

# Capítulo 1

## Introducción

El presente informe busca dar a conocer al lector las tareas y actividades desarrolladas por el autor, en el marco del Trabajo Final de la carrera Ingeniería Electrónica, dictada en la Facultad de Ingeniería de la Universidad Nacional de San Juan. El objetivo del trabajo es diseñar e implementar una interfaz para la transmisión de datos hacia una computadora personal (PC), adquiridos por sistemas desarrollados en arreglos de compuertas de campo programables (FPGA) para aplicaciones científicas, a través del Bus Serial Universal (USB). A lo largo de este documento, se comprenderá la problemática que se resuelve y la configuración, fundamentos y modo de uso del sistema propuesto.

En la sección 1.1 se presentan las motivaciones de este trabajo y se detalla la problemática a resolver. Luego, se detallan los objetivos que persigue este trabajo. Seguido a esto, se otorga un esquema que describe la solución planteada y se justifica el protocolo elegido. Finalmente, se repasan algunos conceptos importantes de la norma USB que luego se utilizan en el trabajo desarrollado.

### 1.1. Motivación

El grado de avance que han experimentado la electrónica y la tecnología en general, gracias a la industria de los semiconductores, permite que la producción científica pueda adquirir una gran cantidad de datos. Para llevar a cabo la producción del conocimiento, es necesario el relevamiento y registro de diferentes tipos de magnitudes físicas y/o químicas sobre el objeto o proceso a investigar. En muchas ocasiones, estas magnitudes resultan difíciles de observar y cuantificar, por lo que es conveniente transformar las variables a conocer en otras más sencillas de medir. Para este propósito, se utilizan transductores.

Se conoce como transductor a cualquier dispositivo que recibe estímulos energéticos de una condición, situación o fenómeno físico y/o químico y los convierte en una señal asociada y definida de otra forma de energía[1, 2]. En otras palabras, los transductores son conversores de energías[1, 2, 3]. Se denomina sensor a una clase particular de transductor que genera, como variable de salida, una señal eléctrica que está especialmente adaptada para ser ingresada en un circuito electrónico, o adecuada al sistema de medida que se utilice [4, 5, 6].

Las altas escalas de integración de circuitos alcanzadas en la actualidad posibilitan el diseño de sistemas sensoriales cada vez más complejos, en los cuales se logra agrupar miles de sensores en áreas reducidas, obteniendo medidas simultáneas y flujos crecientes de datos. Este trabajo se

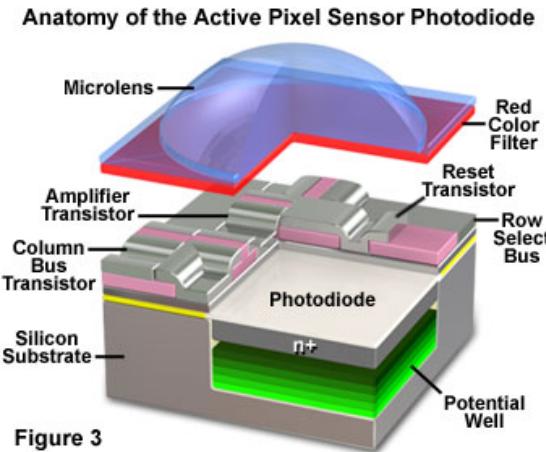


Figura 1.1: Esquema físico de un APS[8]

centrará en la transmisión de datos provenientes de sensores de imagen, uno de los desarrollos que se encuentra en boga.

Una imagen, desde un punto de vista digital, es un arreglo bidimensional de números, los cuales pueden ser exhibidos en una pantalla en forma de intensidad y colores de luz. Cada punto del arreglo que se muestra en pantalla se denomina pixel, acrónimo del inglés *PIcture EElement*, o elemento de imagen. Por esto, un sensor de imagen puede estar compuesto, bien por un arreglo bidimensional de sensores lumínicos (como la cámara de un teléfono celular), como por un transductor que es simultáneamente desplazado y medido, (método utilizado, entre otras, para la microscopía de fuerza atómica [7]), o por una combinación de ambos métodos. Por ejemplo, un scanner posee un arreglo lineal de transductores que son desplazados a través de la hoja para generar una imagen digital. En cualquiera de los casos, es de suma utilidad que la lectura de imágenes sea realizada en el menor tiempo posible, ya que cada imagen conlleva una cantidad no menor de datos.

Uno de los trabajos que más aportó al desarrollo de sensores de imágenes modernos, fue la introducción de los APS (*Active Pixel Sensor*, o sensor de píxeles activos) [9]. Este sensor integra en un proceso CMOS (acrónimo inglés de Metal-Óxido-Semiconductor Complementario, que es el método actualmente más económico para integrar transistores en una única pastilla de silicio), un fotodiodo, un transistor de reset (utilizado para controlar el tiempo de integración, es decir, de exposición a la luz) transistores de selección (utilizados para conectar un pixel determinado dentro del arreglo) y un amplificador seguidor de fuente en cada pixel[8]. El fotodiodo, previamente cargado, transduce la luz en una descarga eléctrica y el amplificador convierte la carga remanente en tensión para facilitar su lectura. La Figura 1.1 muestra el dibujo de un APS. Se observa el área sensible a la luz y los diferentes transistores que intervienen en su funcionamiento. Además se incorpora una micro-lente cuya función es la de enfocar los fotones sobre el área sensible y un filtro utilizado para identificar colores. En el caso de sensores monocromáticos, se omite la colocación del filtro de color durante la fabricación.

A partir del desarrollo de los APS, se fue perfeccionando el método hasta obtener circuitos integrados con mayor cantidad de píxeles y que pueden tener diversas aplicaciones. Por ejemplo, en los trabajos [10] y [11] se presentan sensores CMOS basados en la arquitectura MIMOSA (de *Minimum Ionizing particule MOS Active pixel Sensor*). Estos sensores se desarrollaron con el objetivo específico de detección de radiación ionizante.

También existen desarrollos de sensores de radiación a través de APS comerciales. Perez *et al.* identificaron eventos producidos por partículas alfa en campos de radiación mixtos mediante el procesamiento de imágenes adquiridas con sensores comerciales CMOS[12] y desarrollaron detectores de neutrones térmicos con sensores APS cubiertos con una capa de  $\text{Gd}_2\text{O}_3$ [13]. Galimberti *et al.* utilizaron un sensor de imágenes comercial para realizar un detector de gas Rn en el ambiente[14]. En otro trabajo, Hizawa, *et al.* fabricaron un sensor que adquiere imágenes midiendo el pH con cada uno de los píxeles[15], pudiendo observar de fenómenos químicos en tiempo real.

Como se mencionó antes, una imagen digital es un arreglo de datos. Esto quiere decir que un sensor de imágenes con  $n$  píxeles de largo y  $m$  de ancho, captura  $n \times m$  datos en cada lectura. A su vez, para digitalizar valores, un circuito debe poseer, al menos, un conversor analógico-digital (ADC) de  $x$  cantidad de bits, lo que implica que cada dato estará compuesto por  $x$  dígitos binarios, es decir, un volumen importante de datos por cada lectura. Como ejemplo, un sensor comercial VGA, en su configuración más básica, posee 640 líneas horizontales y 480 verticales, con una resolución de 8 bits por cada pixel, lo que otorga 2.457.600 bits por cada lectura del sensor.[16] Si además se incorpora la cantidad de imágenes que se toman en función del tiempo (cuadros por segundo o fps), nos otorga un flujo de datos para nada despreciable.

Desde el punto de vista de la electrónica digital, para poder adquirir y transmitir grandes volúmenes de datos, se requiere de circuitos que sean capaces de operar a altas frecuencias de conmutación. El diseño de dichos circuitos no es trivial, ya que cuando las longitudes de onda de las señales presentes son comparables con las dimensiones físicas de dichos circuitos, debe considerarse el uso de líneas de transmisión[17]. Esto implica que no se puede diseñar utilizando un criterio de uniformidad en los parámetros y exige un análisis mas detallado y preciso.

Otro problema que presentan los circuitos electrónicos digitales tiene que ver con los tiempos de propagación de las corrientes y tensiones que circulan a través de ellos. Cuando se aplica un impulso en un conductor, debido a las capacidades propias de los materiales utilizados, las tensiones pueden demorar unos instantes en establecerse. Puede suceder que varias señales lleguen a los puertos de un dispositivo por conductores con distintas longitudes y generen retardos diferentes. Esto puede ocasionar un comportamiento indeseado si no se toman los recaudos adecuados.

Aún suponiendo un perfecto diseño, los circuitos digitales de alta velocidad se encuentran limitados en la frecuencia de conmutación por la temperatura que se necesita disipar. La potencia consumida por estos dispositivos es proporcional a la frecuencia de funcionamiento[18]. Parte de esta potencia se transforma en calor y produce un aumento en la temperatura. Si el incremento es indiscriminado, puede destruir los circuitos.

Una posible solución para disminuir la frecuencia de las señales sin perjudicar la tasa de transferencia es la incorporación de varios conductores para enviar datos en paralelo. La cantidad de conductores a través de los cuales circula la información, se denomina ancho de bus. Idealmente, para lograr una tasa de transferencia determinada, se podría disminuir la frecuencia tantas veces como conductores se agreguen. Por ejemplo, transmitiendo por cuatro filamentos, se podría enviar la misma información a un cuarto de la frecuencia que se necesitaría con uno solo de iguales características.

Existen distintas tecnologías para efectuar la lectura de los datos generados por los sensores y su posterior transmisión. La incorporación y evolución de microcontroladores permite capturar y procesar volúmenes crecientes de datos. Sin embargo, este tipo de dispositivos posee una

estructura rígida: su capacidad de procesamiento se encuentra limitada a una instrucción por ciclo de reloj y a un ancho de bus definido. Para aumentar los volúmenes de datos que circulan a través de ellos, no es posible aumentar el ancho de bus, sino que se torna necesario incrementar la frecuencia de funcionamiento, generando los problemas anteriormente detallados.

Una solución óptima, sin considerar los costos asociados a esto, sería el desarrollo de un circuito integrado de aplicación específica (ASIC del inglés *Application Specific Integrated Circuit*). En este tipo de circuitos, el diseñador elabora un circuito que puede operar a altas velocidades y, a su vez, obtener un ancho de bus sin restricciones, más que las dimensiones físicas del área donde será realizado el circuito. Sin embargo, cuando sí se considera el costo asociado a este enfoque, se vuelve una solución ineficiente en bajas cantidades. La manufactura de este tipo de dispositivos puede tener un costo de miles hasta cientos de miles de dólares, dependiendo del proceso de fabricación utilizado. Gran parte de estos costos son no recurrentes, es decir, solo se pagan una vez por proyecto. En grandes cantidades de dispositivos, este tipo de soluciones se vuelven más convenientes.

Otro enfoque, es la utilización de Arreglos de Compuertas Programables por Campo (FPGA, acrónimo del inglés *Field-Programmable Gate Array*). Un FPGA es un dispositivo electrónico que posee la capacidad de sintetizar casi cualquier circuito digital. En esencia, es una matriz de bloques lógicos (también llamadas *slices* o celdas lógicas, dependiendo del fabricante), que contienen Tablas de Verdad(LUTs o *Look-Up-Table*) y flip-flops (ff), entre otras cosas, y pueden ser interconectadas entre sí, según el criterio del usuario. Así, permite implementar una solución digital en un circuito físico, a diferencia de los microcontroladores, lo realiza a través de un algoritmo almacenado en una memoria, incorporando la ventaja de definir el ancho de bus necesario para relevar una gran cantidad de datos y transmitirlos a frecuencias de trabajo menores, además de ejecutar tareas en paralelo, disminuyendo los tiempos de procesamiento. A su vez, al ser implementado en un área muy pequeña, debido a la integración del sistema, este tipo de sistemas puede trabajar a frecuencias muy elevadas, lo que implica una mayor tasa de datos aún. A pesar de la gran diversidad de precios existentes en el mercado, una FPGA de costos menores a la centena de dólares suele tener muy buenas prestaciones para la mayor parte de las aplicaciones.

Existen diversas publicaciones en donde se observa el uso de FPGAs para la implementación de sistemas que producen imágenes. Por ejemplo, el desarrollo de un detector de radiación ionizante utilizando una sensor de imagen CMOS comercial. Para ello, los autores utilizaron una FPGA para configurar diversos parámetros del sensor con el fin de generar estrategias para la identificación de partículas alfa en campos de radiación mixtos y transmitir imágenes a una computadora personal (PC) a través de un puerto UART[19]. Se denomina ultrasonografía a la técnica de adquirir imágenes basándose en reflexiones de ultrasonido. Sus aplicaciones son múltiples, en las que se destaca el diagnóstico médico debido. Un trabajo reciente desarrolló un sistema que mejora la obtención de ecografías médicas con bajo costo utilizando una FPGA[20]. El autor presentó un algoritmo para la supresión de ruido de impulso en tiempo real para imágenes codificadas como JPEG 2000 realizado y probado en Matlab e implementado en una FPGA. Yanagisawa *et al*, desarrollaron un sistema con telescopios pequeños para explorar objetos de campo cercano con la finalidad de monitorear cuerpos celestes que puedan colisionar con el planeta[21]. En este trabajo, se aprovechó la velocidad de los circuitos implementados en FPGA para minimizar el tiempo de adquisición.

El desarrollo de nuevos sensores brinda a los investigadores un gran volumen de datos. En

muchos casos, la obtención de datos por si misma no otorga información, sino que es necesario procesar y analizar los mismos. La invención y evolución de las computadoras, como así también el desarrollo de nuevos algoritmos, dan lugar a procesamiento de datos cada vez más complejos en tiempos mucho menores. Las primeras ENIAC, computadoras de propósito general desarrollada en el año 1946 para el cálculo de tablas balísticas de las fuerzas armadas estadounidenses, podía ejecutar 20 operaciones cada  $10\text{ }\mu\text{s}$  [22], es decir, ejecutaba instrucciones con una frecuencia máxima de 200 kHz. A su vez, tuvo un costo aproximado de U\$S 500.000, pesaba 5 t y consumía 175 kW. En contraste con aquello, es posible conseguir en el mercado actual, computadoras con tamaño y peso reducido, que ejecutan instrucciones en cuenstión de nanosegundos, (5 ordenes de magnitud menos), consumen menos de 1 kW y cuestan algunos cientos de U\$S. A tal punto ha evolucionado esta tecnología, que se cuenta con computadoras muy potentes en casi cualquier laboratorio, oficina u hogar. La capacidad de cálculo que exhiben estos dispositivos, sumada al desarrollo de nuevos métodos y algoritmos de cálculo, permite a los investigadores procesar datos en tiempo reducido, facilitando el análisis y la generación de nueva información.

En todos los casos que se consideran en este trabajo, la generación de datos y el procesamiento de lo mismos se da en sistemas diferentes. Es decir, los datos son relevados por los sensores y adquiridos luego por los FPGAs. Finalmente llegan a una PC para su posterior procesamiento y análisis. Se requiere, por tanto, de una conexión a través de la cual los datos puedan ser transferidos del sistema de adquisición, la FPGA, a la PC y viceversa. Se torna de suma utilidad, entonces, proveer una comunicación efectiva y robusta que permita transmitir grandes volúmenes de datos en poco tiempo, y de esta forma facilitar los tiempos de desarrollo, pruebas, depuración, procesamiento y análisis.

La implementación de un sistema de comunicación en una FPGA puede ser resuelta de muchas maneras, quedando a criterio del desarrollador utilizar algún protocolo estándar, o bien diseñar uno propio. Sin embargo, en una computadora, las formas de comunicar datos se vuelven un poco más restrictivas y acotadas a los puertos y señales que puede manejar el equipo, conforme el fabricante haya establecido. Este trabajo busca implementar una comunicación entre una computadora personal y una FPGA, utilizando un protocolo estándar, que esté disponible en cualquier computadora comercial y que posea una tasa de bit suficiente para poder transmitir imágenes.

## **1.2. Protocolos disponibles para la transmisión de datos entre PC y FPGA**

El estándar más exigente de la norma americana de la SCTE (Sociedad de Ingenieros de Comunicación por Cable) utilizada Televisión Digital, posee una tasa de  $38.8\text{ Mbit s}^{-1}$ [23]. Por su parte, la serie de sensores para adquirir imágenes monocromáticas MT9M001, comercializado por ON Semiconductors posee  $1280\times 1024$  pixeles, con profundidad de 10 bits y puede operar hasta a 30 cuadros por segundo[24]. La tasa de transmisión necesaria es, por tanto, de  $393.2\text{ Mbit s}^{-1}$ .

Un requerimiento que posee cualquier periférico informático es el de compatibilidad. No es conveniente utilizar puertos que requieran acceso a la placa madre, como el caso de tarjetas de tipo PCI o PCI express, debido a que no todos los equipos los tienen accesible, como ser computadoras portátiles, y en algunos casos estos pueden estar todos ocupados. Se opta, entonces, por alguno de los tres puertos de moda: Ethernet, dedicado principalmente a conexión de redes

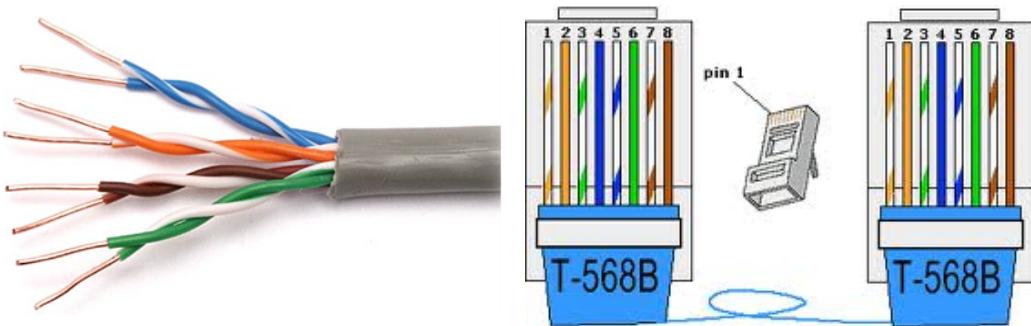


Figura 1.2: Par Trenzado y un dibujo de su ficha de conexión.

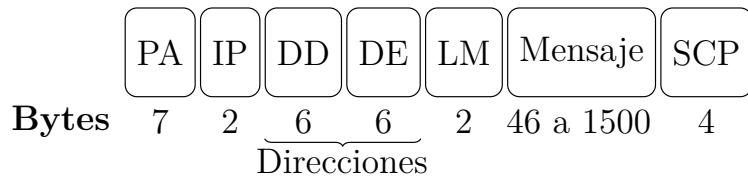
mediante cables; Wi-Fi, utilizado para el acceso a la red de forma inalámbrica; y USB, dirigido a la comunicación de periféricos con la PC.

Al hablar de Ethernet o Wi-Fi, se hace referencia a dos formas diferentes de conectarse a una red de computadoras. En otras palabras, se habla de dos o más nodos, compuestos por PCs o cualquier dispositivo electrónico con capacidad de realizar cálculo binario, que pueden intercambiar datos a través de una trama bastante compleja de componentes diferentes. Ambos protocolos hacen referencia solo a la conexión física de los dispositivos y el control de acceso de cada uno de ellos a la conexión. Quedando a cargo de otros sistemas, con sus protocolos, que los datos enviados puedan ser correctamente recibidos por el usuario de la PC. La gran diferencia entre ellos radica en el medio físico que utilizan: Wi-Fi emplea ondas electromagnéticas emitidas mediante radiofrecuencia, mientras que en Ethernet, estas ondas son acarreadas por uno o más conductores, como ser cable coaxial, cables de par trenzado o fibra óptica.

Ethernet, también conocido como IEEE 802.3, es una norma que define cómo se deben conectar nodos a través de conductores para conformar redes de área local (LAN o *Local Area Network*), es decir, redes pequeñas, como ser domésticas, de oficinas o de pequeñas empresas, de forma que puedan transmitir información a velocidades seleccionables entre 1 Mbit/s y 400 Gbit/s [25]. Utiliza una tecnología denominada Acceso Múltiple Sensando la Portadora con Detección de Colisiones (CSMA/CD del inglés *Carrier Sense Multiple Access with Collision Detection*). En una red con CSMA/CD, cada dispositivo debe sensar en forma permanente la conexión a la red, es decir, no existe un dispositivo que dirija el uso del bus, sino que cada uno debe identificar el estado de la red. Los mensajes se envían modulados. Cuando una señal portadora es detectada, todos chequean la dirección del paquete de información que viaja y el mensaje es recibido solamente por el dispositivo que se corresponda con esa dirección. Siempre que exista una señal portadora en el bus, los dispositivos que deseen transmitir información deberán esperar a fin de evitar colisiones, o sea, que dos dispositivos envíen mensajes a la vez y estos se interfieran.

Dependiendo de la frecuencia de la portadora y la tasa de transferencia a la que transporta el mensaje, la norma especifica el conector y la distancia máxima a la que debe conectarse una repetidora, es decir, un dispositivo que reciba, reconstruya y emita la señal recibida. Estos conectores pueden ser cable coaxial, fibra óptica o cable de par trenzado. Este último es el más usual en las PCs comerciales y se muestra, junto a su ficha característica en la Figura 1.2.

La información se estructura en paquetes para permitir la comunicación entre muchos nodos de la red. Un paquete, como se observa en la Figura 1.3, se compone de un preámbulo con 7 B que sirve para sincronizar los dispositivos en cada extremo de la conexión, 1 B de inicio,



### Referencias

**PA:**Preámbulo

**IP:** Inicio de Paquete

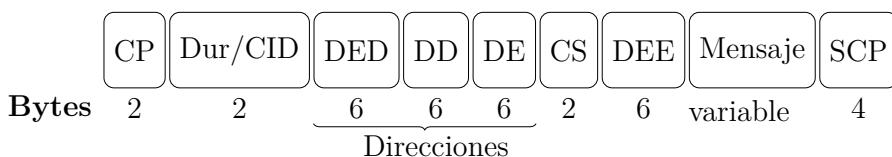
**DD:** Dirección de Destino

**DE:** Dirección de Emisión

**LM:** Longitud del Mensaje

**SCP:** Secuencia de chequeo del paquete

Figura 1.3: Estructura de un paquete Ethernet



### Referencias

**CP:** Control de Paquete

**Dur/CID:** Duración del paquete/Identificación de conexión

**DED\*:** Dirección de Enrutador de Destino

**DD\*:** Dirección de Destino

**DE:** Dirección de Emisión

**CS\*:** Control de Secuencia

**DEE\*:** Dirección de Enrutador de Emisión

**SCP:** Secuencia de chequeo del paquete

\*Pueden no estar dependiendo del tipo de mensaje

Figura 1.4: Estructura de un paquete Wi-Fi

12 B de direcciones, que corresponden 6 al nodo destinatario y 6 al emisor respectivamente, 2 B que indican la longitud del mensaje, entre 46 y 1500 B de datos y 4 B para la verificación de la transmisión. Otra definición importante de la norma, son las características eléctricas de las señales, pero no se detallan en este trabajo porque varían en función de la velocidad del puerto.

Por su parte Wi-Fi, perteneciente a la asociación de compañías denominada Wi-Fi Alliance, se rige por la norma que estableció esta última. Existe una norma equivalente, encuadrada en la especificación IEEE 802.11, referida a las redes de área local inalámbrica, o WLAN (siglas del inglés *Wireless Local Area Network*). Wi-Fi se enfoca en las que se refieren a las comunicaciones de radiofrecuencia con portadora de 2.4 GHz, que se incorporan en las revisiones b, g y n de la norma IEEE. IEEE 802.11 está pensado especialmente para dispositivos portátiles y móviles. La norma define a los dispositivos portátiles como aquellos que pueden ser trasladados con facilidad pero operan estáticos y los móviles se identifican por poder trabajar en movimiento [26]. La principal característica que posee este tipo de comunicación es la falta de conductores para la elaboración de la red, sin contar las conexiones entre los transceptores que emiten y reciben las

señales de radiofrecuencias y los nodos, en donde la información es producida y/o consumida. En cuanto al formato del paquete de datos, el cuál se muestra en la Figura 1.4, es bastante similar al de Ethernet. En primer lugar, se envían dos bytes de control que indican el tipo de paquete a enviar. Luego siguen dos bytes que, dependiendo de la etapa de la comunicación puede indicar la duración del mensaje a transmitir o un identificador de una conexión establecida previamente. Siguen entre 6 y 18 bytes de direcciones del enrutador que recibe los datos, el nodo emisor y el destinatario. Continúan, dos bytes de control de secuencia se utilizan para fragmentar transmisiones largas. Continua un campo más para dirección que corresponde a la red emisora de 6 bytes. Todos los campos de dirección pueden variar en función del tipo de mensaje que se envía. Los últimos dos campos de la trama corresponden a la información que se quiere comunicar (hasta 2312 bytes) y un código de chequeo por redundancia cíclica de 32 bits (4 bytes).

Existen múltiples ventajas de utilizar radiofrecuencias para conectarse a la red, tales como la libertad de mover el punto de trabajo y la economía a la hora de armar redes con muchos nodos. Sin embargo, posee algunas desventajas notorias, propias del medio de propagación, que lo hacen no tan óptimo para los fines del presente trabajo. Las redes inalámbricas tiene la característica de que no son del todo confiables: posee múltiples fuentes de interferencia, ya que varias tecnologías que utilizan la misma frecuencia (Bluetooth, Zig-Bee, WUSB, microondas). A su vez, suele presentar variaciones temporales y asimetrías en las propiedades de propagación, lo que puede provocar interrupciones en la comunicación.

Ambos protocolos proporcionan una solución de conexión de redes de nivel físico y ejecutan tareas de control de acceso al medio (MAC) a fin de evitar colisión en los datos, es decir, que dos dispositivos transmitan en forma simultánea e interfieran la comunicación. Sin embargo, para establecer una red, faltan componentes físicos y lógicos tales como un sistema de control enlace lógico (Logic Link Control), un sistema de direccionamiento, como el Protocolo de Internet (IP), una capa de transporte de datos, (como el protocolo TCP) y las capas de software que permiten acceder a los protocolos anteriormente mencionados.

A pesar de lo anterior, es posible establecer comunicaciones punto a punto con ambos protocolos, simplificando mucho el sistema de transmisión de datos. Sin embargo esta solución presenta un inconveniente no menor: se le quita a la PC un acceso a la red, que en la mayoría de los casos es el único. Esto no es deseable ya que la conectividad es un requisito fundamental en cualquier hogar u organización, ya sea empresarial, gubernamental, científica o de cualquier tipo.

Por su parte el protocolo USB (acrónimo de *Universal Serial Bus*), es una norma desarrollada por seis de las empresas más grandes de la industria informática, pensada y desarrollada para la conexión de teléfonos y periféricos a PCs [27]. En la versión original, USB posee conectores cableados de 4 conductores y presenta una topología de bus, es decir todos los dispositivos se conectan a un mismo circuito conductor. La conexión es manejada por una PC y solo transmite o recibe un dispositivo a la vez. Tal fue la penetración de USB en el mercado, que se transformó en una norma de facto. Actualmente es incorporada casi por defecto en casi todas las computadoras disponibles en el mercado y es necesaria a la hora de comprar e instalar periféricos.

USB presenta diferentes versiones de su norma, cada cual con una o más tasas de transmisión y señalización. La versión 1 posee dos revisiones, 1.0 fue lanzada al mercado en el año 1996 y 1.1 que se presentó en Agosto de 1998. La primera alcanza una tasa máxima de  $1.5 \text{ Mbit s}^{-1}$  y la segunda hasta  $12 \text{ Mbit s}^{-1}$ . USB 2.0 fue presentado en Septiembre del 2000 y es capaz

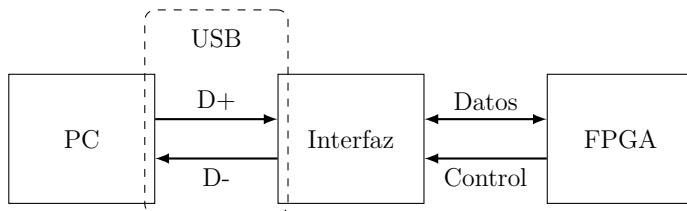


Figura 1.5: Esquema propuesto para implementar la comunicación

de transmitir a  $480 \text{ Mbit s}^{-1}$ . La tercera versión, USB 3.0, fue lanzada al mercado en 2011 y transmite a una tasa de  $5 \text{ Gbit s}^{-1}$ . Esta última versión fue revisada en julio de 2013 y en septiembre de 2017, ofreciendo  $10 \text{ Gbit s}^{-1}$  y  $20 \text{ Gbit s}^{-1}$  respectivamente.

Se elige para el desarrollo de este trabajo la norma ya que USB 2.0 presenta una tasa de transferencia de datos suficiente para la transmisión de imágenes. A su vez, resulta ideal para los objetivos buscados debido a encontrarse presente en la mayoría de las computadoras y no interferir en la conexión a internet de las mismas. En el Capítulo 2 se profundizarán en conceptos específicos de la norma USB.

Es posible implementar una comunicación USB completa a través de una FPGA. Sin embargo, esto sería demasiado oneroso en términos de tiempos de desarrollo y de recursos de FPGA disponibles para la implementación de otros sistemas, los cuales son el objetivo de la comunicación. Se plantea, entonces, un esquema como el que se observa en la Figura 1.5 en la cual se utiliza una interfaz externa al FPGA. La comunicación USB propiamente dicha será efectuada entre la interfaz y la PC, mientras que se plantea una comunicación diferente entre la interfaz y el FPGA. Este último, por su parte, tendrá la tarea de realizar el control de esta comunicación.

## 1.3. Objetivos

### 1.3.1. Objetivo Principal

El objetivo del presente trabajo es obtener una comunicación USB 2.0 de alta velocidad entre una PC y un FPGA.

Esta comunicación debe realizarse y documentarse de forma tal que pueda ser usado posteriormente en aplicaciones científicas desarrolladas con FPGA's.

### 1.3.2. Objetivos Particulares

Para la consecución del objetivo general, se deben cumplir los siguientes objetivos particulares:

- Comprender el funcionamiento del protocolo USB.
- Seleccionar los componentes a utilizar.
- Configurar los componentes seleccionados.
- Desarrollar un núcleo en VHDL que sirva de interfaz.
- Diseñar e implementar la interconexión de los componentes seleccionados.

- Verificar el sistema desarrollado.
- Desarrollar un documento que explique el modo de uso del código VHDL utilizado.

## 1.4. Estructura del Informe

El presente informe se divide en 2 bloques principales: uno referido al desarrollo del sistema y el siguiente a su forma de uso y verificación.

Dentro del bloque referido al desarrollo del sistema, se encuentran los primeros 5 capítulos:

1. **Introducción:** En este capítulo se intenta exponer lo que motiva el presente trabajo, la propuesta que da solución a la motivación, el objetivo y alcance que el trabajo busca y la estructura del mismo. Se brindan, además, conceptos importantes de la norma USB que son significativos para los objetivos de este trabajo.
2. **Elección de las herramientas para la realización de la interfaz:** Se describe aquí todas las herramientas de las que se vale este trabajo para cumplir con los objetivos propuestos.
3. **Interfaz USB:** Se presenta la arquitectura, configuración y código desarrollado para el presente trabajo, como así también las herramientas específicas provistas por el fabricante, que facilitan el desarrollo.
4. **Desarrollo del sistema maestro para la comunicación entre la FPGA y la interfaz Cypress:** Este capítulo detalla lo desarrollado para implementar la comunicación entre la FPGA y la interfaz. Se expone una máquina de estados descrita en VHDL y sintetizada en FPGA. También se describe un circuito impreso realizado para conectar ambas partes.
5. **Verificación y validación del sistema:** Se desarrolla las tareas desarrolladas a fin de realizar las depuraciones del sistema y la verificación del cumplimiento de las especificaciones.

## 1.5. Sumario del capítulo

En el presente capítulo se expone la necesidad de la elaboración de un sistema de comunicación que permita la transferencia de datos entre una PC y un FPGA para ser utilizados por sistemas implementados con este último dispositivo.

Se plantea una solución utilizando una interfaz comercial que sirve de intermediario entre estas herramientas y se brinda una justificación del empleo del protocolo USB 2.0 de alta velocidad como la implementación óptima del sistema.

Se presenta también la estructura del presente informe y se dan detalles relevantes para este trabajo de la norma USB.

## Capítulo 2

# Bus Serial Universal 2.0

El Bus Serial Universal, o USB por sus siglas en inglés, es un sistema de comunicación diseñado durante los años 90 por seis fabricantes vinculados a la industria informática, Compaq, Intel, Microsoft, Hewlett-Packard, Lucent, NEC y Philips, con la idea de proveer a su negocio de un sistema que permita la conexión de PCs con teléfonos y periféricos con un formato estándar, fácil de usar y que permita la compatibilidad entre los distintos fabricantes.

Hasta ese momento, el gran ecosistema de periféricos, sumado a los nuevos avances y desarrollos, hacia muy compleja la interoperatividad de todos ellos. Cada uno de los fabricantes desarrollaba componentes con características, tales como fichas, niveles de tensión, velocidades, drivers, lo cuál dificultaba al usuario estar al día y poder utilizar cada componente que compraba. Esto también complicaba a las mismas empresas productoras, por que la introducción de un nuevo sistema requería de mucho soporte extra para poder conectar lo existente en forma previa.

Todo esto, quedó saldado con el aparición de la norma USB que, gracias a la gran cuota de mercado de sus desarrolladores, fue adoptada en forma rápida y se transformó en la especificación casi por defecto a la hora de seleccionar un protocolo para periféricos. La penetración en el mercado fue tal que hoy, más de 20 años después, es difícil encontrar PC con otro tipo de puertos, salvo que en el momento de compra se solicite de manera especial. No obstante, cualquier PC nueva disponible en el mercado debe poseer puertos USB para la conexión de, al menos, los periféricos.

El diseño de la norma USB busca resolver tres problemáticas interrelacionadas, que son: La conexión de teléfonos con las PC, la facilidad de uso, es decir, que el usuario solo conecte su dispositivo y pueda utilizarlo, y la expansión en la cantidad de puertos disponibles para conectar periféricos[27]. Para satisfacer estas tres demandas, la norma USB 2.0 busca alcanzar un conjunto de metas que apuntan a la facilidad del uso, la compatibilidad entre versiones diferentes de la misma tecnología, la robustez en el flujo de datos, y la convivencia de diferentes configuraciones temporales en único bus. Para alcanzar estas metas, la norma provee una interfaz estándar, ancho de banda que soporte comunicaciones audiovisuales de calidad aceptable y un bajo coste.

El presente capítulo intenta ser un breve resumen con los aspectos más relevantes de la norma en cuanto a su composición física, su topología, los dispositivos que intervienen, la importancia de los mismos y como los datos son transmitidos desde y hacia una PC.

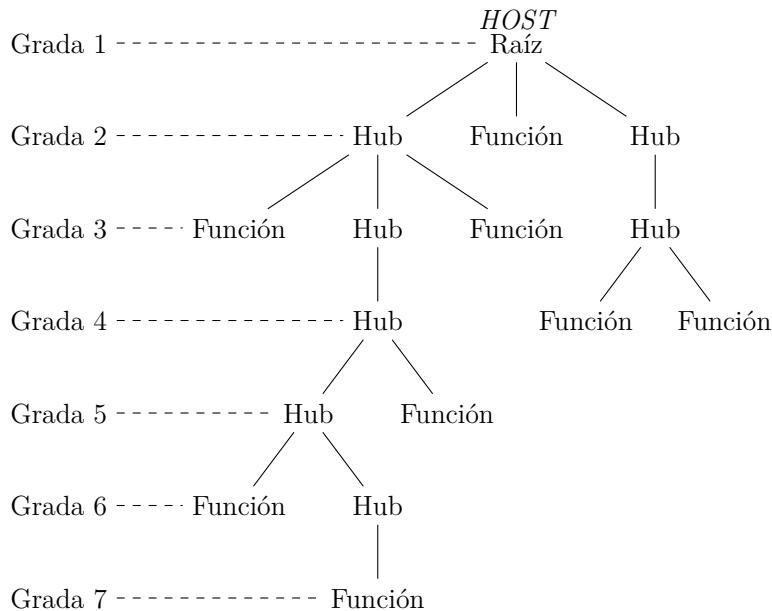


Figura 2.1: Topología de un sistema USB

## 2.1. Descripción general de un sistema USB

Un sistema USB posee un esquema en forma de árbol cuyo nodo principal es el host. Es decir, la comunicación se realiza siempre a través de una única línea a la que se conectan todos los dispositivos (bus). Dado el campo de direcciones provisto por la norma, un sistema USB puede conectar hasta 128 dispositivos. El acceso al bus es administrado por un maestro. El maestro se encarga de solicitar a cada uno de los dispositivos su intervención. Posteriormente, el dispositivo debe responder al pedido del maestro. Este esquema es lo que se conoce como maestro-esclavo. De esta forma, el sistema se asegura que el bus sea utilizado por un dispositivo a la vez para enviar o recibir datos.

En un sistema USB no cualquier dispositivo puede ser maestro. Este rol lo cumple solo uno: una PC, o cualquier dispositivo con capacidad de llevar a cabo las tareas asignadas (que se detallan más adelante); denominado Host por la norma. La palabra *HOST* proviene del habla inglesa y se traduce como anfitrión, aunque en la jerga se conoce comúnmente por su nombre en inglés.

La topología del bus, que se observa en la Figura 2.1, posee forma de árbol, es decir, puede ser pensada como una comunicación vertical, donde en el punto más alto se encuentra el Host. Siguiendo hacia abajo, el bus puede encontrar dos tipos diferentes de dispositivos: Funciones, cuyo rol es el de proveer una utilidad al sistema, como ser la de captura de imagen, reproducción de audio o el ingreso de comandos; y Hubs (concentradores o distribuidores), que se encargan de conectar una o más funciones al sistema. La norma USB establece gradas, en donde cada Hub introduce una nueva grada que contiene a las Funciones conectadas. En otras palabras, las gradas configuran una suerte de distancia lógica entre las Funciones y el Host, separada por Hubs. Por cuestiones de restricciones temporales y tiempos de propagación en los cables, no se permiten más de 7 gradas, incluyendo al Host en la primera. Es decir, no se puede conectar más de 5 Hubs en cascada. La grada 7 sólo puede contener Funciones[27].

Cada uno de estos dispositivos diferentes, se interconectan entre sí a través de cables y

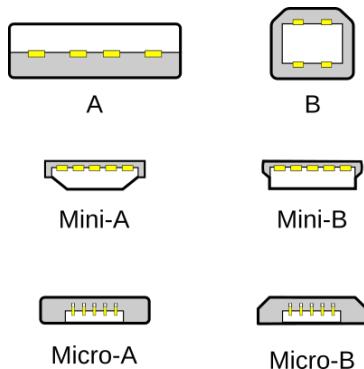


Figura 2.2: Tipos de conectores USB. Los tipo A deben ser usados en el extremo del Host y los tipo B hacia los periféricos[28]

conductores específicos, diseñados en forma tal que no sea posible conectarlos en forma equivocada. Para cumplir con la norma, el Host debe tener siempre un zócalo compatible con conectores tipo A y los periféricos, de tipo B. Se observan las diferencias entre uno y otro en la Figura 2.2. Los cables de conexión poseen dos pares de conductores: uno para la señal de alimentación de 5 V ( $V_{BUS}$  y  $GND$ ) y otro para el flujo de datos ( $D+$  y  $D-$ ).

A nivel eléctrico, la interconexión de datos en los dispositivos se lleva a cabo a través de una codificación de inversión sin retorno a cero, es decir, el cambio de nivel de tensión representa un '0' y la invariabilidad, un '1'. Además, la señal de datos es diferencial. Esto implica que cuando  $D+$  es positivo,  $D-$  debe ser negativo y viceversa.

Cabe destacar que, al tener una señal diferencial, la norma USB es half-dulpex , es decir, puede transmitir en los dos sentidos (desde Host hacia Funciones y viceversa), pero no puede hacerlo en simultáneo[29], sino que primero debe transmitir un dispositivo y, al finalizar este, el bus queda liberado para que otros dispositivos puedan transmitir.

La velocidad de conmutación en los niveles de tensión de la señal de comunicación puede darse a diferentes valores, dando lugar a tres tipos de tasa de bit: Alta-velocidad (*High-Speed*) implica una tasa de bit de  $480 \text{ Mbit s}^{-1}$ , Velocidad-completa (*Full-Speed*) posee una tasa de bit de  $12 \text{ Mbit s}^{-1}$  y baja-velocidad (*Low-Speed*) transmite a una tasa de bit de  $1.5 \text{ Mbit s}^{-1}$ .

Cuando el Host se comunica con las diferentes Funciones, lo realiza a través de paquetes. Los paquetes implican que la información que se transmite a través del bus está encapsulada en un formato establecido. Cada vez que un dispositivo accede al bus, lo debe hacer de una manera particular, definida por el tipo de transferencia, por su rol (Host, Hub o Función) y por el estado de la transmisión dentro del protocolo establecido.

## 2.2. Dispositivos que componen un sistema USB

Dentro de un sistema USB existen tres tipos diferentes de dispositivos: Host, Hubs y Funciones. Cada uno de ellos tiene asignado un rol específico dentro de la comunicación. Se detallan a continuación las tareas pertinentes a cada uno de ellos.

### **2.2.1. Host USB**

El Host es quien comanda las comunicaciones. Este dispositivo debe tener capacidades de memoria y procesamiento necesarias para almacenar y ejecutar el software de control. A su vez, necesita de hardware que le permita llevar un monitoreo y control de los eventos que suceden en el bus. Entre las tareas que debe llevar a cabo, se encuentran:

- Detectar la conexión y desconexión de dispositivos.
- Administrar el flujo de los comandos de control con los diferentes dispositivos.
- Administrar el flujo de la información entre él (Host) y los diferentes dispositivos.
- Llevar estadísticas de actividad y estado del bus.
- Proveer potencia a los dispositivos conectados, cuando estos así lo requieran.

Debido a que las tareas que ejecuta el Host requiere una cantidad de recursos de almacenamiento y procesamiento, es usual que el sea una PC la que lleve el rol. El Host es quien inicia la comunicación con las Funciones. Las Funciones, a su vez, responden a lo que fue solicitado por el Host, cuando él lo indique.

### **2.2.2. Hubs USB**

Un Hub USB tiene la función de proveer puertos al bus. El primer Hub esta incorporado en el Host y cada vez que se requiere más puertos a los cuales incorporar periféricos, se puede ir agregando a través de Hubs. Otra función importante es la de servir como interfaz entre dispositivos con diferentes velocidades, optimizando así el ancho de banda disponible para la comunicación.

### **2.2.3. Funciones USB**

La norma define como Función a todo aquel dispositivo que se conecta al bus y brinda al Host la capacidad de realizar una nueva tarea. Por ejemplo, un teclado otorga un método de entrada adicional, un mouse permite manejar un puntero de la interfaz gráfica, un parlante y un micrófono posibilitan la emisión y recepción de sonidos, respectivamente. Cada una de estas utilidades, compone una Función USB. A su vez, un dispositivo que brinda más de una capacidad es visto por el Host como Funciones separadas conectadas a través de un Hub. Por ejemplo, si se piensa en unos auriculares con micrófono, aunque se presenten integrados en un mismo producto y tengan un único puerto de conexión al bus, el Host los considera como dos Funciones separadas. Las Funciones, desde un punto de vista de software, son independientes unas de otras, por lo que cuando un programa, llamado cliente, necesita utilizar una de ellas, puede acceder a ésta directamente sin conocer cuantas y cuales funciones diferentes existen en el bus.

Cada Función se compone de un conjunto de extremos. Un extremo es una porción de dispositivo identificable en forma unívoca<sup>[27]</sup>. Cada extremo tiene características definidas por el diseñador del sistema que deben estar adecuadas a los requerimientos de cada dispositivo. Los extremos tienen un solo sentido de comunicación y un tamaño máximo de mensaje a transmitir

o recibir. Cuando se conecta al bus, un dispositivo debe enviar una descripción en donde consten sus extremos y las diferentes formas de configuración de cada uno, con el tipo de mensajes que soporta, el sentido de la comunicación, el tamaño, entre otros parámetros. Esta descripción se lleva a cabo través de lo que la norma llama descriptores.

Todo dispositivo debe contener un extremo con dirección cero dedicado exclusivamente al control de la Función por parte del Host. Debe, como mínimo, poder comunicarse a velocidad completa, es decir, con una señal de  $12 \text{ Mbit s}^{-1}$  y, a su vez, responder a los comandos de control básicos cómo adquirir la dirección, recibir la configuración y enviar los descriptores del dispositivo y sus diferentes configuraciones. Dependiendo de los diferentes requerimientos, el dispositivo puede incorporar otros extremos (15 de entrada y 15 de salida como máximo). Cada extremo no-cero tiene diferente latencia, acceso al bus, ancho de banda, manejo de errores, tamaño máximo de paquete soportado y dirección.

## 2.3. Paquetes USB

Los dispositivos transmiten información a través del bus con un formato particular, establecido por el protocolo que dicta la norma USB. Cada 1 ms, el Host debe emitir una señal de sincronismo. El intervalo que transcurre entre una señal y la siguiente, se denomina cuadro. El Host asigna una porción de cuadro a cada uno de los dispositivos, asignando ancho de banda y tiempos de retardo a cada uno, según los requerimientos. A su vez, en comunicaciones de Alta-Velocidad, cada cuadro se subdivide en 8 microcuadros de  $125 \mu\text{s}$  cada uno. Los fragmentos de información que envían los dispositivo mientras transcurre un cuadro, se denominan paquetes. Un paquete está compuesto por diferentes campos. El sistema reconoce cada campo, decodifica su información e identifica cada paquete, su emisor, el tipo de datos que envía, el sentido de circulación. Luego, corrobora que los datos transmitidos llegaron a destino en forma satisfactoria.

### 2.3.1. Campos de paquetes

Existe un número finito de campos y todos pueden resumirse en el presente documento. Sin embargo, se detallan a continuación los que el autor considera más relevantes para el objetivo de este trabajo, quedando de lado algunos comandos, por ejemplo, inherentes a los hubs que conectan dispositivos de diferentes velocidades.

#### Identificador de paquete

El campo identificador de paquete (PID del inglés *Packet Identifier*) le da a conocer a los distintos dispositivos el tipo de información que contiene el paquete. Por ejemplo, indica si el Host solicita envío o recibo de datos, si envía un comando o si un dispositivo está transmitiendo los datos. Se compone de un campo de 8 bits, de los cuales 4 corresponden al identificador propiamente dicho y los otros cuatro son el complemento a uno de los mismos datos, permitiendo corroborar que no hubo una pérdida de información.

Existen 4 tipos de PID: Token, que antecede a cualquier transmisión y es emitido por el host; Data, indica paquetes que contienen datos transmitidos; Handshake, a través del cual los componentes del sistema se enteran si la comunicación fue efectiva o no y Special, cuya función no es de interés para este trabajo.

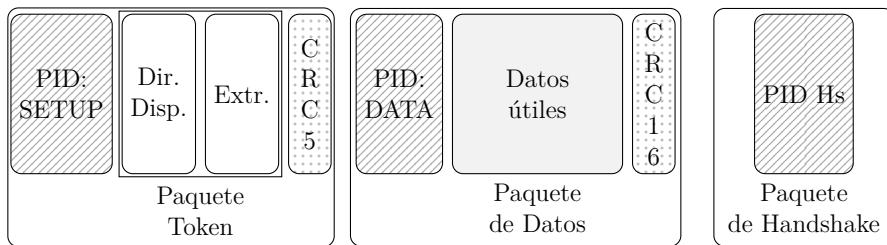


Figura 2.3: Formatos de paquetes

A su vez, los PID Token se dividen en 4 tipos: IN, para indicar que se va a realizar una envío de datos desde un extremo al Host; OUT, antecede a una transmisión de datos en el sentido contrario, es decir del Host a un extremo; SETUP, que señala una secuencia de comandos y SOF (del inglés Start of Frame) que emite una señal de inicio de cuadro, utilizada para sincronismo y control.

Dentro de los PID Data, solo existen diferentes etiquetas que se usan dependiendo del tipo de transmisión. Los PID de Handshake contienen 4 mensajes diferentes: ACK para indicar que el mensaje fue recibido satisfactoriamente y NAK señala que no se pudo enviar o recibir, STALL significa que el extremo se detuvo y NYET de cuenta sobre demoras en la respuesta del receptor.

## Dirección

El campo de Dirección señala cuál es la Función que debe responder o recibir alguna directiva emitida por el host. A su vez, se divide en dos subcampos: uno que indica un dispositivo y la segunda que señala el extremo específico con el cual desea comunicarse.

## Datos

Es el campo que contiene la información útil transferida. Puede tener un largo de hasta 1024 bytes. Cada byte enviado se ordena con el bit menos significativo (LSb del inglés *Less Significative bit*) primero y el bit mas significativo (MSb por sus siglas en inglés) al final.

## Chequeos de redundancia cíclica

El campo de chequeo de redundancia cíclica (CRC) contiene verificadores para corroborar que no hubo pérdida de información. Dependiendo de que tipo de paquete se esté transmitiendo, el CRC puede tener 5 o 16 bits.

### 2.3.2. Formato de paquetes

Cada uno de los paquetes que intervienen en la comunicación USB utilizan diferentes tipos de campos, dando lugar a distintos tipos de paquetes. La figura 2.3 muestra como se conforman algunos de ellos.

Un paquete de tipo Token está conformado por los campos PID, Dirección y CRC-5 (CRC de 5 bits). Un paquete Token que indica SOF en su campo PID, lleva un formato un poco diferente. En lugar de la dirección, se envía un contador de 11 bits que señala la cantidad de cuadros que han transcurrido desde la puesta en marcha del sistema, seguido de un código CRC-5.

Cada 1 ms el host transmite un SOF e incrementar el contador de cuadros. En sistemas USB 2.0 de Alta velocidad, además, se transmiten 8 subcuadros de 125  $\mu$ s por cada cuadro. Cada uno de estos subcuadros inicia con un paquete SOF. Sin embargo, el host no actualizará el número de cuadros hasta pasado 1 ms.

El paquete de datos iniciará con un PID que indique que es un paquete de este tipo, luego enviará los datos desde el LSb hasta el MSb y, finalmente, enviará un código CRC-16 (CRC de 16 bits de longitud).

Los paquetes de tipo Handshake (Hs) solo envía un PID con información sobre si el mensaje fue recibido en forma correcta o no.

## **2.4. Tipos de Transferencias**

Cada extremo presente en un dispositivo USB, puede estar configurado, en simultaneo, con un solo tipo de transferencias. Es importante, para el diseñador del dispositivo, entender y seleccionar el tipo de transferencia adecuada para cada uso debido a que, de ello depende las características que poseerán las comunicaciones que se efectúen.

Existen cuatro tipos de transferencias definidas por la norma USB: Transferencias de Control, transferencias en masa, transferencias isocrónicas y transferencias de interrupción. Cada una de ellas tiene un propósito y características diferentes, las que se detallan a continuación.

### **2.4.1. Transferencias de control**

Las transferencias de control son utilizadas por el Host para configurar, emitir comandos y conocer el estado de los distintos dispositivos acoplados al bus. Se caracteriza por ser una comunicación de ráfagas, es decir, de corta duración, tener alta prioridad y ser no periódica. Habitualmente, son utilizadas para emitir comandos hacia los dispositivos, o bien, para conocer su estado. Sin embargo, esto no quiere decir que pueda ser empleada para transmitir mensajes que no sean específicamente de comando. Debido a la sensibilidad que los mensajes de control poseen para el sistema USB, estos están dotados del protocolo más estricto de chequeo, corrección y/o retransmisión de datos.

Las transferencias de control poseen dos o tres etapas en su ejecución. En la primera de ellas, el Host debe enviar un Paquete Token que indique SETUP, luego envía un paquete Data con 8 B y esto es respondido por el dispositivo con un paquete Handshake indicando la recepción. Si hiciese falta enviar información extra, en una segunda etapa, el Host transmitirá un paquete Token indicando la necesidad de información. Luego, dependiendo del sentido de los datos solicitado, se enviará un paquete Data con hasta 64 B más y el receptor responderá con un paquete Handshake. Finalmente, en la última etapa, se le permite al dispositivo informar su estado. Para ello, el Host le envía un paquete Token de solicitud de datos, luego la Función responderá con un paquete Data y el Host emitirá con un paquete Handshake, indicando si recibió o no la información.

### **2.4.2. Transferencias en masa**

Las transferencias en masa son usadas para transferir paquetes grandes en forma de ráfagas, en forma no periódica. Su utilidad consiste en que aprovecha al máximo cualquier espacio

de ancho de banda disponible. Gracias al sistema de chequeo de errores, es posible solicitar retransmisiones, asegurando la integridad de la comunicación. Esta transferencia es ideal para comunicar cantidades relativamente grandes de datos que requieren una comunicación fidedigna a costa de sacrificar velocidad en los tiempos de entrega, por ejemplo, una impresora. En un bus que no posee un gran uso, los mensajes alcanzarán el destino en tiempos cortos. Sin embargo, cuando exista una gran cantidad de dispositivos conectados y el ancho del bus se encuentre congestionado, un mensaje largo puede verse demorado.

Cuando se lleva a cabo una operación de este tipo, el Host envía un paquete Token de tipo OUT cuando desea transmitir datos o IN si desea recibirlas, la dirección de la Función y su extremo. Luego, el emisor comunica un paquete Data, y finalmente, el receptor de la transferencia responde con un paquete Handshake. Una transferencia en masa (*bulk transfer*) puede poseer un tamaño máximo de 512 B de datos por paquete transmitido.

#### **2.4.3. Transferencias isocrónicas**

El término isócrono o isocrónico está referido a sistemas digitales sincrónicos con la particularidad de que se supone que sucede una cantidad determinada de sucesos en intervalos regulares de tiempo. Esto puede ser logrado compartiendo la misma fuente de sincronismo, o bien, sincronizando los relojes de cada componente.

En un sistema USB, el Host envía una señal SOF por cada cuadro de 1 ms y por cada subcuadro de 125 µs, en los sistemas de alta velocidad. Es posible sincronizar sistemas que poseen fuentes de reloj diferentes a través de la captura de esta señal. Esto permite tener este comunicaciones de tipo isocrónico, aún con señales de reloj provenientes de fuentes diferentes.

La principal característica de las transferencias isocrónicas es que son periódicas y continuas entre el Host y las Funciones. Se utiliza este tipo de transferencias para comunicar datos que pierden validez cuando no son entregados en un tiempo establecido. Para lograr esto, el Host asigna una porción fija de ancho de banda por cada cuadro (1 ms) a cada dispositivo que se comunique por transferencias de tipo isocrónicas. Gracias a que los datos pierden su validez a lo largo del tiempo, también los errores la pierden, por lo que no se prevé una retransmisión de los datos enviados por este sistema.

La ejecución de una transferencia isocrónica se da cuando el host envía un paquete Token con la dirección de un extremo de este tipo de transferencias. Luego, el emisor envía un paquete Data cuyo campo de datos puede poseer hasta 1024 B y un CRC-16. Finalmente el receptor envía un paquete Handshake. Si, dado el caso, el receptor envía un Handshake indicando que el paquete no pudo ser recibido en forma correcta, el mensaje es descartado, sin existir una retransmisión posterior del mismo paquete.

#### **2.4.4. Transferencias de interrupción**

Cuando se requiere de una comunicación cuya demora en la entrega de datos sea menor que un tiempo máximo y que, a su vez, posea una baja probabilidad de ocurrencia, el tipo de transferencia óptimo para utilizar, son las transferencias de interrupción. En este tipo de transferencias, el Host consulta cada un periodo de tiempo determinado el estado de los extremos que se encuentran configurados para efectuar este tipo de transferencias. Para ello, envía un paquete Token, luego el emisor transmite un paquete de datos con hasta 64 B, si se trata de

dispositivos de velocidad completa, y 1024B, en el caso de una comunicación de alta velocidad. Finalmente, el receptor responderá con un paquete Handshake.

## 2.5. Descriptores

Cuando un dispositivo es conectado al bus, debe informar sus características al Host a través de descriptores. Un descriptor es una estructura de datos con formato definido. De esta forma, el sistema conoce las diferentes configuraciones que puede tener cada una de las Funciones conectadas. El conocimiento detallado de estos descriptores por parte de los diseñadores de dispositivos, facilita luego la tarea de selección de cada uno de los atributos que tendrá, como así también, la elaboración de software de control en la PC.

Cada uno de los descriptores comienzan con su longitud en bytes y el tipo de descriptor que se está enviando. En orden jerárquico, se utilizan categorías de descriptores que van desde los atributos generales a los particulares. En primer lugar, se envía el descriptor DEVICE que informa la versión de la norma USB que cumple el dispositivo, un número que identifica al fabricante y otro que corresponde al producto, es decir al dispositivo. Esto le permite saber al Host que software de control debe utilizar para comunicarse con el dispositivo. A su vez, comunica la cantidad de posibles configuraciones. Luego, si el dispositivo cumple con la norma 2.0 (o más moderna) envía un descriptor de tipo DEVICE\_QUALIFIER con información sobre otras velocidades de comunicación soportadas.

El protocolo USB diferencia una configuración de otra dependiendo de las necesidades de energía. Un dispositivo podría operar conectado a una fuente de energía externa, o bien, ser alimentado por el mismo bus. Si las potencias de la fuente y del bus son diferentes, podrían verse limitadas las utilidades que ejecutaría la Función. Entonces, cuando el dispositivo funcione con la fuente podría tener una configuración pero cuando se desconecta, deberá informar esta situación al Host, indicando que se debe cambiar la configuración. Esta comunicación se lleva a cabo a través del descriptor de tipo CONFIGURATION. Debe haber tantos descriptores de este tipo como se indicó en el descriptor DEVICE.

Debido a que cada configuración puede tener diferentes limitaciones en sus funciones dependiendo de la potencia que consuma, se establece que cada configuración tenga a su vez diferentes interfaces. La cantidad de interfaces que tiene una configuración, también debe estar informada en el descriptor CONFIGURATION.

Una interfaz puede verse como el conjunto de extremos que son utilizados por un dispositivo para realizar una función específica. Por ejemplo, se podría pensar en una impresora multifunción. Se puede tener una interfaz para la parte que imprime y otra para el scanner. A su vez, cada interfaz puede variar el ancho de banda requerido a través de los denominados AlternateSettings. Las interfaces y sus diferentes alternativas, se comunican al Host a través del descriptor de tipo INTERFACE.

A su vez, un extremo define la dirección de la comunicación, es decir, si es desde o hacia el Host, un tipo de transferencia, si la comunicación es sincrónica o no, el tamaño máximo de paquete y el ancho de banda necesario. Los extremos se describen a través del descriptor ENDPOINT.

En resumen, la comunicación entre los dispositivos y el Host se efectúa a través de los extremos. Los extremos, a su vez, se agrupan en interfaces y un grupo de interfaces conforman una configuración. Una característica a tener en cuenta es que un dispositivo puede tener

diferentes interfaces activas a la vez y las interfaces pueden cambiar durante la operación de características alternativas (AlternateSetings). Sin embargo, para cambiar de configuración, todos los extremos y las interfaces se desactivan.

También existe un tipo de descriptores, denominados STRING, que sirven para colocar a cada uno de los atributos una forma legible por el usuario, aunque puede no ser utilizada.

## **2.6. Sumario del Capítulo**

En el presente capítulo se abordaron las partes más relevantes a los fines de este trabajo que corresponden a la norma USB. Resulta importante comprender la norma a fin de lograr una correcta configuración de las partes involucradas en cada etapa del desarrollo.

A partir de la descripción global de un sistema, se pudo comprender cómo identificar los diferentes puertos que intervienen en la norma, las velocidades de operación de un sistema USB, las tensiones que intervienen, la codificación usada y la topología del bus. Se identificaron a su vez, los dispositivos que intervienen y el rol que ocupan dentro del sistema.

Se desarrollaron conceptos sobre los componentes más importantes del protocolo, la forma en que los mensajes se empaquetan, los campos y etiquetas que componen un paquete. Se detallaron los tipos de transferencias USB y sus características. También, se abordaron los descriptores a través de los cuales los dispositivos comunican sus características al Host.

## Capítulo 3

# Elección de las herramientas para la realización de la interfaz

En el Capítulo 1 se justificó la elección del protocolo más conveniente, conforme a los objetivos perseguidos. Luego, se llegó a la conclusión de que la norma USB 2.0 es el más apropiado para la implementación de la comunicación entre un FPGA y la PC. Para establecer dicha comunicación, como se menciona en la Sección 1.2, se utiliza una interfaz que ejecute las tareas relativas al protocolo, la transmisión y recepción de los datos.

La implementación de la comunicación que se describe en el presente informe consta de tres etapas. La etapa principal esta compuesta por una interfaz que efectúa todas las tareas relativas a la comunicación USB 2.0 propiamente dicha. Luego, se utiliza una FPGA para corroborar que dicha comunicación se lleva a cabo conforme a los requerimientos. Finalmente, se realiza un programa de PC que envía y recibe datos, efectuando una prueba completa del sistema desarrollado.

En este Capítulo se detallan las características de cada una de las herramientas utilizadas durante el desarrollo de la comunicación y se brinda una breve justificación de la elección y la utilizaciójn de las mismas.

### 3.1. Elección de la FPGA

En la implementación de una comunicación, para poder transmitir y recibir datos, los componentes que intervienen deben seguir un protocolo establecido, de forma tal que cada dispositivo sepa que procedimiento efectuar. Por este motivo, una vez definido que se utiliza una interfaz intermedia entre la PC y un FPGA y que dicha interfaz es el circuito integrado EZ-USB FX2LP de Cypress, se deberá configurar un FPGA para que reciba y envíe datos a la interfaz.

Se elige para este trabajo, la placa de desarrollo MOJO v3 desarrollada por la empresa Embedded Micro. Esta placa, la cual se observa en la Figura 3.1, posee un FPGA Spartan-6 de Xilinx. El FPGA brinda la posibilidad de elaborar sistemas digitales complejos de alta velocidad y permite, al desarrollador de sensores y sistemas de adquisición de datos, la síntesis de circuitos que resuelvan problemas en la medida de los requerimientos. Dispone también de 84 puertos digitales configurables como entrada y/o salida, 8 entradas analógicas, 8 LED's de propósito general, un botón de tipo pulsador.

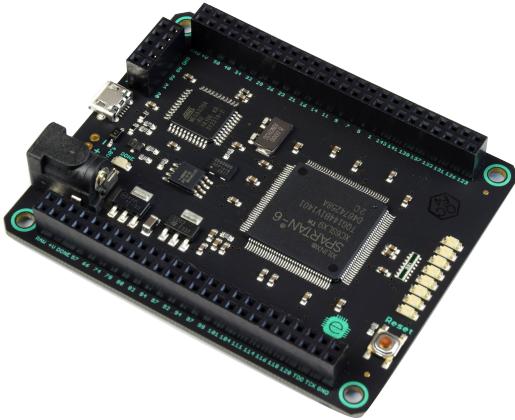


Figura 3.1: Placa de prototipado rápido MOJO v3, diseñada por Embedded Micro

La placa MOJO es una placa de desarrollo muy económica para prototipado, es decir, la fabricación de modelos funcionales. Para ello los puertos se disponen en un arreglo de pines a través de los cuales es posible acoplar el dispositivo que sea necesario. Se dispone en el mercado de otros circuitos impresos que se conectan a los pines y contienen un grupo de periféricos para propósito general. Estos impresos, se denominan shields (*escudo* traducido al castellano). se obtiene así una placa de desarrollo a la medida de las necesidades de cada proyecto. El usuario también puede diseñar sus shields o conectar las entradas y salidas de otros dispositivo mediante cables.

Además de los shields, los diseñadores pensaron en que no sea necesario ninguna herramienta extra a la hora de programar la FPGA. Para ello, dotaron al sistema de un microcontrolador ATmega32U4 de Atmel con un programa de tipo bootloader, que se encarga de transferir la configuración del FPGA cargada desde una memoria flash incorporada, o transmitida por el usuario desde una PC, a través de un transceptor USB que contiene el microcontrolador. Luego, el controlador es colocado en modo esclavo y se configura de forma tal que dota al sistema de una comunicación entre la FPGA y una PC, vía USB y se utiliza su ADC para leer los puertos analógicos.

Una vez llegado a este punto, el lector podría preguntar con toda razón ¿por qué es necesario realizar un sistema de comunicación USB extra, si ya cuenta con un microcontrolador que se encarga de dicho asunto? La respuesta se basa en el ancho de banda del sistema de comunicación que dispone la placa. La línea de controladores ATmega incorpora puertos USB 2.0 full-speed. Esto quiere decir que puede enviar datos a una tasa de  $12 \text{ Mbit s}^{-1}$ . Además, la comunicación entre ambos chips se realiza via SPI (*Serial Peripheral Interface*, o en español Interfaz Serie de Periféricos), comandada por un cristal de cuarzo de 50 MHz, ofreciendo una velocidad de salida que puede resultar insuficiente a los fines de este trabajo. Se pretende dotar al sistema del mayor ancho de banda posible, utilizando la capacidad de USB 2.0 High-Speed, de hasta 480 Mbps.

La placa de desarrollo Mojo fue seleccionada debido a su bajo costo, la versatilidad que ofrece y que está dotada por un Spartan-VI de Xilinx, que es un FPGA con una buena relación entre recursos, rendimiento, velocidad y precio.

## **3.2. Elección de la biblioteca libusb-1.0**

La tercer parte en la que se divide el trabajo es relativa a la comunicación entre la interfaz y una PC. Ya que la interfaz se encarga en gran medida de lo relativo al empaquetamiento, codificación y decodificación y que las PC, por su parte, vienen equipadas con el hardware necesario, este trabajo debe implementar el software que comande y gestione, desde el sistema operativo el correcto acceso a los datos que se envían y reciben. Para la elaboración de software que permita el manejo de los puertos USB, se utiliza la biblioteca **libusb**.

**libusb** es una biblioteca de código abierto, muy bien documentada, escrita en C, que brinda acceso genérico a dispositivos USB. Las características de diseño que persigue el equipo de desarrollo que mantiene la biblioteca es que sea multiplataforma, modo usuario y agnóstico de versión[32].

- Multiplataforma: Se apunta a que cualquier software que contenga esta biblioteca pueda ser compilado y ejecutado en la mayor cantidad de plataformas posibles, dotando al software de portabilidad, es decir, esta biblioteca puede ser ejecutada en Windows, Linux, OS X, Android y otras plataformas sin necesidad de realizar cambios en el código.
- Modo usuario: No se requiere acceso privilegiado de ningún tipo para poder ejecutar programas escritos con esta biblioteca.
- Agnóstico de versión: Sin importar la versión de la norma USB que se utilice, el programa se podrá comunicar siempre con el dispositivo USB que se requiera.

La biblioteca **libusb** no posee un autor formal. Es decir, no hay una persona, empresa u organización formal que se encargue de la creación y el mantenimiento del software. Existe una comunidad de más de 130 desarrolladores que en forma voluntaria cooperan en el mantenimiento y desarrollo de esta biblioteca. Se garantiza así que el proyecto esté documentado en forma detallada, existiendo amplios ejemplos y tutoriales de su uso.

Se elige esta biblioteca para la realización del software que gestionara el envío y la recepción de datos debido a su amplio soporte, la factibilidad de ejecutarlo en diferentes sistemas operativos y por ser totalmente gratuito.

## **3.3. Sumario del capítulo**

En este capítulo se resumen las partes y que herramienta compone cada una de ellas. Tal como se observa en la Figura 3.2, el programa de PC que gestionará la comunicación desde ese extremo se realiza en lenguaje C, con la biblioteca **libusb**. Como interfaz, se utiliza el controlador EZ-USB FX2LP, que viene montado en la placa de desarrollo CY3684 EZ-USB FX2LP de Cypress Semiconductor. Finalmente, desde el extremo del FPGA, se utilizará un Spartan-6 de Xilinx, a través de la placa de desarrollo Mojo v3.

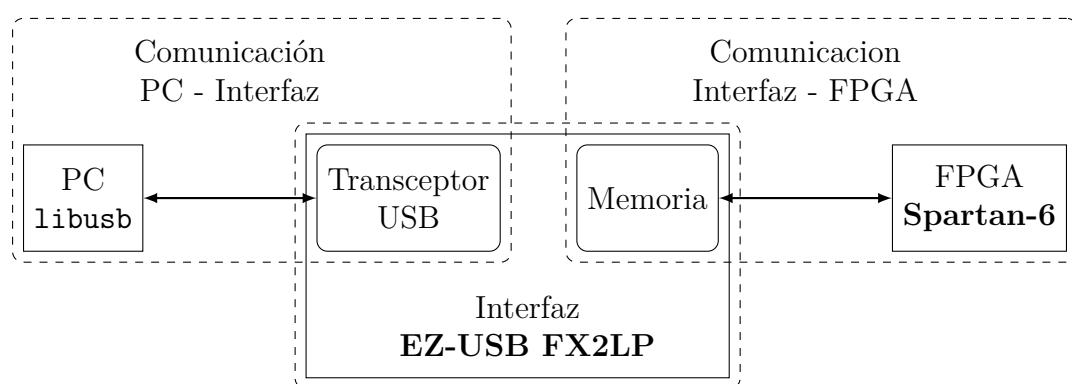


Figura 3.2: Partes en que se desglosa el trabajo con sus componentes

# Capítulo 4

## Interfaz USB

La interfaz que se escoge y utiliza en el desarrollo de la comunicación que se implementa, para que los datos fluyan entre un FPGA y una PC está compuesta por el controlador EZ-USB FX2LP de Cypress, el cual viene incorporado en el kit de desarrollo CY3684.

El kit de desarrollo CY3684 puede ser descompuesto en dos partes: una de hardware, que posibilita la conexión eléctrica entre los componentes y una parte de software que facilita al desarrollador tanto la elaboración del programa que es cargado y ejecutado por el microcontrolador, denominado firmware, como las pruebas del sistema en desarrollo.

En este capítulo se presenta el estudio y la selección de la configuración que mejor se adapta a los objetivos, se detalla el firmware elaborado y se abordan algunos aspectos conceptuales sobre la estructura y arquitectura del circuito integrado seleccionado como interfaz y las herramientas utilizadas.

### 4.1. Elección de la Interfaz

La comunicación entre la PC y el FPGA se realiza mediante tres bloques, los que se pueden apreciar en la Figura 4.1: la comunicación entre el FPGA y la interfaz, la configuración de la interfaz misma y la conexión entre la PC y la interfaz. El desarrollo de cada etapa cuenta con herramientas específicas que facilitan en gran medida la tarea que se realiza. En este capítulo se detalla por separado las características de cada una de estas herramientas.

La parte central del sistema desarrollado en el presente trabajo está constituida por el módulo

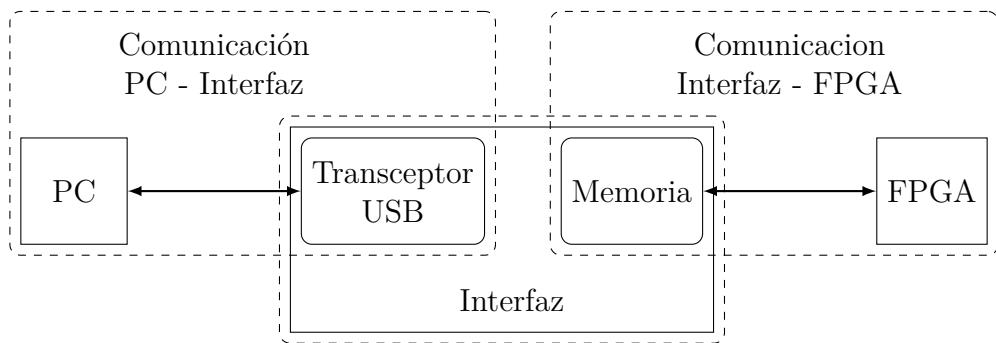


Figura 4.1: Partes en que se desglosa el trabajo



Figura 4.2: Circuito impreso principal del kit de desarrollo CY3684 EZ-USB FX2LP

de interfaz entre el FPGA y la PC, que es el que permite cumplir el objetivo de proveerle al sistema la comunicación, es decir, que cualquier dispositivo implementado con FPGA pueda recibir y transmitir datos a través del protocolo USB desde y hacia una PC, respectivamente.

La interfaz está constituida por el controlador EZ-USB FX2LP fabricado por Cypress Semiconductor, un circuito integrado que posee en su interior un microcontrolador 8051, con algunas mejoras destinadas a satisfacer mejor los requerimientos del sistema USB; una interfaz serie, que permite ingresar datos uno tras otro y los entrega en forma paralela y viceversa; un transceptor USB encargado de todas las tareas de codificación y decodificación de paquetes USB; memoria RAM para programas y datos de 16 kB. Posee, a su vez, tres tipos de interfaces hacia periféricos externos: I<sup>2</sup>C, una memoria FIFO (Primero Entrado, Primero Salido, del inglés *First In First Out*) destinada a sistemas con poder de iniciativa para escribir y leer datos, y un sistema de propósito general que puede ser comandado a través del 8051[30].

El controlador viene montado en un circuito impreso que posee una serie de componentes adicionales que facilitan la interacción del desarrollador, tales como pulsadores, display de 7 segmentos, módulos de memoria adicional, etc. Este tipo de circuitos impresos armados con la intención de favorecer desarrollo de otros sistemas, se denomina placa de desarrollo. Una placa de desarrollo que, además, incorpora algunas herramientas extra como software, cables de conexión, fuentes, etc. toma el nombre de kit de desarrollo.

En este trabajo, se utiliza el kit de desarrollo CY 3684 EZ-USB FX2LP, fabricado por Cypress Semiconductor[31]. El kit posee una placa de desarrollo como la que se observa en la Figura 4.2. El componente principal del kit es el controlador EZ-USB FX2LP e incorpora un display de 7 segmentos, 4 luces led multipropósito, 6 pulsadores, de los cuales 4 son de propósito

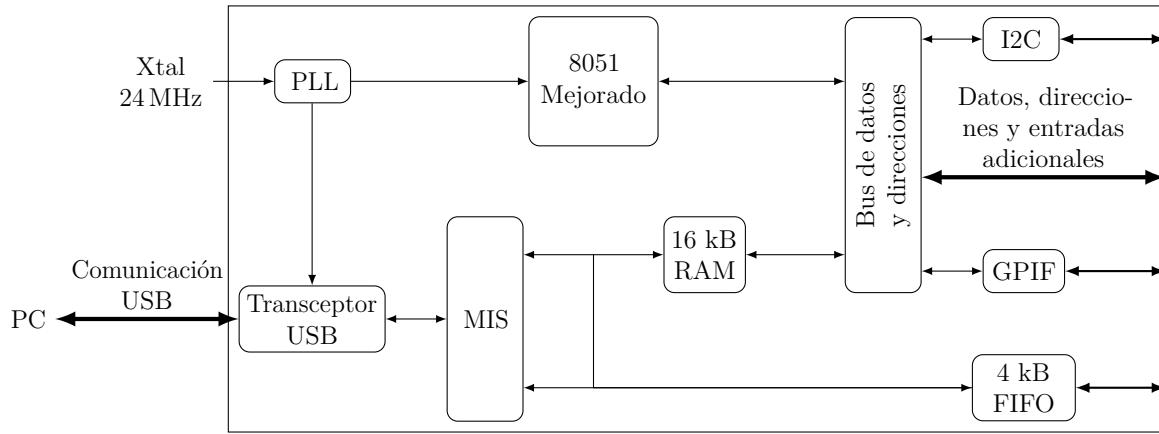


Figura 4.3: Arquitectura FX2LP

general, uno de reinicio y otro que envía una señal especial para salir de un modo de bajo consumo. También tiene dos bloques de memorias EEPROM destinadas al almacenamiento del firmware (programa que ejecuta un microcontrolador), lo que otorga la posibilidad de realizar una carga no volátil de la configuración del controlador, memoria flash de 64 kB utilizados como RAM por el programa del controlador, un puerto USB y dos puertos UART con zócalos DE-9. Adicionalmente, cuenta con 6 puertos de 20 pines que permiten comunicarse con el controlador y 1 puerto de 40 pines, compatible con el protocolo ATA.

Se selecciona este controlador como interfaz ya que cuenta con una gran cantidad de herramientas que permiten realizar la comunicación USB, además de poseer memoria suficiente para datos y una interfaz de comunicación para periféricos simple lo que facilita el objetivo de utilizar la menor cantidad de los recursos configurables del FPGA, de forma tal que queden, estos últimos, disponibles para el desarrollo de otros sistemas.

## 4.2. Arquitectura FX2LP EZ-USB

El núcleo del Kit de Desarrollo CY3684 es el controlador EZ-USB FX2LP. La serie de controladores FX2LP se caracteriza por brindar una conexión USB 2.0 de alta velocidad y bajo consumo energético. Está diseñada, preferentemente más no exclusivamente, para periféricos con autonomía limitada.

La arquitectura de controlador FX2LP, tal como se presenta en la Figura 4.3, integra un controlador USB completo. Incluye un transceptor USB, un Motor de Interfaz Serie (MIS), buffers de datos configurables, un microcontrolador ( $\mu$ C) 8051 que contiene registros y funciones adicionales orientadas a mejorar el rendimiento de la comunicación USB y una interfaz programable hacia los periféricos implementada con memoria tipo FIFO (*FirstInFirstOut*; Primero Entrado, Primero Salido). Además posee un PLL con divisor configurable a través del cual provee las señales de reloj adecuadas para el correcto funcionamiento del sistema.

El flujo de datos posee dos puntas (una PC y un FPGA) entre las cuales el controlador cumple el rol de interfaz. Para ello necesita poder comunicarse tanto con el host como con los periféricos. El usuario puede transmitir datos desde y hacia el host a través del mismo puerto USB. Sin embargo, también posee dos puertos DE-9 que permiten comunicarse con la PC a través del protocolo UART (acrónimo de Transmisión y Recepción Asíncrona Universal, en

inglés) que facilitan en gran medida la tarea de depuración del desarrollo gracias a que posee una configuración simple.

En cuanto a la interfaz con uno o mas perifericos, el controlador posee un puerto  $I^2C$ , una interfaz de propósito general (GPIF), para sistemas que necesitan ser comandados en forma externa; y una interfaz con memorias FIFO esclavas, a través de las cuales se puede conectar sistemas que cumplen un rol activo en el envío y recepción de información. Estas tres interfaces posibilitan la conexión de dispositivos que poseen tanto puertos estandarizados (ATA, PCMCIA, EPP, etc.), como personalizables (DSP, FPGA, microcontroladores, etc).

Este trabajo utiliza particularmente las memorias FIFO en modo esclavo, que responden a las diferentes señales que les proporciona un maestro externo implementado con un FPGA; por lo que a continuación se explicitan algunos detalles referidos a ellos, con lo que se busca aclarar el funcionamiento y que el lector comprenda los fundamentos de las configuraciones que se plasman en el código del firmware.

#### 4.2.1. Motor de Interfaz Serial

El Motor de Interfaz Serial (MIS) es un módulo incorporado al circuito integrado que se encarga de tomar datos en paralelo y convertirlos en una secuencia seriada.

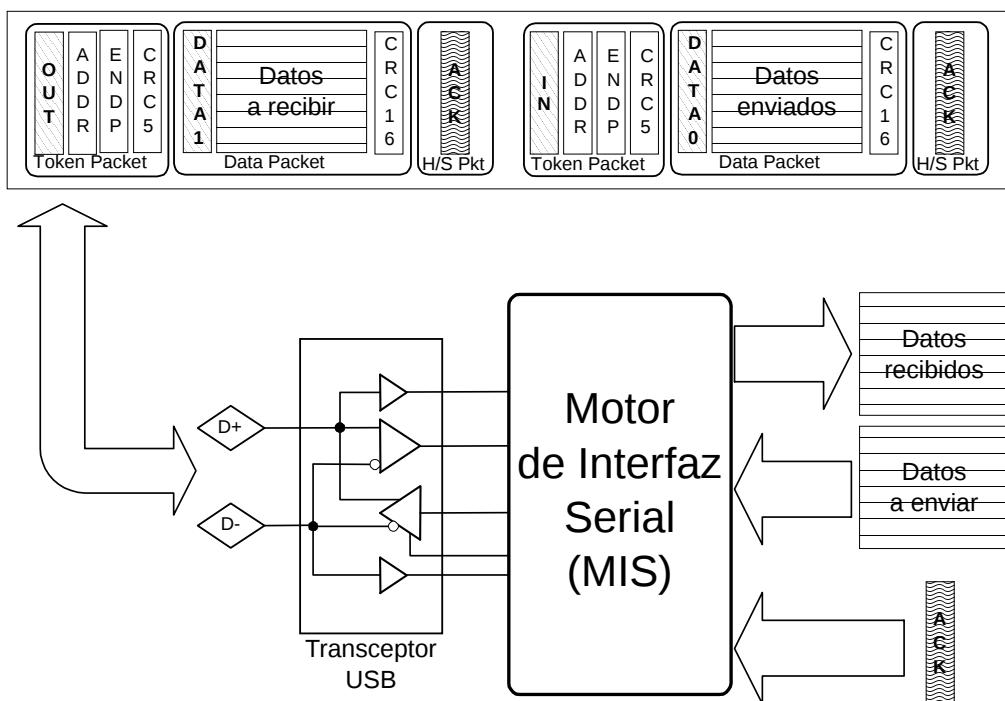


Figura 4.4: Implementación del enlace USB realizado por el EZ-USB[30]

La comunicación USB entre el controlador FX2LP y la PC se realiza a través del transceptor, unido al MIS. Para realizar el intercambio de datos, el firmware solo debe colocar o extraer los datos de buffers programables y modificar las banderas de handshaking. En forma automática, el MIS se encargan de empaquetar, enviar, recibir y desempaquetar toda la información, así como leer los tokens que emite el host, calcular y corroborar los códigos cílicos de detección de

errores y todo lo relacionado al protocolo propiamente dicho. El transceptor codifica y decodifica todo a nivel físico.

La Figura 4.4 muestra la función del MIS. Toma los datos colocados en los buffers de extremos, agrega la información que corresponde al encabezado y a la cola y, finalmente, coloca el registro de handshaking. Esto último, se observan como ACK (abreviación del inglés *acknowledge*, que significa reconocer, aceptar o agradecer) en la Figura 4.4. En el extremo del controlador, estas banderas se colocan en un registro especial que indica si el sistema está disponible, si los datos fueron colocados o leídos, dependiendo el caso tratado.

#### 4.2.2. Buffers de extremos

El MIS guarda los datos que aún no han sido enviados y/o los que han sido recibidos pero no leídos por ningún periférico en una memoria RAM específica, denominada buffer de extremo.

La norma USB define a un dispositivo extremo como una porción exclusiva e identifiable de una dispositivo USB que es fuente o un sumidero de información. En otras palabras, USB ve a cada extremo como una memoria FIFO de donde surge o finaliza la información. En inglés, el término extremo recibe el nombre de *endpoint*, por lo que, en adelante, cuando se hable de ellos se abreviará como EP o EPx, siendo la x un número que indica la dirección del extremo.

La serie de controladores FX2LP dispone de hasta 7 EP programables, los cuales deben poseer al menos dos buffers. La norma USB indica que cualquier dispositivo USB debe poseer un EP con dirección 0 que se destina para control y configuración, por lo que el controlador está dotado de 64 B para este fin. Es el único EP que puede ser bidireccional en el sentido del flujo de datos. A través de él, host y dispositivo envían y reciben transferencias de control. Luego, se incorporan dos EP1, que poseen un buffer de 64 B cada uno. Estos EP se identifican por la dirección de los datos, ya que uno de ellos es de salida y el otro de entrada de datos.

Finalmente, se incorpora una memoria de 4 KiB que debe ser configurada para los EP2, EP4, EP6 y EP8. La configuración de los EP la realiza el microcontrolador una vez que su programa se encuentra en ejecución. Las variables, conforme a los requerimientos de ancho de banda y acceso al bus son:

- Tamaño: Dependiendo del extremo a configurar puede ser de 512 o 1024 bytes.
- Tipo de acceso al bus: Definido según la norma USB, este tipo puede ser por bultos, isocrónico o de interrupción. No se admiten en estos EP paquetes de control.
- Cantidad de buffers: Dependiendo del extremo, puede ser dos, tres o cuatro buffers por extremo.
- Habilitación: Se debe indicar al sistema si los extremos se usan o no. El EP no válido, no responderá a un pedido de entrada o salida.

La Figura 4.5 muestra solo dos de las posibles configuraciones de los EP. A la izquierda se observa la configuración por defecto del controlador FX2LP. Esto es, los cuatro EP habilitados, con 512 bytes cada uno, buffers dobles y comunicación por bultos. A la derecha se muestra la configuración elegida para este trabajo, es decir, solo son EP válidos el EP2 y EP8. EP2 posee tres buffers de 1024 bytes y el EP8 dos buffers con 512 bytes de capacidad cada uno. Siempre se debe considerar que se dispone hasta 4 KiB de memoria.

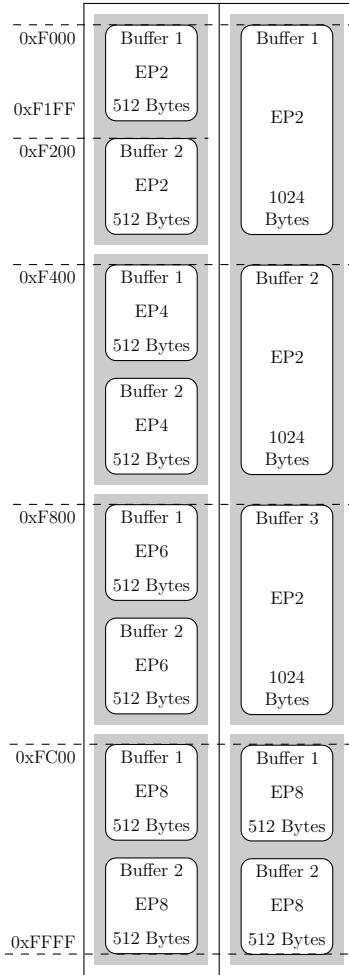


Figura 4.5: Buffers de extremos con sus direcciones de memoria. El cuadro de la izquierda muestra la configuración por defecto. El derecho, la implementada en este trabajo.

La característica de los buffers múltiples evita la congestión de datos. Con doble buffer, un periférico (o el microcontrolador) coloca o extrae datos de un buffer, mientras otro, del mismo EP, se encuentra enviando o recibiendo datos mediante el MIS. Cuando se configura un triple o cuádruple buffer, se agrega una o dos porciones mas de memoria a la reserva, respectivamente. De esta forma, se le otorga al sistema una gran capacidad de datos y ancho de banda.

Un detalle importante de los buffers múltiples es que, a la vista del controlador y/o de un periférico, el buffer posee una sola y única dirección y, es la propia interfaz FX2LP quien se encarga de seleccionar el buffer que corresponde en cada caso. Esto quiere decir que, por ejemplo, teniendo 4 buffers de 512 B cada uno, el 8051 verá solo uno de 512 B, sin necesidad de identificar a través de su firmware con cuál de los cuatro está trabajando.

#### 4.2.3. Memorias FIFO esclavas

Desde el punto de vista de la electrónica digital, el MIS es un dispositivo que recibe y envía datos desde y hacia el puerto USB utilizando una señal de reloj de 24 MHz. Esta señal, es provista por un cristal de cuarzo incorporado en el circuito impreso del Kit de Desarrollo

CY3684 EZ-USB FX2LP. Por su parte, un sistema externo puede o no proveer una señal de reloj y manejo de datos propio cuya fuente de reloj es a priori desconocida por quien configura el circuito integrado. El controlador USB incorpora memorias FIFO que se encargan de proveer una interfaz entre el MIS y un dispositivo externo, salvando el problema de poseer dos relojes diferentes e independientes.

Estas memorias funcionan en modo esclavo, es decir, se debe conectar un dispositivo capaz de proveer una lógica maestra externa que comande la entrada y salida de datos desde una memoria FIFO hacia o desde el exterior. Para los fines del presente trabajo, este modo de funcionamiento es óptimo ya que, dotando al FPGA de una máquina de estados, se logra la transferencia de datos en los tiempos requeridos.

El sistema de bus permite conectar a estas memorias hasta cuatro dispositivos diferentes. Por esto, existe un registro que permite seleccionar una porción de memoria FIFO para cada uno de los EP programables en el buffer de extremos.

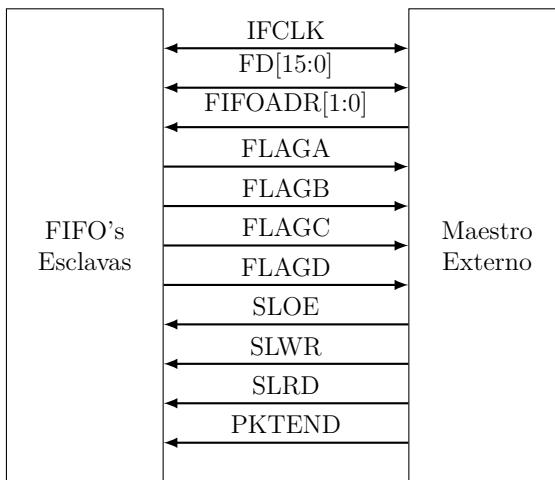


Figura 4.6: Puertos de interfaz entre las FIFO's y un maestro externo

La Figura 4.6 muestra las señales de la interfaz entre las memorias FIFO's y un maestro esclavo. Estas son:

- IFCLK: señal de reloj. No es necesario en caso de conectar la interfaz en modo asincrónico. La señal de reloj puede ser provista por el controlador o por el dispositivo de control en forma programable.
- FD[15:0]: constituye el bus de datos. Según se programe, este puede ser de 8 o 16 bits, en forma independiente para cada EP.
- FIFOADDR[1:0]: puerto de direcciones. A través de él se selecciona la memoria activa en el bus.
- FLAGx: Los cuatro puertos de flag son configurables e indican memoria llena, vacía o un nivel programable. También pueden indicar el estado de una memoria específica o de la que se encuentra activa a través de FIFOADDR.
- SLOE, SLWR, SLRD: son las señales de control. A través de ellas el maestro entrega las ordenes de lectura y escritura.

- PKTEND: a través de este puerto el maestro indica que terminó una transferencia de datos.

#### 4.2.4. Modos de entrada y salida automáticos

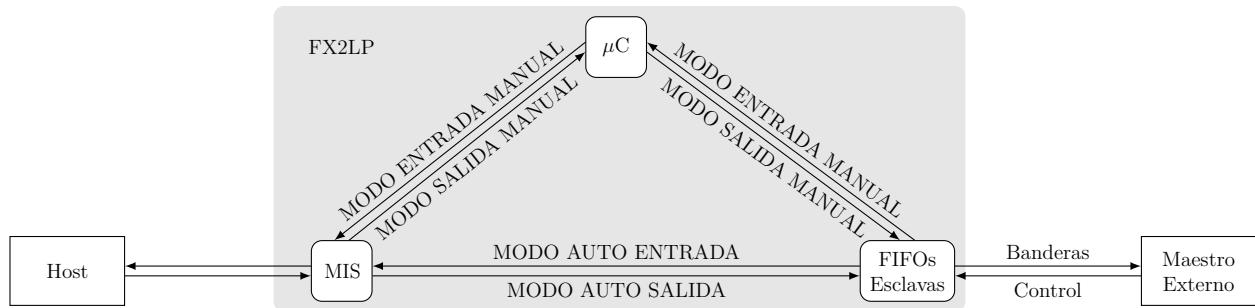


Figura 4.7: Modos de conexión de la memoria FIFO, el micrcontrolador y el MIS

Los datos que se reciben o envían a través del MIS. Pueden ser enviados en forma automática desde y hacia las memorias FIFO, o bien, pueden ser dirigidos a través del microcontrolador. Esto último permite leer, modificar, suprimir, agregar y/o generar nuevos datos antes de ser remitidos como paquete, es decir, todos juntos, a su respectivo EP. Estos caminos se pueden ver en la Figura 4.7.

Los fabricantes llaman a estos caminos "MODO MANUAL", en caso de enviar los datos a través del 8051, y "MODO AUTOMÁTICO", cuando la comunicación es directa entre el MIS y las FIFO. Además, se programan en forma independiente para cada extremo, sea este de salida o entrada.

Se debe notar en la Figura 4.7 que se refiere a paquetes de entrada cuando estos poseen una dirección que se inicia en un periférico y termina en el host y de salida cuando llevan el sentido contrario. Esto se debe al rol central que ejerce el host en la comunicación USB.

### 4.3. Kit de desarrollo de Software de Cypress

A través del kit de desarrollo CY3684 EZ-USB FX2LP, Cypress provee un amplio conjunto de herramientas que facilitan en gran medida la implementación del software. Estas soluciones se denominan Kit de Desarrollo de Software (SDK, acrónimo del inglés, *Software Development Kit*). Este SDK abarca una gran cantidad de aspectos, como ser la elaboración del firmware implementado en el 8051 del controlador, el desarrollo de programas de control para PC, la conversión de archivos de programación y ejecutables que graban la información en las diferentes memorias que posee la placa de desarrollo, sea RAM o EEPROM, para cargar programas no volátiles que posteriormente serán ejecutados por el 8051. No todas las herramientas se utilizan en este trabajo, por lo que se desarrollan las más destacadas.

#### 4.3.1. Framework Cypress

Con la intención de dotar al desarrollador con una herramienta que le potencie la velocidad de diseño, Cypress provee una plantilla de código en lenguaje C para microcontroladores 8051.

Esta plantilla posee precargados todos los registros que posee la serie de controladores FX2LP con los mismos nombres que figuran en el manual de usuario. Además incorpora las funciones que el microcontrolador debe llevar a cabo para efectuar la comunicación USB y algunas que permiten interactuar con la placa de desarrollo CY3684. Los archivos que incorpora son:

- fw.c: es el código fuente principal. Contiene la función main() y lo necesario para manejar la comunicación USB.
- periph.c: contiene la implementación de las funciones invocadas por fw.c. Aquí se encuentran TD\_Init() y TD\_Poll(), las funciones a través de las cuales el usuario implementa el programa que necesita. También contiene todas las funciones de interrupción vectorizadas.
- fx2.h: posee la definición de constantes, macros, tipos de datos y funciones prototipo de la biblioteca.
- fx2regs.h: declara registros y máscaras.
- dscr.a51: este archivo es el descriptor de dispositivo. Es la información que USB necesita para poder registrar el dispositivo en el host, asignarle dirección y establecer los parámetros.
- ezusb.lib: biblioteca que implementa funciones provistas por el fabricante.
- syncdely.h: macro de sincronismo. Algunos accesos de registro requieren un tiempo de establecimiento específico que en el código se implementa deteniendo el microcontrolador ciertos ciclos de reloj.
- usbjmpbt.obj: especifica las direcciones en memoria de las interrupciones vectorizadas.

La Figura 4.8 muestra una versión modificada del diagrama de flujo que sigue el código fuente provisto por Cypress. De ella se quitan funciones que no son necesarias para los objetivos del presente trabajo.

Cuando el programa es cargado al controlador, este se encarga de inicializar todas la variables de estado a su valor por defecto. También en este punto establece la comunicación con el anfitrión y le envía los descriptores provistos en el archivo `dscr.a51`. Acto seguido, ejecuta la función `TD_Init()`, a través de la cual el usuario programa e inicia la configuración del sistema. Luego, es necesario habilitar las interrupciones necesarias y finalmente, se invoca repetidamente la función `TD_Poll()`, en donde el usuario escribe las tareas que ejecutará el 8051.

#### 4.3.2. Entorno de desarrollo y compilador

Si bien Cypress no desarrolla software para escribir, compilar y depurar códigos, distribuye junto con el kit CY3684 una versión para evaluación de Keil  $\mu$ Vision con el compilador C51 para programar microcontroladores basados en 8051. Aún la versión limitada de este entorno, resulta suficiente para la programación del controlador USB.

Keil  $\mu$ Vision es un entorno de desarrollo integrado (IDE). Se entiende por IDE a un software que integra en un entorno gráfico las herramientas que permiten elaborar un programa que ejecutará un procesador, desde la escritura del algoritmo en uno o más lenguajes, su compilación, las pruebas y el depurado.

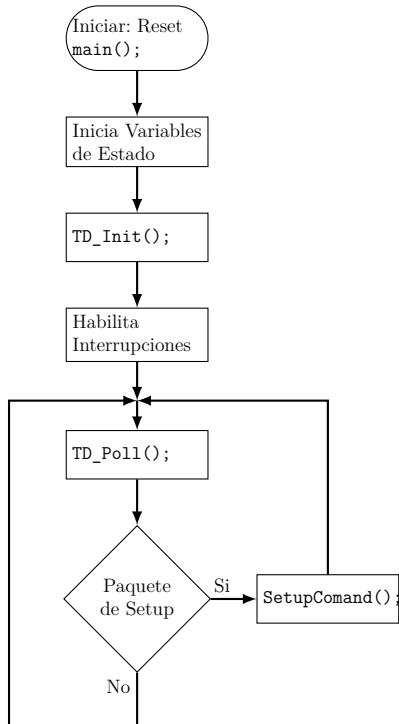


Figura 4.8: Diagrama en bloques simplificado

El programa utilizado posee, entre otras cosas, editor de textos con atajos de teclado, comandos que aceleran la escritura de código y resaltado de palabras claves para diferentes lenguajes de programación, navegador de archivos. También ejecuta, con solo un click, el compilador con la sintaxis correcta, y posee un depurador que, a través de un intérprete, permite ir ejecutando el código línea por línea o en bloques.

Para realizar un programa en este entorno, Cypress provee, junto con su framework, un proyecto vacío que puede ser copiado y pegado. Sin embargo, se puede realizar la configuración manual. Las instrucciones de este procedimiento se ubican en el Apéndice ??.

En cuanto al compilador se refiere, el utilizado es C51. Éste es un programa que otorga un archivo hexadecimal con un código que será ejecutado por microcontroladores que estén implementados con la misma estructura que un Intel 8051, como lo es el microcontrolador que posee el FX2LP.

### 4.3.3. Cypress USB Control Center

Para grabar los programas desarrollados en el microcontrolador, el SDK provee de la aplicación USB Control Center. Además de la capacidad para programar el microcontrolador, este programa posee posibilita el envío y la recepción de datos a través del puerto USB conectado con el dispositivo programado, y muestra la configuración que el mismo envió en forma de descriptores. Esta característica brinda una realimentación, aunque mínima, de la entrada y salida de datos, facilitando las pruebas sobre el sistema en desarrollo. En el Apéndice ?? se explica su funcionamiento y manejo.

## 4.4. Desarrollo del firmware

A continuación se desarrollan los aspectos más relevantes del código final elaborado, compuestos por la función de inicialización y los descriptores del dispositivo USB.

Al establecer el modo automático de funcionamiento, la función `TD_Poll()` se encuentra vacía.

Luego, en el Capítulo 5 se abordará nuevamente algunos detalles realizados para la depuración del presente código, tales como la conexión del puerto serie y la utilización de algunas interrupciones específicas.

### 4.4.1. Inicialización del dispositivo

La inicialización del dispositivo se realiza a través de la función `TD_Init()`. Ésta es invocada solo una vez en el código, antes de ejecutar el loop principal, donde el programa ejecuta tareas específicas una y otra vez.

En primer lugar, se debe configurar la frecuencia a la que corre el reloj principal. Esta puede ser de 12 MHz, 24 MHz o de 48 MHz. Para ello se deben colocar los registros CPUCS.4=1 y CPUCS.3=0. A través del framework de Cypress, esto puede ser escrito de la siguiente forma:

```
CPUCS = ((CPUCS & ~bmCLKSPD) | bmCLKSPD1); // 48 MHz
```

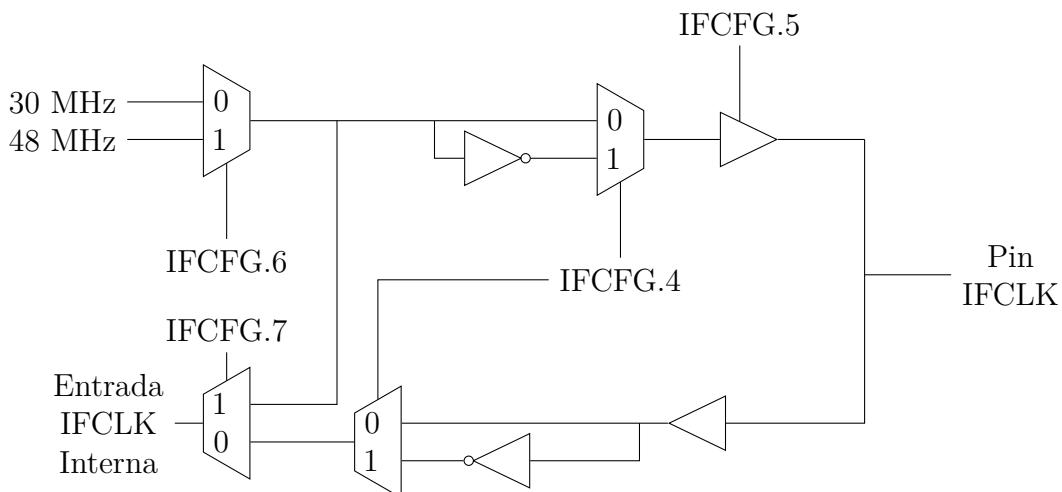


Figura 4.9: Esquema funcional para la entrada de reloj de la Interfaz ;

Luego se debe configurar el funcionamiento del reloj de la interfaz. El esquema de la Figura 4.9 muestra la configuración del hardware. Cómo se observa, La interfaz puede funcionar con frecuencias de 48 MHz o 30 MHz provistas por el FX2LP. Además, la señal puede ser dirigida al exterior, o bien, ser provista por un periférico. Los inversores se programan de forma tal que la interfaz sea activa en el flanco positivo o negativo del reloj fuente. En este trabajo, el registro se programa para tomar como reloj la señal de 48 MHz provisto por el mismo chip, la polaridad es de flanco ascendente y no se posee señal de reloj en el pin externo IFCKL. Además, se programa de modo asíncrono y en modo FIFO esclavo, a cuya configuración se accede por este puerto.

El autor entiende sobre la posibilidad y conveniencia de utilizar el modo sincrónico para conectar la interfaz. Más aún, es factible proveer la señal necesaria de reloj desde la FPGA y evitar así problemas de desajustes y fallas de sincronismo. Sin embargo, por error del alumno en el diseño del impreso de interconexión, la entrada de reloj no quedó conectada al chip de la FPGA. Durante la escritura de este informe, se encuentra en viaje el impreso con las correcciones pertinentes y espera ser implementado en trabajos futuros.

La configuración, entonces, queda definida por la sentencia de código:

```
// colocar Interfaz FIFO esclava a 48MHz
// clk interno, no_salida, no_invertir clk, no_asincr
// fifo esclava (11) = 0xC3
// 0xCB; para asincr.
IFCONFIG = 0xCB;
SYNCDELAY;
```

A continuación, se debe establecer el funcionamiento de los pines bandera. Estos avisan cuando un EP está vacío, completo o a un nivel programable. Para este trabajo, son necesarios solo una bandera que señale el nivel vacío del puerto por el que entran los datos a la FPGA y otra que señale el nivel completo del EP donde escribe los datos a enviar. Si bien no son leídos en ningún momento, para evitar problemas se configuran las banderas vacías y completas para ambos EP's:

```
//Pin Flags Configuración
PINFLAGSAB = 0xBC; // FLAGA <- EP2 Full Flag
                    // FLAGD <- EP2 Empty Flag
SYNCDELAY;
PINFLAGSCD = 0x8F; // FLAGC <- EP8 Full Flag
                    // FLAGB <- EP8 Empty Flag
```

Luego, se deben programar los EP's. La configuración por defecto define a todos los EP como transferencias por bultos con doble buffer de 512 B. El EP2 y EP4 son salidas (desde la PC). El EP6 y EP8 son entradas (hacia la PC).

La programación de este trabajo es con una entrada isocrónica de dos buffers con 512 B de capacidad, cada uno, definida en el EP2 y una salida por bultos de dos buffers de 512 B configurada en el EP8. Los otros EP's son deshabilitados. Sin embargo, EP1IN y EP1OUT se dejan configurados por defecto, ya que no interfieren en nada con el presente trabajo.

```
EP1OUTCFG = 0xA0;
SYNCDELAY;
EP1INCFG = 0xA0;
SYNCDELAY;
EP4CFG &= 0x7F;
SYNCDELAY;
EP6CFG &= 0x7F;
SYNCDELAY;
EP8CFG = 0xA0; //EP8 is DIR=OUT, TYPE=BULK, SIZE=512, BUF=2x
SYNCDELAY;
EP2CFG = 0xD2; // EP2 is DIR=IN, TYPE=ISOC, SIZE=512, BUF=2x
SYNCDELAY;
```

Como siguiente paso, se limpian las memorias FIFO de cualquier dato espúreo que contengan al momento del inicio del programa.

```
FIFORESET = 0x80 ;  
SYNCDELAY;  
FIFORESET = 0x02 ;  
SYNCDELAY;  
FIFORESET = 0x04 ;  
SYNCDELAY;  
FIFORESET = 0x06 ;  
SYNCDELAY;  
FIFORESET = 0x08 ;  
SYNCDELAY;  
FIFORESET = 0x00 ;  
SYNCDELAY;
```

De esta forma, el controlador se encuentra listo para configurar las memorias asignadas a cada uno de los EP's. Se debe notar que en primer lugar se coloca 0x00 y luego el valor estipulado. Esto se basa en que el modo automático se prepara para transmitir ante un flanco ascendente del registro que lo habilita.

```
EP8FIFOFCFG = 0x00 ;  
SYNCDELAY;  
EP2FIFOFCFG = 0x00 ;  
SYNCDELAY;  
  
//setting on auto mode. rising edge is necessary  
EP8FIFOFCFG = 0x11 ;  
SYNCDELAY;  
EP2FIFOFCFG = 0x0D ;  
SYNCDELAY;
```

Finalmente, se habilitan las interrupciones necesarias.

```
USBIE |= bmSOF;
```

Así, queda completa la inicialización y el dispositivo listo para enviar y recibir datos de forma automática.

#### 4.4.2. Encabezado y declaraciones importantes

Para el correcto funcionamiento de este código, es necesario incorporar el encabezado que se observa a continuación.

```
#pragma noiv // No generar vectores de interrupción  
#include "fx2.h"  
#include "fx2regs.h"  
#include "syncdly.h" // SYNCDELAY macro  
#include "leds.h"
```

Las primeras 4 líneas de encabezados son provistas por Cypress, a través de su framework. En ellas, la directiva de ensamblador `#pragma`, le indica al compilador que no debe habilitar las interrupciones vectorizadas. Estas, en cambio, serán manejadas y direccionadas a través del archivo objeto `usbjmpbt.obj`.

El encabezado `leds.h` cuyo código se muestra a continuación, sirve para encender y apagar las luces de la placa de desarrollo. Estos leds se encuentran conectados a través de un decodificador y su funcionamiento se da con la sola lectura de direcciones específicas.

```
xdata volatile const BYTE D2ON _at_ 0x8800 ;
xdata volatile const BYTE D2OFF _at_ 0x8000 ;
xdata volatile const BYTE D3ON _at_ 0x9800 ;
xdata volatile const BYTE D3OFF _at_ 0x9000 ;
xdata volatile const BYTE D4ON _at_ 0xA800 ;
xdata volatile const BYTE D4OFF _at_ 0xA000 ;
xdata volatile const BYTE D5ON _at_ 0xB800 ;
xdata volatile const BYTE D5OFF _at_ 0xB000 ;
```

Luego, el framework define algunas variables globales que utiliza en las funciones implementadas para el manejo de las tareas relacionadas al protocolo USB. Se listan estas variables a continuación.

```
extern BOOL GotSUD;           // Received setup data flag
extern BOOL Sleep;
extern BOOL Rwu;
extern BOOL Selfpwr;
```

```
BYTE Configuration;          // Current configuration
BYTE AlternateSetting = 0;    // Alternate settings
```

```
//
// Task Dispatcher hooks
// The following hooks are called by the task dispatcher.
//
```

```
WORD blinktime = 0;
BYTE inblink = 0x00;
BYTE outblink = 0x00;
WORD blinkmask = 0;           // HS/FS blink rate
```

#### 4.4.3. Descriptores USB

Los descriptores son una estructura definida de datos. A través de ellos, el dispositivo USB le comunica al anfitrión sus atributos, tales como velocidad de trabajo, cantidad de configuraciones e interfaces posibles, número de EPs, dirección de cada uno de ellos, tamaño máximo en bytes de paquetes que puede enviar en una comunicación, entre otros.

El framework de Cypress coloca toda la información sobre los descriptores de el dispositivo desarrollado en un archivo escrito en lenguaje de ensamblador, denominado `dscr.a51`. Este

archivo contiene una plantilla en donde el desarrollador coloca la descripción de la configuración realizada en el firmware. Luego, el archivo es enviado por el dispositivo al host comunicando las características del sistema desarrollado.

En primer lugar, posee un encabezado en donde se establecen etiquetas que hacen más legible el código para el programador:

```

1 DSCR_DEVICE    equ   1    ; ; Descriptor type: Device
2 DSCR_CONFIG    equ   2    ; ; Descriptor type: Configuration
3 DSCR_STRING    equ   3    ; ; Descriptor type: String
4 DSCR_INTRFC    equ   4    ; ; Descriptor type: Interface
5 DSCR_ENDPNT    equ   5    ; ; Descriptor type: Endpoint
6 DSCR_DEVQUAL   equ   6    ; ; Descriptor type: Device Qualifier
7
8 DSCR_DEVICE_LEN equ   18
9 DSCR_CONFIG_LEN equ   9
10 DSCR_INTRFC_LEN equ   9
11 DSCR_ENDPNT_LEN equ   7
12 DSCR_DEVQUAL_LEN equ   10
13
14 ET_CONTROL     equ   0    ; ; Endpoint type: Control
15 ET_ISO          equ   1    ; ; Endpoint type: Isochronous
16 ET_BULK         equ   2    ; ; Endpoint type: Bulk
17 ET_INT          equ   3    ; ; Endpoint type: Interrupt

```

Luego, el programador se debe asegurar que el código se guarda en un lugar de memoria adecuado.

```

1 DSCR  SEGMENT  CODE PAGE
2
3 ;;
4 ; ; Global Variables
5 ;;
6 rseg DSCR      ; ; locate the descriptor table in on-part memory.

```

Debido a la estructura rígida del formato de los descriptores, impuesto por la norma USB y por la implementación de Cypress, la memoria debe contener en primer lugar el descriptor de dispositivo.

```

1 DeviceDscr:
2   db  DSCR_DEVICE_LEN      ; ; Largo del descriptor
3   db  DSCR_DEVICE    ; ; Tipo de descriptor
4   dw  0002H      ; ; Versión de la norma (BCD)
5   db  00H       ; ; Clase de Dispositivo
6   db  00H       ; ; Sub-Clase de Dispositivo
7   db  00H       ; ; Sub-sub-Clase de dispositivo
8   db  64        ; ; Tamaño máximo de paquete
9   dw  0B404H     ; ; Identificador de Vendedor (Cypress)
10  dw  0310H     ; ; Identificador de Producto (Sample Device)
11  dw  0000H     ; ; Identificador de versión del producto

```

```
12      db 1          ; ; Indice de Fabricante en string  
13      db 2          ; ; Indice de Producto en string  
14      db 0          ; ; Indice de número de serie en string  
15      db 1          ; ; Número de configuraciones
```

El descriptor de tamaño máximo de paquete está referido especialmente al EP0, es decir, al extremo que el host y el dispositivo utilizan para intercambiar mensajes de control.

Se debe notar que los penúltimos tres parámetros mostrados corresponden a una descripción realizada en cadena de caracteres al final del archivo, a fin de poder mostrar un mensaje que pueda ser leído por un usuario humano.

El último parámetro está relacionado con el número de configuraciones que posee este dispositivo. Esto determina también la cantidad de descriptores de configuración que debe tener el archivo de descripción. No obstante, antes de comenzar con el descriptor de configuración, se debe especificar el descriptor Calificador de Dispositivo, el cual dà información sobre otras velocidades de operación. Este descriptor es necesario debido a que el sistema implementado cumple con la versión 2.0 de la norma USB.

```
1 org (( $ / 2 ) +1) * 2  
2 DeviceQualDscr :  
3     db DSCR_DEVQUALLEN    ; ; Largo del descriptor  
4     db DSCR_DEVQUAL      ; ; Tipo de descriptor  
5     dw 0002H            ; ; Versión de la norma (BCD)  
6     db 00H              ; ; Clase de dispositivo  
7     db 00H              ; ; Sub-clase de dispositivo  
8     db 00H              ; ; Sub-sub-clase de dispositivo  
9     db 64               ; ; Tamaño máximo de paquetes  
10    db 1                ; ; Número de comunicaciones  
11    db 0                ; ; Reservado
```

Debido a la implementación hecha por Cypress, cada descriptor debe encontrarse en una dirección de memoria par. Por ello se recurre al comando de la línea 1 del código de la sección anterior. A continuación, se especifica el descriptor de Calificador de Dispositivo, con información muy similar al descriptor de Dispositivo.

Luego, se procede a detallar cada una de las configuraciones y sus Interfaces que se indicaron en los descriptores anteriores. En este caso, se deben especificar dos: una de alta velocidad, indicada en el Descriptor de Dispositivo y otra de velocidad completa, indicada en el Calificador de Dispositivo. Se muestra en seguida el Descriptor de la configuración de alta velocidad, unido al Descriptor de Interfaz (línea 16 en adelante) y los Descriptores de Extremos a partir de la línea 27, que determinan la configuración total de Alta Velocidad.

```
1 org (( $ / 2 ) +1) * 2  
2 HighSpeedConfigDscr :  
3     db DSCR_CONFIG_LEN    ; ; Largo del descriptor  
4     db DSCR_CONFIG        ; ; Tipo de descriptor  
5     db (HighSpeedConfigDscr_End-HighSpeedConfigDscr) mod 256  
6                           ; ; Largo total (LSB)  
7     db (HighSpeedConfigDscr_End-HighSpeedConfigDscr) / 256  
8                           ; ; Largo total (MSB)
```

```

9      db 1      ; ; Número de interfaces
10     db 1      ; ; Índice de configuración
11     db 0      ; ; String de configuración
12     db 80H    ; ; Atributos (b7->1, b6 - selfpwr , b5 - rwu)
13     db 50     ; ; Consumo de potencia (div 2 ma)
14
15     ; ; Alt Interface 0 Descriptor – Bulk IN
16     db DSCR_INTRFC_LEN   ; ; Largo del descriptor
17     db DSCR_INTRFC      ; ; Tipo de descriptor
18     db 0                 ; ; Índice de interfaz
19     db 0                 ; ; Índice de ajuste alternativo
20     db 2                 ; ; Número de extremos
21     db 0ffH              ; ; Clase de interfaz
22     db 00H              ; ; Sub-clase de interfaz
23     db 00H              ; ; Sub-sub-clase de interfaz
24     db 0                 ; ; Índice de string descriptor de interfaz
25
26     ; ; Iso IN Endpoint Descriptor
27     db DSCR_ENDPNT_LEN  ; ; Largo del descriptor
28     db DSCR_ENDPNT      ; ; Tipo de descriptor
29     db 82H              ; ; Extremo de entrada EP2
30               ; ; b7 -> IN/OUT, b[4:0] -> dir
31     db ET_ISO            ; ; Tipo de transferencia
32     db 00H              ; ; Tamaño máximo de paquete (LSB)
33     db 02H              ; ; Tamaño máximo de paquete (MSB)
34     db 01H              ; ; Intervalo de consulta
35
36     ; ; Bulk OUT Endpoint Descriptor
37     db DSCR_ENDPNT_LEN  ; ; Largo del descriptor
38     db DSCR_ENDPNT      ; ; Tipo de descriptor
39     db 08H              ; ; Extremo de salida EP8
40     db ET_BULK           ; ; Tipo de transferencia
41     db 00H              ; ; Tamaño máximo de paquete (LSB)
42     db 02H              ; ; Tamaño máximo de paquete (MSB)
43     db 00H              ; ; Intervalo de consulta
44
45 HighSpeedConfigDscr_End :

```

En la línea 12 del código anterior se debe colocar cuál es la fuente de la potencia que consume el dispositivo, es decir, de donde proviene la energía utilizada para el funcionamiento. El bit7 debe estar siempre establecido a 1 por razones históricas de la norma USB[27]. El bit6 en 1 define que el dispositivo está energizado por una fuente propia. En el caso contrario, toma potencia del bus. El bit5, por su parte, señala que el dispositivo tiene modo de baja energía y que es posible establecer el modo de funcionamiento de mayor consumo con un comando del host. La línea 13 se informa cuánta potencia consume, lo que le brinda al host la posibilidad de establecer un control de la potencia suministrada en el bus.

El último campo del descriptor de extremo, correspondiente al intervalo de consulta, se utiliza para establecer cada cuanto tiempo el host debe asignar ancho de banda para transferencias isocrónicas.

Luego de enviar la configuración de Alta Velocidad, se informa de la misma manera el descriptor de Velocidad Completa, cuyo código se observa a continuación.

```
1 org (( $ / 2 ) +1) * 2
2 FullSpeedConfigDscr:
3     db DSCR_CONFIG_LEN      ; ; Largo del descriptor
4     db DSCR_CONFIG          ; ; Tipo de descriptor
5     db (FullSpeedConfigDscr_End - FullSpeedConfigDscr) mod 256
6                     ; ; Largo total (LSB)
7     db (FullSpeedConfigDscr_End - FullSpeedConfigDscr) / 256
8                     ; ; Largo total (MSB)
9     db 1      ; ; Número de interface
10    db 1      ; ; Numero de configuraciones
11    db 0      ; ; Indice de string de configuración
12    db 80H    ; ; Atributos (b7 -<'1', b6 - selfpw, b5 - rwu)
13    db 50     ; ; Requerimiento de potencia (div 2 ma)
14
15    ; ; Interface Descriptor
16    db DSCR_INTRFC_LEN ; ; Largo del descriptor
17    db DSCR_INTRFC      ; ; tipo de descriptor
18    db 0                  ; ; Indice de interfaz
19    db 0                  ; ; Indice de ajuste alternativo
20    db 2                  ; ; Número de extremos
21    db 0ffH              ; ; Clase de interfaz
22    db 00H              ; ; Sub-clase de interfaz
23    db 00H              ; ; Sub-sub-clase de interfaz
24    db 0                  ; ; Indice de string de interfaz
25
26    ; ; Endpoint Descriptor
27    db DSCR_ENDPNT_LEN   ; ; Largo del descriptor
28    db DSCR_ENDPNT        ; ; Tipo de descriptor
29    db 82H                ; ; Dirección y sentido de extremo
30    db ET_ISO              ; ; Tipo de extremo
31    db 0FFH              ; ; Tamaño máximo de paquete (LSB)
32    db 03H              ; ; Tamaño máximo de paquete (MSB)
33    db 01H              ; ; Intervalo de consulta
34
35    ; ; Endpoint Descriptor
36    db DSCR_ENDPNT_LEN   ; ; Largo del descriptor
37    db DSCR_ENDPNT        ; ; tipo de descriptor
38    db 08H                ; ; Dirección y sentido de extremo
39    db ET_BULK              ; ; Tipo de extremo
40    db 040H              ; ; Tamaño máximo de paquete (LSB)
```

```
41      db  00H          ; ; Tamaño máximo de paquete (MSB)
42      db  01H          ; ; Intervalo de consulta
43
44 FullSpeedConfigDscr_End :
```

Finalmente, el diseñador puede escribir todos los mensajes en formato de cadena de caracteres, para una lectura más sencilla por parte del usuario. En este trabajo solo se usan dos que sirven como ejemplo pero no se profundizó más en el estudio de estos mensajes debido a que no son relevantes para los objetivos.

```
1 org (( $ / 2 ) +1) * 2
2 StringDscr :
3
4 StringDscr0 :
5     db  StringDscr0_End - StringDscr0      ; ; Largo del descriptor
6     db  DSCR_STRING                      ; ; Tipo de descriptor
7     db  09H,04H
8 StringDscr0_End :
9
10 StringDscr1 :
11    db  StringDscr1_End - StringDscr1      ; ; Largo del descriptor
12    db  DSCR_STRING                      ; ; Tipo de descriptor
13    db  'E',00                            ; ; Mensaje
14    db  'd',00
15    db  'w',00
16    db  'i',00
17    db  'n',00
18    db  ' ',00
19    db  'B',00
20    db  'a',00
21    db  'r',00
22    db  'r',00
23    db  'a',00
24    db  'g',00
25    db  'a',00
26    db  'n',00
27 StringDscr1_End :
28
29 StringDscr2 :
30    db  StringDscr2_End - StringDscr2      ; ; Largo del descriptor
31    db  DSCR_STRING                      ; ; Tipo de descriptor
32    db  'L',00                            ; ; Mensaje
33    db  'a',00
34    db  ' ',00
35    db  'T',00
36    db  'e',00
37    db  's',00
```

```
38     db    'i ',00
39     db    's ',00
40 StringDscr2_End :
```

## 4.5. Sumario del capítulo

En el presente capítulo se explicaron las herramientas provistas por Cypress a través de su Kit de Desarrollo CY3684 EZ-USB FX2LP. Este kit consiste en una parte física cuya parte más destacada es el controlador USB FX2LP y una parte informática consistente en un set de herramientas que permiten la programación del controlador provisto.

También se fundamentó y se explicó en forma detallada la configuración seleccionada y la implementación de esta configuración en el código desarrollado.

## Capítulo 5

# Desarrollo del sistema maestro para la comunicación entre la FPGA y la interfaz Cypress

Hasta el momento, se posee una comunicación que intercambia datos entre una PC y el controlador FX2LP de Cypress. Sin embargo, esto sólo no es suficiente, ya que el sistema debe estar dotado, además, de un dispositivo que sea emisor y receptor de los datos que el controlador intercambia con la PC. Se desprende de los objetivos de este trabajo, que el dispositivo que cumple el rol de suministrar y usar los datos con los que opera el controlador, está compuesto por un FPGA.

Como se menciona en la Sección 3.1, el FPGA que se utiliza en la implementación de este trabajo es un Spartan-VI de Xilinx, provisto en la placa de desarrollo Mojo v3, desarrollada por Embedded Micro.

La Figura 5.1 muestra un esquema en el cual se observa, como productor y consumidor de datos, un desarrollo genérico, implementado dentro del FPGA. Los datos fluyen desde el FPGA al controlador FX2LP a través de una máquina de estados algorítmica (MEA), que también provee las señales de control.

A lo largo de este capítulo se explica en detalle el protocolo que debe seguir el dispositivo maestro que comanda la lectura y escritura de datos en los buffers disponibles en la memoria FIFO esclava que posee el controlador de Cypress. Luego, se desarrolla la máquina de estados que comanda ese intercambio de datos y el código escrito en VHDL que la describe para ser sintetizada en el FPGA. Finalmente se explica el desarrollo de un circuito desarrollado como interconexión entre las distintas placas de desarrollo que se utilizan en este trabajo.

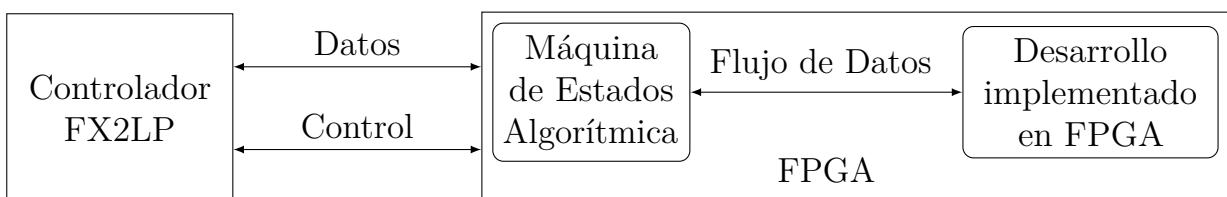


Figura 5.1: Esquema conceptual del flujo de datos hasta el controlador

## 5.1. Señales de Control

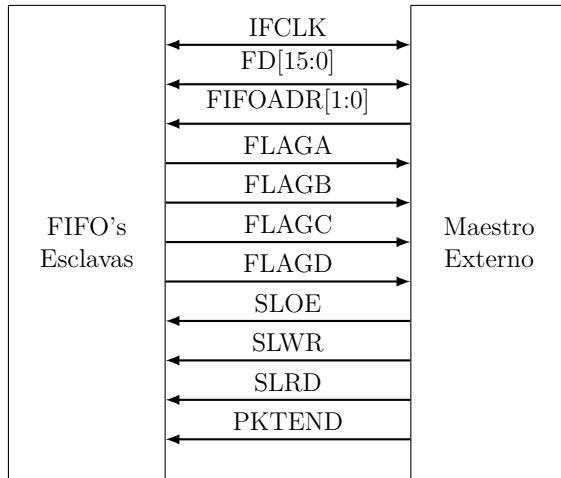


Figura 5.2: Puertos de interfaz entre las FIFO's y un maestro externo

La Figura 5.2 muestra los puertos a través de los cuales se conectan el controlador FX2LP con el FPGA Spartan-IV. Como se detalla en la Sección 4.2.3, Las señales FIFOADR[1:0] se utilizan para seleccionar la memoria FIFO sobre la que se escriben o leen los datos. Cada una de estas memorias está asociada a un extremo (EP) determinado. Estos extremos poseen dirección hexadecimal 02, 04, 06 y 08 para el sistema USB comandado por el  $\mu$ C 8051 incorporado en el integrado FX2LP. Las memorias FIFO tienen dirección binaria "00", "01", "10" y "11" en los puertos FIFOADR[1:0]. Se muestra en la Tabla 5.1 las direcciones asociadas entre cada una de las memorias FIFO y los EP. Se destaca que '0' y '1' en cada puerto FIFOADR equivale a niveles de tensión bajo y alto, respectivamente.

FIFOADR[1:0]	EP (USB)
00	0x02
01	0x04
10	0x06
11	0x08

Tabla 5.1: Direcciones de selección de memoria activa

Los puertos que indican la ocurrencia de eventos particulares en las memorias, como que la memoria se encuentra llena, vacío o el alcance de una cantidad de datos determinada, son programables. Es decir, al momento de realizar la configuración del controlador FX2LP, el desarrollador puede seleccionar que señales estarán presentes en los puertos FLAGA, FLAGB, FLAGC y FLAGD.

La configuración que se implementa en este trabajo, tal como se menciona en el Capítulo 4, dispone a la memoria FIFO 02 como puerto de entrada USB (es decir, salida desde el FPGA) y al puerto 08 como salida USB (o sea, entrada para el FPGA). A su vez, el puerto FLAGA señala que la memoria FIFO 02 está llena y el FLAGB indica que la memoria FIFO 08 está vacía.

El lector puede notar que no se detalla en ninguno de los diagramas temporales presentes en este informe la señal del puerto IFCLK. Esto se debe a que por errores de diseño del alumno,

no es posible implementar un funcionamiento sincrónico, quedando sin uso la señal del puerto señalado.

### 5.1.1. Lectura de datos desde la memoria FIFO

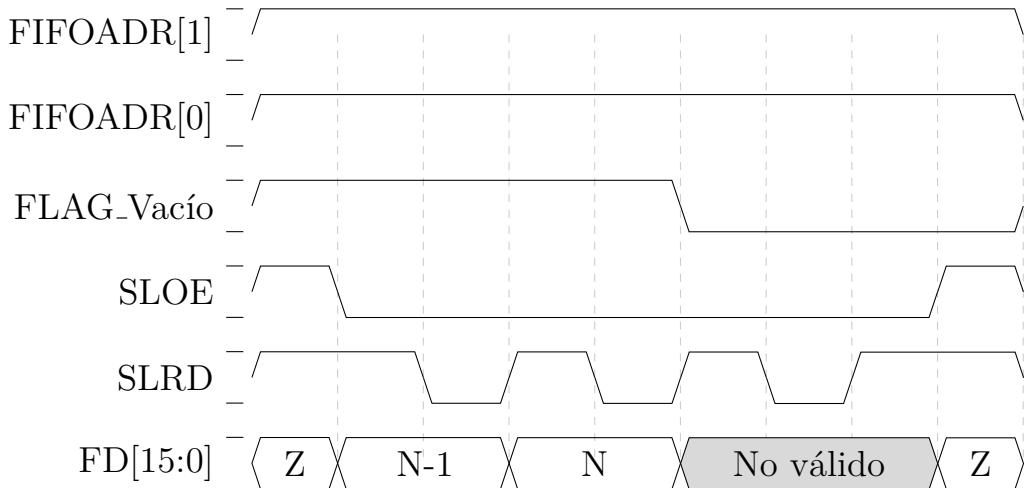


Figura 5.3: Diagrama temporal de la lectura de datos desde la memoria FIFO por un FPGA

Para efectuar operaciones de lectura en régimen asíncrono, como se muestra en la Figura 5.3, en primer lugar, el FPGA debe colocar en los puertos FIFOADR[1:0] la dirección de la memoria sobre la que desea efectuar esta operación. En el caso de la configuración planteada en este trabajo, "11", la que corresponde al EP8. Luego, debe ser activada la señal SLOE, la cual coloca en los puertos FD[15:0] los datos almacenados en la memoria FIFO activa por FIFOADR[1:0]. El dato disponible en la salida de la memoria FIFO siempre será el más antiguo, es decir, el que se almacenó antes. En el cambio de asertiva a negativa de la señal SLRD, la memoria FIFO aumenta un contador que selecciona la dirección del próximo dato, y coloca este dato en el puerto FD[15:0].

Una vez que todos los datos fueron leídos, es decir, que el contador de la memoria ha alcanzado un valor N de datos, iguales a los almacenados, se activa la señal FLAG\_Vacio (para este trabajo, FLAGB). Mientras SLOE no está activo, el puerto FD[15:0] permanece en estado de alta impedancia. En la Figura 5.3 se puede observar también que tanto las señales FLAG\_Vacio, SLOE y SLRD son asertivas en '0'. En otras palabras, dichas señales son activas cuando tienen un bajo nivel de tensión.

### 5.1.2. Escritura de datos en la memoria FIFO

Las señales que intervienen en el proceso de escritura de datos en la memoria FIFO, se encuentran detalladas en el diagrama temporal de la Figura 5.4. Para escribir datos en una memoria FIFO, el FPGA debe seleccionar la memoria a través de FIFOADR[1:0] en primer lugar. Para la configuración de este trabajo, esto es "00", correspondiente al EP2. Luego, se coloca en el bus de datos, donde se encuentran conectados los puertos FD[15:0], el dato a escribir. Se debe tener en cuenta que SLOE debe estar no asertivo, de modo tal que el bus FD[15:0] se

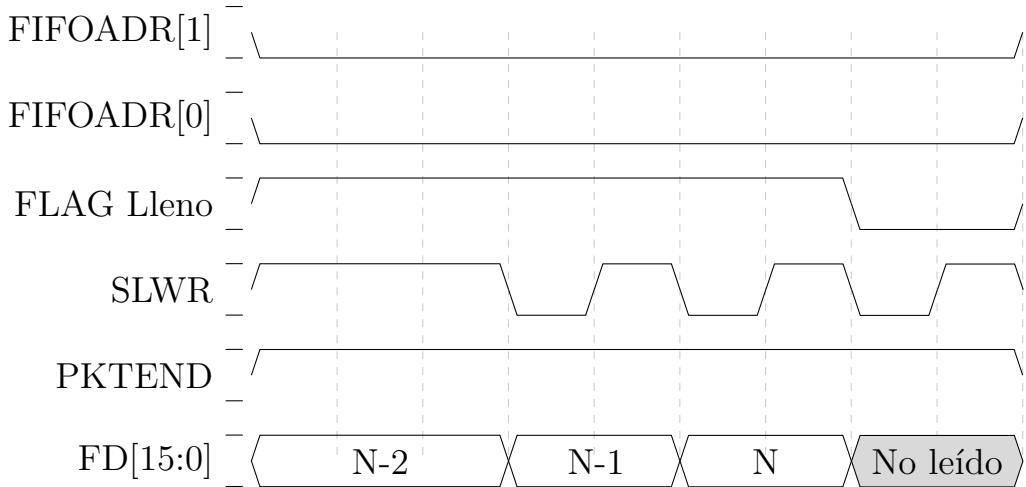


Figura 5.4: Diagrama temporal de la escritura de datos en la memoria FIFO desde un FPGA

encuentre en modo de alta impedancia por parte del controlador FX2LP y no interfiera con la escritura.

Una vez colocado el dato en el bus, se debe activar la señal SLWR. En el flanco asertivo de SLWR, el controlador incrementa el contador que indica la dirección de memoria en donde será almacenado el dato siguiente y deja guardado el dato que leyó en los puertos del bus FD. Como se observa en el diagrama de la Figura 5.4, SLWR es activo en bajo.

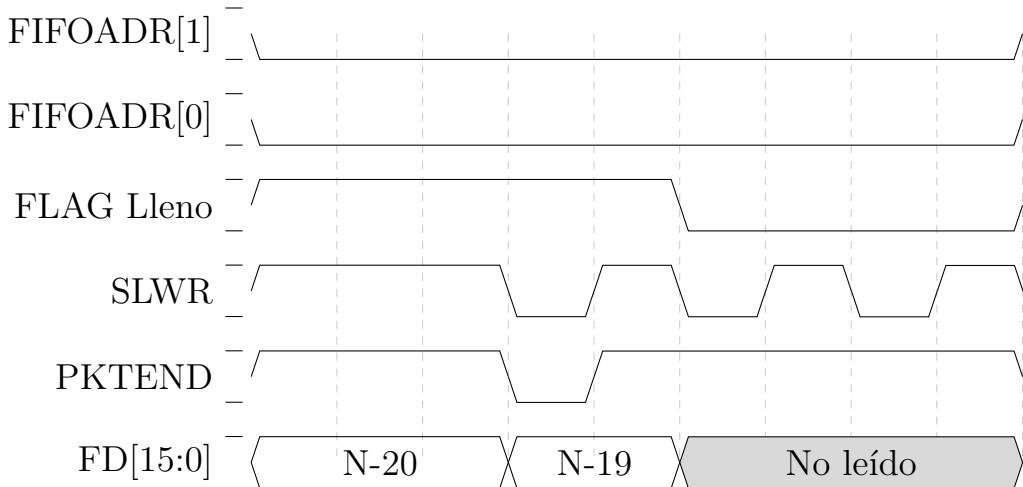


Figura 5.5: Diagrama temporal del funcionamiento del finalizado manual de mensajes

La interfaz FX2LP espera siempre un número determinado de datos, señalizado como N en los diagramas de la Figura 5.4 y la Figura 5.5. Una vez alcanzado dicho número, el paquete queda listo para ser enviado cuando el host lo solicite. Sin embargo, puede ser enviado un número menor de datos en forma manual. Este funcionamiento es provisto a través de la señal PKTEND. Como se observa en la Figura 5.5, cuando PKTEND es asertiva (activa en bajo), la señal FLAG\_Lleno se activa y la memoria FIFO ignora cualquier dato que se envíe a continuación.

## 5.2. Diseño de la Máquina de Estados Algorítmica

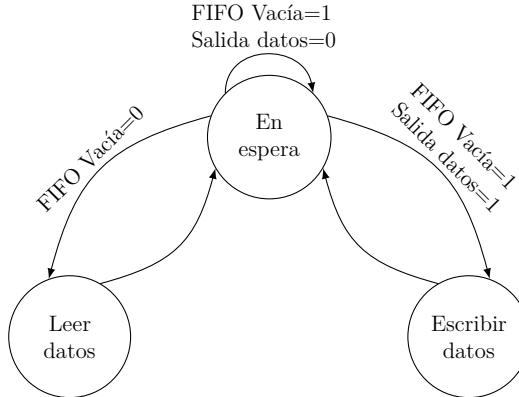


Figura 5.6: Diagrama conceptual de la MEA

A modo conceptual, la maquina de estados algorítmica (MEA) que se implementa en este trabajo es capaz de realizar dos tareas, bien definidas: leer datos desde la memoria FIFO destinada al EP de salida (desde la PC) y escribir datos en la memoria FIFO que corresponde al EP de entrada (hacia el host). En la Figura 5.6 se muestra que cuando la memoria FIFO de salida del host señala que no está vacía, se ejecuta la tarea de lectura de datos. Si la memoria FIFO asignada a la salida del host se encuentra vacía y está activa la salida de datos del FPGA, estos datos serán escritos en la memoria FIFO correspondiente.

Se puede notar que la implementación de la MEA le otorga mayor prioridad a la operación de lectura que a la de escritura de datos. Es decir, siempre que existan datos para leer en la memoria FIFO, serán leídos, aún cuando hayan datos para ser escritos. Se otorga esta prioridad con el objetivo de que la comunicación sea utilizada por sensores que adquieran datos y los transmitan en forma inmediata a la PC y, a su vez, desde la PC se envíe datos que permitan configurar parámetros del sensor en particular de manera esporádica. Así, se prevé que los datos que provengan de la PC sean más distanciados en el tiempo que los de los datos que estuviesen siendo adquiridos por el sensor.

Se considera que la maquina de estados, además comunicarse con la interfaz FX2LP, sea capaz de intercambiar datos con algún sistema genérico implementado en el mismo FPGA que se sintetiza la MEA. A su vez, dicho sistema debe poder indicar el momento en que se producen los datos y deben ser enviados. Como señal de sincronismo, se utilizan las producidas para leer y escribir en las memorias, con el objetivo de optimizar el diseño al mínimo de recursos utilizados.

Con todo esto, la maquina de estados deberá poseer las variables que se observan en la Figura 5.7. Como se muestra, la interfaz que se comunica con que se comunica con el controlador FX2LP posee como variables de entrada para la operación de lectura son el bus de datos FD[15:0] y el FLAG\_Vacio (FLAGB), conforme al diagrama temporal de la Figura 5.3. En el caso de la operación de escritura, se utiliza además el FLAG\_Lleno (FLAGA).

Las variables de salida son los puertos de dirección FIFOADR[1:0], SLRD y SLOE, utilizadas en la operación de lectura, y SLWR para la escritura. Otra señal que se utiliza para la operación de escritura es PKTEND. Esta señal sirve para enviar paquetes más cortos que los esperados por la interfaz. El bus de datos FDATA[0:15] es bidireccional ya que se usa de entrada o salida en función de la operación que se efectúa.

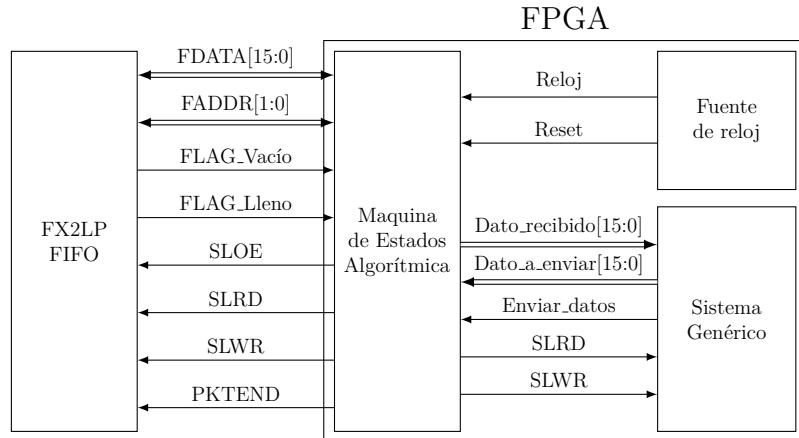


Figura 5.7: Diagrama conceptual donde se observan las variables que intervienen en la MAE

Hacia la comunicación interna del FPGA, se plantea una señal de envío de datos, Enviar\_datos, como puerto de entrada, que le permita al sistema indicar que los datos están siendo producidos. Como señales de salida, se toman SLRD y SLWR como handshake, de forma tal que el sistema conozca cuando un dato es leído y/o escrito en el controlador FX2LP. También se desdoblaron los datos en dos buses diferentes, Dato\_a\_enviar[15:0] de entrada y Dato\_recibido[15:0] de salida.

A su vez, se incorpora una señal de reloj, que estará encargada de temporizar la MEA y una señal de reset, que se encargara de iniciar todos las señales a un valor conocido de referencia, previo al comienzo del ciclo de la MEA. Con base en los diagramas temporales y las variables anteriormente mencionadas, se diseña la maquina de estados que se observa en la Figura 5.8. Se debe notar que FLAGA, FLAGB, SLOE, SLRD y SLWR son activos en bajo.

### 5.3. Descripción de la Máquina de Estados Algorítmica en VHDL

Una vez definidos los puertos y la MAE que se implementa, se está en condiciones de describirlo en un formato que sea sintetizable en una FPGA. El formato utilizado es el lenguaje VHDL (acrónimo del inglés *Very high speed Hardware Description Language*). A su vez, para sintetizar la descripción realizada, se recurre al programa ISE de Xilinx.

Considerando las señales descriptas en la Figura 5.7, se obtendría un sistema que cumple con las especificaciones. Sin embargo, a fin de no dejar señales provistas por la interfaz al aire, se agregan todos FLAGS que brinda el controlador FX2LP en la entidad descripta. Además, se incorporan tres constantes para modificar a criterio del desarrollador las direcciones de entrada y salida, y el ancho del bus de datos, que puede ser de 8 o 16 bits. Por defecto, se utilizan las direcciones y ancho de bus especificados en el Capítulo 4, es decir "11" y "00" como puertos de entrada y salida respectivamente y 16 bits de ancho de bus.

De lo anterior, podemos declarar la entidad que tiene los puertos detallados a continuación:

```

entity fx2lp_interfaz is
    generic(
        constant in_ep_addr : std_logic_vector(1 downto 0) := "00";
        constant out_ep_addr: std_logic_vector(1 downto 0) := "11";
    );

```

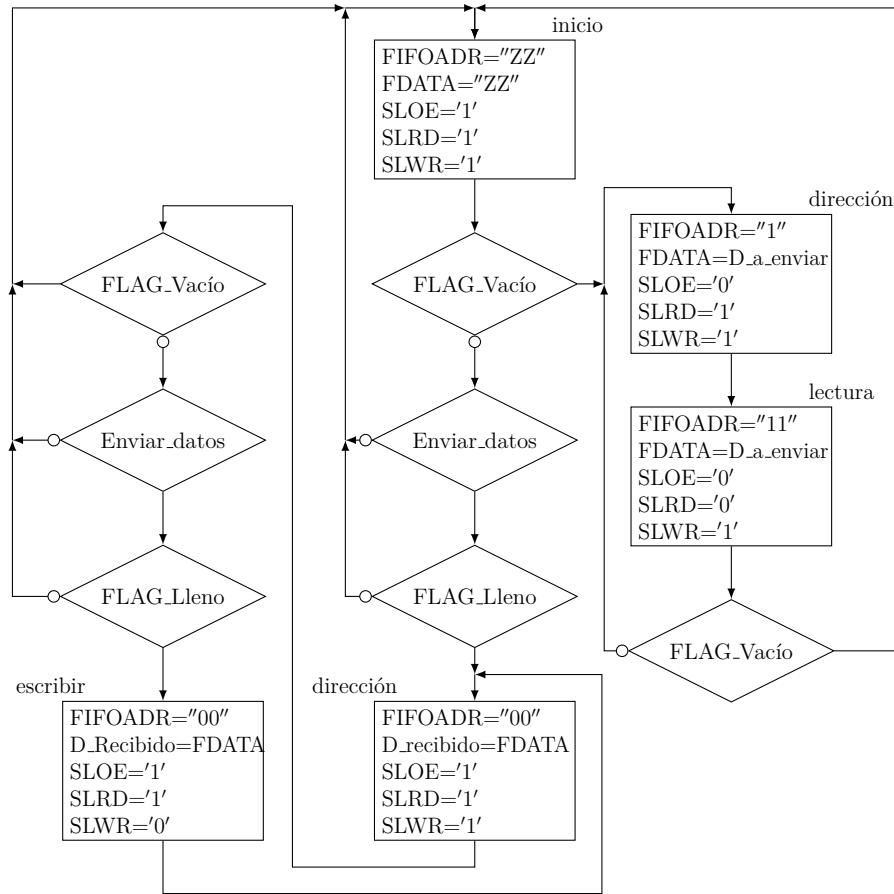


Figura 5.8: Maquina de estados que se implementa

```

constant port_width: integer := 16
);
port(
    reloj: in std_logic;
    reset: in std_logic;
    — desde y hacia la interfaz
    fdata: inout std_logic_vector(port_width-1 downto 0);
    fifoaddr: out std_logic_vector(1 downto 0);
    sloe: out std_logic;
    slrd: out std_logic;
    slwr: out std_logic;
    pktend: out std_logic;
    — EP2 isoc in (hacia pc)
    — EP8 bulk out (desde pc)
    flaga: in std_logic; — EP2_full—>FLAG_Lleno
    flagb: in std_logic; — EP8_empty—>FLAG_Vacio
    flagc: in std_logic; — EP8_full—>sin uso
    flagd: in std_logic; — EP2_empty—>sin uso
    — desde y hacia el sistema

```

```

    enviar_dato: in std_logic;
    d_recibido: out std_logic_vector(port_width-1 downto 0);
    d_a_enviar: in std_logic_vector(port_width-1 downto 0)
);
end fx2lp_interfaz ;

```

Luego, se describe el comportamiento de la máquina de estados que se implementa. El estilo elegido para la descripción cuenta con un registro que determina el estado próximo de la MAE a través de un proceso secuencial. El valor de dicho registro, es volcado a otro que indica el estado actual de la MAE, a través de los flancos del reloj, en una secuencia diferente. Las señales de salida son implementadas en forma concurrente, de manera externa a los procesos que comanda la MEA. Las señales de entrada se encuentran incorporadas en el proceso que determina el estado próximo. La MEA es descripta a través del código que se muestra a continuación:

```

architecture Behavioral of fx2lp_interfaz is
    -- maquina de estados de la interfaz
    type estados_mea is
    (
        inicio ,
        lec_direccion , lectura ,
        esc_direccion , escritura
    );
    signal estado_actual , prox_estado: estados_mea := inicio ;
begin
    --implementacion de la maquina de estados
    interfaz_mea: process(estado_actual , flag_lleno ,
        flag_vacio , enviar_dato)
    begin
        case estado_actual is
            when inicio =>
                if flag_vacio = '0' then
                    prox_estado <= lectura ;
                elsif enviar_dato = '1' then
                    if flag_lleno = '0' then
                        prox_estado <= escritura ;
                    else
                        prox_estado <= inicio ;
                    end if ;
                else
                    prox_estado <= inicio ;
                end if ;

            when lec_direccion =>
                prox_estado <= lectura ;

            when lectura =>
                if flag_vacio = '0' then

```

```

        prox_estado <= lec_direccion ;
    else
        prox_estado <= inicio ;
    end if ;

when esc_direccion =>
    prox_estado <= escritura ;

when escritura =>
    if enviar_dato = '1' then
        if flag_vacio = '1' and flag_lleno = '0' then
            prox_estado <= esc_direccion ;
        else
            prox_estado <= inicio ;
        end if ;
    else
        prox_estado <= inicio ;
    end if ;

when others =>
    prox_estado <= inicio ;
end case ;
end process interfaz_fsm ;
end Behavioral ;

```

En la descripción detallada anteriormente, los flags no coinciden con los puertos declarados. Para salvar esta inconsistencia, se declaran las señales utilizadas, *flag\_lleno* y *flag\_vacio* y se las asigna de forma concurrente a las señales *flaga* y *flagb*, respectivamente. Además, se coloca un inversor para hacer las señales activas en alto. Todo esto apunta a facilitar la lectura y el desarrollo de la descripción.

```

architecture Behavioral of fx2lp_interfaz is
    signal flag_vacio: std_logic ;
    signal flag_lleno: std_logic ;
begin
    flag_lleno <= not flaga ;
    flag_vacio <= not flagb ;
end Behavioral ;

```

A su vez, también son necesarias señales que sirvan como conectores internos desde los puertos hacia los diferentes componentes que se describen.

```

architecture Behavioral of fx2lp_interfaz is
    signal slwr_int:      std_logic := '1';
    signal slrd_int:      std_logic := '1';
    signal sloe_int:      std_logic := '1';
    signal pktend_int:    std_logic := '1';
    signal faddr_int:     std_logic_vector(1 downto 0) := "ZZ";

```

```

signal fdata_sal: std_logic_vector (port_width-1 downto 0);
signal fdata_inent: std_logic_vector (port_width-1 downto 0);
signal reloj_sistema: std_logic;
begin
    reloj_sistema <= reloj;
    slwr <= slwr_int;
    slrd <= slrd_int;
    sloe <= sloe_int;
    faddr <= faddr_int;
    pktend <= pktend_int;
    d_recibido <= fdata_ent;
    fdata_sal <= d_a_enviar;

end Behavioral

```

Con todas las señales definidas y asignadas, y la maquina de estados que se detalló anteriormente, se pueden asignar las señales de salida:

```

architecture Behavioral of fx2lp_interfaz is
    with estado_actual select
        faddr_int <= out_ep_addr when lec_direccion | lectura,
                    in_ep_addr when esc_direccion | escritura,
                    (others => 'Z') when others;

        slwr_int <= '0' when prox_estado = esc_direccion else
                    '1';

        slrd_int <= '0' when estado_actual = lec_direccion else
                    '1';

        pktend_int <= ((not falg_vacio) or enviar_dato);

        with estado_actual select
            sloe_int <= '0' when lectura | lec_direccion,
                        '1' when others;

        with estado_actual select
            fdata <= fdata_sal when escritura | esc_direccion,
                        (others => 'Z') when others;

        with estado_actual select
            fdata_ent <= fdata when lectura | lec_direccion,
                        fdata_ent when others;
end Behavioral

```

Finalmente, resta el reloj que hace avanzar la MAE. A este reloj, se le acoplan dos temporizadores de habilitación. Esto se debe a que se espera que el sistema trabaje a 50 MHz. Sin embargo, para respetar los tiempos de establecimiento y ancho de pulso de las distintas

señales[33], cuando el próximo estado es esc\_dirección se deben esperar tres ciclos de reloj y en el caso de que el próximo estado sea escritura, lec\_direccion o lectura, se debe esperar dos ciclos de reloj. Esto se implementa con dos contadores diferentes, los cuales habilitan o no el cambio de estado. Esto se detalla a continuación:

```

architecture Behavioral of fx2lp_interfaz is
    signal cont3: natural range 0 to 4 := 0;
    signal cont2: natural range 0 to 3 := 0;
    signal disparo3: std_logic := '0';
    signal disparo2: std_logic := '0';
begin
    contador3: process(reloj_sistema, reset, disparo3)
    begin
        if reset = '0' then
            cont3 <= 0;
        elsif rising_edge(reloj_sistema) then
            if cont3 > 0 then
                cont3 <= cont3 - 1;
            elsif disparo3 = '1' then
                cont3 <= 4;
            end if;
        end if;
    end process contador3;

    disparo3 <= '1' when (prox_estado = esc_direccion) else '0';

    counter2: process(reloj_sistema, reset, disparo2)
    begin
        if reset = '0' then
            cont2 <= 0;
        elsif rising_edge(reloj_sistema) then
            if cont2 > 0 then
                cont2 <= count2 - 1;
            elsif disparo2 = '1' then
                cont2 <= 3;
            end if;
        end if;
    end process contador2;

    with prox_estado select
        disparo2 <= '1' when lec_direccion | lectura | esc_direccion,
        '0' when others;

    reloj_mea: process (reloj_sistema, reset)
    begin
        if reset = '0' then

```

```

        estado_actual <= idle;
elsif rising_edge( reloj_sistema) then
    if cont2 = 0 and cont3 = 0 then
        estado_actual <= prox_estado;
    end if;
end if;
end process reloj_mea;
end Behavioral

```

El código completo se puede encontrar en el Anexo ??

## 5.4. Placa de Interconexión

Tal como se menciona en el Capítulo 3, los circuitos integrados utilizados para la implementación de la comunicación USB, estos son el controlador FX2LP de Cypress y el FPGA Spartan VI de Xilinx, vienen incorporados en sendas placas de desarrollo. Para la conexión eléctrica de estos dos chips, se desarrolla una placa de interconexión, es decir, un circuito impreso (PCB, del inglés *Printed Circuit Board*) que sirve para conectar en forma eléctrica dos o más dispositivos. Esto brinda una conexión mucho más robusta, compacta y prolífica que si fuese realizada mediante cables o alambres.

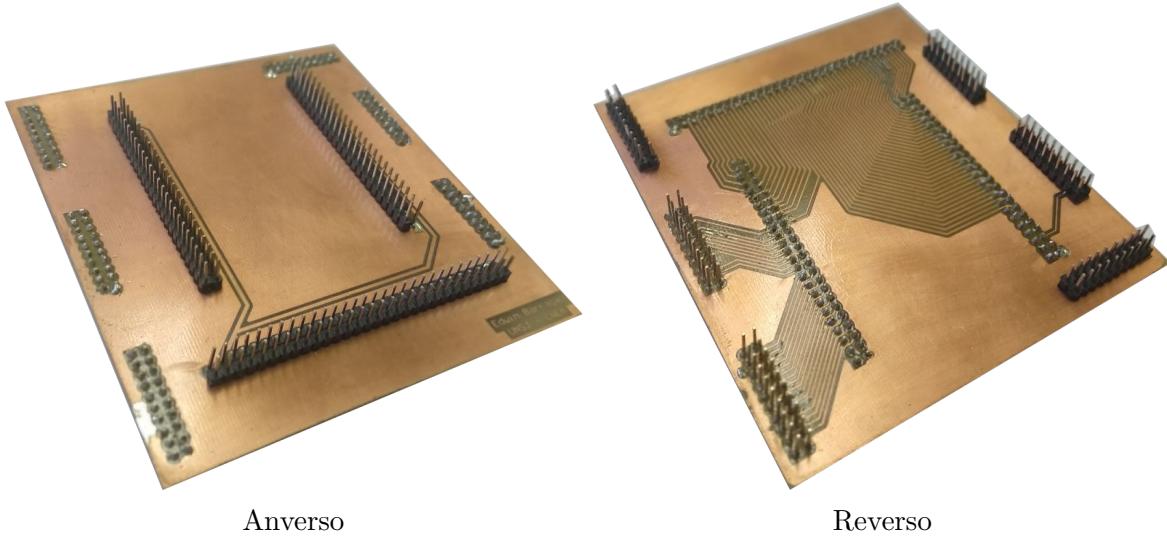


Figura 5.9: Versión 1 de la placa de interconexión

El desarrollo de la placa de interconexión, necesitó de tres versiones para obtener un correcto funcionamiento. La versión número 1, la cual se observa en la Figura 5.9, presenta un problema de contacto eléctrico entre sus dos caras conductoras, debido a no se contaba con la tecnología suficiente para realizar la metalización de los agujeros que conducen la señal de un lado al otro del circuito impreso durante el proceso de fabricación. Por otro lado, en la etapa de montaje, el alumno confunde los pines que debe ser colocados, ya que el reverso necesita pines hembra en lugar de pines macho.

Se realiza una segunda versión, la que se observa en la Figura 5.10. A este PCB se le incorporan vías pasantes para poder conectar las distintas pistas que recorren el circuito

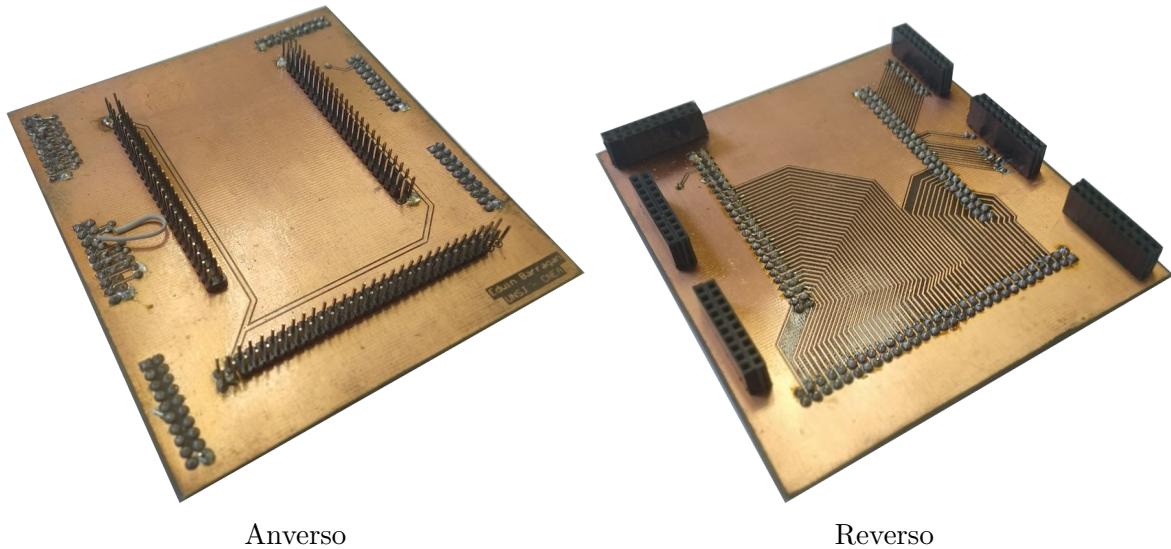


Figura 5.10: Versión 2 de la placa de interconexión

mediante la soldadura de alambres, lo que soluciona el problema de conexión eléctrica. En esta placa se tiene mayor cuidado en la etapa de montaje de los pines. Sin embargo, durante la revisión de los pines se encuentra un defecto en el puerto asignado a la señal de reloj, la cual posee un terminal en un pin que no se encuentra disponible. Esto obliga a la implementación de la comunicación del presente trabajo de forma asíncrona.

Otro defecto que presenta la versión 2 de la placa de interconexión es la inexistencia de un punto para soldadura, que se ocasiona al momento del perforado del impresor. Esto obliga a realizar una conexión mediante un pequeño cable, el cual se puede observar en el anverso de la Figura 5.10.

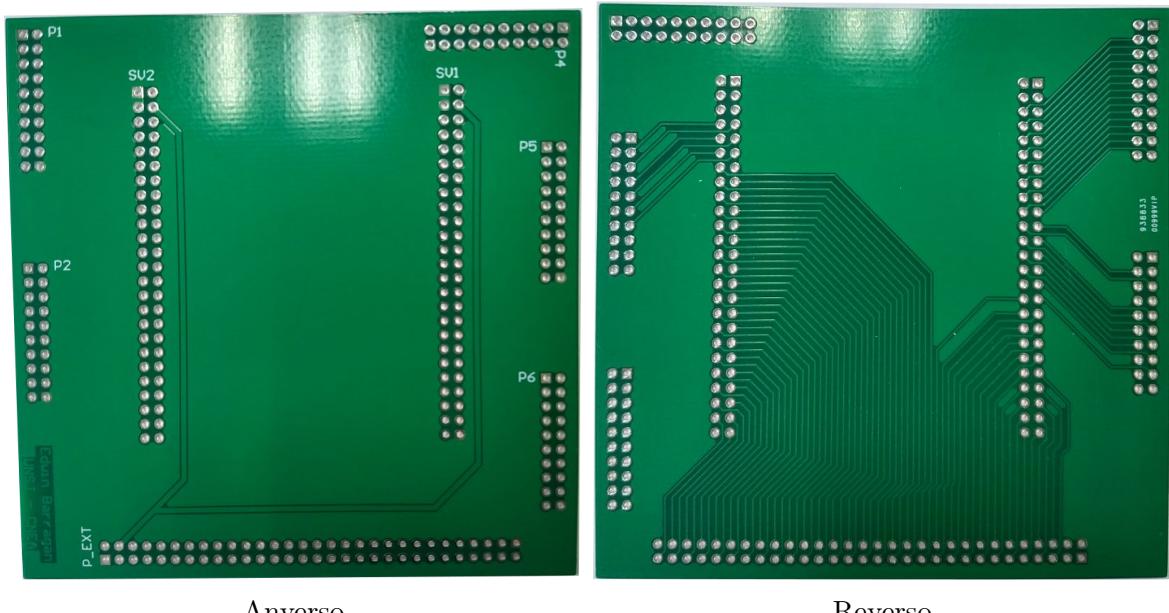


Figura 5.11: Versión 3 de la placa de interconexión

Finalmente, se decide rehacer el circuito de conexión en una tercera versión y solicitar su fabricación en una empresa especializada en la manufactura de PCB para prototipo, radicada en China. En esta versión, fue redirigida la línea de conexión defectuosa, lo que permite conectar las fuentes de reloj e implementar una comunicación síncrona. Esto no fue realizado al momento de la escritura del presente informe, aunque se espera su implementación en trabajos futuros. Además, debido a restricciones en el proceso de fabricación, se debe redimensionar el PCB y hacerlo más compacto.

Gracias a la mejora que brinda la empresa en el proceso de fabricación, se eliminan las conexiones entre las dos caras del impreso mediante la soldadura de alambre y se cambian por agujeros metalizados. Además, se incorporaron conexiones adicionales entre el controlador y el FPGA. Esto permite utilizar el  $\mu$ C 8051 incorporado al controlador FX2LP para realizar tareas adicionales, junto al FPGA. Se adjunta en el Anexo ?? un plano esquemático con las conexiones de la versión 3 del circuito impreso. elaborado.

## 5.5. Sumario del capítulo

Durante el presente capítulo se desarrolló cuales son las señales de control que intervienen en las operaciones de lectura y escritura externa en las memorias FIFO. En base a ellas se explicó el diseño de la maquina de estados algorítmica y su descripción en lenguaje VHDL, para su posterior síntesis en FPGA. Luego se detalló la placa de interconexión realizada para la conexión eléctrica de las placas de desarrollo que contienen al controlador FX2LP y al FPGA Spartan VI

## Capítulo 6

# Verificación y validación del sistema

- 6.1. Depuración de la configuración de la interfaz FX2LP**
- 6.2. Pruebas de la síntesis de la MEA en el FPGA**
- 6.3. Pruebas de la comunicación entre el FPGA y la PC**
- 6.4. Resultados**
- 6.5. Conclusiones**
- 6.6. Trabajos Futuros**



# Bibliografía

- [1] R. Pallàs-Areny and J. G. Webster, *Sensors and signal conditioning*. Wiley-Interscience, 2001.
- [2] D. M. Considine, *Encyclopedia of instrumentation and control*. McGraw-Hill, Inc., 1971.
- [3] A. Perez Garcia, “Curso de instrumentación,” p. 261, 2008.
- [4] J. Fraden, *Handbook of modern sensors: physics, designs, and applications*. New York, NY: Springer New York, 2010.
- [5] E. Slawiński and V. Mut, *Humanos y máquinas inteligentes: conocimiento educativo sobre el comportamiento interno de robots que actúan juno y para el hombre*. Saarbrücken, Alemania: Editorial Académica Española, 2011.
- [6] K. Ogata, *Modern control engineering*. Aeeizh, 2002.
- [7] G. Binnig and H. Rohrer, “Scanning tunneling microscopy,” *Surface Science*, vol. 126, pp. 236–244, mar 1983.
- [8] R. Turchetta, K. R. Spring, and M. W. Davidson, “Digital Imaging in Optical Microscopy - Introduction to CMOS Image Sensors,” (accessed in July 2019).
- [9] S. Mendis, S. Kemeny, and E. Fossum, “CMOS active pixel image sensor,” *IEEE Transactions on Electron Devices*, vol. 41, pp. 452–453, mar 1994.
- [10] C. Hu-Guo, J. Baudot, G. Bertolone, A. Besson, A. S. Brogna, C. Colledani, G. Claus, R. D. Masi, Y. Degerli, A. Dorokhov, G. Doziere, W. Dulinski, X. Fang, M. Gelin, M. Goffe, F. Guilloux, A. Himmi, K. Jaaskelainen, M. Koziel, F. Morel, F. Orsini, M. Specht, Q. Sun, I. Valin, and M. Winter, “CMOS pixel sensor development: a fast read-out architecture with integrated zero suppression,” *Journal of Instrumentation*, vol. 4, pp. P04012–P04012, apr 2009.
- [11] J. Baudot, G. Bertolone, A. Brogna, G. Claus, C. Colledani, Y. Değerli, R. De Masi, A. Dorokhov, G. Dozière, W. Dulinski, M. Gelin, M. Goffe, A. Himmi, F. Guilloux, C. Hu-Guo, K. Jaaskelainen, M. Koziel, F. Morel, F. Orsini, M. Specht, I. Valin, G. Voutsinas, and M. Winter, “First test results of MIMOSA-26, a fast CMOS sensor with integrated zero suppression and digitized output,” *IEEE Nuclear Science Symposium Conference Record*, pp. 1169–1173, 2009.

---

**BIBLIOGRAFÍA**

---

- [12] M. Pérez, J. Lipovetzky, M. Sofo Haro, I. Sidelnik, J. J. Blostein, F. Alcalde Bessia, and M. G. Berisso, “Particle detection and classification using commercial off the shelf CMOS image sensors,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 827, pp. 171–180, aug 2016.
- [13] M. Pérez, J. J. Blostein, F. A. Bessia, A. Tartaglione, I. Sidelnik, M. S. Haro, S. Suárez, M. L. Gimenez, M. G. Berisso, and J. Lipovetzky, “Thermal neutron detector based on COTS CMOS imagers and a conversion layer containing Gadolinium,” *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 893, pp. 157–163, jun 2018.
- [14] C. L. Galimberti, F. Alcalde Bessia, M. Perez, M. G. Berisso, M. Sofo Haro, I. Sidelnik, J. Blostein, H. Asorey, and J. Lipovetzky, “A Low Cost Environmental Ionizing Radiation Detector Based on COTS CMOS Image Sensors,” in *2018 IEEE Biennial Congress of Argentina (ARGENCON)*, pp. 1–6, IEEE, jun 2018.
- [15] T. Hizawa, J. Matsuo, T. Ishida, H. Takao, H. Abe, K. Sawada, and M. Ishida, “ $32 \times 32$  pH image sensors for real time observation of biochemical phenomena,” *TRANSDUCERS and EUROSENSORS '07 - 4th International Conference on Solid-State Sensors, Actuators and Microsystems*, pp. 1311–1312, 2007.
- [16] ON Semiconductor, “NOIP1SN0300A Global Shutter CMOS Image Sensors,” 2014.
- [17] N. Ida, *Engineering Electromagnetics*. Cham: Springer International Publishing, 3th ed., 2015.
- [18] J. F. Wakerly, *Digital Design: principles and practices*, vol. 1. Pearson, 1999.
- [19] M. Perez, F. Alcalde, M. S. Haro, I. Sidelnik, J. J. Blostein, M. G. Berisso, and J. Lipovetzky, “Implementation of an ionizing radiation detector based on a FPGA-controlled COTS CMOS image sensor,” in *2017 XVII Workshop on Information Processing and Control (RPIC)*, pp. 1–6, IEEE, sep 2017.
- [20] R. Biswas, *An Embedded Solution for JPEG 2000 Image Compression Based Back-end for Ultrasonography System*. PhD thesis, IIT, Kharagpur, 2018.
- [21] T. Yanagisawa, T. Ikenaga, Y. Sugimoto, K. Kawatsu, M. Yoshikawa, S.-i. Okumura, and T. Ito, “New NEO search technology using small telescopes and FPGA,” in *2018 IEEE Aerospace Conference*, vol. 2018-March, pp. 1–7, IEEE, mar 2018.
- [22] H. H. Goldstine and A. Goldstine, “The Electronic Numerical Integrator and Computer (ENIAC),” *Mathematical Tables and Other Aids to Computation*, vol. 2, p. 97, jul 1946.
- [23] S. of Cable Telecommuniocations Engineers, *American National Standard ANSI/SCTE 07 2006. Digital Transmission Standard for Cable Television*. Society of Cable Telecommunications Engineers, Inc., 2006.
- [24] I. Micron Technology, “1 / 2-Inch Megapixel CMOS Digital Image Sensor MT9M001C12STM (Monochrome),” pp. 1–35, 2004.

## *BIBLIOGRAFÍA*

---

- [25] IEEE Computer Society, *IEEE Standard for Ethernet*, vol. 2018. 2018.
- [26] IEEE Computer Society, *Part 11 : Wireless LAN Medium Access Control ( MAC ) and Physical Layer ( PHY ) Specifications IEEE Computer Society Specific requirements Part 11 : Wireless LAN Medium Access Control ( MAC ) and Physical Layer ( PHY ) Specifications*, vol. 2012. 2016.
- [27] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips, *Universal Serial Bus Specification*, vol. Revision 2.0. 2000.
- [28] “Usb hardware.” [https://en.wikipedia.org/wiki/USB\\_hardware](https://en.wikipedia.org/wiki/USB_hardware). Ingreso: 8 de agosto del 2019.
- [29] T. Riihonen, *Desing and analysis of duplexing Modes and Forwarding Protocols for OFDM(A) Relay Links*. PhD thesis, 2015.
- [30] Cypress Semiconductor, “EZ-USB ® Technical Reference Manual,” tech. rep., 2014.
- [31] Cypress Semiconductor, “CY3684/CY3684 EZ-USB Development Kit User Guide,” tech. rep., 2014.
- [32] libusb, “libusb 1.0 <https://libusb.info/> - acceso: 04/11/2019.”
- [33] Cypress, “CY7C68013A, CY7C68014A, CY7C68015A, CY7C68016A, EZ-USB(R) FX2LP(TM) USB Microcontroller High-Speed USB Peripheral Controller,” 2017.