Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño e implementación de entorno de ejecución para sistemas embebidos con arquitectura ARM Cortex-A

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Leandro Rafael Llano Sánchez

Tutor: José Ismael Ripoll Ripoll

2019-2020

Dedicatoria

"Le dedico este trabajo a mis padres por su gran esfuerzo y dedicación, por su cariño y apoyo incondicional, por haberme dado la oportunidad de estudiar esta carrera y apoyarme durante esta bonita etapa de mi vida…por confiar siempre en mí, incluso cuando yo no lo hacía…

A mi hermana mayor y hermano pequeño, por ser también personas increíbles y estar en mi vida en todo momento.

A mi mejor amigo por estar siempre en las buenas y en las malas, y a todas aquellas personas que de alguna u otra forma han aportado cosas positivas en mi vida…"

Agradecimientos

"*Quiero dar las gracias a José Ismael Ripoll Ripoll por aceptar ser mi tutor para la realización de este trabajo, por haberme dado interesantes ideas y sugerencias para elaborarlo.*

A mi amigo Víctor Martínez, por haberme enseñado tantas cosas a lo largo de estos últimos años y por haberme motivado en cierta medida a aprender sobre las cosas que hoy en día me apasionan de la informática."

Resumen

Cada vez más los sistemas embebidos tienen mayor relevancia en la vida cotidiana, están presente alrededor de todos y en las cosas que menos se piensan.

Debido al gran uso de estos sistemas, en el mundo laboral existen muchas oportunidades para trabajar con el desarrollo de aplicaciones embebidas, sin embargo, no hay gran demanda de estas oportunidades. Probablemente, porque trabajar con sistemas embebidos requiere un conocimiento profundo sobre arquitecturas de computadores, sistemas operativos y lenguajes de programación de bajo y medio nivel; cosas que resultan mucho más complejas que desarrollar aplicaciones típicas de alto nivel.

Ante dicha situación, en este proyecto se diseña y posteriormente se implementa un entorno de ejecución ligero de 32 bits para sistemas embebidos con arquitectura ARM Cortex-A, donde el usuario podrá ejecutar una o varias aplicaciones de forma concurrente en dicho sistema, ya que utiliza la técnica de multiprogramación. Para lograr esto, el entorno de ejecución incluye una versión reducida de la interfaz POSIX Threads.

También ofrece al usuario un conjunto de servicios para el uso de periféricos y protocolos de comunicaciones, todo ello sin tener que realizar complejas configuraciones. De esta forma se busca que el usuario se sienta atraído en el mundo de los sistemas embebidos y que obtenga los conocimientos básicos de una forma sencilla.

Palabras clave: sistemas embebidos, arquitecturas de computadores, entorno de ejecución, sistemas operativos, lenguajes de programación, ARM Cortex-A, POSIX Threads.

Abstract

Embedded systems grow more relevant in our day to day lifestyle. They are present amongst everyone of us; in things you would not suspect. Due to the extensive potential use of these systems, opportunities to work with embedded application development grow extensively in the workplace, however, the demand for these opportunities is low. Most likely because working with embedded systems requires deep knowledge about computer architecture, operating systems, and programming languages from low to medium level; which happen to be much more complicated than developing typical high level apps.

In this project, a light 32 bits runtime system is designed and implemented for embedded systems, with ARM Cortex-A architecture. The user will be able to run one or more apps, in a concurrent manner making use of multiprogramming technic. To achive this, the runtime system includes a reduced version of the POSIX Threads interface.

It also offers to the user a set of services for the use of peripherals and communication protocols All of it without the need to perform complex configurations. Thus, making the user more receptive to the world of embedded systems, and making it more likely that they would acquire the basic knowledge in a simpler manner.

Keywords : embedded systems, computer architecture, runtime system, operating system, programming languages, ARM Cortex-A, POSIX Threads.

Índice de contenido

Índice de figuras

Índice de bloques de código

1. Introducción

Hoy en día ARM cuenta con más de 170 billones de procesadores producidos [1], siendo la arquitectura más usada en todo el mundo. Muchas de las cosas que nos rodean en nuestro día a día están dotadas de un chip ARM, como puede ser, un teléfono móvil, Smart TV, coche, dispositivo de IoT, aparato de monitorización médica, entre otros.

Debido a la sencillez, bajo consumo y bajo coste de estos chips son usados por un gran número de sectores para el desarrollo de aplicaciones embebidas. Cada vez más los sistemas embebidos tienen más relevancia y son utilizados para casi cualquier cosa que se pueda imaginar, como por ejemplo los objetos mencionados anteriormente.

Puesto que, nos encontramos tan rodeados de sistemas embebidos resulta interesante tener conocimientos más profundos sobre ellos, sobre su funcionamiento, saber cómo hacen aquello que desde una visión no tan profunda pueda parecer magia. Logrando así, llegar a estar más integrado en este mundo donde la tecnología cada día está más presente en la vida cotidiana.

1.1. Motivación

La motivación para realizar este trabajo surge prácticamente desde el inicio del grado, cuando se nos enseñó el lenguaje de bajo nivel ensamblador. Aprender dicho lenguaje, el aprender a escribir instrucciones máquina, jugar con los registros del procesador y mover datos entre la memoria, aparte de darme una gran satisfacción, sensación de poder de control sobre lo que hace la CPU, me aportó una visión interesante y algo más profunda de cómo funciona una computadora.

Luego con el estudio de asignaturas como fundamentos de sistemas operativos y estructura de computadores sentí totalmente la curiosidad, necesidad y ganas de aprender de una forma profunda como funciona internamente una computadora, y muchos de los dispositivos que nos rodean.

Querer saber cómo la CPU hace las cosas y cómo hago que realice aquello que deseo, aprender cómo un sistema operativo funciona, cómo es capaz de ejecutar uno o varios procesos, querer saber cómo el sistema operativo puede establecer comunicaciones con los periféricos para comunicarnos con otros dispositivos y/o también poder interactuar con el usuario. Esas son las principales cuestiones que me motivaron a realizar este trabajo.

Durante el aprendizaje de las herramientas y tecnologías necesarias para la realización de este trabajo me interese mucho por los sistemas embebidos ya que están presente de forma constante en nuestras vidas.

De ahí que, otra motivación que me surgió fue la de crear un entorno de ejecución para ofrecerle al usuario un sistema donde poder desarrollar y ejecutar sus aplicaciones embebidas de una forma sencilla y así fomentar sus conocimientos e interés por este apasionante mundo.

1.2. Objetivos

El objetivo de este trabajo es diseñar e implementar un entorno de ejecución ligero de 32 bits para sistemas embebidos con procesadores ARM. Al ser ligero se consigue ejecutar en máquinas con muy pocos recursos, permitiendo que se pueda ejecutar en un gran número de dispositivos.

También se busca ofrecer al usuario un conjunto de servicios para hacer uso de las distintas interfaces de comunicaciones del chip y así poder establecer comunicación con los periféricos de una forma fácil, sin tener que hacer complejas configuraciones de estas.

1.3. Estructura de la memoria

El documento está estructurado por una serie de capítulos que se detallarán a continuación:

- **Introducción:** se pone de manifiesto la idea en sí del proyecto, la motivación que hay para realizarlo y el objetivo que se persigue.
- **Estado del arte:** se presenta una tecnología similar a la de este proyecto y se hace una comparación entre las tecnologías mencionadas, exponiendo la solución que ofrece este proyecto.
- **Tecnología utilizada:** se detallan las tecnologías y lenguajes de programación utilizados, a la vez que se explica por qué se han elegido para realizar este proyecto.
- **Diseño:** se detallan los bloques funcionales que componen el proyecto y se explican sus necesidades y funciones.
- **Implementación:** se muestran detalles de la implementación de los bloques funcionales que componen el entorno de ejecución.
- **Pruebas de funcionamiento:** en este capítulo se muestran pruebas del correcto funcionamiento del entorno de ejecución.
- **Conclusiones:** se expondrán las conclusiones obtenidas tras la realización de este proyecto y la relación de este con los estudios cursados en el grado.
- **Trabajos futuros:** se muestran las ideas que se pretenden llevar a cabo con este proyecto en un futuro.
- **Bibliografía:** referencias de documentos consultados para la realización del proyecto.
- **Glosario:** listado de términos técnicos utilizado en este documento.

2. Estado del arte

Cada vez más el entorno de las personas se encuentra rodeado de sistemas embebidos y son utilizados para llevar a cabo tareas desde simples a muy complejas.

Debido a este creciente uso de los sistemas embebidos, es normal que surjan tecnologías que permitan hacer un manejo adecuado de estos y que ofrezcan al desarrollador de aplicaciones embebidas ciertas características para poder llevar a cabo tareas específicas.

Algunas de estas tecnologías son:

- **FreeRTOS [2]:** es un sistema operativo de tiempo real [3] para sistemas embebidos. Está disponible para varias arquitecturas, ARM es una de ellas, está más orientado a los Cortex-M, pero también puede ser ejecutado en un Cortex-A.

  Está disponible para unas 35 plataformas distintas de microcontroladores, haciendo que sea un sistema versátil, a la vez que se consigue que las aplicaciones puedan ser portables en distintos microcontroladores sin apenas hacer grandes cambios.

  Es un sistema bastante pequeño, apenas cuenta con unos 3 ficheros. Lo que ofrece básicamente es el uso de subprocesos, temporizadores y mecanismos de sincronización como los mutexes y semáforos.

  Realmente más que un sistema operativo, podría ser considerado como una librería para usar subprocesos y planificarlos, ya que no ofrece otras utilidades que podrían encontrarse en un sistema operativo de propósito general.

- **JamaicaVM [4]:** es una máquina virtual para desarrollar y ejecutar aplicaciones Java para sistemas embebidos. Está diseñada principalmente para aplicaciones de tiempo real.

  Ofrece librerías estándar de Java para un rápido desarrollo e incluye herramientas para hacer compilación cruzada, trazas en el código y emularlo. Además, está disponible para varias plataformas.

- **eCos [5]:** es un sistema operativo de tiempo real para sistemas embebidos, disponible para arquitecturas como ARM, X86, MIPS y PowerPC. Entre las funcionalidades que ofrece este sistema operativo se encuentran: abstracciones de hardware, manejadores de excepciones y dispositivos, soporte de subprocesos, soporte de red y APIs compatible con POSIX.

Existen muchas más tecnologías similares, pero describirlas todas no es el objetivo de este trabajo, por eso sólo se han nombrado unas pocas.

2.1. Crítica al estado del arte

FreeRTOS es un sistema bastante interesante, utilizado mundialmente para sistemas de tiempo real, además, resulta muy interesante por ser ligero y estar disponible para un gran número de plataformas.

Una de las ventajas que presenta el entorno de ejecución de este proyecto frente a FreeRTOS, es que ofrece ciertos controladores de dispositivos, sin que el usuario tenga que hacer complejas configuraciones, lo que lo hace práctico para el desarrollo de aplicaciones embebidas, para usuarios que apenas se adentran en este mundo.

Frente a JamaicaVM, la ventaja principal, es que las aplicaciones se ejecutan directamente en la máquina, no necesitando de una máquina virtual que las tenga que interpretar, para luego llevar a cabo la ejecución. Además, no se utiliza un lenguaje como Java para desarrollar las aplicaciones, lo que aumenta las prestaciones del entorno de ejecución. Esto último se comentará con detalle más adelante.

La solución creada en este trabajo resulta ser más similar a eCos, comparten muchas características.

3. Tecnología utilizada

Para llevar a cabo la realización de este proyecto ha sido necesaria la utilización de un grupo de tecnologías, tanto a nivel de hardware como a nivel de software.

A continuación, se presentan las tecnologías empleadas.

3.1. Procesador

Este proyecto se realiza para la arquitectura ARM [6]. ARM es una arquitectura RISC de 32 bits. También cuenta con chips de 64 bits en la versión 8 de ARM. Al ser una arquitectura con un diseño RISC, los procesadores necesitan una menor cantidad de transistores para su producción, a diferencia de una arquitectura CISC como la x86_64.

El hecho de necesitar menos transistores, le otorga unas características interesantes, tales como, un diseño más sencillo, un coste de producción más bajo y sobre todo un bajo consumo energético, lo que hace que a su vez se calienten menos.

Al hacer un bajo uso energético y, por tanto, calentarse menos, hace que esta arquitectura sea la perfecta para aquellos dispositivos que trabajan con baja potencia energética y/o que dispongan de baterías, como por ejemplo una Tablet, un móvil o incluso una sonda espacial.

En 2005, casi el 100% de los teléfonos móviles producidos estaban dotados de un procesador ARM y aproximadamente a partir de 2009 los procesadores ARM abarcan cerca del 90% de los procesadores de 32 bits RISC utilizados.

ARM ofrece varias alternativas de chips en base a la necesidad de la aplicación. Estas alternativas son:

1. **ARM Cortex-R (Tiempo real):** orientados a ser utilizados en sistemas de tiempo real, donde se requiere de una respuesta de los estímulos recibidos del entorno en un intervalo de tiempo determinado.

2. **ARM Cortex-M (Microcontrolador):** orientado a sistemas de bajo coste y donde se necesite baja latencia en el procesamiento de las interrupciones.

3. **ARM Cortex-A (Aplicación):** orientados más al uso general y al alto rendimiento, contando con varios núcleos y con todas las características para ser capaz de ejecutar un sistema operativo de propósito general en óptimas condiciones.

Estos procesadores suelen estar integrados en lo que se conoce como sistema en chip, dado el gran interés que hay en diseñar chip que sean de un tamaño reducido y así tener mayor capacidad de integración y dispositivos más pequeños. Un SoC es un chip que tiene integrado todos los componentes necesarios para el correcto funcionamiento del sistema [7].

Algunos de los muchos componentes que componen un SoC son:

1. Un microcontrolador o un microprocesador con varios núcleos.
2. Módulos de memoria como RAM, ROM, EEPROM, eMMC.
3. Controladores de memoria.
4. Generadores de frecuencia.
5. GPU.
6. Periféricos de reloj (contadores, temporizadores).
7. Controladores para comunicarse con interfaces como SPI, UART, I2C, USB, Ethernet, entre otros.
8. CAD y CDA

Para la realización de este trabajo y prueba de este, se ha elegido la placa Orange Pi PC, que cuenta con un SoC Allwinner H3, cuyo SoC cuenta con una CPU ARM Cortex-A7 de cuatro núcleos, a una velocidad de 1.6 GHZ y una memoria RAM DDR3 de 1 GB.

Perfectamente pudo haber sido elegida otra placa con un Cortex-A, como por ejemplo una Raspberry Pi.

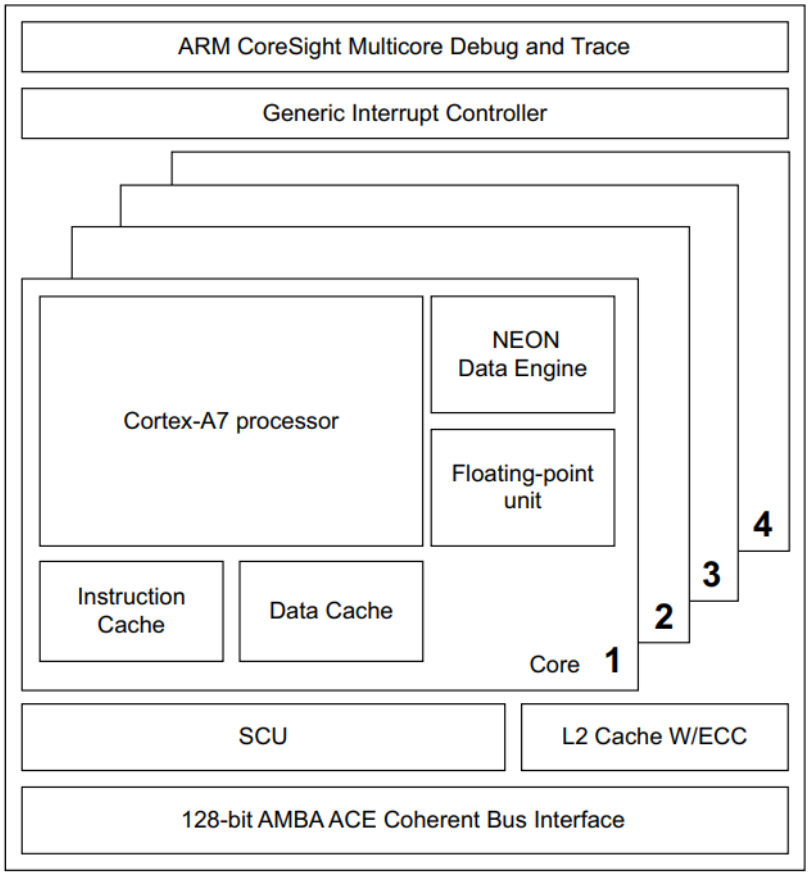

Figura 1. Diagrama procesador ARM Cortex-A7. Copyright 2011 – 2013 ARM [8].

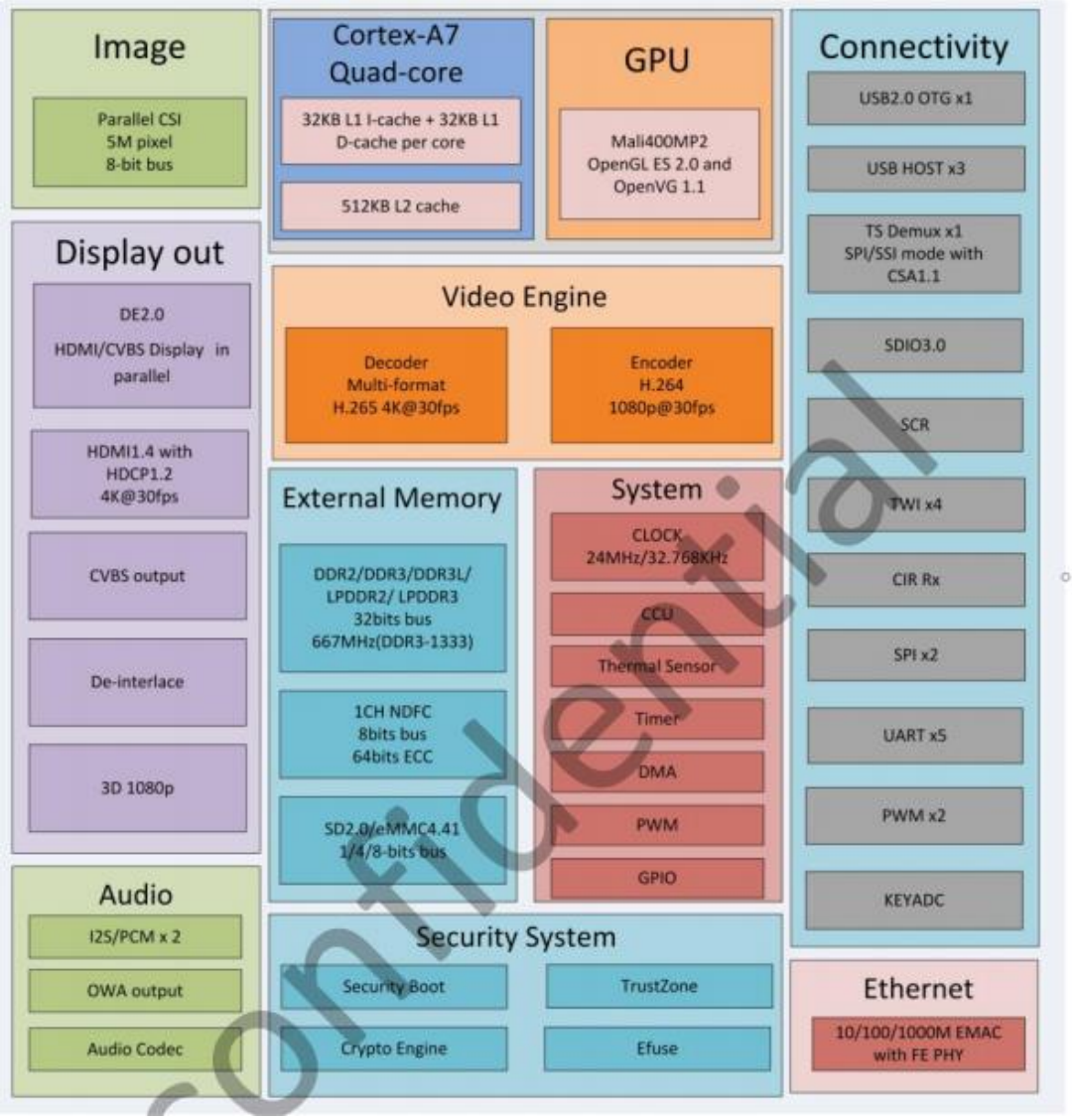

Figura 2. Diagrama del SoC Allwinner H3. Copyright 2014 Allwinner Technology Co [9].

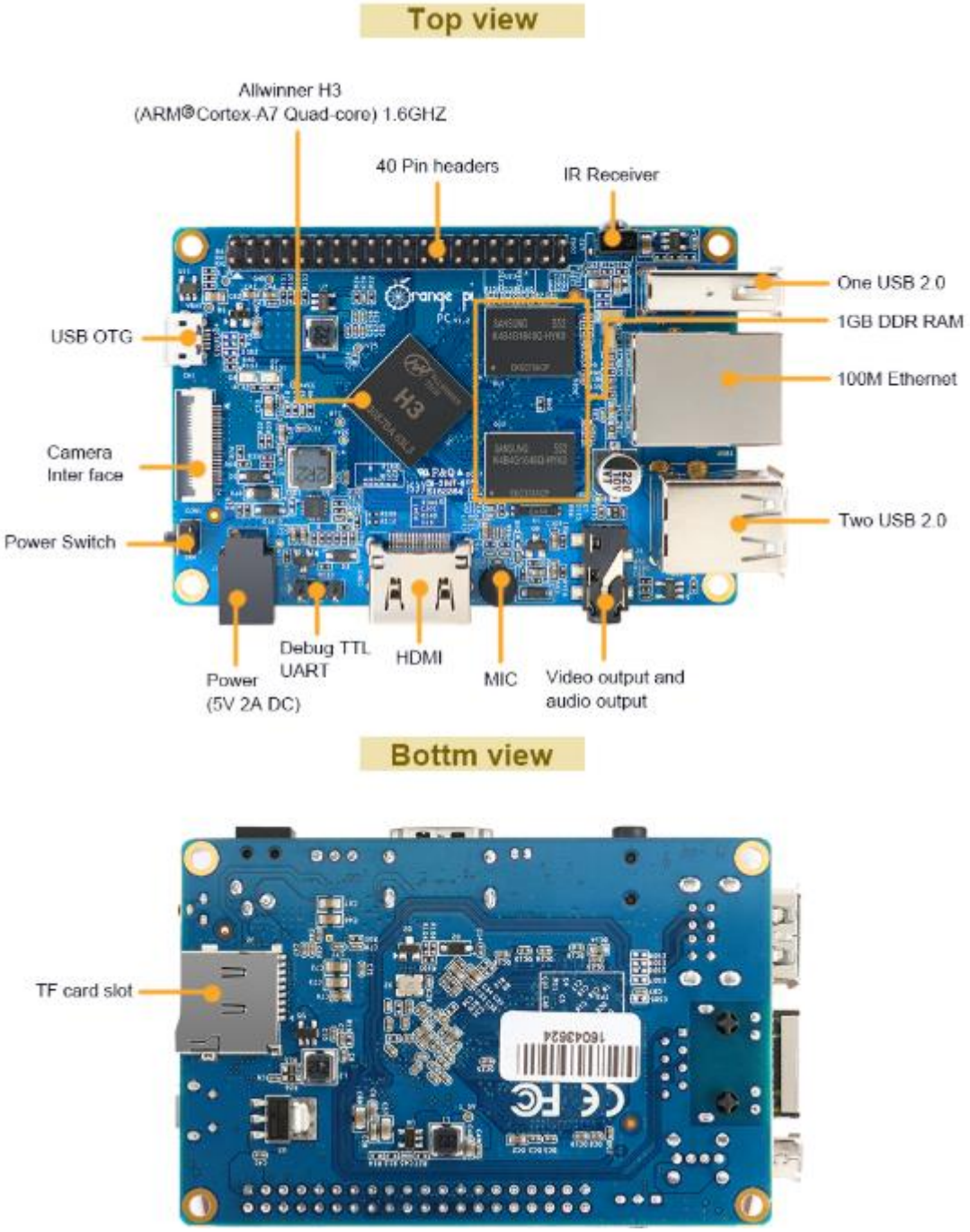

Figura 3. Orange Pi PC.

3.2. Toolchain

Un toolchain es un conjunto de herramientas de programación utilizadas para el desarrollo de software, desde una aplicación cualquiera hasta el desarrollo de un sistema operativo.

Algunas de ellas son:

1. **GNU Compiler Collection:** conjunto de compiladores para distintos lenguajes de programación. Además, permite la realización de compilación cruzada[1].

2. **GNU Binutils:** entre las muchas herramientas de programación que tiene esta colección, se pueden encontrar las siguientes.

   Un ensamblador.

   Un enlazador, herramienta que se encarga de obtener un fichero ejecutable a partir de los ficheros objetos.

   Objdump, usado para desensamblar ficheros objetos y obtener más información al respecto. Resulta útil a la hora de depurar código, al igual que la herramienta readelf.

   Otra herramienta interesante es objcopy, para copias de ficheros objetos.

3. **GNU Debugger:** es un depurador. Que ofrece la posibilidad de hacer trazas de código en un determinado punto, visualizar el contenido de los registros de la CPU en un momento dado y ver el contenido de una zona de memoria, entre otras cosas.

4. **GNU Make:** es una herramienta que gestiona las dependencias entre los archivos de código fuente para llevar a cabo su recompilación en caso de que sea necesario. De forma que sólo se recompilará aquellos ficheros que hayan sufridos cambios y no todos los ficheros de código fuente, así se ahorra tiempo de compilación, ya que grandes proyectos pueden tardar horas en compilarse.

Concretamente para este proyecto se utilizará GNU ARM Embedded Toolchain [10].

3.3. El gestor de arranque

El gestor de arranque es una de pieza de código, fundamental. Es el código, cuya misión, es cargar el kernel en la RAM, de hacer las inicializaciones básicas del hardware y preparar el sistema para la ejecución de código de medio o alto nivel, entre otras cosas.

Una vez este termina de hacer todo lo mencionado anteriormente, le cede el control al kernel del sistema y este a su vez termina de inicializar todo el sistema por completo. Todo esto será descrito con detalle más adelante.

[1] **compilación cruzada:** proceso de obtener un ejecutable para una plataforma diferente a la de donde se compila.

Para este proyecto se utilizará Das U-boot [11], que es un gestor de arranque universal, muy utilizado en sistemas embebidos. En este proyecto u-boot es cargado por el código que hay en la ROM del SoC. Dicho gestor de arranque inicializa las memorias y realiza otras configuraciones muy básicas en el hardware. Posteriormente carga la imagen del kernel en RAM y cede el control a un gestor de arranque de segunda etapa, que será descrito más adelante.

3.4. Sistema operativo

Entre las funcionalidades que ofrecen muchos de los sistemas operativos de hoy en día, se puede encontrar la interfaz POSIX Threads[2] [12]. Esta interfaz de threads proporciona una librería estándar para uso de subprocesos en un sistema operativo o entorno de ejecución.

El entorno de ejecución de este trabajo ofrecerá una versión reducida de la interfaz de threads de POSIX [13], pero compatible con esta, para permitir el desarrollo de software más complejo y útil, permitiendo utilizar técnicas de concurrencia. A la vez que ofrece portabilidad con sistemas basados en Unix en cuanto al uso de threads se refiere.

Además, ofrecerá un conjunto de manejadores de dispositivos para la comunicación con dispositivos de una forma sencilla. Se verá con detalle más adelante.

3.2. Lenguajes de programación utilizados

A la hora de desarrollar un software es conveniente utilizar lenguajes de programación adecuados, ya que permitirán realizar ciertas funciones que en un determinado contexto otro lenguaje de programación no podría realizar. Además, utilizar el lenguaje de programación adecuado dará como resultado un software más eficiente y robusto, todo esto acompañado de buenas prácticas de programación, por supuesto.

Para el desarrollo de este entorno de ejecución se utilizará el lenguaje ensamblador y C.

El entorno de ejecución será un sistema “bare metal” ya que se ejecutará de forma independiente y directa en el hardware, sin la necesidad de un sistema operativo para su ejecución.

Debido a que se necesitará hacer uso directo del hardware es imprescindible usar el lenguaje de bajo nivel ensamblador. Ya que será necesario tener acceso a los registros del procesador para poder configurar de forma adecuada los distintos modos en los que se puede encontrar la CPU en un momento dado.

También se necesitará hacer uso de los distintos coprocesadores que tiene el chip ARM, con el objetivo de limitar el número de núcleos que se encuentran activo en un momento dado, configurar la MMU en caso de que se vaya a utilizar, configurar la memoria caché, entre otras cosas más. Todo esto será explicado más adelante detalladamente.

[2] **thread:** subproceso o tarea que es ejecutada por un sistema operativo.

Es por todo ello, por lo que se precisa del lenguaje ensamblador, porque hay que hacer ciertas funciones sólo accesibles mediante instrucciones máquina. Además de que al principio la máquina no podrá ejecutar código de un lenguaje de medio ni alto nivel, sin antes preparar el sistema para ello.

También será imprescindible hacer uso de un lenguaje de medio nivel como C, ya que a la vez que proporciona características de un lenguaje de alto nivel, permite tener un control del sistema a muy bajo nivel. Además de que un código hecho en C tiene prácticamente una traducción muy directa al lenguaje ensamblador, que permite la incrustación de código ensamblador dentro de un código C, lo que lo hace un lenguaje muy versátil.

Resulta imposible usar un lenguaje de programación como Java para este proyecto. Ya que Java no permite al programador tener un control de bajo nivel sobre el hardware. No permite hacer acceso a una dirección de memoria cualquiera, incluso aunque esta perteneciera a tu espacio de direccionamiento. Java tampoco genera un binario, sino que, tras compilar un programa de este lenguaje, lo que se obtiene es un bytecode[3] que posteriormente es interpretado por la máquina virtual Java.

Y, por último, Java no podría usarse, porque para poder ejecutar código Java se necesita de una máquina virtual y dicha máquina virtual a su vez necesita ser ejecutada en un sistema operativo.

[3] **bytecode:** código intermedio que contiene instrucciones para ser interpretadas por un intérprete o máquina virtual.

4. Diseño

El entorno de ejecución de este proyecto tendrá una serie de bloques funcionales y características, de tal forma que todo ello da coherencia al sistema y hace posible el correcto funcionamiento del entorno de ejecución.

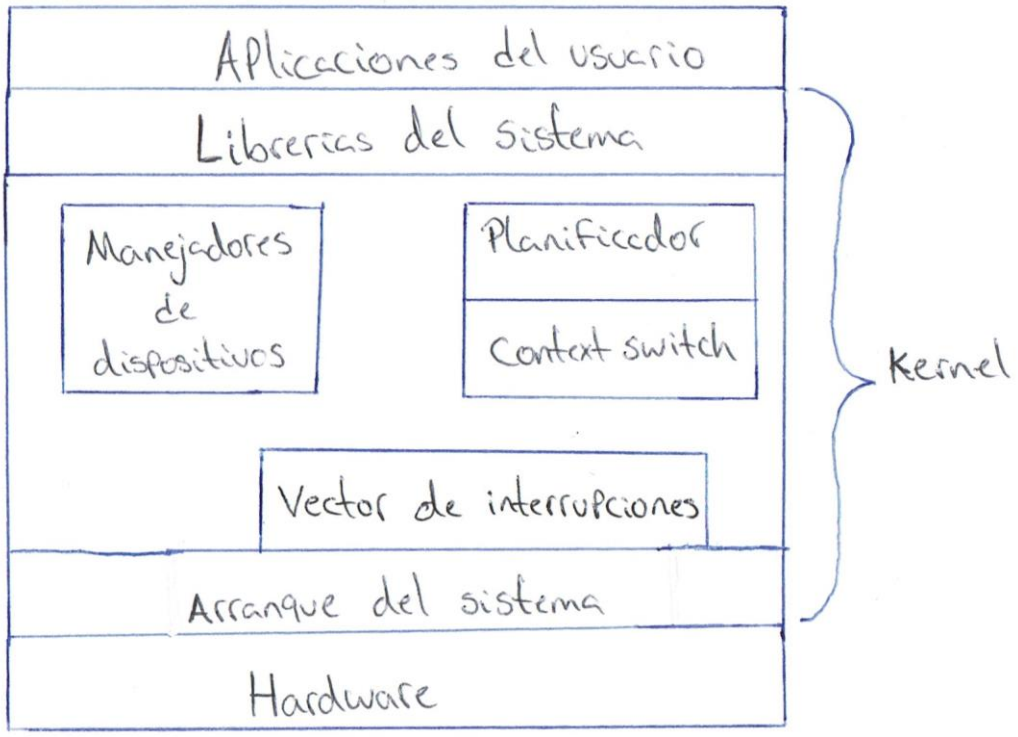

Figura 4. Diagrama del entorno de ejecución.

En el diagrama anterior, se pueden observar los bloques funcionales que componen el kernel del sistema. Dichos bloques funcionales serán descritos a continuación.

4.1. Vector de interrupciones

El vector de interrupciones es un vector donde se encuentran almacenadas las direcciones de las ISR, que son ejecutadas tras producirse una interrupción.

Dicho vector de interrupciones va alojado en una dirección específica. En ARM esta dirección puede ser la 0x00000000 o la 0xFFFF0000. La selección de dicha dirección puede hacerse poniendo a cero o a uno un bit de un registro de un coprocesador del chip ARM. Por defecto va en la 0x00000000 [14].

En el momento que se produce una excepción, antes de saltar a una de las direcciones almacenadas en el vector de interrupciones, el núcleo ARM almacena el estado actual de la CPU en el registro de almacenado de estado actual del modo en que se manejará la interrupción.

Luego guarda la dirección de retorno en el registro de enlace del nuevo modo y, por último, modifica el estado actual para cambiar el modo de la CPU.

Es muy importante tener en mente esta serie de pasos para manejar la nueva excepción de forma adecuada y posteriormente poder retornar al estado en el que se encontraba la CPU antes de que se produjera la interrupción tratada.

Para hacer uso del mecanismo de interrupciones es necesario configurar de forma adecuada el controlador de interrupción genérico.

El GIC [15] es un componente hardware dentro del chip ARM que se encarga de gestionar todo en cuanto a interrupciones se refiere. Es el hardware que hace el encaminamiento de las interrupciones desde un dispositivo hasta una o varias de las CPUs. También establece las prioridades de las interrupciones, de tal forma que, si en un momento dado llegan dos interrupciones, aquella interrupción con más prioridad será la primera en ser encaminada a la CPU.

Es capaz de gestionar interrupciones compartidas, interrupciones privadas e interrupciones software. Sin el funcionamiento del GIC, ninguna de las CPUs podrá ser interrumpida por sus "líneas de interrupción". Por tanto, es imprescindible inicializar y configurar el GIC.

Para ello, es conveniente contar con un controlador, para poder iniciarlo y configurarlo, logrando así que los dispositivos puedan interrumpir la CPU en un momento dado.

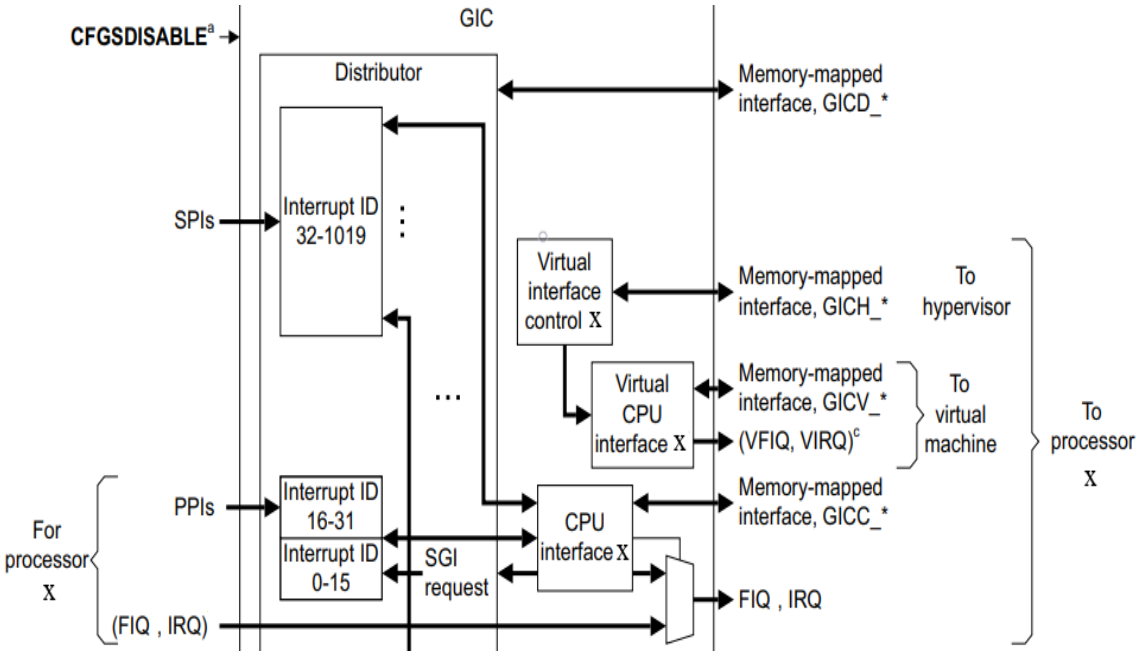

Figura 5. Diagrama del GIC. Copyright 2008, 2011, 2013 ARM [16].

4.2. Manejadores de dispositivos

Los manejadores de dispositivos son programas que permiten al sistema poder comunicarse con los periféricos, de tal forma que se logra el intercambio de información entre ambos.

El entorno de ejecución contará con una serie de manejadores para hacer posible el uso de algunas de las interfaces de comunicaciones y uso de los periféricos del SoC.

Algunos de estos manejadores de dispositivos son:

1. **UART:** dispositivo usado para la comunicación serial, de modo que permite la transmisión de datos en serie entre un dispositivo y el SoC.

2. **SPI:** estándar de comunicación usado para la transmisión serial sincrónica de información entre dispositivos electrónicos. Entre sus grandes ventajas, está la de ofrecer una alta velocidad de transmisión con sus periféricos.

3. **I2C:** estándar de comunicación serial y sincrónica.

4. **Microwire:** derivado de SPI, que ofrece conexión entre chips, periféricos y EEPROMs.

5. **Timer:** es imprescindible el uso de este periférico, ya que es el encargado de interrumpir al sistema de forma periódica, con el objetivo de llevar a cabo la ejecución del planificador y hacer un cambio de contexto cuando sea necesario. También el timer puede ser utilizado como un temporizador para determinados eventos o incluso para llevar cuenta del tiempo transcurrido desde que se inició el sistema.

Puede haber muchos manejadores más, pero a priori sólo se utilizarán estos manejadores por su relevancia y gran uso en los sistemas embebidos.

4.3. El planificador

El planificador [17] es quizás de los elementos más interesantes en este entorno de ejecución, ya que dotará al sistema de capacidad de multiprogramación, haciendo que puedan llegar a ejecutarse varios threads de forma concurrente a la vez que se hace un mejor uso de la CPU. Este entorno de ejecución por cuestión de sencillez se hará utilizando una sola CPU, por eso es imprescindible hacer multiprogramación.

De los distintitos posibles planificadores a ser elegidos para el entorno de ejecución, se eligió un planificador Round Robin.

Al elegir este planificador, se procedió a elegir el quantum que establecerá el tiempo máximo que un thread podrá hacer uso de la CPU en un momento dado. Para el entorno de ejecución en cuestión, se eligió un quantum de 100 ms. Usar un quantum menor haría que se produzcan muchos cambios de contextos, haciendo que el sistema no resulte tan eficiente.

Y usar un quantum mayor haría que el cambio de contexto se produzca más tarde y por tanto, los threads que no están haciendo uso de la CPU y que estén preparados para usarla tengan que estar mucho más tiempo inactivo y hará que el sistema no sea tan concurrente, por lo que no se aprovechará tan bien el mecanismo de multiprogramación.

Cada thread para dejar la CPU no tendrá necesariamente que agotar su quantum, abandonará la CPU cuando se bloqueé por intentar hacer uso de un recurso, cuando dicho recurso no se encuentre disponible o cuando decida dejar voluntariamente la CPU mediante la función yield.

El timer utilizado para la planificación de threads generará interrupciones cada 1 ms. Tras cada interrupción se ejecutará la función “context_switch”, que a su vez llamará al planificador y posteriormente hará un cambio de contexto, en caso de que el planificador haya elegido un nuevo thread para pasar a ejecución.

Se dispondrá de colas de tipo FIFO, para alojar a los threads que no están ejecutándose. Y se podrán en dichas colas según el estado en el que se encuentre un thread.

Habrá una cola para aquellos threads con estado “ready”, que son los threads que están preparados para pasar a ejecución. Un thread estará directamente en dicha cola tras su creación; también cuando el planificador lo saque de ejecución por agotar su quantum, por ceder la CPU voluntariamente, o bien cuando se desbloquee un thread que antes había estado bloqueado por esperar hasta poder hacer uso de un determinado recurso.

El planificador sólo les cederá la CPU a los threads en la cola para threads con estado “ready”.

Las demás colas son para aquellos threads que pasen a tener un estado “waiting”, que será cuando pasen a estar bloqueados por la espera de un recurso o por algún mecanismo de sincronización.

El hecho de que un thread se bloquee, cuando un recurso que desea utilizar no se encuentre disponible, resulta muy útil, mejora la eficiencia de la multiprogramación. Ya que, si un thread en vez de bloquearse se quedara haciendo una espera activa, estaría haciendo un uso innecesario de la CPU, ya que sólo estaría comprobando si el recurso se encuentra disponible.

Y esto no resulta eficiente, porque habrá otros threads queriendo usar la CPU para hacer operaciones más importantes y que requieran hacerse lo antes posible. Por tanto, lo ideal es ceder la CPU y pasar a una cola de espera, en vez de hacer una espera activa. Cuando el recurso esté disponible, el thread bloqueado será desbloqueado por otro thread o por un manejador.

De esta forma se consigue un uso más eficiente de la CPU, se utiliza de una forma útil y por tanto se consigue un ahorro energético, que es un tema de gran interés en los sistemas embebidos.

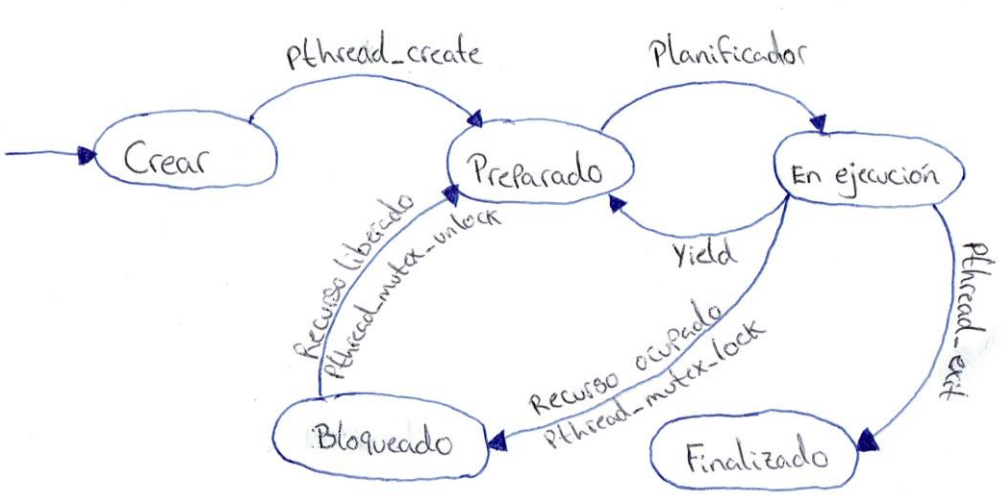

Figura 6. Diagrama del planificador.

4.4. El Context switch

El context switch es el trozo de código que es ejecutado tras producirse una interrupción de reloj y su función es alternar la CPU entre el proceso principal y los threads. La CPU será alternada por el context switch cuando el planificador así lo decida.

Sin el context switch la multiprogramación no sería posible ya que no se podría alternar la CPU entre distintos threads. Su funcionamiento será detallado más adelante.

5. Implementación

Antes de entrar en detalles con la implementación del trabajo, conviene tener claro la estructura de los registros del Cortex-A de 32 bits.

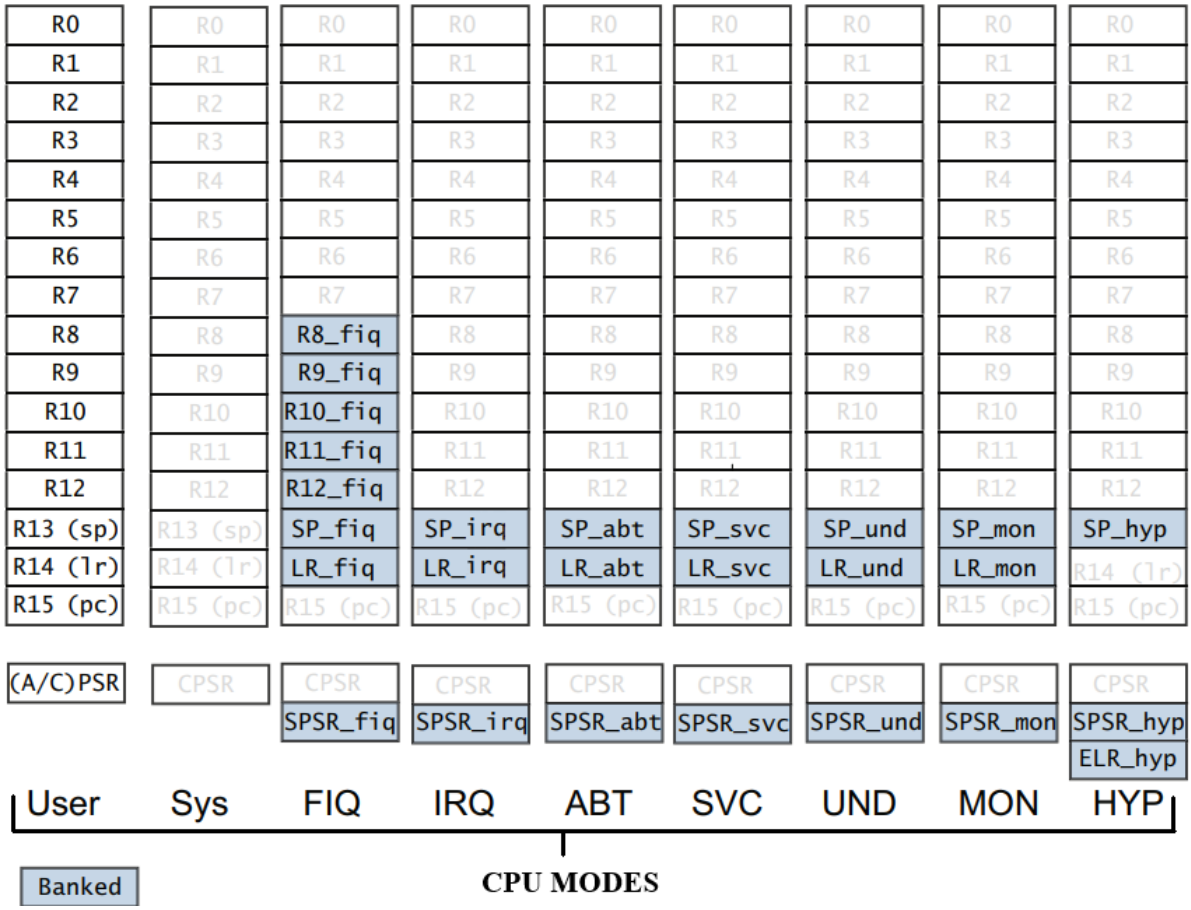

Figura 7. Conjunto de registros de ARM Cortex-A 32 bits [18].

Como se puede observar en la figura anterior, la CPU cuenta con varios modos de funcionamiento. A continuación, serán descritos los modos más relevantes para este proyecto, los demás modos pueden consultarse en el manual ARM que se deja como referencia en la memoria.

Modos del procesador:

1. **USER:** modo en el que se ejecutan las aplicaciones de usuario.

2. **SYS:** modo privilegiado usado por el sistema operativo y con el que puede acceder de forma directa a los registros y pila del modo USER.

3. **FIQ:** modo usado para interrupciones que deben atenderse super rápido.

4. **IRQ:** modo usado para atender el resto de las interrupciones.

5. **ABT (abort):** modo en el que entra la CPU tras producirse una excepción en el acceso a una dirección de memoria.

6. **UND (undefined):** modo en el que entra la CPU tras ejecutarse una instrucción indefinida.

7. **SVC:** modo kernel o supervisor, es el modo usado tras el reinicio de la máquina y tras una llamada al sistema. También es el modo usado por el kernel en general.

Todos estos modos son privilegiados, excepto el modo USER.

También se puede observar que se tiene 16 registros, todos ellos pueden ser usados como registros de propósito general, excepto el registro R15 que es el contador de programa. Además, hay un registro llamado CPSR que es el que almacena el estado actual de la CPU. Dentro de este registro, hay bits para comprobar los flags de resultados de operaciones aritméticas, bits para activar y desactivar las interrupciones, un bit para cambiar el juego de instrucciones de un juego de instrucciones de 32 bits a 16 bits o viceversa, bits para cambiar el modo de la CPU, etc.

Los únicos dos modos que comparten los 16 registros son los modos USER y SYS, mientras que los demás modos, sólo comparten un parte de ellos. Tienen lo que se conoce como “banked register” que son unos registros alternativos, sólo accesibles desde el modo en el que están disponible. Son los sombreados en azul en la figura anterior.

Cada uno de los modos, a excepción de USER y SYS, cuenta con su propio registro de pila, registro de enlace y un registro SPSR. En el registro SPSR [19] es donde se almacena el CPSR que hay en un determinado momento antes de que se produzca una excepción. Al salvarse el CPSR en el SPSR_mode, la CPU es capaz de volver al estado actual en el que se encontraba antes de que se produjera la excepción. También haría falta salvar el contexto de los demás registros.

En el único modo que no haría falta salvar el contexto de los registros generales, es en el modo FIQ, ya que es el modo que cuenta con más registros alternativos, los suficientes para poder hacer muchas operaciones sin necesitar escribir en los demás. Sólo habría que salvar el contexto de los registros, si se utilizan aquellos registros que son los comunes a los otros modos.

5.1. El arranque del sistema y la función init

Una vez que U-boot ceda el control de la máquina, se procede a terminar de hacer las configuraciones necesarias para arrancar el sistema por completo. Lo primero que se procede a hacer es desactivar todas las CPUs que tiene el procesador menos una. Si no se desactivan las demás CPUs estarían haciendo las mismas configuraciones en el sistema cuatro veces, cosa que

no conviene porque no habría coherencia luego. Además, en este entorno de ejecución sólo se hará uso de una CPU. Las demás CPUs se dejan inactivas en vez de estar en espera activa, así se ahorra más energía.

Luego, el vector de interrupciones, que se encuentra en el ejecutable, se copia a la dirección de memoria 0x00000000 que es a donde irá el contador de programa una vez se produzca una excepción o interrupción. Si dicho vector no se copia a la dirección correspondiente, el sistema dejará de funcionar tras producirse una excepción, debido que, al producirse una excepción se intentará ejecutar una instrucción que no es válida, lo que produciría a su vez otra excepción que no se sabría cómo manejar.

A continuación, se puede ver el código que realiza lo que se ha comentado anteriormente.

```
1     @ disable all cores except core #0
2     mrc p15, 0, r0, c0, c0, 5
3     and r0, r0, #0x03
4     cmp r0, #0x00
5     beq init_boot
6
7     wait_loop:
8       wfi
9       b wait_loop
10
11  init_boot:
12    @ copy the vector table from __vectors_start to 0x00000000
13       ldr r0, =__vectors_start
14       mov r1, #0x00
15       ldmia r0!, {r2-r9}
16       stmia r1!, {r2-r9}
17       ldmia r0!, {r2-r9}
18       stmia r1!, {r2-r9}
```

Código 1. Selección de CPU y copia de vector de interrupciones.

En la línea 2 las CPUs acceden al coprocesador 15 para cargar el valor del registro que almacena su ID y cuyo valor se obtiene tras la instrucción “and” en la línea 3. Luego, dicho valor se compara con cero. El procesador principal, el que tiene ID igual a cero, es el que saltará a “init_boot” para copiar el vector de interrupciones a la dirección correspondiente. Las demás CPUs se suspenderán en la línea 8.

En la línea 13 la CPU con ID cero, carga en el registro r0 la dirección fuente del vector de interrupciones y en la próxima línea carga el registro r1 con la dirección destino. En las líneas posteriores mediante instrucciones load/store múltiples se leen los bytes de la dirección fuente y se copian en la dirección destino. En total se copian 64 bytes.

Otras de las cosas que se hacen en el arranque del sistema, es hacer un flush (limpiar e invalidar) en la memoria caché, debido a que tras el reinicio su contenido estará indefinido; también se desactiva la MMU, se invalida la TLB debido a que se trabajará con direcciones físicas y se limpia la sección BSS.

Se inicializa la pila de los modos SYS, IFQ, IRQ, ABORT, UNDEF con un espacio de 4 KB y posteriormente se inicializa la pila del modo SVC con un espacio de 8 KB y cuya pila es la del proceso principal. Las pilas de los threads también serán de 8 KB.

Una vez, inicializada la pila del modo SVC, el sistema ya podrá ejecutar código en C, por lo que se procede a llamar a la función "init" que termina de inicializar todo el entorno de ejecución.

A continuación, se puede ver el código que inicializa las pilas y llama a init. Por simplificar sólo se mostrará la inicialización de la pila del modo IRQ y modo SVC, ya que para el resto de los modos se hace igual.

```
1    @ initialize stacks of core modes
2     ldr r0, =sys_stack_base
8
9     @ IRQ mode
10   msr cpsr_c, #0xD2
11   mov sp, r0
12   sub r0, r0, #0x1000
13
14   @ switching to SVC mode and setting stack
15   msr cpsr_c, #0xD3
16   mov sp, r0
17   ldr r1, =sys_stack
18   str r0, [r1]
19
20   @ call init
21   b init
```

Código 2. Inicialización de las pilas.

En la línea 2, en el registro r0 se carga la dirección base de pila, es decir, la dirección a partir de la cual se asignarán las pilas. Para inicializar la pila del modo IRQ y por tanto tener acceso al registro sp de dicho modo, hace falta poner la CPU a modo IRQ, que es lo que se hace con la instrucción de la línea 10. Posteriormente en la línea 11, se pone la dirección base en el registro sp y en luego en la línea 12 a la dirección base se le decrementa el valor 0x1000 que se corresponde con un desplazamiento de 4 KB.

En la línea 15 se pasa nuevamente a modo SVC, para luego asignar el puntero de pila al registro sp en la línea 16 y cuyo puntero de pila se corresponde con la pila del proceso principal. Luego en la línea 17 y 18 dicho puntero de pila se almacena en una variable de C llamada "sys_stack" que es a partir de donde se asignarán las pilas para los threads, previamente decrementando 8 KB, por tanto, la pila del proceso principal y de los threads creados serán de 8 KB. Y, por último, se llama a la función principal init en la línea 21.

Una vez empieza a ejecutarse la función init se termina de inicializar la PCB del proceso principal, se inicia el GIC para que las interrupciones puedan ser encaminadas a la CPU utilizada, se inicializa el timer para que genere interrupciones cada 1 ms y por último se habilitan las interrupciones IRQ en el procesador.

Hecho todo esto, el entorno de ejecución estará listo para ejecutar varios threads.

5.2. Vector de interrupciones

Normal Vector offset	High vector address	Non-secure	Secure	Hypervisor[a]	Monitor
0x0	0xFFFF0000	Not used	Reset	Reset	Not used
0x4	0xFFFF0004	UNDEFINED instruction	UNDEFINED instruction	UNDEFINED instruction from Hyp mode.	Not used
0x8	0xFFFF0008	Supervisor Call	Supervisor Call	Secure Monitor Call	Secure Monitor Call
0xC	0xFFFF000C	Prefetch Abort	Prefetch Abort	Prefetch Abort from Hyp mode.	Prefetch Abort
0x10	0xFFFF0010	Data Abort	Data Abort	Data Abort from Hyp mode,	Data Abort
0x14	0xFFFF0014	Not used	Not used	Hyp mode entry	Not used
0x18	0xFFFF0018	IRQ interrupt	IRQ interrupt	IRQ interrupt	IRQ interrupt
0x1C	0xFFFF001C	FIQ interrupt	FIQ interrupt	FIQ interrupt	FIQ interrupt

Figura 8. Tabla de vector de interrupciones ARM Cortex-A [20].

A continuación, se puede observar la implementación en código ensamblador del vector de interrupciones mostrado en la figura anterior.

```
1     .section ".int_vector"
2     .align 4
3
4     .global __start
5     __start:
6
7     ldr pc, reset_vector          @ reset
8     ldr pc, undefined_vector      @ undefined instruction
9     ldr pc, syscall_vector        @ supervisor call
10    ldr pc, prefetch_vector       @ prefetch abort
11    ldr pc, data_vector           @ data abort
12    ldr pc, hyp_vector            @ hyp mode entry
13    ldr pc, irq_vector            @ IRQ interrupt
14    ldr pc, fiq_vector            @ FIQ interrupt
15
16    reset_vector:       .word reset_handler
17    undefined_vector:   .word undefined_handler
18    syscall_vector:     .word syscall_handler
19    prefetch_vector:    .word 0 @ prefetch_handler
20    data_vector:        .word data_handler
21    hyp_vector:         .word 0 @ hyp_handler
22    irq_vector:         .word irq_interrupt
23    fiq_vector:         .word fiq_handler
```

Código 3. Vector de interrupciones.

5.3. Manejadores de dispositivos

```
void init_uart1(void);
void uart1_send(uint8_t);
uint8_t uart1_recv(void);

void init_spi1(void);
void spi1_send(uint8_t);
uint8_t spi1_recv(void);

void init_microwire(void);
uint16_t microwire_read(uint16_t);
void microwire_write(uint16_t, uint16_t);

void init_i2c1(void);
void i2c1_send(uint8_t);
uint8_t i2c1_recv(uint8_t);

void init_lcd(void);
void lcd_send_cmd(uint8_t);
void lcd_send_char(uint8_t);

void init_timer1(void);
void clean_pending_timer1(void);
void enable_int_timer1(void);
void disable_int_timer1(void);
```

Código 4. Librería de manejadores de dispositivos.

Estas son las funciones de la librería que ofrece el entorno de ejecución para el uso de los distintos manejadores de dispositivos. Gracias a esta librería el usuario podrá desarrollar aplicaciones embebidas de una forma sencilla y sin hacer complejas configuraciones.

Se pueden encontrar funciones para uso de UART, SPI, Microwire, display lcd y uso de un timer.

5.4. Bloque de control de proceso

La PCB [21] es la estructura donde se almacena toda la información necesaria para trabajar con procesos. Cada vez que se crea un proceso se inicializa una PCB.

Dicha PCB es usada por el planificador para almacenar la información necesaria para poder llevar el proceso a ejecución por primera vez y tras haberse hecho un cambio de contexto.

A continuación, se puede observar la PCB utilizada para el entorno de ejecución realizado en este trabajo.

```
typedef struct pcb {
    struct pcb *next;
    uint32_t ticks;
    uint32_t pid;
    uint32_t status;
    uint32_t threads;
    uint32_t irq_lr;
    uint32_t sp;
    uint32_t sp_index;
    uint32_t struct pcb *queue;
    pthread_t thread_info;
} pcb_t;
```

Código 5. Implementación de la PCB.

Se puede observar que dicha estructura contiene varios campos. La función de cada campo es la siguiente:

1. **next:** es un puntero que almacena la PCB del próximo proceso a ser puesto en ejecución por el planificador.

2. **ticks:** son los ticks de reloj que lleva el proceso en ejecución. En este caso cada tick representa 1 ms transcurrido. Se utiliza para saber cuándo el proceso ha agotado su quantum.

3. **status:** es el estado actual en que se encuentra un proceso. Dichos estados pueden ser: ready, running, waiting y exit.

4. **threads:** indica el número de threads que tiene un proceso.

5. **irq_lr:** se almacena la dirección de memoria que tiene que ponerse en el contador de programa. Tras la creación del proceso, dicho campo guardará la dirección de memoria de la función main del programa a ejecutar y tras la planificación, almacenará la instrucción que se tenía que ejecutar justo en el momento de producirse una interrupción.

6. **sp:** es el puntero de pila del proceso.

7. **sp_index:** la zona de memoria utilizada para la pila está distribuida como un array, de tal forma que cuando se crea un proceso al pedirse un puntero de pila, también se necesitará el índice que le toco de aquel array. De esta forma luego cuando un proceso acabe podrá liberar dicha zona de memoria para ser reutilizada por otro proceso.

8. **queue:** es una lista que almacena a los procesos que se bloquean esperando que el proceso en cuestión libere un recurso que necesita otro proceso. Se utilizaría en un semáforo, por ejemplo.

9. **thread_info:** utilizada por los threads para tener acceso a su estructura pthread_t.

Todo lo mencionado sobre los procesos es igual para los threads, ya que estos también utilizan la misma PCB que los procesos. La única diferencia es que los threads no utilizan el campo threads. En este entorno de ejecución sólo se contará con un proceso ya que no se hace uso de la MMU y dicho proceso es el proceso principal que se comentó anteriormente. Por tanto, todas las demás PCBs utilizadas en el sistema pertenecerán a threads.

No se utiliza MMU debido a sencillez y restricciones de tiempo. Por tanto, el kernel y todos los threads serán ejecutados en un mismo espacio de direcciones físicas. Esto hace que haya más eficiencia al hacerse menos uso de hardware y es algo que interesa en los sistemas embebidos. Por el contrario, hay que ser precavidos a la hora de programar, ya que no hay protección de memoria.

Para el desarrollo de este entorno de ejecución se ha elegido poner un máximo en el número de threads que hay en un momento dado. El número máximo de threads que puede haber en un momento dado es de 127.

Todo esto puede ser modificado a gusto del usuario de una forma sencilla. Estos valores se establecieron a modo de prueba.

También se ha elegido establecer 1 MB para la zona de memoria de la pila. Y cada pila es de 8 KB por lo que se tiene un total de 128 punteros de pilas para asignar. De los cuales uno es para el proceso principal y los 127 restantes para los threads. El número máximo de punteros de pilas coinciden con la suma del número máximo de threads que puede haber en un momento determinado y el proceso principal, por lo que ambos valores deben coincidir si se modifican.

A partir de ahora por tema de comodidad se hablará de threads, en vez de distinguir siempre threads y proceso principal, pero se debe recordar que existe un proceso principal que es el que crea los threads y cuyo threads comparten su espacio de direcciones.

5.5. El planificador

```
void scheduler(void)
{
  if (running->ticks == QUANTUM || running->status == WAITING ||
running->status == EXIT) {
    running->ticks = 0;
    running->next = NULL;
    if (running->status == RUNNING) {
      running->status = READY;
      enqueue_pcb(&ready_queue, running);
    }
    running = dequeue_pcb(&ready_queue);
    running->status = RUNNING;
  } else {
    running->ticks++;
  }
}
```

Código 6. Planificador.

Como se puede observar, el planificador es una pieza de código muy sencilla. Este es llamado por la función “context_switch” cada 1 ms.

Tras cada llamada, siempre se incrementará en una unidad el número de ticks que lleva el thread actual, excepto en tres casos que se detallarán a continuación. En los tres casos, siempre se restablecerá a cero el número de ticks del thread y su puntero next será puesto a NULL por cuestiones de implementación de la función de encolar los PCBs.

Casos en los que el planificador elegirá otro thread:

1. **Quantum alcanzado:** cuando el número de ticks sea igual al quantum, entonces habrá agotado el quantum. En dicho caso, lo que hará el planificador es poner el estado del thread actual a “ready”, lo encolará en la cola de threads preparados y por último elegirá el thread en cabeza de la cola de threads preparados y cambiará su estado a “running”.

2. **Thread actual con estado “waiting”:** un thread pasará de estado “running” a “waiting” cuando se bloquee por la espera de un recurso que no se encuentra disponible. Cuando ese es el caso, el planificador no encola dicho thread a la cola de preparados, porque ya ha sido encolado en una cola de espera en otra parte de código. Por lo tanto, lo único que hace el planificador es elegir un nuevo thread de la cola de preparados al cual ceder la CPU.

3. **Thread terminado:** similar al caso anterior, pero aquí el thread no es encolado en ninguna cola. Y previamente se habrán liberado los recursos usados por el thread.

5.6. El context switch

El context switch es el corazón de la multiprogramación, es realmente el código encargado de alternar la CPU de un thread a otro, salvando de forma correcta el contexto de los registros del thread actual, para posteriormente restaurar el contexto de otro thread y posteriormente dar paso a su ejecución. Es ahí cuando empieza la multiprogramación.

A continuación, se verá detalladamente la función “context_switch” de este entorno de ejecución.

```
1 irq_interrupt:
2  sub lr, lr, #4
3  stmfd sp!, {r0-r12, lr}
4  ldr r0, =irq_lr
5  str lr, [r0]
6  movw r0, #0x2000
7  movt r0, #0x01C8
8  ldr r0, [r0, #12]
9  mov r1, #0x03FF
10 and r0, r0, r1
11 cmp r0, #50
12 beq context_switch
13 bl irq_handler
```

```
14 ldmfd sp!, {r0-r12, pc}^
15
16 context_switch:
17   movw r0, #0x2000
18   movt r0, #0x01C8
19   mov r1, #50
20   str r1, [r0, #16]
21   movw r0, #0x0C00
22   movt r0, #0x01C2
23   mov r1, #0x01
24   str r1, [r0, #4]
25   ldmfd sp, {r0-r3}
26   ldr r12, [sp, #0x30]
27   add sp, sp, #0x38
28   msr cpsr_c, #0x93
29   stmfd sp!, {r0-r12, lr}
30   ldr r1, =irq_lr
31   ldr r1, [r1]
32   ldr r0, =running
33   ldr r0, [r0]
34   str r1, [r0, #20]
35   str sp, [r0, #24]
36   bl scheduler
37   ldr r0, =running
38   ldr r0, [r0]
39   ldr r1, [r0, #20]
40   ldr sp, [r0, #24]
41   ldr r2, =irq_lr
42   str r1, [r2]
43   ldmfd sp!, {r0-r12, lr}
44   msr cpsr_c, #0x92
45   ldr lr, =irq_lr
46   ldr lr, [lr]
47   movs pc, lr
```

Código 7. Context switch.

Tras producirse una interrupción, la CPU entra modo IRQ y salta al manejador "irq_interrupt" para atender dicha interrupción. Lo primero que hace el manejador, es ajustar la dirección de retorno y guardar el contexto de los registros del thread. Dichas acciones se realizan con las instrucciones de la línea 2 y 3.

Posteriormente en las líneas 4 y 5, en una variable global llamada "irq_lr" se guarda la dirección de retorno, que es la que se encuentra almacenada en el registro de enlace, R14 o LR.

Luego, con las dos instrucciones de la línea 6 y 7 se carga en el registro r0 la dirección del distribuidor de interrupciones a la CPU del GIC, para posteriormente acceder a uno de sus registros y reconocer la interrupción, de esa forma se obtiene un ID, mediante el cual se sabe que interrupción se tiene que atender. Para obtener dicho ID hay que leer el registro "Interrupt Acknowledged Register (GICC_IAR)" del GIC. Todo ello se lleva a cabo en las líneas 8, 9 y 10.

Una vez obtenido el ID, en la línea 11, dicho ID se compara con 50 y si es igual a 50, entonces, se ha producido una interrupción de reloj. El ID de la interrupción del timer en el SoC utilizado es 50.

Al ser una interrupción de reloj, se procede a llamar a la función "context_switch" en la línea 12. En dicha función, lo primero que se hace es escribir en el registro "End Of Interrupt (GICC_EOIR)" del GIC, todo ello para notificar al GIC de que la interrupción en cuestión ha sido o está siendo tratada; además hace falta quitar el estado de "pending interrupt" en el timer, todo ello se hace con las instrucciones que hay entre las líneas 17 y 24. Si ninguna de estas acciones se realizan, la CPU siempre estará interrumpida y no será capaz de realizar nada más.

Hecho esto, en la línea 25 y 26 se procede a restaurar el valor de los registros que han sido usados para realizar las instrucciones intermedias y en la línea 27 se restablece el puntero de pila del modo IRQ a su dirección base, ya que dicha pila no se usará más.

En la línea 28 se cambia la CPU a modo SVC para tener acceso a la pila del thread que actualmente ocupa la CPU. Una vez cambiado el modo de la CPU, entre las líneas 29 y 35 se guardan los valores de los registros en la pila del thread en curso, se carga la PCB del thread y se almacena el valor de la variable global irq_lr en el campo irq_lr de la PCB, también se almacena el puntero de pila en el campo correspondiente.

Todo ello para poder restaurar el puntero de pila y la dirección de retorno en caso de que exista un cambio de contexto. Hecho esto, en la línea 36 se procede a llamar al planificador. Una vez se vuelve del planificador en la línea 37 se carga de nuevo la PCB del thread actual y dicho thread actual será un thread distinto en caso de que el planificador haya elegido un nuevo thread para que vaya a ejecución. Se accede a dicha PCB a través del puntero "running".

Posteriormente entre las líneas 38 y 42, en el registro de pila, R13 o SP se almacena el puntero de pila que se encuentra en la PCB cargada previamente y, además, el valor del campo irq_lr de la PCB se guarda en la variable global irq_lr. Hecho esto, entre las líneas 43 y 47 se restauran los registros con los valores almacenados en la nueva pila, se cambia nuevamente a modo IRQ y se carga el contador de programa con el valor de la variable global irq_lr, que contiene la dirección por donde se quedó un thread cuando se interrumpió por última vez, o bien contiene la dirección de la función para la que el thread fue destinado en el momento de su creación, tras una llamada a la función pthread_create.

La razón por la que context_switch se hace en lenguaje ensamblador, es por su facilidad de hacer operaciones de memorias y otras. También se necesita cambiar la CPU de modo varias veces y eso sólo puede hacerse desde el lenguaje ensamblador. Si se hace en C habría que mezclarlo con código ensamblador y además no es conveniente hacerlo en C ya que habría que hacer más operaciones de pila de las que realmente son necesarias.

Volviendo un poco atrás, si en la rutina irq_interrupt el ID de la interrupción es distinto a 50 y, por tanto, no es una interrupción de reloj, en la línea 13 se procede a llamar a un manejador de interrupciones secundario y está vez hecho en C y tras su vuelta, en la línea 14 se restaura el contexto de los registros y se retorna.

La función yield de la que se ha hablado anteriormente, sirve para ceder la CPU voluntariamente o cuando un thread se bloquea. La implementación de dicha función es en parte igual a context_switch, salvo que se ejecuta siempre en modo SVC. Por tanto, no habría que

hacer todas las operaciones que involucran cambios de modo de la CPU y tampoco haría falta reconocer ni limpiar ningunas interrupciones pendientes.

5.7. Interfaz POSIX Threads

El entorno de ejecución ofrece una versión reducida de la interfaz POSIX Threads [22]. Dichas funciones, permiten hacer uso de threads de una forma básica, como su creación, espera y terminación.

También permite la sincronización de estos por medio de mutexes para poder proteger secciones criticas o el acceso a un determinado recurso.

```
typedef struct {
  uint32_t tid;
  pcb_t *queue;
} pthread_mutex_t;

int pthread_create(pthread_t *, const pthread_attr_t *, void
*(*) (void *), void *);
int pthread_join(pthread_t, void **);
void pthread_exit(void *);
pthread_t pthread_self(void);
int pthread_mutex_init(pthread_mutex_t *, const
pthread_mutexattr_t *);
int pthread_mutex_lock(pthread_mutex_t *);
int pthread_mutex_unlock(pthread_mutex_t *);
int pthread_mutex_trylock(pthread_mutex_t *);
```

Código 8. Interfaz POSIX Threads.

5.8. Detalles de implementación

El entorno de ejecución se desarrollará de tal forma que habrá que hacer muy pocas implementaciones para ejecutarlo en un SoC distinto al usado para el desarrollo de este proyecto.

Para ejecutarlo en otro SoC, sólo habría que añadir nuevos ficheros de interfaces, para el uso de los periféricos y otros elementos del hardware. Estos ficheros serían necesarios porque cada SoC tiene mapeada las cosas en distintas direcciones. Por lo tanto, sólo haría falta esto, además, sólo se utilizará el lenguaje ensamblador cuando sea estrictamente necesario, de esa forma se consigue que el código sea menos dependiente de un hardware en específico.

Hecho esto, simplemente habría que elegir para que plataforma compilar el entorno de ejecución.

6. Evaluación

A continuación, se pueden observar capturas de pantalla de la ejecución del sistema.

Es importante aclarar que no se han hecho pruebas para medir el rendimiento del entorno de ejecución, ya que el objetivo sólo era validar la implementación.

6.1. Planificador

Figura 9. Demostración del planificador.

La figura anterior es la ejecución del planificador. En la primera línea, tras la llamada del planificador, se puede observar que el proceso principal es el que tiene actualmente la CPU.

La segunda línea es la cola de threads con estado “ready” que están en cola de preparados para pasar a ejecución.

La tercera línea es el resultado tras encolar el proceso principal a la cola de preparados.

La cuarta representa la cola tras desencolarse el siguiente thread a ejecutarse. Y, por último, el thread nuevo, reanuda su contexto de ejecución. Así de sencilla es la función del planificador.

6.2. Multiprogramación

```
#include <llanox/printk.h>
#include <asm/cpu.h>
#include <asm/gic.h>
#include <asm/timer.h>
#include <llanox/string.h>
#include <llanox/pthread.h>

void *test(void *data)
{
  pthread_t t = pthread_self();
  for (;;)
    printk(Thread  %d\n", t.tid);

  return NULL;
}

void init()
{
  pcb_t *p = process;
  p->next = NULL;
  p->ticks = 0;
  p->pid = 0;
  p->status = RUNNING;
  running = p;
  init_gic();
  init_timer0();
  enable_irq();
  printk("Starting llanox... kernel loaded OK\n\n");
  print_cpu_status();
  pthread_t thread1, thread2, thread3, thread4;
  pthread_create(&thread1, NULL, test, NULL);
  pthread_create(&thread2, NULL, test, NULL);
  pthread_create(&thread3, NULL, test, NULL);
  pthread_create(&thread4, NULL, test, NULL);

  for (;;)
    printk("Process %d\n", running->pid);
}
```

Código 9. Programa de prueba de multiprogramación [23].

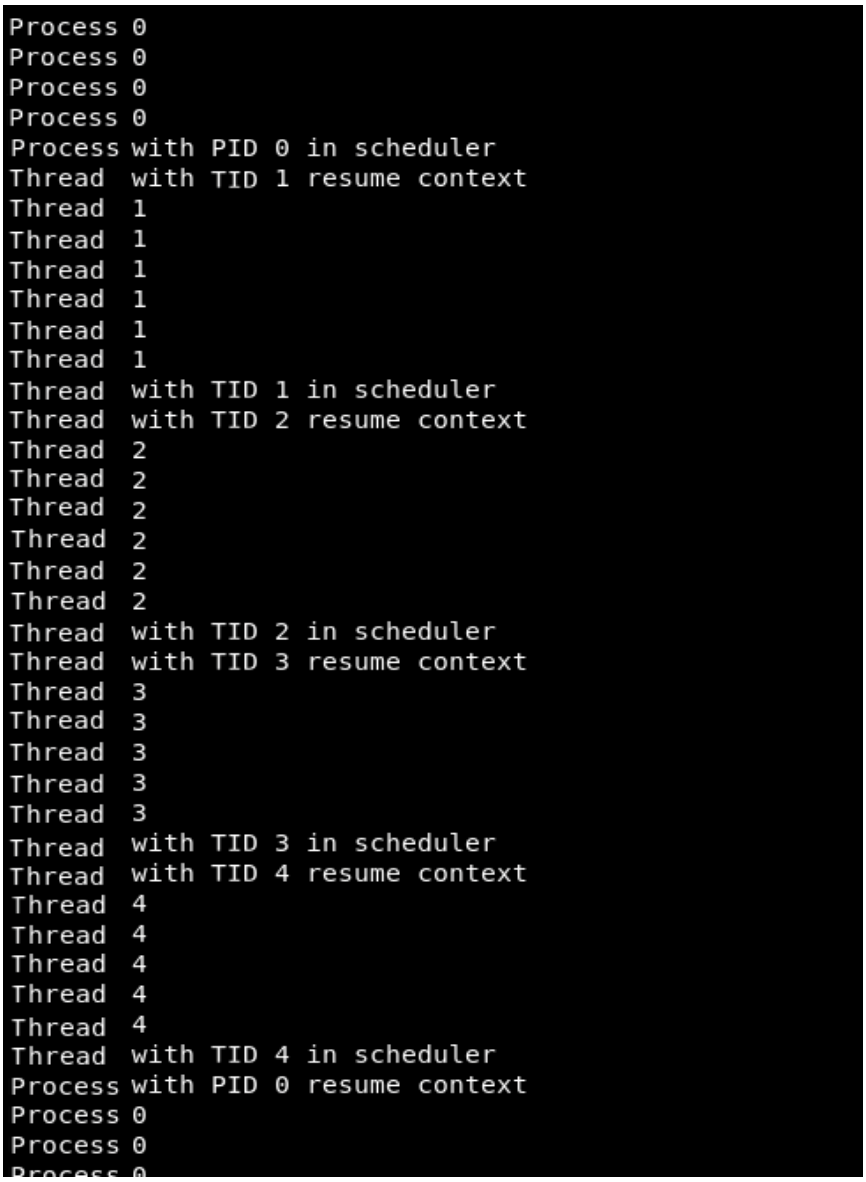

Figura 10. Demostración de multiprogramación.

La figura anterior representa la ejecución del código fuente mostrado anteriormente. En dicha figura se puede observar el proceso principal y cuatro threads ejecutándose de forma concurrente gracias a la implementación de la multiprogramación.

6.4. Proceso principal y thread usando join

```
Starting llanox... kernel loaded OK

---------------cpu status---------------

cpu mode: SVC mode
machine state: ARM state
FIQ: disabled
IRQ: enabled
ABORT: enabled
endianness: little endian
----------------------------------------

Process 0
Process 0
Process 0
Process 0
Process 0
Process 0 waiting thread 1
Process with PID 0 in scheduler
Thread  with TID 0 resume context
thread 1
thread 1
thread 1
thread 1
thread 1
thread 1
thread 1
thread 1
thread 1
thread 1
```

Figura 11. Sincronización entre el proceso principal y un thread.

6.5. Uso de los manejadores de dispositivos

Figura 12. Uso de display 16x2.

Uso de display mediante el manejador ofrecido en el entorno de ejecución. La comunicación entre el display y el SoC se hace mediante I2C.

7. Conclusiones

El resultado del trabajo ha sido muy satisfactorio. Se ha logrado cumplir con todos los objetivos propuestos al inicio de este. Se ha llevado a cabo todo de manera rigurosa, desde el diseño del entorno de ejecución hasta la implementación de este.

Cada bloque funcional que lo compone ha sido testeado hasta estar seguro de que cumple adecuadamente con su función y así poder ofrecerle al usuario un sistema con alta fidelidad. También se ha obtenido un sistema muy ligero como se describió en el objetivo. El entorno de ejecución tiene un peso de 13 KB, lo que lo hace perfecto para poder ser incrustado en muchos dispositivos que cuenten con pocos recursos.

El desarrollo de este trabajo no ha resultado nada fácil, ha requerido adquirir muchos conocimientos, incluso antes de poder hacer un simple diseño. Para llevarlo a cabo ha sido necesario estudiar rigurosamente la documentación sobre la arquitectura ARM Cortex-A, y así tener un dominio sobre el lenguaje ensamblador y en general, sobre el procesador utilizado para este proyecto. También ha sido necesario dominar el lenguaje de programación C, el toolchain utilizado y aprender sobre el funcionamiento y característica del SoC elegido.

Otras de las cosas fundamentales que ha sido necesario profundizar y quizás la más importante de todas es aprender sobre sistemas operativos, alcanzando de forma importante el conocimiento, sobre cómo están diseñados y la función de cada uno de los elementos que los componen. Así mismo, ha sido indispensable aprender sobre sistemas embebidos y sobre sus aplicaciones y necesidades. Todo ello ha sido la base para poder realizar este trabajo.

Además, desarrollar este proyecto, me ha permitido ampliar los conocimientos adquiridos en la carrera y en la rama de ingeniería de computadores, dada su relación. Me ha permitido reforzar conocimientos sobre arquitecturas de computadores, fundamentos y diseño de sistemas operativos, sistemas embebidos y sobre muchas cosas más de una forma menos directa.

Ha sido un desafío, pero gracias a ello, he alcanzado un grado de conocimiento que me será útil en el mundo laboral. Dichos conocimientos son las bases que demanda el mundo laboral para el desarrollo de aplicaciones embebidas.

A nivel personal me llevo la gran satisfacción de haber logrado lo que me he propuesto, de entender cómo funcionan internamente muchos de los dispositivos que nos rodean y de poder ofrecerle a otras personas un sistema para poder iniciarse en este bonito mundo, que puedan motivarse y aprender bastante de ello.

8. Trabajos futuros

Algo que resultaría interesante para este entorno de ejecución sería convertirlo en un sistema operativo de propósito general para sistemas embebidos.

Para lo cual habría que ampliar bastante el diseño e implementar muchas cosas más. Todo ello llevaría un buen tiempo de desarrollo, pero quedaría como resultado un sistema muy versátil y que ofrecería al usuario muchas características para poder desarrollar y ejecutar aplicaciones mucho más complejas.

Para lograr esto, sería necesario hacer uso de la MMU a fin de disponer de direccionamiento virtual en vez de uno físico, de esta forma cada proceso tendría su propio espacio de direccionamiento, se ofrecería protección de memoria y cada aplicación de usuario se ejecutaría en modo USER en vez de modo SVC. Ya que en un sistema operativo de uso general no interesa que los programas de usuario puedan ejecutar instrucciones privilegiadas del procesador.

En este entorno de ejecución eso no es realmente un problema, ya que está más destinado a un uso en especial y por tanto el programador hará y ejecutará sus aplicaciones embebidas de forma consciente para asegurar el correcto funcionamiento de todo. Y ya que se suelen trabajar con muchos dispositivos conviene tener una rápida comunicación con ellos, por eso tampoco se utiliza memoria virtual en este sistema.

Además de la memoria virtual, se dotaría al sistema de un sistema de ficheros. Así se podrá hacer uso de memorias secundarias y poder guardar de forma fácil cualquier documento y los ejecutables, ya que de momento el entorno de ejecución tiene cargado todos los ejecutables en memoria principal porque no cuenta con sistema de ficheros ni con un lector de ELF.

Por tanto, tiene todos los binarios de las aplicaciones embebidas en la misma imagen del kernel. Así que, el sistema de ficheros haría que la imagen fuera más ligera. Y otras de las características interesantes a poder implementar, sería hacer que el sistema sea multinúcleo y por tanto, pueda usar todas las CPU disponibles en el chip ARM.

Así se podrán aprovechar mejor los recursos del hardware, se podrán ejecutar procesos en paralelos y a su vez cada CPU ejecutaría procesos de forma concurrente. Por lo que se mejoraría bastante la eficiencia del sistema, se lograría una mayor velocidad y por tanto menor tiempo de respuesta.

También sería interesante aumentar el número de manejadores de dispositivos, para poder establecer comunicación con muchos dispositivos con un amplio rango de protocolos de comunicaciones. Y, dotar al sistema de acceso a internet.

Y por último y no por ello menos importante, tocaría implementar las llamadas al sistema.

Las llamadas al sistema son un mecanismo super importante y algo imprescindible en un sistema operativo, ya que permiten a los programas de usuario tener acceso a los periféricos y a ciertos recursos del sistema, a la vez que no se compromete la seguridad del sistema operativo, ya que se necesitan hacer llamadas al sistema cuando se necesitan ejecutar instrucciones

privilegiadas o acceder a un recurso de forma segura y esto, un programa de usuario no debe poder hacerlo, por eso son necesarias las llamadas al sistema.

Además, todo ello se haría siguiendo el estándar POSIX, para la compatibilidad con sistemas basados en Unix, ya que sería interesante que el sistema fuera capaz de ejecutar un programa compilado en Linux para ARM, por ejemplo. Así se lograría que fuera muy portable y más útil.

9. Bibliografía

[1] *Arm* [en línea]. arm.com [consulta: 17 de agosto de 2020]. Disponible en:

https://www.arm.com/products/silicon-ip-cpu

[2] *Wikipedia* [en línea]. wikipedia.org [consulta: 17 de agosto de 2020]. Disponible en:

https://es.wikipedia.org/wiki/FreeRTOS

[3] Cooling, Jim. *Real-time Operating Systems Book 1: The Theory*. Lindentree Associates, 2017. ISBN 978-1549608940.

[4] *Aicas* [en línea]. aicas.com [consulta: 17 de agosto de 2020]. Disponible en:

https://www.aicas.com/wp/products-services/jamaicavm/

[5] *Wikipedia* [en línea]. wikipedia.org [consulta: 17 de agosto de 2020]. Disponible en:

https://es.wikipedia.org/wiki/ECos

[6] *Wikipedia* [en línea]. wikipedia.org [consulta: 10 de agosto de 2020]. Disponible en:

https://es.wikipedia.org/wiki/Arquitectura_ARM

[7] *Arm* [en línea]. arm.com [consulta: junio de 2018]. Disponible en:

https://www.macs.hw.ac.uk/~hwloidl/Courses/F28HS/Docu/DEN0013D_cortex_a_series_PG.pdf

[8] *Arm* [en línea]. arm.com [consulta: junio de 2018]. Disponible en:

https://www.macs.hw.ac.uk/~hwloidl/Courses/F28HS/Docu/DEN0013D_cortex_a_series_PG.pdf

[9] *Linux-sunxi* [en línea]. Linux-sunxi.org [consulta: junio de 2018]. Disponible en:

http://dl.linux-sunxi.org/H3/Allwinner_H3_Datasheet_V1.0.pdf

[10] *Arm Developer* [en línea]. developer.arm.com [consulta: 10 de agosto de 2020]. Disponible en:

https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm

[11] *Wikipedia* [en línea]. wikipedia.org [consulta: 21 de marzo de 2019]. Disponible en:

https://es.wikipedia.org/wiki/Das_U-Boot

[12] *Man7* [en línea]. man7.org [consulta: 15 de julio de 2020]. Disponible en:

https://man7.org/linux/man-pages/man7/pthreads.7.html

[13] Bill, Gallmeister. *POSIX.4 Programmers Guide: Programming for the Real World.* O'Reilly Media, 1995. ISBN 978-1565920743.

[14] *Arm Developer* [en línea]. developer.arm.com [consulta: 25 de noviembre de 2019]. Disponible en:

https://developer.arm.com/documentation/ddi0471/b/

[15] *Arm Developer* [en línea]. developer.arm.com [consulta: 25 de noviembre de 2019]. Disponible en:

https://developer.arm.com/documentation/ddi0471/b/

[16] Love, Robert. *Linux Kernel Development*. 3ª edición. Addison-Wesley, 2010. ISBN 978-0-672-32946-3.

[17] *sourceware* [en línea]. sourceware.org [consulta: 20 de marzo de 2019]. Disponible en:

https://sourceware.org/binutils/docs/ld/Scripts.html

[18] *math.utah.edu* [en línea]. [consulta: 20 de marzo de 2019]. Disponible en:

https://www.math.utah.edu/docs/info/ld_3.html

[19] *Arm* [en línea]. arm.com [consulta: 23 de julio de 2020]. Disponible en:

https://www.macs.hw.ac.uk/~hwloidl/Courses/F28HS/Docu/DEN0013D_cortex_a_series_PG.pdf

[20] *ArmWiki* [en línea]. heyrick.eu [consulta: 7 de abril de 2019]. Disponible en:

https://heyrick.eu/aw/index.php?title=The_Status_register

[21] *Arm* [en línea]. arm.com [consulta: 23 de julio de 2020]. Disponible en:

https://www.macs.hw.ac.uk/~hwloidl/Courses/F28HS/Docu/DEN0013D_cortex_a_series_PG.pdf

[22] Love, Robert. *Linux Kernel Development*. 3ª edición. Addison-Wesley, 2010. ISBN 978-0-672-32946-3.

[23] Brandford, Nichols. *PThreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly Media, 1996. ISBN 978-1565921153.

10. Glosario

ARM: Advanced RISC Machine.

BIOS: Sistema básico de entrada-salida.

BSS: Block Started by Symbol.

Bootloader: gestor de arranque.

CAD: conversor analógico-digital.

CDA: conversor digital-analógico.

CISC: conjunto de instrucciones complejas.

CPU: unidad central de procesamiento.

EEPROM: ROM programable y borrable eléctricamente.

ELF: Executable and Linkable Format.

eMMC: MultimediaCard embebida.

FIFO: primero en entrar, primero en salir.

FIQ: fast interrupt request.

GIC: controlador genérico de interrupciones.

GNU: GNU (GNU's Not Unix) es un sistema operativo de software libre.

GPU: unidad de procesamiento gráfico.

I2C: circuito inter-integrado.

IoT: internet de las cosas.

IRQ: interrupt request.

ISR: rutina de servicio de interrupción.

MMU: la unidad de gestión de memoria es el hardware encargado de traducir direcciones lógicas a físicas y de ofrecer protección en la memoria.

PCB: bloque de control de proceso.

POSIX: POSIX (interfaz portátil de sistema operativo para Unix) es una norma definida por la IEEE, con el objetivo de definir una interfaz estándar del sistema operativo.

Quantum: máximo intervalo de tiempo que un proceso o thread ocupa la CPU.

RAM: memoria de acceso aleatorio.

RISC: conjunto de instrucciones reducido.

ROM: memoria de solo lectura.

Sistema embebido: a diferencia de un ordenador corriente, que es usado para propósito general, un sistema embebido es un sistema de cómputo utilizado para funciones específicas y pocas en general. Normalmente cuentan con todos los elementos hardware incrustados en una misma placa.

SoC: sistema en chip.

SPI: serial peripheral interface.

SVC: modo supervisor o modo kernel.

SYS: modo sistema.

TLB: buffer de traducción anticipada. Es una memoria caché utilizada por la MMU.

UART: transmisor-receptor asincrónico universal.