ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Proyecto Fin de Carrera

Desarrollo multiplataforma de un PSoC basado en microprocesador ARM para aplicaciones empotradas

AUTOR: Daniel Sánchez Gallego

DIRECTOR: Francisco Javier Garrigós Guerrero
José Javier Martínez Álvarez

Septiembre / 2014

Autor	Daniel Sánchez Gallego
E-mail del Autor	noessys@gmail.com
Directores	Francisco Javier Garrigós Guerrero José Javier Martínez Álvarez
E-mail del Director	javier.garrigos@upct.es jjavier.martinez@upct.es
Título del PFC	Desarrollo multiplataforma de un PSoC basado en microprocesador ARM para aplicaciones empotradas
Descriptores	FPGA, Zynq, Xilinx, Ethernet

Resumen

Uno de los pilares sobre los que se asientan las Telecomunicaciones viene determinado por la circuitería electrónica que permite la comunicación entre los sistemas y las redes, así como la computación propiamente dicha. En este proyecto se plantea el desarrollo de una arquitectura de cómputo de bajo coste y escalable, que además pueda ser configurada para adaptarse a una aplicación específica, como en el caso de los sistemas de cómputo empotrados/dedicados.

Para ello, el objetivo principal del proyecto es la definición de un framework adecuado para la portabilidad de PSoCs basados en el microprocesador ARM y las librerías proporcionadas por Xilinx para la plataforma Zynq de cores prediseñados.

En estas condiciones, el desarrollo de un framework adecuado para facilitar el diseño de un SoC basado en la plataforma Zynq-ARM y orientado hacia la portabilidad, facilitaría enormemente la adopción de esta arquitectura en todo tipo de aplicaciones. Por ello, el objetivo principal de este proyecto es la definición de una serie de estándares, directrices de diseño y herramientas para facilitar la instanciación de arquitecturas de cómputo basadas en ARM sobre plataformas de desarrollo de aplicaciones específicas.

Titulación	Ingeniería de Telecomunicación
Departamento	Departamento de Electrónica, Tecnología de Computadoras y Proyectos
Fecha de presentación	Septiembre - 2014

Índice

1. Introducción y Objetivos

1.1. Introducción

En el marco del procesamiento de señales y de información surge constantemente la problemática de la migración de proyectos a nuevas plataformas. La continua evolución de la electrónica y de las tecnologías de computación da lugar a nuevas arquitecturas de cómputo que incluyen funcionalidades como procesadores vectoriales, plataformas multiprocesador y soluciones para el cómputo paralelo.

De esta forma, se ponen a nuestro alcance sistemas no solo más potentes y rápidos, sino también más eficientes, ofreciendo soluciones para problemas específicos y siendo estos cada vez más flexibles, modulares y escalables.

Cuando se requiere procesar gran cantidad de información, o bien cuando el algoritmo a aplicar sobre ésta es de una complejidad considerable, paralelizar la ejecución puede ser la solución a los largos tiempos de procesamiento requeridos, a la vez que permite evitar los cuellos de botella cuando la tasa de datos ofrecida al sistema es continua. Por ello, son necesarios sistemas hardware donde implementar la lógica de estos algoritmos.

Hace años que las FPGAs (*Field-Progammable Gate Array*) constituyen una importante opción para solucionar estos problemas, debido a su cada vez mayor número de celdas y a las nuevas herramientas de desarrollo. La velocidad de diseño y, por supuesto, su inherente naturaleza reprogramable, las hace mucho más adecuadas en entornos de investigación que los ASICs (*Application-Specific Integrated Circuit*), más costosos en desarrollo y adquisición y completamente rígidos.

Con la reciente salida al mercado de la plataforma Zynq-7000 de Xilinx, se ha pensado en la posibilidad de la migración de proyectos y de nuevos diseños en estos sistemas, que ofrecen nuevas ventajas tanto en el rendimiento como en el desarrollo. Por ello, este proyecto consistirá en un acercamiento a dicha familia de productos, con la finalidad de estudiarlas y documentar los procedimientos y resultados para facilitar la labor de futuros desarrolladores a la hora de resolver problemas de aplicación específicos.

1.2. El Proyecto

La plataforma Zynq-7000 fue lanzada por Xilinx a finales de 2012, y consta de dos partes diferenciadas:

- Una parte no programable (PS o *Processing System*) donde se ejecuta el software, compuesta por un procesador Dual-Core ARM Cortex-A9 que gobierna el sistema.
- Una parte programable (PL o *Programmable Logic*) en la que implementar dispositivos que se comunicarán con la PS a través de una interfaz AXI.

Debido al corto periodo de tiempo transcurrido desde el lanzamiento de la familia de plataformas Zynq-7000 y a su enorme potencial, tanto por el rendimiento que brindan los procesadores y componentes como por la mayor sencillez de desarrollo debido a las nuevas herramientas de Xilinx, este proyecto tendrá como objetivo su caracterización y su investigación en diferentes enfoques para que los resultados obtenidos puedan ser utilizados posteriormente para desarrollar soluciones a problemas específicos.

Ya que utilizar una FPGA para un proyecto implica en muchos casos la transferencia de gran cantidad de información, es necesario contar con un sistema de comunicaciones que pueda hacerse cargo y que no vaya a constituir un cuello de botella para el procesamiento.

Por ejemplo, en una aplicación para el procesado y reducción de imágenes astronómicas es necesario proveer a la placa de las imágenes estelares sobre las que se ejecutará el algoritmo de procesado en la FPGA. Esto implica ser capaces de transmitir estas imágenes y toda la información necesaria con una velocidad suficiente, de forma que el factor limitante sea siempre el procesado de los datos y no el medio de comunicación.

Para este fin, se buscará y estudiará un sistema de transmisión de datos y se tratará de manejarlo y cuantificar sus prestaciones en los diferentes escenarios considerados.

Por otra parte, debido a la posibilidad de desarrollar aplicaciones que se ejecuten directamente en el procesador (modo Standalone) o sobre un sistema operativo (Linux), será necesario analizar las ventajas y los inconvenientes de estas alternativas, así como tratar de resolver los problemas presentados en ambos escenarios.

Por último, los dispositivos desarrollados deben estar bien integrados con el resto del sistema, por lo que se analizarán los mecanismos de comunicación con los diferentes módulos presentes tanto en la PS (*Processing System*) como en la PL (*Programmable Logic*).

1.3. Objetivos

A modo de síntesis del apartado anterior, y analizando con más detalle los objetivos que persigue este proyecto, podemos diferenciar una serie de fases de que constará:

1. Búsqueda de documentación e información sobre la nueva plataforma Zynq-7000 y sus herramientas de desarrollo.

2. Familiarización con las herramientas de desarrollo Vivado y el entorno de programación Xilinx SDK. Para este fin se cuenta con una serie de labs o tutoriales proporcionados por Xilinx que cubrirán desde la tarea de desarrollo hasta la de programación y testeo en la plataforma.

3. Estudio del medio de comunicación que se utilizará para transferir información a la placa. Se justificará la opción escogida (Gigabit Ethernet) entre las distintas alternativas y se tratará de conocer y utilizar las librerías encargadas de su uso, tanto en un medio Standalone como en uno con un sistema operativo Linux.

4. Una vez se disponga de un medio de comunicación capaz de transmitir y recibir información, se realizarán diferentes benchmarks o medidas de rendimiento con el fin de optimizar y caracterizar este canal de comunicación para conocer así sus posibilidades y, especialmente, sus limitaciones.

5. Se estudiará el problema de la integración de dispositivos de catálogo o personalizados (implementados en la FPGA o PL) con el resto del sistema. El manejo de estos componentes puede no presentar demasiados problemas cuando se realiza desde una aplicación Standalone, pero la solución no es directa cuando existe un sistema operativo de por medio.

6. Complementando a la fase anterior, y con el fin de poder controlar debidamente el estado de los dispositivos y la transferencia de información, se tratará de capturar las interrupciones que puedan lanzar los periféricos de la parte programable cuando se trabaja con un sistema operativo. Se desarrollará un ejemplo sencillo y se implementarán soluciones para hacer accesibles estos dispositivos desde una aplicación de usuario, así como para transferir información entre éstos.

2. Estado de la Técnica

2.1. Introducción

A finales de 2012, Xilinx sacó al mercado la plataforma Zynq-7000 con su nueva familia de productos. La plataforma Zynq-7000 es un sistema PSoC (*Programmable System on a Chip*) constituido por dos partes diferenciadas pero interconectadas: la parte no programable (PS o *Processing System*) y la parte programable (PL o *Programmable Logic*). La PS la constituyen dos núcleos de procesamiento ARM Cortex-A9 que gestionan los dispositivos externos, y ofrece la posibilidad de interconexión (mediante un bus AXI) con la lógica que pueda ser implementada en la zona reconfigurable o PL.

Con las FPGAs tradicionales era posible integrar sistemas completos, programando su lógica con los procesadores, memorias y periféricos o interfaces requeridos. Sin embargo, el espacio ocupado por estos componentes suponía un problema en muchos casos en los que la inclusión de coprocesadores era necesaria. Las plataformas Zynq solucionan este problema separando la parte de procesamiento secuencial con la incorporación de los procesadores ARM Cortex-A9 Dual-Core, potentes y ampliamente soportados y documentados. Éstos ejecutan el software, dejando así todo el espacio de la FPGA libre para ser implementado por el usuario.

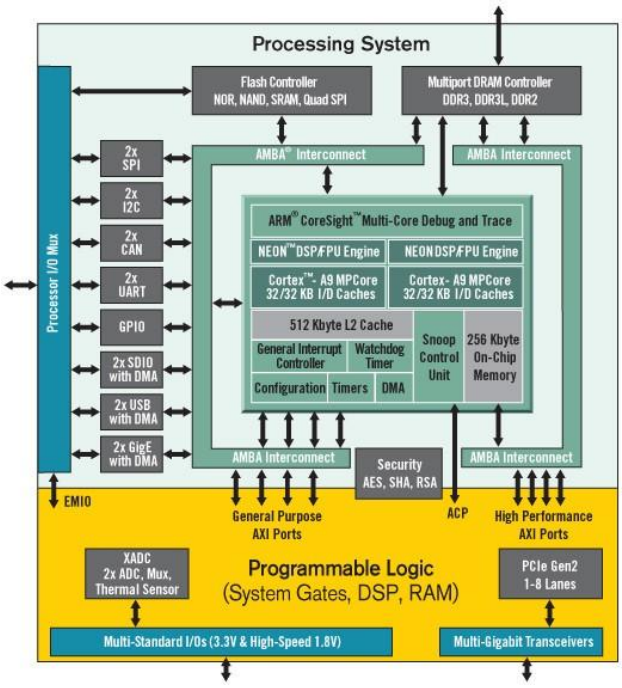

Figura 2.1: Diagrama de la arquitectura de una plataforma Zynq

La figura anterior muestra un diagrama de la estructura y características que presentan los PSoC Zynq-7000. En ella pueden distinguirse claramente las dos partes de que consta la plataforma. La PS constituye un sistema completo formado por los procesadores y conexiones a las memorias y distintas unidades internas, así como interfaces a los dispositivos físicos externos de cada placa. La PL la constituye una FPGA con comunicación física directa con el exterior y la posibilidad de interconexión entre la lógica implementada y la PS a través de buses AXI.

2.2. Hardware

En la actualidad, dos grandes compañías compiten en el mercado de los dispositivos programables como las FPGAs: Xilinx y Altera.

Tras el anuncio de Xilinx de sus SoCs Zynq hace unos años, Altera respondió anunciando sus SoCs Cyclone V y Arria V, que integran también procesadores ARM Cortex A9 Dual-Core y periféricos físicos con una parte de lógica programable.

Históricamente, Xilinx ha acostumbrado a estar un paso por delante en cuanto a tecnología, innovación en sus productos, herramientas de desarrollo y soporte. Esto ha hecho que posea una mayor cuota de mercado que sus rivales, por lo que también cuenta con una comunidad más grande, un incentivo fundamental para los desarrolladores.

Por estas razones se ha decidido dedicar este proyecto al estudio de una plataforma Zynq, con el fin de desarrollar sobre ellas futuros trabajos.

Dentro de la familia Zynq-7000 se encuentra se encuentra la placa que se utilizará: una **ZedBoard**. Se trata de una placa de desarrollo completa y muy versátil, con multitud de periféricos y funciones para soportar un amplio rango de aplicaciones, e ideal para prototipado rápido y prueba de proyectos.

Otra opción disponible es la **MicroZed**, una placa muy interesante debido a que incorpora todo el sistema en el tamaño de una tarjeta de crédito, incluyendo interfaz USB, Gigabit Ethernet y GPIO, entre otros. Si bien prescinde de muchos puertos presentes en placas como la Zedboard, es mucho más adecuada que éstas para la implementación final de proyectos donde se busque potencia de procesamiento, debido a su bajo coste y su reducido tamaño, que permitiría la adquisición e interconexión de varias de estos dispositivos en racks, logrando así la potencia de un supercomputador en un espacio muy reducido y por apenas unos pocos miles de euros.

En un punto intermedio a las dos placas anteriores se encuentra la placa **ZYBO**, y para potencias y posibilidades superiores (a precios también considerablemente mayores) se encuentran la **ZC702** y la **ZC706** y placas base como la **Zynq MMP** y la **Zynq Mini-ITX**, que completarían a la familia Zynq-7000[3].

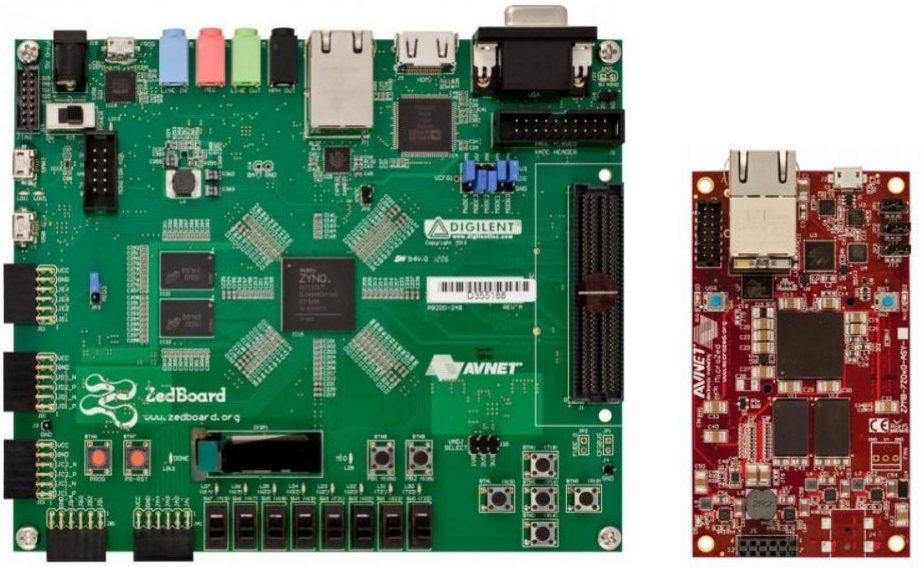

Figura 2.2: Zedboard (izquerda) y MicroZed (derecha)

2.3. Software

Para completar introducción de la plataforma Zynq-7000, Xilinx lanzó un nuevo conjunto de herramientas de desarrollo para sustituir al anterior entorno de diseño, ISE, que no estaba completamente preparado para todas las funcionalidades que ofrecían los nuevos productos. Así surgió la suite de diseño **Vivado**, que incluye diversas herramientas:

- **Vivado**: Es la herramienta principal para el diseño hardware de la plataforma. Completamente preparada para las plataformas Zynq, este entorno permite tanto la configuración y personalización de la parte del *Program System* (PS) de estas plataformas como la adición e interconexión de bloques de lógica o IPs (*Intellectual Properties*) en la FPGA o PL, todo ello en un entorno que integra ambas secciones. Ofrece soluciones para la automatización de procesos y otras ventajas que permiten agilizar y acelerar considerablemente el tiempo de desarrollo de un diseño, al tiempo que ofrece integración entre el resto de herramientas que componen esta suite.

- **Vivado HLS**: Con el crecimiento cada vez más rápido de la complejidad en los algoritmos que implementan los bloques de coprocesamiento hardware, los lenguajes de descripción hardware (HDL) presentan cada vez más desventajas en cuanto a la complejidad y el tiempo en el desarrollo de éstos. Por ello, las herramientas HLS (*High-Level Synthesis*) son cada vez más populares. Vivado HLS es una de estas herramientas, la cual permite acelerar de forma muy considerable el tiempo de creación de IPs mediante su descripción en lenguaje C, C++ y System C, que posteriormente será traducido a un lenguaje de descripción hardware y exportado como un bloque más al catálogo de IPs de Vivado.

- **Xilinx SDK**: Es la herramienta para el desarrollo de aplicaciones software que se ejecuten sobre una plataforma Zynq diseñada previamente en Vivado. Permite la configuración de librerías y de los drivers de los dispositivos utilizados, la programación de la FPGA y la generación de FSBLs (*First Stage Boot Loader*) para sistemas Linux, entre otras opciones.

3. El Medio de Comunicación

3.1. Introducción

Todo mecanismo de procesado de datos requiere, lógicamente, una introducción de datos. La naturaleza de las FPGAs como dispositivos configurables que permiten la implementación en hardware de algoritmos y sistemas señala siempre a una de las principales razones que las hacen tan atractivas: el alto grado de paralelismo que ofrecen. Por ello, es natural que su uso esté destinado en muchas ocasiones al procesado de un flujo constante de información.

Aunque la velocidad de procesado dependa de la lógica implementada en cada aplicación, ésta nunca debe ser mayor al del flujo de datos suministrado por el sistema. De poco sirve desarrollar dispositivos muy veloces en cuanto a procesamiento si el sistema de comunicación que le transfiere los datos no es lo suficientemente rápido. En resumen: el medio de comunicación jamás debe ser el factor limitante del sistema. Por ello, es importante escogerlo adecuadamente, o podría convertirse en un cuello de botella.

En la mayoría de estos casos, el escenario que se presentará será parecido al siguiente:

Figura 3.1: Esquema del sistema

Debido a que se busca un sistema de transferencia sencillo, rápido y ampliamente soportado, las alternativas que se plantearon fueron reducidas a dos:

- Gigabit Ethernet: Es el sistema de transferencia más extendido y flexible con el que se cuenta. Soporta una transferencia de datos a un máximo teórico de 1Gbps, y la creación de una red que pudiera estar formada por racks de PSoCs no implicaría más que uno o varios switches y los cables necesarios.

- USB 3.0: Debido a que la plataformas Zynq-7000 cuenta con soporte para USB 3.0 sería posible utilizar este medio de comunicación, que cuenta con una tasa de transferencia teórica de 5Gbps. Sin embargo, este estándar no está tan extendido, y nunca podríamos alcanzar velocidades reales cercanas a la teórica.

Teniendo en cuenta que Ethernet ofrece una mayor compatibilidad y sencillez a la hora de manejar, está más documentado y extendido que el protocolo USB 3.0, permite una mayor modularidad, y que a priori la velocidad que permite alcanzar es suficiente para nuestros requerimientos, escogeremos la opción de Gigabit Ethernet para nuestras comunicaciones. Esto no implica que no fuera muy interesante estudiar la alternativa de emplear USB 3.0, que podría ofrecer una velocidad de transferencia mayor. Por tanto, será un tema que caerá en el apartado de líneas futuras para la investigación sobre la plataforma.

3.2. Descripción del sistema

El escenario que se empleará será el descrito en el apartado anterior: un PC conectado de forma directa a una placa de desarrollo Zynq a través de un cable Ethernet. El PC ejecutará un sistema operativo Windows 7 (por supuesto, esto solo es relevante desde el punto de vista del software que se ejecutará sobre él) y cuenta con una tarjeta Gigabit Ethernet. La placa a la que está conectado es una Zedboard, aunque debido a que únicamente utilizaremos un puerto Ethernet (Gigabit también) y un puerto serie, el escenario y el software deberían ser compatibles con el resto de placas de la familia Zynq.

En este escenario se llevará a cabo un proceso de transferencia de archivos con el fin de determinar la velocidad que puede ser alcanzada por la plataforma Zynq tanto en transmisión como en recepción. Para ello, cada sistema (PC y Zedboard) ejecutará un programa: El PC transmitirá un archivo determinado por Ethernet, y la Zedboard lo recibirá. Una vez concluida esta transmisión, la Zedboard retransmitirá al PC el archivo recibido. En ambos extremos se medirán las estadísticas de transferencia (velocidad, tasa de pérdida de paquetes, etc.) y se verificará la integridad de los archivos recibidos.

El protocolo utilizado para la transferencia a través de paquetes Ethernet se describe en el siguiente apartado.

Debido a la dependencia del hardware utilizado en ambos extremos a la hora de establecer un entorno de pruebas de rendimiento, se exponen a continuación las especificaciones y consideraciones más relevantes relativas al PC que actuará como extremo opuesto a la Zedboard durante las comunicaciones:

- Especificaciones:
    - CPU Intel® Core™ i5 540M @ 2.53GHz (2 núcleos, 4 hilos).
    - 4GB DD3 SDRAM 1066 MHz
    - Tarjeta de red: Intel® 82577LC Gigabit Network Connection

- Consideraciones:
    - El entorno de trabajo es el sistema operativo Windows 7.
    - La aplicación cliente-servidor para pruebas de rendimiento sobre Ethernet, y que implementa el protocolo descrito en el siguiente apartado, será desarrollada utilizando como base la librería WinPcap[7] [8].

3.3. Formato de trama y protocolo de pruebas

Aunque existen herramientas muy fiables y populares (Iperf, por ejemplo) para la medida del rendimiento de una red de datos Ethernet, se debe desarrollar un sistema propio debido al problema de la portabilidad entre las distintas plataformas que se utilizarán (Windows, Linux y Standalone).

Debido a los requisitos de la comunicación, el protocolo empleado debe ser robusto y garantizar la entrega de todos los paquetes sin errores, por lo que debe incorporar un sistema de solicitud de retransmisión para los paquetes perdidos. En un escenario como el que se presenta, donde las pruebas realizadas implican llevar al límite la velocidad de transmisión y recepción de los dispositivos, las características de los sistemas puede suponer fácilmente un cuello de botella. Para tramas Ethernet de longitud máxima (1514 bytes), una velocidad de transferencia teórica de 1Gbps supondría la recepción de un paquete de datos cada 12 microsegundos. En ese intervalo de tiempo se debe inspeccionar la trama, actuar conforme las características del protocolo de transferencia y gestionar su copiado y almacenamiento en memoria, lo que puede resultar imposible si el proceso no se realiza eficientemente.

Por todo esto, el protocolo empleado debe ser sencillo pero robusto, ya que la pérdida de paquetes puede resultar frecuente. Por tanto, este protocolo incorporará un sistema de transferencia por ventanas para optimizar la velocidad de la transmisión y un sistema de confirmaciones (ACKs) y de solicitudes de retransmisión de paquetes perdidos (*Selective ACKs* o SACKs).

El formato de trama utilizado es el siguiente:

MAC Destino	**MAC Origen**	**Tipo**	**Nº de paquete**	**Payload**
6 bytes	6 bytes	2 bytes	2 bytes	1498 bytes

El tamaño de las tramas utilizadas es el máximo permitido, sin incluir CRC. El objetivo es minimizar el tamaño del encabezamiento de los paquetes para maximizar la velocidad. Hubiera sido muy interesante y conveniente la utilización de Jumbo Frames para extender la longitud del payload de los paquetes hasta 9000 bytes (seis veces más que las tramas convencionales), pero éstas carecen de soporte en las plataformas Zynq.

Por último, se muestra un diagrama del funcionamiento básico del protocolo desarrollado. Se trata, por tanto, de un protocolo basado enteramente en software y dependiente de las librerías de manejo de Ethernet y otras como temporización o listas dinámicas. Debido a su carácter de uso para pruebas y *benchmarking*, y existiendo una gran variedad de protocolos robustos y estándares, más conocidos y documentados y con una implementación mucho más eficiente, se aconseja el estudio y utilización de otros protocolos alternativos para el desarrollo de aplicaciones que requieran alta fiabilidad y rendimiento.

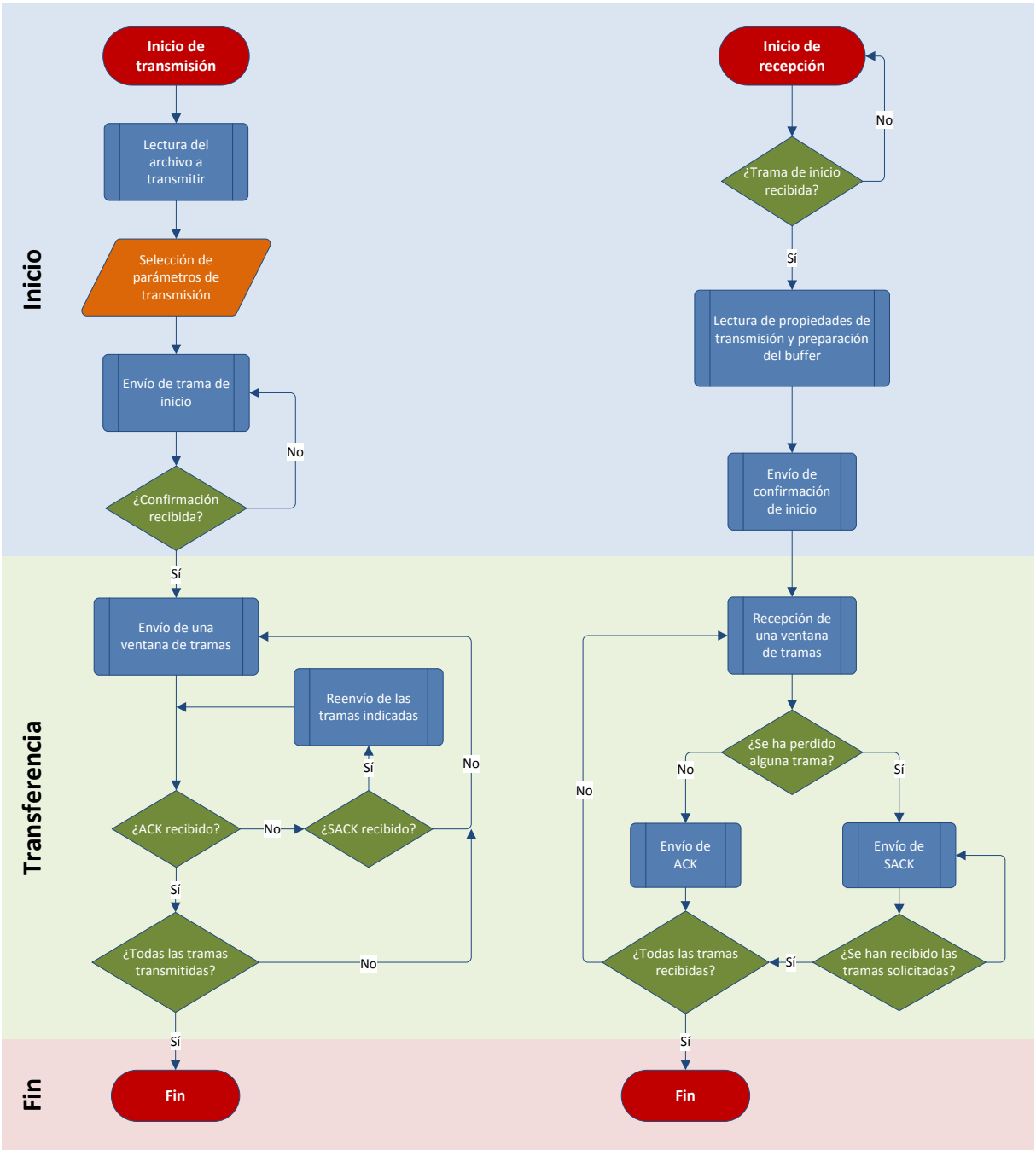

Figura 3.2: Diagrama de flujo del protocolo diseñado

3.4. Preparación de la plataforma

El primer paso para tratar de trabajar con el dispositivo Ethernet del sistema es configurar la plataforma, de forma que ésta incluya, al menos, las funcionalidades que necesitamos: un puerto Ethernet para transferir datos y otro RS232 para poder recibir los mensajes que envíe la placa a través de la comunicación por consola. La plataforma Zynq cuenta en su PS con soporte para muchos más periféricos y funcionalidades; sin embargo, no los necesitaremos por el momento.

Este apartado servirá de guía básica para el uso de la herramienta de desarrollo Vivado, con la que se configurará la PS y la PL de acuerdo a nuestras especificaciones; y del entorno Xilinx SDK, basado en Eclipse, con el que se desarrollará el software que se ejecutará en la plataforma anteriormente definida.

3.4.1. Configuración de la plataforma: Vivado

El proceso de configuración se dividirá en pasos. En esta memoria, el software empleado ha sido la versión Vivado 2014.2. Versiones anteriores (y, previsiblemente, algo posteriores) no deberían presentar cambios significativos que impidan el seguimiento del proceso. Sin embargo, se recomienda trabajar con la versión más moderna disponible, ya que las actualizaciones pueden introducir cambios importantes, especialmente en el aspecto de funcionamiento y corrección de errores.

1. Creación del proyecto en Vivado

1.1. Ejecutar el software Vivado 2014.2 (o la versión disponible).

1.2. Para crear un nuevo proyecto se hace clic en **Create New Proyect** en la ventana de bienvenida, o bien se utiliza el menú **File->New Proyect**…

1.3. Se pulsa **Next** para avanzar al formulario **Project Name** de selección del nombre del proyecto y su ubicación. La opción *Create project subdirectory* creará un directorio con el nombre del proyecto en la ruta especificada, por lo que es útil para separar y organizar los proyectos que se realicen. Este proyecto se llamará ***"ethernet_test"***. Clic en **Next**.

1.4. En el formulario **Project Type** encontraremos una serie de opciones para escoger el tipo de proyecto. Se seleccionará **RTL Project** debido a que se quiere poder editar todo nuestro diseño y realizar el análisis, la síntesis y la implementación (aunque en este caso no será necesario ya que no incluiremos lógica en la PL). Clic en **Next**.

1.5. En las siguientes tres ventanas (**Add Sources**, **Add Existing IP (optional)** y **Add Constaints (optional)**) se incluirán los archivos de código HDL, se podrán importar las IPs o dispositivos de algún directorio y se podrán añadir archivos de definición de constantes al proyecto. En este proyecto no es necesario incluir ninguno de estos archivos, y siempre podrá hacerse posteriormente. Por tanto, se hace clic en **Next** hasta llegar al formulario **Default Part**.

1.6. En nuestro caso, como estamos utilizando una placa de desarrollo ZedBoard, seleccionaremos *Boards* en el cuadro *Specify* y marcaremos la opción **ZedBoard Zynq Evaluation and Development Kit** en el cuadro inferior. La selección de otra placa de la misma familia no debería suponer cambios en el desarrollo del proyecto debido a las similitudes. Clic en **Next**.

1.7. Llegaremos a la última ventana del asistente. Pulsamos **Finish** para crear el proyecto y comenzar a editarlo.

2. Creación del Block Design y generación del sistema

1.8. Para crear un nuevo diseño basado en bloques hardware seleccionamos **Create Block Design** en la barra **Flow Navigator->IP Integrator**. Seleccionamos un nombre para el diseño ("*system*" en este caso).

1.9. En la pestaña Diagram que aparecerá podremos añadir bloques de hardware prediseñados del catálogo o bloques personalizados creados con anterioridad. Se pulsará **Add IP** para abrir el catálogo de IPs.

1.10. En el cuadro que aparece buscamos el componente **ZYNQ7 Processing System** y hacemos doble clic para añadirlo al diseño. Esto añadirá el sistema basado en el procesador ARM Cortex-A9 correspondiente a la plataforma Zynq sobre la que estamos desarrollando.

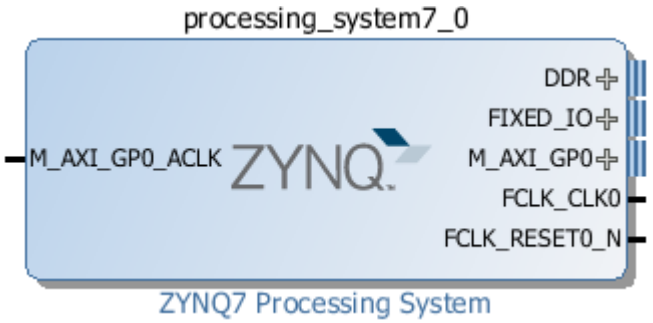

Figura 3.3: Bloque PS de un sistema Zynq

1.11. El nombre asignado a los bloques incorporados puede ser modificado. En nuestro caso, seleccionaremos el bloque creado y en la ventana **Block Properties** introduciremos el nuevo nombre: **processing_system7**.

1.12. Pulsaremos sobre la opción **Run Block Automation** en la parte superior de la ventana de diseño y seleccionaremos el componente /**processins_system7**. Esto mostrará un cuadro de diálogo como el mostrado en la siguiente figura. Pulsamos **OK** y veremos que dos puertos del sistema, DDR y FIXED_IO, han sido añadidos como puertos externos, a la vez que otros son ahora visibles. A pesar de que el proceso ha sido automático, todos estos cambios pueden introducirse de forma manual utilizando las herramientas que ofrece Vivado.

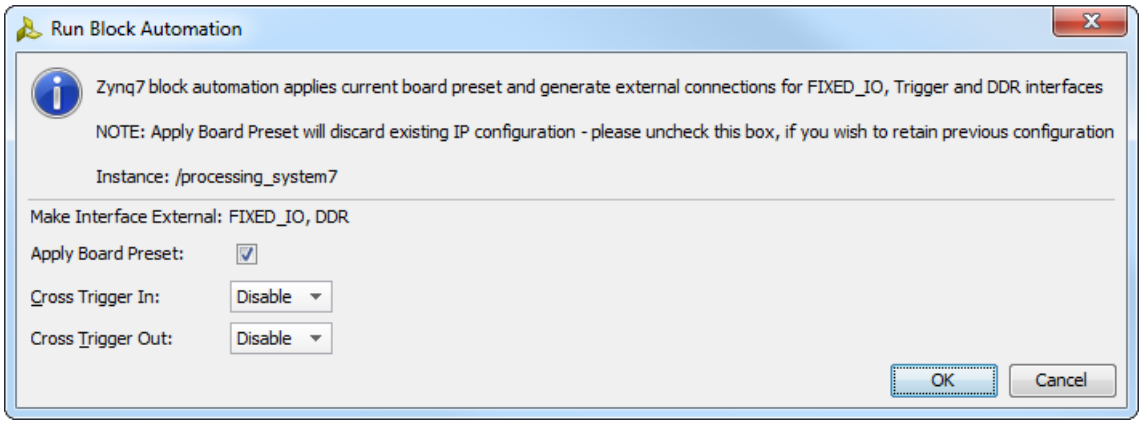

Figura 3.4: Asistente de generación de conexiones externas

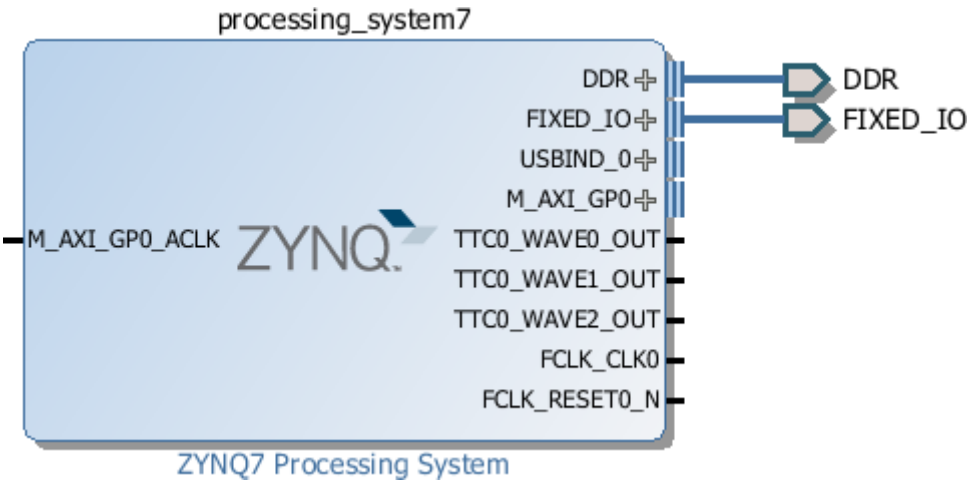

Figura 3.5: Bloque Zynq con conexiones externas

NOTA: Todo el proceso realizado hasta este punto podría considerarse genérico para la mayoría de los proyectos realizados sobre una plataforma Zynq. Proyectos más complejos podrán requerir la inclusión de bibliotecas de componentes o de archivos de código para describir secciones del sistema, aunque puede realizarse más adelante. Sin embargo, se trata de una buena inicialización de proyecto.

3. Configuración de la plataforma ZYNQ

El siguiente paso será configurar la plataforma para que se adapte a las especificaciones de nuestra aplicación. En este caso, y como se ha comentado antes, únicamente será necesario que seamos capaces de utilizar el dispositivo Ethernet y un puerto de comunicaciones RS232. Por tanto, habilitaremos estas funcionalidades y deshabilitaremos el resto para no sobrecargar el proyecto de forma innecesaria. No obstante, el proceso de habilitación de otras características es intuitivo y similar al que se va a describir.

1.13. Para configurar la plataforma, hacemos doble clic sobre el bloque **processing_system7** para que se abra la ventana **Re-costumize IP**. Ésta muestra un diseño con todos los bloques con los que cuenta el sistema (PS), y que podremos habilitar y deshabilitar. Los bloques configurables son aquellos marcados en verde.

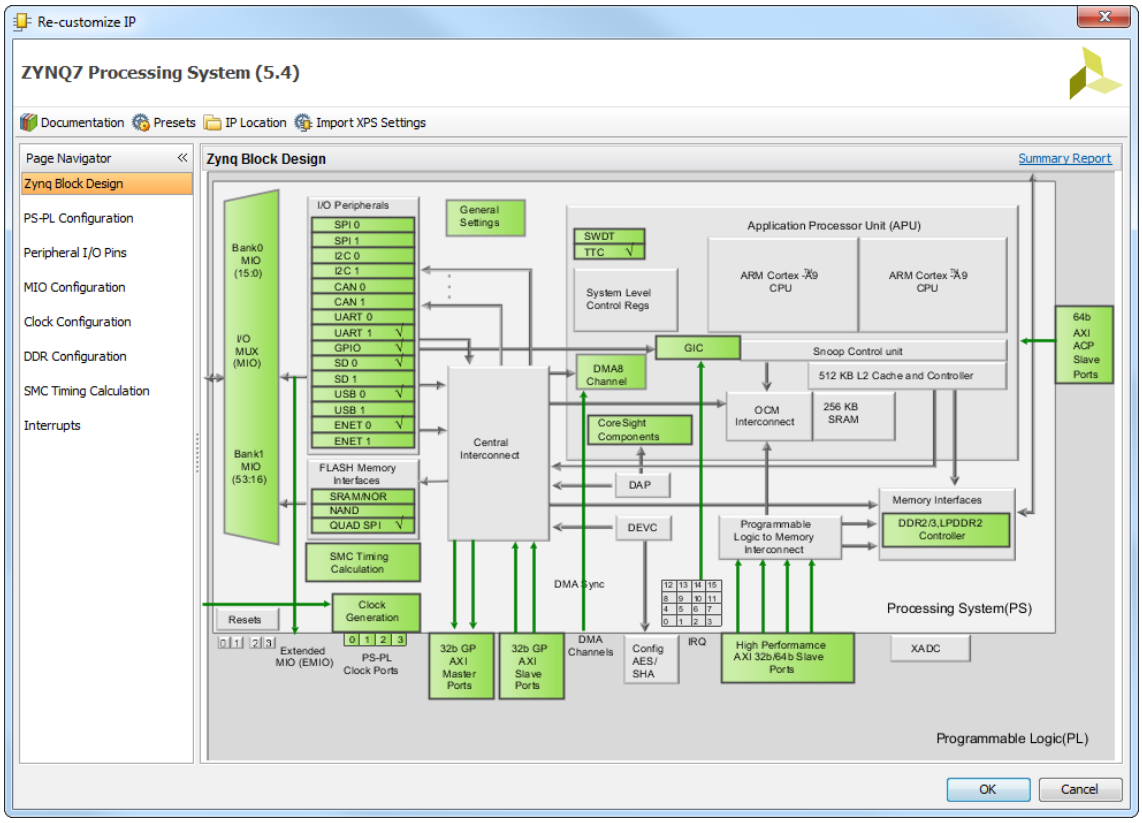

Figura 3.6: Ventana de configuración del bloque Zynq (PS)

1.14. En **MIO Configuration** desplegaremos **I/O Peripherals** y deshabilitaremos todos los periféricos excepto **ENET0** (Ethernet) y **UART1** (RS232). Deshabilitamos también los timers y las interfaces de memoria para simplificar el diseño.

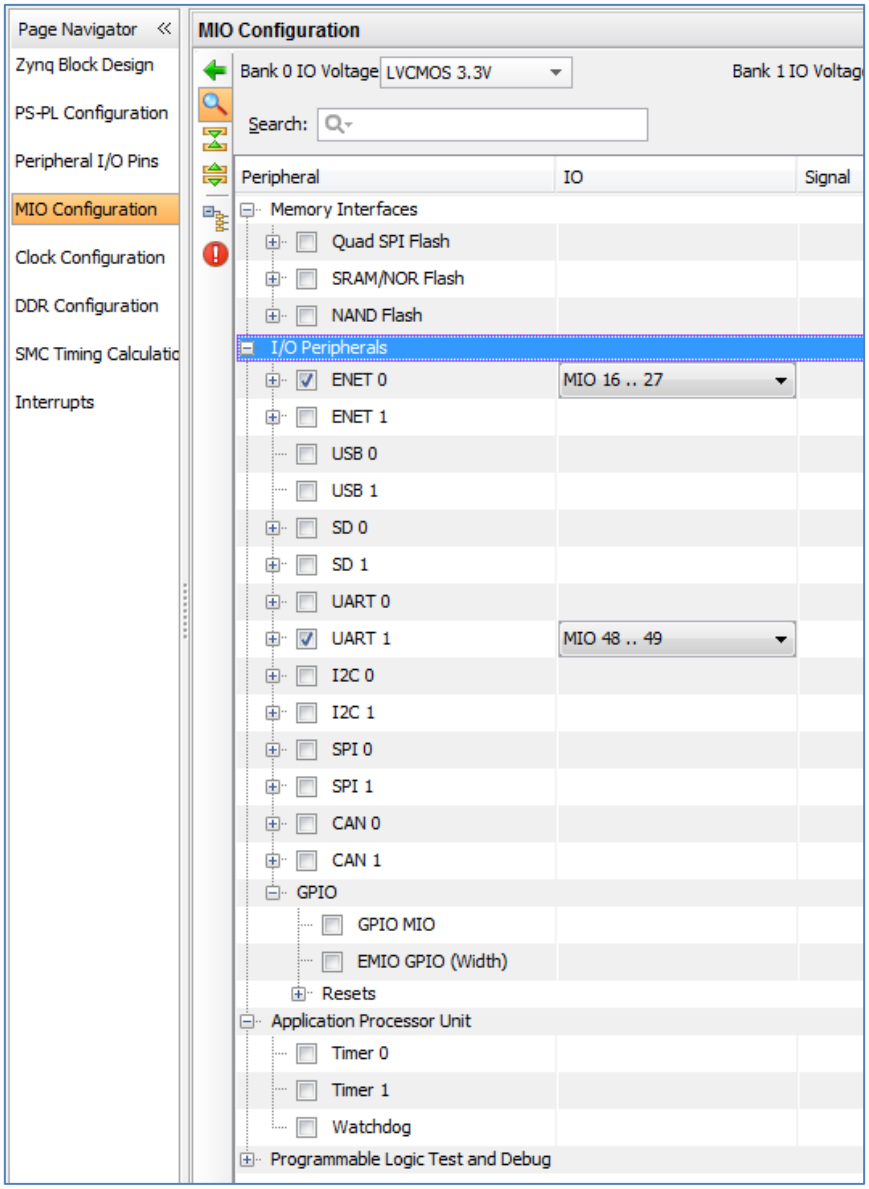

Figura 3.7: Selección de dispositivos de la PS

1.15. Como no vamos a incorporar ningún bloque en la FPGA (PL), podemos deshabilitar la interfaz AXI para la interconexión entre PS y PL. En **PS-PL Configuration**, y dentro de **GP Master AXI Interface**, desactivamos la opción **M AXI GP0 interface**. Desmarcamos también la opción **FCLK_RESET0_N** en **General->Enable Clock Resets**.

1.16. Por último, en **Clock Configuration**, expandimos **PL Fabric Clocks** y desactivamos **FCLK_CLK0**, ya que no incluiremos lógica en la PL.

1.17. Pulsamos **OK** y el bloque se reconfigurará. Varios puertos desaparecerán, ya que hemos desactivado las características que los controlan. Así, obtendremos un sistema totalmente básico con las funcionalidades que vamos a utilizar en nuestras pruebas.

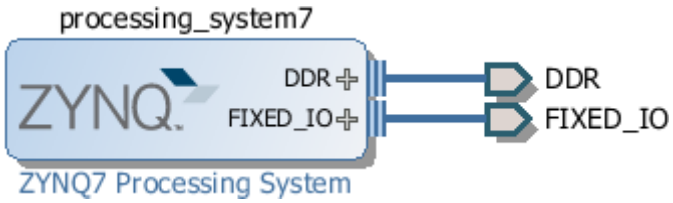

Figura 3.8: Aspecto final del bloque Zynq

Nota: En cualquier momento es posible realizar la verificación del diseño pulsando **F6** o a través de **Tools->Validate Design**.

4. Creación del top-level y exportación al SDK

1.18. En el panel **Sources**, hacemos clic derecho sobre el elemento *system.bd* para desplegar un menú, en el cual escogeremos la opción **Generate Output Products…**. Pulsamos **Generate** en la nueva ventana para generar los archivos de implementación, simulación y síntesis del proyecto.

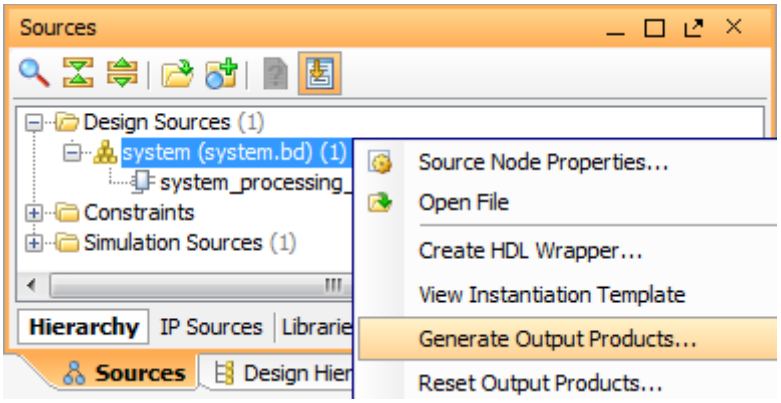

Figura 3.9: Generación de archivos de implementación, simulación y síntesis

1.19. Hacemos de nuevo clic derecho sobre *system.bd* y seleccionamos la opción **Create HDL Wrapper…** para generar el modelo en VHDL del top-level del sistema. En la ventana que aparecerá, se selecciona **Let Vivado manage wrapper and auto-update** para que Vivado realice el proceso automáticamente.

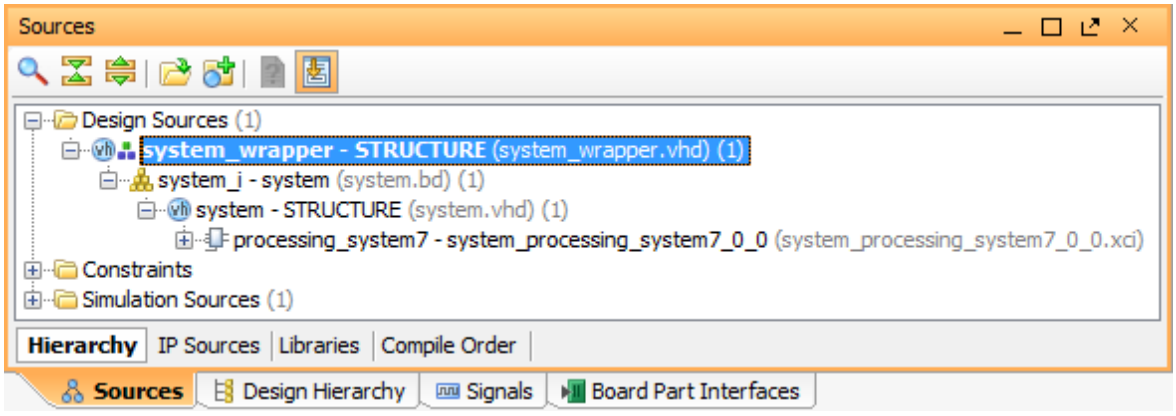

Figura 3.10: Aspecto final de la jerarquía de archivos del diseño

1.20. Por último, exportaremos el proyecto al SDK para poder programar aplicaciones sobre el hardware diseñado. Antes de hacerlo, es necesario que el **Block Design** esté abierto (Si

no lo está, pulsamos sobre **Open Block Design** bajo el grupo **IP Integrator** en el **Flow Navigator**). Para exportar el diseño, pulsaremos **File->Export Hardware**. En este caso, la casilla *Include bitstream* del diálogo que aparecerá estará deshabilitada, ya que no hemos producido un archivo bitstream al no haber incluido lógica en la PL.

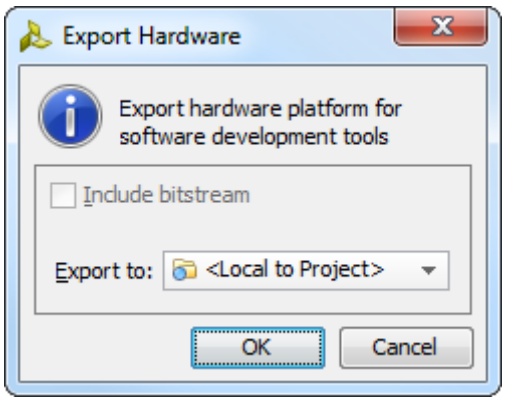

Figura 3.11: Exportación del diseño al SDK

Nota: En proyectos en los que se incluyan elementos en la PL o FPGA, será necesario realizar la síntesis e implementación del diseño antes de exportarlo al SDK.

1.21. Hacemos clic en **File->Launch SDK** para abrir el entorno de desarrollo Xilinx SDK. Una vez abierto, podremos cerrar la aplicación Vivado.

3.4.2. Desarrollo del software: Xilinx SDK

Una vez exportado un proyecto desde Vivado al SDK de Xilinx, contaremos con una descripción de la plataforma diseñada en el explorador de proyectos. El proceso, a partir de ella, será crear un *Board Support Package* (BSP) que contendrá las librerías y controladores de dispositivos para la plataforma y, a continuación, se creará un nuevo proyecto sobre el que desarrollaremos el software que ejecutaremos en la placa.

1. **Hardware Platform y creación del Board Support Package**

1.1. En **Project Explorer** veremos un único elemento, llamado en nuestro caso *system_wrapper_hw_platform_0*. Se trata de una descripción de la plataforma diseñada en Vivado. Abriendo el archivo *system.hdf* se verá la lista de dispositivos que la componen junto a las direcciones de memoria asignadas a cada uno de ellos. En el caso actual, encontraremos entre ellos los elementos *ps7_ethernet_0* y *ps7_uart_1*, referentes a un dispositivo Ethernet y a otro de comunicación serie.

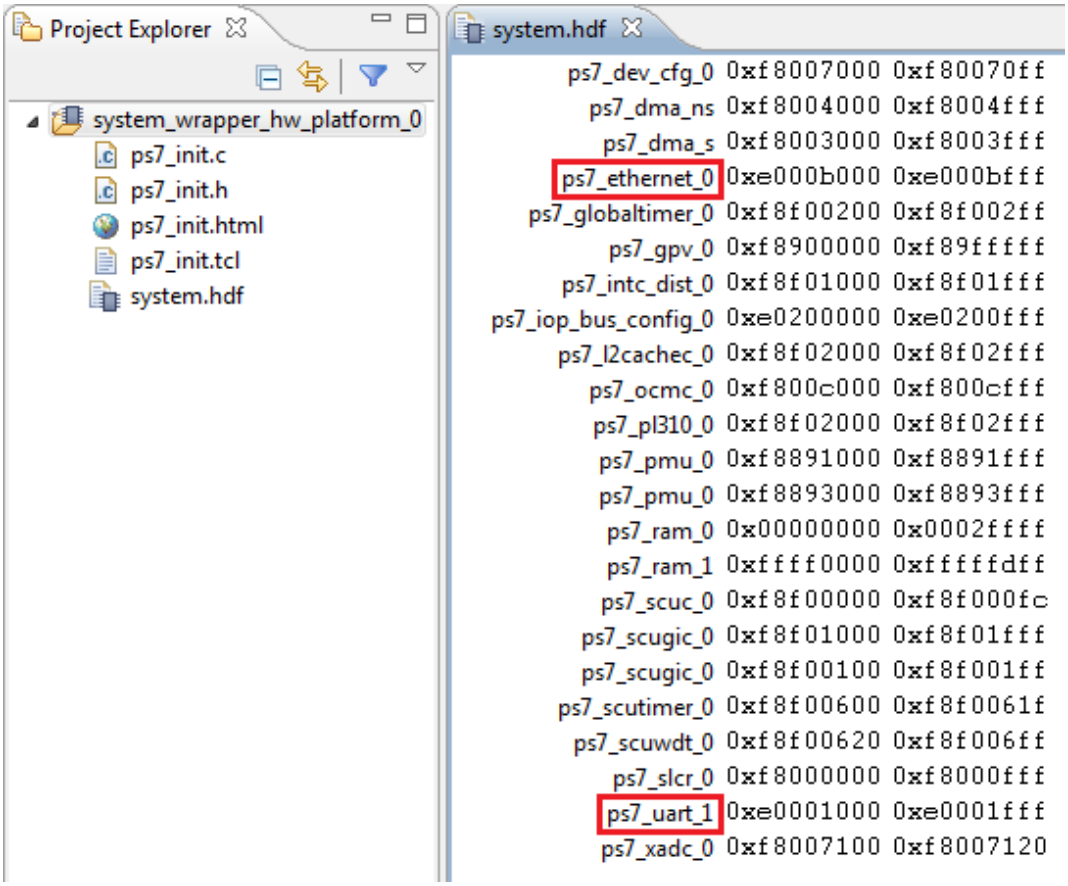

Figura 3.12: Descripción de la plataforma con los periféricos que incluye (Ethernet y UART resaltados)

1.2. El siguiente paso será crear un **Board Support Package**, que incluirá todas las librerías y controladores necesarios para describir y controlar la plataforma descrita en *system_wrapper_hw_platform_0*. Seleccionamos **File->New->Board Support Package** para abrir el formulario de creación.

1.3. En nuestro caso, el *Board Support Package* se llamará *standalone_bsp*. En **Target Hardware** se selecciona la plataforma hardware para la que se crea el BSP (*system_wrapper_hw_platform_0*) y el núcleo que describe. En **Board Support Package OS** nos aseguraremos de seleccionar *standalone*, ya que con este BSP crearemos aplicaciones que se ejecuten nativamente en el procesador. Clic en **Next**.

1.4. Se abrirá una ventana para configurar el BSP. En ella se podrá, entre otras cosas, incluir librerías adicionales con funcionalidades específicas (**Supported Libraries**, en **Overview**) o modificar los drivers que controlan cada dispositivo de la plataforma seleccionada (**Overview->drivers**). En este último caso, por ejemplo, podemos ver que el componente *ps7_ethernet_0* tiene asignado el driver *emacps*, que es la librería controladora de Ethernet. Por el momento no se modificará nada, aunque podrá hacerse cuando se quiera más adelante. Clic en **OK** para generar el BSP.

1.5. Una vez creado, el archivo *system.mms* mostrará una lista de los periféricos de la plataforma y de sus drivers asignados. En el caso de los drivers específicos, se mostrará también un vínculo a la documentación y otro a los ejemplos de dicho controlador, lo que será fundamental para aprender a trabajar con nuevos periféricos.

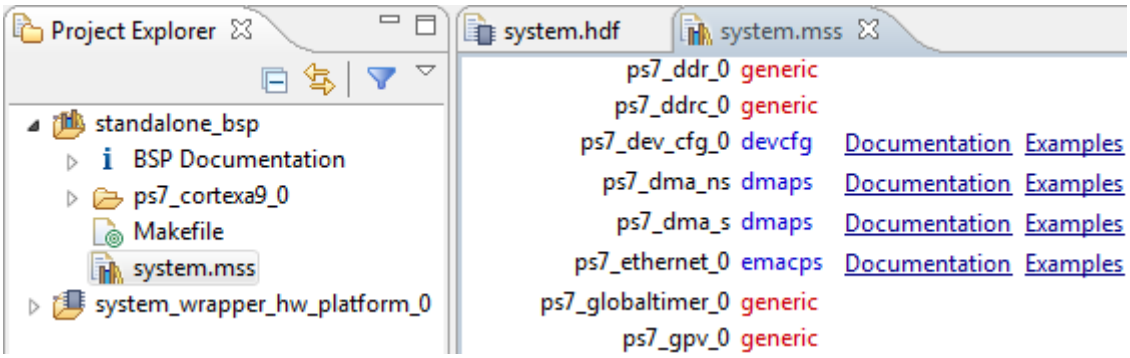

Figura 3.13: Listado de periféricos del BSP

2. Creación de un proyecto para una nueva aplicación

1.6. Para crear un proyecto de aplicación para la plataforma diseñada seleccionaremos el menú **File->New->Application Project**.

1.7. En el formulario de creación del proyecto, introduciremos en este caso el nombre *ethernet_test* para desarrollar una aplicación que maneje el controlador Ethernet. En **Target Hardware** seleccionaremos la misma plataforma que la que utilizamos para generar el BSP. En **Target Software** escogeremos el lenguaje de programación (C o C++), el sistema operativo (si queremos una aplicación que se ejecute en Standalone o sobre un SO Linux) y el *Board Support Package* al que se vinculará la aplicación (solo en el caso de una aplicación Standalone, ya que en Linux no es necesario emplear un BSP debido a que se utiliza sus propias librerías y recursos). Comenzaremos con un proyecto Standalone. Clic en **Next**.

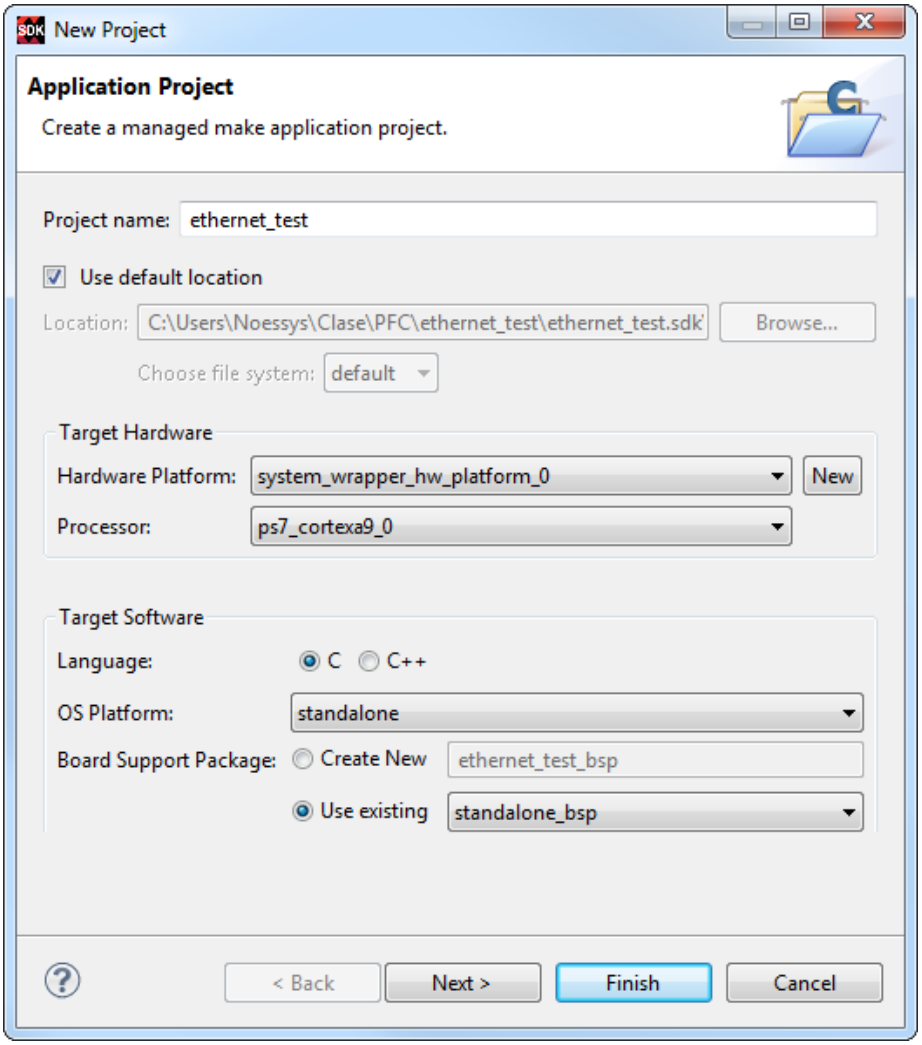

Figura 3.14: Asistente de creación de un nuevo proyecto

1.8. Por último, podremos escoger si queremos empezar con una plantilla de proyecto existente. Es útil para comenzar a familiarizarse con el proceso de desarrollo y comprobar y comprender el funcionamiento de ciertas librerías. En nuestro caso, seleccionamos **Empty Application** para generar un proyecto vacío del que partiremos de cero. Clic en **Finish**.

1.9. En el directorio *src* dentro del proyecto creado se incluirán los archivos de código de la aplicación.

3. Configuración y ejecución de un programa Standalone

1.10. Incluiremos, a modo de prueba, un archivo *hello_world.c* al proyecto, dentro del directorio **src**. Tras una compilación correcta, ejecutaremos el programa.

```
/***  hello_world.c  ***/

#include <stdio.h>
#include "xil_cache.h"

int main() {
        xil_printf("Hello World!\n");
        return 0;
}
```

Código 3.1: Ejemplo "Hello World" para una aplicación Standalone

1.11. Se conecta la Zedboard a través de los cables USB correspondientes al puerto serie y al puerto JTAG. Se enciende la placa. Tanto ella como los puertos deberían ser reconocidos automáticamente como dispositivos.

1.12. Si contáramos con un archivo *.bit* en nuestro proyecto, el paso previo a la ejecución sería la programación en la FPGA. Para ello, clic en **Xilinx Tools->Program FPGA**. El archivo *.bit* debería ser reconocido automáticamente a partir del *hw_platform*. Clic en **Program** y la FPGA quedará programada.

1.13. Para ejecutar el programa, se hace clic derecho sobre el proyecto (en este caso *ethernet_test*) en el explorador de proyectos y, en el menú desplegable, se selecciona **Run As->Launch on Hardware (GDB)**. El programa se ejecutará en la placa.

1.14. Si es la primera vez que se ejecuta un programa, es probable que no se reciba información de la placa por la consola, y se mostrará un aviso en ella (*Process STDIO not connected to console*). Para solucionarlo, clic en **Run->RunConfigurations…** (o **Debug Configurations…)** y, en la pestaña **STDIO Connection** de la ventana que se mostrará, configurar la conexión serie. Clic en **Apply**.

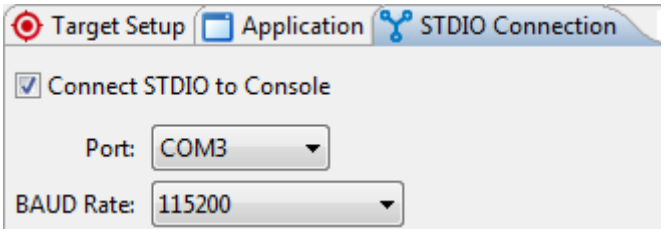

Figura 3.15: Configuración de la comunicación serie

1.15. Ejecutando nuevamente el programa deberá mostrarse la información correcta por la consola.

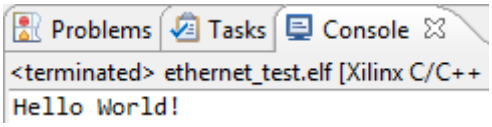

Figura 3.16: Resultado en consola de la ejecución del programa

3.5. Standalone

En primer lugar se mostrará el uso del dispositivo Ethernet ejecutando una aplicación Standalone sobre la plataforma Zynq. Para ello será necesario estudiar el funcionamiento del driver Ethernet desarrollado por Xilinx, denominado *XEmacPs*.

Se programará una aplicación capaz de transmitir y recibir paquetes Ethernet y se realizarán pruebas de transmisión de archivos entre un PC y la placa con el fin de descubrir la velocidad que se puede alcanzar en esta comunicación punto a punto, haciendo uso en ambos extremos del protocolo descrito en el apartado 3.3.

3.5.1. Controlador del dispositivo Ethernet (Driver XEmacPs)

En la plataforma Zynq, la librería controladora del dispositivo Ethernet se denomina *XEmacPs*. Ésta se encarga de la transmisión y recepción de tramas Ethernet, así como de la configuración y del control del dispositivo. De éste, podemos destacar las siguientes características:

- DMA como sistema de acceso a los datos en transmisión y recepción.
- Tamaño máximo de tramas de 1536 bytes.
- Soporte para memoria virtual
- Modo de operación full y half dúplex.
- Cálculo de checksum por hardware.

La principal característica del funcionamiento de este driver es la del uso de acceso directo a memoria (DMA) como sistema único para la transmisión y recepción de datos. Esto difiere con el mecanismo tradicional de las librerías para el control de dispositivos Ethernet, que consiste en especificar directamente buffers de memoria con el contenido de cada trama a enviar o en donde se almacenarán las tramas recibidas. Si bien el mecanismo con DMA promete ser más eficiente, el funcionamiento del driver y su utilización es considerablemente más complejo, ya que son necesarias tareas como preparar las regiones de memoria para transmisión y recepción durante la inicialización del dispositivo, realizar operaciones con la memoria caché y utilizar el sistema de *Buffer Descriptors*.

En el controlador XEmacPs, el elemento fundamental es una instancia de una estructura de datos llamada, también, ***XEmacPs***, que hará referencia al dispositivo Ethernet utilizado y

que almacenará todos los datos sobre él (identificador del dispositivo Ethernet y del controlador de interrupciones, opciones de configuración, dirección MAC, velocidad de operación, punteros a las funciones manejadoras de eventos en transmisión y recepción, etc).

Los ***Buffer Descriptors*** (o BDs) son estructuras de datos que describen un paquete Ethernet completo o parcial. Para ello, cuentan con una dirección de memoria que especifica la región de los datos a enviar o en la que se almacenarán los datos recibidos, así como de otra serie de parámetros (longitud de trama, estado de transmisión, fin de trama, etc).

El motor DMA accede a las regiones de memoria especificadas por los BDs durante el proceso de transmisión y recepción. Sin embargo, y como es lógico, los BDs deben estar ordenados. Para ello existen otras estructuras de datos denominadas ***Buffer Descriptor Rings*** o BdRings.

Los ***BdRings*** consisten en una serie de BDs añadidos a ellos y que han sido parcialmente inicializados. Podrían describirse como listas ordenadas de BDs que el motor DMA va siguiendo para acceder a ellos en el orden correcto.

Existe un BdRing para transmisión y otro para recepción, y son creados y asignados a la instancia XEmacPs durante la inicialización del dispositivo. A cada uno de ellos se asigna una región de memoria en la que se almacenarán los BDs y que ha sido reservada previamente en memoria (*uncached*), y se establece el número máximo de BDs que podrá contener.

Los BDs incluidos en cada BdRing están, como se ha comentado, parcialmente inicializados desde su inclusión, conteniendo datos de configuración. Los restantes datos deberán ser suministrados en cada proceso de preparación de los BDs previa a una transmisión o recepción, por lo que cuantos más campos sean configurados durante la inicialización, a menos será necesario acceder durante este proceso. El mejor caso de inicialización requerirá que el usuario establezca únicamente una dirección de memoria para el paquete y la longitud de éste antes de que el hardware inicie las operaciones.

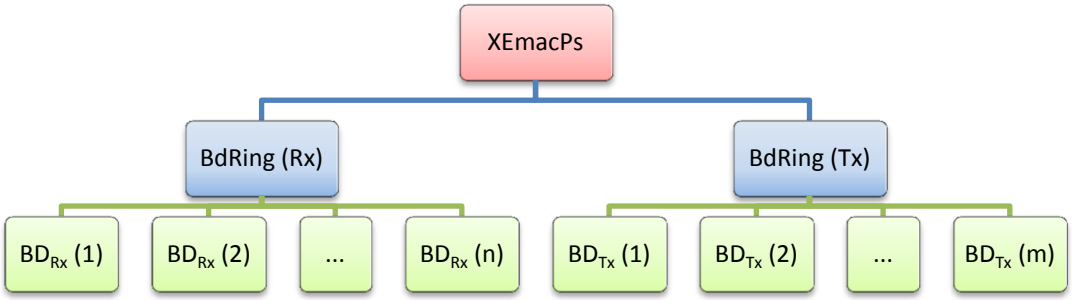

Figura 3.17: Esquema de la configuración de elementos en el driver XEmacPs

El proceso seguido para el empleo del driver XEmacPs en una aplicación estándar para transmisión y recepción de paquetes puede dividirse en fases. La siguiente tabla muestra de forma superficial estas fases que debe seguir una aplicación, así como las funciones del driver empleadas en ellas. Por supuesto, el empleo de todas las funciones que aparecen puede no ser necesario, y otras tantas que no figuran en la tabla pueden ser utilizadas cuando se requiere profundizar más en la configuración del dispositivo.

Fases	Procesos	Funciones empleadas

Inicialización	Configuración de la instancia XEmacPs	`XEmacPs_LookupConfig()`
		`XEmacPs_CfgInitialize()`
		`XEmacPs_SetMacAddress()`
		`XEmacPs_SetMdioDivisor()`
		`XEmacPs_SetOperatingSpeed()`

	Reserva de memoria y creación de BdRings	`Xil_SetTlbAttributes()`
		`XEmacPs_BdClear()`
		`XEmacPs_BdRingCreate()`
		`XEmacPs_BdRingClone()`

	Establecimiento de manejadores para transmisión, recepción y errores	`XEmacPs_SetHandler()`

	Configuración y habilitación de interrupciones (para los manejadores)	`XScuGic_LookupConfig()`
		`XScuGic_CfgInitialize()`
		`Xil_ExceptionRegisterHandler()`
		`XScuGic_Connect()`
		`XScuGic_Enable()`

Habilitación (Start)	Inicio del dispositivo	`XEmacPs_Start()`
Operación	Transmisión y recepción	
Inhabilitación (Stop)	Detención del dispositivo	`XEmacPs_Stop()`

Tabla 3.1: Proceso básico de trabajo con el driver XEmacPs

En la tabla anterior se ha omitido la descripción del proceso de transmisión y recepción de paquetes, ya que se ha creído más conveniente explicarlo de forma independiente.

La preparación para la transmisión o recepción de un paquete Ethernet es muy similar, aunque, como es natural, difieren en algunos puntos. En ambos casos, el proceso se divide en dos fases:

- Preparación de los BDs y entrega de éstos al dispositivo Ethernet para iniciar el proceso de transferencia.
- Recuperación de los BDs tras el envío o recepción. Esto puede realizarse en una subrutina cualquiera, pero lo más habitual y lógico es realizarlo en una rutina manejadora de interrupciones que es disparada con cada envío o recepción.

A continuación se muestra una tabla en la que se describe de forma resumida los pasos básicos a seguir para la transmisión de uno o más paquetes de datos, así como las funciones de la librería XEmacPs empleadas.

Fases	Procesos	Funciones empleadas
Transmisión	Reserva de espacio en el BdRing para los BDs que se incluirán	`XEmacPs_BdRingAlloc()`

	Preparación de los BDs (datos, longitud, fin de paquete, etc)	`Xil_DCacheFlushRange()`
		`XEmacPs_BdSetAddressTx()`
		`XEmacPs_BdSetLength()`
		`XEmacPs_BdClearTxUsed()`
		`XEmacPs_BdSetLast()`
		`XEmacPs_BdRingNext()`

	Se encolan los BDs del BdRing en el dispositivo Ethernet	`XEmacPs_BdRingToHw()`
	Se transmiten los BDs	`XEmacPs_Transmit()`
Manejador de interrupciones (Tras la transmisión de cada BD)	Se recuperan los BDs enviados	`XEmacPs_BdRingFromHwTx()`
	Se recorren los BDs enviados y se vuelven a habilitar para futuras transmisiones	`XEmacPs_BdClearTxUsed()` `XEmacPs_BdRingNext()`
	Se liberan del BdRing los BDs enviados	`XEmacPs_BdRingFree()`

Tabla 3.2: Fase de transmisión con el driver XEmacPs

Al igual que con la transmisión, se muestra a continuación una tabla con los pasos necesarios para la recepción de uno o más paquetes Ethernet.

Fases	Procesos	Funciones empleadas
Recepción	Reserva de espacio en el BdRing para los BDs que se incluirán	`XEmacPs_BdRingAlloc()`
	Preparación de los BDs (datos, longitud, fin de paquete, etc)	`Xil_DCacheInvalidateRange()` `XEmacPs_BdSetAddressRx()`
	Se encolan los BDs del BdRing en el dispositivo Ethernet	`XEmacPs_BdRingToHw()`
Manejador de interrupciones (Tras la recepción de cada BD)	Se recuperan los BDs recibidos	`XEmacPs_BdRingFromHwRx()`
	Se recorren los BDs recibidos y se vuelven a habilitar para futuras transmisiones	`XEmacPs_BdClearRxNew()` `XEmacPs_BdRingNext()`
	Se recupera el paquete recibido	`XEmacPs_BdGetBufAddr()`
	Se liberan del BdRing los BDs recibidos	`XEmacPs_BdRingFree()`

Tabla 3.3: Fase de recepción con el driver XEmacPs

Cachés

En el proceso de utilización del driver XEmacPs, especialmente cuando se está trabajando con BDs, será necesario el uso de las funciones `Xil_DCacheFlushRange()` y `Xil_DCacheInvalidateRange()`. Estas funciones están relacionadas con la memoria caché, y realizan funciones necesarias para que el sistema de DMA funcione correctamente al enviar o recibir paquetes de datos.

La función `Xil_DCacheFlushRange()` realiza una operación *flush* en un rango especificado de memoria "cacheada". Eso significa que ese contenido de la memoria en caché es escrito de vuelta a la memoria principal. Esta operación es necesaria en el proceso de

transmisión de paquetes: si los datos a transmitir están almacenados en memoria “cacheada”, es necesaria una operación *flush* para descargar estos datos a la memoria principal antes de pasarlos al hardware.

La función `Xil_DCacheInvalidateRange()` invalida el rango especificado en la memoria caché, de modo que la siguiente lectura tendrá lugar en memoria principal sobre los datos que existieran en ella. Esta operación es necesaria en la recepción de paquetes: si el buffer de datos está almacenados en memoria “cacheada”, ésta debe ser invalidada antes de que el sistema acceda a los datos.

3.5.2. Librería implementada

Debido a la complejidad del driver XEmacPs o a la falta de claridad que pudiera originar la escritura de aplicaciones utilizando de forma directa sus funciones, se creyó conveniente el desarrollo de una librería que sirviese de intermediaria entre el usuario y las funciones del driver.

Esta librería ofrecería funciones básicas para el funcionamiento del dispositivo Ethernet y para los procesos de transmisión y recepción de tramas, trabajando de forma transparente al funcionamiento del driver XEmacPs y minimizando, en la medida de lo posible, el paso de parámetros a las funciones. Se trataría, en definitiva, de una interfaz simple pero totalmente funcional que agilizaría considerablemente el desarrollo de aplicaciones que hicieran uso de un dispositivo Ethernet.

Los procesos de inicialización y configuración del dispositivo, algo tediosos utilizando las funciones de driver, se realizarían automáticamente llamando solo a una o dos funciones, al igual que sucedería con la transmisión y recepción de paquetes. Como es obvio, esta abstracción del funcionamiento del driver y la simplificación introducida repercutiría en el nivel de configuración del dispositivo, ya que la inicialización de éste se realizaría con parámetros por defecto o que garantizasen un buen funcionamiento bajo unas condiciones de uso tradicionales. Sin embargo, es un tema que se trataría de resolver.

La librería desarrollada se denominó ***EthZynq***, y trata de simplificar el uso del driver XEmacPs y ofrecer una interfaz sencilla y similar a otras como libpcap o WinPcap, ofreciendo así ventajas a la hora de portar código entre diferentes plataformas. Está desarrollado en lenguaje C++ con el fin de poder trabajar con estructuras dinámicas de datos.

El siguiente cuadro presenta la declaración de las funciones que componen la librería.

```
/****************************** Prototipo de funciones *******************************/

/* Control del dispositivo */
int  Eth_Initialize(u16 EmacPsDeviceId, u16 EmacPsIntrId, u16 IntrDeviceId, void *MAC_AddressPtr);
void Eth_Start();
void Eth_Stop();
int  Eth_Reset();
XEmacPs* Eth_GetInstance();

/* Transmisión */
void Eth_SendQueue_Alloc(u32 packetsNum);
void Eth_SendQueue_Destroy();
int  Eth_SendQueue_Queue(u16 packetLength, u32 packetAddress);
int  Eth_SendQueue_Transmit();
int  Eth_Send(u16 packetLength, u32 packetAddress);

/* Recepción */
int  Eth_Receive_Setup();
int  Eth_Receive(unsigned char* &buffer);

/* Configuración de interrupciones */
int  Eth_SetHandler(u32 HandlerType, void *HandlerFunctionPtr);
int  Eth_Interruptions_Enable();
void Eth_Interruptions_Disable();

/* Manejo de interrupciones */
void Eth_SendHandler(void *Callback);
void Eth_RecvHandler(void *Callback);
void Eth_ErrorHandler(void *Callback, u8 direction, u32 word);

/*************************************************************************************/
```

Código 3.2: Prototipo de las funciones de la librería EthZynq desarrollada

A continuación se describen las funciones que componen la librería, con el fin de tratar de explicar su funcionamiento y de facilitar la introducción de modificaciones.

Control del dispositivo

Las funciones para el control principal del dispositivo Ethernet son cuatro:

- `Eth_Initialize()`: Inicializa el dispositivo con una configuración básica. Establece las opciones y las variables de control, crea los BdRings y registra los manejadores de interrupciones. Esta función pide como argumentos todos los valores que necesita para la inicialización: el identificador del dispositivo Ethernet, el identificador de interrupciones, el identificador del controlador de interrupciones y la dirección MAC. El valor de los tres primeros parámetros vienen dados por la plataforma y se obtiene de los archivos de definiciones del BSP: *`xparameters.h`* y *`xparameters_ps.h`*.
- `Eth_Start()`: Inicia el controlador Ethernet, permitiendo la transmisión y recepción de paquetes de datos.
- `Eth_Stop()`: Detiene el controlador Ethernet.
- `Eth_Reset()`: Restablece todos los valores iniciales del controlador Ethernet (opciones, manejadores, BdRings…) y lo reinicia.
- `XEmacPs* Eth_GetInstance()`: Devuelve un puntero a la instancia del controlador XEmacPs utilizada por la librería, de forma que el usuario pueda realizar a través de él modificaciones en el controlador que no estén implementadas por el resto de funciones de la librería.

En una aplicación debe llamarse en primer lugar a `Eth_Initialize()`. Si el controlador se ha configurado correctamente, se llamará entonces a `Eth_Start()` para iniciar el proceso de transmisión y recepción de paquetes. Por último, con `Eth_Stop()` se detendrá este proceso una vez que se desee.

Función	Descripción	Argumentos / Return	
`int Eth_Initialize()`	Inicializa y configura el dispositivo Ethernet (ETH0).	`u16 EmacPsDeviceId`	Identificador del dispositivo Ethernet.
		`u16 EmacPsIntrId`	Identificador de interrupciones del dispositivo Ethernet.
		`u16 IntrDeviceId`	Identificador del controlador de interrupciones.
		`void* MAC_AddressPtr`	Array de tipo char con la dirección MAC del dispositivo.
		Return: XST_SUCCESS si todo ha ido bien, XST_FAILURE en caso de fallo.	
`void Eth_Start()`	Inicia el dispositivo Ethernet para habilitar la transmisión y recepción de paquetes.		
`void Eth_Stop()`	Detiene el dispositivo Ethernet.		
`int Eth_Reset()`	Restablece y reinicia el dispositivo Ethernet.	**Return**: XST_SUCCESS si todo ha ido bien, XST_FAILURE en caso de fallo.	
`XEmacPs* Eth_GetInstance()`	Devuelve un puntero a la instancia XEmacPs del dispositivo.	**Return**: `Puntero a la instancia XEmacPs del controlador usado por la librería.`	

Tabla 3.4: Funciones de control del dispositivo de la librería EthZynq

Transmisión

Existen dos procesos implementados para la transmisión de paquetes Ethernet:

- Directa: la función `Eth_Send()` envía una única trama de datos, indicando la dirección de memoria de los datos y la longitud del paquete.
- Colas de transmisión: Permite alcanzar velocidades más elevadas que mediante la transmisión independiente de tramas. Para ello se hace uso de la capacidad del BdRing de transmisión para contener tramas y se va rellenando a medida que los paquetes van siendo enviados. De esta manera se obtiene una transmisión constante de las tramas. El proceso a seguir para su uso es el siguiente:
    1. Reservar espacio en la cola de transmisión con la función `Eth_SendQueue_Alloc()`, indicando el número máximo de paquetes que contendrá.
    2. Poner en cola cada paquete que se desee transmitir mediante `Eth_SendQueue_Queue()`. Para ello, se especifica la dirección de memoria de los datos y la longitud de la trama.
    3. Transmitir los paquetes registrados en la cola de transmisión mediante `Eth_SendQueue_Transmit()`.
    4. Por último, se libera la memoria utilizada por la cola de transmisión con `Eth_SendQueue_Destroy()`.

Función	Descripción	Argumentos / Return	
`void Eth_SendQueue_Alloc()`	Reserva espacio para la cola de transmisión.	`u32 packetsNum`	Número de paquetes de la cola de transmisión.
`void Eth_SendQueue_Destroy()`	Elimina la cola de transmisión y libera el espacio.		
`int Eth_SendQueue_Queue()`	Añade un paquete a la cola de transmisión.	`u16 packetLength`	Longitud del paquete.
		`u32 packetAddress`	Dirección del buffer de memoria del paquete.
		Return: 0 si ha ido bien, -1 si se sobrepasa el tamaño de la cola.	
`int Eth_SendQueue_Transmit()`	Envía la cola de transmisión.	**Return**: XST_SUCCESS si todo ha ido bien, XST_FAILURE en caso de fallo.	
`int Eth_Send()`	Envía una trama especificada.	`u16 packetLength`	Longitud del paquete.
		`u32 packetAddress`	Dirección del buffer de memoria del paquete.
		Return: XST_SUCCESS si todo ha ido bien, XST_FAILURE en caso de fallo.	

Tabla 3.5: Funciones de transmisión de la librería EthZynq

Recepción

La función `Eth_Receive()` devuelve el último paquete recibido. Si no hay ningún paquete pendiente para ser devuelto, la función espera a recibir uno. Como argumento se pasa un puntero que la función modificará, haciéndolo apuntar a la dirección de memoria del paquete recibido, y devuelve como valor de retorno la longitud de éste.

Por su parte, la función `Eth_Receive_Setup()` se encarga de preparar los BDs del BdRing de recepción, que contendrán los datos recibidos en una lista enlazada compuesta por estructuras de datos (`struct packet`) que describen cada paquete (trama y longitud). De esta forma, los paquetes recibidos son almacenados de forma ordenada en esta lista, y son eliminados cuando se devuelven al usuario mediante `Eth_Receive()`.

Nota: La función Eth_Receive_Setup() se utiliza para el funcionamiento interno de la librería, por lo que no debe ser llamada por el usuario.

Función	Descripción	Argumentos / Return	
`int Eth_Receive_Setup()`	Prepara los BDs en el BdRing de recepción y la lista de paquetes recibidos.	**Return**: XST_SUCCESS si todo ha ido bien, XST_FAILURE en caso de fallo.	
`int Eth_Receive()`	Añade un paquete a la cola de transmisión.	`unsigned char* &buffer`	Buffer de recepción.
		Return: Longitud de la trama recibida.	

Tabla 3.6: Funciones de recepción de la librería EthZynq

Configuración de interrupciones

Nota: Las funciones de esta sección se utilizan para el funcionamiento interno de la librería, por lo que no deben ser llamadas por el usuario.

Las siguientes funciones se encargan de temas referentes a la configuración de las interrupciones del controlador Ethernet:

- `Eth_SetHandler()`: Establece una función que será el manejador de las interrupciones producidas (del tipo especificado).
- `Eth_Interruptions_Enable()`: Configura y habilita las interrupciones del controlador Ethernet.
- `Eth_Interruptions_Disable()`: Inhabilita las interrupciones del dispositivo Ethernet.

Función	Descripción	Argumentos / Return	
`void` **`Eth_SetHandler()`**	Establece una función para el manejador de interrupciones especificado.	`u32` **`HandlerType`**	Tipo del manejador (SEND, RECV, ERROR).
		`void*` **`HandlerFunctionPtr`**	Dirección de la función que manejará la interrupción.
		Return: 0 si ha ido bien, -1 si se sobrepasa el tamaño de la cola.	
`void` **`Eth_Interruptions_Enable()`**	Habilita las interrupciones del dispositivo Ethernet.	**Return**: `XST_SUCCESS` si todo ha ido bien, `XST_FAILURE` en caso de fallo.	
`int` **`Eth_Interruptions_Disable()`**	Inhabilita las interrupciones del dispositivo Ethernet.		

Tabla 3.7: Funciones de configuración de interrupciones de la librería EthZynq

Manejo de interrupciones

Nota: Las funciones de esta sección se utilizan para el funcionamiento interno de la librería, por lo que no deben ser llamadas por el usuario.

Las siguientes funciones son los manejadores de interrupciones del controlador Ethernet:

- `Eth_SendHandler()`: Manejador de interrupciones de transmisión. Se dispara cuando se envía un paquete. Libera los BDs transmitidos y actualiza las estadísticas de transmisión.
- `Eth_RecvHandler()`: Manejador de interrupciones de recepción. Se dispara cuando se recibe un paquete. Gestiona los paquetes recibidos introduciéndolos en la lista de recepción, libera los BDs utilizados, llama a Eth_Receive_Setup() para prepararlos para siguientes paquetes y actualiza las estadísticas de recepción.
- `Eth_ErrorHandler()`: Manejador de interrupciones de errores. Se dispara cuando se produce un error, e indica el tipo y la dirección (transmisión o recepción) de éste.

Función	Descripción	Argumentos / Return	
`void Eth_SendHandler()`	Manejador de interrupciones de transmisión. Se dispara cuando se envía un paquete.	`void* Callback`	Puntero a la instancia del dispositivo EmacPs.
`void Eth_RecvHandler()`	Manejador de interrupciones de recepción. Se dispara cuando se recibe un paquete.	`void* Callback`	Puntero a la instancia del dispositivo EmacPs.
`int Eth_ErrorHandler()`	Manejador de interrupciones de errores. Se dispara cuando se produce un error.	`void* Callback`	Puntero a la instancia del dispositivo EmacPs.
		`u8 direction`	Indica la dirección del error (RECV o SEND).
		`u32 word`	Valor del registro de estado que indica el error.

Tabla 3.8: Funciones de manejo de interrupciones de la librería EthZynq

3.5.3. Resultados

Utilizando la librería desarrollada para implementar un sencillo cliente de transmisión y de recepción se realizaron pruebas de rendimiento del canal Ethernet, empleando como servidor la aplicación desarrollada en Windows.

Las pruebas consistieron en la transmisión y recepción repetitiva de un archivo siguiendo el protocolo expuesto en el apartado 3.3 y en la obtención de las estadísticas de las transferencias. Los parámetros que se observaron más representativos a la hora de caracterizar una transmisión de paquetes Ethernet fueron la velocidad, la longitud de las tramas, la tasa de pérdidas de paquetes y el tamaño de la ventana de transmisión. Por tanto, se realizaron pruebas de transferencia modificando estos parámetros y relacionando los resultados medios obtenidos.

La siguiente gráfica muestra la velocidad máxima de transferencia que ha sido posible obtener en función de la longitud del *payload* o campo de datos de los paquetes.

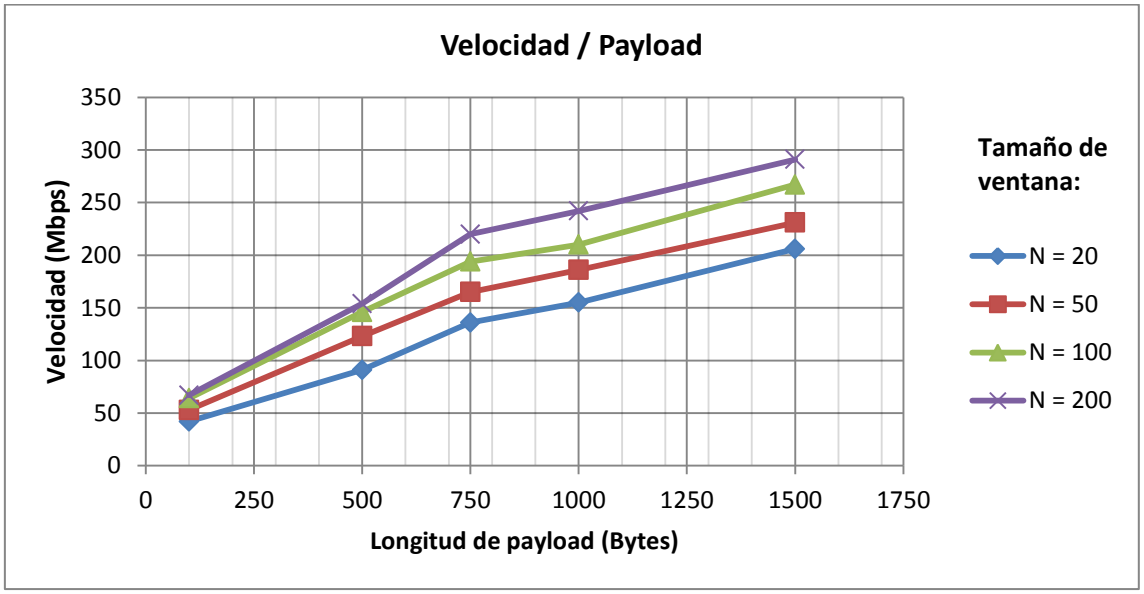

Figura 3.18: Relación entre velocidad alcanzada y longitud de payload (modo Standalone)

Como era de esperar, a mayor longitud de *payload*, mayor será la velocidad obtenida si no se alteran el resto de características de transferencia. Esto es debido a que se consigue un mayor aprovechamiento de cada trama (con respecto al tamaño de la cabecera) y al menor número de paquetes necesarios para completar una transferencia (lo que se traduce también en una cantidad menor de tramas de control).

A continuación se muestra el porcentaje de pérdida de paquetes en función de la velocidad de transmisión deseada. Es importante remarcar que la velocidad indicada es aquella a la que se envían los datos en cada ventana de transmisión, pero no es la velocidad real alcanzada (ésta se mostrará más adelante).

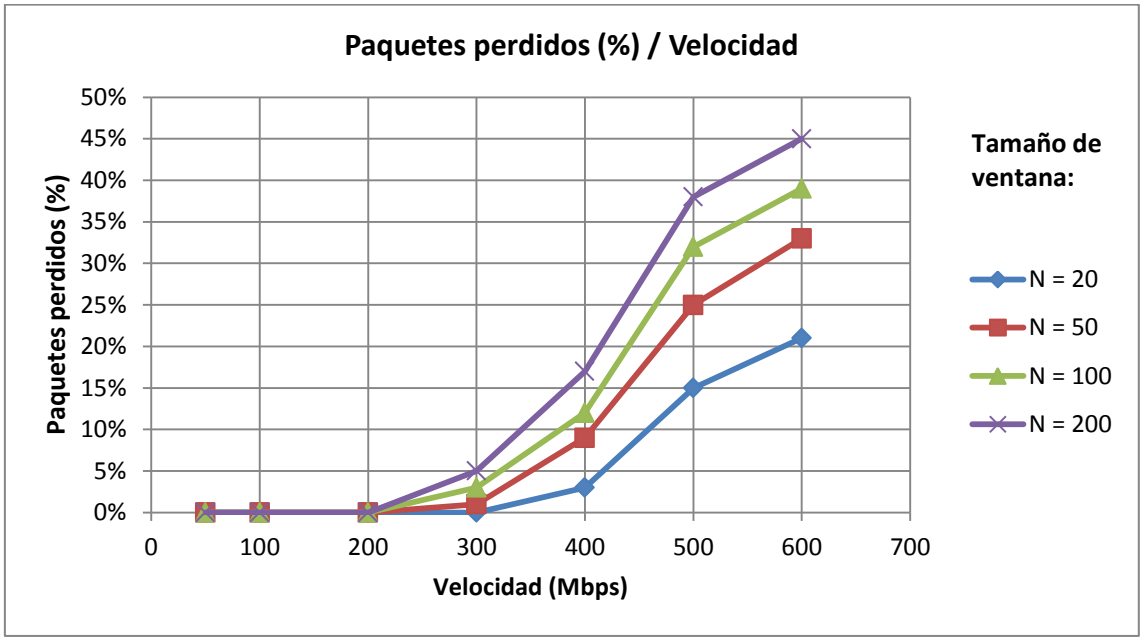

Figura 3.19: Relación entre pérdida de paquetes y velocidad de transmisión (modo Standalone)

Por último, se muestra la velocidad real de transferencia alcanzada para una velocidad específica de transmisión de los paquetes en las ventanas. Teniendo en cuenta la gráfica anterior es evidente que, a pesar de la velocidad de transmisión, las pérdidas de paquetes repercutirán negativamente en la velocidad final de una transferencia, debido a la necesidad de retransmisión de paquetes y al envío redundante de paquetes de control.

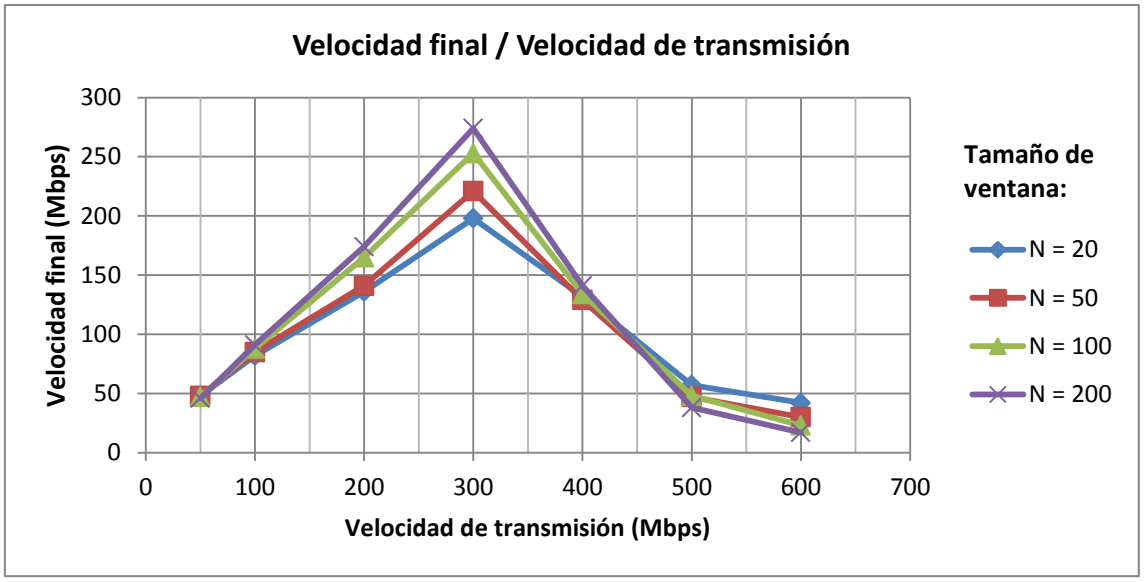

Figura 3.20: Relación entre velocidad final alcanzada y velocidad de transmisión (modo Standalone)

Iperf

Iperf es una herramienta para realizar pruebas de rendimiento de redes Ethernet. Se encuentra disponible para multitud de entornos, entre ellos Linux y Windows.

Utilizando la librería lwIP para plataformas Zynq, que implementa el protocolo TCP/IP y otras características de comunicaciones Ethernet, es posible hacer uso de la herramienta Iperf para obtener un resultado complementario a los anteriores. Si bien estas medidas se realizan utilizando un protocolo completamente diferente al descrito en el apartado 3.3, los resultados no dejan de ser valiosos y muy representativos para la caracterización del medio de comunicación. De hecho, es muy interesante realizar estas medidas, ya que tanto la aplicación Iperf como la librería lwIP son herramientas robustas, fiables y documentadas, por lo que los resultados obtenidos ofrecerán una mejor idea de hasta qué punto puede llegar el rendimiento del canal de comunicaciones Ethernet.

Para realizar estas pruebas se seguirá la *Application Note* de Xilinx sobre los ejemplos de aplicación de la librería lwIP[6] y se utilizarán los archivos ofrecidos para su funcionamiento. Tanto este documento como los archivos necesarios se han incluido en la documentación entregada junto con esta memoria.

Los ejemplos ofrecidos y utilizados son dos: aplicación *RAW*, que controla la comunicación a nivel de transporte; y aplicación *SOCKET*, que utiliza una API para el manejo de conexiones TCP/IP. Ambas se desarrollan en un entorno multiproceso (FreeRTOS), también ofrecido y configurado. Se comenzó utilizando la aplicación *RAW.*

PC como cliente:

```
> iperf -c 192.168.1.100 -t 30 -w 64k
------------------------------------------------------------
Client connecting to 192.168.1.10, TCP port 5001
TCP window size:  128 KByte (WARNING: requested 64.0 KByte)
------------------------------------------------------------
[  3] local 192.168.1.100 port 43822 connected with 192.168.1.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec  2.88 GBytes   825 Mbits/sec
```

Figura 3.21: Resultados de Iperf utilizando un PC como cliente (modo Standalone)

Zedboard como cliente:

```
> iperf.exe -s -w 64k
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size:  128 KByte (WARNING: requested 64.0 KByte)
------------------------------------------------------------
[  4] local 192.168.1.100 port 5001 connected with 192.168.1.10 port 49153
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-30.0 sec  2.91 GBytes   833 Mbits/sec
```

Figura 3.22: Resultados de Iperf utilizando la Zedboard como cliente (modo Standalone)

El mismo procedimiento es utilizado para obtener ambos valores empleando la aplicación *SOCKET*. Los resultados obtenidos fueron los siguientes:

Modo RAW		Modo SOCKET	
Recepción (Mbps)	Transmisión (Mbps)	Recepción (Mbps)	Transmisión (Mbps)
825	833	507	519

Tabla 3.9: Resumen de resultados obtenidos con Iperf (modo Standalone)

3.6. Linux

Existen numerosas distribuciones de Linux adaptadas para trabajar sobre las plataformas Zynq, tanto de pago como de código abierto.

El empleo de un sistema operativo que no sea un RTOS (*Real-Time Operative System*) podría provocar que obtuviéramos peores resultados que utilizando una aplicación Standalone en nuestro interés por contar con un canal Ethernet de comunicación lo más rápido posible, debido a la multitud de procesos y recursos en uso que se dan en un sistema operativo.

No obstante, las ventajas que ofrece la utilización de un sistema Linux lo hacen enormemente interesante para casi cualquier propósito al contar con un sistema de datos persistente, librerías de terceros, acceso más sencillo a los periféricos, sistema de archivos, etc.

Por tanto, a pesar de la premisa de que tal vez supondría un pequeño descenso en el rendimiento de la comunicación Ethernet, merece mucho la pena comprobar su funcionamiento sobre un sistema operativo.

De entre todas las distribuciones disponibles, se escogió **Xillinux**[9], por ser bastante completa y estar bien documentada. Sin embargo, otras distribuciones de Linux no deberían presentar problemas de compatibilidad.

La información sobre la puesta en marcha del sistema operativo Xillinux en la Zedboard se tratará en el apartado 4, por lo que éste solo se centrará en lo referente al medio de comunicación Ethernet.

3.6.1. Ethernet sobre Linux

La integración en Linux del controlador Ethernet es completa y permite su utilización directamente con las librerías propias que ofrece. Por tanto, y considerando la multitud de ejemplos existentes sobre la utilización de un dispositivo Ethernet para comunicación, esta fase del proyecto debería limitarse casi únicamente a la adaptación del algoritmo de transferencia de archivos en una aplicación para Linux.

Para ello es necesario conocer las librerías que ofrece Linux y que permiten trabajar con un controlador Ethernet a bajo nivel para transmitir y recibir paquetes de datos. Esto se puede conseguir con una combinación de funciones de varias librerías de *networking*, tales como la librería de sockets (libsocket) o la X/Open Network Service Library (libxnet).

Estas librerías están ampliamente documentadas y existen una gran variedad de códigos de ejemplo que pueden encontrarse en Internet y otros medios. Debido a esto, la sencillez de su uso y a que existen otras librerías que podrían desempeñar las mismas tareas, no se describirán las funciones utilizadas. El código del programa desarrollado, presente en el material que acompaña a esta memoria, debe ser lo bastante explicativo a través de sus comentarios.

3.6.2. Resultados

De la misma forma que para la caracterización del medio Ethernet en un entorno Standalone (apartado 3.5.3), se implementaron en la aplicación Linux unos clientes sencillos para la transferencia de datos, utilizando nuevamente una aplicación Windows como servidor. Los parámetros a tener en cuenta fueron los mismos (velocidad de transferencia, tamaño de tramas, tasa de pérdida de paquetes y tamaño de la ventana de transmisión), y los resultados medios obtenidos se muestran a continuación.

En primer lugar, se muestra la relación entre la velocidad máxima alcanzada y el tamaño de *payload* empleado por los paquetes de datos:

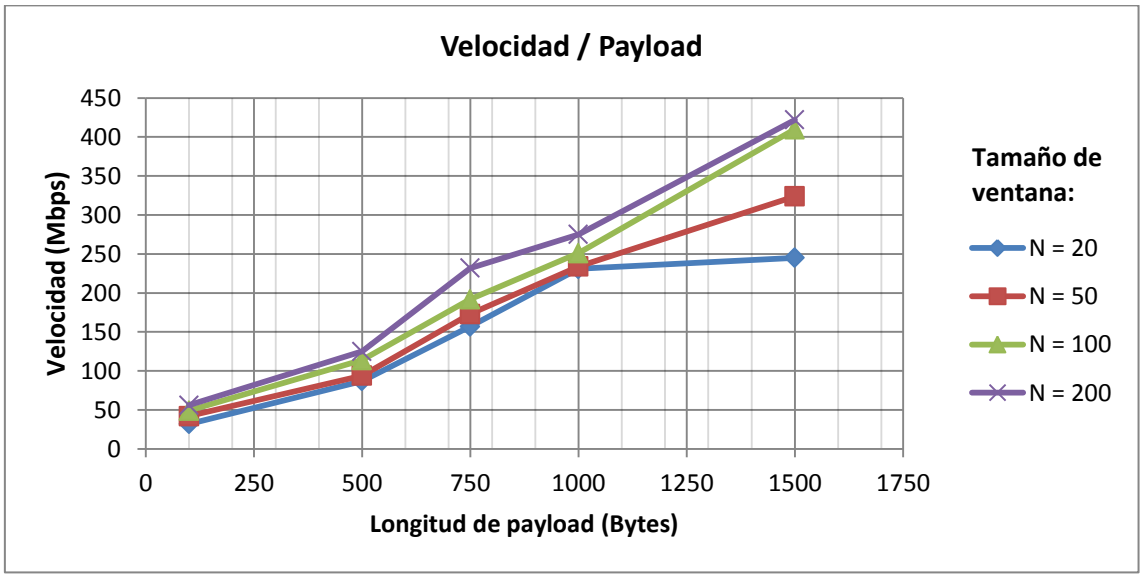

Figura 3.23: Relación entre velocidad alcanzada y longitud de payload (Linux)

A continuación se muestra la relación entre el porcentaje de paquetes perdidos durante la transferencia y la velocidad de transmisión de los paquetes. Como se indicó en el apartado 3.5.3, estas velocidades de transmisión mostradas son aquellas a las que se envían los paquetes de datos en cada ventana, pero no representan la velocidad final de la transferencia, la cual será menor debido a las pérdidas y otros factores.

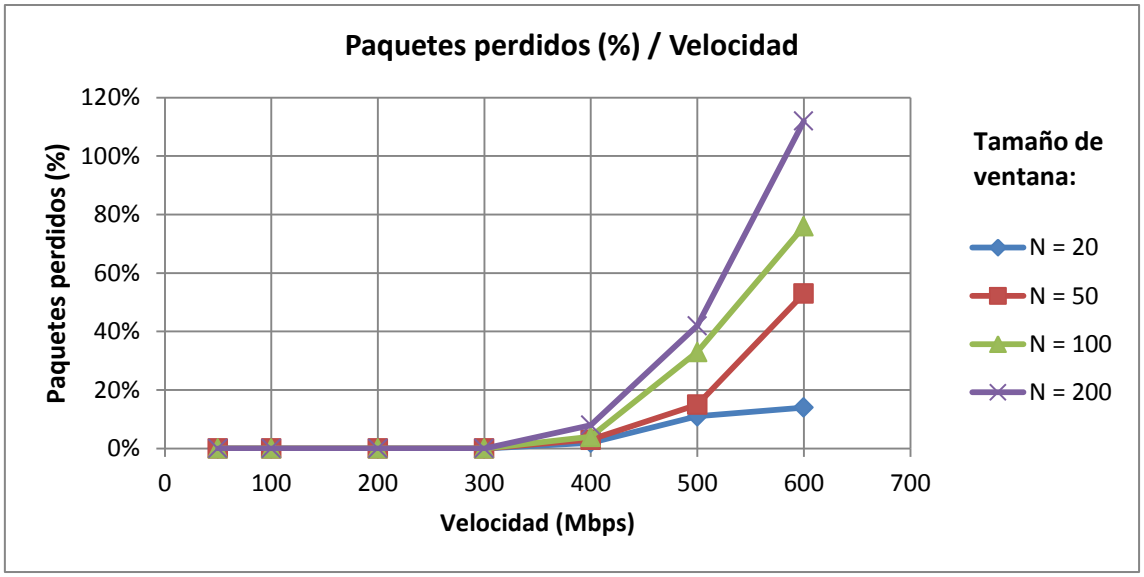

Figura 3.24: Relación entre pérdida de paquetes y velocidad de transmisión (Linux)

Por último, se muestra la gráfica que relaciona la velocidad real alcanzada en las transferencias con la velocidad de envío de los paquetes en cada ventana de transmisión.

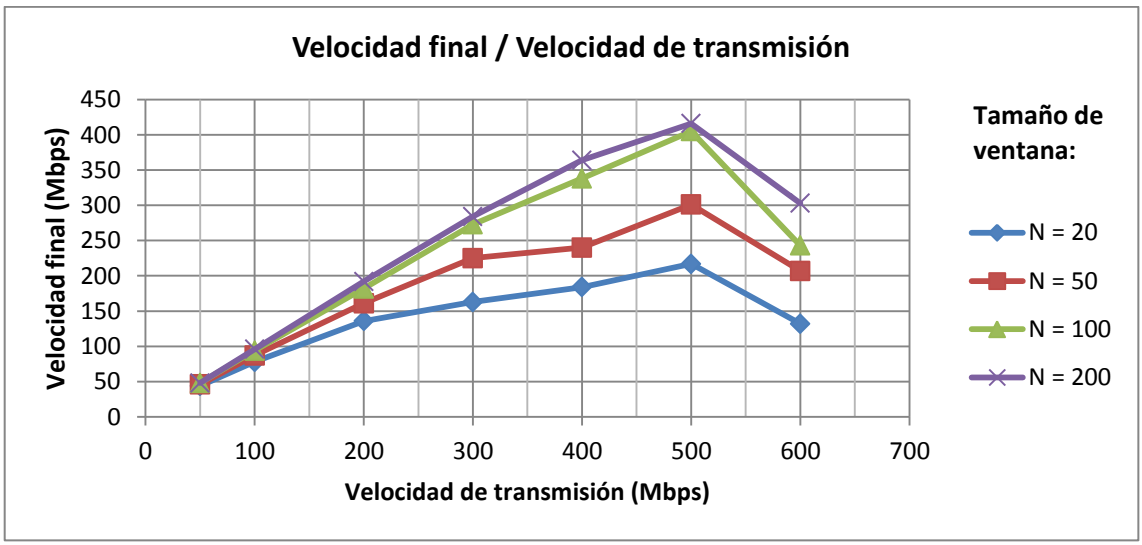

Figura 3.25: Relación entre velocidad final alcanzada y velocidad de transmisión (Linux)

Iperf

De la misma forma que en el apartado de resultados en entorno Standalone, se utilizó en Linux la herramienta Iperf[1] para obtener una segunda medida de la velocidad de transferencia que se podía alcanzar entre un PC ejecutando Windows y una Zedboard con sistema operativo Xillinux.

Zedboard como cliente:

```
$ ./iperf -c 192.168.1.100 -t 30 -w 64k
------------------------------------------------------------
Client connecting to 192.168.1.100, TCP port 5001
TCP window size:  64.0 KByte
------------------------------------------------------------
[  3] local 192.168.1.10 port 37965 connected with 192.168.1.100 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec  1.59 GBytes   461 Mbits/sec
```

Figura 3.26: Resultados de Iperf utilizando la Zedboard como cliente (Linux)

PC como cliente:

```
> iperf.exe -c 192.168.1.10 -t 30 -w 64k
------------------------------------------------------------
Client connecting to 192.168.1.10, TCP port 5001
TCP window size: 64.0 KByte
------------------------------------------------------------
[  3] local 192.168.1.100 port 50634 connected with 192.168.1.10 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-30.0 sec  1.76 GBytes   503 Mbits/sec
```

Figura 3.27: Resultados de Iperf utilizando un PC como cliente (Linux)

Los resultados obtenidos tanto en recepción como en transmisión fueron los siguientes:

Recepción (Mbps)	Transmisión (Mbps)
461	503

Tabla 3.10: Resultados obtenidos con Iperf (Linux)

3.7. Resultados y conclusiones

De los resultados obtenidos pueden obtenerse diversas conclusiones. Atendiendo a las características de transferencia, se pueden citar las siguientes:

- Es evidente que la longitud del *payload* de los paquetes es un factor muy significativo: a mayor tamaño del campo de datos, mayor es la velocidad alcanzada. Esto es debido al mayor aprovechamiento de cada paquete para transmitir datos y a la disminución de paquetes necesarios que ello implica.

- En cuanto a la tasa de pérdidas de paquetes, se observa que éstas aumentan al hacerlo tanto la velocidad como el tamaño de la ventana. Esto es debido al limitado tamaño del buffer de los dispositivos a la hora de recibir tramas, lo que puede resultar en un desbordamiento y en la consiguiente pérdida de paquetes cuando la velocidad de transferencia es demasiado alta y la velocidad de los sistemas para procesar estos paquetes constituye entonces un cuello de botella. Por ello es muy importante que este procesamiento sea lo más eficiente posible.

- El compromiso fundamental para la transferencia de datos se da entre el tamaño de la ventana de transmisión y la velocidad de transmisión. Tamaños de ventana pequeños conllevarán un gran número de paradas para envío de tramas de control, por lo que conviene utilizar ventanas mayores, aunque respetando las capacidades de recepción del sistema cliente.

Atendiendo a los resultados obtenidos:

- La velocidad máxima obtenida en Standalone, que ha alcanzado los **291 Mbps**, se siente por debajo de las velocidades que cabían esperar de un dispositivo Gigabit Ethernet. En contraste con los valores más elevados obtenidos tanto con la herramienta Iperf como con el protocolo de pruebas en Linux, puede atribuirse esta baja velocidad de transferencia a la implementación de la librería Ethernet, por lo que en el apartado de Líneas Futuras se comentará la posibilidad de una revisión de la librería para mejorar su rendimiento.

- En un entorno Linux, la velocidad más alta que se ha alcanzado manteniendo un comportamiento estable está en torno a los **420 Mbps**. Se trata de un resultado que,

si bien es algo inferior a los resultados obtenidos utilizando la herramienta Iperf, es coherente con ellos y con el hecho de utilizar un sistema de *benchmarking* menos sofisticado.

- Los resultados obtenidos en el entorno Standalone con la herramienta Iperf ponen de manifiesto el buen funcionamiento de la librería lwIP, aunque se haya utilizado en combinación con otros sistemas (*FreeRTOS*). Especialmente, los resultados conseguidos con el modo RAW hacen muy atractiva la idea de estudiar el funcionamiento de la librería lwIP (en el caso de trabajar en modo Standalone), además de que ello permitiría disponer del soporte que ésta ofrece de diversos protocolos como TCP y UDP.

En líneas generales, y en vista de los resultados obtenidos, la opción que parece más recomendable para el desarrollo de aplicaciones que hagan uso de un medio de comunicación Ethernet es la de hacerlo sobre un sistema operativo Linux. Aunque depende de las características de cada aplicación, la velocidad que ofrece este sistema parece satisfactoria, y esto unido a la mayor sencillez en el desarrollo y a la modularidad y los servicios que ofrece un sistema operativo completo como Linux, parecen hacerla la opción más favorable.

4. Comunicación PS-PL

4.1. Introducción

Como ya se comentó en el capítulo 3, las ventajas de utilizar un sistema operativo Linux sobre una plataforma Zynq en lugar de desarrollar aplicaciones Standalone que se ejecuten directamente sobre ella son numerosas, y justificarán en muchos casos esta primera opción.

Las facilidades de manejo y gestión del sistema cuando se dispone de un entorno multiproceso con sistema de archivos, multitud de librerías, consola de comandos, etc., hacen que, si bien para aplicaciones concretas un software *bare-metal* pueda ser una solución sencilla y muy eficiente, el empleo de un sistema operativo se haga muy atractivo y en algunos casos casi imperativo.

Por ejemplo, la utilización de un sistema de computación distribuida sería impensable en un entorno Standalone cuando existen soluciones como OpenSSI para un sistema operativo Linux.

Por estas razones resulta muy interesante la familiarización con Linux sobre una plataforma Zynq. Sin embargo, para que su uso tenga sentido debe cumplirse un requisito: garantizar una buena integración entre el sistema y los periféricos presentes en la zona reconfigurable (PL) de la plataforma.

4.2. Linux y los periféricos en la PL

Las distribuciones de Linux para las plataformas Zynq presentan un buen soporte para dispositivos como Ethernet, USB, SD, GPIO, VGA, HDMI… En resumen, se podría decir que los periféricos estándar con los que cuenta una placa podrán ser manejados sin problemas desde el sistema operativo Linux que ejecute. Sin embargo, no se puede decir lo mismo de los periféricos programados en la FPGA de una Zynq, esto es, en su PL o *Programmable Logic*, a pesar de que su configuración y conexiones sean correctas.

En estos casos, manejar dichos periféricos desde una aplicación Standalone suele resultar mucho más sencillo, ya que permitirá acceder sin restricciones a recursos tales como áreas de memoria, controladores de interrupciones o funciones de drivers específicos que no están disponibles cuando se da el salto a Linux.

Sin embargo, puede no resultar complicado el desarrollo de un driver personalizado que se incluya en el kernel del sistema operativo y que permita acceder a sus funcionalidades.

En los siguientes apartados se tratará de integrar un dispositivo sencillo con una distribución de Linux. La distribución será **Xillinux**, y su puesta en marcha en la placa se describirá en el siguiente apartado. En el apartado 4.4 se mostrará cuáles son los pasos con los que se puede conseguir manejar un periférico programado en la PL. Será un sencillo contador Timer, que servirá de ejemplo ilustrativo para esta tarea ya que, además, cuenta con una funcionalidad muy interesante: un sistema de lanzamiento de interrupciones que se tratará de controlar para conseguir una comunicación más eficiente con los dispositivos que utilicen esta característica.

4.3. Puesta en marcha de un SO Linux

La carga de la distribución de Linux empleada se realizará desde una tarjeta de memoria SD. En este caso, esta distribución será **Xillinux**[9], cuya versión de evaluación (aunque sin limitaciones) está disponible de forma gratuita para las plataformas Zedboard, ZyBo y MicroZed de la familia Zynq de Xilinx.

La descripción de cómo empezar a usar Xillinux en una plataforma Zynq se describe detalladamente en la *Getting Started Guide for Zynq*, disponible en la web de *Xillybus* (http://xillybus.com/xillinux). Por tanto, en este apartado se realizará solo una descripción muy rápida del proceso a partir de la documentación presentada junto con esta memoria, donde se encuentran los archivos de la distribución Xillinux listos para ser utilizados con una Zedboard.

1. Escritura de la imagen (*xillinux-1.3.img* en nuestro caso) en la tarjeta SD. El sistema variará según el sistema operativo empleado (en Windows puede utilizarse el programa *Win32 Disk Imager*, por ejemplo; en Linux, puede usarse el comando *dd*). Tras la copia de la imagen, dos particiones serán creadas en la tarjeta SD.

2. La menor de las particiones creadas es la partición de inicio (formato es FAT32). En ella existirá únicamente un archivo, *uImage*, que es el archivo binario del kernel de Linux, y no depende de la placa. Es necesario copiar en esta partición tres archivos más:

    - boot.bin, el bootloader o lanzador del sistema.
    - devicetree.dtb, el árbol de dispositivos de este sistema Linux. Contiene información del hardware para el kernel de Linux, por lo que éste será uno de los archivos que se modificará más adelante para dar soporte a nuestros periféricos.
    - xillydemo.bit, el archivo de programación en la PL del sistema Xillinux. El archivo incluido en la documentación es una versión precompilada para su uso desde una placa Zedboard.

3. Se configuran los *jumpers* de la placa para arrancar el sistema desde la tarjeta SD.

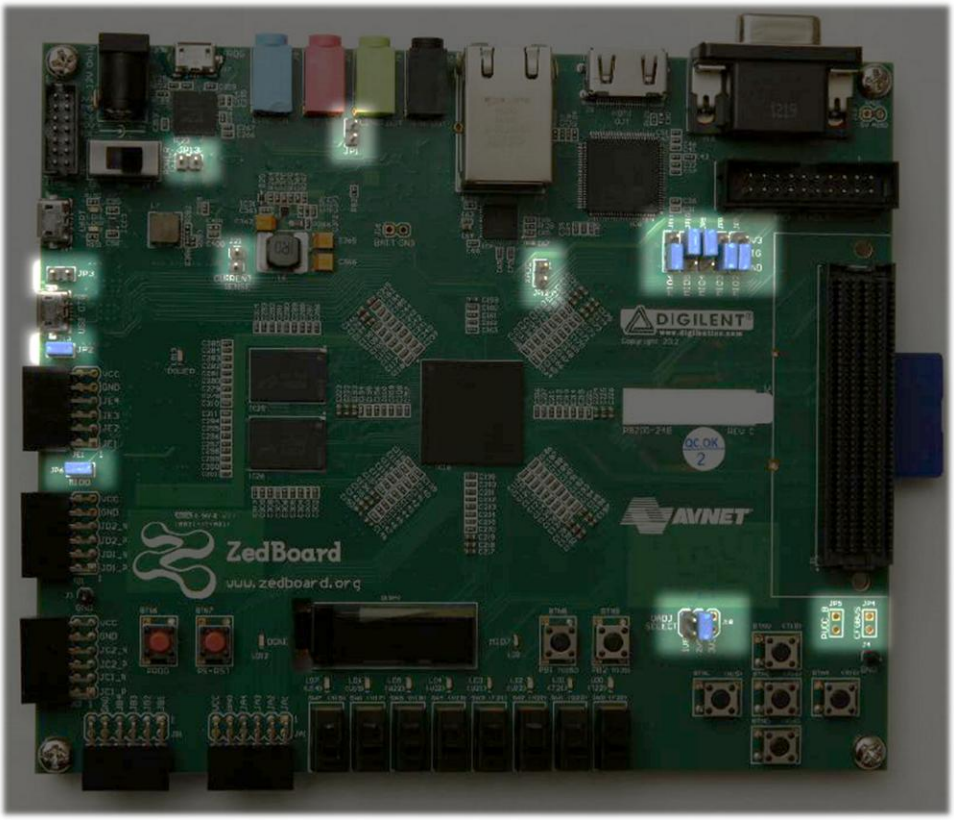

Figura 4.1: Configuración de los *jumpers* de una Zedboard para arrancar el sistema desde la tarjeta SD

4. Por último, se alimenta y enciende la Zedboard y el sistema se cargará. El acceso a la consola puede realizarse tanto por conexión serie como por SSH a través de Ethernet.

4.4. Control de dispositivos

En este apartado se mostrará un procedimiento para manejar periféricos programados en la zona reconfigurable (PL) de una plataforma Zynq. Este proceso cubrirá varias partes:

- Preparación en Vivado de un proyecto que incluye una sencilla IP en la PL (AXI Timer).
- Obtención de algunos parámetros del dispositivo relativos a la memoria y a la plataforma en el SDK.
- Edición del árbol de dispositivos de Linux.
- Desarrollo de un módulo para el kernel que controle el dispositivo.
- Desarrollo de una aplicación de usuario que interactúe con el dispositivo a través del módulo del kernel.

4.4.1. Preparación de la plataforma

En apartado se cubrirá la inclusión y configuración de un dispositivo AXI Timer en la PL de la plataforma, un sencillo contador con sistema de interrupciones. Tras este proceso, realizado en Vivado, se obtendrá en el SDK el código de interrupciones utilizado y las direcciones de memoria de los diferentes registros del Timer. Con estos datos se podrá, posteriormente, hacer uso del dispositivo desde un sistema operativo Linux.

Vivado

El proceso de preparación de la plataforma es muy similar al descrito en el apartado 3.4.1, pero en este caso se incluirá un AXI Timer y se conectará con el sistema Zynq. Por ello, se creará un nuevo proyecto siguiendo los mismos pasos que los descritos en los apartados 1 y 2 de dicha sección, hasta contar con el siguiente diseño:

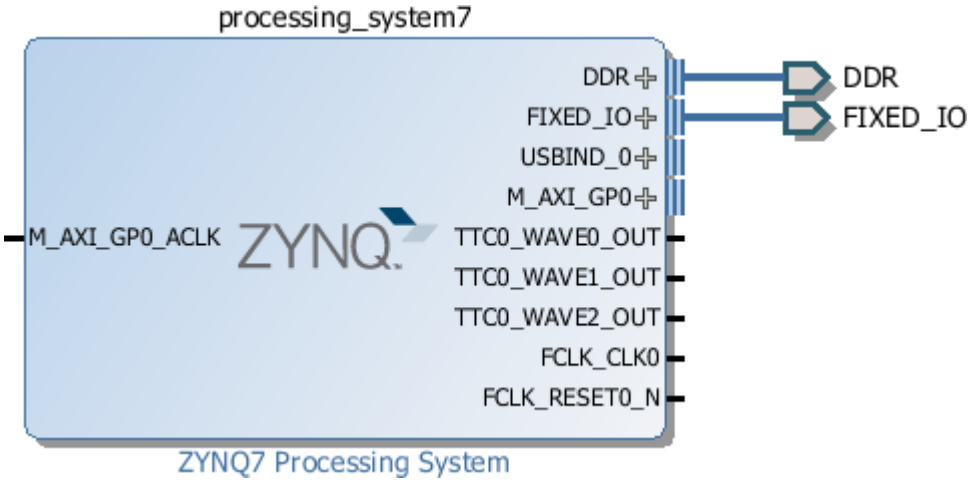

Figura 4.2: Bloque Zynq con conexiones externas

A partir de este punto:

1. Se incluirá un AXI Timer al diseño. Clic en **Add IP** y se busca y selecciona el elemento **AXI Timer**, que se renombrará como *axi_timer*.

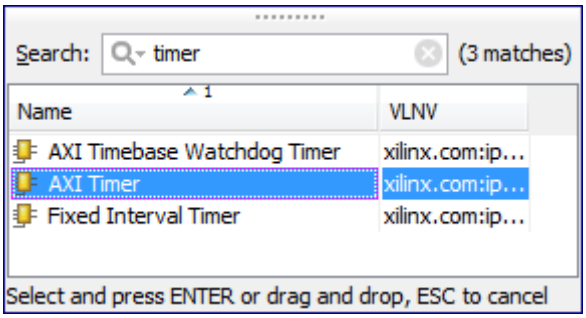

Figura 4.3: Selección de un AXI Timer en el catálogo de IPs

2. Una vez agregado al diseño, se realizará una conexión automática de bloques. Clic en **Run Connection Automation -> /axi_timer/S_AXI**. Dos bloques intermedios entre el sistema y el periférico se agregarán, quedando el diseño de la siguiente forma:

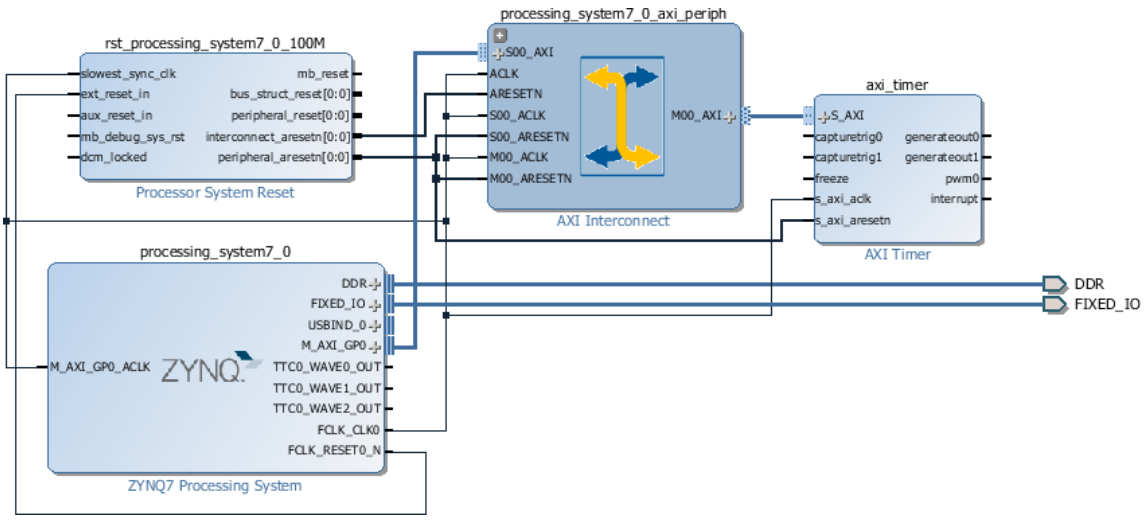

Figura 4.4: Aspecto del diseño tras la conexión automática de bloques

3. Se hará ahora doble clic sobre el bloque Zynq para configurarlo. Como ya se describió en el apartado 3.4.1, se habilitarán e inhabilitarán los dispositivos y características del sistema en función de las necesidades del proyecto.

4. Para recibir las interrupciones lanzadas por el AXI Timer que se ha agregado al diseño iremos al apartado **Interrupts**. Dentro de él, habilitamos la casilla **Fabric Interrupts** y la casilla **IRQ_F2P[15:0]** dentro de **PL-PS Interrupt Ports**. Esto hará posible que podamos conectar la salida de interrupciones del AXI Timer (u otros periféricos en la PL) al sistema Zynq a través de la línea IRQ, encargada de recibir las interrupciones.

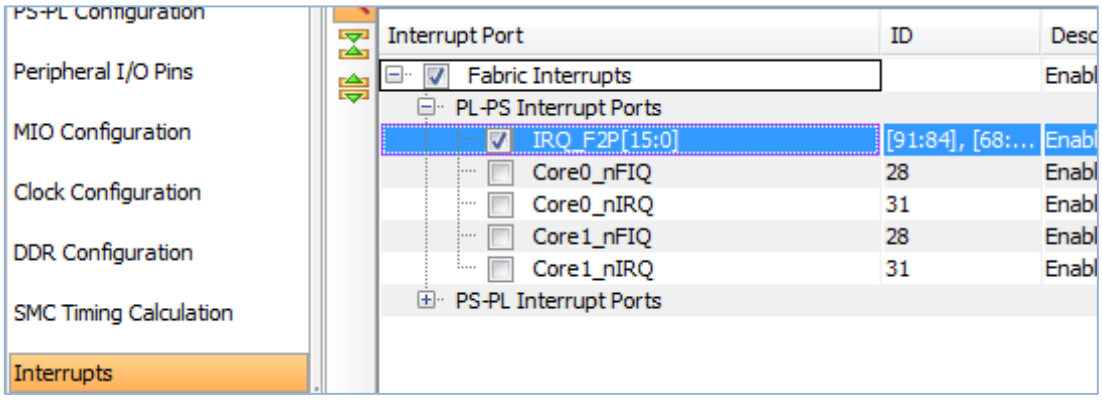

Figura 4.5: Habilitación de las interrupciones

5. Una vez configurada la plataforma, se hará clic en **OK**. Un nuevo puerto (IRQ_F2P[0:0]) se agregará al sistema.

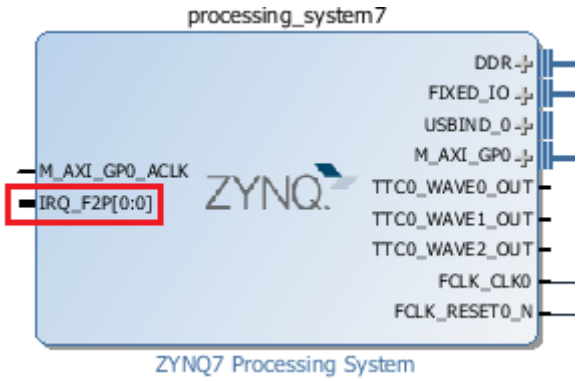

Figura 4.6: Bloque Zynq con puerto de entrada de interrupciones (resaltado)

6. Manualmente se conectará la salida de interrupciones (**interrupt**) del AXI Timer con esta nueva entrada de interrupciones al sistema.

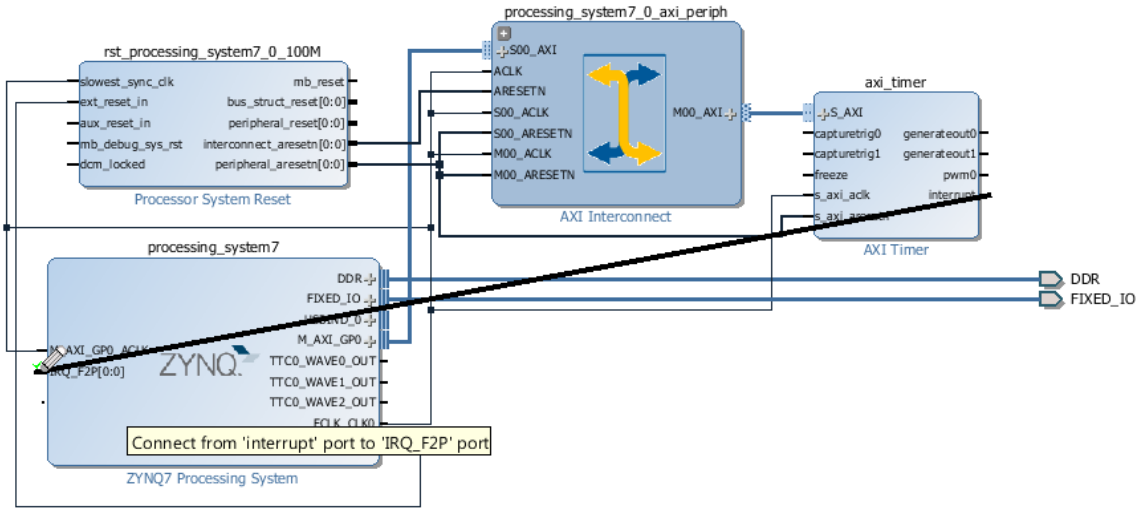

Figura 4.7: Conexión de los puertos de interrupciones

7. Una vez finalizado el diseño se procederá a la **creación del top-level y a la exportación al SDK**, tal como se describió en el apartado 3.4.1. Sin embargo, en este caso hemos agregado lógica en la PL, por lo que antes de exportar el proyecto será necesario generar el *bitstream* que define esta lógica y que permitirá programar la FPGA con nuestro dispositivo. Para ello se hace clic en **Generate Bitstream**, dentro del apartado **Program and Debug** del **Flow Navigator**. Una vez generado, exportamos el proyecto al SDK.

SDK

En las especificaciones de la plataforma podremos ver que nuestro AXI Timer se ha agregado correctamente, y sabremos qué región de la memoria tiene reservado.

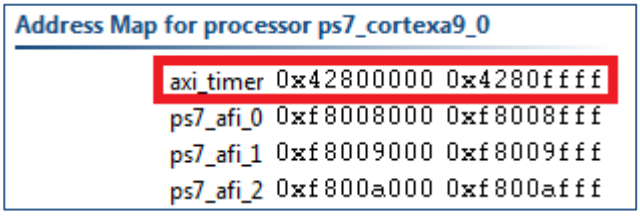

Figura 4.8: Mapa de direcciones de la plataforma diseñada (archivo *system.hdf*)

A continuación se obtendrán los datos necesarios para el funcionamiento del dispositivo en un sistema Linux: el número de la línea de interrupción y la dirección de memoria de los registros del Timer.

1. Se crea un BSP para la plataforma generada.

2. En el archivo *system.mms* del BSP generado, en el apartado **Peripheral Drivers** podremos acceder a través de un link a la documentación correspondiente al driver del dispositivo AXI Timer. Inspeccionando los diferentes archivos que lo componen encontraremos uno que contiene el valor de offset de los registros del timer. Este valor indica el número de bytes de diferencia de la dirección del registro con respecto a la dirección base del dispositivo (0x42800000 en este ejemplo).

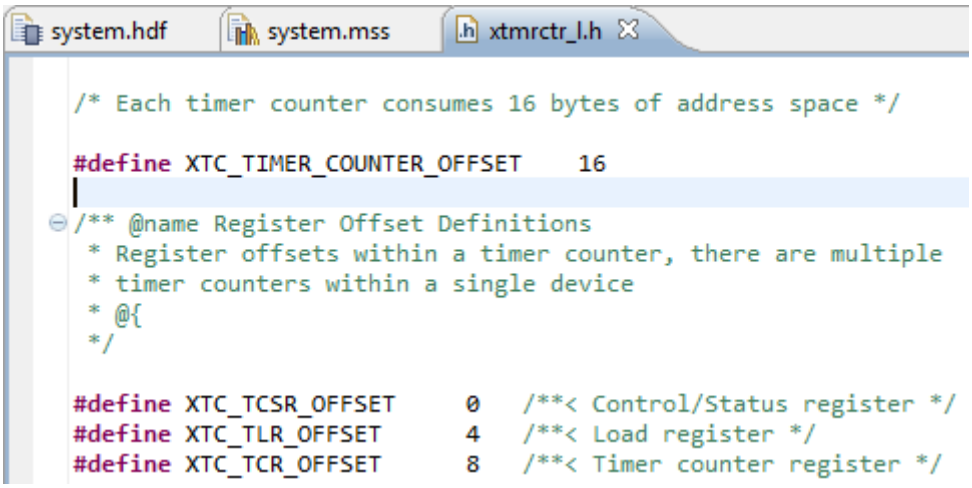

Figura 4.9: Offset de los registros de un AXI Timer desde la dirección base

A partir de estos valores se podrá acceder desde Linux a los diferentes registros, con los que se tendrá el control del dispositivo. Por supuesto, puede no ser necesario integrar todas las funcionalidades del dispositivo, así que dependerá de la aplicación y del desarrollador escoger aquellos datos que garantizarán un correcto funcionamiento del periférico.

3. Por último, es necesario conocer qué línea utiliza el dispositivo para lanzar interrupciones al sistema. Este valor se encuentra en el archivo **xparameters.h**, en el directorio *ps7_cortexa9_0/include* del BSP.

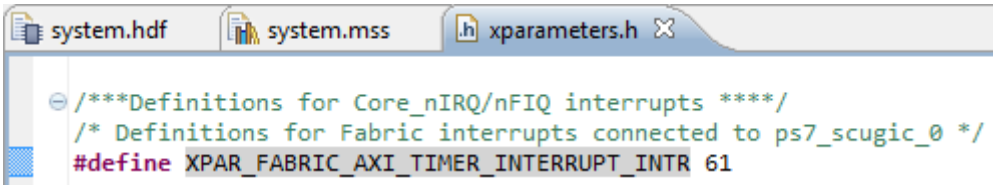

Figura 4.10: Número de la línea de interrupción utilizada por el AXI Timer

4.4.2. Desarrollo del driver en Linux

El desarrollo de un driver para manejar un dispositivo en un sistema operativo Linux pasa por distintas fases. Sin pretender abarcar en profundidad detalles más relacionados con el conocimiento del sistema operativo que con la simple aplicación que se ha propuesto en este apartado, se explicarán a continuación estas fases que componen el proceso.

Edición del Device Tree de Linux

El *Device Tree*[11] es un archivo que contiene una estructura de datos de descripción del hardware de un sistema. Esta estructura está formada por nodos que definen las características y parámetros de cada dispositivo, de modo que el kernel de Linux pueda configurar el sistema correctamente y utilizar todos estos periféricos y dispositivos.

Por tanto, para poder hacer uso del AXI Timer implementado en nuestro diseño el primer paso será incluir un nodo que defina sus características fundamentales.

Para esta tarea se editará el *Device Tree* en su forma de archivo de texto (*.dts). A continuación se compilará dicho archivo para obtener el archivo binario (*.dtb), que es el que leerá el kernel para configurar la plataforma.

1. Se abre en un editor de texto el archivo **xillinux-1.3-zedboard.dts** (esto variará según la distribución y la plataforma, pero siempre será un archivo de texto de extensión *.dts*) incluido en los archivos descargados con la distribución.

   Este archivo estará compuesto por nodos agrupados en categorías, cada uno de los cuales define un dispositivo diferente (CPUs, memorias DDR, controlador DMA, Ethernet, etc.)

2. Dentro del grupo encabezado por la etiqueta `ps7_axi_interconnect_0` se encontrarán los nodos de los dispositivos conectados a través del bus AXI del sistema. Por tanto, en este grupo incluiremos el nodo del AXI Timer de nuestro diseño. Del mismo modo en que están definidos los demás, se incluirá un código como el siguiente (en negrita):

```
zed_oled {
        compatible = "dglnt,pmodoled-gpio";
        /* GPIO Pins */
        vbat-gpio = <&ps7_gpio_0 55 0>;
        vdd-gpio = <&ps7_gpio_0 56 0>;
        res-gpio = <&ps7_gpio_0 57 0>;
        dc-gpio = <&ps7_gpio_0 58 0>;
        /* SPI-GPIOs */
        spi-bus-num = <2>;
        spi-speed-hz = <4000000>;
        spi-sclk-gpio = <&ps7_gpio_0 59 0>;
        spi-sdin-gpio = <&ps7_gpio_0 60 0>;
};

axi_timer: axitimer@42800000 {
        compatible = "xlnx,axi-timer";
        interrupt-parent = <&ps7_scugic_0>;
        interrupts = < 0 29 4 >;
        reg = <0x42800000 0x10000>;
};

// Xillybus related:

xillyvga@50001000 {
        compatible = "xillybus,xillyvga-1.00.a";
        reg = < 0x50001000 0x1000 >;
} ;
```

Código 4.1: Detalle del código del dispositivo AXI Timer en el *Device Tree* (resaltado)

A continuación se explica cada una de las líneas resaltadas:

- **`axi_timer: axitimer@42800000:`** La primera parte (*axi_timer*) es la etiqueta del nodo, y sirve simplemente de identificador. La segunda parte (*axitimer@42800000*) indica el nombre de la entrada (*axitimer*) del dispositivo y su dirección física base.

- **`compatible = "xlnx,axi-timer":`** Especifica el driver del dispositivo.

- **`interrupt-parent = <&ps7_scugic_0>:`** Indica la etiqueta al nodo que define el controlador de interrupciones del sistema. Es importante debido a que el dispositivo que estamos definiendo (AXI Timer) lanzará interrupciones que deben ser capturadas.

- **`interrupts = < 0 29 4 >:`** El primer valor (0) indica que las interrupciones lanzadas son SPI o *shared peripheral interrupt*. El segundo valor indica el número de la línea de interrupciones empleada. Este valor será el obtenido previamente en el SDK (en nuestro caso, 61) al que es necesario restar 32 (61 – 32 = 29). El tercer valor indica el tipo de la interrupción: 0 dejarlo como está, pues el sistema puede haberlo definido; 1 para activación con flanco ascendente (*rising edge*); y 4 para activación en estado alto. No existen más valores, de modo que si se quisiera disparar una interrupción en flanco descendente o estado bajo habría que añadir una puerto NOT a la lógica.

- **`reg = <0x42800000 0x10000>:`** Especifica la zona de memoria empleada por el dispositivo. El primer valor indica la dirección base, y el segundo su tamaño.

3. Una vez editado el archivo DTS, deberá ser guardado y almacenado en algún directorio accesible desde el sistema Xillinux, ya que se procederá a su compilación.

4. El compilador de *Device Trees* se encuentra, en el sistema Xillinux, en el directorio

   `/usr/src/kernels/3.12.0-xillinux-1.3/scripts/dtc/`

   Se navegará hasta este directorio y se compilará el archivo DTS modificado para generar el archivo binario DTB:

   ```
   $ dtc -I dts -O dtb -o /ruta/a/device-tree.dtb /ruta/a/device-tree.dts
   ```

5. El último paso será sustituir el archivo DTB de la partición de inicio de la tarjeta SD por el nuevo que ha sido compilado. Tras esto se reiniciará el sistema y se procederá al desarrollo del driver del dispositivo.

Desarrollo del módulo para el kernel

El kernel de Linux es un nivel de software que interactúa con el hardware de una computadora, desempeñando la tarea de interconectar los procesos que se ejecutan en el espacio de usuario con los dispositivos del sistema. También se encarga de la administración de la memoria y de la repartición del tiempo de ejecución de los procesos, entre otras funciones.

Por tanto, se pueden distinguir dos ámbitos diferentes en un sistema Linux: el kernel y el espacio de usuario. El kernel es un sistema protegido, fuertemente integrado con el hardware y el resto de componentes software que hacen funcionar el sistema operativo, lo que lo hace, por tanto, más rígido y poco accesible para los usuarios. El espacio de usuario es aquel en que se ejecutan los procesos que utiliza el usuario final (programas, comandos, etc.) y que están gestionados por el kernel.

Esta básica explicación sobre los dos espacios que existen en un sistema Linux dan una idea de lo que se necesita en este punto del proyecto: un componente, módulo o driver que se ejecute en el kernel del sistema y que sea capaz de comunicarse con el periférico que hemos agregado en la FPGA (el AXI Timer), y una aplicación de usuario que se comunique a su vez con este driver y permita interactuar con el dispositivo de la forma en que el desarrollador desee.

Se desarrollará ahora ese módulo que se integrará con el kernel de Linux y controlará en primera instancia el Timer. Cabe destacar que, si bien se podría manejar este dispositivo accediendo directamente a las zonas de memoria adecuadas (registros) desde una aplicación de usuario (utilizando la función *mmap*), realizar esta tarea desde el kernel es la opción más segura y elegante, además de que permite contar con funcionalidades que solo están disponibles en el kernel, como la notificación de interrupciones.

En este proyecto no se pretende dar más que una muy ligera guía de cómo se podría realizar un sencillo módulo para el kernel (*kernel module*), dado que tratar de explicar las técnicas de programación[4] de estos módulos sería tremendamente ambicioso y no es en absoluto la pretensión de este apartado. Por tanto, simplemente se comentarán algunos puntos sobre la programación que se han creído interesantes, así como la compilación e inserción de estos módulos en el sistema.

En primer lugar, y como se ha comentado ya, el kernel es un espacio diferente al de usuario. Es importante tener muy presente esta distinción, ya que la programación en él será diferente a la programación que normalmente se lleva a cabo para una aplicación. En el kernel no se tiene acceso a las librerías que habitualmente utilizan los programas (funciones matemáticas, gestión de archivos, acceso a redes, etc.), sino solo a aquellas que conforman el núcleo del sistema. Es por esto que el kernel debe ser un "puente" entre dispositivos y el espacio de usuario, ya que las funcionalidades con que cuenta son limitadas.

En esta aplicación cabe destacar algunos aspectos del módulo desarrollado:

- **Interrupciones**: El AXI Timer es un dispositivo capaz de lanzar interrupciones cuando su contador alcanza un cierto valor. Capturar estas interrupciones solo es posible desde el kernel, y para ello será necesario conocer el número de línea de interrupciones utilizada, que ya se obtuvo. Por ello, registrar esta línea de interrupciones será uno de los puntos fundamentales de este módulo, así como habilitar una función manejadora que se ejecutará cuando se produzcan.

- **Señales**: Como se ha dicho, las funciones con que cuenta el kernel son limitadas, y no es correcto implementar en el manejador de interrupciones del módulo la lógica a realizar. Así, cuando las interrupciones se produzcan será necesario disparar una rutina específica en la aplicación de usuario que esté utilizando el driver. Para evitar el *polling* se ha optado por el uso de señales como medio para comunicar de inmediato a la aplicación de usuario de la ocurrencia de estas interrupciones.

- **Comunicación kernel-userspace**: el kernel module y la aplicación de usuario deben intercambiarse información, ya sea para recibir datos sobre el dispositivo o para especificar alguna operación que deba realizarse. En este caso se ha usado para ello un archivo de configuración del módulo habilitado para lectura y escritura por las aplicaciones del espacio de usuario.

Como podría deducirse, los métodos de comunicación y de desarrollo del kernel module no son únicos. No existe una única forma de hacer las cosas, por lo que en cada aplicación la elección de las posibilidades podrá atender a razones diferentes.

El kernel module desarrollado, ***timer_module.c***, se encuentra en la documentación entregada con esta memoria y en el anexo.

Una vez escrito el código, el siguiente paso será la compilación en el sistema Linux. Para ello, en el mismo directorio del módulo se creará un archivo llamado **Makefile** con el siguiente contenido:

```
obj-m += timer_module.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Código 4.2: Contenido del archivo Makefile para la compilación del módulo timer_module.c

El archive *Makefile* permitirá la compilación del módulo. Ejecutando el comando

```
$ make
```

se realizará la compilación, generando diversos archivos de salida. El principal será ***timer_module.ko***, que es el que se insertará al kernel mediante *insmod*:

```
$ insmod timer_module.ko
```

Si se desea "sacar" el módulo insertado se utilizará *rmmod* (desde cualquier directorio):

```
$ rmmod timer_module
```

Por último, es importante saber que los mensajes que imprima un kernel module serán almacenados en un buffer de mensajes del núcleo del sistema. Para imprimirlos en la pantalla de la consola se utilizará

```
$ dmesg
```

Una vez insertado el *kernel module*, el último paso será ejecutar una aplicación de usuario que se comunique con él y con la que se hará uso de las funcionalidades del dispositivo.

Comunicación kernel-espacio de usuario

Para la comunicación con el dispositivo desde el espacio de usuario a través del module kernel insertado, se ha desarrollado una sencilla librería llamada ***axi_timer.h*** que implementa funciones para el manejo de las interrupciones (a través de *signals*) y para la lectura y escritura de los registros del AXI Timer a través de la lectura y escritura del archivo de configuración creado por el módulo. Un programa todavía más sencillo, ***timer_user.c***, hace uso de esta librería llamando a algunas de sus funciones y configurando al timer para que se lance una interrupción cada 5 segundos, que será capturada.

El envío de una señal entre un módulo del kernel y una aplicación de usuario se sintetiza en las siguientes porciones de código. Para realizar esta comunicación, el módulo debe conocer el PID (*Process ID*) de la aplicación de usuario, por lo que ésta última debe comunicárselo al módulo antes de comenzar a recibir señales (en el ejemplo utilizado, esto se hace utilizando la comunicación con el archivo de configuración de módulo).

El envío de una señal (que, en este caso, se realiza en el manejador de interrupciones del *kernel module*) se puede realizar de la siguiente forma:

```
/******** Envío de la señal al userspace ********/

struct siginfo info;

memset(&info, 0, sizeof(struct siginfo));
info.si_signo = SIG_TEST;
info.si_code = SI_QUEUE;
info.si_int = timervalue;  // Las señales real-time pueden llevar hasta 32 bits de datos

rcu_read_lock();
t = pid_task(find_pid_ns(app_pid, &init_pid_ns), PIDTYPE_PID);
if(t == NULL){
        printk("No existe el PID.\n");
        rcu_read_unlock();
        return -ENODEV;
}
rcu_read_unlock();
ret = send_sig_info(SIG_TEST, &info, t);    // Se envía la signal
```

Código 4.3: Configuración y envío de una señal a una aplicación de usuario

Para recibir esta señal en una aplicación de usuario, hay que especificar el tipo de señal que se espera capturar y la función que se disparará cuando esto ocurra. Esto puede realizarse con unas simples líneas de código:

```
// Se configura la señal para la notificación de interrupciones desde el kernel
struct sigaction sig;
sig.sa_sigaction = Timer_Int_Handler;    // Timer_Int_Handler() se ejecutará al recibir la señal
sig.sa_flags = SA_SIGINFO;
sigaction(SIG_TEST, &sig, NULL);
```

Código 4.4: Configuración de la recepción de señales en el espacio de usuario

El código completo del programa y de la librería se encuentran tanto en el anexo de esta memoria como en el material presentados con este proyecto.

Conclusión

Como se ha comentado, este apartado ha pretendido principalmente ilustrar un ejemplo para demostrar cómo es posible trasladar la funcionalidad de un dispositivo desde un entorno Standalone a otro Linux. Si bien esta portabilidad no es en absoluto directa y debe ser trabajada (siendo tarea del programador estudiar las particularidades del hardware y la implementación de los módulos), el resultado puede merecer mucho la pena al ponerse a disposición del desarrollador un sistema operativo completo.

5. Conclusiones y Líneas Futuras

5.1. Conclusiones

Llegados a este apartado, y dando por finalizada la investigación de la plataforma Zynq dentro del marco de este proyecto, resulta interesante hacer una síntesis del trabajo realizado y exponer aquellas conclusiones que pueden ser extraídas a partir de los resultados observados.

- Se ha estudiado la plataforma Zynq-7000 de Xilinx (utilizando una placa de desarrollo Zedboard), así como las herramientas de desarrollo ofrecidas (Vivado y Xilinx SDK).
- Se ha escogido un medio de comunicación Ethernet como enlace de estudio a la hora de transferir información entre un PC y la plataforma de Xilinx.
- Con el fin de caracterizar las propiedades de este canal Ethernet, se ha diseñado un sencillo sistema para la plataforma Zynq utilizando la herramienta Vivado, indicando todos los pasos básicos necesarios.
- Se ha estudiado la librería XEmacPs de Xilinx, que controla los dispositivos Ethernet disponibles en las placas Zynq y permite la transferencia de información a nivel de enlace en entornos Standalone.
- Utilizando la herramienta de desarrollo Xilinx SDK, se ha desarrollado a partir del driver XEmacPs una librería propia con el fin de simplificar el uso del dispositivo Ethernet y facilitar la tarea de desarrollo de aplicaciones que hagan uso de este controlador.
- Se ha diseñado un protocolo Ethernet para la transmisión de archivos, y se ha desarrollado un servidor y un cliente de transferencia que implementa este protocolo. Se han realizado pruebas de velocidad entre un PC (con SO Windows) y la placa Zedboard (tanto con SO Linux como en modo Standalone) con el fin de caracterizar el proceso de transmisión, intercambiando los papeles de servidor y cliente.
- Utilizando la aplicación Standalone en la Zedboard, se ha obtenido una velocidad de transferencia máxima de **291 Mbps**.

- Con el fin de estudiar el comportamiento de la plataforma Zynq en un entorno con sistema operativo, se ha utilizado la distribución Xillinux del sistema operativo Linux.

- Se han ejecutado las pruebas de rendimiento de Ethernet en Linux, obteniendo una velocidad máxima de transferencia de **422 Mbps**.

- Con el fin de conseguir un buen control sobre dispositivos implementados en la PL cuando se utilizan un sistema operativo Linux se ha añadido un sencillo periférico (un AXI Timer) al sistema, y se ha descrito el proceso seguido para llegar a manejarlo en Linux.

La tarea fundamental de este proyecto ha sido la de caracterizar el medio Ethernet para conocer sus posibilidades a la hora de utilizarlo como enlace entre plataformas en futuras aplicaciones específicas.

Como era de esperar, Linux se presenta como una alternativa muy atractiva al desarrollo de aplicaciones en entorno Standalone. Los resultados obtenidos, así como la mayor sencillez de desarrollo de aplicaciones y el sinfín de posibilidades y comodidades que ofrece un sistema operativo, justificarán en la mayoría de casos esta elección.

No obstante, ejecutar una aplicación nativamente sobre un sistema tiene ventajas cuando lo que busca la aplicación es sacar el máximo rendimiento de la plataforma, o, por el contrario, cuando ésta es lo suficientemente sencilla como para no justificar la necesidad de un sistema operativo. No obstante, en el aspecto de la comunicación Ethernet en este entorno Standalone quedaría un amplio margen de mejora de las características, y puede ser necesario el estudio de otras aproximaciones para su uso cuando se requiera una muy alta velocidad de transferencia de datos.

5.2. Líneas futuras

Debido a la enorme cantidad de posibilidades que se presentan a la hora de abordar una cuestión tan amplia como es la investigación de una nueva plataforma PSoC, se hace imposible la tarea de abarcarlas todas en el marco de un simple proyecto. No obstante, sería muy interesante el estudio de todos o algunos de los temas que han ido surgiendo a lo largo del desarrollo del trabajo como posibles soluciones a los problemas que se han ido presentando.

Por otra parte, lo presentado en esta memoria se trata de un acercamiento a la plataforma Zynq en la que se ha intentado dar respuesta a una serie de problemas planteado. Sin embargo, esto no significa que no pueda existir un gran margen de mejora sobre lo aquí realizado, así como la posibilidad de completar o complementar estos resultados.

De esta forma, se comentan a continuación algunas tareas que se han considerado interesantes para seguir mejorando la experiencia ofrecida por la plataforma:

- **Jumbo Frames**: la limitación del tamaño de los paquetes utilizados en una transmisión a través de Ethernet (**~**1500 bytes) limita también la velocidad de transferencia de datos al separar éstos en más paquetes y al exigir una mayor capacidad de procesamiento por parte del receptor. Si bien no parece haber soluciones directas y sencillas para el soporte de Jumbo Frames por el dispositivo Ethernet en las plataformas Zynq, sería interesante estudiar el asunto y su evolución, ya que previsiblemente ofrecerían considerables ventajas a las transferencias al permitir longitudes de paquete mucho mayores (**~**9000 bytes).

- **Overclocking**: Como se ha comentado, parte del problema a la hora de obtener una alta velocidad de transferencia de datos es la de procesar las tramas recibidas en tiempo real, sin que se produzca en este punto un cuello de botella que lleve a la pérdida de paquetes. Por ello, sería interesante estudiar la posibilidad de aumentar la frecuencia de reloj del procesador y las memorias (dentro de límites seguros) y cuantificar su efecto en la mejora de las tareas de transferencia de información.

- **lwIP**: Los resultados arrojados por las pruebas realizadas con Iperf en entorno Standalone hacen destacar el buen rendimiento de la librería *LightWeightIP*. Por ello, conocer su funcionamiento podría ser una buena solución al problema de transferencia de paquetes Ethernet en entorno Standalone.

- **Otros protocolos**: El uso de protocolos de transmisión fiables, conocidos y bien implementados como TCP o UDP podría constituir un mecanismo más eficiente para la transferencia de información. Este aspecto resalta la importancia del estudio de la librería lwIP cuando se requiera trabajar en un entorno Standalone, ya que incorpora soporte para estos y otros protocolos.

- **DMA**: Como se ha visto, trabajar en un sistema operativo Linux presenta ciertas dificultades a la hora de interactuar con periféricos programados en la PL. Aunque se ha realizado una aproximación a la integración de estos dispositivos con el sistema operativo, aún quedarían pendientes tareas como el uso de mecanismos DMA cuando los periféricos implementados soporten esta funcionalidad.

No cabe duda de que las plataformas Zynq ofrecen tanto rendimiento como ventajas a la hora del desarrollo, todo ello unido a una nueva arquitectura de diseño que ofrece muchas posibilidades.

Por tanto, aunque en este proyecto se hayan dado respuestas a diversas cuestiones planteadas, queda claro que existe un amplio rango de escenarios por los que se podría continuar con el estudio de estas prometedoras plataformas.

Anexo. *Kernel module* y aplicación de usuario

Debido a su relativa brevedad, se muestran en este anexo los códigos de ejemplo utilizados para la comunicación entre el kernel y el espacio de usuario, y que permiten manejar un dispositivo AXI Timer.

En primer lugar, se muestra el código del *kernel module* desarrollado (**timer_module.c**):

timer_module.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/interrupt.h>
#include <linux/irq.h>
#include <asm/io.h>
#include <linux/fs.h>                   // Para fops
#include <linux/signal.h>               // Para señales kernel-userspace
#include <asm/siginfo.h>                // siginfo
#include <linux/rcupdate.h>             // rcu_read_lock
#include <linux/sched.h>        /       / find_task_by_pid_type
#include <linux/debugfs.h>
#include <linux/uaccess.h>              // Para 'copy_from_user' and 'copy_to_user'

MODULE_LICENSE("GPL");

#define SIG_TEST 44     // Definición de una señal propia (rango de 33 a 64 para real-time signals)

#define IRQ_NUM         61                      // Línea de interrupción
#define TIMER_BASE      0x42800000

#define TIMER_SIZE      0x0000FFFF
#define TCSR0           0x00000000
#define TLR0            0x00000004
#define TCR0            0x00000008
#define T0INT           0x00000100

#define TIMER_TCSR0     0x42800000      // Dirección del control and status register del Timer
0
#define TIMER_TLR0      0x42800004      // Dirección del load register del Timer 0
#define TIMER_TCR0      0x42800008      // Dirección del counter register del Timer 0
unsigned long *pTIMER_TCSR0;            // Puntero al control and status register del Timer 0
unsigned long *pTIMER_TLR0;             // Puntero al load register del Timer 0
unsigned long *pTIMER_TCR0;             // Puntero al counter register del Timer 0

#define DEVICE_NAME "timer"             // Nombre del dispositivo
#define SUCCESS 0                       // Valor de retorno Success


struct dentry *file_debugfs;            // Archivo para comunicación kernel-userspace
static int app_pid = 0;                 // PID de la aplicación en el userspace
```

```
// Función write (llamada cuando write() es usado en el espacio de usuario)
ssize_t syscall_write(struct file *flip, const char *buf, size_t length, loff_t *offset)
{
        char operation;
        unsigned int value;

        if(length > 10) return -EINVAL;

        // Lectura del valor
        operation = buf[0];
        copy_from_user(&value, buf+1, sizeof(unsigned int));

        printk("syscall_write.\n");
        printk("Value received: %u.\n", value);

        switch (operation)              // El primer elemento del array buf indica la operación
        {
                case '0':                       // Set control and status register
                        iowrite32(value, pTIMER_TCSR0);
                        break;
                case '1':                       // Set load register
                        iowrite32(value, pTIMER_TLR0);
                        break;
                case '2':                       // Set counter register
                        iowrite32(value, pTIMER_TCR0);
                        break;
                case '3':                       // Set PID
                        app_pid = value;
                        printk("pid = %d\n", app_pid);
                        break;
                default:
                        return 0;
        }
        return (int)length;
}

// Función read (llamada cuando read() es usado en el espacio de usuario)
ssize_t syscall_read(struct file *flip, char *buf, size_t length, loff_t *offset)
{
        unsigned int value;
        printk("syscall_read.\n");

        switch (buf[0])         // El primer elemento del array buf indica la operación
        {
                case '0':                       // Read control and status register
                        value = ioread32(pTIMER_TCSR0);
                        printk("Control and status register = %u\n", value);
                        break;
                case '1':                       // Red load register
                        value = ioread32(pTIMER_TLR0);
                        printk("Load register = %u\n", value);
                        break;
                case '2':                       // Read counter register
                        value = ioread32(pTIMER_TCR0);
                        printk("Counter register = %u\n", value);
                        break;
                default:
                        return 0;
        }

        // Se copia el valor en el espacio de usuario
        if (copy_to_user(buf, &value, sizeof(unsigned int)) != 0)
                return -EFAULT;
        else
                return 0;
}

// Archivo de operaciones
struct file_operations syscall_fops = {
        .read = syscall_read,           // read()
        .write = syscall_write,         // write()
};
```

```
// Manejador de interrupciones del Timer
static irqreturn_t irq_handler(int irq,void*dev_id)
{
        int ret;
        struct siginfo info;
        struct task_struct *t;

        unsigned long temp;
        unsigned long timervalue;

        timervalue = ioread32(pTIMER_TCR0);      // Read Timer/Counter Register
        printk("Interrupt! Timer counter value : %lu Cycles\n", timervalue);

        temp = ioread32(pTIMER_TCSR0);           // Se limpia el bit de interrupción ocurrida
        temp |= T0INT;
        iowrite32(temp, pTIMER_TCSR0);



        /******** Envío de la señal al userspace ********/
        memset(&info, 0, sizeof(struct siginfo));
        info.si_signo = SIG_TEST;
        info.si_code = SI_QUEUE;
        info.si_int = timervalue;  // Las señales real-time pueden llevar hasta 32 bits de datos

        rcu_read_lock();
        t = pid_task(find_pid_ns(app_pid, &init_pid_ns), PIDTYPE_PID);
        if(t == NULL){
                printk("No existe el PID.\n");
                rcu_read_unlock();
                return -ENODEV;
        }
        rcu_read_unlock();
        ret = send_sig_info(SIG_TEST, &info, t);    // Se envía la signal
        if (ret < 0) {
                printk("Error enviando al señal.\n");
                return ret;
        }

        return IRQ_HANDLED;
}

// Módulo init
static int __init mod_init(void)
{
        printk(KERN_ERR "Init syscall module.\n");

        // Se registran las interrupciones del Timer
        if (request_irq(IRQ_NUM,irq_handler,IRQF_DISABLED, DEVICE_NAME, NULL))
        {
                printk(KERN_ERR "Not Registered IRQ. \n");
                return -EBUSY;
        }
        printk(KERN_ERR "Registered IRQ. \n");

        pTIMER_TCSR0 = ioremap_nocache(TIMER_TCSR0,0x4);// Mapeo del control and status register
        pTIMER_TLR0 = ioremap_nocache(TIMER_TLR0,0x4);  // Mapeo del load register del Timer 0
        pTIMER_TCR0 = ioremap_nocache(TIMER_TCR0,0x4);  // Mapeo del count register del Timer 0

        iowrite32(0x000000B0, pTIMER_TCSR0);    // Carga del TLR
        iowrite32(0x000000D0, pTIMER_TCSR0);    // Generate mode, downcounter, reload generate

                                                // value, no load, enable IRQ, enable Timer

        // Se utiliza un archivo de configuración de lectura y escritura para el paso de información
        file_debugfs = debugfs_create_file("timer_conf", 0644, NULL, NULL, &syscall_fops);

        return SUCCESS;
}
```

```
// exit module
static void __exit mod_exit(void)
{
        debugfs_remove(file_debugfs);
        iounmap(pTIMER_TCSR0);                  // Se "des-mapean" los registros del Timer 0
        iounmap(pTIMER_TLR0);
        iounmap(pTIMER_TCR0);
        free_irq(IRQ_NUM, NULL);                // Se desregistran las interrupciones
        printk(KERN_ERR "Exit syscall Module. \n");
}

module_init(mod_init);
module_exit(mod_exit);

MODULE_AUTHOR ("Tom");
MODULE_DESCRIPTION("Testdriver for the Xilinx AXI Timer IP core & system calls. Modificado por
Daniel Sánchez Gallego.");
MODULE_LICENSE("GPL v2");
MODULE_ALIAS("custom:syscall");
```

Código A.1: Archivo timer_module.c

A continuación, se muestra el código principal de la aplicación de usuario (**timer_user.c**):

timer_user.c

```
#include <stdio.h>
#include "axi_timer.h"

#define LOAD_VAL 0xE2329AFF     // Valor para cargar al registro de carga del Timer, de modo
                                // que tras ~5 segundos (500000000 ciclos) se  llegue al final
                                // de la cuenta y se lance una interrupcion

int  interrupt_count = 0;

void my_handler() {
        printf("User-space Handler.\n");
        interrupt_count++;
}

int main ( int argc, char **argv )
{
        Timer_Set_Handler(my_handler);  // Se establece el manejador de interrupciones

        Timer_Init();                   // Se inicializa el Timer

        Timer_SetLoadReg(LOAD_VAL);     // Se establece el load register para tener
                                        // interrupciones cada ~5 segundos

        // Se obtienen los valores de los registros
        printf("Control and Status Register = 0x%08x\n", Timer_GetControlStatusReg());
        printf("Load Register = %u\n", Timer_GetLoadReg());
        printf("Counter Register = %u\n", Timer_GetCounterReg());

        while(interrupt_count < 5);     // Se espera a que se lancen 5 interrupciones

        Timer_Exit();

        return 0;
}
```

Código A.2: Archivo timer_user.c

Por último, se muestra el código de la librería de manejo del AXI Timer desde la aplicación de usuario (**axi_timer.h**):

axi_timer.h

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>

#define SIG_TEST 44     // Definición de una señal propia (rango de 33 a 64 para real-time signals)

typedef unsigned int u32;

void (*int_handler) (); // Puntero al manejador de interrupciones de la aplicación de usuario

int  configFile;        // Archivo de comunicación con el dispositivo
char buf[10];           // Buffer utilizado para la comunicación con el kernel
u32  temp;

/****************************** Prototipo de funciones ********************************/
int  Timer_Init();
void Timer_Exit();
int  Timer_SetControlStatusReg(u32 value);
int  Timer_SetLoadReg(u32 value);
int  Timer_SetCounterReg(u32 value);
u32  Timer_GetControlStatusReg();
u32  Timer_GetLoadReg();
u32  Timer_GetCounterReg();
void Timer_Set_Handler();
void Timer_Int_Handler(int n, siginfo_t *info, void *unused);
/**********************************************************************************/
/************************      Implementación de funciones      ************************/

// Inicializa el AXI Timer
int  Timer_Init()
{
        // Se configura la señal para la notificación de interrupciones desde el kernel
        struct sigaction sig;
        sig.sa_sigaction = Timer_Int_Handler; // Timer_Int_Handler() se ejecutará al recibir la señal
        sig.sa_flags = SA_SIGINFO;
        sigaction(SIG_TEST, &sig, NULL);

        // Se abre el archivo de configuración para lectura y escritura
        configFile = open("/sys/kernel/debug/timer_conf", O_RDWR);
        if(configFile < 0) return -1;

        // El kernel necesita saber el PID del la aplicación para enviar las señales de las interrupciones
        buf[0] = '3';                   // El primer carácter indica la operación
        temp = getpid();                // Se obtiene el PID
        memcpy(buf+1, &temp, sizeof(u32));// Se almacena el valor a partir del segundo elemento
        if (write(configFile, buf, strlen(buf) + 1) < 0) return -1;

        return 0;
}

// Cierra el archivo de comunicación con el dispositivo
void  Timer_Exit()
{
        close(configFile);
}

// Establece un valor para el registro de control y estado
int  Timer_SetControlStatusReg(u32 value)
{
        buf[0] = '0';                       // El primer carácter indica la operación
        memcpy(buf+1, &value, sizeof(u32));  // Se almacena el valor a partir del segundo elemento
        if (write(configFile, buf, strlen(buf) + 1) < 0) return -1;

        return 0;
}
```

```
// Establece un valor para el registro de carga
int  Timer_SetLoadReg(u32 value)
{
        buf[0] = '1';                          // El primer carácter indica la operación
        memcpy(buf+1, &value, sizeof(u32));  // Se almacena el valor a partir del segundo elemento
        if (write(configFile, buf, strlen(buf) + 1) < 0) return -1;

        return 0;
}

// Establece un valor para el registro del contador del timer
int  Timer_SetCounterReg(u32 value)
{
        buf[0] = '2';                          // El primer carácter indica la operación
        memcpy(buf+1, &value, sizeof(u32));  // Se almacena el valor a partir del segundo elemento
        if (write(configFile, buf, strlen(buf) + 1) < 0) return -1;

        return 0;
}

// Devuelve el valor del registro de control y estado
u32  Timer_GetControlStatusReg()
{
        buf[0] = '0';                    // El primer carácter indica la operación
        if (read(configFile, buf, strlen(buf) + 1) < 0) return -1;

        memcpy(&temp, buf, sizeof(u32));
        return temp;
}

// Devuelve el valor del registro de carga
u32  Timer_GetLoadReg()
{
        buf[0] = '1';                    // El primer carácter indica la operación
        if (read(configFile, buf, strlen(buf) + 1) < 0) return -1;

        memcpy(&temp, buf, sizeof(u32));
        return temp;
}

// Devuelve el valor del registro del contador del timer
u32  Timer_GetCounterReg()
{
        buf[0] = '2';                    // El primer carácter indica la operación
        if (read(configFile, buf, strlen(buf) + 1) < 0) return -1;
        }
        memcpy(&temp, buf, sizeof(u32));
        return temp;
}

// Establece una función para el manejo de las interrupciones
void Timer_Set_Handler(void *handler)
{
        int_handler = handler;
}

// Manejador de interrupciones de la librería
void Timer_Int_Handler(int n, siginfo_t *info, void *unused) {
        // printf("User-space interrupt! ->  %u cycles.\n", info->si_int);
        if (int_handler != NULL)
                int_handler();                  // Se llama al manejador externo especificado
}
```

Código A.3: Archivo axi_timer.h

Bibliografía

[1] Avnet. (16 de 05 de 2013). *Open Source Linux Ethernet Performance Test Tutorial [Recurso en línea].* Obtenido de http://zedboard.org/support/design/1521/11

[2] brogath. (30 de 07 de 2013). *AR# 50572 (Axi-timer) interrupts in linux [Mensaje en Foro]*. Obtenido de http://forums.xilinx.com/t5/Embedded-Linux/AR-50572-Axi-timer-interrupts-in-linux/m-p/342533#M6553

[3] Jeff Johnson. (4 de 3 de 2014). *Comparison of Zynq boards [En línea]*. Obtenido de http://www.fpgadeveloper.com/2014/03/comparison-of-zynq-boards.html

[4] Peter Jay Salzman. (18 de 05 de 2007). *The Linux Kernel Module Programming Guide [En línea]*. Obtenido de http://tldp.org/LDP/lkmpg/2.6/html/index.html

[5] Phil Dykstra. (s.f.). *Issues Impacting Gigabit Networks: Why don't most users experience high data rates? [En línea]*. Obtenido de http://sd.wareonearth.com/~phil/issues.html

[6] Sarangi, A., & MacMahon, S. (14 de 08 de 2014). *LightWeight IP Application Examples (XAPP1026) [Application Note].* Obtenido de http://www.xilinx.com/support/documentation/application_notes/xapp1026.pdf

[7] Silver Moon. (13 de 01 de 2008). *Raw socket programming on windows with Winpcap [En línea]*. Obtenido de http://www.binarytides.com/raw-sockets-packets-with-winpcap/

[8] Silver Moon. (18 de 12 de 2011). *Code a packet sniffer in C with winpcap [En línea]*. Obtenido de http://www.binarytides.com/code-packet-sniffer-c-winpcap/

[9] Xillinux. (2014). *Xillinux: A Linux distribution for Zedboard, ZyBo, MicroZed and SocKit [En línea]*. Obtenido de http://xillybus.com/xillinux

[10] Xilinx. (s.f.). *Xilinx User Community Forums [En línea]*. Obtenido de http://forums.xilinx.com

[11] Xillybus. (2014). *A Tutorial on the Device Tree (Zynq) [En línea]*. Obtenido de http://xillybus.com/tutorials/device-tree-zynq-1

[12] Zedboard.org. (s.f.). *Zed English Forum [En línea]*. Obtenido de http://zedboard.org/forums/zed-english-forum