Memories		Attributes	
Description	Type	Size	MEMRAM	TSK300...	Copy	No Copy
⊟ Code and Constants				✓	☐	
⊞ Code	.text	3380		✓	☐	
⊞ Constants	.rodata	28		✓	☐	
Startup Code	.text.cstart	176		✓		
⊟ Variables and Data			✓	✓		☐
⊞ Initialized global variables	.data	8	✓	✓		☐
⊟ Cleared (zero) global variables			✓	✓		
⊞ drv_ioport	.bss	12	✓	✓		
⊞ drv_led	.bss	16	✓	✓		
⊞ pal_timers	.bss	96	✓	✓		
⊞ Global variables in small data area	.sdata/.sbss	28	✓	✓		☐
Stack - Local variables and parameter values	stack	32768	✓	✓		
Heap - Dynamically allocated data	heap		✓	✓		
⊟ Special Read-Only Items				✓		
Read-only images of RAM sections	[*]	12		✓		
Startup actions registery	table	80		✓		

Figura 4.3.5: Localización de datos en memoria interna y externa.

El tamaño total del código fuente del TSK3000 es de 36604 bytes, sin embargo se implementó una memoria externa de 1MB para permitir futuros desarrollos al sistema sin preocuparse por espacios de memoria. Esta memoria es de tipo SRAM asíncrona, se seleccionó de este tipo ya que las DRAM necesitan ser refrescadas periodicamente para mantener los datos mientras que las SRAM los mantienen sólo con estar alimentadas. El tamaño total de memoria que puede ser implementado con los bloques disponibles en la NanoBoard 3000 es de 16MB. En la ventana de configuración de memoria que se muestra en la Figura 4.3.6 se observa que el tamaño de memoria es de 1MB y en la pestaña *Memory Layout* se indica que se utilizan dos bloques de memoria con un bus de datos de 16 bits cada uno. En la pestaña *Interface Type* se indica *Signal Harnesses,* esto quiere decir que los buses y los conductores individuales necesarios para manejar las memorias están contenidos en *Harnesses* y de esa forma se representarán en la hoja *Schematic* que se presenta mas adelante.

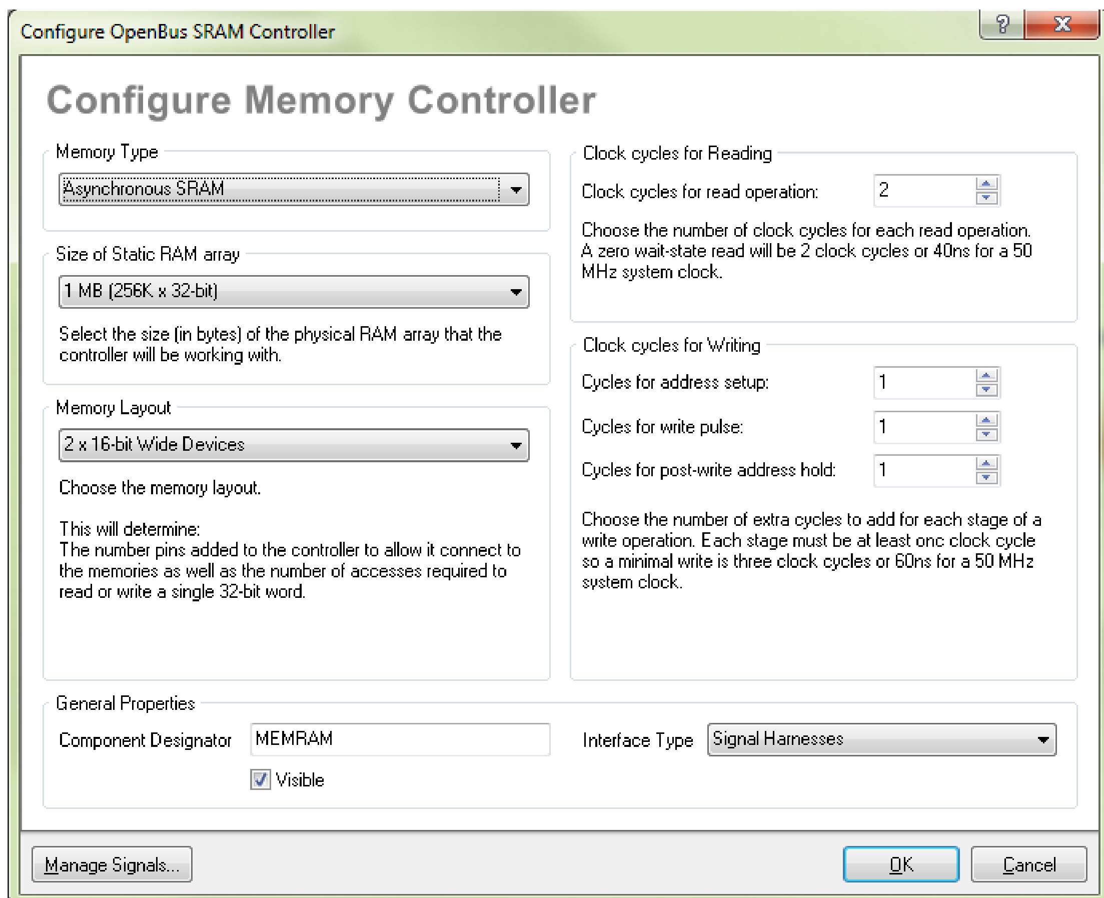


Figura 4.3.6: Ventana de configuración de memoria externa.

Las conexiones que están simplificadas en el *OpenBus* mediante flechas se pueden observar y configurar en el *OpenBus Signal Manager* (ver Figura 4.3.7) , allí se encuentran todas las conexiones requeridas por los periféricos añadidos al TSK3000 en el OpenBus (ver Figura 4.3.2). Las conexiones de los bloques de memoria se representan en símbolos de color azul, esto indica que utilizan *harnesses* mientras que las conexiones del controlador LED y del controlador de los puertos de entrada y salida se representan con un símbolo amarillo, esto indica que utilizan pines y buses tradicionales.

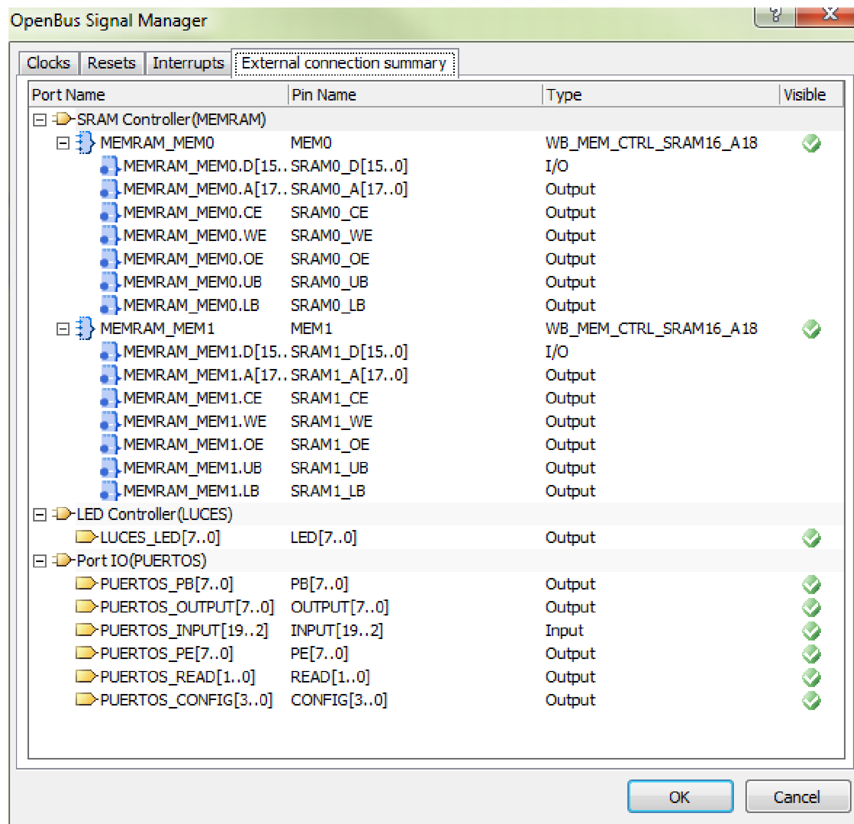


Figura 4.3.7: *OpenBus Signal Manager.*

El controlador de los puertos de entrada y salida permite crear múltiples puertos de distintos tamaños. Cada puerto se creó y se configuró con el fin de recibir y transmitir la información proveniente de los FIFOs de la placa FX2LP (ver Tabla 4.3.1).

Tabla 4.3.1: Puertos creados en el Port IO.

Identificador	**Tamaño**	**Dirección**	**Función**
PB[7..0]	8 bits	Salida	Puerto de prueba.
OUTPUT[7..0]	8 bits	Salida	Salida de datos (User Header B).
INPUT[19..2]	18 bits	Entrada	Entrada de datos FIFO (User Header A).
PE[7..0]	8 bits	Salida	Puerto de prueba.
READ[1..0]	2 bits	Salida	Enviar señales de lectura a los pines SLOE y SLRD de los FIFOs.
CONFIG[3..0]	4 bits	Salida	Seleccionar FIFO, configurar SLCS y PKTEND.

El diseño del sistema *OpenBus* constituye un subsistema del sistema *Schematic* que se presenta en la siguiente sección, debido a esto, tiene entradas que provienen del *Schematic* y las salidas van a este mismo sistema. La señal de *clock* es necesaria para hacer funcionar y sincronizar todos los elementos presentes en el OpenBus (ver Figura 4.3.8) mientras que la señal de *reset* está presente en todos los elementos exceptuando los *interconnect* (ver Figura 4.3.9). Dichas señales provienen de elementos físicos presentes en el *Schematic*.

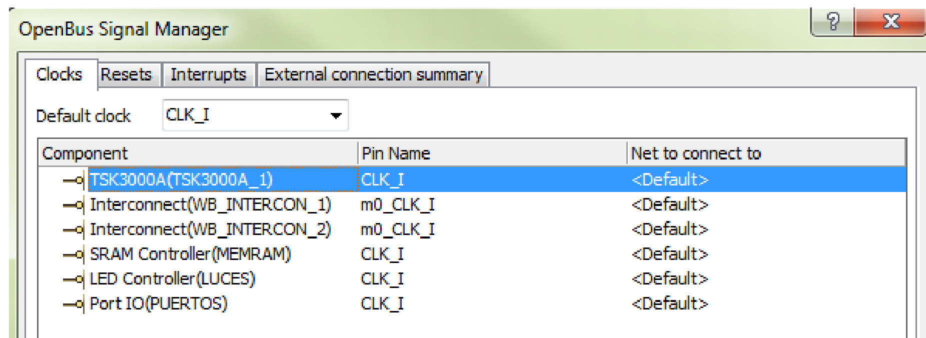


Figura 4.3.8: Elementos del *OpenBus* que reciben señal de *clock.*

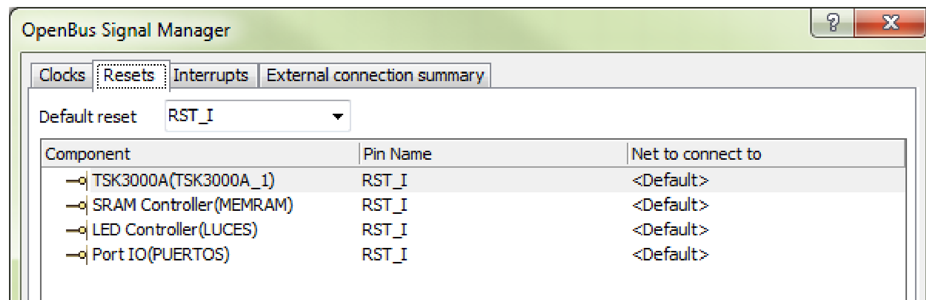


Figura 4.3.9: Elementos del *OpenBus* que reciben señal de *reset.*

4.3.2. *Schematic*

La hoja de diseño Schematic es el documento principal del proyecto FPGA desarrollado en Altium Designer ya que en ella está incluido el sistema *OpenBus* anteriormente descrito, este sistema se reduce a un *sheet symbol* dentro del *Schematic*. En la Figura 4.3.10 se aprecia el *sheet symbol* de color verde que contiene todo el sistema *OpenBus*, en el mismo se aprecian las entradas y salidas de dicho sistema.

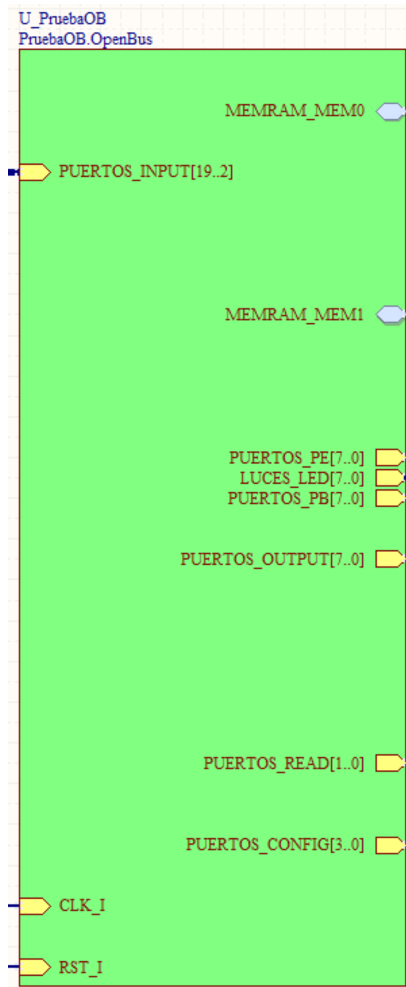


Figura 4.3.10: *Sheet symbol* del sistema *OpenBus.*

Además de la hoja que contiene el *OpenBus*, en el *Schematic* están presentes los bloques de memoria SRAM, los *User Headers* que permiten la entrada y salida de datos, los LEDS RGB, el reloj principal del sistema, el botón de *reset*, un analizador lógico que monitorea las salidas y todas las conexiones entre estos elementos. A continuación se describen cada uno de los sistemas que componen el *Schematic*, el sistema completo se observa en la Figura 4.3.17.

- Memorias SRAM

Estos bloques de memoria son utilizados por el TSK3000 para aumentar su capacidad, se puede obervar que tienen exactamente las mismas conexiones vistas en el OpenBus (Figura 4.3.7), se aprecian las conexiones que entran a los *harnesses*, tanto buses de datos como conexiones individuales y estas son llevadas hacia el *sheet symbol* del *Schematic.*

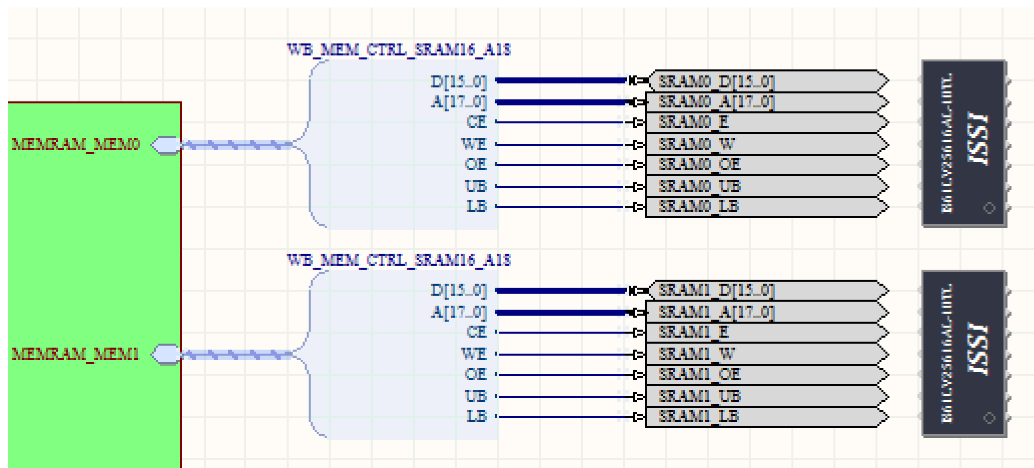


Figura 4.3.11: Conexiones entre *OpenBus* y bloques de memoria SRAM.

- CLK_BRD - TEST_BUTTON - FPGA STARTUP

El símbolo de CLK_BRD representa el reloj del sistema y el TEST_BUTTON representa un botón físico de la NanoBoard3000 que sirve como *reset*. Se hace uso de un elemento llamado FPGA_STARTUP ya que añade un pequeño delay que evita tener que presionar el botón de *reset* cada vez que se sintetiza un diseño y se carga en la NanoBoard3000.

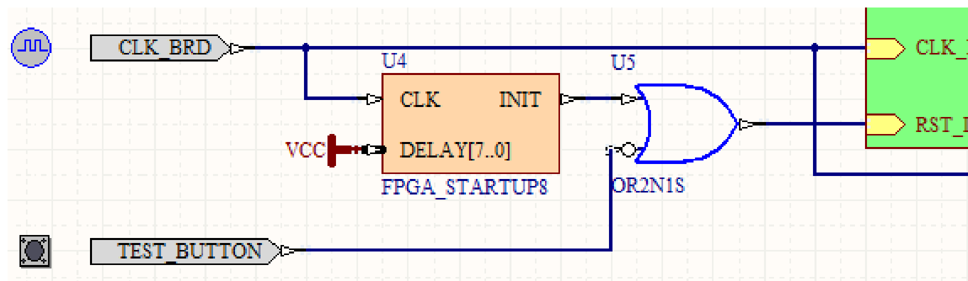


Figura 4.3.12: Conexiones entre CLK-BRD, TEST_BUTTON, FPGA_STARTUP y el *OpenBus.*

- Leds RGB

Leds de tipo RGB que están presentes en la NanoBoard 3000, se utilizan como prueba de los puertos de salida del TSK3000 y para representar los bytes de datos que llegan a la nanoBoard 3000.

Figura 4.3.13: Conexiones entre el *OpenBus* y los leds RGB.

Los *user header* son conectores Berg, también llamados *pin headers*, disponibles en la NanoBoard como pines de entrada y salida de propósito general:

- USER_HEADER_A

El *user header* A es utilizado para recibir los datos y las banderas (EF y FF) provenientes del bus FIFO de la placa FX2LP, en total se utilizan 10 pines ya que los datos se transmiten en paquetes de 8 bits y as banderas ocupan 2 bits. El *user header* A se configura en modo *output* ya que entrega los datos al *Schematic*. La interconexión entre el *user header* A (HA[19..2]) y la hoja del OpenBus se realiza de forma directa ya que el puerto INPUT es de la misma dimensión que el *user header*, aunque no son necesarios todos los pines, se hace de esta forma para evitar errores de compilación. La información entrante es recibida por la hoja del OpenBus a través del puerto INPUT (ver Figura 4.3.14), el TSK3000 se encarga de separar los datos de las banderas y transmitirlos hacia el *user header* B.

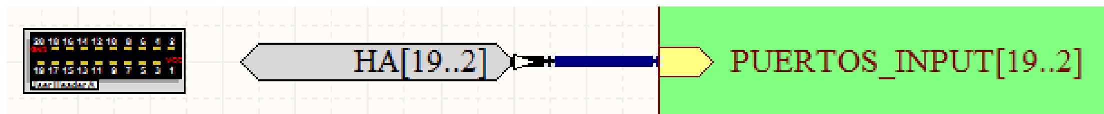


Figura 4.3.14: Conexiones entre el *user header* A y el *OpenBus.*

- USER_HEADER_B

El *user header* B es utilizado para enviar los datos recibidos por el *user header* A, para indicar a la placa FX2LP la lectura de datos provenientes del bus FIFO (fijando los pines SLOE y SLCS) y para fijar los pines FIFOADR (2 pines), SLCS y PKTEND que indican a la placa FX2LP cual FIFO activar, estado de actividad y fin del paquete respectivamente. Los datos

a enviar salen de la hoja del OpenBus a través del puerto OUTPUT[7..0], los valores de los pines SLOE y SLCS a través de READ[1..0] y FIFOARD, SLCS y PKTEND salen a través de CONFIG[3..0]. En total se utilizan 14 pines, los 4 restantes son conectados a tierra.. El *user header* B se configura en modo *input* ya que el recibe los datosque salen del *OpenBus*. La interconexión entre el *user header* B (HB[19..2]) y la hoja del OpenBus se realiza mediante tres *Splitter* y un *Joiner,* estos elementos son configurables y permiten el enrutamiento adecuado de las señales (ver Figura 4.3.15).

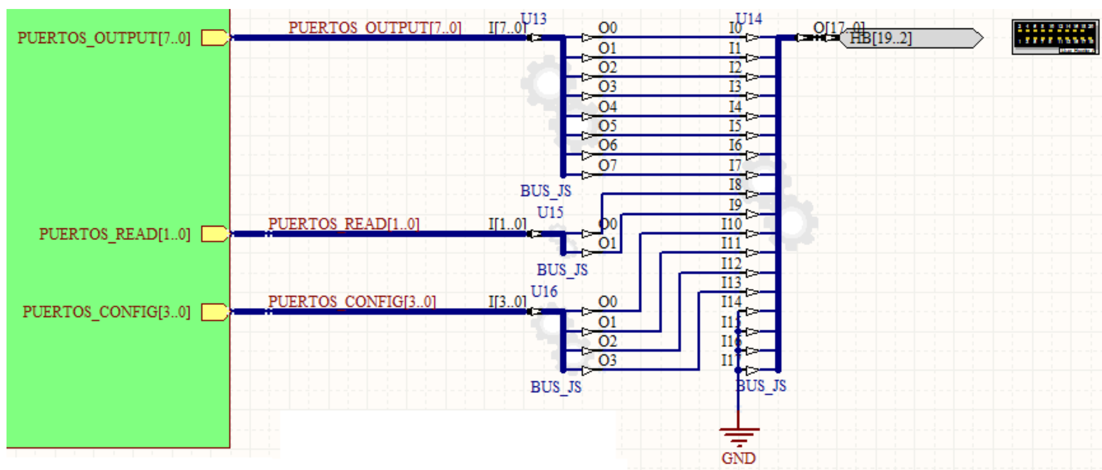


Figura 4.3.15: Conexiones entre *OpenBus* y el *user header* B.

- Analizador lógico (LAX)

El analizador lógico es una herramienta de Altium que permite observar el comportamiento de una señal o un grupo de señales en un intervalo de tiempo determinado, en este caso se utiliza para observar las señales que salen de la NanoBoard 3000 a través del *user header* B (ver Figura 4.3.16).

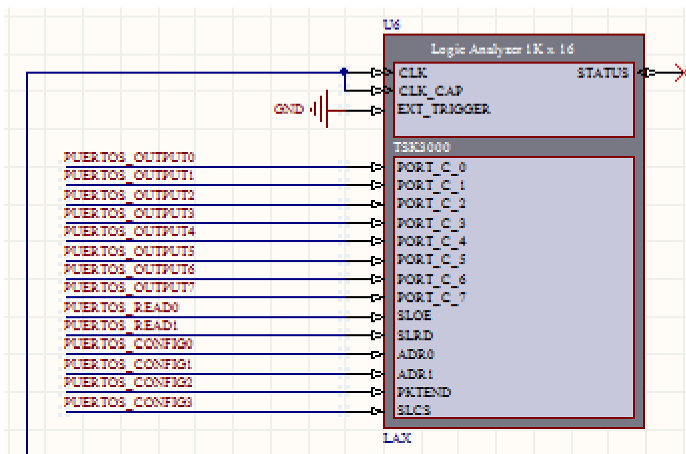


Figura 4.3.16: Señales monitorizadas por el analizador lógico.

En la Figura 4.3.17 se pretenta el sistema *Schematic* completo donde se aprecia la interconexion de cada uno de los bloques antes descritos.

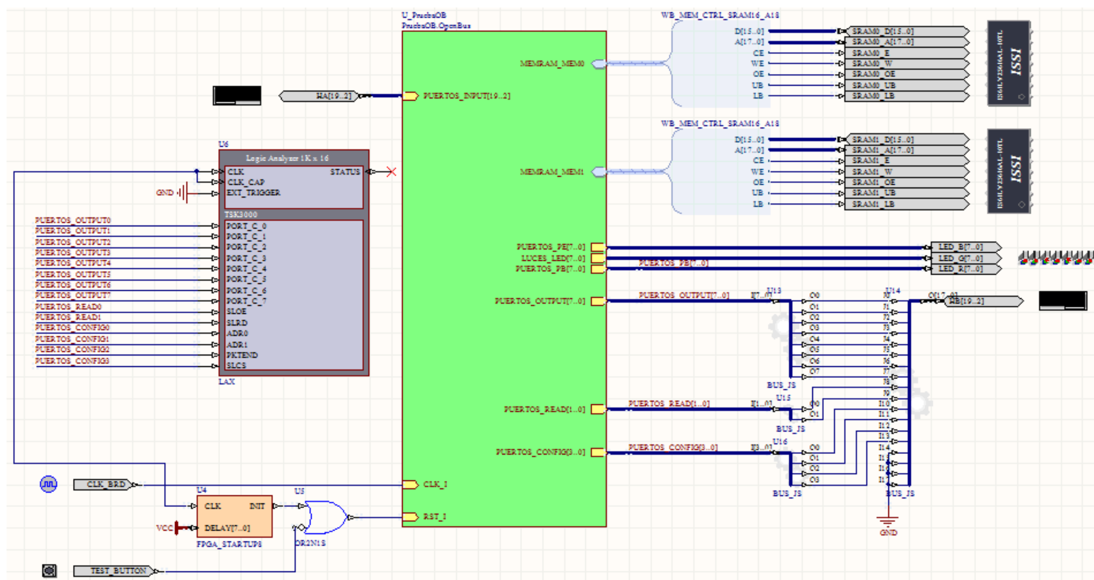


Figura 4.3.17: Sistema *Schematic.*

4.3.3. Proyecto Embebido

El proyecto embebido da soporte de software al microprocesador embebido TSK3000, siempre que se incluya este *soft processor* u otro similar al proyecto de FPGA se debe crear este proyecto que está compuesto por el *Software Platform* y un archivo de código fuente, en nuestro caso el lenguaje seleccionado es C.

4.3.3.1 *Software Platform*

En el *Software Platform* se crea un *stack* o pila por cada dispositivo incluido en el diseño de FPGA que necesite un *driver* para su funcionamiento, en nuestro caso solamente son necesarias dos pilas: una para el *driver* de los leds y otra para el *driver* de los puertos de entrada y salida (ver Figura 4.3.18).

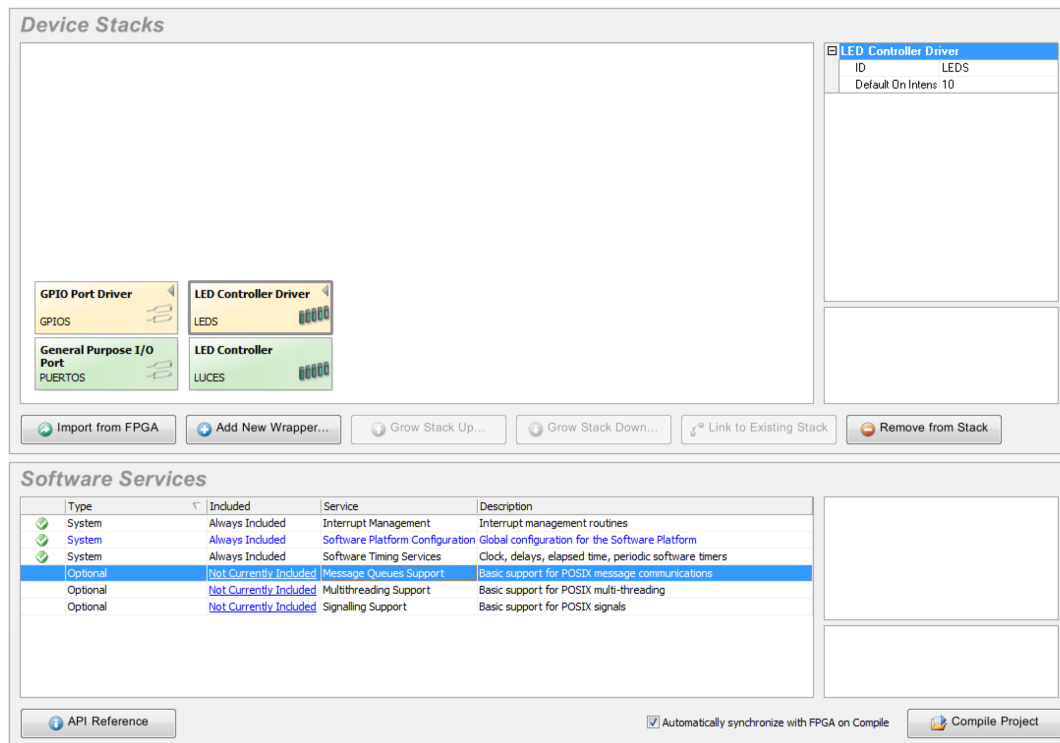


Figura 4.3.18: *Device Stacks* utilizadas en el *Software Platform.*

Los *drivers* inicializan los dispositivos y además permiten su configuración y uso por medio de punteros que utilizan tipos de datos específicos para cada *driver*. También proveen funciones, estructuras y tipos de datos contenidos en librerías que pueden ser cargadas al código fuente del TSK3000, en la Figura 4.3.19 se muestran las funciones que proporciona el *driver* de los leds y en la Figura 4.3.20 se muestran la enumeración, estructuras y funciones que provee el *driver* de los puertos de entrada y salida. En los documentos de cabecera generados por el *Software Platform* se encuentran todos los identificadores para los puertos creados en el *OpenBus* y para los leds, también se genera un documento de código fuente en C llamado swplatform.c en el cual se crean los punteros de los drivers y se inicializan los dispositivos de la pila.

Functions

led_turn_on	Turn on the LED
led_turn_off	Turn off the LED
led_turn_all_on	Turn on all LEDs
led_turn_all_off	Turn off all LEDs
led_set_on_intensity	Sets the intensity of the LED when it is turned on
led_set_all_on_intensity	Sets the intensity of all LEDs when turned on
led_set_intensity	Sets the intensity of an LED
led_open	Open an instance of an LED Controller driver

Figura 4.3.19: Funciones que permiten controlar los leds.

Enumerations

ioport_datawidth_t	I/O Port data widths

Structures

ioport_t	I/O Port driver configuration data

Functions

ioport_open	Open an instance of an I/O Port driver
ioport_set_value	Writes a value to a specific port of the I/O Port
ioport_get_value	Read a value from a specific port of the I/O Port

Figura 4.3.20: Funciones, estructuras y enumeración que permiten el uso de los puertos de entrada y salida.

4.3.3.2 Código Fuente

El código fuente que corre el TSK3000 está escrito en lenguaje C y se encarga de sincronizar la entrada y salida de datos con los FIFOs del FX2LP través de los puertos físicos implementados en el TSK3000 (ver Tabla 4.3.1). Además de procesar los datos, también se encarga de mostrarlos en los leds de la NanoBoard 3000 para que el usuario verifique los datos recibidos (esto es útil solamente a bajas velocidades).

El TSK3000 provee todas las señales necesarias para el correcto funcionamiento del FX2LP: selecciona el FIFO a utilizar, activa el dispositivo mediante el pin SLCS#, activa la salida de datos mediante el pin SLOE y también la lectura mediante el pin SLRD. Esto se explica con detalle en el apartado siguiente y se muestra en la Figura 4.3.22.

4.3.4 Proceso de lectura de datos en el FIFO esclavo de modo asíncrono

La lectura de datos es realizada por el maestro externo, en este caso la NanoBoard 3000. El FX2LP puede entregar los datos de modo síncrono (con la utilización de su propio reloj o uno externo) o asíncrono. Se utilizó el modo asíncrono ya que de esta manera el FX2LP responde automáticamente a cada solicitud de datos hecha por la NanoBoard 3000 mediante la activación de las señales SLOE y SLRD, esto garantiza que la NanoBoard 3000 procese todos los datos provenientes del FX2LP. El ancho de datos del bus FIFO es de 8 bits ya que para implementar uno de 16 bits se requieren más pines en la NanoBoard 3000. En la Tabla 4.3.2 se resumen las conexiones entre la NanoBoard 3000 y la placa FX2LP, estas pueden observarse también en la Figura 3.4 del capítulo anterior.

Tabla 4.3.2: Conexiones entre el FX2LP y la Nanoboard 3000.

Pines FX2LP	**Pines FIFO**	**FPGA *user header A***	**FPGA *user header B***	**Descripción**
PB[7:0]	Datos	2 - 9	---	Bus de datos 8bits
SLRD	SLRD	---	10	Señal de lectura esclavo
SLWR	SLWR	---	---	Señal de escritura esclavo
CTL0	PG	---	---	Bandera programable
CTL1	FF	11	---	Bandera búfer lleno
CTL2	EF	10	---	Bandera búfer vacío
PA2	SLOE	---	11	Habilitador lectura FIFO
PA4	FIFOADR0	---	12	Bits selectores del FIFO
PA5	FIFOADR1	---	13	
PA6	PKTEND		14	Fin del paquete
PA7	SLCS#	---	15	Habilitador del FX2LP

Los pines SLWR y CTL0 no son necesarios ya que corresponden a la escritura en el FIFO esclavo y a la bandera programable respectivamente.

Como se puede observar en la Figura 4.3.21, la lectura de datos no depende de la señal de reloj, esta es usada únicamente para tranferir los datos del puerto USB al bus FIFO y este es un proceso autómatico. Las señales FIFOADR0 y FIFOADR1 permiten seleccionar el *endpoint* a utilizar, en este caso se selecciona el *endpoint* 2 llevando a cero ambas señales (ver Tabla 4.3.3). Una vez seleccionado el *endpoint*, el maestro externo debe esperar a que la bandera de vacío se desactive, esto indica que ya hay datos disponibles en el *endpoint* seleccionado,una vez que esto ocurre, el maestro externo debe activar simultáneamente las señales SLOE y SLRD, de esta manera, los datos estarán disponibles en el bus FIFO y el maestro debe proceder a leerlos y luego desactivar SLO y SLRD, al hacer esto, incrementa el FIFO *pointer* del FX2LP que apunta al siguiente dato a transmitir en espera nuevamente de la activación de las señales SLOE y SLRD. Este proceso se implementó en el TSK3000 para realizar la lectura de datos de la forma descrita, el software del TSK3000 es representado en el diagrama de estados mostrado en la figura 4.3.22.

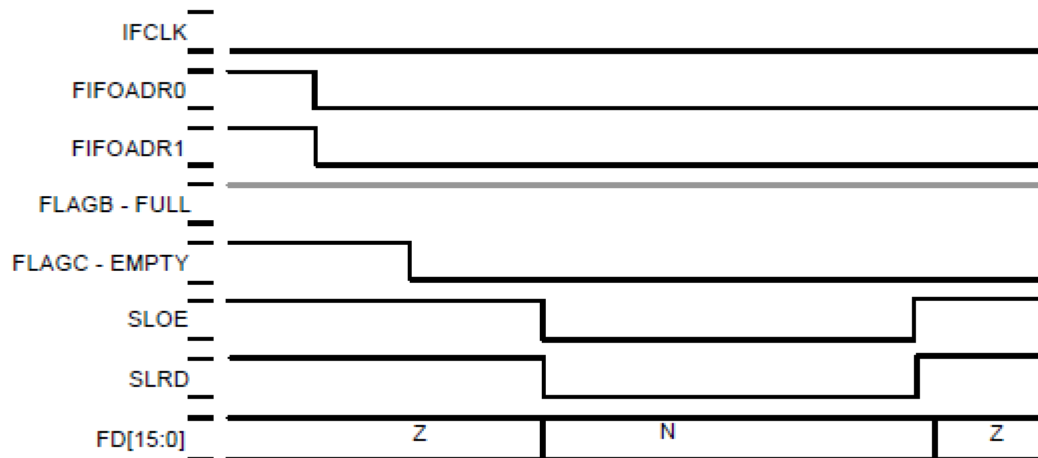


Figura 4.3.21: Diagrama de tiempos: lectura de datos en el FIFO esclavo.

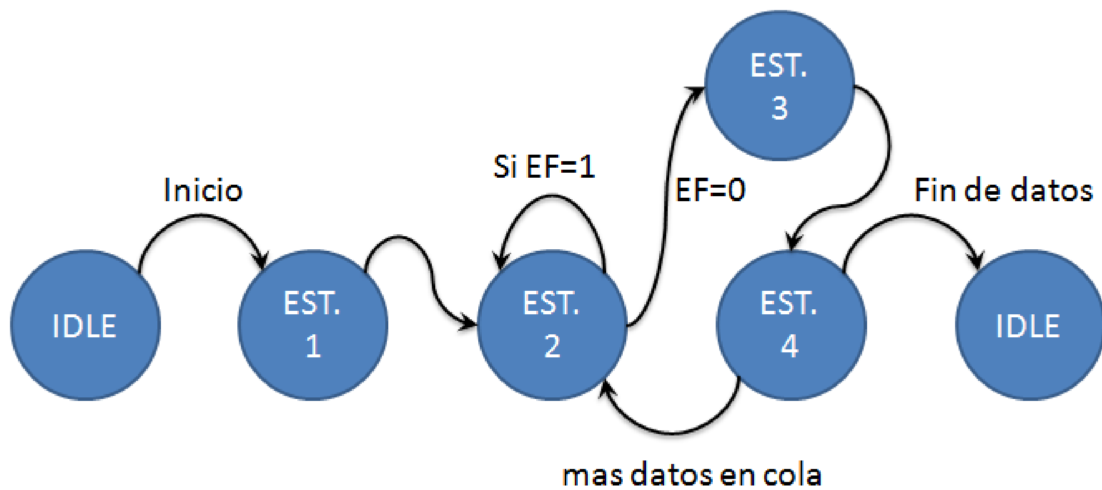


Figura 4.3.22: Diagrama de estados: lectura de datos en el FIFO esclavo por el TSk3000.

Acciones en cada estado del diagrama presentado en la Figura 4.3.22:

IDLE: Cuando ocurre un evento, se hace transición al estado 1.
Estado 1: Activar FIFOADR[1:0] para seleccionar *endpoint*, transición al estado 2.

Estado 2: Si la bandera de *endpoint* vacío es *false* (FIFO no está vacío) se hace transición al estado 3, de lo contrario se queda en este estado hasta que la bandera sea *true*.

Estado 3: Activar SLRD y SLOE, leer datos en el bus FIFO, desactivar SLRD y SLOE. Transición al estado 4.

Estado 4: Si hay más datos que leer regresa al estado 2, sino, hacer transición al IDLE.

La bandera que indica *endpoint* lleno (FF) puede ser útil si se quiere esperar a que el búfer esté completamente lleno para empezar a leer datos, pero en nuestro caso el tiempo es crítico y se empiezan a leer los datos justo en el instante en el cual empiecen a estar disponibles.

Para la lectura y escritura de datos en los puertos del TSK3000 se utilizan las siguientes funciones:

La función **ioport_set_value()** tiene tres argumentos, el driver del puerto en el que se escribirán los datos, el identificador del puerto y la variable que contiene los datos a escribir, esta variable puede ser de hasta 32 bits.

La función **ioport_get_value()** tiene dos argumentos, el driver del puerto en el cual se leerán los datos y el identificador del puerto, esta función retorna el valor leído en dicho puerto, este valor puede ser de hasta 32 bits.

CAPÍTULO V

ANÁLISIS DE RESULTADOS

5.1 Aplicación de control de datos

En la Figura 5.1.1 se muestra una captura de la aplicación de control de datos en la cual se puede apreciar que el VID, el PID y el nombre del dispositivo se muestran en el DevicesComboBox, en el EndpointsComboBox se despliegan los *endpoints* del dispositivo definidos en cada *alternate setting* indicando el tipo de *endpoint* (la aplicación no se limita a transferencias isócronas, también permite *Bulk* e *Interrupt*), el tamaño de los paquetes que puede transmitir, el número del *endpoint* y la dirección de datos. La información visualizada en el DevicesComboBox y en el EndpointsComboBox comprueba que las solicitudes de descriptores, VID, PID y del *string* que contiene el nombre han sido realizadas satisfactoriamente al dispositivo a través del *endpoint* de control. También se observa como el monitor de rendimiento indica el uso del CPU en la CpuBar desde que se comienza a ejecutar la aplicación.

En las Figuras 5.1.2 y 5.1.3 se observa que tanto el PpxBox como el QueueBox permiten configurar la cantidad de paquetes por transferencia y transferencias en cola respectivamente, dichos valores pueden ir desde 1 hasta 128 en ambos casos. Mediante la configuración de estos valores se permite a la aplicación llevar los datos de los búferes al puerto USB de manera más rápida y eficiente, esto ocurre al seleccionar tamaños de paquetes y cantidad de transferencias en cola más grandes.

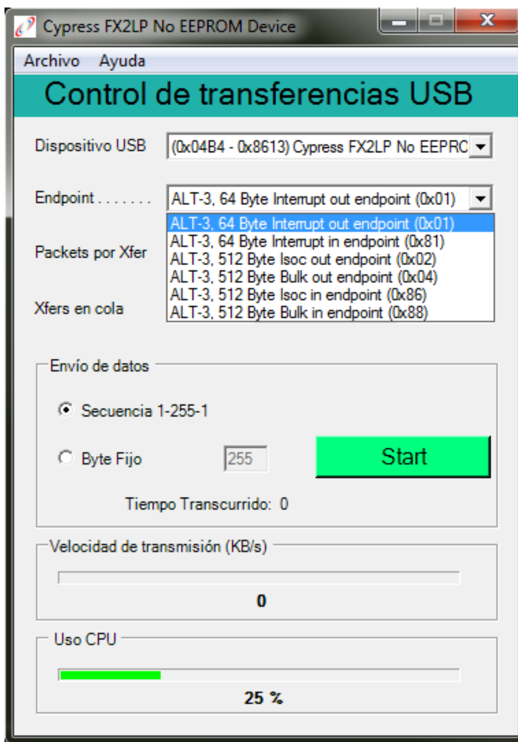


Figura 5.1.1: Aplicación control datos 1.

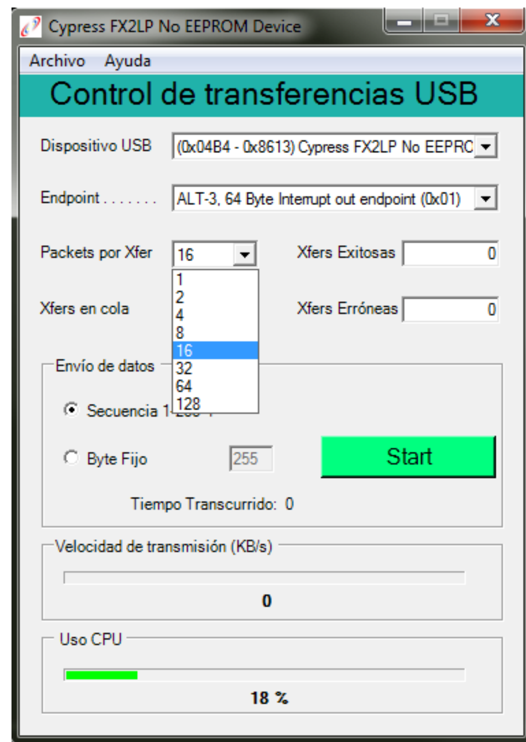


Figura 5.1.2: Aplicación control datos 2.

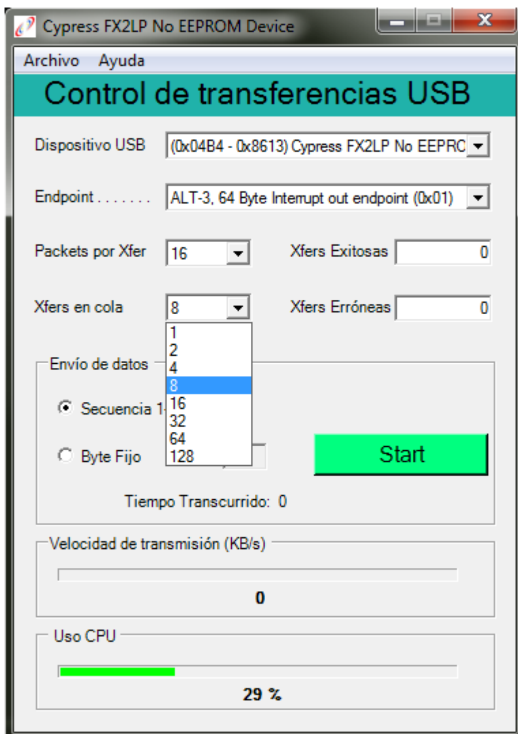


Figura 5.1.3: Aplicación control datos 3.

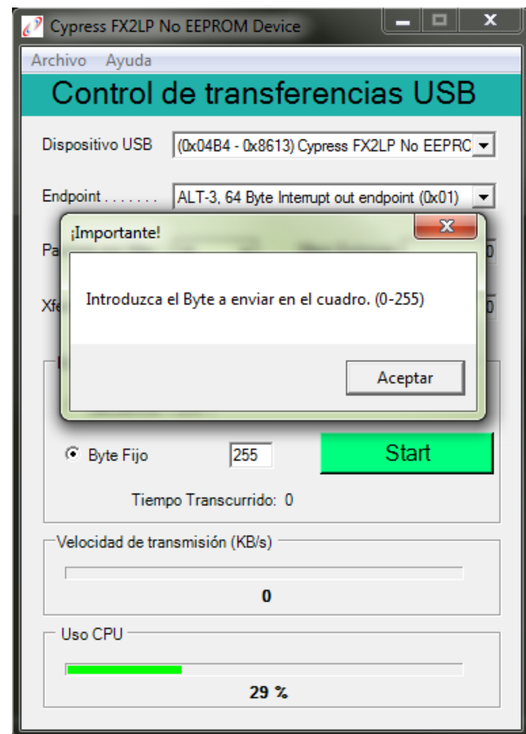


Figura 5.1.4: Aplicación control datos 4.

En la Figura 5.1.4 se observa el mensaje que muestra la aplicación cuando el usuario decide introducir un byte fijo que será el dato a enviar para probar la velocidad de transferencia. Debe estar entre 0 y 255 para evitar el truncamiento.

El dispositivo FX2LP conectado al host, tal como se muestra desde la Figura 5.1.1 hasta la Figura 5.1.4 tiene el firmware por defecto que lo identifica como Cypress FX2LP NO EEPROM device, esto se corresponde con el PID 0x8613, a continuación se muestran las capturas de la aplicación una vez cargado al FX2LP el firmware diseñado, en dicho firmware se especifica el PID 0x1003 (ver Figura 5.1.5) que corresponde al dispositivo de pruebas de Cypres: Cypress FX2LP StreamerExample, estos VID y PID pueden ser sustituídos para que el host identifique el dispositivo con otro nombre y marca, para esto la empresa se debe asociar a usb.org y cancelar las cuotas correspondientes para obtener sus propios VID y PID.

Desde la Figura 5.1.6 hasta la Figura 5.1.10 se muestra la aplicación realizando transferencias de datos a la placa FX2LP, además se aprecia el color rojo en el botón de transferencias y la etiqueta "Stop" sobre él, también se observa que para distintas configuraciones de paquetes por transferencia y paquetes en cola se alcanzan distintas velocidades de transmisión, siendo la cantidad de paquetes por transferencia el parámetro más importante ya que si está fijo en 8, la velocidad varía considerablemente según las transferencias en cola pero al colocarlo en 16 la velocidad alcanzada es mucho mayor que al seleccionar 8 y tiene variaciones mínimas al seleccionar distintos números de transferencias en cola. El tamaño máximo de bytes a transmitir para un *endpoint* de tipo isócrono *high-speed* es de 1024 bytes, en este caso se implementaron paquetes de 64 bytes en los búferes de la aplicación de modo que 8 paquetes corresponden a 512 bytes y 16 paquetes a 1024 bytes. En la aplicación se despliegan valores de hasta 128 paquetes (8kB en total) para dar soporte a transferencias USB 3.0 en un futuro.

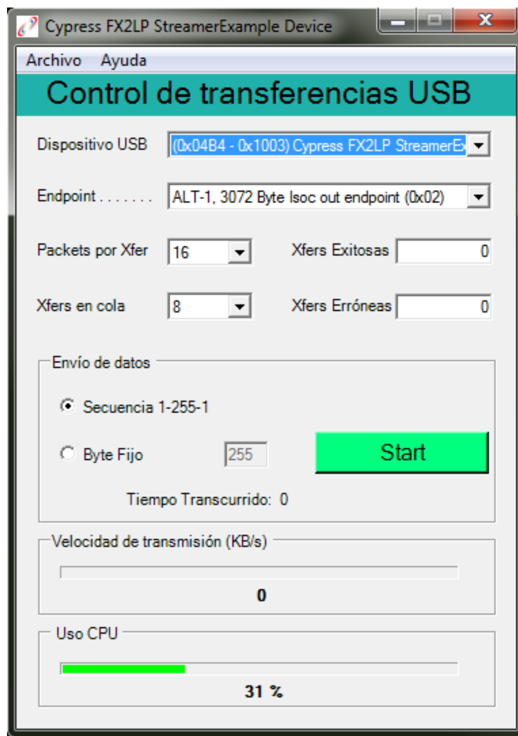


Figura 5.1.5: Aplicación control datos 5.

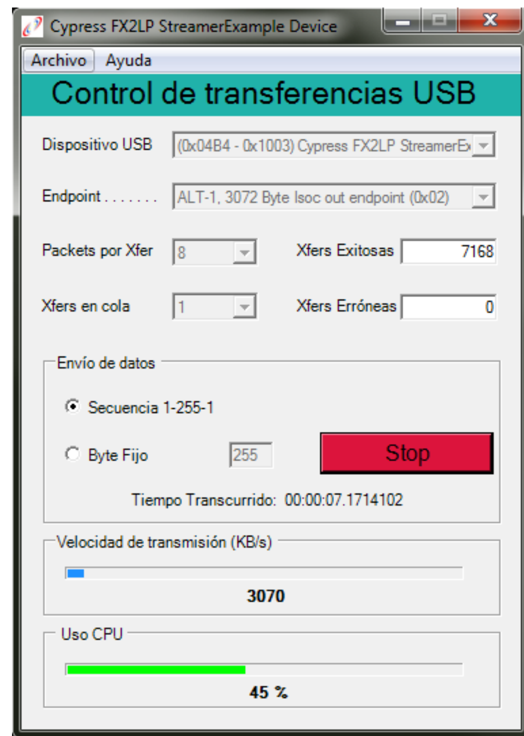


Figura 5.1.6: Aplicación control datos 6.

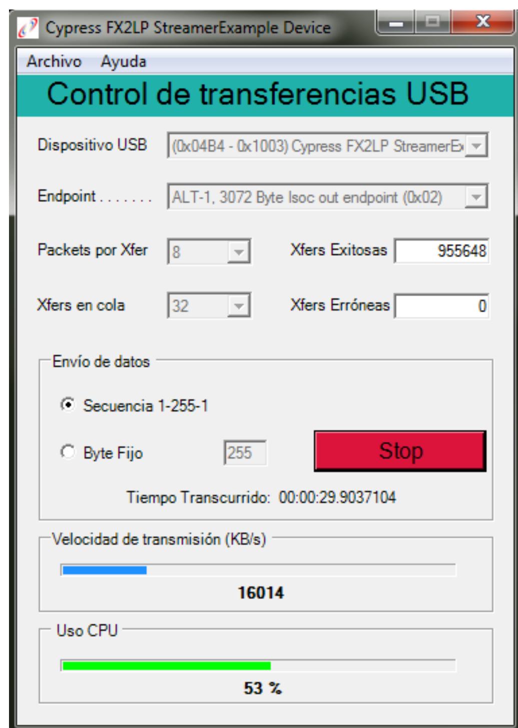


Figura 5.1.7: Aplicación control datos 7.

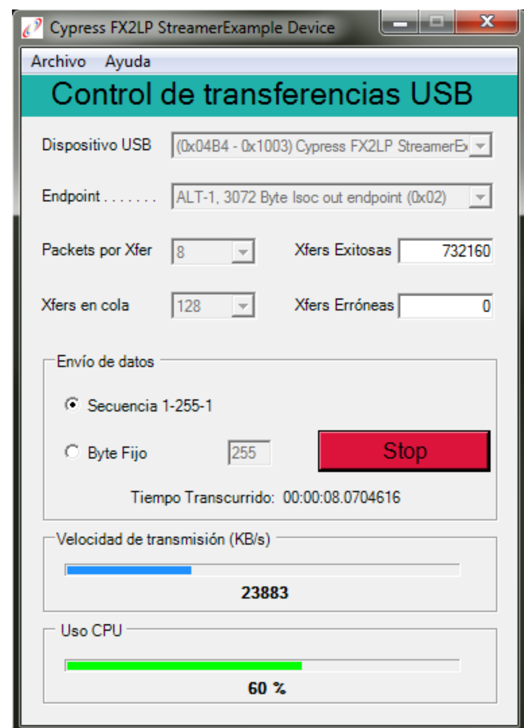


Figura 5.1.8: Aplicación control datos 8.

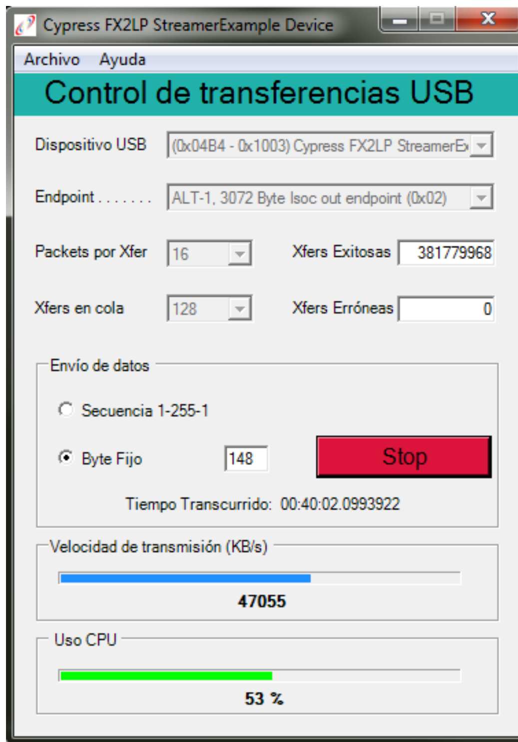


Figura 5.1.9: Aplicación control datos 9.

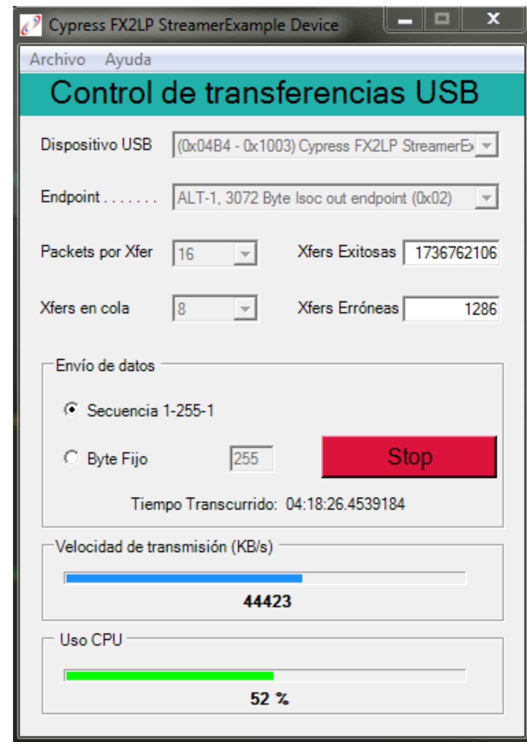


Figura 5.1.10: Aplicación control datos. 10.

Tabla 5.1.1: Resultados de las pruebas de transferencia de datos.

Paquetes por transferencia	Transferencias en cola	Datos enviados	Velocidad alcanzada kB/s	Velocidad alcanzada Mbps
8	1	Secuencia	3070	24,56
8	32	Secuencia	16014	128,112
8	128	Secuencia	23883	191,064
16	128	Dato fijo	47055	376,440
16	8	Secuencia	44423	355,384

En la Figura 5.1.10 se muestra que la velocidad se mantiene en altos niveles incluso tras 4 horas continuas de funcionamiento, durante ese tiempo se realizaron 1736762106 transferencias mientras que ocurrieron 1286 transferencias erróneas, esto constituye un 0,000074%, este resultado es satisfactorio ya que en las transferencias

isócronas son las más vulnerables a los errores y sin embargo su porcentaje es sumamente bajo.

En la Tabla 5.1.1 se observa que se obtuvo una velocidad sostenida alrededor de los 44423 kB/s durante 4 horas con picos de 47000 kB/s y se registró 42000 kB/s como velocidad más baja durante este tiempo. Esta velocidad corresponde a 355 Mbps, hay que mencionar el hecho de que para transferencias isócronas *high-speed* solamente está disponible el 80% del bus por lo que el máximo teórico del USB 2.0 (480 Mbps) se reduce a 384 Mbps, y esto se refiere a picos de rendimiento, por lo que la velocidad estable alcanzada por la aplicación constituye aproximadamente un 92,55% de la velocidad máxima posible, cuando se transmite un byte fijo este porcentaje aumenta hasta 98,03%. Las velocidades alcanzadas son cercanas al máximo que permite el estándar USB.

5.2 *Firmware* de la tarjeta de desarrollo FX2LP

Para cargar el firmware al FX2LP se utiliza una aplicación de Cypress llamada USB Control Center, esta permite almacenar el firmware en la RAM interna (ver Figura 5.2.1) o en una memoria EEPROM que se encuentra en la placa de desarrollo FX2LP, también permite visualizar los descriptores de cualquier dispositivo conectado al computador. En la Figura 5.2.2 se puede observar como la aplicación muestra los descriptores del dispositivo una vez cargado el firmware diseñado, en el *alternate setting* 0 no está definido *endpoint* alguno mientras que en el *alternate setting* 1 se muestra un solo *endpoint* isócrono de tipo *OUT* con un tamaño máximo de paquete de 3072 bytes, y de dirección 02, esto significa: *endpoint* 2, dirección OUT. En caso de tener un 8 delante en lugar de un 0, sería dirección IN. Estas características del *endpoint* son las mismas que muestra la aplicación de control de datos diseñada (ver Figura 5.1.5).

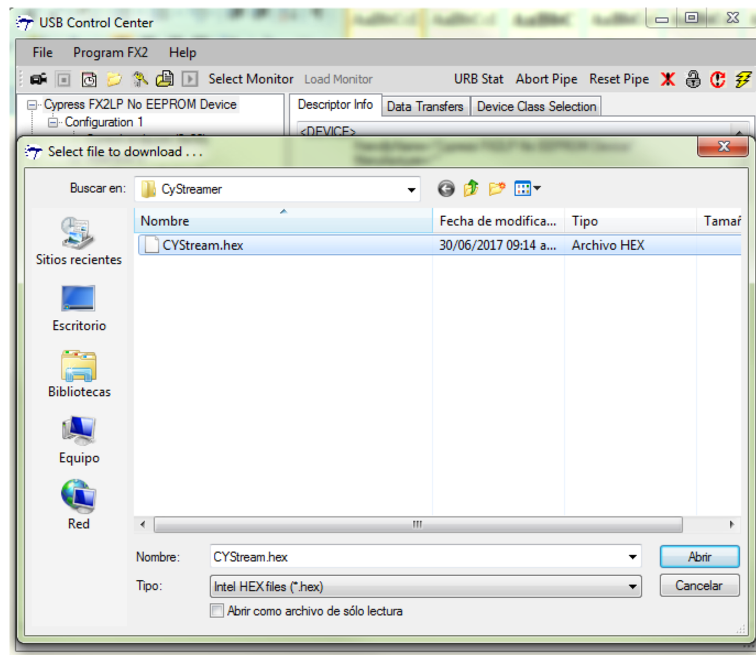


Figura 5.2.1: Cargando archivo .hex del firmware.

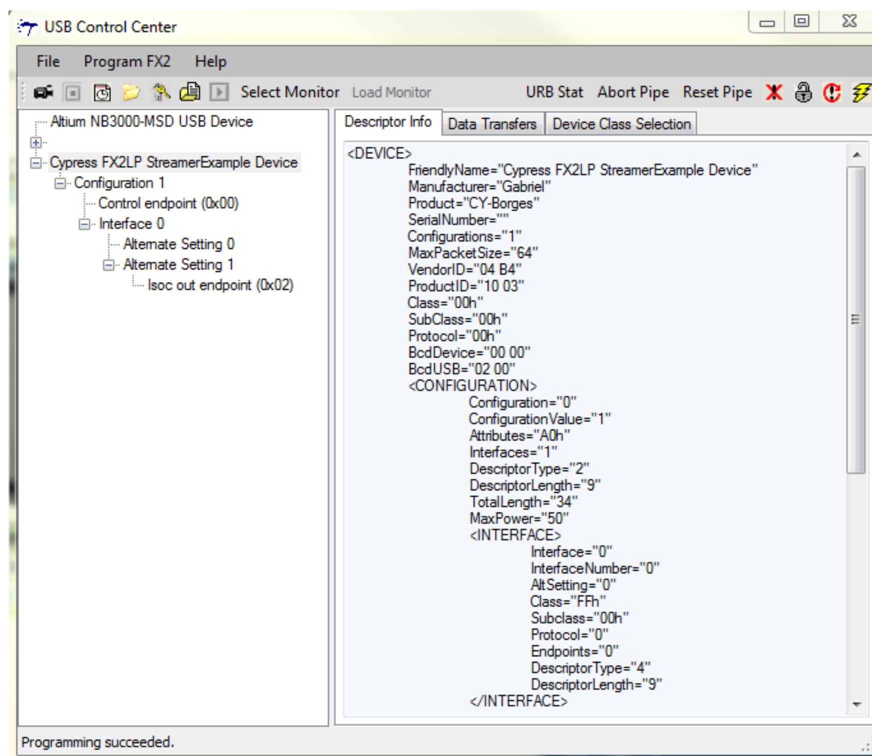


Figura 5.2.2: Información del dispositivo.

Para realizar pruebas de recepción de datos en la placa FX2LP se hizo un montaje en un *protoboard* conectando leds a la salida del bus FIFO, como el bus FIFO es de 8 bits, se requirió la misma cantidad de leds para poder apreciar visualmente los datos recibidos, la activación y desactivación de los pines SLOE y SLRD necesarios para recibir los datos se hizo de forma manual, por lo tanto la velocidad de recepción es muy baja comparada con la salida de datos del computador, aun así los datos se recibieron con la secuencia adecuada debido a que los datos son almacenados en el búfer del *endpoint* 2 cuya capacidad es de 4 kB y hasta que este no se vacíe, no refresca los datos. Evidentemente esto produce pérdida de datos ya que el computador sigue enviando datos que no van a ningún lugar porque el búfer está ocupado, por este motivo, es siempre recomendable que la velocidad de lectura sea aproximada o no muy alejada a la velocidad de escritura del búfer. Los valores lógicos del FX2LP son 0 V y 3,3 V.

El montaje realizado se puede apreciar desde distintos puntos de vista en las figuras 5.2.3, 5.2.4 y 5.2.5.

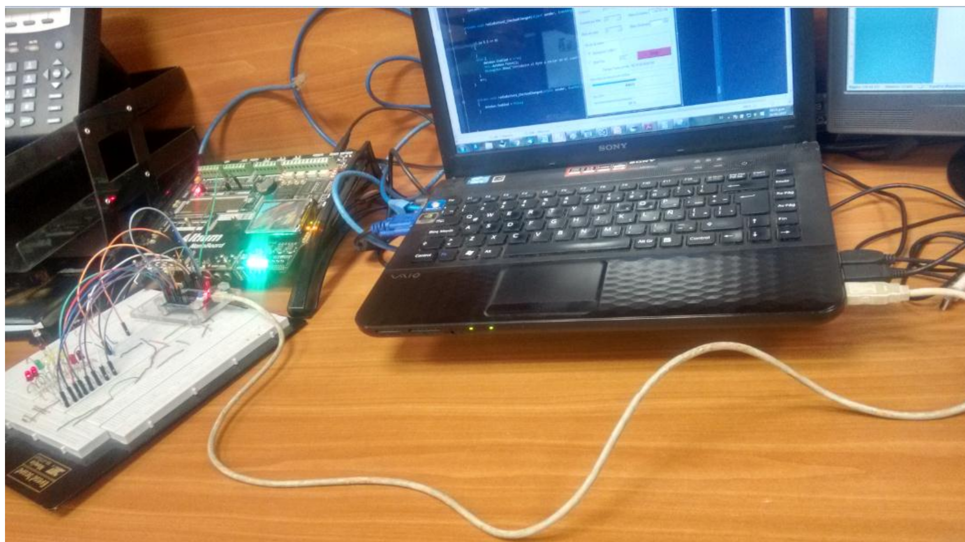


Figura 5.2.3: Conexión computador-placa FX2LP a través de cable USB.

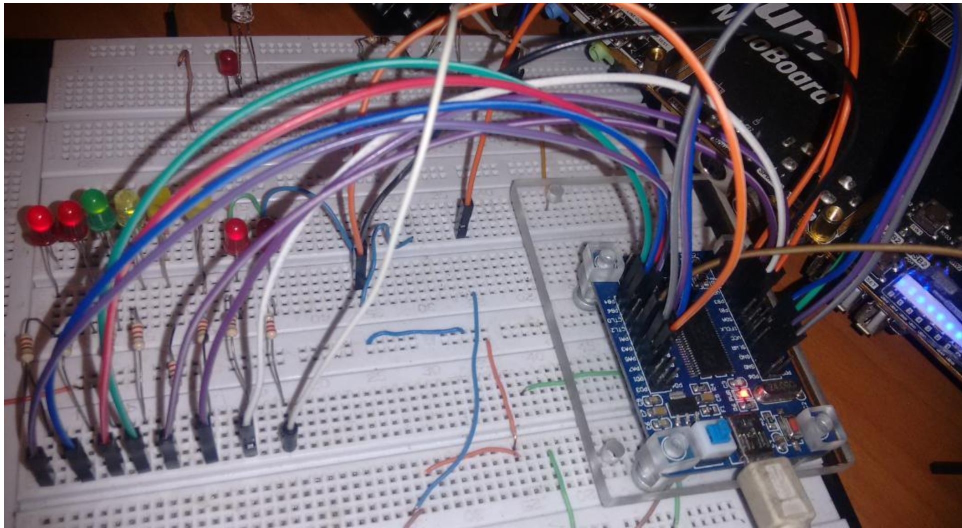


Figura 5.2.4: Conexiones FIFO necesarias para la recepción de datos.

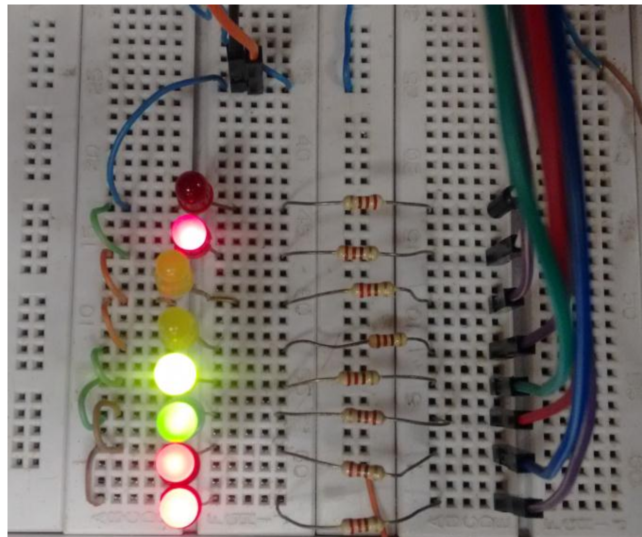


Figura 5.2.5: Datos FIFO recibidos.

La Figura 5.2.5 muestra las conexiones ilustradas en el esquema de la Figura 3.5, las resistencias son para limitar la corriente que entrega la NanoBoard 3000.

5.3 *Core* FPGA NanoBoard 3000

Para realizar la lectura de datos se conectó la placa FX2LP a la NanoBoard 3000 según las conexiones indicadas en el esquema de la Figura 3.4, el montaje realizado se muestra en la Figura 5.3.1. En las salidas de datos del *user header* B se conectaron 8 leds dispuestos en un *protoboard* para visualizar los datos recibidos. En la Figura 5.3.2 se muestra un plano más abierto del montaje realizado.

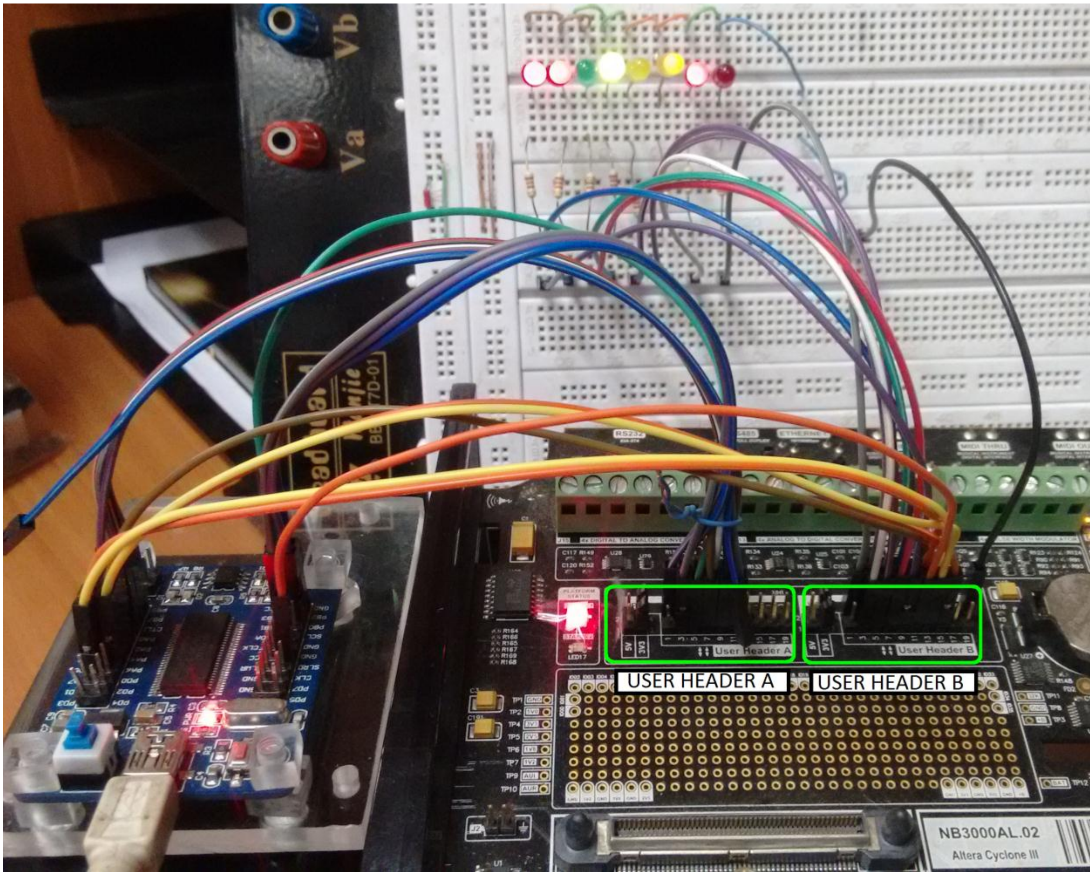


Figura 5.3.1: Conexiones FX2LP – NanoBoard 3000.

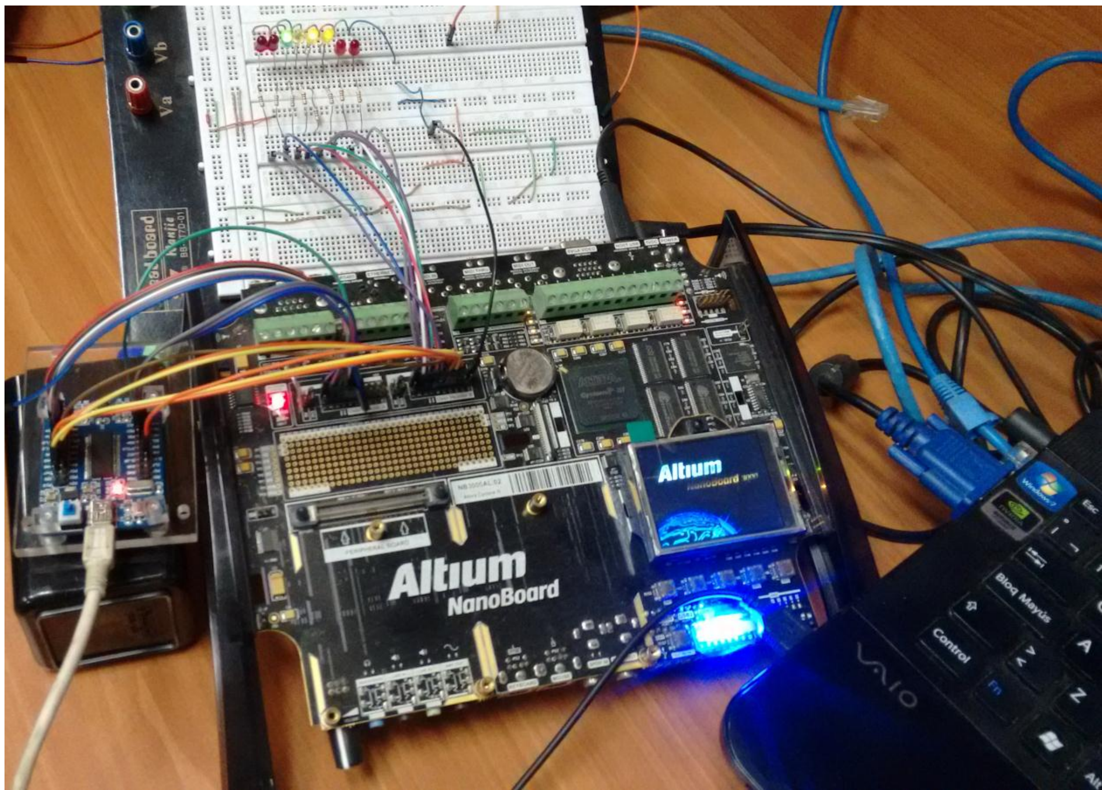


Figura 5.3.2: Montaje PC - FX2LP - NanoBoard 3000.

Una vez realizadas las conexiones físicas se procedió a programar el *core* diseñado en la FPGA, al presionar el botón de programar, el sistema pasa por las 4 etapas: Compilar, Sintetizar, Construir y Programar. Cuando se realiza el proceso sin errores, los puntos que están a la izquierda de cada etapa se tornan de color verde como se aprecia en la Figura 5.3.3, 5.3.4 y 5.3.5. Los resultados de la etapa de construcción se muestran en la Figura 5.3.6, ahí se indica el porcentaje de recursos de la FPGA utilizados para implementar el *core*.

El porcentaje de recursos utilizados es bastante bajo y se puede destacar que se hace uso de 144 pines de entrada/salida y en la FPGA hay 536 disponibles, a pesar de esto, el usuario de la NanoBoard 3000 solamente tiene a su disposición 36 de esos

pines para uso general (asociados a los *user headers*), los otros están reservados para los distintos sistemas que componen la placa de desarrollo.

También se muestra un resumen de tiempo donde se presenta el tiempo mínimo de entrada requerido antes del reloj, el retardo de salida máximo después del reloj y el tiempo mínimo de retención del reloj, estos valores son importantes para sincronizar de forma adecuada la lectura realizada a las salidas FIFO y agregar temporizadores o *delays* de ser necesarios, en nuestro caso no fue necesario.

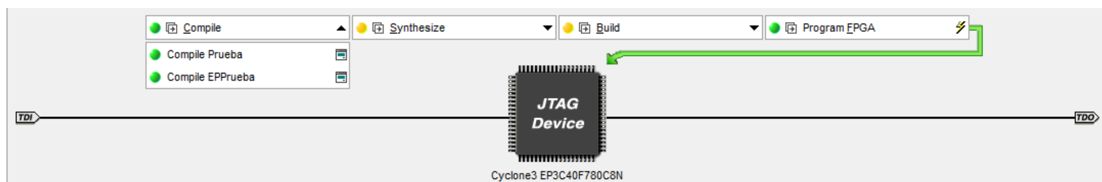


Figura 5.3.3: Resultado de la compilación.

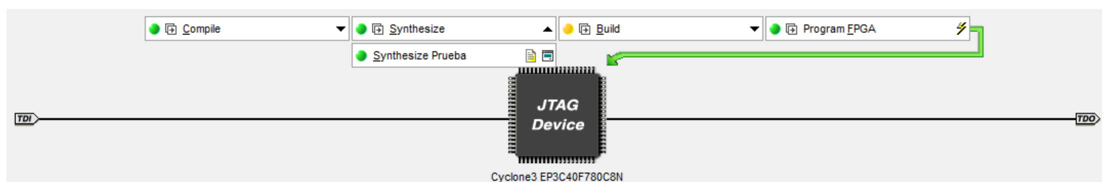


Figura 5.3.4: Resultado de la síntesis.

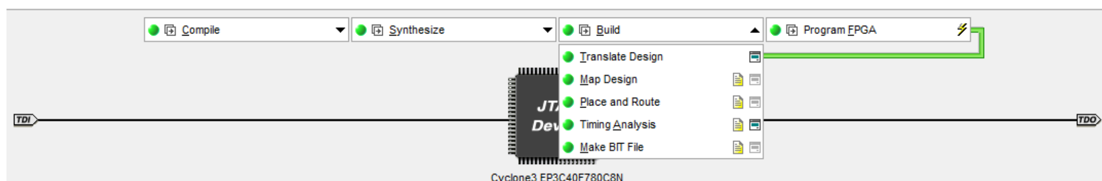


Figura 5.3.5: Resultado de la construcción.

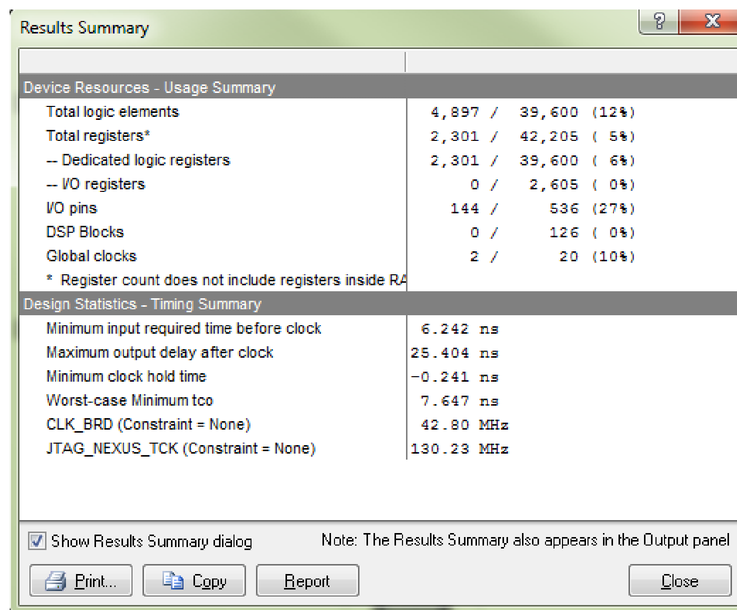


Figura 5.3.6: Resumen del proceso de construcción.

Una vez programada la FPGA, la página de vista de dispositivo de Altium permite configurar el reloj del sistema y manejar los instrumentos virtuales que hayan sido incluidos en el diseño. En este caso se puede configurar solamente el reloj del sistema y el analizador lógico, que se puede apreciar en la Figura 5.3.7 enmarcado con una línea punteada.

El reloj del sistema es configurado a la máxima frecuencia posible que en teoría es de 200 MHz pero a partir de 80 MHz el TSK 3000 no ejecuta instrucción alguna por lo que se fijó la frecuencia de reloj del sistema en 75 MHz utilizando el controlador de frecuencia de Altium (ver Figura 5.3.8).

El analizador lógico se maneja desde un instrumento virtual (ver Figura 5.3.9) que permite iniciar la captura de datos y su visualización de forma analógica o digital, también permite configurar el dispositivo mediante la ventana de configuración (ver figura 5.3.10) en esta ventana se configura el analizador lógico para captar todos los ciclos de reloj del sistema, esto permite obtener lecturas fiables en tiempo real.

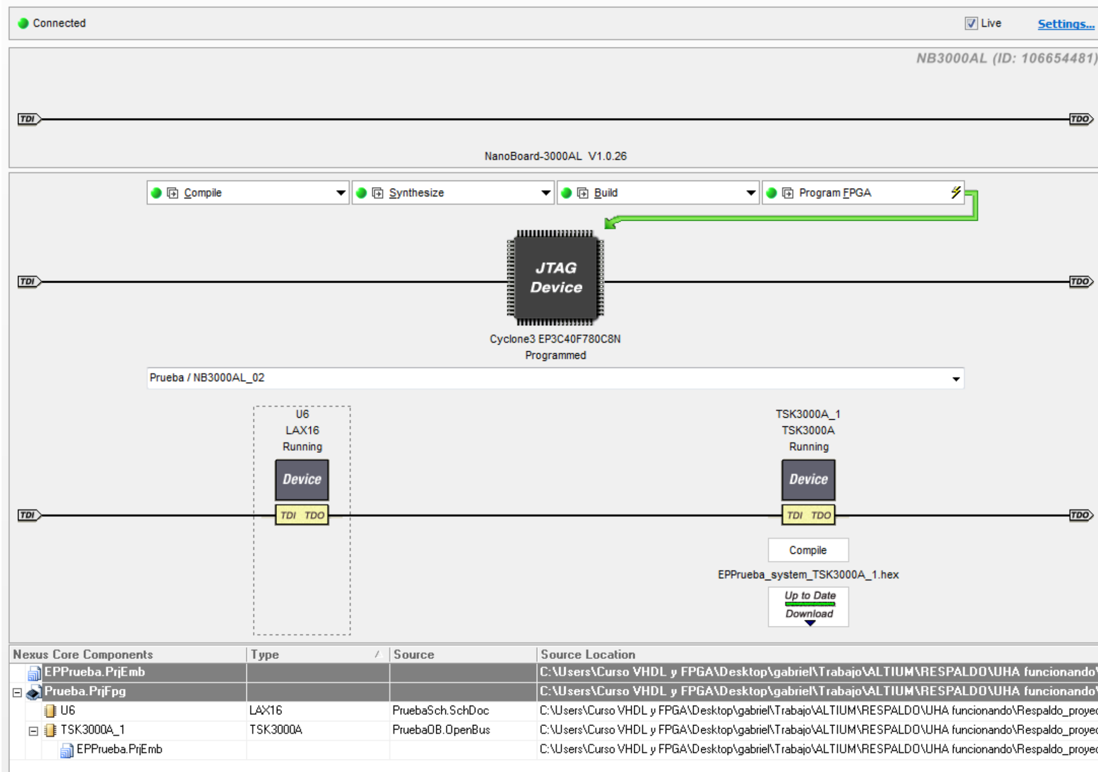


Figura 5.3.7: Página de vista de dispositivo con la FPGA ya programada.

Figura 5.3.8: Instrumento controlador de frecuencia.

Figura 5.3.9: Interfaz del analizador lógico.

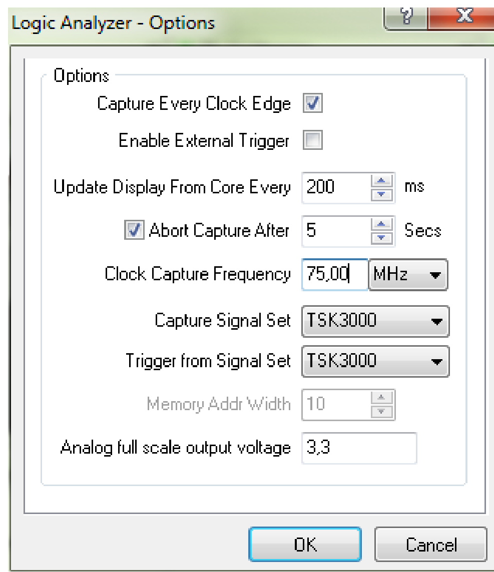


Figura 5.3.10: Ventana de configuración del analizador lógico.

Una vez configurado todo el sistema se procedió a visualizar las salidas del analizador lógico (ver Figura 5.3.11), en ellas se observa que los datos recibidos están en la secuencia adecuada como lo indica la tabla 5.3.1 ya que el FX2LP se configuró para enviar números desde el 1 hasta el 255 y luego invertir la secuencia, permanece en ese ciclo hasta que el usuario detenga la transferencia de datos. El tiempo que transcurre entre dos flancos de subida de las señales SLOE y SLRD indica el período de lectura y escritura de datos de la NanoBoard 3000 ya que entre estos eventos se leen y se escriben los datos.

Tabla 5.3.1: Bytes recibidos visibles en el analizador lógico.

Dato recibido		
Binario	Decimal	Ciclo de Reloj
00111000	56	1
00111001	57	2
00111010	58	3
00111011	59	4
00111100	60	5

El período de la salida de datos es de 2973,333 ns, haciendo uso de la Ecuación 1 obtenemos una frecuencia de transmisión de datos: 336,3229 kHz.

$$f = \frac{1}{T}$$

Ecuación 1: Frecuencia en función del período.

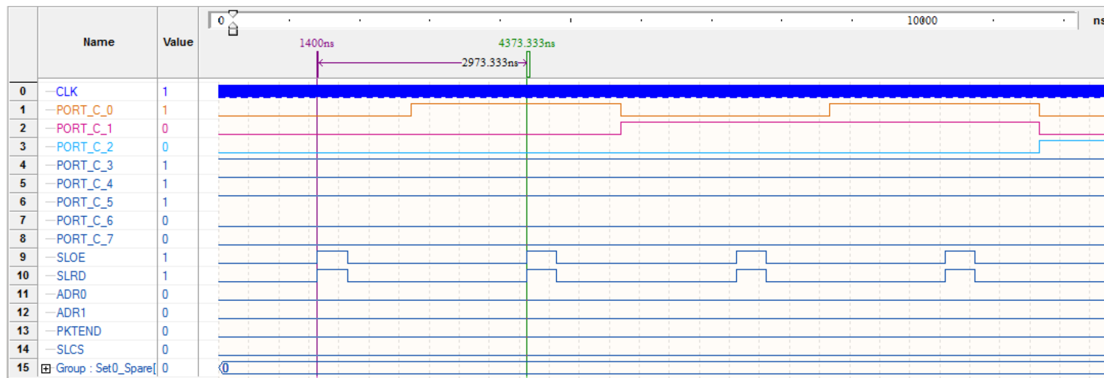


Figura 5.3.11: Señales de salida a través del *user header* B.

Haciendo zoom sobre el diagrama de tiempo de la Figura 5.3.11 y colocando marcas de tiempo entre el flanco de subida y el flanco de bajada de las señales SLOE y SLRD se obtiene el diagrama de tiempo de la Figura 5.3.12. En Dicho diagrama se observa que el tiempo entre ambos flancos es de 426,667 ns, esto representa el tiempo que tarda el TSK3000 en ejecutar cada instrucción ya que las instrucciones de habilitar y deshabilitar dichos pines están una a continuación de la otra en el código y no existen interrupciones o *delays* que puedan afectar los tiempos de respuesta. Cada ciclo de reloj tiene un período de 13,333 ns, esto significa que cada instrucción necesita 32 ciclos de reloj para ser llevada a cabo, este es el rendimiento máximo que se puede obtener al utilizar las funciones disponibles en las librerías del *Software Platform*.

El tiempo de ejecución de cada instrucción solo puede ser modificado por el desarrollador modificando el reloj de todo sistema o programando el TSK3000 sin hacer uso del *Software Platform*, ya que esta capa de abstracción de software maneja todos los procesos de bajo nivel en Altium y no permite modificar las funciones y librerías que trae definidas. Para programar el TSK3000 sin el *Software Platform* se deben crear todas las librerías que permiten controlar cada uno de los periféricos utilizados, para esto se tiene un herramienta llamada *Tasking* TSK3000 *Toolset* donde se describen las funciones de ANSI C bajo el estándar C99. Realizar dichas librerías es una tarea extensa que sale del alcance de este proyecto y no está garantizada la obtención de mejores tiempos de respuesta que los obtenidos por Altium utilizando las librerías del *Software Platform*.

Aunque no se tiene acceso a las definiciones de las funciones de las librerías del *Software Platform*, el código desarrollado para el TSK3000 es bastante resumido y aun así su tamaño en memoria es bastante grande, esto evidencia la complejidad de cada función invocada para manejar los dispositivos periféricos mediante *drivers* generados por cada pila o *stack*.

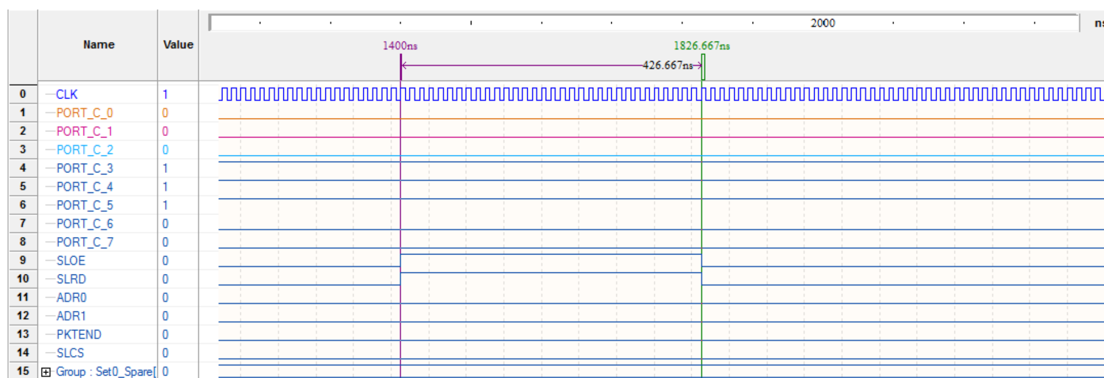


Figura 5.3.12: Zoom de señales de salida del *user header* B.

En las figuras 5.3.13 y 5.3.14 se observan las formas de onda de los pines SLOE y SLRD obtenidas con un osciloscopio digital Agilent Technologies modelo DSO3062A, los períodos obtenidos en ambos casos son muy parecidos a los valores arrojados por el analizador lógico, las diferencias se pueden asociar a la posición de los cursores en el osciloscopio ya que son ajustados de forma manual, pero al estar tan cercanos ambos resultados se comprueba la frecuencia de lectura y salida de datos de la NanoBoard 3000 (ver Tabla 5.3.2).

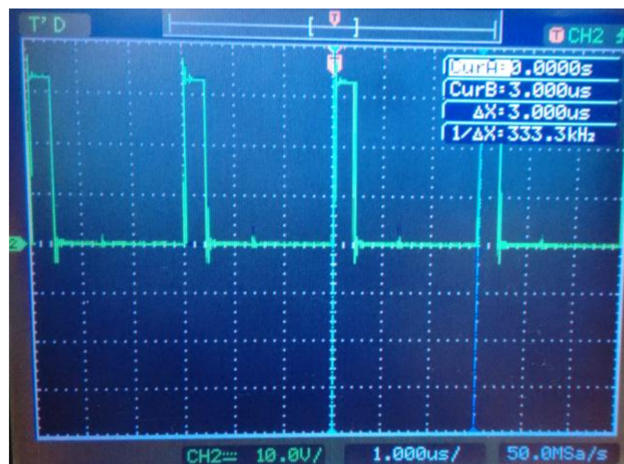


Figura 5.3.13: Tiempo entre flancos de subida de la señales SLOE y SLRD.

Figura 5.3.14: Tiempo entre flanco de subida y bajada de la señales SLOE y SLRD.

Tabla 5.3.2: Resultado de las mediciones con el analizador lógico y el osciloscopio.

	Analizador lógico	**Osciloscopio**	**Diferencia porcentual**
Tiempo entre flancos ascendentes de las señales SLOE y SLRD	2973,33 ns	3 µs ± 0,001 µs	0,889 %
Tiempo entre flanco ascendente y flanco descendente de las señales SLOE y SLRD	426,667 ns	420 ns ± 0.1 ns	1,562 %

Para hallar la tasa de transferencia de bits por segundo (*bitrate*) se utiliza la Ecuación 2 en donde f es la frecuencia de salida de datos y N_{bits} es el ancho del bus de datos.

$$bitrate = f * N_{bits}$$

Ecuación 2: Tasa de transferencia de bits.

Tabla 5.3.3: Velocidades de transmisión con 8, 16 y 32 bits.

Frecuencia (kHz)	**Número de bits**	**Velocidad (MBps)**	**Velocidad (Mbps)**
336,229	8	0,3362	2,6905
336,229	16	0,6724	5,381
336,229	32	1,3449	10,7623

La tasa de transferencia de bits obtenida en los pines de salida de la NanoBoard 3000 es de 2,6905 Mbps, se incluyen los cálculos con 16 y 32 bits porque aunque los pines de entrada y salida no alcanzan para dichos buses, si en un futuro se implementa el diseño con el chip SX2 acoplado a la NanoBoard, se pueden extraer datos de 16 o 32 bits a través de los *user headers* en el mismo intervalo de tiempo ya que las instrucciones del TSK3000 son de 32 bits y la entrada de datos hacia la NanoBoard 3000 se haría directamente por el puerto USB tipo B mini.

La frecuencia de transmisión de datos obtenida en los puertos de salida del TSK3000 se debe a que las funciones del *Software Platform* utilizadas para la lectura y escritura en dichos puertos requieren 32 ciclos de reloj para ejecutarse, motivo por el cual el reloj del sistema configurado a 75 MHz (máximo soportado por el TSK3000) produce una lectura y escritura de datos tan lenta en comparación a la magnitud del reloj. La frecuencia obtenida produce una tasa de bits adecuada para transmitir video en HD pero no soporta la transmisión de múltiples canales en simultáneo.

CONCLUSIONES

Las FPGAs son herramientas sumamente poderosas para el diseño de cualquier tipo de circuito o dispositivo electrónico, permitiendo al desarrollador gran libertad y flexibilidad a la hora de implementar y probar su diseño sin que esto implique pérdida de materiales o componentes. En el caso de las telecomunicaciones siempre hay que tener en cuenta el gran flujo de datos que pueden llegar a manejar debido a su naturaleza y a las grandes frecuencias a las que operan, para esto se debe escoger el dispositivo adecuado que se adapte a dichas velocidades. El uso del TSK3000 sintetizado en la NanoBoard 3000 para implementar la salida de datos de un transmisor de TDA no es factible debido a que dicho *soft processor* presenta limitaciones en cuanto al procesamiento de datos y no es capaz de procesar los datos requeridos por un transmisor de TDA a la velocidad y frecuencia adecuada.

En el caso del USB 2.0 si es factible su uso para aplicaciones de comunicaciones ya que su velocidad de datos es más que suficiente para transmitir o recibir toda la información contenida en las tramas de comunicación de dicho estándar a la frecuencia adecuada siempre que se usen *endpoints* de tipo *interrupt* o isócronos.

En un computador personal se pueden generar y enviar o incluso recibir datos para la TDA sin problema alguno ya que los microprocesadores que estos incorporan son especialmente diseñados para el manejo de grandes cantidades de datos y operan a frecuencias sumamente altas, pudiendo realizar diversas tareas además de la recepción o transmisión de dichos datos.

RECOMENDACIONES

Debido al gran flujo de datos que requiere la transmisión de múltiples canales de TDA es necesario hacer uso de un dispositivo con mayor capacidad de procesamiento de datos que el TSK3000 para enviar los datos a la velocidad y frecuencia requerida. Por esta razón se recomienda trasladar el diseño del *core* FPGA presentado en este trabajo a otro *soft processor* con mayor capacidad de procesamiento que el TSK3000, pudiendo probarse en primera instancia el Nios II propio de Altera disponible para ser implementado en la FPGA Cyclone III de la NanoBoard 3000 y si no cumple los requisitos se recomienda trasladar el diseño a otra FPGA que permita configurar su reloj a frecuencias mayores que los 75 MHz que permite la NanoBoard 3000 como máximo. De igual manera se recomienda la adquisición de una licencia actualizada de Altium Designer para sacar mayor provecho al diseño de FPGA con la NanoBoard 3000 ya que esto garantiza acceso a foros, ejemplos, y acceso a componentes actualizados.

El firmware diseñado utiliza paquetes de 8 bits para la transmisión de datos debido a la limitación de pines de la NanoBoard 3000, es recomendable configurarlo para que la transmisión sea de 16 bits o 32 bits si se trabajará con otra FPGA que posea mayor cantidad de pines de entrada y salida. Se recomienda también la adquisición de un *Vendor* ID USB para poder utilizar el logo del USB en los productos desarrollados y así evitar problemas de compatibilidades y de licencia de uso.

En cuanto a la aplicación de control de datos, se recomienda acoplar el software de transmisión de datos de TDA a la aplicación diseñada ya que de esta forma se tendría comunicación por medio del USB desde el computador hasta los puertos de salida de la FPGA.

REFERENCIAS BIBLIOGRÁFICAS

[1] Pisciotta N, Liendo C, Lauro R. *Transmisión de Televisión Digital Terrestre en la Norma ISDB-Tb*. (Libro).-- Buenos Aires: Argentina: Cengage Learning. Primera Edición, 2013, cap 1.

[2] Axelson J. *USB Complete: The Developer´s Guide*. (Libro). Lakeview Research LLC, Third Edition, 2005, cap. 1-14.

[3] Axelson J. *Serial Port Complete: COM Ports, USB Virtual Com Ports, and Ports for Embedded Systems*. (Libro). Lakeview Research LLC, 2007, cap. 14-16.

[4] *Universal Serial Bus – OSDev Wiki.* [En línea]. <http://wiki.osdev.org/Universal_Serial_Bus>. [Consulta: Abril 2017].

[5] *Beyond Logic – USB in a NutShell.* [En línea]. <http://www.beyondlogic.org/usbnutshell/usb1.shtml>. [Consulta: Abril 2017].

[6] *EZ-USB® FX2LP™ USB Microcontroller High-Speed USB Peripheral Controller*. [En línea]. <http://www.cypress.com/documentation/datasheets/cy7c68013a-cy7c68014a-cy7c68015a-cy7c68016a-ez-usb-fx2lp-usb>. [Consulta: Mayo 2017].

[7] *EZ-USB® Technical Reference Manual*. [En línea]. <http://www.cypress.com/file/126446/download>. [Consulta: Mayo 2017].

[8] *AN65209 – Getting Started with FX2LP™*. [En Línea]. <http://www.cypress.com/documentation/application-notes/an65209-getting-started-fx2lp-ja>. [Consulta: Mayo 2017].

[10] *USB-C en los smartphones del futuro.* [En línea].
<https://hipertextual.com/2015/05/usb-c-en-los-smartphones>.
[Consulta: Noviembre 2017].

[11] Pinto L. *Diseño de la interfaz de salida de un modulador ISDB-Tb implementado en software*. (Tesis).-- Caracas: Venezuela: Universidad Simón Bolívar, 2015.

[12] *MalLog –CY7C68013A Mini Board.* [En línea]. <http://blog.malcom.pl/wp-content/uploads/2015/02/CY7C68013A_board.jpg>. [Consulta: Julio 2017].

[13] *EZ-USB SX2™ High speed USB Interface Device – Cypress.* [En línea]. <http://www.cypress.com/file/144251/download>. [Consulta: Julio 2017].

[14] *Introducción a la tecnología FPGA: 5 beneficios principales – National Instruments.* [En línea]. < http://www.ni.com/white-paper/6984/es/>. [Consulta: Julio 2017].

[15] Host FPGA (NanoTalk Controller) – Altium Techdocs. [En línea]. <http://techdocs.altium.com/display/HWARE/NanoBoard+3000+-+Host+FPGA+(NanoTalk+Controller)>. [Consulta: Julio 2017].

[16] *Types of Projects – Altium Techdocs.* [En línea].
<http://techdocs.altium.com/display/ADOH/Types+of+Projects+in+Altium+Designer>. [Consulta: Julio 2017].

[17] *TSK3000A – Altium Techdocs.* [En línea].
<http://techdocs.altium.com/display/FPGA/TSK3000A>. [Consulta: Julio 2017].

[18] *Introducción a la programación en VHDL – Universidad Complutense de Madrid.* [En línea]. < http://eprints.ucm.es/26200/1/intro_VHDL.pdf>.
[Consulta: Julio 2017].

[19] *Active-HDL™ – ALDEC.* [En línea].
< https://www.aldec.com/en/products/fpga_simulation/active-hdl>.
[Consulta: Agosto 2017].

[20] *Introducción a Visual Studio – MSDN.* [En línea].
< https://msdn.microsoft.com/es-es/library/fx6bk1f4(v=vs.100).aspx>.
[Consulta: Agosto 2017].

[21] *Introducción a .NET Framework – MSDN.* [En línea].
<https://msdn.microsoft.com/es-es/library/hh425099(v=vs.110).aspx>.
[Consulta: Agosto 2017].

[22] *Lenguajes de programación – MSDN.* [En línea].
<https://msdn.microsoft.com/es-es/library/aa292164(v=vs.71).aspx>.
[Consulta: Agosto 2017].

[23] *Release notes for Altium Designer Update 18 [10.1051.23878] – Altium.* [En línea].<http://techdocs.altium.com/display/ADOH/Release+notes+for+Altium+Designer+Update+18+(10.1051.23878)#>. [Consulta: Agosto 2017].

[24] *Cypress CyUSB .NET DLL Programmer´s Reference – Cypress.* [En línea].
<http://www.cypress.com/file/53111/download>. [Consulta: Agosto 2017].

GLOSARIO

Búfer: Espacio de memoria reservado para el almacenamiento temporal de datos mientras están esperando ser procesados.

Clase (C#): Construcción que engloba tipos personalizados agrupando las variables de otros tipos y eventos.

***Firmware*:** Programa que establece la lógica de más bajo nivel que controla circuitos de un determinado dispositivo.

IP *Core*: Un núcleo IP (propiedad intelectual) es un bloque de lógica o datos que se utiliza en la fabricación de una matriz de FPGA o un circuito integrado de aplicación específica para un producto.

***Namespace* (C#):** Se utiliza para declarar un ámbito que contiene un conjunto de objetos relacionados. Puede utilizar un espacio de nombres para organizar elementos de código y crear tipos únicos a nivel global.

Objeto (C#): Un objeto es un bloque de memoria que se ha asignado y configurado de acuerdo con el plano definido en una clase. Los objetos también se denominan instancias y pueden almacenarse en una variable con nombre, o en una matriz o colección.

***Stack*:** Estructura de datos que permite almacenar y recuperar datos, el modo de acceso es LIFO (del inglés Last In First Out, último en entrar, primero en salir).

Transceptor: Dispositivo que cuenta con un receptor y un transmisor.

***Wishbone*:** Bus de datos *open-source* especialmente diseñado para *soft processors* implementados en FPGA.