Universidad de Alcalá
Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN
INGENIERÍA DE TELECOMUNICACIÓN

Especialidad en Tecnologías Espaciales y de Defensa

Trabajo Fin de Máster

Generación de plataforma SoC sobre OcPoC con integración en petalinux de periféricos personalizados para el funcionamiento y localización del sistema a partir de tecnología de ultrasonidos

Autor: Álvaro Cortés Sánchez-Migallón

Tutor/es: Álvaro Hernández Alonso

2019

2019

UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

Ingeniería de Telecomunicación

Trabajo Fin de Máster

Generación de plataforma SoC sobre OcPoC con integración en petalinux de periféricos personalizados para el funcionamiento y localización del sistema a partir de tecnología de ultrasonidos.

Autor: Álvaro Cortés Sánchez-Migallón

Director: Álvaro Hernández Alonso

Tribunal:

Presidente: Ignacio Bravo Muñoz

Vocal 1º: Óscar Rodríguez Polo

Vocal 2º: Álvaro Hernández Alonso

Calificación: .....

Fecha: .....

A todas las personas que ven el futuro borroso e incierto...

"Mucha gente no progresa en la vida, no por no apuntar a grandes metas y fallar, sino por apuntar bajo y acertar."
Anónimo

Agradecimientos

Detrás de los grandes logros siempre hay esfuerzos que la gente no ve.

Anónimo

Este trabajo es el fruto de muchas horas de trabajo. Horas entre las que se encuentran también las requeridas para poder dar comienzo a este trabajo. Los esfuerzos requeridos por los estudios más el los esfuerzos requeridos por el trabajo han servido para conocerme mejor en situaciones de estrés y no solo a mi, sino a las personas que han permanecido conmigo a lo largo de este tiempo.

Quiero empezar agradeciendo a Álvaro, mi tutor, todo lo que me ha enseñado, no solo la parte técnica que he necesitado para la realización de este proyecto, sino la humildad y bondad que residen en él.

A mi familia, en especial mis padres y hermano, por confiar y aguantar sin prisas y desesperación a que los frutos de tantos años de estudios y trabajo den sus frutos.

A mis antiguos amigos que me han aguantado en la lejanía durante estos tres últimos años, pero sobre todo a los nuevos, que han aparecido en los momentos más dificiles. Con especial mención a Javi y Alex, compañeros del máster, ya que sin su ayuda en las noches de biblioteca y de despejarnos saliendo y haciendo viajes no guardaría tan buenos recuerdos de esta etapa.

A mis compañeros de trabajo, los que se van y los que vienen, de los que he aprendido muchas cosas que no se aprende en la universidad pero se necesitan en el duro día a día.

Por último, a todas aquellas personas que han permitido los avances en las tecnologías y gracias a los cuales he estudiado Ingeniería de Telecomunicaciones y soy quien soy.

Resumen

La velocidad en el avance de la tecnología permite la aplicación de nuevas técnicas a más campos requiriendo personas con amplios conocimientos del funcionamiento de los sistemas y su funcionamiento. La aparición de los SoC's permite flexibilizar los sistemas para adaptarlos a la gran mayoría de aplicaciones, uniendo las ventajas de los sistemas basados en hardware reconfigurable con los recursos de un sistema ASIC.

Para el manejo de esta complejidad de recursos es aconsejable la integración de los recursos hardware disponibles bajo un sistema operativo embebido, optimizado para los mismos. Se requiere por lo tanto un acceso desde el nivel del sistema operativo al hardware para el control de periféricos y módulos generados, para esta tarea se utilizan los device drivers. Un device driver es un software de bajo nivel que permite conectar el kernel del usuario con el nivel hardware.

Una vez el sistema es capaz de acceder al hardware a través de los device drivers, requiere la posibilidad de ejecutar aplicaciones de alto nivel, que permitan al sistema conseguir los recursos requeridos en las especificaciones del sistema. La flexibilidad que se consigue en el sistema con la integración del hardware reconfigurable, el sistema operativo, los device drivers y las aplicaciones de alto nivel los hace muy útiles en prototipado e investigación.

Este proyecto se ha centrado en la generación e integración de un sistema completo. Incluyendo la generación de la base hardware que se requiere utilizar, en la que se integra un periférico hardware de procesado de datos. Un sistema operativo embebido basado en Linux optimizado para el hardware utilizado que permite la gestión de las comunicaciones y recursos disponibles para el control del sistema. La generación de los device drivers necesarios para el control de los periféricos hardware específicos desde el espacio de kernel del sistema operativo.

Por último, la creación de una aplicación de alto nivel que permite la ejecución de un algoritmo de localización a partir de trilateración hiperbólica a partir del procesado hardware de las señales de ultrasonidos recibidas desde una baliza de posición conocida.

Palabras clave: System on Chip (SoC), Embeded Linux(Petalinux), Firmware, Zynq, Multiplatform systems.

Abstract

The speed in the advancement of technology allows the application of new techniques to more fields requiring people with extensive knowledge of the operation of systems and their operation. The appearance of the SoC's allows flexibility of the systems to adapt them to the vast majority of applications, joining the advantages of hardware-based systems reconfigurable with the resources of an ASIC system.

For the management of this complexity of resources it is advisable to integrate the hardware resources available under an embedded operating system, optimized for same. Therefore, access from the operating system level to the hardware for the control of peripherals and modules generated, for this task they are used the device drivers. A device driver is a low level software that allows you to connect the user's kernel with the hardware level.

Once the system is able to access the hardware through the device drivers, it requires the possibility of executing high-level applications that allow the system to achieve resources required in the system specifications. The flexibility that is achieved in the system with the reconfigurable hardware integration, the operating system, the devices High-level drivers and applications make them very useful in prototyping and research.

This project has focused on the generation and integration of a complete system. Including the generation of the hardware base that is required to be used, in which a Peripheral data processing hardware. An embedded Linux-based operating system optimized for the hardware used that allows communications management and resources available for system control. The generation of device drivers necessary for the control of specific hardware peripherals from the space of operating system kernel.

Finally, the creation of a high level application that allows the execution of a location algorithm from hyperbolic trilateration from the hardware processing of the ultrasound signals received from a radio beacon of known position.

Keywords: System on Chip (SoC), Linux embeded (Petalinux), Firmware, Zynq, Multiplatform systems.

Índice general

Índice de figuras

Capítulo 1

Introducción y objetivos del trabajo

1.1 Motivación

El rápido avance de la tecnología degeneración de sistemas junto al aumento de la capacidad de miniaturazación produce en la sociedad un aumento de dispositivos cada vez más complejos utilizados en todos los campos de la industria. Este avance produce la necesidad de ingenieros con las capacidades necesarias para realizar un correcto desarrollo del sistema, desde las especificaciones, diseño desarrollo y llegando a las fases de validación de los mismos.

Este proyecto trata de combinar los distintos conocimientos necesarios para poder hacer que una persona sea capaz de generar un complejo sistema a partir de una plataforma hardware comercial. Se han utilizado conocimientos del campo de la electrónica y el hardware, sistema operativo y programación de aplicaciones de alto nivel para la generación del sistema final. Estos conocimientos junto a una buena metodología de trabajo y control de las herramientas utilizadas han permitido generar e integrar el sistema.

La idea final del proyecto es habilitar a una persona para poder realizar una comprensión completa del sistema, desde los aspectos más tecnológicos hasta aspectos a nivel de proyecto como puedan ser tiempos de desarrollo y posibles problemas en fase de integración y validación.

1.2 Objetivos del proyecto

El objetivo fundamental del proyecto es el diseño, desarrollo, generación e integración de un sistema completo basado en SoC. Incluyendo la generación de la plataforma hardware, un sistema operativo embebido y una aplicación de posicionamiento de alto nivel. A continuación, se explican los objetivos de forma más detallada.

- *Análisis del periférico hardware de adquisición de señal.*

  Análisis del sistema receptor de señal basado en ultrasonidos, incluyendo el sistema de transmisión de señales basados en códigos ortogonales.

- *Análisis del periférico hardware de procesado de datos.*

  Análisis del módulo procesador de datos de ultrasonido, sus interfaces y control para su uso integrado en una aplicación de posicionamiento como parte de codiseño hardware-software.

- *Diseño y desarrollo de la plataforma hardware utilizada.*

  Diseño hardware del sistema y periféricos necesarios para la generación del sistema incluyendo los distintos desarrollos de pruebas de aprendizaje de niveles superiores

- *Diseño y desarrollo del sistema operativo.*

  Diseño, configuración y desarrollo del sistema operativo embebido, basado en Linux para la ejecución de aplicaciones de alto nivel sobre la plataforma hardware previamente diseñada.

- *Diseño y desarrollo de los device drivers del sistema.*

  Diseño y desarrollo de drivers utilizados para la comunicación entre aplicaciones de alto nivel ejecutadas en un sistema Linux y los periféricos hardware incluidos en la plataforma hardware

- *Diseño y desarrollo de la aplicación de posicionamiento.*

  Diseño y desarrollo del flujo de la aplicación y la aplicación para poder determinar una posición a partir de los datos recibidos por el micrófono de ultrasonidos, procesados por el periférico hardware y procesados por el algoritmo software de trilateración hiperbólica.

- *Integración del sistema.*

  Integración de los distintos bloques tecnológicos del proyecto como sistema final pudiendo realizar un uso del mismo para la finalidad seleccionada.

- *Validación de la arquitectura seleccionada.*

  Validación del sistema en conjunto a partir de la validación, individual y en conjunto del sistema generado, con las posibles mejoras y trabajos futuros que realizar sobre el mismo para generar una optimización del uso de sus recursos.

1.3 Descripción de capítulos

La generación de la documentación está ideada para la replicación del sistema, pudiendo a partir de la información de esta memoria y las herramientas adecuadas la generación del mismo. Los capítulos se han escrito desde la tecnología de menor nivel a la de mayor nivel. Iniciando por un capitulo de introducción seguido por un capítulo de *Estado del arte*. Este capítulo comienza explicando los antecedentes de la tecnología utilizada ayudando al lector ala comprensión del punto de partida del proyecto y el uso de la tecnología seleccionada.

Continua con el capítulo de tecnología hardware en el que se explica la tecnología hardware que se ha seleccionado para este proyecto, explicando los recursos que capacitarán al sistema y la explicación de la plataforma hardware generada con los distintos periféricos utilizados.

El cuarto capítulo inicia las explicaciones de la herramienta utilizada para crear el sistema operativo específico para el hardware generado, incluyendo los pasos y comandos utilizados para su configuración y metodología de carga y depuración. Para completar los sistemas de interconexión entre el sistema operativo y el hardware se ha creado el capítulo cinco, en el que se explica la teoría necesaria para poder realizar device drivers con todas sus características y permitir que funcionen de forma correcta. Este es el capítulo más largo y por lo tanto el núcleo del proyecto.

Por último se explica la aplicación de usuario, junto a las comunicaciones requeridas entre los distintos módulos y el flujo que debe tener la misma, terminando con las conclusiones obtenidas a lo largo del proyecto y las posibles opciones de mejora del sistema integrado.

Capítulo 2

Estado del Arte

Para poder comprender la necesidad del trabajo realizado es necesario tener en cuanta la situación del campo de estudio, entendiendo como campo de estudio la relación entre sociedad y tecnología en en ámbito de aplicación. Es por esto por lo que este capítulo trata de ayudar al posicionamiento en tecnologías, con un ameno resumen de la evolución de la tecnología hasta llegar a la tecnología utilizada.

Incluye también un resumen del campo de aplicación y características más importantes del uso de estos sistemas y en qué situaciones tienen ventaja, también la combinación de tecnologías que permiten su uso y cómo realizar una correcta integración desde el punto de vista de diseño y desarrollo.

Por último se entra en el campo de aplicación, será el momento de hablar sobre los sistemas existentes de localización de interiores, desde *RF* llegando a ultrasonidos explicando las ventajas e inconvenientes que tiene cada uno y la relación existente entre ellos y las posibles formas de complementarse.

2.1 Marco tecnológico

La historia de la tecnología, aunque relativamente reciente, es excesivamente larga para incluirla entera por lo que se centrará en la historia reciente de los dispositivos programables, iniciando con los *PLD's* o dispositivos lógicos programables, que son la base del hardware reconfigurable, diseñados para la realización de dispositivos con capacidad de cambiar de funcionalidad en función de los datos almacenados en el mismo.

Al tender la tecnología a la miniaturazación se inicia el uso de *ASIC*, que se trata de un circuito integrado para una aplicación específica, esto ayuda a crear componentes específicos con mayor eficiencia y mucho menos tamaño respecto a los comentados anteriormente, pero no son capaces de cambiar su funcionalidad, por lo que se busca una combinación de los dos.

En el momento en el que aparecen las *FPGA* o array de puertas lógicas programables, se trata de un dispositivo compacto de altas prestaciones que permite la generación de circuitos internos, permite una alta integración de circuitos lógicos programables capaces de ser reprogramados al igual que los *PLD* pero con la potencia de los *ASIC*, es por este motivo por el que se utilizan en los campos más exigentes de la computación e ingeniería.

La evolución de las *FPGA's* produce los *SoC's* o lo que sus siglas significan, *system on chip*. Se trata de una combinación entre microprocesadores y *FPGA's* interconectados de forma interna permitiendo una combinación entre la flexibilidad y potencia de la *FPGA* y la velocidad de depuración que permite el

software de los microprocesadores y su uso en combinación. A continuación, se explica con más exactitud cada uno de los dispositivos comentados hasta el momento.

2.1.1 Sistemas basados en dispositivos lógicos programables, PLD

En el campo de la electrónica y la computación uno de los avances más importantes que ha ocurrido es la aparición de los *PLD's*, se trata de circuitos integrados que permiten una reconfiguración del mismo para poder cambiar su funcionalidad, siendo lo contrario a lo utilizado hasta el momento, la lógica cableada.

Antes de la existencia de los *PLD's* se utilizaban memorias *ROM*, permitiendo lectura únicamente para crear funciones combinacionales en función de tablas de verdad, esta forma de operar tenía una gran capacidad de almacenamiento de resultados y combinaciones de entradas aportando gran capacidad a los sistemas.

Son las desventajas de estas memorias las que generan una necesidad de evolución tecnológica, las desventajas que se detectan como las más importantes son la lentitud de los circuitos lógicos diseñados, los problemas de sincronización entre salidas y entradas de datos sobre las mismas y el alto consumo de estas memorias, ya que aparecen otros, como el uso de una pequeña fracción de las mismas generando un uso ineficiente.

En 1970 se diseñó un *CI* o circuito integrado que incluía los primeros biestables *JK* para la aportación de capacidad de sincronismo y la posible generación de máquinas de estados internas al chip. Este dispositivo es el precursor de las actuales *FPGA's* mucho más potentes y eficientes.

El siguiente paso era aumentar la capacidad de este dispositivo, llegando a la aparición de las *PAL* o arrays de lógica programable, se trata de componentes mucho más pequeños y rápidos, llegando a tener dos decenas de pines aproximadamente y una gran miniaturización respecto a sus precursores, la arquitectura estaba basada en una matriz de puertas lógicas **OR** como interconexión de los biestables internos.

Para aumentar la generalización de los dispositivos aparecen las *GAL*, arrays lógicos programables, con las mismas propiedades de las *PAL* y su capacidad de reconfiguración pero con la capacidad de ser borrado y reprogramado, muy utilizados en la fase de prototipado de los sistemas al poder corregir fallos en un diseño inicial por la reprogramación.

Permite la implementación de cualquier suma de productos con variables definidas a partir de la activación y desactivación de celdas *EECMOS*, dispositivos *CMOS* con borrado eléctrico, conectando las puertas de las capas *OR* o *AND* para obtener la operación requerida. Se trata de este borrado a partir de electricidad lo que permitió que se mantuviesen tanto tiempo en uso y fuesen la base de la tecnología de la época.

Por último aparecen los *CPLD's*, al igual que los *PLD's* son dispositivos integrados cuyas siglas significan dispositivos de lógica programable compleja, están basados en *PAL's* internas entrelazadas por interconexiones programables. Es sobre estos dispositivos sobre los que se estandariza el estándar de programación **JTAG**, *join test action group* para su programación.

La evolución de estos dispositivos genera las **FPGA**.

2.1.2 Dispositivos de aplicación específica, ASIC

Los *ASIC* se definen como circuitos integrados para aplicaciones específicas, se trata de circuitos para un sistema o propósito específico, lo contrario a los *PLD's* que se generan para un ámbito más amplio y

genérico.La composición de estos circuitos habitualmente está definida por uno o varios microprocesadores, comunicados con memorias *ROM* y *RAM*, usadas para almacenar datos de las variables utilizadas y la memoria del programa o aplicación a ejecutar basados en tecnologías *EEPROM* y *FLASH*.

Los *ASIC's* definen la tecnología *Full Custom*, en la que el diseño se hace completamente a medida, los beneficios de este método es que al tratarse de un propósito específico no se añaden elementos innecesarios reduciendo el espacio o tamaño del sistema, a base de mayor tiempo de desarrollo y coste, habitualmente incluyen elementos analógicos específicos como microprocesadores o actualmente *SoC's*. Un *ASIC* tiene un diseño estructurado, con capas lógicas predefinidas, pudiendo llegar a estar hechas a medida, creando conexic s y optimizad

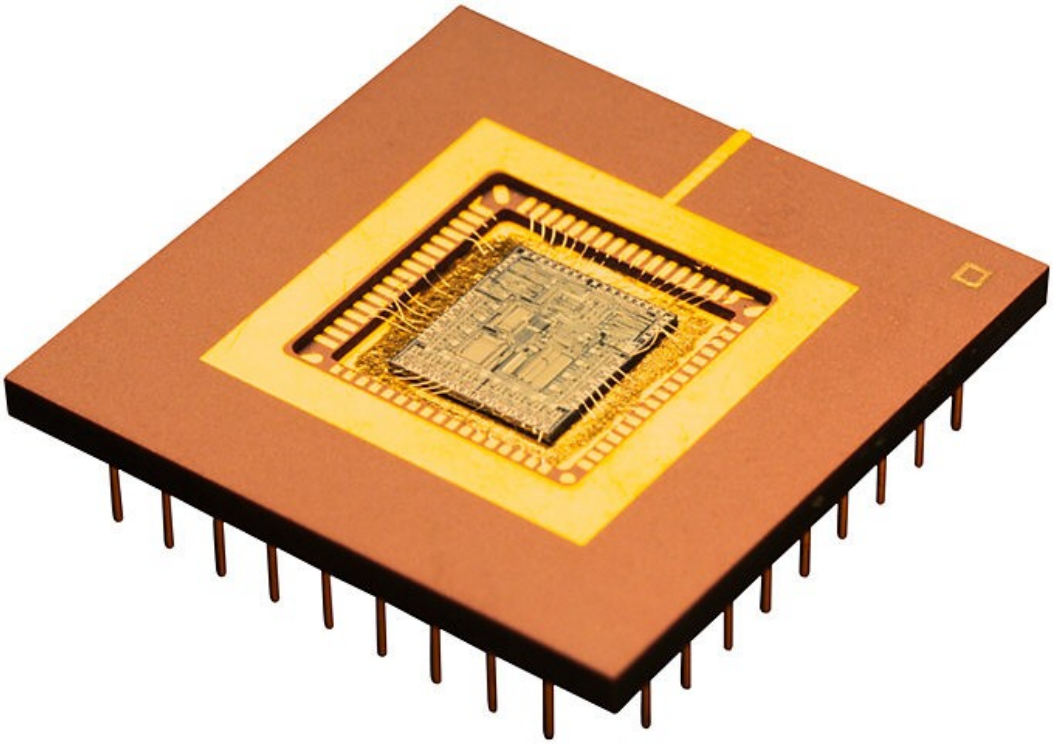

Figura 2.1: Imagen de sistema ASIC [**?**].

Otro aspecto importante en los *ASIC* es que permite el uso de *IP* o núcleos de propiedad intelectual específicos para algunas aplicaciones concretas o segmentos específicos de la industria, se entienden estos *IP* como estructuras predeterminadas que tienen la capacidad de ser incluidos en una estructura genérica. Estos elementos creados a base de primitivas se suelen vender como propiedad intelectual del fabricante con un diseño físico predefinido con su uso como bloques cerrados, se suele entender como diseño de terceros incluidos en un *ASIC*, actualmente genera negocio en distintos sectores de la industria.

2.1.3 Dispositivos FPGA

Las *FPGA's* son dispositivos programables basados en bloques de lógica cuya interconexión y funcionalidad puede ser configurada a partir de lenguajes de descripción hardware especializado, pudiendo ejecutar sencillos circuitos lógicos o en su combinación complejos sistemas de alto rendimiento. Estos sistemas se utilizan en aplicaciones similares a los *ASIC*, teniendo como desventaja la velocidad de funcionamiento y mayor consumo, a pesar de esto, presentan la ventaja de la reconfiguración, reduciendo notablemente sus costes y tiempo de desarrollo.

Como se ha visto, las *FPGA's* son una mezcla entre los *CPLD's* y los *ASIC*, es por ello por lo que es importante tener una visión comparativa entre estos sistemas para poder seleccionar el mejor en cada

caso de aplicación, si se comparan las *FPGA's* a los *PLD's* se llega a la conclusión que las *FPGA's* son *CPLD's* con mucha más capacidad y mucho mayores, ya que sus tecnologías están basadas en puertas *NAND* con la diferencia que en los primeros se tiene una densidad de decenas de miles de puertas y el la segunda entre uno y dos ordenes de magnitud más.

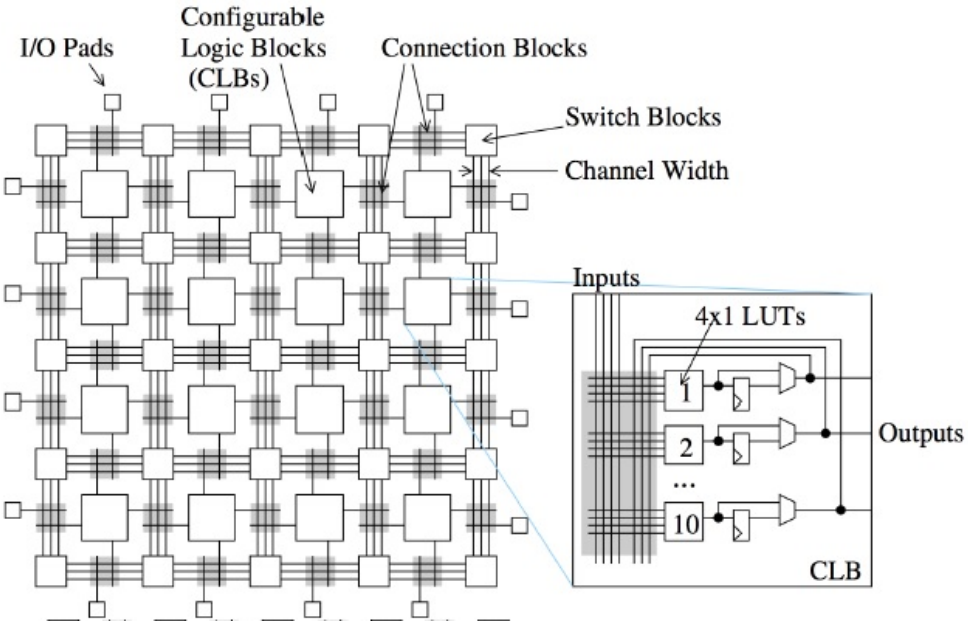

Figura 2.2: Imagen de sistema ASIC.

La diferencia más notable es la arquitectura. La arquitectura de la *CPLD* es más rígida basada en sumas y productos programables, sin embargo, la arquitectura de las *FPGA's* está basada en la combinación de bloques de operaciones lógicas sencillas que cuentan con biestables, aportando la capacidad de sincronismo. Es por la libre conexión de estos bloques por lo que tienen más libertad de diseño y flexibilidad. En las *FPGA's* también se incluyen *IP cores*, de propósito específico que añaden funcionalidades que no pueden tener los *CPLD's*.

Al comparar las *FPGA's* con los *ASIC's* tienen claras diferencias aunque sus campos de aplicación sean prácticamente los mismos, como inconvenientes las *FPGA* tienen un mayor consumo de energía, siendo más lentas y con la capacidad de hacer sistemas menos complejos, teniendo como ventajas la reprogramación, los costes en tiempos de desarrollo y diseño son menores.

Para la programación de estos dispositivos se utiliza habitualmente lenguajes de descripción hardware como puedan ser *VHDL* y *Verilog*, pero en los últimos años se han creado herramientas que permiten la generación de algoritmos de alto nivel, que generan de una forma pseudoautomática este código de bajo nivel para la programación de las mismas, permitiendo ser programadas en *C/C++* con *HLS*, en *Matlab* o *LabView*.

Las aplicaciones en las que se aplican este tipo de sistemas suelen ser los procesadores digitales de señal, sistemas aeroespaciales y de defensa, como prototipado de *ASIC's* o tratamiento de imagen médica, en definitiva tratamiento de señales de alta capacidad con flujos de datos que otros sistemas no son capaces de tratar.

2.1.4 Dispositivos SoC

Por último, se llega al sistema utilizado en este proyecto, los *SoC's* o *System on Chip*, describe la tendencia más frecuente en las tecnologías de fabricación por integrar las partes más importantes de cada uno de los sistemas vistos con anterioridad. Se utilizan para el tratamiento de señal analógica, digital o mixta, incluyendo habitualmente módulos de radiofrecuencia, se utilizan principalmente en sistemas embebidos. La arquitectura que emplean este tipo de sistemas está basada en los elementos mostrados a continuación.

- Módulos procesadores.

  Incluyen microprocesadores, microcontroladores o *DSP's*, estos sistemas están compuestos por procesadores *multinúcleo*, pudiendo denominarse como *MPSoC* o *MultiProcessor System on Chip*.

- Módulos de memoria.

  Los módulos de memoria son distintos en función de su necesidad pudiendo utilizar un tipo o en combinación, *ROM* para acciones de lectura y carga de programas, *RAM* o memorias de acceso rápido, *EEPROM* de acceso solo lectura pero con capacidad de borrado electrónico o *Flash* de muy rápido acceso.

- Generadores de reloj.

  Para cualquier sistema síncrono es importante la capacidad de generación de uno o varios relojes para el control del sistema, para ello se utilizan primitivas basadas en osciladores que permiten controlar aspectos como el *jitter* o la pureza de la señal de salida, a partir de **PLL** *phase locked loops* o **DLL** *selay locked loops*.

- Contadores y resets.

  Los contadores y resets son necesarios en sistemas a tiempo real y para un correcto control del encendido y la activación del sistema permitiendo el arranque en un estado determinado y estable.

- Interfaces de comunicaciones.

  Todos los sistemas son interconexiones de sistemas más sencillos por lo que son muy importantes las interfaces de comunicación, desde una correcta programación hasta una correcta comunicación con memorias u otros sistemas, estas interfaces podrían ser *RS232 (UART)*, *Ethernet*, *SPI*, *IIC* etc.

- Interfaces analógicas.

  Para la mayor parte de aplicaciones es importante recibir o muestrear datos del exterior, para ello se requieren conversores analógico digital, *ADC's* y para poder generar señales conversores digital a analógico *DAC's*, pudiendo generar protocolos específicos de comunicaciones o combinarlos con otros sistemas analógicos para generar señales.

- Reguladores de tensión.

  Todos los circuitos electrónicos requieren de una transformación de energía, desde el punto de entrada o conexión a la *PCB* hasta la utilizada en la mínima, pudiendo tratarse de varias, en una *FPGA* se suelen utilizar varias en función de los periféricos disponibles, pudiendo ser diferentes la alimentación del core, a la de los *QSFP*, los bloques de memoria interna o procesador interno.

Los *system on chip* están ideados para un flujo de desarrollo específico ya que están compuestos por las partes hardware indicadas anteriormente y otra por el software que maneja los microprocesadores, a parte de los puertos e interfaces, para un correcto desarrollo del sistema se debe realizar un diseño paralelo hardware software.

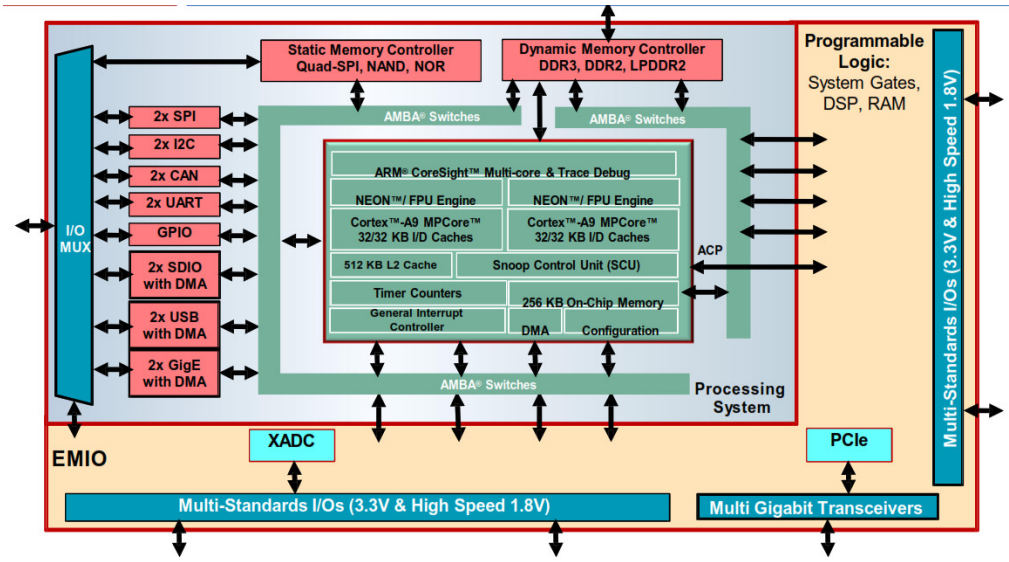

Figura 2.3: Imagen de sistema SoC.

La mayor parte de los *SoC's* necesitan generar un software que controle los distintos periféricos hardware que incluyen el sistema y las comunicaciones entre los mismos. Actualmente la función de los protocolos de Internet es básica al igual que los protocolos *plug and play* basados en *USB* o *RS232*. Un paso clave en el diseño del *SoC* es la emulación, *mapeando* el hardware tal y como sería generado en una *FPGA*, reproduciendo el comportamiento fiel del *SoC* con el fin del testeo final del software.

2.2 Tendencias tecnologícas

2.2.1 Sistemas embebidos

La terminología **Sistemas embebidos** hace referencia a un sistema computacional, diseñado habitualmente para la realización de funciones específicas. La mayor parte de estos sistemas son sistemas en tiempo real, esto quiere decir que son capaces de realizar funciones o actividades en momentos específicos o con una temporización específica.

La mayor parte de estos sistemas son sistemas electrónicos diseñados para existir sobre una placa base, la cual es integrada en un encapsulado mecánico específicamente preparado para el soporte de unas condiciones de trabajo específicas. Estos sistemas se utilizan comúnmente en nuestra sociedad, pudiendo tratarse desde un sistema de seguridad, para la apertura de puertas de un hospital, hasta un pulsador de un semáforo, existiendo un amplio rango de aplicaciones y complejidades soportadas por los mismos.

Por lo general para poder definirse como sistemas en tiempo real soportan una aplicación baremetal, o un sistema empotrado basado en *GNU* como puedan ser la mayor parte de las distribuciones de Linux para sistemas embebidos. Una aplicación baremetal es una aplicación que corre sobre un procesador, habitualmente diseñada y desarrollada en lenguajes de bajo nivel como puedan ser ensamblador, raramente utilizado por su extenso tiempo de desarrollo respecto a la mejora en recursos obtenidos para estos sistemas. Los lenguajes como puedan ser *C/C++*, aportan grandes ventajas como un uso común para muchos tipos de sistemas, rápido, eficiente y estable.

Son los dispositivos embebidos de alta complejidad o rendimiento los que requieren un sistema operativo embebido, al ser necesaria la automatización de ciertas tareas y control de recursos para la centra-

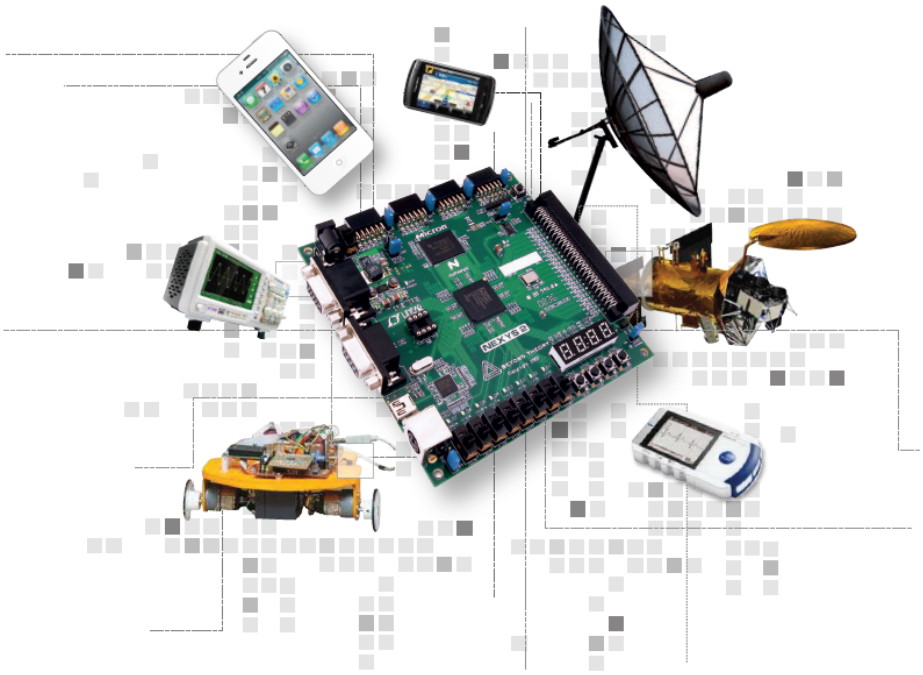

Figura 2.4: Imagen de sistemas embebidos.

lización en las tareas objetivo de la aplicación. Estos sistemas operativos están basados en Linux, estos tienen la capacidad de aumentar su eficiencia siendo generados para un hardware específico con unos recursos específicos o limitados. Estos sistemas operativos se han de ver como un software específico para la gestión de recursos y aplicaciones que ejecutar en sistemas *multiprocesador*, permitiendo la ejecución de aplicaciones en tiempo real de forma autónoma respecto al hardware sobre el que se ejecutan.

Los componentes de un sistema embebido están definidos por la funcionalidad del sistema, habitualmente definidos por los ingenieros de sistemas, que han de conocer los requisitos y especificaciones de sistema para una vez se definan que sean los ingenieros de diseño y desarrollo los que lleven a cabo el sistema participando con los de sistemas y los de pruebas o calidad. Aún pudiendo tener objetivos muy distintos, siempre son sistemas que necesitan comunicarse, ya sea entre ellos o con el exterior, necesitando por lo tanto buses de comunicaciones como puedan ser *RS-232*, *SPI*, *IIC*, *Etherner*, *4G* o la tecnología seleccionada, surgiendo en ocasiones la necesidad de interacción con seres humanos, por lo que requieren las interfaces necesarias para estas acciones con pantallas, teclados u otro tipo de sensores.

Habitualmente tienen interfaces analógicas y digitales para la interconexión con otros sistema o la transmisión y recepción de señales analógicas, pudiendo controlar desde periféricos o sensores sencillos hasta motores complejos. Al tratarse de sistemas en tiempo real requieren una serie de parámetros de reloj especifico, estos, son generados por el oscilador de referencia del sistema, a partir del cual se extraen los relojes necesarios para las comunicaciones y sincronismo del sistema.

Los procesadores o *multiprocesadores* tienen una de las capacidades más importantes del sistema, son los sistemas encargados del control de los periféricos y el procesado de la información, para luego su posterior transmisión o actuación en consecuencia, pudiendo equipararse al cerebro de un ser humano, se trata de la parte más compleja ya que se interconexiona con el software a través del firmware creado para el mismo. Por último, estos sistemas montan componentes activos por lo que requieren una alimentación específica pudiendo requerir distintos tipos de alimentaciones y a distintos niveles, es por esto por lo que estos sistemas requieren unas capacidades de potencia específicas muy importantes y habitualmente

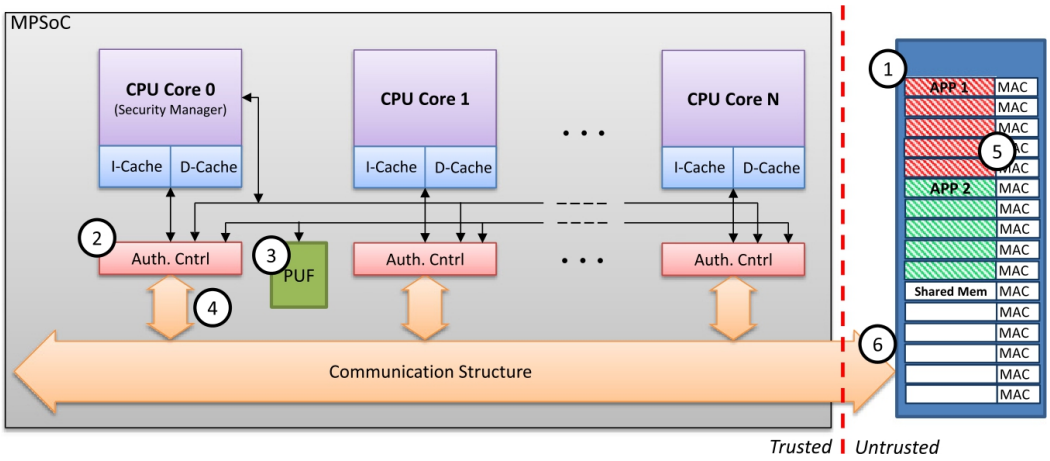

Figura 2.5: Estructura interna de un MPSoC.

redundadas ya que sin este módulo no se podría utilizar el sistema.

2.2.2 Herramientas avanzadas de diseño hardware

Al igual que avanzan las tecnologías utilizadas, avanza la forma de diseñar y desarrollar un sistema, gracias en muchos aspectos a la complejidad de las herramientas utilizadas para el diseño de estos sistemas. En un inicio, cuando la tecnología electrónica se encontraba en sus primeras etapas de desarrollo, los sistemas se desarrollaban a partir del hardware disponible, por lo que se tenía que realizar un desarrollo basado en un diseño verificado desde las primeras etapas de desarrollo, haciendo de los mismos, sistemas muy costosos en dinero y tiempo de desarrollo. Esto comenzó a cambiar con los primeros sistemas basados en hardware reconfigurable, esta característica de reconfiguración del hardware o reconfiguración de los sistemas aportó un nuevo punto de vista en el trabajo, permitiendo una validación de un hardware estable, solapando la fabricación del hardware con las etapas de diseño firmware y software que se precargaba en memorias para un futuro uso. La evolución de estas tecnologías nos ha llevado a sistemas capaces de configurarse por partes y reconfigurarse desarrollando unas capacidades completamente distintas al sistema existente antes de la reconfiguración, es por ello por lo que ha cambiado la forma de afrontar los problemas de diseño y las herramientas utilizadas.

Los *SoC's* o *system on chip* son la combinación de sistemas basados en procesadores a los que se le ha añadido prácticamente como un periférico un apartado de hardware reconfigurable que precargar para su uso y recargar si es necesario para una adaptación del sistema manteniendo los requisitos de velocidad y exigencia necesarios. Esta combinación crea un paradigma de diseño hardware, firmware y software que combina distintas técnicas de desarrollo para flexibilizar los sistemas manteniendo la eficiencia. Estas técnicas son habitualmente nombradas como *codiseño hardware-software*.

El **codiseño** *hardware-software* es el paradigma que aparece a la hora del diseño de un sistema complejo, en el que se tiene que tomar decisiones importantes como puede ser la funcionalidad de cada módulo o la localización de cada funcionalidad en función de las tecnologías disponibles a nivel *harware* o *software* en función del tiempo de desarrollo de la misma. Es por este motivo por lo que para realizar este tipo de labores se requieren profundos conocimientos sobre el sistema y las capacidades disponibles de cada una de las tecnologías utilizadas. Son las herramientas de diseño *hardware* y *software* las que permiten una ayuda en este tipo de diseños.

Hasta este punto, el diagrama de diseño clásico ha estado basado en el *diagrama en V* basado en

fases en las que se comienza un diseño a partir de la petición de un cliente, generado requisitos y fases de desarrollo, con un diseño de la *arquitectura hardware-software* definiendo el tipo de sistema y los elementos hardware específicos a utilizar en el mismo. El desarrollo *hardware-software* se inicia en este momento con la implementación y la integración del hardware y software generado. Son también necesarias pruebas que verifiquen que el sistema funciona correctamente respecto a los requisitos definidos en un inicio del proyecto, teniendo como último paso la validación del sistema respecto al cliente.

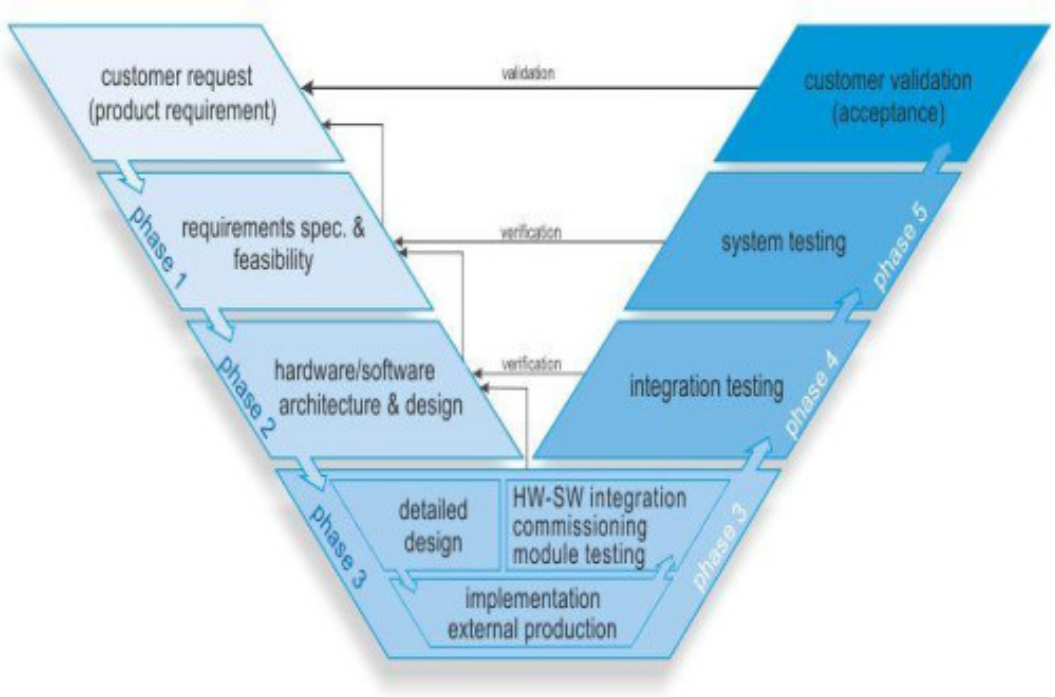

Figura 2.6: Diagrama de diseño y desarrollo clásico en V.

Los *SoC* tienen una gran versatilidad para la definición de sistemas al combinar la parte de *ASIC* y la parte de *hardware* reconfigurable, es por ello por lo que se puede iniciar el desarrollo del sistema solapado a la definición de las últimas características realizando complejos sistemas de diseño. En la siguiente imagen se muestra un ejemplo, conseguido gracias a la gran flexibilidad aportada por el *hardware* reconfigurable, de mover entre el *hardware* y el *software* ciertas funcionalidades permitiendo desarrollar y probar en *software* migrando más tarde a *hardware*. Esto posibilita la reducción del tiempo de puesta en mercado. La posibilidad de mover funcionalidades entre *hardware* y *software* puede ayudar en el balance de recursos como consumo, capacidad de ocupación del sistema y temporización, rendimientos exigidos y aumento de la fiabilidad.

Se podría resumir un diseño *hardware-software* con una especificación inicial de los requisitos del sistema respecto a su interfaz con el exterior, modelando las características de funcionamiento en internas y flexibles o externas y fijas, a partir de una exploración del espacio de diseño. Esta exploración permite realizar una correcta elección del particionado *hardware-software*, al tener el particionado realizado, se inicia un diseño con la síntesis y optimización de los distintos módulos hardware y software con las interfaces de comunicación para terminar el diseño con la versificación, emulación e implementación del mismo.

La parte más compleja del diseño se encuentra en la toma de decisión del particionado, esto viene dado en función de la complejidad del algoritmo a implementar y de la situación de los recursos necesarios para su ejecución. Es por lo que en la práctica se realiza un estudio de diferentes arquitecturas para preveer la

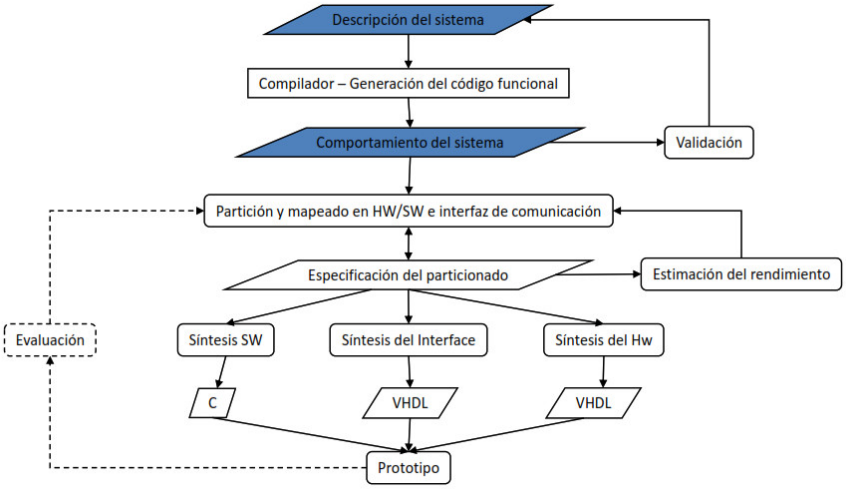

Figura 2.7: Esquema de flujo de codiseño hardware-software

arquitectura final de uso en función de las necesidades del algoritmo, formando parte de la optimizan del mismo, por ello es muy importante la selección de que elementos se compone y los recursos utilizados por cada uno. Esta será la tarea del diseñador, seleccionanado la tecnología utilizada por cada uno, incluye los apartados que se deben ejecutar a nivel *hardware* y *software*. En este proyecto se ha integrado un algoritmo de generación *hardware-software* para paralelizar procesos y optimizar recursos, manteniendo la flexibilidad y velocidad de desarrollo.

En la actualidad, estas herramientas de ayuda en el diseño electrónico tratan de conseguir un correcto particionado *hardware-software* de forma automática dividiendo los algoritmos en función de los recursos utilizados por las operaciones del mismo y a partir de métricas temporales. Este proceso se lleva a cabo a partir de la reiteración de combinaciones de **síntesis** (*análisis del hardware*) y **compilaciones** (*análisis del software*) sin tratarse de un método realmente ejecutivo y optimizado. Se definen tres variables de optimización, *tiempo*, *área* y *comunicación*, basados actualmente en conjuntos de librerías de uso y buses de comunicación estándar. Esta comunicación no se debe despreciar por ser capaz de afectar al sistema con hasta un 50 % del consumo total de la energía y un alto porcentaje del tiempo de los procesos en transferencias de datos, apareciendo así la definición de *NoC* o *Network on chip* que se trata de un *SoC* complejo en el que los recursos hardware se comunican como sistemas independientes a través de buses de comunicaciones, todo ello interno al sistema.

2.3 Localización en interiores

Los sistemas utilizados para la localización y posicionamiento en interiores son denominados como *LPS*, este es un amplio campo de estudio. Esta localización se divide ente localización en interiores y exteriores, dejando como sistema universal de posicionamiento en exteriores al sistema *GPS*, siendo muy difícil de utilizar en interiores por este motivo se han tenido que investigar nuevos métodos de localización en interiores.

Las aplicaciones de uso de estas tecnologías han aumentado con el aumento de la tecnología, pudiendo utilizarse en sistemas de rescate y emergencias, como pueda ser el guiado de personas en un edificio incendiado, posicionamiento de personas dentro de una empresa o el posicionamiento de robots autónomos en los interiores de un almacén.

Estos sistemas pueden proporcionar servicios en entornos inteligentes. Un entorno inteligente es el

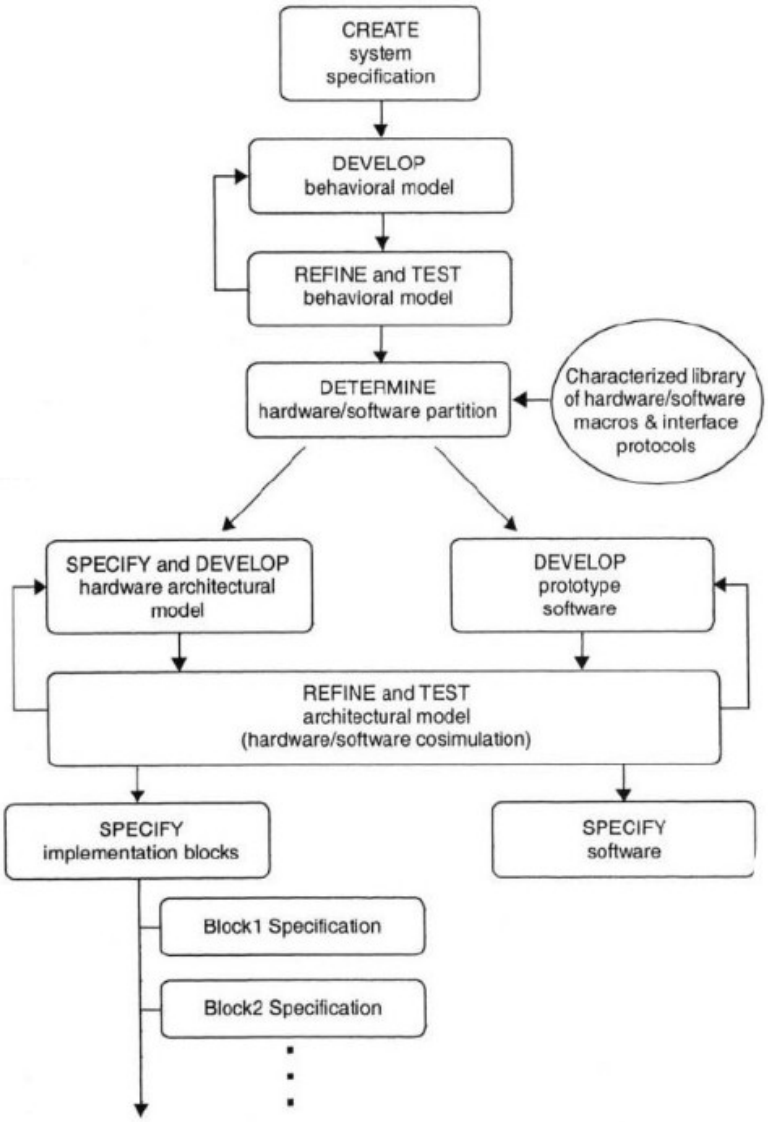

Figura 2.8: Flujo práctico de codiseño hardware-software

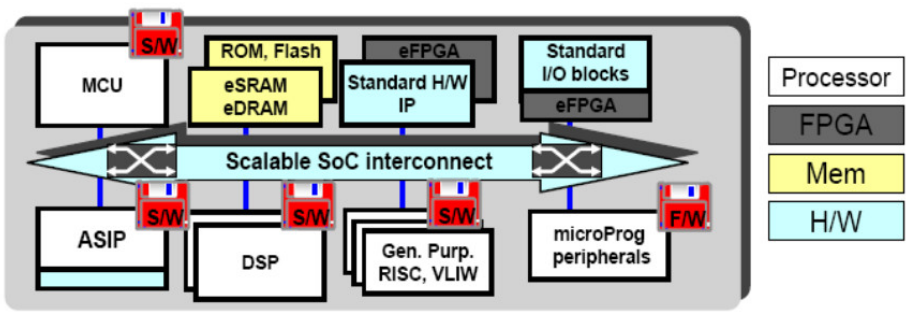

Figura 2.9: Imagen de sistema Network on chip

nombre que se le da a un espacio en el que cohabitan una multitud de sensores que permiten a un conjunto de máquinas conocer su localización y los posibles obstáculos que los rodean.

Estos sensores pueden variar en función de las técnicas de uso, pero los más habituales, están basados en técnicas de posicionamiento basadas en láser, ultrasonidos, cámaras de visión espacial o radiofrecuencia, con su combinación se puede crea un sistema inteligente capaz de navegar por un entorno desconocido, consiguiendo con técnicas de *SLAM* llegar desde un sitio desconocido a una localización determinada por el usuario, de forma automática, sin conocer a priori el camino incluyendo la evasión de obstáculos.

Estos sistemas de localización en interiores están experimentado un aumento muy importante en la tecnología utilizada ya que hay muchas empresas trabajando en ellos, mejorando la tecnología día a día. Al tratarse de sistemas muy versátiles se pueden utilizar en distintas actividades, como puedan ser grandes almacenes, para organizar paquetes, muy utilizados en empresas de envíos, para aparcamientos, en los que permite y saber en que punto se está y cuál es la plaza a la que se pretende llegar o campus académicos como puede ser el de universidades de España que empiezan a tener vehículos autónomos. No solo se trata de la movilidad de objetos ya que en el ámbito más social, se pueden utilizar en localización en hospitales, centros comerciales, aeropuertos, turismo y visitas guiadas a museos.

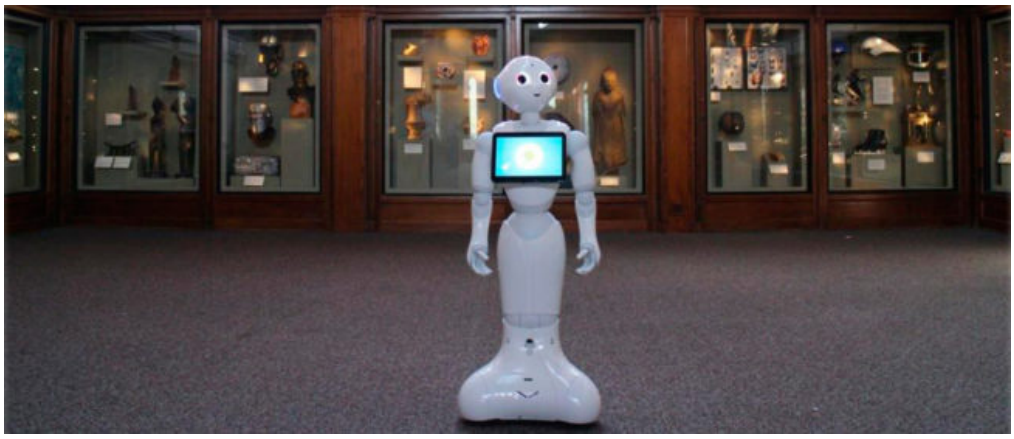

Figura 2.10: Pepper, robot guía.

La tecnología de localización en interiores basada en radiofrecuencia es una de las más utilizadas, ya que se aplica a partir de los sistemas de comunicación, integrados en todos los sistemas móviles de la actualidad. Dentro de esta tecnología de *radiofrecuencia* se utiliza la comunicación *WIFI* que permite en función de la amplitud de señal, velocidad de transmisión y tasa de errores saber la distancia respecto a un emisor, siendo la combinación de señales de estos emisores la magnitud que permite realizar una localización aproximada. Uno de los problemas que suele tener es que la variación de la amplitud puede estar relacionada con el espacio y los objetos cercanos como muros o personas que absorben la energía pudiendo falsear los datos recibidos. En combinación también se utilizan señales de *Bluetooth*, *RFID* para una localización de mayor percepción.

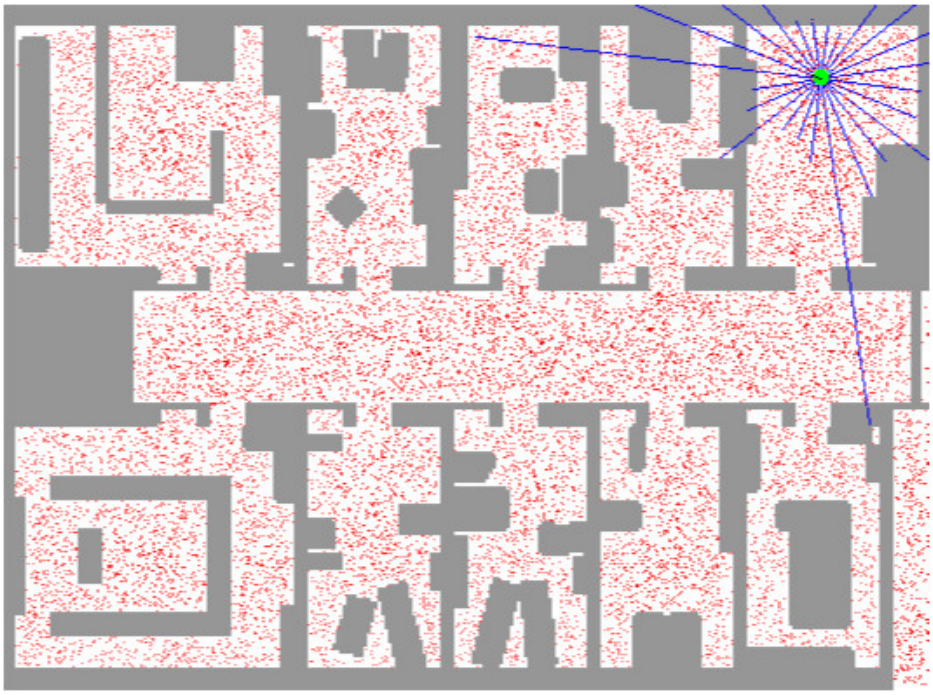

Figura 2.11: SLAM.

Para una localización más precisa se tiene a utilizar otro tipo de tecnologías, como pueden ser las tecnologías basadas en luz, como puedan ser los *infrarrojos* y el *lidar* o las tecnologías basadas en *ultrasonidos*, como son los *sonares* que integran los sensores de aparcamiento de los coches. Es la combinación de este tipo de tecnologías las que se permite una mayor eficiencia en interiores, en combinación con algoritmos de *SLAM*, que permiten una generación del mapa en el que se encuentra el dispositivo, a la vez que se localizan en el mismo gracias al conocimiento previo del mapa.

Estos sistemas se pueden diferenciar en dos tipos, los que realizan un procesado interno de los datos que reciben a través de los sensores que integran o los que son guiados a partir de una máquina procesadora de información que proviene de un entrono *inteligente* basado en sensores. En este proyecto los emisores o balizas de *radiofrecuencia* se han instalado en el espacio externo, de forma fija conociendo sus posiciones, siendo el sistema móvil o dron, el encargado de recibir esta información procesándola para poder realizar una correcta localización.

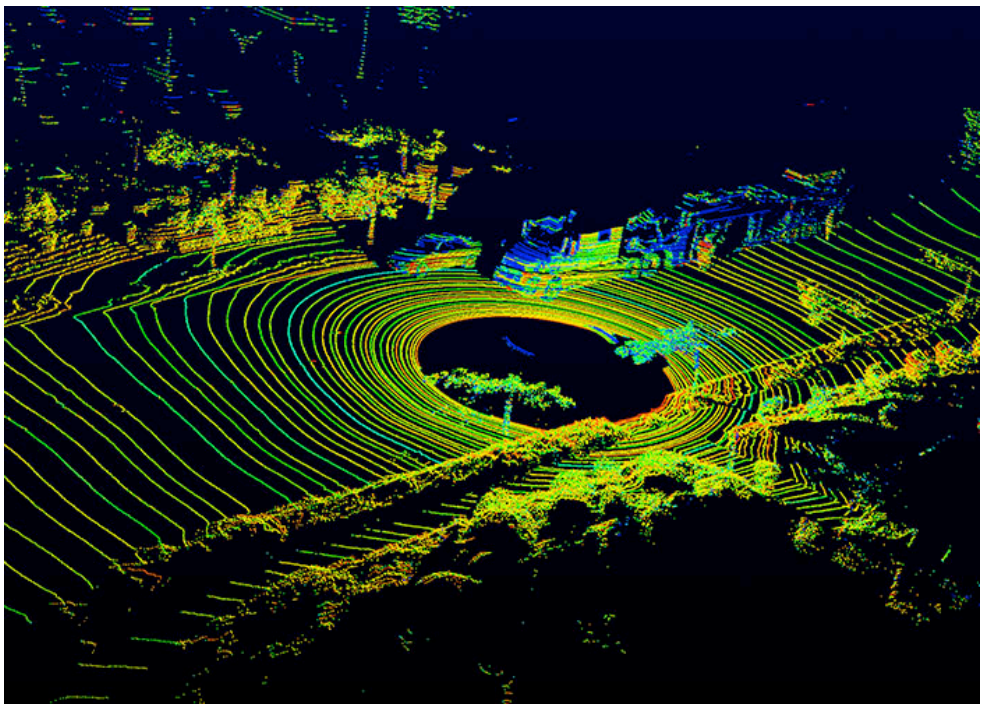

Figura 2.12: Mapeo con ledar velodyne.

Capítulo 3

Arquitectura Hardware

En el apartado de arquitectura hardware se pretenden explicar o justificar los motivos del uso del sistema seleccionado para realizar la aplicación, explicando los motivos de uso de la pieza seleccionada con sus características más notables y el porqué de su necesidad, la distribución del *block design* y los motivos de los elementos que lo componen.

También se explicará el periférico hardware de procesado de datos instanciado en el sistema y los porqués de la arquitectura respecto a la aplicación y justificando el uso de el módulo correlador para realizar a nivel hardware parte del software como codiseño *hardware-software* del algoritmo, incluyendo posibles partes del algoritmo que al realizarlas a nivel software se podrían ejecutar más rápido.

3.1 Hardware comercial

La plataforma sobre la que se se ha trabajado en este proyecto es la tarjeta de evaluación de la empresa comerciante de *FPGA's Xilinx*, esta ha sido utilizada por integrar un *SoC* de la *familia 7*, la **Zynq 7000**, cuyas características y recursos internos se explican en el apartado de plataforma hardware. Se ha utilizado esta tarjeta por ser la utilizada en un sistema de localización para interiores, basado en señales de ultrasonidos, el cual, se pretende integrar para el uso en localización de drones en interior.

Este dron es comercializado por la empresa *Aerotenna* junto a otros drones y sistemas embarcados. El controlador de este dron incluye una *Zynq* como la de la tarjeta *ZedBoard* con otros controladores externos, muy parecidos al sistema generado. El nombre de la plataforma que realiza las funciones de controlador es la plataforma **OcPoC**, integrada en un *cuadricoptero* en el cual realiza y gestiona el control de las comunicaciones, los motores y los sensores para hacer posible su manejo a distancia.

Este sistema integra un revolucionario controlador de vuelo basado en SoC, sus siglas, **OcPoC**, significan *Octagonal Pilot on Chip*. Se trata de una plataforma de código abierto, ideada para el diseño y desarrollo de distintos tipos de sistemas, con capacidades muy mejoradas respecto a su potencia de procesamiento y capacidad de control de sus salidas y entradas. Esta plataforma incluye pines de *PWM* con los que se pueden controlar los motores, *PPM* y *GPIO's*, completamente programables para su interconexión con distintos sensores y su capacidad de conexión con periféricos como son el *GPS*, cámara *CSI* o la tarjeta *SD* que almacenará el sistema operativo.

Las características más importantes de este controlador es que es el primer controlador basado en los *SoC* de la empresa de *FPGA's Xilinx*, su doble núcleo *FPGA + Dual ARM Cortex A9*, sus más de 100 entradas y salidas, internamente con figurables y compatible con *APM* y *PX4*.

Figura 3.1: Controlador OcPoC.

Por tratarse de un sistema basado en código abierto, lo hace muy llamativo para desarrolladores e investigadores ya que permite su modificación y la generación rápida de algoritmos de prototipado para el uso del dron. En este proyecto, se ha pretendido aprender y conocer todos los recursos necesarios para poder realizar una integración de un módulo hardware sobre el sistema operativo utilizado por el controlador para poder ejecutar un algoritmo de posicionamiento en interiores, basado en señales ultrasónicas.

Idealmente el controlador integra los requisitos mínimos para la ejecución de vuelos, integrando sensores *IMU* con sistemas y algoritmos de localización. La potencia de esta tarjeta reside en la gran cantidad de *GPIO's* que tiene comparados con otro tipo de controladores, lo que permite su interconexión a muchos sensores y buses de comunicación. Esta *FPGA* puede incluir un sistema operativo Linux sobre el cual se puede ejecutarla aplicación del controlador junto a otras más, aumentando la flexibilidad del sistema.

Este sistema tiene un equipo de personas trabajando en la actualización de los paquetes de software y sistemas operativos a sacar para ampliar el tipo de plataformas, recursos y capacidades de desarrollo disponibles para los clientes. Actualmente se ha probado con el sistema operativo de Linux utilizado en los *SoC's* para *Zynq*, *Petalinux*, el cual se explica más tarde y una distribución de *Ubuntu 14.04* sobre la que se ejecuta *ROS* de código abierto. Se trata de una gran opción pero no llega a ser tan sencillo como se idea ni los paquetes y sistema operativo cuadran con los documentos disponibles en la página web y en el GITHUB del proyecto.

La idea de la generación de este sistema y su integración con el periférico de procesado de señal para interiores se apartó de los objetivos finales del proyecto al encontrar en el curso del mismo muchos problemas en la generación del software y sistema operativo tal y como se indica en la documentación del sistema. Esta documentación fue mejorada una vez se consiguieron los nuevos objetivos del proyecto pasando al apartado de trabajos futuros.

Figura 3.2: Dron cuadricoptero controlado por OcPoC

3.2 Plataforma hardware

La plataforma hardware utilizada en este proyecto, sobre la cual se ha creado el Linux de sistema operativo que integra, con los periféricos, comunicaciones y aplicaciones que ejecutan es una **Zynq SoC**. Esta pieza es comercializada por el fabricante de FPGA's Xilinx, incluyéndola dentro de la *familia 7* de piezas, las cuales inician la utilización de la herramienta *Vivado* como herramienta de ayuda en el diseño electrónico en vez de la herramienta de *ISE*, utilizada por las piezas de *familia 6* o anteriores.

Como se ha explicado en temas anteriores, un SoC es un sistema creado a partir de la combinación de un microprocesador y un sistema de hardware programable, habitualmente formado por una *FPGA*. Este SoC integra un *Dual ARM Cortex A9* como procesador, con un doble nucleo, este sistema integra las características necesarias por un microcontrolador, ya que no incluye únicamente el procesador sino las memorias caché y los bloques de memoria RAM. En este tipo de sistema se definen dos partes bien diferenciadas, la parte del microcontrolador que se denomina como *PS* y la parte de hardware reconfigurable y lógica programable nombrada como*PL*.

La parte del **PS** o *processor system* es la que engloba al microcontrolador, es decir, el conjunto entre el microprocesador y los distintos periféricos que requiere para su funcionamiento, estos periféricos se pueden englobar en distintos bloques. El bloque de core más céntrico estaría formado por el *Dual ARM Cortex A9*, microprocesador de 32b, combinado con dos **FPU** o *unidades de coma flotante*, una unidad de memoria caché de *512kB*. Las comunicaciones de la PS han de ser múltiples al tratarse de una pieza con muchos periféricos y posibles usos de aplicación final es por ello por lo que tiene buses *AXI* de comunicación con las memorias externas de carga, *QSPI*, *DDR2* y *DD3* y los distintos buses de comunicación estándares en la electrónica moderna, como *SPI*, *IIC*, *UART* o *USB*. Para las comunicaciones de alta velocidad se tiene el bus *DMA* interconectado a *2GbE* y desde la parte de la *PL* a *PCIe*.

La arquitectura interna del procesador se descompone en los procesadores que utilizan una arquitectura de set de instrucciones basada en *ARMv7*, basado en este diseño incluye una unidad de gestión de memoria incluyendo también una unidad de protección de memoria como soporte de microprocesador en

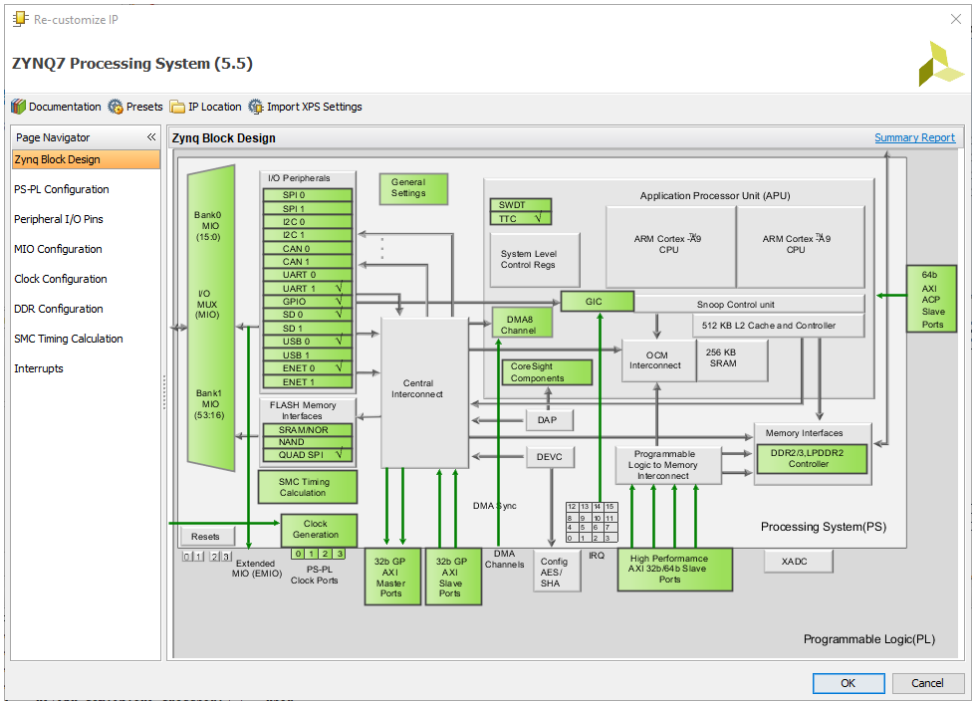

Figura 3.3: Esquema de recursos del sistema.

tiempo real. Incluye dos tipos de set de instrucciones en *16b* ó *32b* con capacidad de multidatos para una misma operación. Las comunicaciones basadas en una arquitectura de bus *AMBA* soportan el protocolo *AXI3* y *AXI4*.

Las memorias caché de primer nivel ofrecen una capacidad operación de *2.5DMIPS/MHz* con hasta una frecuencia de transmisión de *800MHz*, la arquitectura interna del microprocesador en función de la combinación de sus *buses de datos e instrucciones de 64b* separados indica que es una arquitectura Harvard soportando operaciones en *Little endian*. Las memorias caché de primer nivel tienen una capacidad de *32KB de datos* y *32KB de instrucciones* con distintas disposiciones de asociación.

La comunicación entre la **PS** y la **PL** es muy importante ya que será la unión de estos dos sistemas. Esta comunicación está basada en el *protocolo AXI*, en función de los requisitos de las comunicaciones e interconexiones que se generen con los distintos elementos se utilizará el *bus AXI* de alta eficiencia, configurable en ancho entre *32b* y *64b* que comunica el procesador con las memorias *DDR* los conversores de analógico a digital (*ADC's*), incluidos en la placa el módulo *AFI* de interconexión *AXI-FIFO* para largas transferencias de memoria. Respecto al *bus AXI* de propósito general incluye dos *master* desde la parte **PS** a la **PL** con un ancho de dato de *32b* y capacidad de conversión y sincronización entre los distintos dominios de reloj.

También es importante incluir el módulo controlador de interrupciones, que gestiona y habilita las distintas comunicaciones, incluidas por los buses de comunicaciones, periféricos o los posibles módulos incluidos en la parte **PL** del sistema. La parte de la **PL** puede variar en recursos en función de la pieza seleccionada, en este caso se trata de una **XC7Z020** que incluye como lógica programable una *Virtex 7* con sus características de recursos.

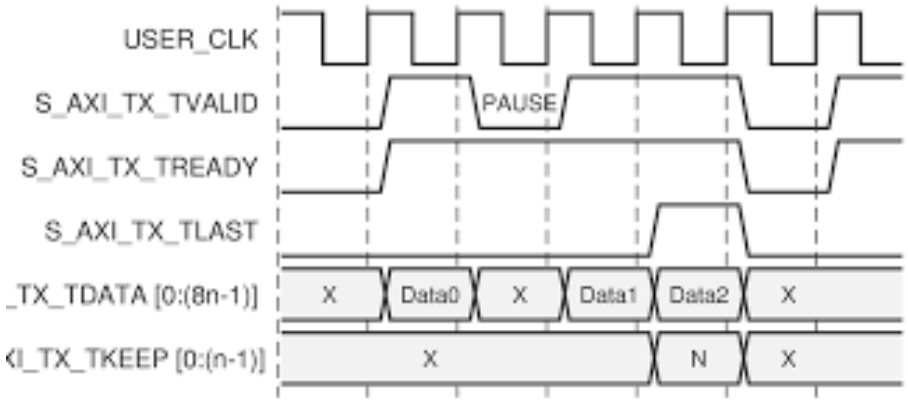

Figura 3.4: Trama de protocolo AXI.

3.2.1 Algoritmo de posicionamiento hardware-software

Este tipo de sistemas se han impuesto a otros por la flexibilidad, versatilidad y potencia que se consigue con la unión de la parte de procesador y la parte de *hardware reconfigurable*, permitiendo diseños mixtos en los que la parte *hardware* y la parte *software* pueden convivir. Esto se utiliza en ciertos campos de aceleración de cálculos sobre algoritmia para poder crear sistemas finales de alta capacidad de computación.

Estos sistemas requieren capacidades inviables a nivel software por tiempos y cantidad de datos. Este motivo hace necesaria la aceleración de algoritmos a nivel *hardware*, permitiendo paralelizar operaciones realizando ejecución simultanea de operaciones independientes.

En este proyecto se ha utilizado en la aplicación de usuario final, en ella se recogen muestras recibidas por un sensor de ultrasonidos llevadas hasta el espacio de usuario del sistema operativo. Una vez la aplicación tiene los datos debe calcular la correlación entre el array de *10 000* muestras adquiridas por el sensor con cinco códigos distintos.

Este algoritmo a nivel *software* ha de hacer cada una de las *correlaciones* de la matriz de datos con la matriz de código de forma consecutiva es decir, utilizando la potencia de un procesador ha de realizar todas las operaciones, sin poder realizar ningún tipo de paralelización o aceleración. El *codiseño hardware-software* permite llevar este conjunto de operaciones a nivel *hardware* pudiendo realizar operaciones en paralelo aumentando en gran medida el tiempo de procesado de señal.

Para una validación inicial del sistema es conveniente recrear todas las operaciones con un lenguaje de alto nivel y entorno de desarrollo como pueda ser *Matlab* o *C/C++* y un *IDE* que permita un desarrollo parcial con puntos de espera. Al validar el funcionamiento se inicia la implementación decidiendo que tipo de operaciones se han de realizar en cada entorno. Para la generación de módulos hardware a partir de fuentes escritas en *C/C++* se utiliza la herramienta de ayuda al diseño electrónico de *Xilinx* llamada *Vivado HLS*. Esta herramienta permite la compilación de fuentes en *C/C++*, su simulación para la observación de posibles errores y depuración del código del algoritmo. La gran ventaja que ofrece esta herramienta es la generación de *IPcores* a nivel hardware con la misma funcionalidad de las fuentes previamente depuradas, siendo optimizadas y devolviendo los parámetros de trabajo del mismo.

Con la funcionalidad optimizada a nivel hardware se puede incluir en un *block design* a partir del cual interconectarlo de forma automática con el procesador para realizar la comunicación entre el módulo y el procesador para la transferencia de datos y ordenes que permiten su uso desde la aplicación de usuario. Este ha sido el método previo de validación del módulo hardware integrado en el proyecto.

3.3 Sistema generado

En este proyecto se han generado distintos escenarios para poder realizar un correcto estudio sobre los device drivers, es por ello por lo que se han introducido distintos elementos no utilizados en el proyecto final que si se han controlado desde la aplicación de usuario del sistema operativo. A continuación se muestra el *block design* generado con los distintos módulos utilizados en el proyecto.

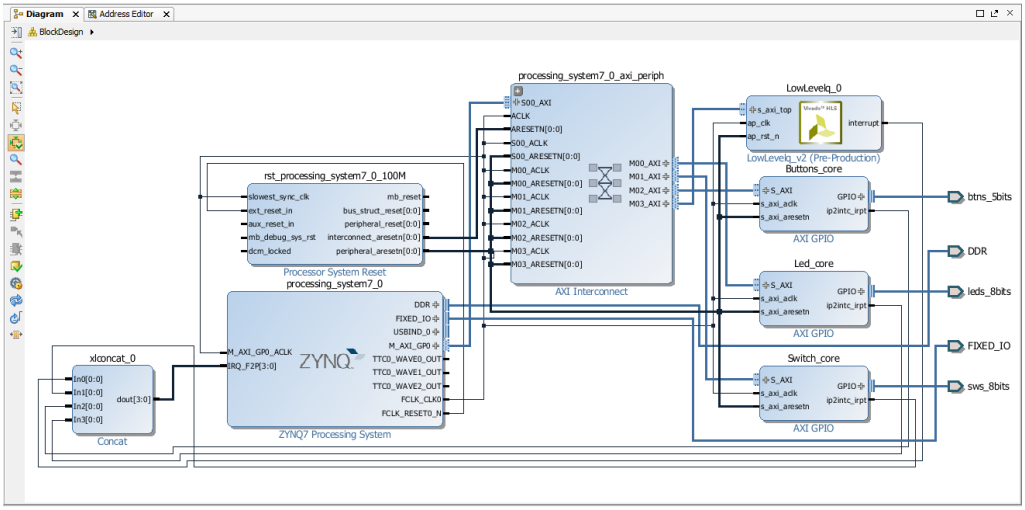

Figura 3.5: Contenido del block design generado.

Como se puede observar la parte central de este *block design* es el módulo correspondiente a la **PS** llamado *processing_system7_0*, explicado en el apartado anterior, en esta imagen aparecen sus interconexiones a traveses del *bus AXI* de propósito general que interconectado con un *AXI Interconnect* para que éste pueda realizar las funciones de switch entre el *master* y los distintos *slaves* que aparecen en el diseño. Es importe observar el módulo generador de *reset* asíncrono, que se trata del *reset* general del sistema, este afecta a cada uno de los módulos mostrados en la imagen reestableciendo su estado iniciándolo si hay algún problema.

Los cuatro *slaves* que aparecen en la imagen son el módulo de procesado de datos, generado a partir de fuentes escritas en *C/C++* y generado con la herramienta *HLS* importándolo como *IPcore*, el módulo de control de los *botones* de la placa, el módulo de interconexión con los *switches* externos y el módulo de control de los *leds* del sistema que permite generar señales perceptibles a la persona que desarrolla el proyecto, sirviendo de una gran ayuda a la hora de la depuración. Estos tres últimos sistemas están interconectados con el sistema hardware de la placa de evaluación *ZedBoard* donde se encuentra la *Zynq 7 000* integrada como un elemento más.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
Led_core	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF
Switch_core	S_AXI	Reg	0x4121_0000	64K	0x4121_FFFF
Buttons_core	S_AXI	Reg	0x4122_0000	64K	0x4122_FFFF
LowLevelq_0	s_axi_top	Reg	0x43C0_0000	64K	0x43C0_FFFF

Figura 3.6: Mapa de periféricos del block design.

La pestaña de **Address Editor** muestra las direcciones de memoria en las que han sido mapeados los distintos dispositivos, se trata de las *direcciones físicas* que se tendrán que introducir desde el sistema

operativo, atravesando la capa de *virtualización*, para acceder a cada módulo. Es por lo tanto, un apartado muy importe del diseño ya que serán estas las direcciones de base que tendrán los distintos *registros* de los módulos para almacenar sus registros de control, actuación o almacenamiento de datos. Estos registros dependen de cada uno de los *IPcores* utilizados, disponibles en los *datasheet* de los mismos.

Address Space Offset[3]	Register Name	Access Type	Default Value	Description
0x0000	GPIO_DATA	R/W	0x0	Channel 1 AXI GPIO Data Register.
0x0004	GPIO_TRI	R/W	0x0	Channel 1 AXI GPIO 3-state Control Register.
0x0008	GPIO2_DATA	R/W	0x0	Channel 2 AXI GPIO Data Register.
0x000C	GPIO2_TRI	R/W	0x0	Channel 2 AXI GPIO 3-state Control.
0x011C	GIER[1]	R/W	0x0	Global Interrupt Enable Register.
0x0128	IP IER[1]	R/W	0x0	IP Interrupt Enable Register (IP IER).
0x0120	IP ISR[1]	R/TOW[2]	0x0	IP Interrupt Status Register.

Figura 3.7: Mapa de registros de periférico.

Por último, se puede observar que se pretende trabajar con las interrupciones de los distintos dispositivos, estas son generadas en los distintos *IPcore*, las cuales se tendrán que configurar para que se activen al cambiar los datos o a la funcionalidad deseada, para poder realizar la interconexión de las mismas es necesario un módulo de concatenación, llamado *xlconcat_0* que permite conectarlas al controlador de interrupciones interno que incluye la *PS* para que el sistema embebido posteriormente controle las interrupciones desde sistema operativo o aplicación de usuario.

Para la configuración del módulo de gestión de la *interrupción* interno a la *PS* es necesario acceder a la configuración del módulo procesador, en el apartado de interrupciones, en él se deben activar las *Fast Interrupt* para que el sistema active las interrupciones, según la descripción indica que desde sistema operativo se observarían entre los números *91:84* y *68:61* pero realmente aparecen en números más altos, la detección de este *error* se explica en el tema de device drivers, cómo detectarlo y cómo solucionarlo.

Es importante tener en cuenta que hay sistemas de la *PS* que están activos por defecto, estos, son los que aparece un *tic* en la zona verde de cada uno de los dispositivos que aparecen en la siguiente imagen. En esta imagen se puede observar que las comunicaciones serie que soporta el *USB1* están activas, posteriormente utilizadas para la comunicación con el sistema operativo en un primer momento y luego con el dispositivo *hardware* de muestreo y almacenamiento de los datos de ultrasonidos recibidos desde las balizas de ultrasonido. La comunicación habitual entre el *PC* y el sistema será a través de *SSH* por *ETH0*, el cual se carga en la *Zynq* desde la comunicación habilitada con la *SD*.

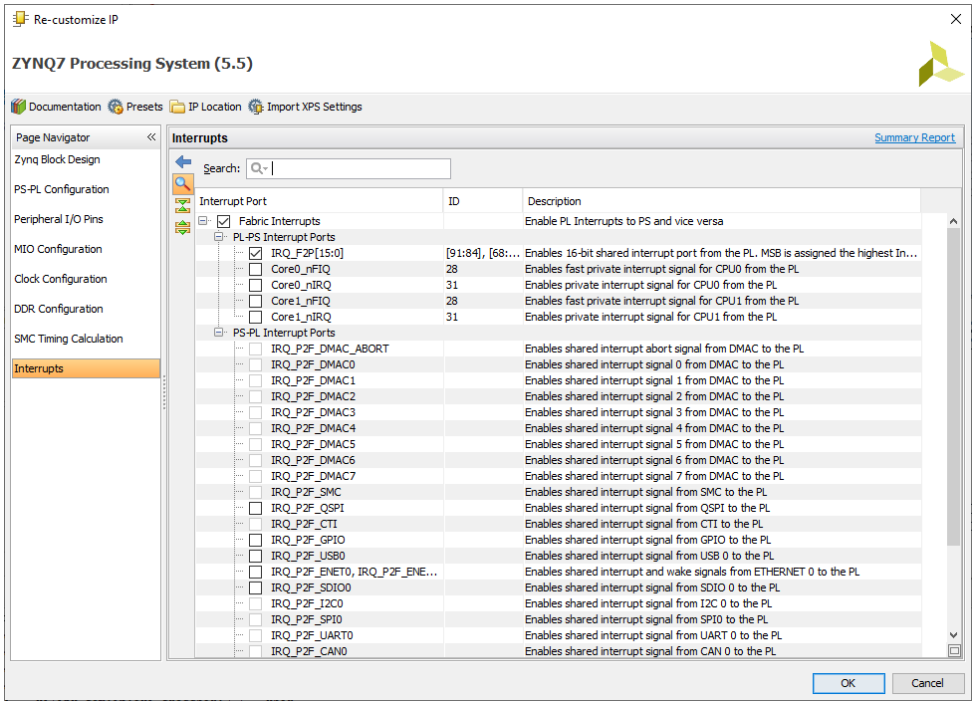

Figura 3.8: Configuración de las interrupciones en block design.

Capítulo 4

Sistema Operativo

Éste capítulo trata temas como la importancia de un entorno de trabajo conocido y automatizado en el que trabajar. Éste ha de soporte las herramientas necesarias y no genere más problemas de los que ya puede llegar a generar el proyecto que se pretende realizar, también se introduce la herramienta *Petalinux* como herramienta de generación del sistema operativo a utilizar sobre la *FPGA*.

En el momento en el que se conoce la finalidad de la herramienta se explica como se ha utilizado para cada uno de los distintos puntos del proyecto, desde el sistema operativo base hasta las aplicaciones de inicialización de los device drivers, incluyendo la generación y compilación de los device drivers.

Por último se explica como llevar a cabo una inicialización y testeo del sistema generado indicando el procedimiento con especial hincapié en los puntos mas importantes para una correcta generación del sistema.

4.1 Automatización del entorno de trabajo

Al tratarse de un proyecto iterativo, en el que se avanzará poco a poco es importante que el entorno de trabajo tenga ciertas características, en este caso se ha escogido el *Sistema Operativo CentOS 6.10*, ya que se trata de un sistema muy estable basado en *Red Hat*, esta característica aparece como recomendación del uso de las herramientas de *Xilinx*. Dentro de las posibles elecciones que tomar como sistema operativo de trabajo, se ha escogido esta por herencia de desarrollo de los profesores que han implementado y depurado este tipo de sistemas previamente, pudiendo haber elegido otros como son *Fedora*, *Centos 6.10* y *Centos 7* los cuales entran dentro de las tres mejores distribuciones para este tipo de trabajo.

CentOS (*Community ENTerprise Operating System*) se trata de un *branch*, o bifurcación a nivel binario de *Linux Red Hat* (*RHEL*), sostenido de forma libre y voluntaria a partir del código liberado por *Red Hat Enterprise*. Está compuesto por software libre de código abierto, compartido con personas afiliadas a la causa a partir de cuotas monetarias, a partir de ese código se crea *CentOS* pero sin ser mantenido por *Red Hat*. Se hace uso del gestor de paquetes *yum* para realizar descargas de software o actualizaciones, herramienta también utilizada por *Fedora*, motivo por el cual entra dentro de las principales opciones estudiadas. Se trata de una versión antigua ya que su última versión es de 2013, aportando fiabilidad y seguridad pero con el inconveniente de la instalación de actualizaciones.

Se instaló por su estabilidad y por ser la opción de trabajo de mi tutor, que conocía el sistema y me podría aconsejar a la hora de solucionar algunos de los problemas que han aparecido con las herramientas. Se ha probado también *CentOS 7* y resultó mucho más cómodo, sobre todo a la hora de las aplicaciones a utilizar y dependencias necesarias en la instalación de las herramientas utilizadas.

Fedora es una distribución Linux basada en *RPM* y de propósito general, se caracteriza por su estabilidad mantenida por una comunidad internacional de ingenieros, diseñadores gráficos y usuarios que detectan e informan de los errores y actualizan el sistema preparándolo para las nuevas tecnologías, una de las grandes ventajas que tiene el sistema para el propósito del proyecto, contando con ayuda de *Red Hat*. Trata de liderar el ámbito tecnológico de software libre y código abierto, siendo en multitud de ocasiones el entorno de pruebas y estabilización de aplicaciones ideadas para *Red Hat*, con una alta densidad de cambios seguro sobre las fuentes originales en vez del uso de parches específicos de la distribución, asegurando actualizaciones a todas las distribuciones y variantes de Linux. Por este motivo se identifica como una distribución a tener en cuenta.

Por motivos de compatibilidad esta distribución se instala en una máquina virtual gestionada por *VirtualBox*, sobre la que se automatiza su generación, instalación de dependencias de desarrollador e instalación de aplicaciones y herramientas necesarias. Es importante tener en cuenta que el lenguaje de instalación tiene que ser inglés para evitar algunos problemas con las herramientas de *Xilinx*, una vez se han resuelto las dependencias de las herramientas este *script* instala las herramientas de *Xilinx* previamente descargadas, en un inicio únicamente la herramienta de *Petalinux*, pero se añaden *Vivado*, *SDK* y *HLS* más adelante para utilizar únicamente este SO, reduciendo el presupuesto del proyecto eliminando las licencias de *Windows*.

4.2 Uso de la herramienta Petalinux

En este proyecto, se utilizan las herramientas de *Vivado 2015.4*, por comodidad y conocimiento de las herramientas, también hay que mencionar que ya que las herramientas necesitan licencias hay que seleccionar las disponibles. La herramienta de *Petalinux* no requiere de ninguna licencia, pero se utiliza la del resto de herramientas, al trabajar sobre una distribución Linux se decide trabajar preferiblemente con la terminal de comandos, evitando en medida de lo posible entornos gráficos.

Petalinux es la herramienta que permite, crear, compilar e implementar cualquier tipo de soluciones basadas en Linux embebido para las piezas de *Xilinx*, diseñada para la optimizar este tipo de sistemas, pudiendo, una vez se genera el *kernel*, generar distintas distribuciones en función de las herramientas y dependencias que se requieran en la aplicación final.

La herramienta *Petalinux* ofrece todo lo necesario para poder diseñar, generar e implementar soluciones basadas en *Linux Embebido* en sistemas de procesamiento de Xilinx, en este caso la base hardware para la que se ha generado ha sido *Zynq 7000*. Esta herramienta ha sido diseñada para la aceleración de la productividad de diseño, ya que es habitualmente utilizada en diseño hardware como ayuda en la simplificación del diseño.

Esta herramienta facilita el desarrollo de soluciones basadas en *Linux*, desde el arranque hasta la ejecución de aplicaciones con las distintas herramientas internas disponibles, como puedan ser la línea de comandos, desde la cual se permite la gestión de todos los recursos disponibles por el sistema. Generadores de aplicaciones, controladores de dispositivos de desarrollo. Generación de imágenes del sistema de arranque para poder replicar estos sistemas, agentes de depuración y automatización de herramientas. Para esto el sistema de depuración aconsejado es el depurador de *Xilinx* a partir del *SDK*.

Esta herramienta permite la generación de *BSP's* personalizadas sincronizando la plataforma software con el diseño del hardware a medida del aumento de capacidades adquiridas por la plataforma hardware. Estas herramientas permiten la generación automática de paquetes de controladores específicos para núcleos *IP* de procesado embebido, configuraciones de kernel y gestión de arranque. Esta capacidad

permite separar el desarrollo software del desarrollo hardware permitiendo un solapamiento en desarrollos agilizando el diseño y desarrollo del producto final.

Las herramientas de configuración Linux como son el gestor de arranque, el kernel de *Linux* y el sistema de archivos son configurables desde la herramienta a partir de los paquetes, bibliotecas y parámetros configurables del sistema. Siendo conscientes en todo momento de hardware personalizado sobre el que se soporta, pudiendo incluir *IPcores* específicos generados por *Xilinx* para la implementación automática de procesos hardware de ayuda al sistema.

Esta herramienta incluye también plantillas de desarrollo que permiten la creación de controladores o device drivers como el generado en este proyecto para controlar de forma personalizada. Una vez se ha creado el sistema operativo estas herramientas permiten el empaquetar preparando para distribución los componentes software para su fácil instalación en un entorno *Petalinux* externo.

Los puntos más importantes que hacen importante esta distribución son el gestor de arranque, la posibilidad de generación de un Kernel optimizado para la *CPU* y hardware que lo soporta, las aplicaciones y bibliotecas estándar de *Linux*. La posibilidad de desarrollar y depurar aplicaciones generadas en *C/C++* e hilo de soporte *FPU* para operaciones en coma flotante a nivel software y un servidor web integrado que permite una gestión más sencilla de las configuraciones de red y el firmware del sistema.

4.3 Generación del Linux básico

Se parte del fichero de descripción hardware, **Top.hdf**, se trata de la cantidad mínima de información que necesita *Petalinux* para la generación del Linux, este fichero se consigue una vez se ha implementado el hardware que se necesita, explicado en el apartado anterior. En este punto lo primero que se hace es seguir el flujo de trabajo en función de las herramientas necesarias y la arquitectura de información del proyecto.

El flujo de trabajo ideado se inicia con la creación del sistema operativo de base, el cual no requiere ningún tipo de configuración, permite la generación de las fuentes que posteriormente habrá que configurar y compilar para adecuarlo a la solución requerida.

```
petalinux-create --type project --template zynq --name linux
```

Como se puede observar en el comando seleccionado previamente, se crea un nuevo sistema, bajo el nombre de Linux, creando la herramienta una carpeta con este nombre en la que almacenará la distribución, utilizando una plantilla específica para *Zynq* al utilizar la *ZedBoard* en el proyecto de prueba y la misma pieza en el proyecto final. Para poder personalizar el diseño es necesario incluir el archivo **Top.hdf** conseguido al exportar el hardware una vez está implementado el sistema hardware, por ello será necesario incluir una carpeta, que en este caso se ha nombrado como *hwdef* dentro de la carpeta Linux, una vez se ha incluido el hardware para el que se va a configurar el SO será el momento para pasar a personalizar y configurar el Linux embebido. Hay distintas formas de configurar el SO, en este caso serán necesarios unos parámetros específicos para poder otorgar al sistema de las capacidades mínimas de comunicación para su futuro control y cómoda depuración. Se utiliza el comando mostrado a continuación.

```
petalinux-config --get-hw-description=./hwdef
```

Este comando proporciona una rudimentaria GUI sobre terminal que permite iniciar las modificaciones del sistema, para empezar es necesario poder comunicarse vía serie con la FPGA, para ello se accede *Kernel*

Bootergs que es el lugar en el que se indican las opciones de configuración más básica del sistema, en este caso se ha utilizado uno de los dos núcleos que tiene disponibles esta pieza, con una memoria utilizada de *768MB* y unas opciones de configuración de *UART* de uso del periférico *ttyPS0* a una velocidad de transmisión de datos de *115200 Baudios*, se le añade la opción de *earlyprintk* para poder depurar en un futuro los periféricos y device drivers del sistema permitiendo imprimir por puerto serie comandos en el arranque del S.O.

```
petalinux-config --get-hw-description=./hwdef
```

Configuración a incluir:

```
console=ttyPS0,115200 earlyprintk maxcpus=1 mem=768M
```

Actualmente la comunicación serie es un método muy utilizado para la depuración de firmware y software en dispositivos embebidos, pero para una correcta comunicación y transferencia de datos es importante disponer de comunicación *IP*. Por defecto, el sistema viene preparado para una configuración de la dirección *IP* dinámica, para al utilizar protocolos *DHCP* conectándose de manera sencilla y cómoda.

Para un diseño en una red lo más segura posible y ambientes controlados y es aconsejable utilizar *IP* estática, en este caso se añade el inconveniente de necesitar un *router* para poder conectarse desde *Ethernet* a la *Zynq*. En este proyecto, será la forma preferente de control del sistema operativo, la ejecución y depuración de funciones y transferencia de datos, por ello es necesaria su configuración.

En el mismo punto de configuración al que se ha accedido anteriormente, en el directorio *Subsystem AUTO Hardware Ethernet settings* y se des-selecciona la opción de *Obtain IP automatically* para fijar una dirección *IP* estática que permita conectarse directamente desde el ordenador por *Ethernet*.

Al salir de la herramienta se aplican las modificaciones realizadas, en este punto será necesario configurar el *rootfs (root file system)*, para poder acceder a los periféricos configurados anteriormente permitiendo la comunicación desde el usuario root, el comando indicado a continuación permite acceder a la pantalla de la *GUI* de configuración.

```
petalinux-config -c rootfs
```

En este apartado se debe acceder al directorio *File systems/Console Network* y se debe seleccionar la opción de *enable dropbear* pulsando la tecla *y* confirmando los cambios al salir.

Un aspecto importante a la hora de la generación del sistema para este sistema en especial, es la capacitación del USB como interfaz de conexión que permita sistemas *plug and play*, esto será necesario si se quiere conectar dispositivos sencillos con estas capacidades como pueda ser un teclado o un ratón, en este caso, se utiliza para la comunicación directa con el periférico hardware de adquisición de señal de ultrasonidos.

Para ello se accederá a la configuración del kernel con el siguiente comando.

```
petalinux-config -c kernel
```

Se trata de uno de los menús más complejos que tiene la herramienta, permite el acceso a una gran cantidad de dispositivos y configuraciones posibles, también permite la configuración del sistema adaptándolo a dispositivos muy específicos de distintos fabricantes.

Este sería el momento en el que se puede empezar a generar un sistema básico Linux, este sistema tal y como se ha configurado, es sencillo y ligero, siendo válido unicamente para ejecutar aplicaciones

precompiladas, explicadas en los siguientes apartados, el comando de generación final del *Petalinux* es el que se muestra a continuación.

```
\$ petalinux-build
```

4.4 Integración de device drivers en la generación del sistema

Si, como en este caso, se pretende realizar un control sobre algún periférico de forma personalizada, ya sea comercial o *custom*, será necesario crear un *device driver*. Un device driver se trata de una función en el espacio de kernel que controla un periférico hardware previamente creado o ejecuta una funcionalidad a nivel kernel.

Partiendo del sistema compilado, tal y como se queda en el apartado anterior será necesario realzar una limpieza del *workspace* de la herramienta *Petalinux* para poder continuar, se realza utilizando el siguiente comando.

```
petalinux-build -x mrproper
```

Al haber limpiado el *workspace* se borran los archivos generados de kernel que son necesarios para poder conocer las características del sistema, para ello se re-compila el *kernel* al que se le quiere añadir los device drivers, pudiendo compilarlos sin problemas.

```
petalinux-build -c kernel
```

En este caso se han añadido al *block design* distintos periféricos para poder estudiar el proceso, un periférico para el control de los switches, un periférico para el control de los botones, uno para el control de los *leds* y por último uno para procesado de señal de un periférico hardware que se utilizará posteriormente para la localización.

Se muestra, a continuación, el comando que se va a ejecutar para indicar al kernel que se va a añadir un device driver genérico, habrá que modificarlo en función del periférico a crear.

```
# Comando genérico
petalinux-create -t modules --name mymodule --enable

# Comandos usados
petalinux-create -t modules --name LedDriver --enable
petalinux-create -t modules --name SwitchDriver --enable
petalinux-create -t modules --name ButtonsDriver --enable
petalinux-create -t modules --name LowLevelq_drv --enable
```

En este punto se debe copiar los *device drivers* que se han preparado, correctamente explicados en el siguiente capítulo, para poder continuar con la generación del sistema.

Para poder estar seguros de no tener ningún fallo en los *scripts* y depurar de una forma cómoda y correcta se copian los device drivers uno a uno, para ello se utiliza el comando mostrado posteriormente, si existiesen errores aparecen reflejados en el *log* del *build* **build/build.log**.

```
# Comando genérico
petalinux-build -c rootfs/mymodule

# Comandos usados
petalinux-build -c rootfs/SwitchDriver
petalinux-build -c rootfs/LedDriver
petalinux-build -c rootfs/ButtonsDriver
petalinux-build -c rootfs/LowLevelq_drv
```

Una vez se tienen los distintos módulos compilados se puede recompilar el sistema, se accede para utilizar los device drivers y empieza a hacer las aplicaciones de usuario para acceder a los periféricos hardware.

4.5 Generación de scripts o aplicaciones de inicialización

Es aconsejable que los device drivers que se han creado se inicien en el arranque, esto permite observar los mensajes de depuración introducidos en el cada uno para ver que correctamente se inicia el sistema. Estos mensajes salen por la *UART* que se configura en el apartado de creación del Linux básico, que será donde están los parámetros de configuración.

Para poder crear aplicaciones de ejecución automática se tiene que aplicar el siguiente comando, este creará una aplicación **.c* con un *Makefile* asociado, en nuestro caso se va a incluir un fichero **.sh* con comandos de terminal que permitirán inicializar los device drivers.

```
# Comando genérico
petalinux-create -t apps --name myscript --enable

# Comando Usado
petalinux-create -t apps --name initDevDri --enable
```

Se procederá a borrar los ficheros *Makefile* y **.c* que se generan dentro de la carpeta de esta aplicación y añadir los que se quieren ejecutar para el descubrimiento de los device drivers, el contenido del *script* de terminal que se ha creado es el incluido a continuación.

```
#
# Inicializo los periféricos.
#                               Led Driver
#                               Switch Driver
#                               Buttons Driver
#                               Software Driver
#                               LowLevelq Driver
modprobe LedDriver
mknod -m666 /dev/LedDriver@41200000 c 240 0
modprobe SwitchDriver
mknod -m666 /dev/SwitchDriver@41210000 c 241 0
modprobe ButtonsDriver
mknod -m666 /dev/ButtonsDriver@41220000 c 242 0
#modprobe SoftwareDriver
#mknod -m666 /dev/SoftwareDriver@deadbeef c 243 0
modprobe LowLevelq_drv
mknod -m666 /dev/LowLevelq@43C00000 c 244 0
```

Se pueden observar dos tipos de comandos, *modprobe* inicializa el módulo, realizando la detección, el comando *mknod* permite dar privilegios al device driver para poder ejecutarse y realizar acciones.

Para obtener las fuentes a partir de las cuales se generarán los archivos que se cargarán en la Tarjeta *SD* que se introducirá en la *SD* de la *FPGA* se recompilará todo el sistema, como se ha hecho anteriormente con el comando *build*.

```
petalinux-build
```

4.6 Generación de archivos de carga

La herramienta *SDK* de *Xilinx* que se instala en la instalación de la herramienta **Vivado Design Suite - HLx Editions 2015.4**, será la que permita pasar de tener las fuentes generadas en el directorio *images/linux* a imágenes de carga sobre la *FPGA*.

Al abrir esta herramienta, se debe acceder a la pestaña *Xilinx Tools* seleccionando *Create Boot Image* permitiendo seleccionar en un primer momento un directorio en el que se almacenarán los archivos finales y un archivo de generación automática que automatiza este proceso. El archivo generado será **output.bif** en el se indica cual es el archivo que se tomará como *bootloader*, indicándole la configuración de kernel que cargará la *Zynq*, también se le deben añadir el que indica las propiedades hardware **Top.bit** y el que que se ejecuta por el kernel como sistema de usuario.

```
the_ROM_image:
{
    [bootloader]A:\TFM\ProyectoFinal\Petalinux\linux\images\linux\zynq_fsbl.elf
    A:\TFM\ProyectoFinal\Petalinux\linux\images\linux\Top.bit
    A:\TFM\ProyectoFinal\Petalinux\linux\images\linux\u-boot.elf
}
```

Este proceso genera un archivo llamado **BOOT.bin** que junto al archivo **image.ub** generado anteriormente por la herramienta *Petalinux* se han de copiar en la tarjeta *SD* para el arranque del sistema operativo.

4.7 Primer Arranque y depuración

En un primer arranque lo primero que se tiene que comprobar en la Tarjeta de evaluación es la configuración de arranque de la *FPGA*, es decir, la localización de la carga, en este caso al cargar la *FPGA* desde la tarjeta flash se tienen que configurar los *Jumpers* de carga tal y como se muestra la documentación de la tarjeta.

A continuación será necesario configurar el *USB* de comunicación para la depuración entre el ordenador y la *FPGA*, para ello se conectará un cable *USB-uUSB* entre el ordenador y la *UART* de depuración que se encuentra al lado del conector de *JTAG* cerca del switch de alimentación del sistema.

Una vez se conecta se tiene que abrir el software de conexión por puerto serie, en este caso se usará *MobaXTerm* ya que se trata de un software muy completo que permite muchos tipos de comunicaciones soportando protocolos desde serie a transferencia de datos en red.

Se indicará que se trata de una comunicación serie a la velocidad configurada en la generación del kernel, *115200 baudios*, y se encenderá la FPGA observando el *log* del sistema al arrancar gracias a la opción *earlyprintk*.

En este caso la parte más importante llega prácticamente al final, en el punto en el que se ejecuta la función de inicialización de los drivers y se pueden comprobar los mensajes devueltos como sistema de depuración del arranque de los módulos, en ellos mostrará la información que se explica en el capítulo en el que se crea y explica el device driver.

Para acceder al sistema, el usuario por defecto y si no se personaliza en la configuración será **root** y la contraseña por defecto será **root**. En este momento ya se ha accedido al sistema, lo primero que hay que hacer es probar que están los device drivers creados en la carpeta de los periféricos, con el comando mostrado a continuación se muestran todos los periféricos y se identifican los utilizados.

```
\$ ls /dev
```

En el caso del proyecto que se ha realizado se muestran los siguientes periféricos, como se puede observar los primeros son los generados en esta práctica.

Para poder ejecutar una aplicación de forma correcta será necesario que se conecte el cable de *Ethernet* desde el puerto del ordenador hasta la *ZedBoard*, este cable debe ser un cable cruzado, ya que no se ha activado el protocolo *Full-Duplex*, por este motivo no se debe usar un cable normal, no llegaría a haber una correcta comunicación.

Se debe configurar el adaptador de *Ethernet* en el ordenador en la misma red con una *IP* válida que haga que se encuentre en la misma red para poder comunicarse con la *FPGA*.

Para comunicarnos a través de *Ethernet* se realizará con el protocolo *SSH* mientras que la transmisión de información como la transferencia de la aplicación a ejecutar se llevará a cabo con *SFTP*, estos protocolos también son soportados por *MobaXTerm* si se trabaja desde Windows y desde la terminal si se trabaja desde **Linux**.

Al conocer la *IP* que tiene la *FPGA* por haber sido fijada en la configuración se puede acceder a ella con el usuario **root**, la transferencia de la aplicación ya es directa y el control de la *FPGA* por lo que no será necesario tener que acceder con el cable *USB*.

Si no se ha configurado la *IP* estática en la configuración inicial, se necesita un *router* que permita la ejecución del protocolo *DHCP* para el descubrimiento y asignación de una *IP* perteneciente a la red para la *FPGA*, en este caso lo conveniente es arrancarla con el cable *USB*, detectando la *IP* con el comando *ifconfig* una vez se ha accedido al sistema.

Capítulo 5

Device Driver

Este apartado trata de explicar el objetivo del device driver como elemento software y su importancia dentro de la arquitectura del sistema, sirviendo, principalmente como comunicación entre el nivel *hardware* y el espacio de aplicación de usuario, actuando en el espacio *kernel* a nivel *software*.

También se explican las distintas partes de las que está compuesto y funcionalidades a partir de uno de los device drivers que se han generado para el proyecto, en este caso se utilizará como ejemplo el de los switches ya que ha sido el más complejo llegando a utilizar las interrupciones.

Existe una gran cantidad de modelos de device driver, con diferencias entre ellos en función del periférico hardware con el que se pretende establecer una comunicación o un control. En este proyecto se han iniciado el estudio de los *char devices*, se trata de device driver de comunicación sencilla con periféricos incluyendo una pequeña gestión de memoria.

La idea de device driver que se pretende generar en este capítulo es la de un elemento software orientado al control de un periférico hardware por lo que será necesario conocer muy bien el dispositivo al que se accede y su interfaz para proporcionar los métodos necesarios que se pueden utilizar desde un futuro software.

5.1 Fundamentos teóricos de los device drivers

Para una correcta comprensión del device driver como estructura de datos y funcionalidades estructurales dentro del sistema final, se ha recurrido al libro **Rubini**, principalmente al capítulo de char devices en el que se explican algunos de los conceptos necesarios para la generación de un device driver de forma correcta.

A continuación, se explican algunos de los conceptos necesarios desde las distintas perspectivas que han sido utilizados en el diseño, tratándose desde una visión conceptual más que desde una perspectiva de funcionalidad de código, explicada en los siguientes apartados.

5.1.1 Diseño del device driver

En el diseño del driver el primer paso es la definición del alcance u objetivos del mismo, sobre todo es importante definir correctamente que capacidades debe llegar a aportar a una futura aplicación *software*. Al tratarse de una parcela de memoria el diseñador puede ejecutarlo o tratarlo de la forma más cómoda para un futuro uso del mismo.

Para poder aumentar la capacidad de uso del *driver* desde una aplicación externa es importante la definición clara de las interfaces o capas de abstracción generadas por el mismo entre el *hardware* y el *device driver* y entre el device driver y el espacio de usuario o capa de aplicación. Esto se implementa como distintos tipos de driver a implementar, requiriendo distintos tipos de dificultades de uso y complejidad.

El tipo de acceso que se realizará sobre el dispositivo hardware definirá el tipo de persistencia con la que se deberá tratara este tipo de memoria, si se trata de un archivo que se abrirá en múltiples ocasiones y se puede abrir por distintas aplicaciones es aconsejable que se trate como memoria global o estática para evitar perdidas de tiempos de apertura y cierre, evitando la perdida de los datos que puede llegar a contener.

Si se trata, por el contrario, de un fichero que se utilizará ocasionalmente se puede tratar como un acceso más directo a la memoria del dispositivo como si se tratase de tuberías hacia la memoria *hardware*, siendo necesario el uso de semáforos para el bloqueo del sistema impidiendo la corrupción de la información obtenida en un acceso, ya sea de lectura o escritura.

Es también en función del número de usuarios permitidos y el tipo de accesos que se realicen si se definirá como un tipo de device driver u otro.

5.1.2 Identificación de los device drivers en el sistema operativo

Los device drivers serán accedidos a partir del sistema de ficheros soportado por el sistema operativo utilizado, es para poder realizar una correcta identificación para lo que se necesitan los **Major y minor numbers**, se trata de los números que nos ayudarán a identificar cada uno de los device drivers operativos en el sistema operativo y el tipo de device driver, el tipo también lo indicará una letra que acompaña, pudiendo ser una *c* si se trata de un *char device* o una *b* si se trata de un *block device*.

Para identificar las características del device driver se accederá a la carpeta del sistema */dev* en la que ejecutando el comando *ls -la* mostrará las características y tipos de cada uno, se puede observar que un mismo driver puede controlar diversos periféricos, teniendo distintos device drivers un mismo *Major* pero la mayor parte de los dispositivos suelen controlarse uno a uno.

El *Minor Number* es un puntero desde el kernel hacia el dispositivos utilizar dentro del driver creado, pudiendo utilizarse como acceso directo desde kernel. Es por ello por lo que no tiene importancia la relación entre *Minor numbers* de distintos dispositivos. La identificación de los *Major y minor numbers* de un dispositivo se pueden obtener a partir de las funciones mostradas a continuación, definidas en la librería *<Linux/types.h>*, siendo posible también la identificación del device driver a partir de sus *major y minor numbers*.

```
MAJOR (st\_dev dev);
MINOR (st\_dev dev);
MKDEV (int major, int minor);
```

Para la **asignación y liberación** de recursos al periférico, el driver tiene que obtener los *major y minor numbers* del dispositivo. Este proceso se realiza en el descubrimiento del periférico hardware por parte del device driver, para incluirlo en el espacio de device drivers se utiliza la siguiente función.

```
int register\_chrdev\_region(dev\_t first, unsigned int count, char *name);
```

El primer parámetro será el inicio de rango que se quiere asignar, el siguiente el número de dispositivos que se quieren registrar y el tercero es un puntero a un array con los nombres de los dispositivos

como cadenas de carácteres, el uso práctico de esta función se demuestra en los siguientes apartados de explicación del código.

Si no existe ningún problema como pudiese ser un solapamiento de device drivers la función devuelve un 0 para indicar que el proceso ha sido correcto, devolviendo un número negativo en caso de haber cometido un error en el proceso de registro.

Para la liberación del dispositivo del espacio de device drivers del kernel, se utilizará la siguiente función, utilizada en el cierre del device driver o apagado del sistema.

```
int alloc\_chrdev\_region(dev\_t *dev, unsigned int firstminor, unsigned int count, char *
    name)
```

El primer parámetro identifica el device driver sobre el que actuar, incluyendo el *minor number* que identifica el periférico hardware controlado por el driver, con el número de periféricos y nombres a liberar.

Un incorrecto proceso de descubrimiento y asignación de *major y minor numbers* puede hacer que un periférico quede inactivo, en el diseño del device driver se puede incluir una funcionalidad avanzada como la **asignación dinámica de major number**.

Este proceso no se ha implementado en este proyecto ya que el número de periféricos a utilizar es reducido y el proceso es costoso en tiempo de desarrollo, siendo necesaria una función que detecte todos los device drivers activos y generando una lista de dispositivos utilizados, asigne números libres de uso.

5.1.3 Estruturas de información

La **estructura fops (file operations)** es la estructura que define la capa de abstracción o interfaz que puede tener una aplicación de usuario respecto al device driver. En ella se definen los métodos disponibles para tratar al device driver como un objeto del sistema.

```
struct file\_operations scull\_fops = {
  .owner = this\_module,
  .llseek = scull\_llseek,
  .read = scull\_read,
  .write = scull\_write,
  .ioctl = scull\_ioctl,
  .open = scull\_open,
  .release = scull\_release,
};
```

El ejemplo que se ha mostrado es el de un posibledevice driver, ya que en función del tipo de device driver se incluirán unos métodos u otros, siendo algunos constantes o de funcionalidad extendida, a continuación se ha muestra una lista de los posibles métodos a incluir para el control del device driver.

- *struct module *owner.*

  el primero de los campos del device driver es el puntero **owner*, no se trata de una operación. Se trata de un campo que evita que sea eliminado mientras ejecuta cualquier operación. Se puede entender como un método de seguridad.

- *loff_t (*llseek) (struct file *, loff_t, int);.*

  Al tratar estos device drivers como ficheros, es necesario el método o acción que permita desplazar el puntero de lectura o escritura sobre el mismo. Este método tiene este objetivo, mover el puntero

sobre el espacio de memoria del dispositivo, en este caso al tratarse de un device driver se utiliza para apuntar a un registro especifico y poder leer un bloque de registros.

- *ssize_t (*read) (struct file *, char ___user *, size_t, loff_t *);*

  Se trata de unos de los métodos más importantes, el método de lectura, utilizado para extraer información desde el espacio de memoria física del dispositivo hacia el espacio de usuario donde se trata habitualmente en una aplicación.

- *ssize_t (*aio_read)(struct kiocb *, char ___user *, size_t, loff_t);*

  Esta función ejecuta una lectura asíncrona, lanzando la acción de escritura y continuado con la ejecución de la siguiente operación antes de la transmisión completa de información, puede ocasionar problemas si no se ejecuta correctamente. Este tipo de lecturas se puede utilizar si se requiere realizar una lectura mientras se ejecuta la inicialización del siguiente dispositivo a utilizar.

- *ssize_t (*write) (struct file *, const char ___user *, size_t, loff_t *);*

  El método de escritura entra dentro de los métodos más importantes y necesarios que ha de tener un dispositivo. Este método permite realizar escrituras sobre el espacio de memoria del dispositivo, permitiendo controlar actuadores o el control del propio módulo.

- *ssize_t (*aio_write)(struct kiocb *, const char ___user *, size_t, loff_t *);*

  Al igual que se puede realizar una lectura asíncrona se debe poder realizar escrituras asíncronas en un dispositivo, pudiendo ser utilizadas en la inicialización de un dispositivo en el transcurso de uso de otro o ejecución de operaciones previas a su uso.

- *int (*readdir) (struct file *, void *, filldir_t);*

  Este método permite la lectura de un sistema de archivos, en este caso no es útil para este proyecto ya que no se realiza ningún uso del sistema de archivos.

- *unsigned int (*poll) (struct file *, struct poll_table_struct *);*

  Esta función es capaz de detectar si existe algún tipo de bloqueo de memoria, como puedan producir las funciones de lectura y escritura, síncronas o asíncronas, en función de los permisos de ejecución sobre el archivo que se quiere utilizar.

- *int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);*

  El método *ioctl* aporta la capacidad de customización de la comunicación con un módulo de memoria o archivo, a partir de un protocolo específico, puede servir también para realizar escrituras en localizaciones específicas de cantidades de tamaño variable. En el módulo de procesamiento de datos integrado en el sistema se utiliza para la configuración de ciertos parámetros.

- *int (*mmap) (struct file *, struct vm_area_struct *);*

  Método que implementa una solicitud de asignación de memoria para un proceso específico. Habitualmente se utiliza en transferencia de grandes cantidades de memoria o cuando es necesaria una compleja operación para el mantenimiento de mayores cantidades de datos.

- *int (*open) (struct inode *, struct file *);*

  El método open entra dentro de los métodos esenciales, realiza la apertura de un dispositivo para su utilización y tratamiento. Sin este método no se podrían utilizar ninguno de los otros módulos. Se entiende como desbloqueo del módulo, archivo o bloque de memoria. Si devuelve un valor *NULL* es que ha habido algún error en la apertura teniendo que reabrirlo o solventar el problema.

- *int (*flush) (struct file *);*

  Flush permite realizar una limpieza sobre el módulo. Esta limpieza se refiere a la finalización o cierre de funciones u operaciones en cola para ejecutar sobre este módulo o la limpieza de sus *buffers* y bloques de memoria. Ejecuta tareas de limpieza y sincronización.

- *int (*release) (struct inode *, struct file *);*

  Todo espacio inicializado en memoria ha de ser liberado antes del cierre del sistema como buena práctica y correcto uso del sistema. Es por este motivo por el cual existe este método. *Release* permite cerrar un módulo o archivo que no va a volver a ser utilizado permitiendo su liberación en memoria.

- *int (*fsync) (struct file *, struct dentry *, int);*

  Al existir operaciones de lectura y escritura asíncrona se requiere una función que permita una resincronización del flojo de trabajo de la aplicación o ejecución de tareas pendientes sobre un módulo para un nuevo uso o cierre del mismo. Esta acción la realiza *fsync* o *file sync.*

- *int (*aio_fsync)(struct kiocb *, int);*

  La versión asíncrona de *fsyn* se utiliza cuando se quiere sincronizar o terminar las ejecuciones pendientes sobre un módulo que se quiere utilizar en un futuro pero no para un uso inmediato. Se trata del mismo método que *fsync* pero ejecutándolo de manera no inmediata.

- *int (*fasync) (int, struct file *, int);*

  Este método permite la detección de un cambio sobre un *flag* de sincronización asíncrona, es decir, el método explicado anteriormente. Este método sólo se puede utilizar con ficheros o módulos preparados para comunicaciones asíncronas, detectando la sincronización total de un módulo asíncrono, o finalización de sus tareas pendientes.

- *int (*lock) (struct file *, int, struct file_lock *);*

  El método *lock* bloquea un archivo temporalmente, en la utilización de ficheros podría utilizarse en el bloqueo de lectura y escritura sobre unas fuentes durante su compilación. Sobre un módulo podría bloquearse la configuración del mismo durante la ejecución de sus tareas pendientes.

- *ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);*

  Este método es utilizado para realizar una lectura de distintos puntos de memoria dentro de un mismo módulo. El uso habitual de este método es el de realizar la lectura de distintos registros de memoria no simultáneos para el control de configuraciones u operaciones en curso.

- *ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);*

  Al igual que el método anterior realiza una modificación de distintas partes de memoria no consecutivas, permitiendo su uso en configuración simultanea de los distintos registros de configuración de un módulo. Habitualmente utilizado en módulos complejos o con la capacidad de realizar distintas operaciones de forma simultanea.

- *ssize_t (*sendfile)(struct file *, loff_t *, size_t, read_actor_t, void *);*

  Este método es muy útil en el uso combinado de distintos módulos hardware independientes a partir de un control software se los mismos. Permite una transmisión de información o datos entre distintos módulos de forma directa, es decir, sin tener que transferir la información desde el nivel hardware a la capa de aplicación para luego volver a transferirlo a la capa hardware.

- *ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);*

  Este método permite la transferencia de memoria de forma directa, al igual que el anterior con la diferencia de que esta transmisión se realiza en un bloque de memoria del tamaño de una pagina. Este método es más utilizado en la transferencia de información entre archivos más que con módulos o periféricos.

- *unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);*

  Este método permite la búsqueda de un espacio de memoria en el que se pueda realizar una reserva de porción de memoria acorde a un dispositivo subyacente. Habitualmente implementada por el gestor de memoria.

- *int (*check_flags)(int).*

  En el uso de operaciones complejas o algoritmos a nivel hardware se suelen utilizar distintos *flags* que indiquen el proceso de operación dentro de la operación o el estado de una tarea larga y compleja. Es por esto por lo que existe el método que permite leer distintos *flags* a la vez para obtener la información de por ejemplo el estado de una operación.

- *int (*dir_notify)(struct file *, unsigned long);*

  Este método es específico del sistema de archivos, permitiendo detectar cambios en un fichero sin que sus *flags* hayan sido leídos. Respecto a módulos se puede utilizar en la detección de la variación de una configuración causada por los distintos posibles resultados que pueda tener una función.

Muchos de estos métodos no tienen una utilidad urgente dentro del *scull*. Este es el motivo por el que se suelen implementar, salvo los básicos y necesarios, en función de las necesidades de comunicación que requiera el algoritmo o sencillez de programación del software ejecutado.

5.1.4 Estructura del fichero percibido

Para el uso de los archivos de forma correcta y por lo tanto de los métodos que se pueden ejecutar sobre cada uno es necesario conocer la estructura del mismo. Esta estructura se controla a partir de punteros, siendo creada en la apertura del kernel con la inicialización y siendo eliminada con el cierre del sistema gracias a la función de cierre del archivo. A continuación, se muestran los campos más importantes que tienen los ficheros indicando su uso y significado.

- *mode_t f_mode;*

  Este campo del fichero indica el tipo de operación a realizar sobre el mismo, como operaciones se entienden como principales operación de lectura y operación de escritura y ejecución. Esta funcionalidad se indica en unos bits internos, los cuales se consultan siempre antes de una operación, evita que se pueda escribir por ejemplo sobre un periférico de lectura del valor de los switches del sistema.

- *loff_t f_pos;*

  Este campo se almacena en *64b*, almacena la posición a la que esta apuntando dentro del fichero, necesario para las funciones de lectura y de escritura, también en la de movimiento del puntero para saber donde se encuentra y realizar un correcto cálculo del avance requerido para llegar al siguiente punto sobre el que leer o escribir.

- *unsigned int f_flags;*

  Este campo almacena los distintos *flags* del archivo. Estos *flags* indican el tipo de operación permitida en el momento, como puede ser el de solo lectura por estar bloqueado *o_rdonly*, el *flag* de no bloqueo o *_nonblock* o el que indica si el fichero está sincronizado o tiene alguna tarea en ejecución o pendiente de ejecución *o_sync*. Permite identificar la forma de trabajar con un archivo respecto a su estado en el tiempo, no respecto a las características intrínsecas del mismo como puede ser el primer campo explicado en este apartado.

- *struct file_operations *f_op;*

  Para poder conocer las distintas operaciones o métodos a ejecutar sobre un fichero se almacena el puntero que apunta a la estructura de operaciones **fops*. Este puntero permite conocer las operaciones disponibles una vez se ha activado el módulo con la función de inicialización, esta información se extrae del kernel, sin guardarlo en memoria para poder realizar cambios en las operaciones si fuese necesario.

- *void *private_data;*

  Al inicializar un archivo desde una aplicación apunta a *NULL*, para su uso es necesaria la realización de una apertura del mismo, en este proceso, el puntero pasa de apuntar a *NULL* a apuntar al espacio de memoria privada o memoria asignada al módulo o fichero en el espacio de usuario. Permite conservar información de estado del mismo.

- *struct dentry *f_dentry;*

  En este campo se almacena una estructura asociada al periférico, que permite al *driver* tener acceso a la estructura *inode*, que se explica en el siguiente apartado.

En un complejo sistema de archivos como es el sistema *Petalinux* generado aparecen muchos más campos con distintos usos y utilidades, en este apartado se han seleccionado los más importantes que tienen relación con el control de periféricos hardware.

5.1.5 Estructura del inode

La estructura *inode* es la estructura que utiliza el *kernel* del sistema operativo para la representación de in fichero de forma interna. Se utiliza para entender como fichero el espacio de memoria en el que se ha asignado un dispositivo, permitiendo desde el sistema operativo, tratar el periférico hardware como un fichero en vez de como un bloque de memoria.

Al depender del periférico subyacente puede variar la forma y campos de la estructura interna. Puede haber muchos tipos de estructuras de archivos y descriptores que apunten a una sola estructura *inode*. Solo dos campos son importantes para los drivers de un periférico, son los explicados a continuación.

- *dev_t i_rdev;*

  Como se ha explicado previamente uno de los parámetros más importantes del device driver es el número de identificación, ese parámetro se almacena en este campo de la estructura *inode*.

- *mode_t f_mode;*

  Este campo almacena la estructura interna que interpreta el *kernel* para la representación de los *char devices*. Almacena el puntero que apunta a la estructura del *device driver* al que hace referencia.

Para el número de identificación del device driver, almacenado en el campo *i_rdev*, está relacionado con los *major* y *minor numbers* del dispositivo, para interralacionarlos entre si a la hora de completar esta estructura se utilizan dos macros previamente definidas en las librerías de Linux.

- *unsigned int iminor(struct inode *inode);*
- *unsigned int imajor(struct inode *inode);*

5.1.6 Registro de un char device

En un sistema basado e Linux, el kernel utiliza de forma interna las estructuras *inode* y *cdev* para representar los char devices, por lo que para acceder a ese tipo de estructuras se necesita invocarlas. Para poder invocar estas estructuras es necesario realizar una asignación de memoria e inicialización de la misma, esto se puede realizar de dos formas distintas. Este será el método para obtener la estructura de forma independiente en tiempo de ejecución.

```
struct cdev *my\_cdev = cdev\_alloc( );
my\_cdev->ops = &my\_fops;
```

En este punto es necesaria la asociación de la estructura *cdev* a una estructura específica del periférico que se quiere utilizar. Para ello se inicializa como se muestra con el siguiente comando.

```
void cdev\_init(struct cdev *cdev, struct file\_operations *fops);
```

Es también necesario inicializar el periférico de operaciones, este permite conocer los métodos que podran ser aplicados al periférico, esta estructura es **fops*.

```
int cdev\_add(struct cdev *dev, dev\_t num, unsigned int count);
```

En la inicialización de la estructura es necesario indicar los números de identificación del dispositivo a inicializar, el primero de ellos corresponde al dispositivo sobre el que se quiere actuar. Esta operación puede fallar por distintos motivos, por ello, es importante asegurarse de que el dispositivo ha sido correctamente inicializado. Esto se puede probar realizando una detección del mismo, si no se encuentra hay que realizar una nueva inicialización del mismo. Si no se ha inicializado correctamente antes de la nueva inicialización será necesario eliminarlo de la lista de periféricos disponibles con la siguiente función.

```
void cdev\_del(struct cdev *dev);
```

Una vez se ha inicializado se debe registrar, la estructura interna a registrar se muestra en el siguiente fragmento de código. Esta estructura compone toda la información que incluye el device driver generado.

```
struct scull\_dev {
  /* pointer to first quantum set */
  struct scull\_qset *data;
  /* the current quantum size */
  int quantum;
  /* the current array size */
  int qset;
```

```
  /* amount of data stored here */
  unsigned long size;
  /* used by sculluid and scullpriv */
  unsigned int access\_key;
  /* mutual exclusion semaphore */
  struct semaphore sem;
  /* char device structure */
  struct cdev cdev;
};
```

La estructura que controla el dispositivo se muestra a continuación.

```
static void scull\_setup\_cdev(struct scull\_dev *dev, int index){
  int err, devno = mkdev(scull\_major, scull\_minor + index);
  cdev\_init(&dev->cdev, &scull\_fops);
  dev->cdev.owner = this\_module;
  dev->cdev.ops = &scull\_fops;
  err = cdev\_add (&dev->cdev, devno, 1);
  /* fail gracefully if need be */
  if (err)
  printk(kern\_notice "error \%d adding scull\%d", err, index);
}
```

Antiguamente, para el registro y la liberación de los device drivers se utilizaban las siguientes funciones. Actualmente este método está obsoleto realizándolo en la inicialización del sistema a partir de los métodos de inicialización y cierre del mismo.

```
int register\_chrdev(unsigned int major, const char *name, struct file\_operations *fops);
int unregister\_chrdev(unsigned int major, const char *name);
```

5.1.7 Métodos de apertura y liberación

Los métodos de apertura en incicialización del device driver son los que permiten la ejecución del resto de métodos, básicos en todos los device drivers. El método de apertura realiza la inicialización del sistema, variables asignación de memoria y configuración de los elementos software necesarios para el uso del periférico asociado.

Las funciones mínimas que tiene que tener son la detección de posibles errores en la inicializción o consecución de los distintos recursos necesarios para su uso, la actualización del puntero de operaciones para que el kernel conozca los distintos métodos a aplicar en el dispositivo y la asignación de memoria de kernel respecto a la memoria física, atravesando la capa de virtualización para poder realizar una escritura directa a memoria.

Para detectar que la inicialización ha sido correcta se ejecuta el método de apertura sobre el dispositivo en cuestión. Si el resultado de la inicialización es erróneo no permite su apertura devolviendo un puntero nulo, si es correcto ejecutará el método. Es por ello por lo que son muy importantes los mensajes de aviso sobre el código.

En la estructura *inode* se completa la información obtenida de *cdev* para el entendimiento del espacio de memoria reservado por esta operación como la apertura de un fichero, es en ese momento en el que el kernel del sistema puede acceder a la información del periférico. Si el kernel del sistema no ha accedido a esta información, no se podrá acceder a ella desde el espacio de usuario.

Una vez se ha abierto el dispositivo se puede acceder al área de memoria privada del dispositivo, al haber unido en la la apertura la memoria virtual donde se almacena el dispositivo y la memoria física real se pueden acceder al mapa de registros del dispositivo para su utilización.

El método de cierre o liberación implementa las acciones contrarias. Este método sirve para poder cerrar de forma segura este dispositivo sin que quede ningún área de memoria reservada haciéndola inservible para el sistema. Se puede decir que las dos funciones más importantes que tiene este método es la de cerrar debidamente los alojamientos de memoria y apagar el dispositivo hasta una nueva apertura.

5.1.8 Memoria de un device driver

Gracias a la gran variedad de dispositivos existentes en el mercado la gestión de la memoria es un tema que puede variar mucho en función del tipo de dispositivo que se pretenda controlar. La cantidad de memoria que requiera gestionar cada dispositivo depende de su uso, no es lo mismo el control de la memoria que requiere un sistema de almacenamiento de datos que el que requiere un módulo de control de Leds.

Los periféricos controlados en este proyecto no requieren una gestión específica de memoria ya que su control está basado en registros, por lo que su control se puede realizar a partir de accesos directos a memoria a partir de la dirección base de su mapa de registros y los desplazamientos necesarios para el acceso individual a cada uno.

Son los sistemas más avanzados basados en ficheros los que requieren la división de la memoria a gestionar en bloques. El sistema solo es capaz de cargar cierta cantidad de memoria para que sea accesible desde el procesador, es por ello por lo que existen distintos niveles de memoria. Estos niveles de memoria tienen distintos tamaños, a más cercano al microprocesador más pequeñas son las capacidades. Es por ello por lo que se realiza una segmentación de la memoria para su uso en bloques, definiendo tamaños de página, cada una de estas páginas se divide en fragmentos menores con los que poder trabajar sin requerir más recursos de los disponibles en el sistema. Pudiendo hacer de la gestión de memoria un tema muy complejo.

5.1.9 Métodos de lectura y escrtura

Los métodos de lectura y escritura son dos de los más importantes dentro de los disponibles para el device driver, es por ello por lo que son los primeros métodos a implementar. Estos métodos realizan la transferencia de información desde el hardware hasta el espacio de memoria del kernel, haciéndolo disponible para las aplicaciones que corren en el espacio de usuario. Al ser métodos muy parecidos, las cabeceras que implementan también los son. Estas se muestran en el siguiente código.

```
ssize\_t read(struct file *filp, char \_\_user *buff, size\_t count, loff\_t *offp);
ssize\_t write(struct file *filp, const char \_\_user *buff, size\_t count, loff\_t *offp);
```

En estas cabeceras se observan los parámetros a utilizar a la hora de ejecutar dichos métodos. El primero de los punteros, *filp*, es el puntero al dispositivo sobre el que se quiere leer o escribir, habitualmente estos dispositivos se definen con la ruta de acceso dentro del sistema operativo, obviamente una vez han sido abiertos e inicializados. La variable *buff* indica el espacio de memoria que se pretende escribir o sobre el cual se pretende leer y almacenar los datos de la transferencia. Estos datos son contados por el método, devolviéndolo como variable *count*, esta variable tiene un significado en función del valor que devuelva, el valor debe ser el del número de datos leídos o escritos, si no se corresponde es que ha habido un error en

la operación. Por último, el parámetro *offp*, se trata de una variable que permite realizar desplazamientos sobre el puntero de escritura o lectura, se entiende como un offset sobre la posición de memoria base del dispositivo.

Estas funciones no pueden ser ejecutadas en el espacio de usuario, únicamente desde el kernel realizando la transferencia de información entre las capas hardware y kernel. Para poder realizar transferencias de información entre la capa de kernel y la de usuario se utilizan dos funciones cuyas cabeceras se muestran a continuación. Estas cabeceras están definidas en las librerías de Linux para poder ser utilizadas como estándar en este tipo de sistemas.

```
unsigned long copy\_to\_user(void \_\_user *to, const void *from, unsigned long count);
unsigned long copy\_from\_user(void *to, const void \_\_user *from, unsigned long count);
```

Los métodos de lectura y escritura se pueden implementar de diversas formas. Al tratarse de transferencias de memoria el tratamiento de estos datos está ligado a la gestión de memoria del dispositivo, más adelante se muestra el código de escritura utilizado en el device driver creado para este proyecto, basado en un bucle de transferencias directas a memoria de *32b* de ancho.

Los sistemas Unix suelen tener dos tipos distintos de llamadas al sistema para la realización de operaciones de lectura y escritura, se trata de *vread* y *vwrite*, mencionadas en el apartado de métodos de los *device driver* con la que se pueden transferir bloques de memoria a distintos lugares del espacio de memoria de dispositivo. Estos métodos no se han utilizado ya que no son necesarios por la complejidad de los dispositivos a controlar. Estas funciones utilizan la estructura *iovec* para relacionar los array de bloques de memoria y direcciones sobre las que escribir.

5.1.10 Interrupciones

Las interrupciones entran dentro de las funciones de optimización del software, es decir, una vez se ha validado el sistema y se ha comprobado que funciona correctamente, se pasaría a la fase de optimización es en este punto en el que entran las interrupciones. Las interrupciones son funciones que permiten la ejecución de ciertas tareas de forma asíncrona a partir de un suceso hardware o software previamente definido.

En este caso las interrupciones han sido previamente configuradas en el *block design* del proyecto en el cual se concatenan las interrupciones de los distintos elementos incluidos, como son los módulos de procesamiento y control de los GPIOS.

En la documentación del sistema aparece como se debe realizar la configuración hardware del dispositivo para su posterior uso, en el se han de identificar unos número de interrupción con los periféricos añadidos al sistema, ya que no son las únicas interrupciones que pueden ejecutarse. Incluyendo en este bloque todas las del sistema operativo.

En este caso hay un *bug* en el sistema y no se corresponde la información de la documentación con la realidad, por lo que es necesaria la realización de una función de detección del número identificador de cada uno de los periféricos hardware forzando una interrupción en la inicialización. Para ello, después de configurar las interrupciones en el controlador general de interrupciones y las interrupciones específicas del dispositivo se fuerza un cambio de valor para detectar el número de interrupción. La ejecución de esta funcionalidad es necesaria una sola vez, ya que el *bug* es consistente y con ello el offset en los números de identificación de la interrupción. Esta función se explica en los siguientes apartados junto a la explicación de su código.

Para una correcta gestión de las interrupciones es necesario tener un método ordenado de pasos de ejecución, a partir de los cuales poder generar una plantilla de funcionamiento de la misma.

- *Detección del periférico que la ejecuta.*

  El primero punto antes de la ejecución de una rutina de interrupción es el filtrado de la misma, esto sirve para detectar que la función de interrupción la lanza el dispositivo correcto, si no es así no debe ejecutarse. Es importante fijarse que dentro de un mismo módulo pueden convivir en paralelo distintos sistemas que lancen la ejecución, es importante que sea el correcto si esto ocurre.

- *Detección de acción a ejecutar.*

  Como interrupciones se pueden realizar dinastas acciones, es importante realizar una lectura de la memoria de configuración del dispositivo interrumpido para detectar sus necesidades. En función de esta necesidad realizar la escritura sobre los *flags* que permiten la realización de ciertas funciones, no realizarlas desde la rutina de interrupción para evitar que se alargue.

- *Desactivar el flag.*

  Para poder realizar otra interrupción es necesario desactivar el *flag* de la misma, esto debe hacer al principio o final de la interrupción teniendo muy en cuenta la posible velocidad de interrupción de la misma para que no interrumpa antes de la finalización de la rutina de interrupción.

5.1.11 Semáforos y barreras software

Los semáforos son variables especiales que constituyen un método para permitir y restringir accesos a recursos compartidos como pueda ser el device driver o regiones específicas de memoria. Sirve para realizar una reserva o bloqueo del periférico mientras esta siendo utilizado por una aplicación, evitando así que sea utilizada por otra a la vez pudiendo producir errores de uso. Habitualmente estas variables son de tipo *booleano* para indicar si se puede o no realizar una operación, indicando da disponibilidad o no del recurso.

Este tipo de recursos son los necesarios para poder implementar una posterior programación *multihilo*, en la cual se tienen distintos procesadores para la ejecución de un mismo programa. En este caso se han de implementar sobre el periférico de procesado, para poder evitar que se transfieran datos mientras el periférico está realizando un procesado de datos.

Un tipo de aplicaciones que recurren a esta gestión son las barreras software. Método utilizado en el device driver para evitar que en la configuración del dispositivo pueda intervenir otra aplicación evitando una correcta configuración y manipulación.

Estos bloqueos de memoria realizan pequeños paquetes de operaciones atómicas respecto a los recursos que se quieren acceder, en este caso se trata de memoria. Para crear estos bloques atómicos de acceso a memoria lo primero que se tiene que hacer es ejecutar una función que permita detectar si esta siendo utilizada. En el caso en el que esté siendo utilizada esta memoria frena la ejecución del programa como una espera activa sobre el hilo de ejecución, si no está activa produce un bloqueo sobre la misma permitiendo únicamente a este hilo el trabajo sobre ella. Para su liberación de utiliza la función de *restore*, permitiendo la liberación del recurso cerrando el paquete de operaciones atómicas.

```
spin\_lock\_irqsave (&q2->queue\_lock, irq\_flags);
spin\_unlock\_irqrestore (&q1->queue\_lock, irq\_flags);
```

A la hora de escribir directamente en memoria, como pueda ser la escritura de un registro, se utiliza funciones *rmb()*, estas funciones se denominan funciones barrera, son utilizadas sobre todo en sistemas *multicore*, cuando se pretende acceder directamente a memoria física y se quiere evitar problemas de solapamientos de escritura.

Estas funciones indican al compilador que no se cambie el orden de los accesos a memoria para la optimización de memoria en el programa, se tienen distintos tipos de barreras que se pueden usar para este fin. Existen distintos tipos de barreras.

- *Barreras generales.*

  Este tipo de barreras impiden al compilador reordenar las distintas operaciones para la optimización del código, la función que se utiliza es *barrier();*

- *Barreras obligatorias.*

  Se trata del tipo de barrera más utilizada en la escritura sobre registros de periféricos.

  mb(); Barrera que frena todo tipo de accesos por parte de otros hilos.

  wmb(); Barrera que frena accesos de tipo escritura por parte de otros hilos, garantiza el orden en accesos de escritura.

  rmb(); Barrera que frena accesos de tipo lectura por parte de otros hilos, garantiza el orden en accesos de lectura.

- *Barreras condicionales.*

  La funcionalidad es la misma que antes asegurando la integridad de orden únicamente en un hilo o el orden de funciones ejecutadas por un procesador, no entre procesadores.

  smp_mb(); Barrera que frena todo tipo de accesos por parte de otros hilos.

  smp_wmb(); Barrera que frena accesos de tipo escritura por parte de otros hilos, garantiza el orden en accesos de escritura.

  smp_rmb(); Barrera que frena accesos de tipo lectura por parte de otros hilos, garantiza el orden en accesos de lectura.

- *Barreras implícitas.*

  Son configuradas especificas para un tipo de procesador en especial, no son de interés.

- *unsigned int imajor(struct inode *inode);.*

5.2 Dependencias de espacios

La base de este proyecto ha sido conocer que es el *espacio del kernel* y el *espacio de usuario*, desde sus parecidos hasta sus diferencias, un conocimiento claro de estos elementos será necesario para avanzar en el proyecto con agilidad sin cometer errores.

A continuación, se explica en que se basan cada uno de los espacios y que características tiene cada uno terminando con una comparación entre los dos marcando la importancia de la existencia de cada uno.

Dentro de la arquitectura del sistema se tienen distintos apartados, como pueda ser el hardware o el firmware, en este punto se pretende explicar los dos espacios básicos en los que se puede dividir el apartado software, uno de mayores privilegios y más complejo y otro más común para un usuario genérico.

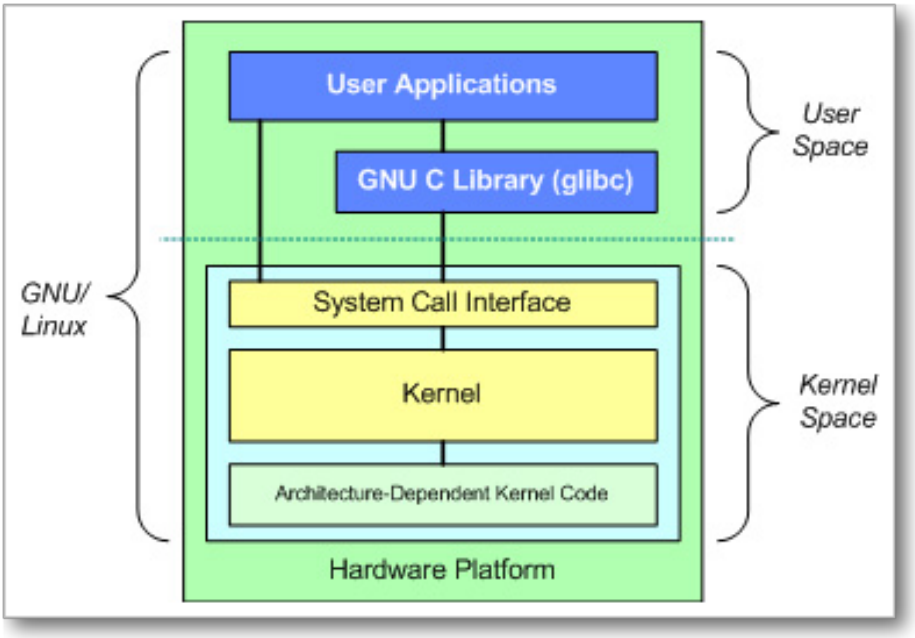

Figura 5.1: Diferencia entre espacios de usuario y kernel.

5.2.1 Espacio Kernel

El **espacio kernel** del sistema operativo es el espacio que se encarga de la gestión de los recursos de los que dispone el sistema, como pueda ser la ejecución de procesos, gestión de memoria, dispositivos y llamadas al sistema de forma estable respecto al tiempo de ejecución y el bloqueo de las mismas si es necesario.

Será en este espacio en el que se incluyan los device drivers, que se pueden definir como *elemento software perteneciente al espacio kernel que es utilizado para el control y gestión de las comunicaciones entre el hardware y el espacio de usuario o espacio de aplicación.*

Constituye el espacio de usuario privilegiado del sistema, al que es necesario acceder para la instalación de programas y control de los recursos de forma avanzada pero no para una ejecución habitual de aplicaciones.

Las características más importantes que tiene el espacio kernel son las siguientes:

- Control de los recursos básicos del sistema como pueden ser tiempo, memoria o capacidad de procesamiento.
- Control de los periféricos que dan soporte al sistema, desde su control hasta las comunicaciones con los mismos de forma ordenada y estable.
- Permite a distintos usuarios compartir recursos y ejecutar procesos específicos para la gestión de recursos y permisos.
- Proporciona un sistema ordenado y estándar de archivos con los que administrar el almacenamiento de la información del sistema y del usuario.

5.2.2 Estructura de usuario

El **espacio de usuario** del sistema es el espacio general en el que se permite la ejecución de procesos por parte de un usuario con unos privilegios previamente especificados, se trata de un espacio de trabajo seguro que permite la ejecución de ciertos procesos o acciones sobre regiones de memoria predefinidas en las que no se puede interaccionar directamente con el hardware.

Será en este espacio en el que un usuario genérico trabaje la gran parte de su tiempo de trabajo ya que habitualmente se permiten todos los procesos u ordenes más genéricos del sistema. Las características más importantes del espacio de usuario se muestran a continuación.

- Espacio de trabajo seguro para la ejecución de procesos de usuario.
- Espacio ideado para la ejecución de procesos controlados respecto a periféricos utilizados y regiones de memoria a la que se pueda acceder.
- Permite la gestión de los archivos e información de usuario.

5.2.3 Comparación entre espacio de usuario y kernel

La idea principal es que el *espacio de usuario* es un espacio seguro en el que se deben ejecutar las aplicaciones que han sido desarrolladas y testadas previamente, las cuales pueden llegar a ejecutar otras aplicaciones del *espacio kernel* en el caso en que se requiera realizar algún tipo de acceso específico y controlado a hardware, memoria o gestión de aplicaciones.

El *espacio kernel* es un espacio ideado para el desarrollo de aplicaciones o gestión del sistema por parte de un usuario avanzado, esto se debe a que un uso indebido de los recursos puede estropear el funcionamiento del mismo creando inestabilidades que pueden llegar a hacer que el sistema deje de funcionar.

El *espacio de usuario* es un espacio ideado para el trabajo de forma segura por parte de cualquier tipo de usuario, cuenta con protecciones que permiten evitar inestabilidades en el sistema base. Se trata del espacio general en el que debe permanecer un usuario genérico salvo en el momento de instalación y desnstalación de herramientas del sistema.

5.3 Explicación del funcionamiento

En este apartado se explica uno de los device drivers que se ha creado para la comunicación con los distintos periféricos del sistema. En este caso se ha escogido el driver de control de los switches.

En un primer momento es importante conocer el dispositivo sobre el que se pretende actuar, para ello es importante recabar información del mismo, en este caso de la página de Xilinx, o en el caso de ser un periférico propietario conocer bien su interfaz, lo aconsejable es, conocer bien su mapa de registros y posibles bloques de memoria que pueda tener.

En el caso del device driver de control de los switches, es importante conocer su mapa de memoria, en el que incluye las zonas de memoria y los distintos registros hardware que controlarán el dispositivo. A continuación, se muestra una imagen del mapa de memoria del periférico hardware que explicará el porqué de algunas características.

5.3.1 Estructura de control de un device driver

Dentro de los aspectos más importantes y generales dentro de la generación de device drivers será la de crear una buena estructura del device driver, se trata de una estructura con los distintos elementos, en este caso software del hardware a controlar, es decir es importante que si para la comunicación con el dispositivo se requiera leer de una memoria, el device driver, incluya un puntero que identifique la memoria de la que se quiere leer bien dimensionada respecto a la que se quiera leer.

Dentro de la estructura específica de cada device driver, tiene que tener ciertos elementos, el primero será un puntero a una estructura que se define, como se ha comentado anteriormente, en una de las librerías de Linux, será la estructura **cdev*** en esta estructura se almacenan los datos que el device driver respecto al identificación del device driver en el sistema operativo y el resto de los device drivers, esta estructura es inicializada en el proceso de arranque del sistema operativo sobre la función de inicialización.

Por la capa de virtualización o abstracción que genera el sistema operativo es importante saber que para la comunicación con el hardware las direcciones reales que tiene el periférico al que se va a ha acceder y las direcciones a las que va acceder la aplicación final no son las mimas, por ello será importante almacenar cual es la dirección real hardware y cual será su traducción después de la virtualización, este segundo valor se obtiene en la función de inicialización del device driver, ejecutada en el arranque del sistema operativo.

Para el trabajo con interrupciones es importante crear flags que permitan almacenar el valor de la interrupción asociada respecto al sistema operativo y algún flag más como pueda ser uno de test que puede ayudar en el momento de la inicialización y primer descubrimiento de la interrupción como se muestra más adelante en el código de la función de inicialización.

Los flags son necesarios, se entiende como flags variables de poca cantidad de memoria que permiten almacenar información puntual sobre algún aspecto, en el caso de los switches, podría utilizarse para saber, si los datos que están en el device driver son los actuales o hay que actualizarlos, elemento que se podría por ejemplo observar desde una función de interrupción para saber si es necesaria la actualización

Si se trabaja con interrupciones es aconsejable crear espacios de memoria para un acceso rápido a la misma sin tener que esperar a que se tenga que leer desde el hardware, manteniendo parte de información actualizada en el software, eso dependerá de la cantidad de información que se necesite almacenar en función de las características del sistema y la agilidad de la que se le quiera dotar.

En este punto es importante pensar bien como se decide trabajar, ya que los consejos extraído de los textos del libro *Linux device drivers* de *Rubini* es tratar este aspecto como lo trataría el sistema operativo, utilizando paginación y con tamaños máximos previamente configurados, si la aplicación no requiere exceso de memoria como puede ser la explicada en este capítulo se puede tratar con un formato más orientado a registros que permite el acercamiento entre la capa más firmware del kernel con el hardware asociado. Para este dispositivo se ha creado un array en el que se almacenan los datos que se han detectado mantenido el valor actual de los switches.

5.3.2 Estructura de operaciones

La estructura de métodos o la estructura de operaciones es uno de los apartados más importantes que compone el device driver ya que en esta estructura se dota de funcionalidad al sistema, es en este punto en el que se decide que acciones se podrán realizar desde el espacio de aplicación y por defecto desde la aplicación final sobre el hardware controlado.

Habitualmente se pretende trabajar con el periférico hardware desde la capa de aplicación como si se tratase de un fichero normal, por lo que para poder trabajar con el, desde la aplicación de usuario será importante poder abrirlo y cerrarlo como cualquier otro fichero software, una vez se pueda abrir correctamente, método que no tiene mucha complejidad al ser siempre igual, se idearán los métodos para poder trabajar con él.

Los métodos de trabajo más utilizados con ficheros son el método de lectura y escritura, es por ello por lo que estos dos métodos serán importantes dentro de la estructura de operaciones. En función del tipo de dispositivo a controlar estos métodos realizarán las mismas acciones pero de distintas formas ya que no es lo mismo leer de un periférico de switches que de un procesador de video.

Por último es importante que se puedan realizar comunicaciones asíncronas con el periférico hardware desde el espacio de usuario de aplicación, para por ejemplo, poder ver el estado del periférico o modificar algún parámetro de sus procesos como la modificación de flags leyendo o escribiendo directamente de memoria, sobre todo si se trata de un periférico complejo y con distintas funcionalidades.

Si en el caso de que se quiera gestionar un grupo de funciones con privilegios para su acceso como desarrollador o como gestión de incidencias es aconsejable, añadir en la estructura del device driver, vista en el apartando anterior, una contraseña que evaluándola en el inicio de las funciones permitiendo añadir capacidades extra o acceso a registros de configuración extra del periférico.

5.3.3 Métodos de inicialización y cierre

La inicialización del device driver será el punto de inicio de uso del mismo, por lo que será el proceso de configuración y preparación del device driver para el uso de los métodos explicados anteriormente, a lo largo de esta función de inicialización será muy conveniente utilizar mensajes de depuración, indicando en el punto en el que se está en cada momento y devolviendo mensajes que permitan entender con claridad el motivo de fallo para poder subsanarlo y avanzar sin errores.

Estas funciones de inicialización y cierre del device driver se ejecutan en el encendido del sistema y el apagado del mismo, para ver estos mensajes será importante haber configurado el sistema con el comando de *earlyprintk* como se explicó en la generación del sistema.

El primer paso que se debe realizar a la hora de ejecutar el device driver es la asignación de un **Major number** que no coincida con ningún otro **Major number** asociado a otro device driver, ya que se trata de un número de identificación que debe ser único, pudiendo aparecer problemas al abrirlo si no se ha respetado esta condición. Para una correcta asignación lo ideal sería mirar antes de compilar el device driver cuales existen dentro del sistema y una vez se tiene introducirlo directamente o generar un sistema de almacenamiento dinámico, sistema eficaz pero mucho más complejo basado en mirar el resto de **Major numbers** de los device drivers asignando uno no utilizado.

Una vez se asigna el **Major number**, se pasa al almacenamiento de los datos del device driver, como puedan ser el nombre o la dirección física, datos que se imponen desde usuario ya que la dirección harware está fijada en mi caso en el *block design* y el nombre del dispositivo.

Para poder utilizar el device driver se debe asignar espacio para la estructura, este es uno de los puntos más importantes por lo que es importante detectar un posible fallo si lo hubiese, también será importante realizar la petición de memoria a la que se quiere acceder dentro del espacio hardware, para ello se asigna memoria y una vez se tenga la memoria se reasignará para cambiarla de posición detectando donde está respecto al espacio de usuario.

También se deberá asignar la necesaria para poder trabajar con las transferencias de datos que requiera la aplicación, en este punto se asociarán las estructuras del device driver y la de operaciones o métodos

para poder ejecutarlas correctamente.

Para la asignación de la interrupción se ha creado una función que aplicación una *Interrupción Artificial* para evitar un *Bug* de la herramienta *petalinux* en la asignación del ID de la interrupción, una vez se detecta correctamente se asocia el número con la función a ejecutar, configurando los registros de interrupción del periférico.

Para el cierre del device driver es más sencillo, se basa en liberar toda la memoria asignada en la inicialización del device driver de forma ordenada para evitar problemas en la estructuración de los datos mientras se realizan las liberaciones, por ello, lo aconsejable es liberar memoria en orden inverso a como se ha asignado en la inicialización.

5.3.4 Métodos para la comunicación con espacio de usuario

Los métodos habituales que se deben ejecutar sobre un fichero los marcan el tipo de periférico y la complejidad del mismo, en este caso se ha creado un método de apertura y de cierre para poder trabajar con el archivo, también uno de lectura y otro de escritura, en casos como la creación de periféricos como los switches o los leds no tiene sentido utilizar el de lectura y el de escritura, por lo que se utilizarán los necesarios, no todos. Por último se explicarán los de configuración y comunicaciones asíncronas que tienen gran utilidad en casos de lectura y escritura directa a memoria sobre los registros de configuración del periférico.

Estos métodos se utilizan o llaman desde el espacio de usuario y desde distintos lenguajes de programación por ello se ha de utilizar una cabecera de función específica para cada uno salvo para el de configuración que puede ser más específica en función del periférico pero lo aconsejable es estandarizar todo lo posible para que sea más accesible.

La traducción entre el espacio de usuario y el nombre de la función se realiza en la estructura de operaciones que se asigna al periférico en la inicialización, será en ese punto en el que se cree el método como tal sobre el objeto del device driver.

Para poder abrir el device driver se utilizará el método open, se trata de un método muy sencillo en el que se realiza un enlace entre el puntero que se pasa y el espacio de datos del device driver, uniendo el *fichero* que se ve desde la aplicación de usuario a la dirección de memoria del espacio del *kernel*. El método close será opcional, ya que se tiene el close que libera el espacio reservado para el device driver, en el método close se podría simplemente borrar la el valor de apuntamiento del espacio de datos del fichero, si se pretende acceder en múltiples ocasiones sobre el fichero es aconsejable no realizar ninguna acción sobre el método de cierre.

Una vez se ha abierto el fichero se pretende utilizar, leyendo o escribiendo sobre el mismo, para poder llevarlo a cabo de una forma estructurada lo primero que será necesario es una asociación entre el fichero que se quiere abrir y el fichero que apunta al device driver, en ese punto se pueden tomar distintos caminos en función de los datos que se quieran leer o sobre como se quiera hacer.

Estos métodos suelen ser sencillos basados en lectura o escritura directa a memoria del número de datos que se quiere transferir a partir de un offset respecto a la dirección de inicio, para la transferencia de datos entre *espacio kernel* y *espacio de usuario* será necesario el uso de unas funciones específicas de transferencia de datos, estandarizadas y explicadas en el siguiente apartado de explicación del código.

Por último, para la función de configuración del device driver también se utilizará un enlace con la estructura del device driver inicial para poder acceder a las posiciones de memoria del mismo y en función del valor de una constante que se le pase al device driver se pueda acceder al registro requerido a partir de una traducción por mapa de memoria.

Como funciones extra es aconsejable añadir métodos como el método *help* que permita conocer los métodos disponibles para ese periférico y cuales son los registros sobre los que se ha de escribir para una correcta configuración o información del device driver y de su funcionamiento.

5.4 Explicación del código

En este apartado se pretende explicar el código que se ha generado como device driver para el control de los periféricos, en función de como aparece en las fuentes escritas en C, haciendo mayor hincapié en los detalles comentados a lo largo del apartado anterior, específicos de los device drivers, realizando explicaciones en mayor detalle del código que no se utiliza en aplicaciones de usuario.

La estructura que se ha decidido realizar para los device drivers a términos generales para la organización de la información del sistema será la mostrada a continuación.

Para iniciar las fuentes, como en todas las escritas y utilizadas hasta el momento se inicia una cabecera que proporcione información general de la fuente que se ha generado, con un pequeño resumen de su funcionamiento, a partir de este punto se inician la inclusión de las cabeceras que incluyen información de la definición de estructuras especificas de los device drivers y las funciones que se van a utilizar, también se define la información general del device driver, su autor y el **Major number** que se utilizará.

Una vez se ha inicializado la fuente en función de lo que se va a utilizar, se definirá la estructura del device driver y la estructura de operaciones que se le asignará junto a las cabeceras de los métodos que se pretenden utilizar, empezando por los de inicialización y cierre llegando hasta las cabeceras de las funciones de usuario a implementar como puedan ser *read* y *write*, incluyendo funciones secundarias como será *CheckIrqID* que permite identificar el valor de interrupción del device driver.

En este punto se deben generar las funciones que se ejecutarán en el *espacio kernel* como son los métodos *___init* y *___exit* de inicialización y cierre del device driver con el encendido y apagado del sistema. Serán las aplicaciones de usuario las que se definirán en este punto, pudiendo ejecutarse desde el *espacio de usuario* como son *open*, *close*, *read*, *write* y *config*. Por último se definirán las funciones secundarias utilizadas en las funciones anteriores, este esquema se muestra en la imagen añadida a continuación.

5.4.1 Cabecera, librerías y definición de constantes

Para una correcta programación en un proyecto en el que pueden trabajar múltiples personas o se va a trabajar a lo largo del tiempo es importante una correcta organización de la información, para ello se requiere que aproximadamente la mitad de las líneas de código de un software sean comentarios de la funcionalidad del mismo, en este proyecto se han llevado a cabo esas buenas prácticas tal y como se ve a continuación. Lo primero que aparece según se abre una fuente de los device driver del sistema es una cabecera, con la estructura aconsejada por la herramienta *Doxygen* para la documentación de código.

```
// * Header

//.-------------------------------------------------------------------------.
//|                                                                         |
//|    |    |   /\   |    |   Trabajo de fin de Máster                      |
//|    |    |  /  \  |____|                                                 |
//|    |    | /____\ |    |                                                 |
//|    |____|/      \|    |   Master en Ingenieria de Telecomunicaciones    |
//|                                                                         |
```

```
//|-----------------------------------------------------------------------------
/**   @file        SwitchDriver.c
 *    @author      Alvaro Cortes Sanchez-Migallon
 *    @contact     alvarocsm.91@gmail.com
 *    @version     01 / A
 *    @date        10/04/2019
 */
//|-----------------------------------------------------------------------------
//|  Description:
/**   @brief       Driver de prueba para asimilar conceptos del funcionamiento
 *                 y creacción del device driver, de inicialización y primera
 *                 carga.
 *
 *                 09/03/2019 -> Comienzo con la definición de estructuras y
 *                 métodos de apertura y cierre (open and close) + printk
 *
 *                 Buscar para mejorar: Hacerlo más genérico
 *
 *-----------------------------------------------------------------------------*/
```

Las cabeceras de linux se almacenan en la carpeta linux de la estructura de carpetas creada por la herramienta *Petalinux*, esto se conoce por haber partido de una plantilla muy básica que permite conocer la localización de los elementos dependientes y la estructura básica del device driver, mucho más sencilla que la mostrada en este comentario del código.

Como se puede observar en el siguiente apartado de código en el último lugar se incluye una cabecera en la que se pueden añadir datos, en este caso es aconsejable crearla para almacenar el mapa de memoria o mapa de registros del periférico, también para limpiar el código almacenando la información utilizada en esta fuente, este apartado, por ejemplo, podría incluirse en la cabecera.

```
// * Global includes
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/interrupt.h>
#include <linux/irq.h>

#include <linux/fs.h>
#include <linux/ioport.h>
#include <linux/cdev.h>
#include <asm/uaccess.h>
#include <asm/irq.h>
#include <asm/io.h>

#include <linux/mutex.h>
#include <linux/ioctl.h>
#include <linux/spinlock.h>
#include <linux/wait.h>
#include <linux/sched.h>

#include <linux/of_address.h>
#include <linux/of_device.h>
#include <linux/of_platform.h>

#include <linux/unistd.h>
```

```
#include <linux/delay.h>

#include "SwitchDriver.h"
```

Como definiciones globales para el funcionamiento del device driver se incluyen el número de dispositivos hardware que se quiere controlar con este device driver, ya que si por ejemplo se tienen tres timers en los que el periférico es el mismo y mantienen una misma estructura de mapa de registros podrían utilizarse en estructura cambiando únicamente las direcciones de base utilizando una sola fuente como device driver de los tres periféricos.

También será muy importante conocer el valor de interrupción del sistema que se podrá incluir directamente al ser constante para el sistema, una vez claro se identifique por primer a vez.

```
// * Global defines
// Hardware parameters
#define XPAR_SWITCHDRIVER_0_IF_0_DEVICE 0
#define XPAR_SWITCHDRIVER_0_IF_0_BASEADDR 0x41210000

#define NUMBER_OF_SWITCHDRIVER 1

#define C_FIFO_SIZE 4

// Interrupt parameters
#define NUMBER_OF_IRQ 155
```

La definición de la información que se incluirá en el device driver es muy importante ya que será accesible y ayudará a identificar el device driver que se está generando desde el *espacio de usuario* para su futuro uso en aplicaciones.

```
// * Standard module information, edit as appropriate
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Alvaro Cortes Sanchez-Migallon");
MODULE_DESCRIPTION("SwitchDriver, device to test open, read, write and close methods");

#define DRIVER_NAME "SwitchDriver"
#define DEVICE_NAME "SwitchDriver@41210000"


// Driver parameter defintion
#define _MAJOR_             241
#define _NAME_              "SwitchDevDri"
```

Por último se generan las definiciones de las constantes que se utilizarán en los métodos generales, como en este caso el número de Bytes leídos cada vez que se ejecuta el método de lectura.

```
// * Specific functions defines
// Read function
#define NO_BYTES_RX 4
#define INIT_BYTES  0
```

5.4.2 Declaración de cabeceras de funciones

Para poder asignar las funciones a las estructuras que definen el device driver es necesario incluir las cabeceras antes de su uso para evitar que aparezcan problemas de compilación, a continuación, se muestra el código de la función de inicialización, ejecutada en el arranque del sistema y la de cierre, ejecutada en el apagado.

```
// * Driver kernel functions declarations
static int  __init initSwitchDevDri(void);
static void __exit exitSwitchDevDri(void);
```

Para las funciones que se ejecutarán como métodos del device driver tienen la siguiente estructura, en el apartado anterior se explica que para mantenerlas como funcione estándar, se tiene que mantener la cabecera mostrada, pudiendo, en la generación de estas funciones, permitir el uso de los parámetros pudiendo obviarlos si no tienen razón de uso.

Se incluyen las funciones secundarias que utilizan los métodos o las funciones de inicialización para en este caso la detección del número de interrupción para este device driver y la función de interrupción que ejecutará para la actualización de los datos.

```
// * Driver user functions declarations
int openDevDri(struct inode *, struct file *);
int closeDevDri(struct inode *, struct file *);
int cfgDevDri(int, unsigned long int, void*);
static ssize_t readDevDri(struct file *filp, char *buf, size_t count, loff_t *ppos);
//static ssize_t writeDevDri(struct file *, const char *, size_t , loff_t *);
static irqreturn_t SwitchISR(int irq, void* dev_id);

// Test functions
// CheckIrqID launch Led IRQ to test which irq number is
int CheckIrqID(SwitchDriver_defs_t* q);
```

5.4.3 Estructura de device driver y métodos

La estructura del device driver que se ha realizado para este device driver se muestra a continuación, en ella pueden observarse la estructura inicial del dispositivo que le da las características de identificación del device driver, esta estructura está definida en una de las librerías incluidas anteriormente.

```
typedef struct {
  // Device structure
  struct cdev*  my_cdev;
```

Los datos mínimos que se requieren para trabajar con el dispositivo serán el nombre del dispositivo con el que se quiere trabajar, para poder referenciarlo de forma externa desde las aplicaciones y la traducción de las direcciones, como se comentó previamente si los dispositivos hardware son iguales se puede hacer un control múltiple desde el mismo device driver, por eso la definición como array de datos.

En este caso se utiliza solo un periférico por device driver por lo que podría no ser necesario este tratamiento de datos, los datos de las direcciones que se incluyen en este apartado hacen referencia a la dirección hardware del dispositivo y a la dirección virtual del mismo después de la aplicación de la capa de virtualización, más adelante se explica como realizar un uso adecuado de estos valores.

```
// Device information parameters
int*    BaseAddresses[NUMBER_OF_SWITCHDRIVER];
void*   RealAddresses[NUMBER_OF_SWITCHDRIVER];
char*   names[NUMBER_OF_SWITCHDRIVER];
```

Respecto a las interrupciones se definen los valores mostrados a continuación, en el primero, *irq_id*, se debe asociar el valor de la interrupción asociada al periférico por el sistema operativo, para ello se ha creado una función llamada *CheckIrqID* que permite realizar una detección inicial de este valor. Para detectar que la función de interrupción se está ejecutando por una variación de los valores del switch y no por la función de identificación se añade la variable *irq_test*, por último se añade la variable de *dataOutOfDate*, por si se prefiere trabajar en modo *polling* a partir de los mensajes asíncronos de comunicación y poder saber cuando es necesaria la lectura de los datos del mismo.

```
// Device IRQ flags
int     irq_id;
int     irq_test;
int     dataOutOfDate;
// Device read parameters
```

La sección de la memoria del device driver ha sido adaptada al dispositivos que se utilizará, en este caso se trata de un periférico con un espacio útil de datos limitado ya que únicamente almacena un byte de datos, por ser representado cada switch por un bit, por eso sería suficiente con un char, pero se crea como genérico para poder hacerlo parametrizable por si se quiere utilizar más de un canal.

```
int     bytesRead;
// Device data memory
char*   Data[C_FIFO_SIZE];
```

Por último se puede añadir una contraseña que permita el acceso extra de algunos parámetros, en este periférico no tiene mucho sentido, pero en un periférico que pueda ser de procesado de datos, puede ayudar a acceder a un modo que permita realizar calibraciones o ajustes de los parámetros de muestreo del mismo.

```
// Device administration password
int     password;
} SwitchDriver_defs_t;
```

La estructura que almacena los *métodos* del device driver tiene el aspecto mostrado a continuación, como se puede observar, en la columna de la izquierda se observan los comandos que realizan acciones a nivel software en espacio de usuario y a su derecha la asociación con el puntero de la función que se pretende ejecutar en función del comando.

```
const struct file_operations fileOperations =
{
  .owner = THIS_MODULE,
  .open  = openDevDri,
  .read  = readDevDri,
  //.write = writeDevDri,
  //.close = closeDevDri,
```

```
  //.unlocked_ioctl = cfgDevDri,
};
```

El primer método presenta información sobre el device driver, es un método por defecto generado en este tipo de estructuras de datos, definido en las librerías mostradas anteriormente, el resto de métodos son completamente personalizables, es aconsejable dar la funcionalidad mínima como fichero pero depende del periférico y de la aplicación final.

5.4.4 Definición de funciones de *espacio kernel*

Las funciones que pertenecen al *espacio kernel* son las que se ejecutan automáticamente en el encendido y apagado del sistema por lo que serán las que permitan la detección del periférico, sin ellas no se podrá acceder al mismo. La primera función que se genera es **___init** que se ejecuta en el encendido realizando toda la configuración inicial y la reserva y asignación de recursos, como pueda ser la memoria requerida por el device driver.

En un inicio del programa, se debe definir una variable llamada *Status* que permitirá detectar si cada una de las funciones ejecutadas se ha realizado de forma satisfactoria o no y por lo tanto, si seguir con la función o abortar en función de la gravedad del fallo.

También es aconsejable generar mensajes de alerta para poder realizar una correcta depuración y saber en que punto se producen fallos si es que se producen, al ejecutarse en *espacio de kernel* se utilizará la función *printk* para mensajes de kernel. El primer dato importante es mostrar es el *Major number* que, aunque se pueda obtener una vez ha arrancado el sistema, ayuda a detectar un solapamiento entre periféricos.

```
// ** Driver init function
static int  __init initSwitchDevDri(void){
  int Status;

  // Inicializo variables globales.
  SwitchDriver_defs_t* pSwitchDevDri = &SwitchDevDri;

  printk("Inicializando módulo controlador de switches.\n");
  printk("Parámetros de inicio:\n");
  printk("\t- Major number: %d.\n", _MAJOR_);
```

Para el almacenamiento de los distintos datos específicos del periférico se tiene que reservar memoria, esta memoria se reserva de forma dinámica para no utilizar más de la necesaria, en el caso de la gestión de la memoria, será importante que en el siguiente método, se libere toda la memoria que se ha reservado en esta función.

Como se ha comentado en el apartado en el que se explica la estructura del device driver se tiene que reservar memoria para el nombre y la traducción de memoria, realizándose primero por ser un proceso necesario en todo device driver.

```
  // Add device name
  // First reserve name space
  pSwitchDevDri->names[0] = kmalloc((strlen(_NAME_)+1+2)*sizeof(char), GFP_KERNEL);
  // Save name
  // AHA pSwitchDevDri->names[0] = _NAME_;
  sprintf(pSwitchDevDri->names[0], "%s %d", _NAME_, 0);
```

```
// Print name
printk("\t- Nombre: %s.\n", pSwitchDevDri->names[0]);

// Add device address
pSwitchDevDri->BaseAddresses[0] = (int*) XPAR_SWITCHDRIVER_0_IF_0_BASEADDR;
// Print address
printk("\t- Dirección: 0x%x.\n",(unsigned int) pSwitchDevDri->BaseAddresses[0]);
```

Otro proceso básico y seguramente el más importante es el de reserva de la región del device driver, es en este punto en el que se utiliza el **Major number** creando el espacio que utilizará el device driver, para ello se utiliza la función *register_chrdev_region* que reservará espacio en el kernel para la generación del device driver. Para poder utilizar esta función se tiene que conocer el **Minor number**, el valor *0* ya que este número no tiene que ser único.

La identificación del device driver se realizará con el dato devuelto por *MKDEV(_MAJOR_, 0)*, el número de periférico controlado y el nombre asociado.

```
// Register char device region
Status = register_chrdev_region(MKDEV(\_MAJOR\_, 0), NUMBER_OF_SWITCHDRIVER, DEVICE_NAME)
    ;
// Print registration result
if(Status >= 0){ printk("Char device registrado correctamente.\n"); }
else { printk("Char device no registrado.\n"); }
```

Para la reserva de memoria de la región de memoria sobre la que se puede trabajar con el device driver se utilizará la función *request_mem_region*. Esta función realiza una reserva de memoria de en este caso *0x2C* bytes a partir de la dirección indicada como *BaseAddresses* para este device driver, la dirección indicada será la del hardware, siendo el espacio reservado a partir de ese punto. Es importante reservar la cantidad de datos correcta para poder escribir y leer de todos los registros, los cuales deben ser abarcados por esta cantidad de datos, en el caso en el que fuese poco tamaño de reserva, no se podría escribir o leer sobre esa región de memoria produciendo importantes errores de funcionamiento, muy difíciles e detectar.

Para poder conocer cual será la dirección virtual de memoria sobre la que acceder desde la aplicación ejecutada en el espacio de usuario, se realiza un movimiento de la región de memoria a partir de la función *ioremap*.

```
// Register char device memory region.
request_mem_region((unsigned long)pSwitchDevDri->BaseAddresses[0], 0x2C, DEVICE\_NAME);

// Print virtual address got
printk("Virtual address: %x.\n",(unsigned int) pSwitchDevDri->BaseAddresses[0]);

// Remap virtual memory reserved to our device real address.
pSwitchDevDri->RealAddresses[0] = ioremap((unsigned long)pSwitchDevDri->BaseAddresses
    [0],0x2C);

// Print device address
printk("Device address: %x.\n",(unsigned int) pSwitchDevDri->RealAddresses[0]);
```

Para crear el device driver se utiliza la función *cdev_alloc()* que reserva un espacio en el apartado de device drivers permitiendo el registro del mismo, en este punto se asociará al device driver recién generado, la estructura de métodos asociados al mismo, esta asociación se realza a partir de la estructura **cdev***.

Para la reserva de los datos del *espacio kernel* con los que trabajará el device driver se utilizará la función *kmalloc* reservando la memoria indicada dentro del kernel, para poder realizarlo hay que pasar los flags de permisos por no estar reservando memoria dentro del *espacio de usuario*.

```
//// Register driver to kernel
// Create device driver, SCULL, structure
pSwitchDevDri->my_cdev = cdev_alloc();
// Asociate my file operations module with device driver structure
pSwitchDevDri->my_cdev->ops = &fileOperations;
// Add device driver structure to kernel space
Status = cdev_add(pSwitchDevDri->my_cdev, MKDEV(_MAJOR_, 0), NUMBER_OF_SWITCHDRIVER);

// Allocate device data memory
pSwitchDevDri->Data[0] = kmalloc(C_FIFO_SIZE*sizeof(char), GFP_KERNEL);

// Print added confirmation.
if(Status >= 0){printk("Switch driver added correctly to kernel space.\n");}
else{printk("Switch driver not added correctly to kernel space");}
```

Por último, se añade la funcionalidad asociada a la interrupción, este apartado no se ha incluido en un primer momento ya que la generación de esta fuente ha sido un proceso iterativo en el que se han ido consiguiendo los distintos puntos antes de pasar al siguiente. En primer lugar, se puede observar la función de identificación de la interrupción asociada al device driver, en este momento se identifica el hardware respecto al software, forzando una interrupción como se explica al final de este apartado. En el momento en el que se detecta el valor de la interrupción se asocia el número detecta en el espacio de interrupciones del sistema, pasando a su correcta configuración para poder ser utilizada, en este punto se activa la interrupción sobre el registro general de interrupciones y el registro del propio *IPcore*, estos registros aparecen dentro del mapa de memoria asociado al periférico, cuyo funcionamiento aparece en la documentación de los *IPcores* utilizados de las referencias del proyecto.

Como se puede observar, al tratarse del device driver un software que pertenece al *espacio kernel* se pueden realizar escrituras directas a memoria del valor de bits, como es el caso, para la configuración de los registros de interrupción del periférico.

```
// Launch Led interrupt
CheckIrqID(pSwitchDevDri);

// Test this device interrupt value
if( request_irq ( pSwitchDevDri->irq_id, SwitchISR, IRQF_SHARED, DEVICE_NAME, /*_MAJOR_*/
      NULL) ){
  printk("Switch interrupt not registered.\n");
} else {
  printk("Switch interrupt configured.\n");
}

// Setting up interrupt
wmb();
iowrite32(0x01, pSwitchDevDri->RealAddresses[0] + GIER_REG);
iowrite32(0x01, pSwitchDevDri->RealAddresses[0] + IP_IER_REG);
wmb();
printk("Interrupt configured.\n");
printk("\n");

return 0;
```

```
}
```

La función **___exit** es la ejecutada en el apagado del sistema con la correspondiente liberación de regiones de memoria previamente reservadas y espacio de device driver dentro del *espacio kernel* del sistema.

Como se puede observar el orden de liberación de memoria de dispositivo es inverso al de asignación de la misma, empezando por la liberación del espacio de memoria del periférico. A continuación, se libera el device driver del espacio de drivers del sistema y la memoria virtual sobre la que se escribiría en el *espacio de usuario*, también se libera la asociación de la interrupción del espacio de interrupciones del sistema.

Por último, se libera la región de memoria de la estructura del device driver eliminando su registro como device driver, borrando así todo registro del device driver respecto al sistema.

```
// ** Driver exit function
static void __exit exitSwitchDevDri(void){
  // Variables
  SwitchDriver_defs_t* pSwitchDevDri = &SwitchDevDri;

  // Free device data memory
  kfree(pSwitchDevDri->Data[0]);

  // Unregistering device from kernel space
  printk("\n");
  cdev_del(pSwitchDevDri->my_cdev);
  printk("Dispositivo eliminado del espacio del kernel.\n");

  // Ummap char device memory
  iounmap(pSwitchDevDri->RealAddresses);

  // Unregister Switch interrupt
  printk("Elimino del registro de interrupciones la interrupción del switch.\n");
  free_irq(pSwitchDevDri->irq_id, /*_MAJOR_*/NULL);

  // Release char device memory
  release_mem_region((unsigned long)pSwitchDevDri->BaseAddresses[0], 0x2C);
  printk("Liberada memoria reservada para el dispositivo.\n");

  // Unload char device
  unregister_chrdev_region(MKDEV(_MAJOR_, 0), NUMBER_OF_SWITCHDRIVER);
  printk("Dispositivo eliminado.\n");

  // Free device names
  kfree(pSwitchDevDri->names[0]);
}
```

Una vez se tienen las funciones de inicialzación del dispositivo y de cierre se asociarán a la inicialización del módulo y a su cierre, este apartado también es estándar para el uso de las funciones de descubrimiento utilizadas en la aplicación de inicialización del device driver, explicada en el capítulo referente a la generación del sistema operativo.

```
module_init(initSwitchDevDri);
module_exit(exitSwitchDevDri);
```

5.4.5 Definición de funciones de *espacio de usuario*

Los métodos utilizados desde el *espacio de usuario* serán los que permitan, como se ha comentado en diversas ocasiones, el control del periférico hardware como si se tratase de un fichero, todo esto visto desde la perspectiva software de aplicación.

Para poder trabajar con un archivo lo primero es **abrirlo** para poder realizar transferencias de información hacia o desde el mismo, para abrirlo, lo único que se tiene que crear en la estructura *inode* que se pasará como datos al puntero del fichero, en este caso el puntero que apunta al device driver, habitualmente en la carpeta */dev/*

```
// ** Driver open function
int openDevDri(struct inode *inode, struct file *filp){
  filp->private_data = (void*) iminor(inode);
  printk("Dispositivo SwitchDriver abierto y preparado para su uso.\n");
  return 0;
}
```

Para poder utilizar la **función de lectura** del dispositivo es necesario utilizar los paráremos de la cabecera predeterminada que se muestra en el siguiente fragmento de código, los argumentos que se le pueden pasar a la función son, el fichero del que se quiere leer, en este caso se ha ignorado este argumento para simplificar el método, utilizando directamente el device driver seleccionado.

El puntero al buffer sobre el que se almacenarán los datos también se pasa como dato conocido desde la aplicación para poder trabajar con el una vez se realice la ejecución de la lectura, se pasa por referencia para evitar transferncia de datos aportando velocidad y agilidad a la función. Por último se pasan el número de bytes que se pretende leer y el offset respeto a la posición de base sobre el que se quiere leer.

En este aspecto para la lectura de datos el *offset* será *0*, se podría utilizar si el creador de la función de aplicación conociese perfectamente el espacio de memoria del driver, pero para simplificarlo se utilizarán otro tipo de lecturas para esa funcionalidad.

```
// ** Driver read function
// This function read in flip file, count number of char and save it in *buf *ppos
// is the offset to read in the filp file.
static ssize_t readDevDri(struct file *filp, char *buf, size_t count, loff_t *ppos){

  // Device structure
  SwitchDriver_defs_t* pSwitchDevDri = &SwitchDevDri;

  // If dont need update
  if( pSwitchDevDri->dataOutOfDate == 0 ){

    // Data hadn't change
    printk("El dato no ha cambiado.\n");

    // Return error
    return -1;

  }
```

La parte más importante que tiene esta función, básica para una correcta lectura es la trasferencia de datos entre espacios, el device driver, al actuar en el *espacio kernel* realiza una lectura directa de memoria para obtener los datos del periférico, estos datos no están disponible para el *espacio de usuario*.

La transferencia de datos al *espacio de usuario* desde el *espacio kernel* se realiza con la función **copy_to_user()**, a esta función se le pasa el puntero del espacio de usuario sobre el que se quieren almacenar los datos leídos y el número de datos a transferir.

Al utilizar interrupciones este proceso se encuentra en la función de interrupción, pero si no se utilizasen debería ir antes de la transferencia de los datos leídos, en la función de escritura se ve más claramente por no utilizarse interrupciones para esa función.

```
  // Copy from kernel space to user space buffer pointer
  copy_to_user(buf, pSwitchDevDri->Data[0], pSwitchDevDri->bytesRead);

  // update data flag
  pSwitchDevDri->dataOutOfDate = 0;
  printk("El dato ha sido actualizado.\n");

  // Return number of bytes read
  return pSwitchDevDri->bytesRead;

}
```

Una vez se ha transferido la información entre los distintos espacios se devuelve el número de bytes leídos, desde la perspectiva de aplicación, en función de la diferencia entre los valores leídos y los valores que se querían leer se puede conocer como ha ido la lectura.

La función de escritura será la función encargada de la escritura de datos desde el *espacio de usuario* hacia el *espacio kernel* para poder realizar la escritura sobre el periférico hardware, en este caso, los leds.

Para iniciar el método de escritura lo primero que se realiza la asociación del device driver para poder conocer sus datos, como pueden ser las direcciones sobre las que tiene que actuar, para realizar la transferencia se debe reservar tanto espacio como se quiera escribir, esto se conoce gracias a los argumentos de los que se dispone con la cabecera de la función de escritura, estos, son los mismos que los que tiene la función de lectura, explicada anteriormente.

```
// ** Driver write function
static ssize_t writeDevDri(struct file *filp, const char *buf, size_t count, loff_t *ppos)
    {

  int i;

  char *buffer;

  // Device structure
  LedDriver_defs_t* pLedDevDri = &LedDevDri;

  // Allocate memory space
  buffer = kmalloc(count*sizeof(char), GFP_KERNEL);

  // Print write confirmation
  printk("Escribiendo en la dirección %x, l valor %s.\n", (unsigned int) pLedDevDri->
      RealAddresses[0], buf);
```

Para realizar el cambio de espacio se utilizará la función **copy_from_user()**, a esta función se le pasan los argumentos del buffer copiado y el buffer reservado y el número de datos a escribir sobre el

periférico, una vez se tienen los datos sobre *espacio kernel* se escriben sobre los datos físicos del periférico hardware, liberando el array reservado para la acción de escritura antes de salir de la función.

Al igual que en la función de lectura, se devuelve el número de datos escritos, para un posible tratamiento de errores de escritura.

```
  // Copy from user space to kernel space buffer pointer
  copy_from_user(buffer, buf, count);

  // Write operation

  for (i = 0 ; i < count; i++){

    // Read peripheral data to buf buffer
    iowrite32((u32) *(buffer+(4*i)), pLedDevDri->RealAddresses[0] + sizeof(char) *(4*i));

    // if the value is not correct exit to read.
    if(*(buf+i) == NULL){
      break;
    }
  }
  kfree(buffer);

  // Return write value
  return i;

}
```

En el el periférico de control de los switches, se ha activado la señal de interrupción, por lo que se ha creado una función que se ejecute cuando se active esa señal, la variación de esa señal generará una interrupción sobre el espacio de interrupción del kernel del sistema, ejecutando, si está asociada como es el caso, la función correspondiente a la interrupción.

Lo primero que se hace, al igual que en el resto de métodos es asociar el device driver sobre el que se quiere actuar con esta función de interrupción, al tratarse de funciones de interrupción no se puede predecir exactamente el momento en el que se ejecuta, siendo muy importante que las acciones se realicen de forma consecutiva, ya que, por ejemplo, no se puede escribir sobre el espacio de dato del kernel antes de leer del hardware.

Para poder evitar que el compilador optimizase el código, variando la secuencia de accesos a memoria se utilizarán barreras software, se trata de elementos software que impiden la optimización de ciertas partes del código, su funcionalidad es parecida a la de los semáforos generando partes de código atómicas.

```
// * Interrupt function
static irqreturn_t SwitchISR ( int irq, void* dev_id ){

  // ** Variables

  // Switch device
  SwitchDriver_defs_t* dev = &SwitchDevDri;

  // Control variable
  int i;

  // Interrupt status
```

```
u32 irqStatus;
```

Una vez se ha iniciado la ejecución de la interrupción es conveniente asegurarse de que la interrupción ha sido provocada por el periférico que la está ejecutando, eso puede evitar problemas y es necesario sobre todo en el momento en el que un mismo device driver controla distintos periféricos, para saber cual ha generado la interrupción.

En función de si ha sido o no este periférico el que ha generado la interrupción, se borrará la interrupción asociada a este periférico y se generará un mensaje de alerta o se pasará a la ejecución de la funcionalidad deseada.

Por la existencia de la función de detección del valor de la interrupción para el sistema operativo, se ha creado el flag de test que pertenece al device driver, activo con valor *1* en la inicialización del sistema, esto evita que se ejecute la funcionalidad de la interrupción, en la inicialización.

```
// ** Read IRQ status in trigger register
rmb();
irqStatus = ioread32 (dev->RealAddresses[0] + IP_ISR_REG);
rmb();

// ** Is not SwitchDevDri IRQ
if( ! (irqStatus & 0x01 ) ) {

  // *** Write irqStatus value in trigger register
  wmb();
  iowrite32(irqStatus, dev-> RealAddresses[0] + IP_ISR_REG);
  wmb();

  // *** Assert message
  printk("Is not Switch IRQ.\n");

  // *** Return
  return IRQ_HANDLED;
}

// ** Is SwitchDevDri IRQ in initial test
if( dev->irq_test == 1 ) {

  // *** Write irqStatus value in trigger register
  wmb();
  iowrite32(irqStatus, dev-> RealAddresses[0] + IP_ISR_REG);
  wmb();

  // *** Assert message
  printk("Switch IRQ Test.\n");

  // *** Return
  return IRQ_HANDLED;
}
```

Una vez se ha comprobado que se trata de este periférico el que ha interrumpido y no se ha sido dentro de la inicialización para la detección del número de interrupción se ejecuta la funcionalidad, en este caso se trata de la interrupción de los switches, por lo que se realizará la lectura de los datos del switch.

Esta función únicamente realiza una transferencia desde los datos que están en nivel hardware al *espacio kernel*, cambiando el valor de un flag, que permite detectar el momento en el que los datos del

espacio kernel no están actualizados y hay datos preparados para ser leídos por la aplicación, esta función no llega a cambiar los datos de la aplicación ya que es independiente de la misma.

```
  // ** Is real SwitchDevDri IRQ

  // *** Assert message
  printk("Switch Real IRQ.\n");

  // *** IRQ functionality
  dev->dataOutOfDate = 1;

  // Read operation
  for (i = 0 ; i < C_FIFO_SIZE; i++){

    // Read peripheral data to buf buffer
    *( dev->Data[0] + i ) = ioread32(dev->RealAddresses[0] + (sizeof(char) * i));

  }

  // Save data read
  dev->bytesRead = i;
```

Para finalizar la función de interrupción es necesario limpiar el flag de interrupción activa para que pueda volver a cambiar volviendo a ejecutar esta función, si no se hiciese no se podría volver a actualizar el valor de los switches, por ello es muy importante usar las funciones de barrera, para asegurar que la limpieza, en este caso, se ejecuta antes de salir de la función de interrupción.

```
  // ** Clear IRQ
  wmb();
  iowrite32( ( irqStatus | 0x01 ), dev->RealAddresses[0] + IP_ISR_REG);
  wmb();

  // *** Assert message
  printk("Switch IRQ Cleared.\n");


  // ** Return
  return IRQ_HANDLED;

}
```

La detección inicial del valor asociado a la interrupción del periférico se realiza con la siguiente función, se ejecuta como se ha comentado anteriormente en el arranque del sistema, la idea, sería ejecutarla en un único arranque del sistema y sustituyendo el valor devuelto por la ejecución de la función, agilizando el arranque.

Esta función pretende generar una interrupción sobre el periférico de los switches, para ello se han probado distintos métodos comentados en el código. Lo primero que se debe hacer junto a la inicialización de las variable es indicar al device driver que si se produce una interrupción a partir de ese momento es porque se está realizando una prueba de detección, poniendo el flag de test a *1*

```
// ** CheckIrqID Autodetect device Irq
int CheckIrqID( SwitchDriver_defs_t* dev ) {
```

```
// *** Variables
// type returned by probe_irq_on/probe_irq_off functions
unsigned long bitmask;
unsigned int dataRead;
unsigned int irqStatus;
unsigned int cntTics;
unsigned int i;

printk("Inicio de detección de interrupción.\n");

// *** Start ID detec

// Enable test flag

dev->irq_test = 1;
```

Para saber que interrupciones están activas se ejecuta la función **probe_irq_on()**, ésta nos devuelve el valor del registro de las interrupciones activas hasta el momento.

El proceso de detección se produce por comparación, es decir, se observan las interrupciones detectadas en el inicio de la función, se provoca una interrupción y se vuelve a detectar cuales han sido detectadas a posteriori, en esa comparación se debe descubrir el valor de registro de la interrupción del periférico.

```
// *** Detect unregistered IRQ

// read irq enable unregistered

bitmask = probe_irq_on();

// assert message bitmask
printk("Bitmask value %d.\n", bitmask);
```

Una vez se ha leído el registro de interrupciones, es necesario conigurar el sistema de interrupciones del periférico para la detección de la interrupción, el proceso es la configuración de la interrupción como entrada, se activa el canal de datos que producirá la interrupción y se activa el registro general de interrupciones.

El orden no siempre es importante pero para evitar errores, es aconsejable seguir los pasos de configuración que aparecen en la documentación del periférico, para ello se utilizan barrera software de escritura.

```
// *** Enable peripheral IRQ

// Enable write memory barrier

wmb();

// Write in memory register to enable irq

// Configure input port as input

iowrite32 (0xffffffff, dev->RealAddresses[0] + GPIO_TRI_REG);

// Enable first channel to interrupt
```

```
iowrite32 (0x03, dev->RealAddresses[0] + IP_IER_REG);

// Enable general interrupt bit 31 <= '1'

iowrite32 (0x80000000, dev->RealAddresses[0] + GIER_REG);

// Disable write memoy barrier

wmb();
```

Al configurar la interrupción se debe forzar un cambio en el valor del registro de datos del periférico para que se ejecute, la idea más cómoda es la que se muestra en la siguiente porción de código.

La idea es leer el valor de los switches, variarlo sumándole 1 al valor leído e intentando escribirlo sobre el registro de datos, habitualmente los periféricos hardware tienen protecciones hardware o firmare para evitar escribir de señales que no se puedan escribir como sería el caso, por si acaso se ideó otro método más rudimentario.

```
// *** Force IRQ device modifing data register

// Read actually data value
dataRead = ioread32 (dev->RealAddresses[0] + GPIO_DATA_REG);

// Enable write memory barrier

wmb();

// Write in memory register to modify data.

iowrite32 ((dataRead + 1), dev->RealAddresses[0] + GPIO_DATA_REG);

// Disable write memoy barrier

wmb();
```

El segundo método ha sido la creación de una espera activa en la ejecución de esta función, creado un tiempo en el que se pueda realizar una modificación de los switches, forzando la interrupción a nivel usuario físico.

```
// *** Wait for IRQ status change
// init variables
irqStatus = 0;
cntTics = 0;

//
printk("Generar Interrupción.\n");
//usleep_range(100000000,100000001);
for (i = 0 ; i<50000;i++){printk("%d.\n",i);}
printk("10s.\n");

// read status register
do {
  irqStatus = ioread32(dev->RealAddresses[0] + IP_ISR_REG);
  printk("Status value %x.\n", irqStatus);
  cntTics++;
} while ((irqStatus == 0) && (cntTics<1e3));
```

```
  // Assert message to know if detect or timeout
  if( irqStatus == 1 ) {

    // IRQ detected
    printk("Interrupción detectada.\n");

  } else {

    // timeout
    printk("Tiempo para detección de interrupción agotado.\n");

  }
```

Al detectar el valor de la interrupción se almacena en el flag de *irq_ID* del device driver, utilizado en la función de inicialización del periférico para la configuración de esta interrupción.

```
  // *** Detect new irq active

  // Detect and store changes in Irq register

  dev->irq_id = probe_irq_off(bitmask);

  // Show ID number

  printk("El periférico %s ha sido asociado con el número de interrupción %d.\n", dev->
      names[0] ,dev->irq_id);


  // *** return Irq ID detected.

  return(dev->irq_id);

}
```

El último método que se va a explica es el de método de comunicaciones asíncronas o **método de configuración** del periférico. Esta función es muy importante en dispositivos complejos, por lo que se explica sobre el device driver del dispositivo integrado, creado por el grupo de trabajo del tutor del proyecto.

La idea de funcionamiento de este método es la escritura o lectura directa sobre memoria hardware del valor de distintos registros de control del sistema, se basa en un mapa de registros a nivel usuario y un case de selección y filtrado del dato a transferir.

```
int cfg_driver (int fd, unsigned long int ioctl_num, void* ioctl_param)
{
    unsigned int data;
    LowLevelq_defs_t* pdefs= &LowLevelq_defs;
    queue_t *q_rx1 = &(pdefs->rx_queue1);
    queue_t *q_rx2 = &(pdefs->rx_queue2);
    queue_t *q_rx3 = &(pdefs->rx_queue3);
    queue_t *q_rx4 = &(pdefs->rx_queue4);
    queue_t *q_rx5 = &(pdefs->rx_queue5);
    queue_t *q_tx = &(pdefs->tx_queue);
```

```
switch ((unsigned int) ioctl_num)
{
    case IOCTL_CMD_GET_IRQ_STATUS:
        rmb();
        data= ioread32(pdefs->RealAddresses[0] + XLOWLEVELQ_TOP_ADDR_ISR);
        rmb();
        *((unsigned int*) ioctl_param)= data;
        break;

    case IOCTL_CMD_SET_IRQ_STATUS:
        wmb();
        iowrite32((int) ioctl_param, pdefs->RealAddresses[0] + XLOWLEVELQ_TOP_ADDR_ISR)
            ;
        wmb();
        // dbg_printk("Core 0. Rx interrupt enabled.\n");
        break;

    case IOCTL_CMD_SET_IRQ_ENABLE:
        wmb();
        iowrite32((int) ioctl_param, pdefs->RealAddresses[0] + XLOWLEVELQ_TOP_ADDR_IER)
            ;
        wmb();
        // dbg_printk("Core 0. Rx interrupt enabled.\n");
        break;

    case IOCTL_CMD_SET_GIRQ_ENABLE:
        wmb();
        iowrite32((int) ioctl_param, pdefs->RealAddresses[0] + XLOWLEVELQ_TOP_ADDR_GIE)
            ;
        wmb();
        // dbg_printk("Core 0. Rx interrupt enabled.\n");
        break;

    case IOCTL_CMD_GET_IRQ_ENABLE:
        rmb();
        data= ioread32(pdefs->RealAddresses[0] + XLOWLEVELQ_TOP_ADDR_IER);
        rmb();
        *((unsigned int*) ioctl_param)= data;
        break;

    case IOCTL_CMD_GET_GIRQ_ENABLE:
        rmb();
        data= ioread32(pdefs->RealAddresses[0] + XLOWLEVELQ_TOP_ADDR_GIE);
        rmb();
        *((unsigned int*) ioctl_param)= data;
        break;

    /* case IOCTL_CMD_SET_RX_TH:
        wmb();
        iowrite32((int) ioctl_param, pdefs->RealAddresses[0] +
            XMB_RECEIVE_TH_REG_OFFSET);
        wmb();
        // dbg_printk("Core 0. Rx interrupt threshold configured at %d.\n", (int)
            ioctl_param);
        break;

    case IOCTL_CMD_GET_RX_TH:
        rmb();
        data= ioread32(pdefs->RealAddresses[0] + XMB_RECEIVE_TH_REG_OFFSET);
```

```
            rmb();
            *((unsigned int*) ioctl_param)= data;
            break; */

        case IOCTL_CMD_GET_STATUS:
            rmb();
            data= ioread32(pdefs->RealAddresses[0] + XLOWLEVELQ_TOP_ADDR_AP_CTRL);
            rmb();
            *((unsigned int*) ioctl_param)= data;
            // dbg_printk("Core 0. Mailbox Status: %d.\n", (int) *((unsigned int*)
                ioctl_param));
            break;

        case IOCTL_CMD_SET_STATUS:
            wmb();
            iowrite32((int) ioctl_param, pdefs->RealAddresses[0] +
                XLOWLEVELQ_TOP_ADDR_AP_CTRL);
            wmb();
            // dbg_printk("Core 0. Rx interrupt enabled.\n");
            break;

        case IOCTL_CMD_GET_RX_BUFFER_DATA_AVAILABLE:
            data= q_rx1->cnt;
            *((unsigned int*) ioctl_param)= data;
            break;

        case IOCTL_CMD_FLUSH_RX_BUFFER:
            queue_flush (q_rx1);
            queue_flush (q_rx2);
            queue_flush (q_rx3);
            queue_flush (q_rx4);
            queue_flush (q_rx5);
            break;

        case IOCTL_CMD_FLUSH_TX_BUFFER:
            queue_flush (q_tx);
            break;

        default:
            break;
    }

    return 0;
}
```

Capítulo 6

Aplicación de usuario

En este capítulo se explica el nivel más alto de la arquitectura, en este nivel se realiza el control de los periféricos internos del sistema, a partir de los drivers creados y explicados en apartados anteriores. La aplicación consiste en un sistema de localización a partir de ultrasonidos, ideadas para espacios interiores.

Como aplicación que se ejecutará sobre el sistema generado se ha ideado un sistema de localización para interiores basado en *tecnología de ultrasonidos*. Esta localización será determinada a partir de unas balizas que generarán cinco señales de ultrasonidos, que captará un micrófono.

Esta idea en principio ha sido ideada para el uso sobre un dron de laboratorio que lleva un controlador basado en un *SoC* de *Xilinx*, una *Zynq 7000*, sistema utilizado hasta el momento para el desarrollo del proyecto.

Las señales que llegan al micrófono se deben leer desde el *SoC* a través de *USB* y a continuación ser transmitidas hacia el módulo de procesado de datos, este una vez finalice su procesado, requerirá una lectura de unas tramas de datos correlados con los *cinco códigos* transmitidos por cada uno de los sensores de ultrasonido.

Será en este punto en el que se detectará la *diferencia entre máximos* determinando el tiempo de vuelo de la señal, indicando, a partir de un algoritmo de posicionamiento hiperbólico la situación del sistema o dron en función de la localización de la baliza de ultrasonidos empleada, permitiendo un guiado del mismo.

6.1 Fundamento matemático

Par poder entender la motivación y necesidad de la aplicación, es importante entender primero los *procesos matemáticos* de base que se aplican, para a partir de cinco señales poder llegar a conocer la localización del sensor. Este tipo de sistema está ideado para su uso en sistemas portables y móviles con capacidad de movimiento en interiores como puedan ser robots de interiores ideados para el transporte de cargas o para drones de localización de personas u objetivos. La característica común que han de tener los sistemas de uso es la capacidad de *muestreo* de señales a partir de sus sensores y ejecución de una aplicación sencilla de procesado de señal.

La *trilateración*, es un método matemático que permite determinar la posición relativa de un sistema receptor, a partir del uso de geometría análoga a la *triangulación*, a partir de la recepción de un *mínimo de tres señales* distintas emitidas por sistemas de transmisión conocidos. La *trilateración* es habitualmente utilizada en dos dimensiones ya que en un plano define un punto, pero la situación es *distinta en el*

espacio, al aumentar una dimensión se tienen que aplicar otro tipo de *trilateración*, por lo que aparecen los algoritmos de *trilateración esférica* y *trilateración hiperbólica*.

El algoritmo de *trilateración esférica* se utiliza habitualmente con tres emisores de señal muy separados en los que lo más probable es que el sistema permita únicamente un punto final real. Esto significa que este algoritmo con tres emisores devuelve dos puntos pero solo uno aceptable dentro de las condiciones del sistema, el sistema más utilizado dentro de este estilo es **GPS**, en este sistema se mejora la precisión a partir del aumento de puntos emisores. A mayor es el número de emisores mayor es la precisión reduciendo el espacio de error común entre las distintas trilateraciones entre las señales recibidas.

La *trilateración hiperbólica* está basada en la recepción de señales en las que se transmiten unas tramas conocidas desde *emisores síncronos*, esto hace que el inicio de transmisión se realice en el mismo instante. Es la diferencia entre las recepciones de estas señales las que permiten conseguir detectar una diferencia en tiempos de vuelo o tiempos de llegada y con esto sabiendo a que velocidad se transmiten permiten calcular la variación de distancia entre cada uno de lo emisores y el receptor, en la imagen mostrada a continuación, se puede observar el funcionamiento con frentes de onda para la detección de la posición.

Las soluciones aportadas por este algoritmo tienen *restricciones algebraicas*, al igual que la trilateración esférica respecto a que las soluciones pueden dar matemáticamente dos puntos distintos, siendo estas *restricciones físicas* las que permiten asegurar el punto final. Este algoritmo se soluciona matemáticamente con el **LSE** (*least squares error*) a partir del uso de las *series de Taylor* y del uso del algoritmo de *Gauss-Newton*. Es importante resaltar que estas no son las únicas opciones que existen para la solución de este problema.

La lógica del *algoritmo* comienza con la *fórmula de la esfera*, en este caso se tienen cinco emisores por lo que se referirán respecto al del centro, al tener cinco se obtienen cinco distancias, estas se pueden *referenciar* a la *primera* pensando que la primera más una variación de distancia es lo mismo que las coordenadas con una variación en una de las tres coordenadas.

Al conocer las posiciones de cada una de las cinco balizas emisoras y la variación de distancia calculada entre las mismas, queda un *sistemas de ecuaciones* que se puede resolver. La estrategia que se ha utilizado para la resolución es la de linealizar las ecuaciones respecto al emisor central y resolverlo por mínimos cuadrados. Este sistema se plantea como un conjunto de matrices, el cual, despejando llega a la siguiente ecuación. Será esta ecuación la que hay que implementar como finalidad de la aplicación, estas operaciones se resolverán en **C/C++** como procesado final *software* para el cálculo de la localización.

Este algoritmo puede presentar distintos *problemas* o *singularidades matemáticas* como el caso en el que la **matriz A** sea una *matriz 0*, esto significa que las tramas son coplanares y/o que el receptor se encuentra a la misma distancia de todos los emisores. La solución de este caso aparece con el diseño de la *radio-baliza* utilizada, que hace imposible que el receptor esté a la misma distancia de todas las tramas y el diseño de las *tramas* transmitidas, que al ser *ortogonales* evita que sean coplanares. La siguiente dificultad aparece en la calidad de la señal recibida y la precisión aritmética del algoritmo que se implemente en la aplicación, sobre todo cuando las variaciones de distancias es muy pequeña.

6.2 Objetivos de la aplicación

Esta aplicación tiene como objetivo el cálculo de la distancia respecto a la baliza de ultrasonido. Para poder llegar a esta aplicación se ha seguido un flujo de necesidades que permite desarrollar la aplicación.

- Validación de los módulos.

Lo primero que hay que hacer a la hora de desarrollar una aplicación es ver que los sistemas que se van a utilizar están preparados para ser utilizados, en este caso se tienen las comunicaciones serie con el hardware externo a partir del *USB*, también, la validación de los *device drivers* del sistema que se pretende montar.

Para esta tarea se utilizará directamente la *terminal* del sistema, permitiendo la lectura de los *periféricos* como si de ficheros se tratase, esto permitirá la validación del algoritmo.

- Validación del algoritmo.

  Para la validación del algoritmo será necesaria la captura de los datos en el sistema, pudiendo acceder en un primer momento de forma individual al micrófono almacenando la trama de datos en un fichero externo. *Matlab* será el programa que se ha utilizado para la validación del algoritmo, con un *script* se validan las operaciones de correlación y detección de las distintas distancias al módulo receptor.

- Validación de las comunicaciones desde la aplicación.

  Para poder observar que la aplicación es correcta es importante validar *las comunicaciones* entre la aplicación y el sistema operativo, para ello es necesario poder acceder a los distintos periféricos que de van a utilizar en la aplicación, en este caso, es importante acceder a las comunicaciones serie via USB.

  Estas permitirán la comunicación con el hardware en el que se tiene el micrófono para poder recibir las tramas de datos, básicas para el desarrollo de la aplicación y base de funcionamiento del sistema de localización.

- Validación del módulo correlador.

  En este punto se tiene que validar los datos obtenidos con una recepción del micrófono entre los obtenidos con el *script* de *Matlab* y el módulo de procesado de señal hardware a través del device driver. Este punto será el que decida si se conseguirá la aplicación o no.

- Detección de tiempos de vuelo.

  En el inicio del algoritmo se utilizan los tiempos de vuelo para el cálculo de la diferencia de distancias a cada uno de los puntos de la baliza desde el micrófono o punto de recepción.

- Cálculo de distancias.

  Para el cálculo de distancias hay que basarse en el algoritmo de la *trilateración hiperbólica* para ello es necesario primero hacer un estudio de los datos necesarios y del algoritmo a ejecutar. Siendo importante conocer las posibles transformaciones de datos para tener la mejor precisión posible, es muy aconsejable hacer este tipo de desarrollos con un buen *IDE* que permita ejecutar paso a paso para verificar que las operaciones se realizan en el orden adecuado y con los resultados adecuados.

Al tener una buena definición de los objetivos y el orden en conseguirlos es importante seguirlos y no avanzar en ellos sin haber validado los anteriores.

6.3 Explicación de funcionalidad

El código de esta aplicación ha sido desarrollado en *C/C++*, especialmente compilado con el *IDE* de depuración *software* de *Xilinx*, **SDK** (*software development kit*) para el sistema operativo que se ha

generado. Es necesario el uso de este sistema de desarrollo en vez de uno basado en un *Linux* cualquiera para que la compilación se realice confirme a la configuración del sistema y a sus características.

Para poder realizar una correcta depuración del sistema será necesario un diseño del algoritmo en función de las necesidades impuestas en el apartado de objetivos. A continuación, se muestra un esquema de desarrollo basado en módulos, que permite conseguir y definir las funciones del mismo.

Tal y como se ha visto se debe seguir el flujo de aplicación mostrado, para ello, es importante tener claras las funcionalidades de cada punto, viendo en que momento se ha conseguido los datos necesarios y pasar al siguiente. Es también necesario definir los datos requeridos en cada apartado, esto ayuda a definir las interfaces de las funciones utilizadas dentro de la función principal.

6.3.1 Función principal

La función principal es la más importante de todas ya que va a definir la ejecución del programa, en esta función se utilizarán las distintas *API's* que se han generado para el control de los distintos módulos del sistema. Cada una de estas contendrá las funciones explicadas más adelante, el código, como se puede observar a continuación, está correctamente comentado.

Lo primero y más importante es definir bien las *librerías* que se van a utilizar, estas almacenan las distintas definiciones de las funciones que se requieren para evitar errores permitiendo el uso de todos los elementos y funcionalidades requeridas.

```
// * Linux Libraries
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#include <stdint.h>   /* Standard types */
#include <string.h>   /* String function definitions */
#include <unistd.h>   /* UNIX standard function definitions */
#include <fcntl.h>    /* File control definitions */
#include <errno.h>    /* Error number definitions */
#include <termios.h>  /* POSIX terminal control definitions */
#include <sys/ioctl.h>
#include <getopt.h>
#include <time.h>

#include <math.h>

#include "ioctl_cmds.h"
```

Al tratarse de una aplicación de procesado de datos es necesaria la definición de estructuras para un correcto análisis y procesado de los datos que se utilizarán en las siguientes funciones. En esta aplicación, se definen como globales tres estructuras de datos, la primera es la que almacena los datos devueltos por el módulo hardware de procesado de datos. Este módulo correlador, devuelve cinco arrays con las muestras recibidas desde el micrófono correladas con los cinco códigos emitidos por cada uno de los emisores de la baliza de ultrasonidos.

La segunda estructura corresponde a la estructura de datos devuelta por la función de detección de picos, en la que por cada una de las señales correladas, por último, se muestra la estructura necesaria para el almacenaje de los resultados devueltos por la función de cálculo del posicionamiento.

```
// * Device driver memory storage structure
struct LowLevelq_CorrData
{
    unsigned int cnt;
    int data1[15000];
    int data2[15000];
    int data3[15000];
    int data4[15000];
    int data5[15000];
};
typedef struct LowLevelq_CorrData LowLevelq_CorrData_t;

// * Peak detector result positions
struct LowLevelq_Result_pos {
  int pos1[2];
  int pos2[2];
  int pos3[2];
  int pos4[2];
  int pos5[2];
}; typedef struct LowLevelq_Result_pos LowLevelq_Result_pos_t;

struct LowLevelq_Result_dist {
  int Xo;
  int Yo;
  int Zo;
  int Ro;
}; typedef struct LowLevelq_Result_dist LowLevelq_Result_dist_t;
```

Antes de empezar a escribir la función principal, es importante indicar qué funciones se va a utilizar en la misma, sobre todo, si se trata de una función sencilla y no se pretende utilizar una librería personalizada para las mismas. En este caso se muestran las funciones que se utilizarán en los módulos explicados en los siguientes apartados, utilizadas como *API* de control de los dos módulos hardware y el software de cálculo de datos.

```
// * HLS Peripheral API
// API for LowLelveq peripheral
int LowLevelq_SendData (int fd, char* data, int number);
int LowLevelq_GetNoRcvdData (int fd);
int LowLevelq_FlushBuffers (int fd);
int LowLevelq_ReceiveData (int fd, LowLevelq_CorrData_t* pdata, int number);

// * HW Peripheral API
int serialport_init(const char* serialport, int baud);
int serialport_writebyte(int fd, uint8_t b);
int serialport_write(int fd, const char* str);
int serialport_read_until(int fd, char* buf, int until);

// * Algorithm function
int peakDetectionSync (LowLevelq_CorrData_t* dataIn, unsigned int numSamples,
    LowLevelq_Result_pos_t* dataOut);
int fightTimes ( LowLevelq_Result_pos_t* dataIn, unsigned int numSamples, int* diffDistRes)
    ;
```

La función principal sería el siguiente punto, en este caso se ha iniciado a partir de un *pseudo-código* mantenido como comentarios, cumpliendo el *flujograma* mostrado en la explicación inicial del apartado.

En el inicio de la función se reserva y se definen, las variables que se utilizarán a lo largo de la ejecución de la misma. En esta aplicación, como método de validación, se han ideado puntos de validación, en ellos se almacena en ficheros la señal procesada en cada momento permitiendo observar la evolución de la señal, validando que se ajusta al *script* de *Matlab*.

```
// * Main function
int main(int argc, char **argv)
{
    char udev_lowlevelq[]= "/dev/LowLevelq@43C00000";
    char rx_buffer[15000];
    LowLevelq_CorrData_t CorrData;
    LowLevelq_Result_pos_t PosRes;
    int fd;
    int fd_lowlevelq;
    ssize_t size;
    int i, j, j_prev=0;
    int count= 0;
    int data_tx=0, data_rx=0;
    clock_t t_ini, t_fin;
    double secs;
    int status;

    // Distnce difference
    int diffDist[4];

    // File pointer
    FILE *datosSinProcesar;
    FILE *datosProcesados1;
    FILE *datosProcesados2;
    FILE *datosProcesados3;
    FILE *datosProcesados4;
    FILE *datosProcesados5;

    // Auxiliar string
    char auxString[100];
    if (argc<2)
        printf("Wrong function format\n");

    printf ("App starting...\n");
```

Una vez se han definido las variables y elementos que se pretenda utilizar se ha de validar los argumentos de entrada a la función, en este caso serán necesarios mínimo dos argumentos, contando como primer argumento el nombre de la función. El segundo será la localización del device driver que identifica el puerto serie del micrófono o receptor de la señal a procesar. Esta funcionalidad se ha añadido a la porción de código mostrada anteriormente.

Como se ha comentado en la explicación y en el *flujograma* lo primero que se debe realizar es el muestreo de los datos, este se realiza a partir del puerto serie, por lo que será necesario inicializarlo. Indicando si no se puede acceder al dispositivo por haber introducido uno incorrecto, en esta función se configura entre otras cosas la velocidad de transmisión, en este caso se configuran *9600* Baudios pero se ha probado con *115200* Baudios obteniendo mejores resultados.

```
// ** Configure HW periheral communication
```

```
fd= serialport_init (argv[1], 9600);

if (fd<1){
   printf ("\nError opening the device \%s...\n", argv[1]);
   return 0;
}
```

Una vez se ha abierto el *puerto serie* y validado la comunicación con el primero de los módulos hardware que tiene el sistema se valida la comunicación con el segundo, abriéndolo en modo lectura.

```
// ** Open HLS pheripheral
fd_lowlevelq= open (udev_lowlevelq, O_RDWR);

if (fd_lowlevelq < 1) {
   printf ("Invalid LowLevelq file.\n");
   return -1;
}
```

Al validar el hardware se inicia el muestreo de los datos del sensor. Este sensor se trata a nivel software como si fuese un fichero más, por lo que será necesario abrirlo y leer los datos que almacena previamente *muestreados*, al realizar un muestreo continuo no es necesario enviar ningún tipo de señal de sincronismo o inicio de muestro, es aconsejable, medir los tiempos que tarda en realizar una lectura, para en el caso de mantenerlo en muestreo continuo, poder saber cuando lanzar las lecturas.

```
// ** Open HLS pheripheral
fd_lowlevelq= open (udev_lowlevelq, O_RDWR);

if (fd_lowlevelq < 1) {
   printf ("Invalid LowLevelq file.\n");
   return -1;
}

// ** Read data from HW peripheral
t_ini = clock();

size= serialport_read_until (fd, rx_buffer, 10000);

t_fin = clock();

secs = (double)(t_fin - t_ini) / CLOCKS_PER_SEC;

printf("\nUSB MEMS Data read size: \%d\n", size);

printf("USB MEMS acquiring: \%.16g milisegundos\n", secs * 1000.0);
```

Para la validación de los datos recibidos por el sensor se genera un fichero, llamado **DatosSinProcesat.txt** en el que se almacenan el array de *10000* muestras leídas desde el sensor, este método ha servido para detectar malas configuraciones sobre la baliza transmisora.

```
// ** Write data received in a file
// Open file
datosSinProcesar = fopen("DatosSinProcesar.txt","wt");
// Write data
for ( i=0; i<size; i++){
```

```
  sprintf(auxString, "\%x \n", rx_buffer[i]);
  fputs(auxString, datosSinProcesar);
}
// Assert end of write
printf("Se almacenan DatosSinProcesar.txt\n\n");
// Close file
fclose(datosSinProcesar);
```

El siguiente paso es la transmisión de los datos recibidos desde el micrófono al periférico hardware de procesado de señal, para asegurar que el procesado se realiza directamente de los datos que se acaban de recibir, se limpian los *buffers* internos del módulo, una vez estén limpios, se envían los datos al sensor con las funciones de transmisión generadas.

```
// ** Clean HLS pheripheral data
// Flushing memory buffers in the device drivers
LowLevelq_FlushBuffers (fd_lowlevelq);

// ** Send data read from HW pheripheral to HLS pheripheral
t_ini = clock();
data_tx= LowLevelq_SendData (fd_lowlevelq, (char*) rx_buffer, size);
t_fin = clock();

secs = (double)(t_fin - t_ini) / CLOCKS_PER_SEC;
printf("Data sending: \%.16g milisegundos\n", secs * 1000.0);
printf ("Datos transmitidos: \%d\n", data_tx);
```

La detección de la finalización del procesado se puede realizar de dos métodos distintos, por *interrupción* o por *pooling*. El método de interrupción sería generar una interrupción a nivel software de usuario para detectar que ha terminado y cargar las muestras. El método *polling* muestrea una variable interna del módulo de procesado, activa únicamente en la finalización del procesado y borrada en la inicialización de los datos del módulo. La forma más adecuada de afrontar esta decisión sería empezar utilizando el *pooling* y una vez se valide pasar a *interrupciones* para evitar esperas activas. Al necesitar los datos del sensor y no ejecutar ningún proceso en paralelo se ha mantenido este método, siendo conscientes que para una futura ejecución en paralelo a otras aplicaciones como puede ser un controlador de un *dron*, es necesario realizar la detección de la finalización del procesado por interrupción.

```
// ** Detect processing HLS pheripheral status
t_ini = clock();
j= 0;
count= 0;

do {
  j_prev= j;

  j= LowLevelq_GetNoRcvdData (fd_lowlevelq);
  count++;

  if (j!=j_prev)
    count= 0;

} while (j < data_tx && count<1e5);

t_fin = clock();
```

```
secs = (double)(t_fin - t_ini) / CLOCKS_PER_SEC;
printf("Pooling process: \%.16g milisegundos\n", secs * 1000.0);
printf ("Datos Rx disponibles: \%d\n", j);
```

Para finalizar la parte de la aplicación en la que se utilizan los periféricos externos, se debe realizar una petición de las muestras procesadas por este módulo, para la recepción se utiliza la estructura diseñada en el inicio del *script*, para detectar que se ha leído se imprimen por pantalla los mensajes de aviso habituales.

```
// ** Recibe process HLS data
t_ini = clock();

data_rx= LowLevelq_ReceiveData (fd_lowlevelq, (LowLevelq_CorrData_t*) &CorrData, j);

t_fin = clock();

secs = (double)(t_fin - t_ini) / CLOCKS_PER_SEC;
printf("Data reading: \%.16g milisegundos\n", secs * 1000.0);

// ** Print data read
printf ("Datos Rx leídos: \%d\n", data_rx);
```

Para la validación del procesado de señal se ha añadido el siguiente código, en el se puede observar que para cada uno de los cinco arrays de datos que se reciben del módulo procesador. Se realiza una extensión de signo, necesaria ya que los datos que devuelve aparecen en un formato de *17b* que sobre los *32b* de un **int** requieren una apropiada extensión de signo, ésto se realiza observando el valor del bit más alto, el cual, si es *'1'* ha de extenderse hasta los *32b*. Una vez se adapta el dato al sistema actual se almacena en un fichero que posteriormente se ha validado en *Matlab*.

```
// ** Write data received in a file
datosProcesados1 = fopen("DatosProcesados1.txt","wt");
for ( i=0; i<data_rx; i++){
  // Sign extension (17 to 32b)
  if (((CorrData.data1[i]>>16) & 0x00000001) == 1){
    CorrData.data1[i] = CorrData.data1[i] | 0xFFFF0000;
  }
  sprintf(auxString, "\%x \n", CorrData.data1[i]);
  fputs(auxString, datosProcesados1);
}
printf("Se almacenan DatosProcesados1.txt\n\n");
fclose(datosProcesados1);

// ** Write data received in a file
datosProcesados2 = fopen("DatosProcesados2.txt","wt");
for ( i=0; i<data_rx; i++){
  // Sign extension (17 to 32b)
  if (((CorrData.data2[i]>>16) & 0x00000001) == 1){
    CorrData.data2[i] = CorrData.data2[i] | 0xFFFF0000;
  }
  sprintf(auxString, "\%x \n", CorrData.data2[i]);
  fputs(auxString, datosProcesados2);
}
printf("Se almacenan DatosProcesados2.txt\n\n");
fclose(datosProcesados2);
```

```
// ** Write data received in a file
datosProcesados3 = fopen("DatosProcesados3.txt","wt");
for ( i=0; i<data_rx; i++){
  // Sign extension (17 to 32b)
  if (((CorrData.data3[i]>>16) & 0x00000001) == 1){
    CorrData.data3[i] = CorrData.data3[i] | 0xFFFF0000;
  }
  sprintf(auxString, "\%x \n", CorrData.data3[i]);
    fputs(auxString, datosProcesados3);
}
printf("Se almacenan DatosProcesados3.txt\n\n");
fclose(datosProcesados3);

// ** Write data received in a file
datosProcesados4 = fopen("DatosProcesados4.txt","wt");
// Sign extension (17 to 32b)
for ( i=0; i<data_rx; i++){
  if (((CorrData.data4[i]>>16) & 0x00000001) == 1){
    CorrData.data4[i] = CorrData.data4[i] | 0xFFFF0000;
  }
  sprintf(auxString, "\%x \n", CorrData.data4[i]);
    fputs(auxString, datosProcesados4);
}
printf("Se almacenan DatosProcesados4.txt\n\n");
fclose(datosProcesados4);

// ** Write data received in a file
datosProcesados5 = fopen("DatosProcesados5.txt","wt");
for ( i=0; i<data_rx; i++){
  // Sign extension (17 to 32b)
  if (((CorrData.data5[i]>>16) & 0x00000001) == 1){
    CorrData.data5[i] = CorrData.data5[i] | 0xFFFF0000;
  }
  sprintf(auxString, "\%x \n", CorrData.data5[i]);
  fputs(auxString, datosProcesados5);
}
printf("Se almacenan DatosProcesados5.txt\n\n");
fclose(datosProcesados5);
```

En este punto se inicia el procesado software del posicionamiento, es decir, la algoritmia explicada anteriormente en el apartado de fundamentos matemáticos. Para ello se han generado funciones específicas, en este punto se ejecutan las funciones de detección de picos en las señales procesadas, cálculo de tiempos de vuelo y cálculo de la posición final por el método de *trilateración hiperbólica*. El código interno de estas funciones se explicará en la siguiente sección, por lo que aquí solo se muestra el código referente a la finalización de la función principal.

```
// Detect peaks
status = peakDetectionSync ((LowLevelq_CorrData_t*) &CorrData, data_rx, (
    LowLevelq_Result_pos_t*) &PosRes);

if (status == -1){
  printf("Error en detección de picos.\n");
  return -1;
}

// Detect distances and angules
```

```
status = fightTimes ( (LowLevelq_Result_pos_t*) &PosRes, data_rx, (int*) diffDist);

if (status == -1){
  printf("Error en detección de distancias.\n");
  return -1;
}

// Calculate flight times and distances
status = dronPosition((int*) diffDist, (LowLevelq_Result_dist_t*) finalPosition);

close (fd_lowlevelq);

return 0;
```

6.3.2 Detección de máximos

La función de *detección de máximos* será la función que procese las señales devueltas por el módulo correlador, al tratarse de señales resultado de correlación, han de tener una forma aproximada a una piramidal, obteniendo el máximo con la correlación absoluta entre la señal recibida y el código transmitido.

Como objetivo principal se ha de detectar el valor máximo de la señal de correlación y el puesto en el que se encuentra dentro del array, ya que sabiendo el tiempo de muestreo o tiempo entre muestras calcular los tiempos de vuelo, pero eso se explicará en la siguiente sección.

En la definición de la función se indica el tipo de dato de entrada a la misma, indicando también el número de muestras sobre las que calcular el dato y la estructura sobre la que se debe almacenar los resultados obtenidos. Para ello es importante la explicación de estos parámetros, comentado en el código mostrado a continuación, definiendo las variables internas que se utilizarán y las características requeridas en la función. Estas características son específicas de cada función, en este caso se trata de umbrales y valores mínimos entre picos, datos obtenidos de un análisis previo de las señales realizado con *Matlab* en la fase de diseño del sistema.

```
// ** Función de detección de picos
int peakDetectionSync (LowLevelq_CorrData_t* dataIn, unsigned int numSamples,
    LowLevelq_Result_pos_t* dataOut) {

  // *** Inputs
  // Correlated signal
  // dataIn     -> signal correlated with 5 patterns
  // numSamples -> Number of samlples for each correlated signal
  // dataOut    -> Five peaks segnal position and amplitude (Pos, Ampl)


  // *** Internal variables
  LowLevelq_Result_pos_t primaryPeak;
  LowLevelq_Result_pos_t secondaryPeak;
  int i=0;


  // **** Threesholds
  // Maximun amplitude second peak
  int maxAmpl2Peak = 50;// \ %
  maxAmpl2Peak = maxAmpl2Peak/100;
```

```
  // Minimun peak separation
  int minPeakDist = numSamples * (5/100);//\ %

  // Ever should be even number, same samples after and before.
  if (minPeakDist\ %2 != 0){
    minPeakDist ++;
  }

  // *** Outputs
  // Maximun value position if meets with the thresholdsn
  // dataOut -> posX(0) -> Position in array in X number of correlation
  // dataOut -> posX(1) -> Amplitude sampled in X number of correlation
  // Data returned, 0 -> Ok, -1 -> Error have not samples.
```

Una vez se han definido los distintos valores y condiciones de la función se crean los casos de error e inicialización de variables para evitar que se produzcan casos extraños. En el código mostrado a continuación, se muestra el caso en el que los datos que llegan a la función llegan sin muestras o corruptos. Es importante pensar en todos los casos de los datos que entran para evitar situaciones de inestabilidad.

```
// *** Algorithm

// **** Not imput samples
if (numSamples == 0){

  // Assert message
  printf("No se ha recibido ningun valor.\n");

  // Return all values as 0
  // Position values
  dataOut->pos1[0] = 0;
  dataOut->pos2[0] = 0;
  dataOut->pos3[0] = 0;
  dataOut->pos4[0] = 0;
  dataOut->pos5[0] = 0;

  // Amplitude Values
  dataOut->pos1[1] = 0;
  dataOut->pos2[1] = 0;
  dataOut->pos3[1] = 0;
  dataOut->pos4[1] = 0;
  dataOut->pos5[1] = 0;

  // Return error
  return -1;
}
```

Una vez se han ideado y controlado los casos erróneos se pasa a la función de detección de los valores de pico, en este algoritmo se recorre, para cada uno de las distintas señales correladas almacenando el máximo y la posición en la que se encuentra, realizando una segunda vuelta para detectar el segundo punto más alto de la señal, para ver que cumple los requisitos de amplitud de correlación.

```
// ***** First pattern signal
if ( dataIn->data1[i] >= dataIn->data1[i+1] ) {
  // If is bigger than last stored
```

```
  if (dataIn->data1[i] > primaryPeak.pos1[1]) {
    // Fist data is bigger than second, store first
    primaryPeak.pos1[0] = i;
    primaryPeak.pos1[1] = dataIn->data1[i];
  }
} else {
  // If is bigger than last stored
  if (dataIn->data1[i+1] > primaryPeak.pos1[1]) {
    // Second data is bigger than first, store second
    primaryPeak.pos1[0] = i+1;
    primaryPeak.pos1[1] = dataIn->data1[i+1];
  }
}

// ...

// Second Peak

// ...

// ***** First pattern signal
// if meets the minimun distance
if ( (i < primaryPeak.pos1[0] - minPeakDist/2) || (i > primaryPeak.pos1[0] + minPeakDist/2)
     ) {
  if ( dataIn->data1[i] >= dataIn->data1[i+1] ) {
    // If is bigger than last stored
    if (dataIn->data1[i] > secondaryPeak.pos1[1]) {
      // Fist data is bigger than second, store first
      secondaryPeak.pos1[0] = i;
      secondaryPeak.pos1[1] = dataIn->data1[i];
    }
  } else {
    // If is bigger than last stored
    if (dataIn->data1[i+1] > secondaryPeak.pos1[1]) {
      // Second data is bigger than first, store second
      secondaryPeak.pos1[0] = i+1;
      secondaryPeak.pos1[1] = dataIn->data1[i+1];
    }
  }
}
```

Por último se han de devolver los datos procesados una vez se ha detectado que cumplen los *requisitos* impuestos por las condiciones definidas en el inicio de la función, el código mostrado a continuación, muestra las condiciones de amplitudes máximas devueltas almacenándolas en el parámetro de salida recibido en la cabecera de la función.

```
// *** Data returned
// Detect if meets amplitude thresholds

// **** Evaluate first threshold value

if (primaryPeak.pos1[1] * maxAmpl2Peak > secondaryPeak.pos1[1]){
  printf("First signal don't meet amplitude threhold.\n");
  primaryPeak.pos1[0] = 0;
  primaryPeak.pos1[1] = 0;
}
```

```
// **** Evaluate second threshold value

if (primaryPeak.pos2[1] * maxAmpl2Peak > secondaryPeak.pos2[1]){
  printf("Second signal don't meet amplitude threhold.\n");
  primaryPeak.pos2[0] = 0;
  primaryPeak.pos2[1] = 0;
}

// **** Evaluate third threshold value

if (primaryPeak.pos3[1] * maxAmpl2Peak > secondaryPeak.pos3[1]){
  printf("Third signal don't meet amplitude threhold.\n");
  primaryPeak.pos3[0] = 0;
  primaryPeak.pos3[1] = 0;
}

// **** Evaluate fourth threshold value

if (primaryPeak.pos4[1] * maxAmpl2Peak > secondaryPeak.pos4[1]){
  printf("Fourth signal don't meet amplitude threhold.\n");
  primaryPeak.pos4[0] = 0;
  primaryPeak.pos4[1] = 0;
}

// **** Evaluate fifth threshold value

if (primaryPeak.pos5[1] * maxAmpl2Peak > secondaryPeak.pos5[1]){
  printf("Fifth signal don't meet amplitude threhold.\n");
  primaryPeak.pos5[0] = 0;
  primaryPeak.pos5[1] = 0;
}

// **** Load values

// Assert message
printf("No se ha recibido ningun valor.\n");

// ***** First position
dataOut->pos1[0] = primaryPeak.pos1[0];
dataOut->pos1[1] = primaryPeak.pos1[1];

// ***** Second position
dataOut->pos2[0] = primaryPeak.pos2[0];
dataOut->pos2[1] = primaryPeak.pos2[1];

// ***** Third position
dataOut->pos3[0] = primaryPeak.pos3[0];
dataOut->pos3[1] = primaryPeak.pos3[1];

// ***** Fourth position
dataOut->pos4[0] = primaryPeak.pos4[0];
dataOut->pos4[1] = primaryPeak.pos4[1];

// ***** Fifth position
dataOut->pos5[0] = primaryPeak.pos5[0];
dataOut->pos5[1] = primaryPeak.pos5[1];

// **** Return correct value
return 0;
```

```
}
```

De este modo se obtiene en la función principal los datos de posiciones y amplitudes de los máximos dentro de las señales procesadas, necesarias para el cálculo de los tiempos de vuelo.

6.3.3 Tiempos de vuelo

La función de cálculo de *tiempos de vuelo* es el objetivo final del proyecto ya que la función de cálculos de distancia se ha decidido meter después. En esta función se han realizado los cálculos conociendo la *frecuencia de muestreo* de *10 kHz*, esto hace un *tiempo de muestreo* de *100 us*, que será el tiempo de separación entre muestras.

El cálculo de distancias se ha realizado desde cada una de las cuatro esquinas de la baliza respecto a la del centro, la resta entre el valor de la muestra central respecto a las de los vértices indica la diferencia en tiempo de llegada. Al saber la variación de tiempos entre las señales y la velocidad a la que viaja el sonido, suponiendo unas condiciones estándar de temperatura y humedad, somos capaces de calcular la diferencia de distancias entre los puntos.

```
// ** Calculate flight time and distances
int fightTimes ( LowLevelq_Result_pos_t* dataIn, unsigned int numSamples, int* diffDistRes)
     {

  // *** Inputs
  // dataIn->posX[0] -> sample number
  // dataIn->posX[1] -> sample value
  // numSamples      -> max number of samples

  // *** Internal variables
  // Frecuencia de muestreo 10 kHz

  // *** Outputs
  // result->distX -> X point distance - center distance

  // *** Algorithm

  // **** Calculate differences
  //diffDist1 = dataIn->pos1[0] - dataIn->pos1[0] ;
  *(diffDistRes+0)= dataIn->pos2[0] - dataIn->pos1[0] ;
  *(diffDistRes+1)= dataIn->pos3[0] - dataIn->pos1[0] ;
  *(diffDistRes+2)= dataIn->pos4[0] - dataIn->pos1[0] ;
  *(diffDistRes+3)= dataIn->pos5[0] - dataIn->pos1[0] ;

  // **** Calculate distances
  // dist1 =  dist1 * 100E-6 * 342.3;
  *(diffDistRes+0) =  *(diffDistRes+0) * 100E-6 * 342.3;
  *(diffDistRes+1) =  *(diffDistRes+1) * 100E-6 * 342.3;
  *(diffDistRes+2) =  *(diffDistRes+2) * 100E-6 * 342.3;
  *(diffDistRes+3) =  *(diffDistRes+3) * 100E-6 * 342.3;

  // *** Return correct value
  return 0;

}
```

La función devuelve la variación de distancia en metros que será lo necesario para poder ejecutar el algoritmo de *posicionamiento hiperbólico*, explicada en el siguiente apartado.

6.3.4 Posición actual

El *algoritmo de posicionamiento* basado en la *trilateración hiperbólica* ha sido explicado anteriormente en este tema, para poder ejecutarlo como función en **C/C++** es necesario partir de las ecuaciones matriciales finales, ya que únicamente se busca obtener la posición, conociendo todos los elementos físicos requeridos en esta función. Al igual que en las funciones anteriores es importante definir bien la cabecera de la función, utilizando de las funciones anteriores las diferencias conseguidas por la función de cálculo de los tiempos de vuelo y definiendo la localización de almacenamiento de los valores de las variables.

Se inicia midiendo los parámetros físicos fijos como son el posicionamiento de cada uno de los emisores de ultrasonidos, para ésto se define un sistema de coordenadas basado en los números indicados en la baliza colocada a *3m* sobre el suelo. Siendo muy importante la definición de las variables que se van a utilizar en el resto del sistema.

```
// ** Hiperbolic trilateration LSE wih extra beacon
int dronPosition(int* diffDist, LowLevelq_Result_dist_t* finalPosition) {
  /// *** Inputs
  // distVar = difference in distance between n tx and 1 tx

  /// *** Internal variables
  // baliza's position in (x, y, z)
  int balizas_x[5] = {2.122, 2.513, 2.515, 1.760, 1.758};
  int balizas_y[5] = {1.224, 1.560, 0.900, 0.890, 1.560};
  int balizas_z[5] = {3.450, 3.460, 3.450, 3.460, 3.450};

  // Matrix index 4 -> 0:3
  int C_NUM_DIFF = 4;
  int C_ROW_NUM = C_NUM_DIFF-1;
  int C_COL_NUM = C_NUM_DIFF-1;
  int C_DIA_NUM;

  // Index to work with bidimensional matrix
  int i = 0, j = 0, k = 0;

  // Store auxiliar values
  int multiplierValue = 0;

  // Detect the number of values in the main diagonal
  if (C_COL_NUM <= C_ROW_NUM) { C_DIA_NUM = C_COL_NUM; }
  else { C_DIA_NUM = C_ROW_NUM; }
```

Una vez se han definido los elementos necesarios externos al algoritmo, se definen las distintas matrices de elementos utilizadas en el algoritmo y las requeridas en su avance, en algoritmos como este es importante definir bien los nombres de cada una de las variables para que se sepa la operación almacenada en cada punto. Respecto al avance de la matriz **A** se han generado las siguientes matrices.

```
// Matrix A [4x4], from differences in baliza's distances
//
//      / 2*(x2-x1)  2*(y2-y1)  2*(z2-z1)  -2*dist21 \
//     |  2*(x3-x1)  2*(y3-y1)  2*(z3-z1)  -2*dist31  |
```

```
// A = |  2*(x4-x1)  2*(y4-y1)  2*(z4-z1)  -2*dist41  |
//     \ 2*(x5-x1)  2*(y5-y1)  2*(z5-z1)  -2*dist51 /
//
int A[C_ROW_NUM][C_COL_NUM];
int At[C_ROW_NUM][C_COL_NUM];
int AtA[C_ROW_NUM][C_COL_NUM];
int AtAinv[C_ROW_NUM][C_COL_NUM];
int AtAinvAt[C_ROW_NUM][C_COL_NUM];
int I[C_DIA_NUM][C_DIA_NUM];

// Identity matrix is only 1's in main diagonal
for (i = 0; i<= C_DIA_NUM; i++) {
    for (j = 0; j<= C_DIA_NUM; j++) {
        if ( i == j ){
            I[i][j] = 1;
        } else {
          I[i][j] = 0;
        }
    }
}
```

El código mostrado anteriormente muestra no solo la definición de las matrices **A** y sus combinaciones sino la matriz identidad I que se utilizará para hallar la inversa de la matriz **A** en el despeje del posicionamiento. En el siguiente código, se muestran las matrices **B** y **X** que serían las utilizadas para el almacenamiento de la variación de distancias y posiciones y del las posiciones finales más la distancia total.

```
// Matrix B [4x1],
//
//      / -dist21^2 - x1^2 + y1^2 + Z1^2 + x2^2 + y2^2 + z2^2 \
//     |  -dist31^2 - x1^2 + y1^2 + Z1^2 + x3^2 + y3^2 + z3^2  |
// B = |  -dist41^2 - x1^2 + y1^2 + Z1^2 + x4^2 + y4^2 + z4^2  |
//      \ -dist51^2 - x1^2 + y1^2 + Z1^2 + x5^2 + y5^2 + z5^2 /
//

int B[C_ROW_NUM];

// Matrix X [4x1], position results
//
//      / Xo \   -> Posoción de coordenada X
//     |  Yo  |  -> Posoción de coordenada Y
// X = |  Zo  |  -> Posoción de coordenada Z
//      \ Ro /   -> Radio a la posición de referencia
//

int X[C_ROW_NUM];

/// *** Outputs
// finalPosition = Dron position.
```

Para la inicializaciones de las matrices se ha utilizado la siguiente porción de código, en ella se observa la carga de los valores de **A**, obtenidos a partir de la posición de los transmisores y la matriz **B**, que se hace a partir de la diferencia de posiciones más la variación de distancias percibidas en el receptor.

```
// *** Algorithm
```

```
// Matrix algorithm:  A*X = B -> X = (A^t * A)^-1 * A^t * B;

// **** Get A matrix
for ( j = 0; j <= C_ROW_NUM; j++ ) {
  A[j][0] = 2 * (balizas_x[j+1] - balizas_x[0]);
  A[j][1] = 2 * (balizas_y[j+1] - balizas_y[0]);
  A[j][2] = 2 * (balizas_z[j+1] - balizas_z[0]);
  A[j][3] = -2 * diffDist[j];
}

// **** Get B matrix
for ( j = 0; j <= C_ROW_NUM; j++) {
  // Problems with pow
  //B[j] = - diffDist[j] - pow(balizas_x[0],2) +
  //       pow(balizas_y[0],2) + pow(balizas_z[0],2) +
  //       pow(balizas_x[j+1],2) + pow(balizas_y[j+1],2) +
  //       pow(balizas_z[j+1],2);
  B[j] = - diffDist[j] - (balizas_x[0]*balizas_x[0]) +
           (balizas_y[0]*balizas_y[0]) +
           (balizas_z[0]*balizas_z[0]) +
           (balizas_x[j+1]*balizas_x[j+1]) +
           (balizas_y[j+1]*balizas_y[j+1]) +
           (balizas_z[j+1]*balizas_z[j+1]);
}
```

Una vez se han definido las matrices es muy importante tener claro el algoritmo que se pretende ejecutar y los procesos que serán utilizados en cada momento, ya que para una misma operación, como pueda ser la de la *matriz inversa*, no se realiza con la misma complejidad manteniendo la *escalabilidad* según el método que se utilice. Para la resolución de la *matriz inversa*, se ha decidido utilizar el *algoritmo* de **Gauss-Jordan**.

Lo primero que se realiza en el algoritmo es el cálculo de la matriz *transpuesta*, el cual es sencillo respecto a la transpuesta, en el se intercambian filas por columnas, se inicia por este punto ya que esta matriz es necesaria para obtener las matrices con las que se conseguirá la posición final.

En este algoritmo se parte de las operaciones realizadas a matrices objetivo y matriz identidad, **I**, intentando realizar una matriz identidad en la posición de la matriz objetivo y obteniendo el resultado en la posición en la que estaba en un inicio la matriz identidad. Para conseguirlo se realiza el cálculo de la matriz diagonal inferior y luego la diagonal superior normalizando los elementos de la diagonal.

```
  // **** Get A^t Matrix
  // i = Column index
  // j = Row index
  for ( i = 0; i <= C_COL_NUM; i++) {
    for ( j = 0; j <= C_ROW_NUM; j++) {
      At[i][j] = A[j][i];
    }
  }

  // **** Get (A^t * A) Matrix
  // i = Row index
  // j = Column index
  // k = Row-Column inter operators
  for (i = 0; i <= C_ROW_NUM; i++) {
    for (j = 0; j <= C_COL_NUM; j++) {
```

```
      for (k = 0; k <= C_COL_NUM; k++) {
        AtA[i][j] += At[i][k] * A[k][j];
      }
    }
  }

  // **** Get (A^t * A)^-1 Matrix
  // (A^t * A) matrix to transform in I

  // Make zeros under main diagonal and normalize
  for (k=0; k<=C_DIA_NUM; k++){
    // Detect digaonal value and normalize.
    for (i=0; i<=C_COL_NUM; i++){
      if (AtA[i][i] != 0){
        AtA[k][i]= AtA[k][i] / AtA[i][i];
        I[k][i]= I[k][i] / AtA[i][i];
      } else {
        printf("Error, 0 en valor de la diagonal, saliendo de la función de posicionamiento
            .\n");
        return -1;
      }
    }

    // Make zeros under diagonal, if is not last diagonal number
    if(k < C_DIA_NUM) {
      for (i = k+1 ; i <= C_ROW_NUM; i++){
        if (AtA[i][k] == 0){
          // It's ok
        } else {
          // Normalize this row
          for (j=0; j<C_COL_NUM; j++){
            AtA[i][j] = AtA[i][j] / AtA[k][i];
            I[k][j] = I[k][j] / AtA[k][i];
          }
          // substract this row with k diagonal row
          for (j=0; j<C_COL_NUM; j++){
            AtA[i][j] = AtA[i][j] - AtA[k][j];
            I[i][j] = I[i][j] - AtA[k][j];
          }
        }
      }
    }
  }
  // Make zeros over diagonal
  for (k = C_DIA_NUM; k >= 0; k --) {
    // Make zeros over diagonal
    if (k > 0){
      for (i = k-1; i >= 0; i--){
        multiplierValue = AtA[i][j];
        if (multiplierValue == 0 ){
          // It's ok
        } else {
          for ( j = 0; j<=C_COL_NUM; j++){
            AtA[i][j] = AtA[i][j] - (AtA[k][j] * multiplierValue);
            I[i][j] = I[i][j] - (I[k][j] * multiplierValue);
          }
        }
      }
    }
```

```
  }

  // (AtA)^-1 = I = AtAinv
  //&AtAinv = &I;

  // **** Get (A^t * A)^-1 * A^t Matrix
  // i = Row index
  // j = Column index
  // k = Row-Column inter operators
  for (i = 0; i <= C_ROW_NUM; i++) {
    for (j = 0; j <= C_COL_NUM; j++) {
      for (k = 0; k <= C_COL_NUM; k++) {
          AtAinvAt[i][j] += I[i][k] * At[k][j];
          //AtAinvAt[i][j] += AtAinv[i][k] * At[k][j];
      }
    }
  }

  // **** Get X Matrix
  for ( i = 0; i <= C_ROW_NUM; i++) {
    for ( j = 0; j <= C_COL_NUM; j++) {
      X[i] += AtAinvAt[i][j] * B[j];
    }
  }

  // *** Return final value
  finalPosition->Xo = X[0];
  finalPosition->Yo = X[1];
  finalPosition->Zo = X[2];
  finalPosition->Ro = X[3];

  return 0;
}
```

Como se puede observar, la complejidad del algoritmo aparece en la generación de la matriz **AtAinvAt** por la que se multiplica a **B** para hallar la posición y distancia final almacenada en la estructura recibida en la cabecera de la función.

6.4 Resultados

La aplicación se ha ideado para que sea capaz de comunicarse con el periférico hardware para recibir datos, que luego transfiere al periférico de procesado de señal. En espe módulo se realiza la correlación con cada uno de los códigos ortogonales que rtansmite cada uno de los cinco emisores. Al recibir estas matercices de correlación el software ha de calcular el tiempo de vuelo de cada una de las señales representandolas por pantalla.

Para ello se han realizado distintas medidas, siendo procesados los resultado y almacenados para la validación del algoritmo. Se ha conseguido la validación del cálculo los resultados de los tiempos de vuelo de forma satisfactoria, validando la arquitectura seleccionada consiguiendo así el objetivo final del proyecto.

Como objetivo extra se ha buscado validar el algoritmo de posicionamiento a partir de la trilateración hiperbólica con las señales procesadas, se ha generado el algoritmo necesario para realiar el cálculo pero se han detectado problemas en los resultados obtenidos.

Después de depurar la aplicación se ha detectado que el algoritmo de cálculo de la matriz inversa tiene problemas con la precisión artimetica utilizada, ya que al tratarse de cifras tan pequeñas como las diferencias de los tiempos de vuelo, en la operación de normalización, trunca los valores a cero. Esto produce errores en los resultados obtenidos por la indeterminación producida por la visión por cero.

Este es el motivo por el cual aparece en los trabajos futuros la optimización del algoritmo de posicionamiento hiperbolico, ya que no se han conseguido obtener los resultados esperados. También hay que tener en cuenta que según el paper en el que está basado este algoritmo, indica que tiene problemas si alguna de las cinco señales correladas no es correcta o si la diferencia entre los tiempos de vuelo de las mismas es muy parecido, tendiendo las diferencias a cero. Situación que aparece al estar enfrentado el receptor a la radiobaliza.

Capítulo 7

Conclusiones y líneas futuras

El principal resultado de este Trabajo Fin de Máster ha sido el diseño e implementación de una arquitectura hardware-software, donde se ha incluido en el procesador un sistema operativo empotrado. Para éste, se ha desarrollado un device driver específico capaz de controlar un periférico de usuario, generado para una aplicación específica. Para ello, se han alcanzado una serie de conclusiones parciales:

7.1 Conclusiones

La conclusión de este proyecto de trabajo de Fin de Máster, ha sido la validación de la arquitectura hardware propuesta para este sistema. Esta validación total del sistema se ha conseguido gracias a la validación parcial de los siguientes apartados.

- Integración de periférico hardware.

  Se ha demostrado que este sistema permite añadir periféricos hardware, en este caso de procesado de señal validando la versatilidad y potencia del mismo. Con una generación a partir de las herramientas de vivado, independientemente a la aplicación final.

- Generación y validación de los device drivers necesarios.

  Se ha demostrado la posibilidad de la generación de un firmware o software de bajo nivel que permite el control de los distintos periféricos del sistema. Su uso como device drivers y generación de los métodos más comunes de uso del mismo desde el espacio de usuario del sistema operativo.

- Generación y uso del sistema operativo Petalinux.

  Se ha comprendido, ejecutado y demostrado el funcionamiento del sistema operativo a partir de una distribución de *Petalinux*, en el cual se ha demostrado el funcionamiento de los módulos hardware incluidos en el sistema a partir de la ejecución de los métodos desde una aplicación de usuario.

- Generación de aplicación final de usuario.

  Se ha demostrado que sobre esta distribución se puede realizar la compilación de aplicaciones de distintos campos siendo ejecutadas en sistemas en tiempo real a partir de los métodos creados para el control de un device driver bajo el que se ejecutan procesos hardware específicos que permiten la optimización de este tipo de aplicaciones.

- Validación del sistema en conjunto.

  Se ha demostrado que el conjunto del sistema ideado y su arquitectura funciona después de haber validado cada una de las funciones en conjunto y por separado.

7.2 Líneas futuras

Las líneas futuras marcan el seguimiento futuro de la vida del sistema. Se han ideado para su mejora y optimización con carácter educativo, pudiendo inspirar a nuevos desarrolladores a continuar la labor hasta la finalización del trabajo. La lista mostrada a continuación indica ciertos aspectos de mejora que se han detectado durante la realización del proyecto, no habiendo sido implementados por no perder de vista el objetivo final del proyecto.

- Optimización de la comunicación entre el módulo hardware y el sensor.

  En el proyecto se ha utilizado un módulo receptor de señales de ultrasonido, conectado al sistema por USB. Como mejora se podría realizar un controlador hardware que a partir del USB transfiriese los datos directamente a memoria hardware. Se evita así la pérdida de tiempo de la lectura de los datos desde el sistema operativo para su transferencia al hardware para su procesado.

- Uso de distribuciones más genéricas como *Ubuntu 14.04*.

  El uso de una distribución como pueda ser *Ubuntu 14.04* permite al sistema el uso de herramientas y gestores de paquetes más comunes que los permitidos por la distribución de Petaliniux. Esta medida puede servir de ayuda a la hora de ampliar el espectro de los futuros desarrolladores de éste tipo de sistemas. También la utilización de herramientas ofrecidas por desarrolladores que permiten simplificar las tareas de desarrollo.

- Mejora de las comunicaciones de acceso remoto al sistema.

  Para una simplificación del sistema se podría añadir un sistema de comunicación basado en *WIFI*, el cual permita un acceso remoto al sistema pudiendo realizar un desarrollo a distancia y la posibilidad de tener un acceso a Internet para la instalación de librerías paquetes o herramientas.

- Añadir persistencia de los archivos del sistema operativo.

  La persistencia de los archivos transferidos al sistema operativo, como pueda ser la aplicación o aplicaciones que se quieran ejecutar y el hecho de poder mantener los resultados bajo una pérdida de alimentación, es un requisito básico en cualquier sistema. Es por este motivo por el que el poder mantener los archivos en memoria puede ser una gran ventaja en el desarrollo del sistema. También el almacenamiento de sistema operativo en memoria interna evitando la *SD*, requisito habitual en sistemas embarcados.

- Mejora de los device drivers a partir de semáforos.

  La idea final de esta aplicación era la de su ejecución en paralelo al controlador de vuelo del *dron*. Por lo que para el uso de múltiples aplicaciones concurrentes sobre el mismo sistema operativo es necesario poder realizar una gestión de los recursos por parte del sistema operativo como pueda ser el tiempo de ejecución. Es por ello por lo que los semáforos pueden ayudar a esta gestión.

- Optimización del algoritmo de cálculo de posicionamiento.

  La idea inicial de este proyecto fue la del uso de este sistema sobre un *dron*. Por ello es importante una depuración a conciencia del algoritmo de posicionamiento aumentando la precisión para evitar problemas con el sistema final.

- Mejora de la aplicación de usuario para poder ser ejecutada en *multi-hilo*.

  Los requisitos finales de tiempos de ejecución de la aplicación de usuario pueden requerir un aumento en los recursos utilizados por el mismo. Éste motivo puede requerir el uso de ambos procesadores de forma concurrente. Para ello es necesaria la programación de la aplicación de usuario en *multi-hilo*.

- Mejora del cálculo matricial a partir de la aceleración hardware.

  Como se ha explicado en este proyecto, la última parte de detección de la posición, a partir de la posición de la baliza se realiza a partir del cálculo matricial. Éste tipo de sistemas basados en *SoC*, comprenden la parte *PL*, existiendo un gran variedad de algoritmos de aceleración de cálculo de matrices. Estos podrían utilizarse para realizar el algoritmo a nivel hardware, consiguiendo un gran aumento en la ejecución de la localización liberando al sistema operativo de aplicaciones bloqueantes.

Apéndice A

Presupuesto del proyecto

Este tema aborda los aspectos económicos que se requieren para una correcta ejecución del proyecto. Este presupuesto se divide en tres apartados en función del material utilizado, en el que se incluyen herramientas y licencias utilizadas, la mano de obra del desarrollador del proyecto y el presupuesto total.

A.1 Presupuesto material

En el presupuesto del proyecto se incluye una lista de materiales requeridos para el correcto desarrollo y pruebas del sistema integrado, desde las herramientas utilizadas para el diseño y desarrollo hasta los materiales sobre los cuales se han realizado las pruebas. En este proyecto no se incluyen herramientas de uso común ni los gastos en recursos como pueda ser la luz o el alquiler del lugar de trabajo entre otros.

- *Ordenador personal Asus.*

  Ordenador portátil AsusGL752VW-T4064D con procesador lintel i7, disco duro SSD de 500GB y 16GB de memoria RAM por un total de **1400 €** [**?**].

- *Pantalla Asus 23".*

  Monitor externo al portátil para la correcta adaptación al entorno de trabajo por parte del usuario **124 €**.

- *Periféricos de acondicionamiento*

  Incluye el teclado mecánico y el ratón inalámbrico para adaptar el entorno de trabajo consiguiendo los requisitos de riesgos laborables necesarios. Precio total **40 €**.

- *Licencia de Windows 10.*

  El ordenador en el que se ha realizado el proyecto tiene como sistema operativo base Windows 10, sobre el cual se ha ejecutado una máquina virtual con CentOS 6.10 sobre la cual se ha trabajado. Si se tiene en cuenta la licencia de Windows ha costado **259 €**, si se tiene en cuenta el uso de Linux CentOS 6.10 la licencia ha sido gratuita.

- *Licencia de herramientas Vivado.*

  La licencia de las herramientas de vivado para la ayuda al diseño electrónico han tenido un precio de **2 644 €**, pudiendo comprobarse en la página referenciada.

- *Herramienta Petalinux.*

  Las herramientas de Petalinux utilizadas en la máquina virtual Linux para la generación del sistema operativo ha sido descargada de forma libre de la página web de Xilinx.

- *Herramienta de comunicación por puerto serie.*

  La herramienta para la comunicación por puerto serie entre el sistema generado y el ordenador personal en la fase de depuración de los device drivers y su inicialización en el sistema operativo ha sido la herramienta de software libre Teraterm

- *Editor de texto Emacs 26.02.*

  Para la generación y edición de las fuentes, desde los apuntes tomados en ORG, las fuentes modificadas en VHDL, las fuentes creadas para los device drivers y las de la aplicación en C/C++ han sido tratadas con este editor de licencia gratuita basado en software libre.

- *Sistema de control de versiones GIT.*

  La herramienta de control de versiones que se ha utilizado en este proyecto ha sido GIT en conjunto con GITHUB privado. Herramienta y plataforma gratuitas.

- *Tarjeta de evaluación.*

  Tarjeta de evaluación ZedBoard comercializada por Digilent **319 €**.

- *Periférico receptor.*

  Periférico receptor de señales de ultrasonido creado en el departamento de electrónica en la Universidad de Alcalá, no aplica.

- *Radiobalizas.*

  Radiobalizas emisora de ultrasonido creado en el departamento de electrónica en la Universidad de Alcalá, no aplica.

- *Herramienta Redmine para gestión de proyectos.*

  Herramienta de gestión de proyectos Redmine para el seguimiento de las distintas tareas del proyecto, gratuita, utilizada en máquina virtual.

- *Gastos comunes como memoria SD más cables de comunicación.*

  Gastos comunes de cables de comunicación como el ethernet cruzado y memoria de almacenamiento **23 €**.

Precio total utilizado en los materiales del proyecto: **4550 €**. IVA incluido.

A.2 Presupuesto de ingeniaría

Para este proyecto se ha requerido durante 8 meses la contratación de personal cualificado, Ingeniero en electrónica de comunicaciones a tiempo parcial, esto equivale a 4 meses a jornada completa. Si se tienen en cuenta los salarios mínimos de cada sector se llega a un presupuesto final de **30000 €** al año. Por lo que el precio a pagar por esa persona son **10000 €**.

Precio total utilizado en la mano de obra de ingeniería del proyecto.

A.3 Presupuesto total

El precio final será la suma de los gastos materiales más los gastos en conceptos de mano de obra del proyecto, ascendiendo a un total de **18550 €**.

Biliografía

- [1] Aerotenna, especificaciones técnicas de la plataforma seleccionada como controlador de vuelo. ”https://aerotenna.readme.io/docs/introduction” [Ultimo acceso 20/septiembre/2019].
- [2] Especificaciones del ordenador del proyecto. ” https://www.asus.com/es/ROG-Republic-Of-Gamers/ROG-GL752VW/” [Ultimo acceso 20/septiembre/2019].
- [3] Petalinux y herramientas Xilinx. ” https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html” [Ultimo acceso 20/septiembre/2019].
- [4] Herramienta de control de versiones. ” https://git-scm.com/” [Ultimo acceso 20/septiembre/2019].
- [5] Editor de texto. ” https://www.gnu.org/software/emacs/download.html” [Ultimo acceso 20/septiembre/2019].
- [6] Herramienta de gestión de proyectos. ” https://www.redmine.org/” [Ultimo acceso 20/septiembre/2019].
- [7] Tarjeta ZedBoard. ” https://www.xilinx.com/products/boards-and-kits/1-elhabt.html” [Ultimo acceso 20/septiembre/2019].
- [8] OcPoC. ” https://aerotenna.readme.io/docs/what-is-ocpoctm-mini-1” [Ultimo acceso 20/septiembre/2019].
- [9] PLD, dispositivo lógico programable. ” https://es.wikipedia.org/wiki/L%C3%B3gica_programada” [Ultimo acceso 20/septiembre/2019].
- [10] Circuito Integrado. ” https://es.wikipedia.org/wiki/Circuito_integrado” [Ultimo acceso 20/septiembre/2019].
- [11] ASIC, sistema de aplicación específica. ” https://es.wikipedia.org/wiki/Circuito_integrado_de_aplicaci%C3%B3n_espec%C3%ADfica” [Ultimo acceso 20/septiembre/2019].
- [12] FPGA. ” https://es.wikipedia.org/wiki/Field-programmable_gate_array” [Ultimo acceso 20/septiembre/2019].
- [13] Lógica programada. ” https://es.wikipedia.org/wiki/L%C3%B3gica_programada” [Ultimo acceso 20/septiembre/2019].
- [14] System on chip. ” https://es.wikipedia.org/wiki/System_on_a_chip” [Ultimo acceso 20/septiembre/2019].
- [15] Prototipado con FPGA. ” https://es.wikipedia.org/wiki/Prototipado_FPGA” [Ultimo acceso 20/septiembre/2019].

- [16] Localización en interiores. ” https://es.wikipedia.org/wiki/Sistema_de_posicionamiento_en_interiores” [Ultimo acceso 20/septiembre/2019].

- [17] Sistemas de Localización en interiores. ” http://oa.upm.es/947/1/PFC_LUIS_DIAZ_AMBRONA.pdf” [Ultimo acceso 20/septiembre/2019].

- [18] Linux Device drivers. Rubini. ”https://aerotenna.readme.io/docs/introduction” [Ultimo acceso 20/septiembre/2019].

- [19Distribuciones Linux. ” https://gutl.jovenclub.cu/diecinueve-distribuciones-basadas-en-red-hat-linux/” [Ultimo acceso 20/septiembre/2019].

- [20] Barreras software. ” https://www.kernel.org/doc/Documentation/memory-barriers.txt” [Ultimo acceso 20/septiembre/2019].

- [21] Barreras Software. ” https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/memory-access-ordering-part-2---barriers-and-the-linux-kernel” [Ultimo acceso 20/septiembre/2019].

- [22] Semáforos. ” https://es.wikipedia.org/wiki/Sem%C3%A1foro_(inform%C3%A1tica)” [Ultimo acceso 20/septiembre/2019].

- [23Llamadas al sistema. ” http://sop.upv.es/gii-dso/es/t5-llamadas-al-sistema/kernel-architecture.png” [Ultimo acceso 20/septiembre/2019].

- [24] Espacio Kernel Linux. ” https://es.slideshare.net/klserranog/el-kernel-en-los-sistemas-operativos” [Ultimo acceso 20/septiembre/2019].

- [24] Barreras software en kernel Linux. ”https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/memory-access-ordering-part-2---barriers-and-the-linux-kernel}” [Ultimo acceso 20/septiembre/2019].

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá