ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA

*Trabajo Fin de Grado*

# Evaluación de la plataforma Raspberry Pi para la docencia de Microprocesadores

## (Evaluation of Raspberry Pi as a Microprocessor Architecture teaching platform)

**Para acceder al Título de**

*Graduado en*
*Ingeniería de Tecnologías de Telecomunicación*

**Adrián Barredo Ferreira**

**Julio - 2015**

**GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN**

CALIFICACIÓN DEL TRABAJO FIN DE GRADO

**Realizado por: Adrián Barredo Ferreira**

**Director del TFG: María del Carmen Martínez Fernández**

**Subdirector del TFG: Esteban Stafford Fernández**

**Título:** "**Evaluación de la plataforma Raspberry Pi para la docencia de Microprocesadores**"

**Title: "Evaluation of Raspberry Pi as a Microprocessor Architecture teaching platform"**

**Presentado a exámen el día:**

para acceder al Título de

**GRADUADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN**

Composición del Tribunal:

Presidente (Apellidos, Nombre):

Secretario (Apellidos, Nombre):

Vocal (Apellidos, Nombre):

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente

Fdo.: El Secretario

Fdo.: El Vocal

Fdo.: El Director del TFG
(sólo si es distinto del Secretario)

Vº Bº del Subdirector

Trabajo Fin de Grado Nº
(a asignar por Secretaría)

# Agradecimientos

Me gustaría agradecer a mis padres, Ángel y Gemma todo el apoyo que he tenido por su parte y el esfuerzo que han realizado para poder formarme como persona y poder llegar a este punto de mi carrera pudiendo así concluir mis estudios universitarios. También me gustaría agradecer la colaboración que me han brindado mis compañeros y la gente más cercana. Además cómo no, de mis directores Carmen y Esteban, sin los cuales no hubiese podido concluir este proyecto. A todos.

Muchas gracias.

# Resumen

El estudio y la comprensión del funcionamiento de los procesadores es uno de los requisitos fundamentales en el proceso formativo de un ingeniero de telecomunicación. Se debe sobre todo, al amplio abanico de posibilidades que proporciona en diversos sectores del mercado actual, como en equipos de comunicaciones tales como routers, moduladores, ordenadores, smartphones y tablets. La necesidad tecnológica de dispositivos cada vez más pequeños y con características cada vez más punteras demanda a profesionales más especializados en su desarrollo.

A lo largo de la historia, han sido muchos los microprocesadores que han pretendido hacerse un hueco en el desarrollo tecnológico. Entre ellos, los denominados ARM han logrado mayor repercusión en el mercado, siendo en la actualidad la arquitectura de 32 bits más exitosa en el mundo respecto al nivel de producción.

Estos hechos han provocado que en la titulación se desee que el alumno adquiera los conocimientos base de la arquitectura ARM, necesitando un entorno capaz de ejecutar y depurar código ensamblador de este tipo de sistemas.

El objetivo de este proyecto es el de buscar una plataforma capaz de instruir al estudiante en la arquitectura ARM, proporcionándole nuevas herramientas para su estudio y así poder profundizar en los conocimientos adquiridos en asignaturas previas de la titulación de temática similar.

Así mismo, se pretende buscar una plataforma asequible que permita al alumno investigar por su cuenta el funcionamiento del procesador, consiguiendo dos fines: por un lado, que se motive al hacer de sus estudios una realidad, y por otro, que aprenda haciendo de la asignatura una forma de vida.

**Palabras clave:** ARM, arquitectura del procesador, microprocesador, estructura de computadores, Raspberry Pi.

# Índice

**Agradecimientos** **3**

**Resumen** **5**

**1 Introducción** **9**
*1.1 Situación actual del laboratorio* *9*
*1.2 Elección del procesador* *10*
*1.3 Elección de la plataforma a implementar* *10*
*1.4 Características generales de la Raspberry Pi* *12*
*1.5 Métodos para abordar el estudio del procesador* *12*

**2 Arquitectura ARM** **15**
*2.1 Introducción* *15*
*2.2 Registros* *15*
2.2.1 Registros de estado 16
*2.3 Modos de ejecución* *17*
*2.4 Excepciones* *18*
*2.5 Repertorio de instrucciones* *19*
*2.6 Coprocesadores* *20*
*2.7 Procedimiento estándar de llamada a subrutinas* *21*
2.7.1 Registros y su función 21
2.7.2 Funcionamiento 22
*2.8 Segmentación* *22*
*2.9 Subconjunto de instrucciones Thumb* *24*
*2.10 Extensión Jazelle* *25*

**3 Elementos físicos de la *Raspberry Pi*** **27**
*3.1 Reloj del sistema* *27*
*3.2 General Purpose Input Output* *27*
3.2.1 El controlador de la GPIO 28
*3.3 Interrupciones* *29*

**4 Programación en RISC OS** **33**
*4.1 Introducción* *33*
*4.2 Programación empleando BASIC* *33*
4.2.1 Diseño de un programa ensamblador en BASIC 33
4.2.2 Ejecución del programa ensamblador 34
*4.3 Programación mediante GNU Compiler Collection* *34*
*4.4 Depuración* *35*
4.4.1 SWI 256 35
4.4.2 Breakpoint 35
4.4.3 Manipulación de memoria 36
4.4.4 Disassembler 36
*4.5 Software vectors* *36*
4.5.1 Claiming vectors 36
*4.6 Software Interrupts* *37*
4.6.1 Identificación 37
4.6.2 Proceso de ejecución 37
4.6.3 Tipos de Software Interrupt 38
4.6.4 Comandos SYS 38
*4.7 Módulos* *38*
4.7.1 Modo de empleo 39

4.7.2 Instancia 39
4.7.3 Tratamiento de errores 39
4.7.4 Formato de un módulo 39
*4.8 Capa de abstracción hardware* *41*
4.8.1 Lista de HAL APIs 42
*4.9 Desarrollo de un driver para el control del reloj del sistema* *42*
4.9.1 Modificación de la frecuencia de trabajo del sistema 42
4.9.2 Desarrollo del módulo 42

**5 Bare Metal** **47**
*5.1 Introducción* *47*
*5.2 Compilación* *47*
5.2.1 Elección de Sistema Operativo 47
5.2.2 Desarrollo de un archivo *Makefile* 47
5.2.3 Archivos requeridos durante el proceso 49
*5.3 Proceso de arranque* *50*
5.3.1 Archivos necesarios 51
*5.4 Modificación de los vectores de excepción* *51*
*5.5 Desarrollo de un contador* *52*

**6 Conclusión** **53**

**Índice de figuras** **55**

**Índice de tablas** **57**

**Bibliografía** **59**

**7 Apéndice** **61**
*7.1 Instrucciones ARM* *61*
*7.2 Raspberry Pi* *65*
*7.3 RISC OS* *68*
7.3.1 Instalación 68
7.3.2 Instalación y compilación mediante GCC 70
7.3.3 Software Interrupts (SWI) 72

# 1 INTRODUCCIÓN

La Ingeniería de Computadores estudia el desarrollo de sistemas automatizados y se enfoca en el análisis, diseño y utilización del hardware y software para la implementación de aplicaciones industriales y telemáticas.

El Grupo de Arquitectura y Tecnología de Computadores de la Universidad de Cantabria cuenta con un laboratorio para sus clases. Está basado en el procesador MIPS y en él los alumnos pueden experimentar el funcionamiento de este tipo de microprocesadores. Sin embargo, al ser un dispositivo caro y difícil de manipular, los estudiantes carecen de la posibilidad de ensayar con él en casa y de explotar todas sus virtudes.

Por otro lado, el avance tecnológico ha hecho mella en este microprocesador, que está quedando obsoleto. La plataforma MIPS fue implementada en el laboratorio en el año 2003 y desde entonces han sido muchos los procesadores que lo han sustituido en el mercado y con mejores prestaciones.

A continuación se va a proceder a argumentar qué tecnología y dispositivos de la actualidad podrían ser los sustitutos y qué estrategias podrían seguirse para abordar su estudio y comprensión por parte de los alumnos en un futuro laboratorio.

## 1.1 *Situación actual del laboratorio*

El laboratorio docente del Grupo de Arquitectura y Tecnología de Computadores actual cuenta con el procesador MIPS y fue el resultado de varios proyectos fin de carrera. Está constituido por placas *Mycable XXS1500* cuyo aspecto y características se muestran resumidas en la Tabla 1:

| | |
|---|---|
| | |
| **CPU** | Au1500 @400 MHz |
| **Memoria RAM** | 64 MB |
| **Memoria Flash** | 16 MB |
| **Conexión de red** | Doble entrada Ethernet 10/100 |
| **Conexión serie** | Doble Puerto RS232 |
| **Precio** | Hasta los 500€ |

Tabla 1 - Características *Mycable XXS1500*

Para su programación, se realiza la comunicación mediante el puerto serie y se envian los códigos a través de un programa llamado *ZOC*. Mediante unas líneas de código C es posible hacer que la placa reciba el código ensamblador que se manda a través de la conexión serie o RS-232. También es posible la interconexión mediante el bus *PCI* aunque la placa no tiene disponibles todas sus líneas y señales. El hecho de incluirlas tiene un sobrecoste de 2500€.

Su precio y soporte no lo hacían muy interesante para el uso individual de cada alumno, quedando restringido a las horas de laboratorio.

## 1.2 *Elección del procesador*

En la actualidad, existen dos tipos de arquitecturas que pretenden hacerse con el dominio del mercado. Se trata de *x86* y *ARM*.

- x86 es el tipo de procesador ó arquitectura más utilizado en equipos de escritorio. Está basado en la arquitectura CISC *(Complex Instruction Set Computer)* con soporte para instrucciones complejas, paralelismo a nivel de instrucciones y de ejecución más lenta. Consigue un alto rendimiento, que viene acompañado de un mayor consumo de energía y de la necesidad de más espacio físico.
- ARM es de tipo RISC *(Reduced Instruction Set Computer)*, cuyas propiedades son que poseen instrucciones de tamaño fijo con pocos formatos y que sólo las instrucciones *load/store* acceden a la memoria de datos. El objetivo de diseñar máquinas con esta arquitectura es facilitar el paralelismo en la ejecución de instrucciones y permitir realizar tareas menores con procesos más cortos. Esto conlleva una disminución de la energía empleada.

Intel y AMD son los grandes fabricantes que han hecho evolucionar la arquitectura x86. Sin embargo, ARM se ha expandido gracias a fabricantes como Qualcomm, Texas Instruments, Apple, Samsung, etc. Todo ello gracias a ARM Holdings, que decidió licenciar su arquitectura haciéndola accesible.

En este proyecto fue elegido ARM debido a su trayectoria y a las estimaciones de cuota de mercado para dentro de unos años. Su disponibilidad va a ser tan alta que va a poder ser localizado en prácticamente cualquier tipo de dispositivo.

## 1.3 *Elección de la plataforma a implementar*

A día de hoy, dos dispositivos son frecuentemente utilizados para iniciarse en la programación y ambos contienen un procesador ARM. Se trata de *Arduino* y *Raspberry Pi.* Los dos tienen en común el coste, tamaño y la posibilidad de ser ampliados mediante hardware diseñado por el usuario, pero sin embargo son muy diferentes.

Primeramente, *Raspberry Pi* es una computadora completamente funcional, mientras que *Arduino* es un microcontrolador. Aunque *Arduino* puede ser programado en C con pequeñas aplicaciones, este no puede ejecutar todo un sistema operativo. La *Raspberry Pi* es un computador independiente que puede ejecutar un sistema operativo actual como Linux. La *Raspberry Pi* es 9 veces más rápida que un *Arduino* cuando se trata de velocidad de reloj y presenta 5.500 veces más memoria RAM. Puede realizar varias tareas, soportar dos puertos USB y conectarse de forma inalámbrica a Internet.

Puede parecer que *Raspberry Pi* es superior a *Arduino*, pero eso es sólo cuando se trata de aplicaciones de software. La simplicidad de Arduino hace que éste sea una apuesta mucho mejor para proyectos de hardware.

En la práctica, se considera que los dos dispositivos presentan funciones complementarias. Existen programas que permiten la comunicación entre ambos. En general, *Arduino* es mucho más simple, más difícil de dañar y cuenta con más recursos de aprendizaje para los principiantes.

Toda la información sobre *Arduino* se puede encontrar en su página oficial [19]. Las noticias de los dos dispositivos se recopilan diariamente en [14]. La comparativa entre *Arduino* y *Raspberry Pi* [15].

Sus características se recogen en la Tabla 2:

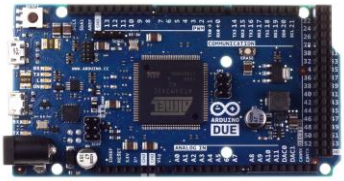
| | **Arduino Due** | **Raspberry Pi B** |
|---|---|---|
| |  | |
| **Tamaño** | 10.4 x 5.3 x 1.5 cm | 8.6 x 5.4 x 1.7 cm |
| **Memoria RAM** | 96KB | 512 MB |
| **Velocidad de reloj** | 84 MHz | 700 MHz |
| **On Board Network** | Ninguna | 10/100 wired Ethernet RJ45 |
| **Multitarea** | No | Sí |
| **Voltaje de entrada** | 7 -12 V | 5 V |
| **Memoria Flash** | 512KB | Tarjeta SD (2 a 16G) |
| **Puertos USB** | Uno | Dos |
| **Sistema Operativo** | Ninguno | Distribuciones de Linux |
| **Entorno de desarrollo integrado (IDE)** | Arduino | Scratch, IDLE, cualquiera con soporte Linux |
| **Precio** | 40 € | 35 € |

Tabla 2 - Comparación *Arduino* y *Raspberry Pi*

En este documento, dado que el objetivo es buscar nuevos métodos para el estudio de los procesadores ARM y futuros proyectos software a desarrollar en ellos, se decidió seleccionar la *Raspberry Pi* como plataforma debido a su versatilidad en este aspecto.

Existen multitud de fabricantes que han desarrollado hardware específico para este dispositivo. Por menos de 50€ es posible hacerse con un *kit* con material diverso con el que empezar a realizar proyectos, como el que se muestra en la Figura 1, disponible en [26].

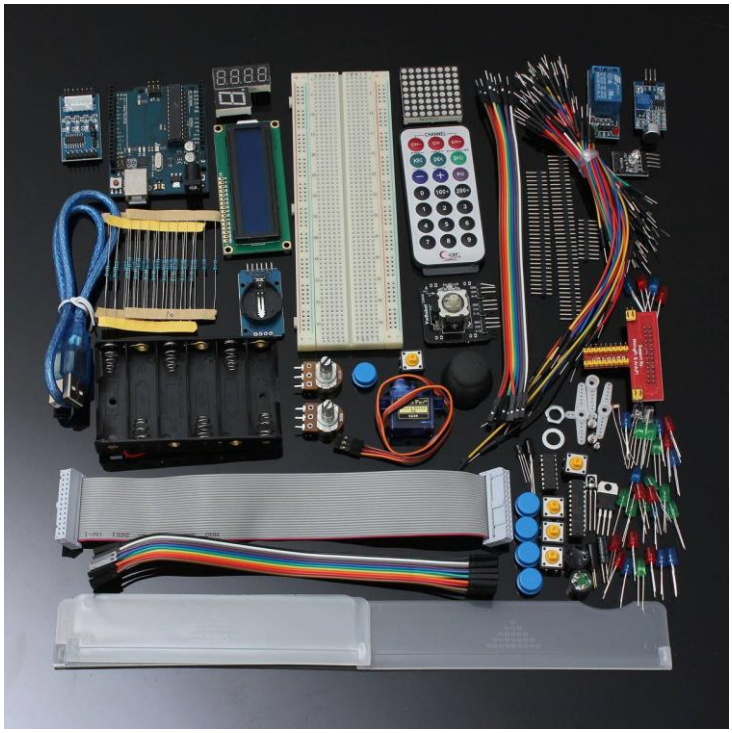


Figura 1 - Ejemplo de *kit* de inicio de la *Raspberry Pi*

## 1.4 *Características generales de la Raspberry Pi*

En 2011, *Raspberry Pi Foundation* desarrolló el dispositivo *Raspberry Pi* como como ordenador de bajo coste para facilitar la enseñanza de la informática en los colegios. Los ordenadores de la actualidad están orientados a tareas informáticas o de ocio, pero no vienen preparados con conexiones que posibiliten pequeños proyectos de hardware. Este es el vacío que cubre la *Raspberry Pi.*

Existen diferentes modelos, que se recogen en la Tabla 3:

| | **Modelo A** | **Modelo A+** | **Modelo B** | **Modelo B+** | **Modelo B** |
|---|---|---|---|---|---|
| **SoC** | Broadcom BCM2835 | Broadcom BCM2835 | Broadcom BCM2835 | Broadcom BCM2835 | Broadcom BCM2836 |
| **CPU** | ARM11 ARMv6 @700 MHz | ARM11 ARMv6 @700 MHz | ARM11 ARMv6 @700 MHz | ARM11 ARMv6 @700 MHz | ARM11 ARMv7 ARM Cortex-A7 4 núcleos @900 MHz |
| **GPU** | Broadcom VideoCore IV @250 MHz OpenGL ES 2.0 | Broadcom VideoCore IV @250 MHz OpenGL ES 2.0 | Broadcom VideoCore IV @250 MHz OpenGL ES 2.0 | Broadcom VideoCore IV @250 MHz OpenGL ES 2.0 | Broadcom VideoCore IV @250 MHz OpenGL ES 2.0 |
| **RAM** | 256 MB LPDDR SDRAM @400 MHz | 256 MB LPDDR SDRAM @400 MHz | 512 MB LPDDR SDRAM @400 MHz | 512 MB LPDDR SDRAM @400 MHz | 1 GB LPDDR2 SDRAM @450 MHz |
| **USB 2.0** | 1 | 1 | 2 | 4 | 4 |
| **Salidas de vídeo** | HDMI 1.4 1920x1200 píxeles | HDMI 1.4 1920x1200 píxeles | HDMI 1.4 1920x1200 píxeles | HDMI 1.4 1920x1200 píxeles | HDMI 1.4 1920x1200 píxeles |
| **Almacenamiento** | SD/MMC | microSD | SD/MMC | microSD | microSD |
| **Ethernet** | No | No | Sí, 10/100 Mbps | Sí, 10/100 Mbps | Sí, 10/100 Mbps |
| **Tamaño** | 85,60x56,5 mm | 65x56,5 mm | 85,60x56,5 mm | 85,60x56,5 mm | 85,60x56,5 mm |
| **Peso** | 45 gramos | 23 gramos | 45 gramos | 45 gramos | 45 gramos |
| **Precio** | 25 dólares | 20 dólares | 35 dólares | 35 dólares | 35 dólares |

Tabla 3 - Características generales de los diferentes modelos de *Raspberry Pi*

Existen multitud de páginas web que ofrecen tutoriales y material para que el usuario desarrolle prácticamente cualquier tipo de proyecto.

## 1.5 *Métodos para abordar el estudio del procesador*

La *Raspberry Pi* cuenta con múltiples sistemas operativos que permiten explotar al máximo sus virtudes. Los sistemas operativos disponibles sobre *Raspberry Pi* se encuentran en la página oficial [7]. Todos ellos se implementan a través de la tarjeta SD. Entre ellos destacan *Raspbian* (que deriva de *Debian*), *Arch Linux, RISC OS* y se está trabajando la posibilidad de utilizar *Android*. La mayoría son distribuciones Linux y su administración no es muy diferente al de dicho sistema. Sin embargo, suelen ser bastante pesados en cuanto a tamaño y la *Raspberry Pi* puede no tener el rendimiento esperado. En este aspecto RISC OS destaca, ya que es un sistema operativo sencillo que fue diseñado por los creadores de la arquitectura ARM. Por esta razón, en este proyecto fue elegido para programar, depurar y estudiar el comportamiento de este procesador.

Por otro lado, dado que el alumno puede desear el no depender de ningún sistema operativo para interactuar con el dispositivo, se propone un nuevo método con el que programar sobre el hardware directamente. Este tipo de programación recibe el nombre de *Bare Metal.*

A lo largo de este proyecto se explican las dos estrategias de análisis y programación del microprocesador ARM mencionados previamente, así como un ejemplo práctico de cada uno. El objetivo final es proporcionar una visión amplia de la plataforma y explicar de forma razonada cuál puede ser la utilidad práctica de los dos entornos de estudio.

Para alcanzar dichos objetivos, este documento se encuentra dividido en tres capítulos principales. El Capítulo 2 recoge los aspectos principales de la arquitectura ARM, con el fin de proporcionar al lector una visión general de su funcionamiento. El Capítulo 3 explica las características físicas y el funcionamiento de los periféricos que van a ser utilizados en los ejercicios prácticos de los Capítulos 4 y 5. El Capítulo 4 recoge métodos para la programación y depuración de programas en RISC OS así como características de su funcionamiento que fueron trascendentes para el desarrollo del *driver* del reloj del sistema. El Capítulo 5 explica otro método de programación de la *Raspberry Pi*, denominado *Bare Metal*, y describe el proceso que se siguió para desarrollar un contador con periféricos de entrada y salida. Finalmente, en la Conclusión se exponen las ventajas de programar utilizando la *Raspberry Pi* y sus posibles aplicaciones en la docencia.

# 2 ARQUITECTURA ARM

Este capítulo tiene como objetivo introducir al lector en la lógica de la arquitectura ARM. Se procederá a describir de forma general el repertorio de instrucciones (ISA), los modos de excepción y diversos aspectos referentes al funcionamiento interno del procesador, con el fin de facilitar la comprensión de los capítulos sucesivos. La información de este capítulo puede encontrarse en [1], [2] y [6].

## 2.1 *Introducción*

ARM es una arquitectura de procesador de tipo **RISC** *(Reduced Instruction Set Computer)* que presenta las siguientes características:

- Respecto a sus instrucciones *(Instruction Set Architecture):*
  - Son de 32 bits de longitud.
  - La mayoría pueden ejecutarse de forma condicional.
  - La mayoría puede producir un desplazamiento en uno de los operandos antes de realizar la operación.
  - Tienen dos o más operandos.
- El acceso a memoria se realiza mediante instrucciones load / store. Sólo las instrucciones "load" y "store" son capaces de acceder a memoria, el resto actúa sobre los registros:
  - Tiene diferentes modos de direccionamiento.
  - Sus instrucciones son flexibles dependiendo del tipo de dato o de la cantidad de información.
- Aumento del repertorio de instrucciones vía coprocesadores, como el coprocesador 15.
- Repertorio de las instrucciones más utilizadas comprimido en 16 bits, formando el repertorio de instrucciones *Thumb*.
- Registros específicos para controlar el estado del programa, llamados registros de estado.

La arquitectura ARM presenta características de la arquitectura Von Neumann dado que los datos y las instrucciones a ejecutar residen en el mismo espacio de memoria pero los buses para acceder a ellos son independientes, tal y como ocurre en la arquitectura Harvard. En consecuencia se constituye una arquitectura Harvard modificada.

## 2.2 *Registros*

ARM posee 31 registros de propósito general de 32 bits. De ellos sólo 16 son visibles y algunos tienen funciones especiales. Los otros se emplean para aumentar la velocidad del procesado de excepciones. Las instrucciones ARM sólo pueden dirigirse a estos 16 registros visibles y éstos son usados en el modo sin privilegios (modo usuario).

El modo *usuario* es diferente respecto a los otros modos (los privilegiados) lo que significa que sólo puede pasar a otro modo generando una excepción (con la instrucción SWI para pasar a modo *supervisor*). También quiere decir que los sistemas de memoria y los coprocesadores podrían permitirle un menor acceso funcional que en el caso de que fueran privilegiados.

De estos 16 registros visibles tres tienen funciones especiales y otros dos dependiendo de la implementación:

- Stack Pointer (SP): es el registro 13. Se usa para acceder a la pila.

- Link Register (LR): es el registro 14. Almacena la dirección de la siguiente instrucción después de un Branch and Link (BL o BLX), que se usa para llamar a una subrutina. También almacena la dirección de retorno en los modos de excepción. En otros casos también se puede usar como registro de propósito general.
- Program Counter (PC): es el registro 15. Almacena la dirección de la siguiente instrucción a ejecutar. Admite operaciones aritmético/lógicas en las cuales se le utilice como registro de destino. Esto permite una implementación fácil de gran variedad de instrucciones de tipo *branch*.

El registro 12 "Intra-Procedure call" (IP) se suele emplear como registro de enlace hacia subrutinas cuando se encuentran fuera del alcance de los "branch". El registro 11 "Frame Pointer" (FP) se suele emplear como auxiliar al 13 SP para fijar determinadas posiciones en la pila. Los otros 11 registros no tienen ningún objetivo hardware concreto.

### 2.2.1 Registros de estado

Existen 6 registros diferentes aparte de los 31 de propósito general y se denominan registros de estado. Contienen información acerca del estado del modo en el que se encuentre el procesador.

El registro de estado que contiene información sobre el estado actual es el *Current Program Status Register* (CPSR). El CPSR, cuya estructura se puede observar en la Figura 2, contiene:

- Cuatro bits o banderas (del inglés *flags*) de condición (*Negative*, *Zero*, *Carry* y *oVerflow*).
- Cuatro flags GE (*Greater than or Equal*) que codifican las siguientes condiciones según operación.
  - Si los resultados de las operaciones con signo han sido no negativas.
  - Si las operaciones sin signo han producido acarreos.
- Dos bits (I, F) para deshabilitar interrupciones, una para las normales (I) y otra para las rápidas (F).
- Un bit (A) para deshabilitar las excepciones por aborto.
- Cinco bits que codifican el modo actual del procesador. Por ejemplo, "10000" es el código del modo usuario.
- Dos bits (J, T) que indican si se están ejecutando *Jazelle opcodes* ó las instrucciones ARM o *Thumb*, que se explican en las Secciones 2.9 y 2.10
- Un bit que controla el formato de almacenamiento de las operaciones load/store (Big Endian o Little Endian).

Cuando se produce una excepción, el CPSR (que contiene el estado del modo anterior) es almacenado en el SPSR del nuevo modo, como se explica en la Sección 2.4.2.

Sólo se puede acceder al CPSR y al SPSR mediante instrucciones específicas, como la instrucción "MRS" que permite modificar los bits del CPSR.

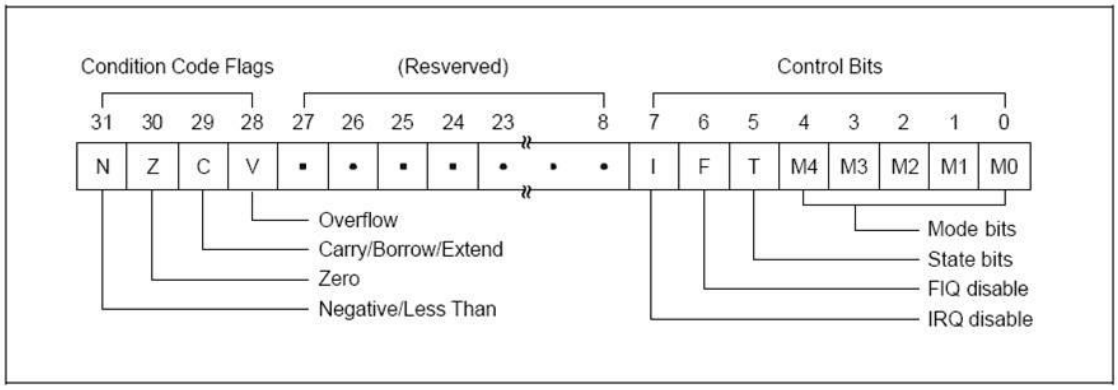



Figura 2 - Estructura del CPSR a nivel de bit [2]

## 2.3 *Modos de ejecución*

La arquitectura ARM soporta 7 modos de ejecución diferentes, de los cuales 5 se consideran modos de excepción, un sexto privilegiado y el último como normal. Los siete tipos de modos son:

- **Modo Usuario**

Es un modo no privilegiado al cual accede el usuario en un principio. El programa que está siendo ejecutado no puede acceder a ciertas partes del sistema, o cambiar de modo, si no es a través de una excepción.

Los sistemas de memoria y los coprocesadores podrían permitirle un menor acceso funcional que en el caso de que fuera un modo privilegiado.

En esta situación sólo se pueden emplear los 16 registros de propósito general.

- **Modo FIQ**

El procesador ARM soporta dos tipos de interrupciones: las normales (regular) o las rápidas (fast). La diferencia es que las rápidas pueden interrumpir a las normales.

El modo *FIQ* provee de nuevos registros (r8-r14_fiq, SPSR) y es útil para situaciones de tiempo real en las que es posible una pérdida de datos o necesaria una extrema rapidez.

- **Modo IRQ**

Se trata del modo de interrupción normal.

Es utilizado para interrupciones que no requieren una gran rapidez (teclado, pantalla…) y provee nuevos registros (r13_irq, r14_irq, SPSR).

- **Modo Supervisor**

Es un modo privilegiado reservado para el sistema operativo.

Es accedido en el momento de producirse una llamada al SO (mediante una instrucción SWI) o un reset.

Provee los registros r13_svc, r14_svc y SPSR.

- **Modo Abort**

Es un modo privilegiado al que se accede en el caso de un fallo al cargar una instrucción *(Prefetch Abort)* o datos de memoria *(Data abort).*

*Prefetch Abort* se produce como consecuencia de un *breakpoint* (instrucción BKPT) y *Data Abort* ocurre cuando el procesador no tiene permisos para cargar el dato de memoria.

Provee los registros r13_abt, r14_abt y SPSR.

- **Modo Undefined**

Es un modo privilegiado que se produce en el momento en el que se ejecuta una instrucción y ningún coprocesador responde, o cuando el procesador no sabe cómo actuar con ella.

Provee los registros r13_und, r14_und y SPSR.

- **Modo System**

Se trata de un modo privilegiado, aunque no de excepción.

Posee exactamente los mismos registros que el modo usuario, y en él no se produce ningún tipo de excepciones.

No está sujeto a las restricciones del modo usuario y se utiliza para tareas que requieren permisos de sistema.

Cuando ocurre una excepción, los registros estándar son sustituidos por registros específicos de excepciones. También se incluyen en estos casos el 13 "Stack Pointer" y 14 "Link Register". El 14 "Link Register" porque almacena la dirección de retorno de la excepción y el 13 "Stack Pointer" para proporcionarle al manejador de excepciones acceso a la pila[1].

El modo *FIQ* también emplea los registros R8-R12 para que el procesado de las interrupciones pueda comenzar ó finalizar sin la necesidad de almacenar ó restaurar estos registros.

El modo *System* se considera privilegiado (aunque no de excepción) y se emplea para realizar tareas que necesitan privilegios para acceder a memoria o a los coprocesadores. De esta forma se evitan las restricciones impuestas en el modo *usuario*.

## 2.4 *Excepciones*

Cuando se produce una excepción, por ejemplo, se ha producido un error en la ejecución de un programa, se ha ejecutado un *breakpoint* o se ha producido una interrupción (entre otros) el procesador ARM para su ejecución en la manera preestablecida y comienza a ejecutar en una de las direcciones de memoria conocidas como los vectores de excepción. Existe una localización independiente para cada tipo de excepción.

El PC se desplaza al vector de excepción correspondiente, que contiene una instrucción "LDR PC, address", donde address es la dirección del manejador de excepciones correspondiente. La razón es que con un "branch" los saltos son de 32MB máximo y con "load" se permiten cualquier dirección.

La Tabla 4 refleja algunas de las características generales de las diferentes excepciones, como por ejemplo su posición en memoria y la prioridad entre ellas.

- El orden de la prioridad está descrito de mayor a menor, siendo el *Reset* la excepción más prioritaria y el *SWI* la menor.
- Tanto en el *Prefetch abort* como en el *Data abort* están incluidos los abortos producidos por el TLB.
- El modo *FIQ* se encuentra en la dirección más baja para evitar el salto a la dirección de su manejador de excepciones (que se encuentra en la dirección posterior), pudiendo tratar la excepción en un menor tiempo y evitando una pérdida de datos.

| Excepción | Modo | Prioridad | Dirección del manejador de excepciones |
|---|---|---|---|
| Reset | Supervisor (svc) | 1 | 0x00000000 |
| Undefined Instruction | Undefined (und) | 7 | 0x00000004 |
| Software Interrupt (SWI) | Supervisor (svc) | 8 | 0x00000008 |
| Prefetch Abort | Abort (abt) | 6 | 0x0000000C |
| Data Abort | Abort (abt) | 2 | 0x00000010 |
| Reserved | Reserved | 5 | 0x00000014 |
| IRQ | IRQ (irq) | 4 | 0x00000018 |
| FIQ | FIQ (fiq) | 3 | 0x0000001C |

Tabla 4 - Tipos de excepciones

[1] Cada modo de ejecución dispone de una región específica de la pila para su propio funcionamiento.

Los pasos para iniciar un tratamiento de excepción son los siguientes:

- Entrada al manejador de excepciones.
    - Se guarda la dirección en la que se le produjo la excepción, en el "Link Register" (r12).
    - Se hace una copia del CPSR en el SPSR del nuevo modo.
    - Se cambia el modo y el repertorio de instrucciones (a ARM) modificándolo en el CPSR. Esto es debido a que todas las excepciones se manejan con el repertorio de instrucciones ARM, nunca con el *Thumb*.
    - Se deshabilita las flags de interrupción correspondientes, dependiendo de la prioridad de la excepción.
    - Se dirige el PC a la dirección de memoria indicada en el "vector de excepciones" correspondiente.
- Salida del manejador de excepciones.
    - Se copia el "Link Register" (r12) en el PC (r15).
    - Habrá que tener en cuenta el offset correspondiente, en el caso de que se haya producido una interrupción.
    - Se copia el SPSR en el CPSR, lo que restaura los cambios de modo, repertorio de instrucciones y banderas de interrupción.

## 2.5 *Repertorio de instrucciones*

El ISA *(Instruction Set Architecture)* de la arquitectura ARM se pueden dividir en seis clases:

- Instrucciones de salto o *branch*.
- Instrucciones para el procesado de datos.
- Instrucciones para la transferencia de datos desde/a los registros de estado.
- Instrucciones de acceso a memoria.
- Instrucciones para el coprocesador.
- Instrucciones para la generación de excepciones.

La mayoría de las instrucciones de procesado de datos y un tipo de las del coprocesador pueden actualizar las cuatro banderas de condición del CPSR (*Negative, Zero, Carry* y *oVerflow*) dependiendo del resultado.

La mayoría de las instrucciones ARM contienen un campo de cuatro bits de condición. Una de sus codificaciones indica que la instrucción se ejecuta de forma incondicional. Si la condición se cumple (es cierta) cuando la instrucción comienza a ejecutarse, se ejecuta de la forma normal. Si no, no se ejecuta. Existen catorce tipos de condición.

El formato de las instrucciones ARM se resume en la Figura 3 [1]. En el anexo de la memoria se encuentra una explicación del funcionamiento de las instrucciones más comunes.

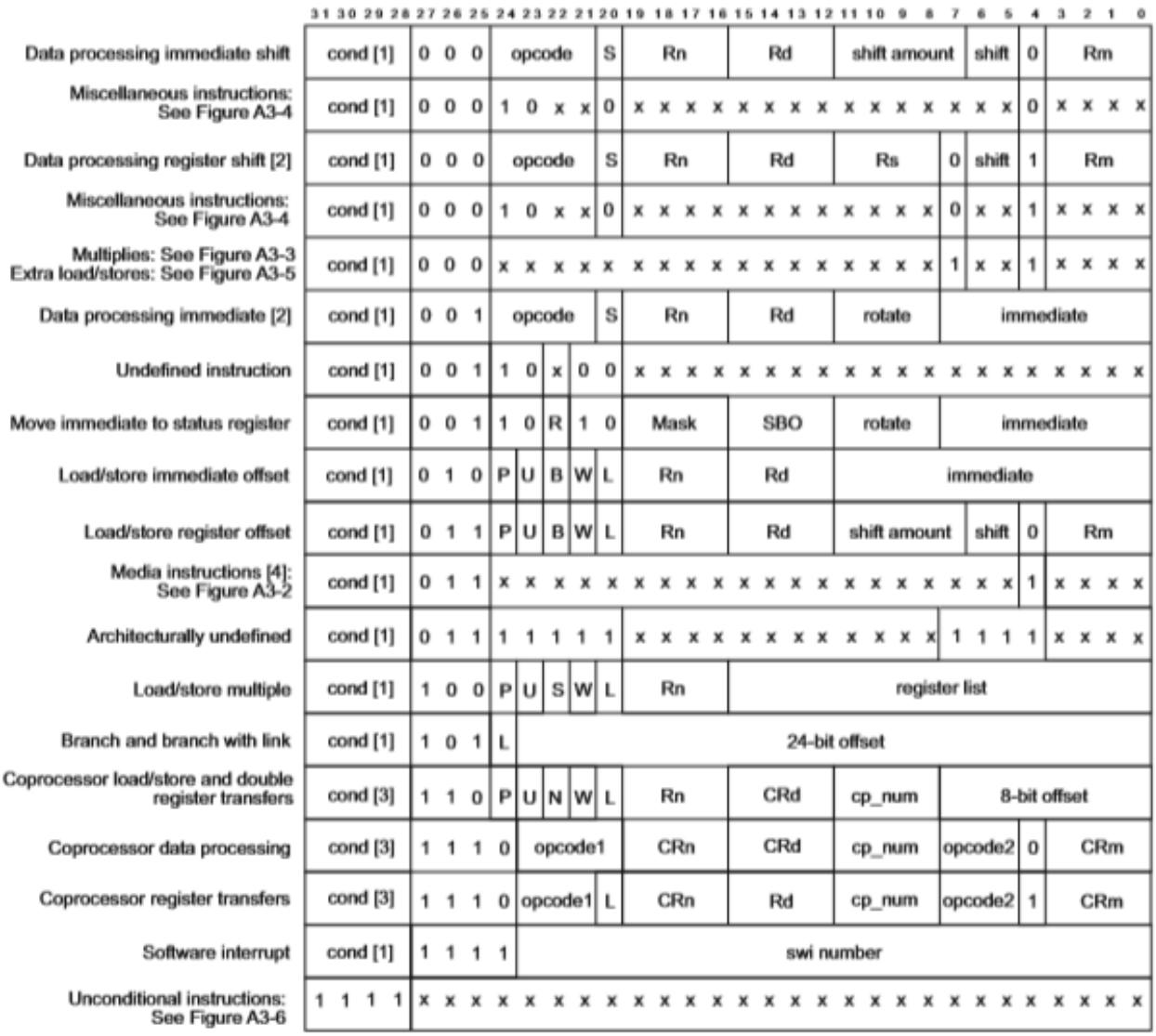



Figura 3 - Formato de las instrucciones ARM

## 2.6 *Coprocesadores*

El procesador ARM puede manejar hasta 16 coprocesadores, a continuación se procede a explicar uno de los más comunes, el número quince.

Aunque ARM no tiene por qué estar vinculado a otros procesadores, el coprocesador CP15 se ha convertido en un estándar desde las primeras versiones de ARM. Se encarga del control de la memoria caché, de los códigos de identificación del sistema y de la gestión de los accesos a la memoria.

Está constituido por 16 registros de 32 bits.

Se utiliza para configurar y controlar las cachés, las *Tightly-Coupled Memories* (TCMs), la *Memory Management Unit* (MMU) y otras opciones de sistema.

Uno de sus registros más interesantes es el llamado "coprocessor access register" encargado de administrar permisos a cada coprocesador sobre el acceso a memoria, con o sin privilegios.

## 2.7 *Procedimiento estándar de llamada a subrutinas*

Este apartado pretende explicar los usos de los diferentes registros de propósito general y el procedimiento a seguir para tratar la llamada y la salida de las subrutinas.

### 2.7.1 Registros y su función

La Tabla 5 recoge la función y el modo de empleo de los 16 registros de propósito general, empleados en el modo *usuario*. En ellos se puede observar que algunos tienen usos especiales, tales como ser utilizados para el envío de argumentos a subrutinas, retornar valores de las mismas o almacenar parámetros a corto o a largo plazo. Los últimos cinco registros fueron explicados en la Sección 2.2.

| **Registro** | **Denominación** | **Función** |
|---|---|---|
| r0 | | Argumento / Resultado / Registro temporal |
| r1 | | Argumento / Resultado / Registro temporal |
| r2 | | Argumento / Registro temporal |
| r3 | | Argumento / Registro temporal |
| r4 | | Registro para variables de larga duración |
| r5 | | Registro para variables de larga duración |
| r6 | | Registro para variables de larga duración |
| r7 | | Registro para variables de larga duración |
| r8 | | Registro para variables de larga duración |
| r9 | | Registro para variables de larga duración.<br>Puede tener otros usos dependiendo de la plataforma. |
| r10 | | Registro para variables de larga duración |
| r11 | FP | Frame Pointer / Registro temporal |
| r12 | IP | Intra Procedure call register / Registro temporal |
| r13 | SP | Stack Pointer |
| r14 | LR | Link Register |
| r15 | PC | Program counter |

Tabla 5 - Registros en ARM y su función

### 2.7.2 Funcionamiento

La Tabla 6 resume los pasos a seguir en caso de llamada a subrutina. Es un convenio entre programadores de para que las subrutinas sean independientes del programador y puedan emplearse en diferentes contextos.

| **Actor** | **Acción** |
|---|---|
| Subrutina llamadora (caller) | (1) Pone los parámetros en los registros r0 - r3. Si necesita más, debe hacer uso del "stack".<br>(2) Debe guardar en el "stack" los registros "caller-saved" (registros r0-r3, r11, r12, r13) que esté utilizando cuyos valores deben conservarse a la vuelta de la subrutina.<br>(3) Llama a la subrutina por medio de la instrucción "bl" o mediante la carga de la dirección en el registro "Intra-procedure call" r12. |
| Subrutina llamada (callee) | (4) Si es necesario crea un "stack frame" r11, realizando una copia del registro que apunta al "stack" r13.<br>(5) Guardar en el "stack" los registros "callee-saved" (r4-r10) que se vayan a utilizar. La dirección de retorno del r14 ó r12 se salvará únicamente en el caso de llamar a otra subrutina.<br>(6) Ejecuta el cuerpo de la subrutina.<br>(7) Para retornar de la función:<br>• Pone el valor o valores de retorno en los registros r0-r1 (y en la pila en caso de devolver más de dos valores).<br>• Restaura el valor de los registros "callee-saved" y elimina variables locales.<br>• Salta a la dirección de retorno almacenada en el "link register" r14 ó r12, usando la instrucción "move" en referencia al "program counter" r15. |
| Subrutina llamadora (caller) | (8) Para finalizar, la subrutina llamadora debe:<br>• Restaura el valor de los registros "caller-saved".<br>• Si se pasaron argumentos adicionales por el "stack", limpiarlos.<br>• Extraer el valor o valores de retorno de r0-r1, (y de la pila en caso de devolver más de dos valores). |

Tabla 6 - Convenio ARM

## 2.8 *Segmentación*

La segmentación en la arquitectura ARM es similar a la de MIPS. En la Sección 2.8 se recoge un procesador ARM segmentado de cinco etapas como modo de ejemplo.

La mayoría de las versiones de ARM poseen las siguientes cinco etapas:

- Fetch (IF): la instrucción se carga de memoria y se coloca en el pipeline.
- Decodificación (ID): la instrucción se decodifica y los registros operandos son leídos del banco de registros. Existen tres operandos, por lo que se debe asegurar su acceso en un ciclo de reloj.
- Ejecución (EX): Uno de los operandos es desplazado y la ALU realiza una operación con ellos. Si la instrucción es load/store, la ALU calcula la dirección de memoria.
- Buffer/Data (MEM): Se accede a la memoria si es requerido. Si no la ALU almacena en el buffer el resultado anterior durante este ciclo de reloj.
- Write Back (WB): Se almacena el resultado generado por la instrucción en el registro destino.

Existen situaciones, denominadas riesgos, que impiden que el siguiente paso en el pipeline sea ejecutado con normalidad, reduciendo la velocidad en el procesado de instrucciones.

Existen tres clases de riesgos:

- Riesgos estructurales: cuando el hardware no soporta las posibles combinaciones de instrucciones.

El *load* necesita acceder a memoria, y lo hace en el cuarto paso (MEM). La otra instrucción requiere ser cargada de memoria para estar en el pipeline (IF). Si el procesador utilizase el mismo bus para acceder a datos en memoria que para instrucciones, esta situación supondría un problema. Dado que ARM emplea una arquitectura de Harvard, el bus de datos y el de instrucciones se encuentran separados. La Tabla 7 refleja esta situación.

| ***Ciclo de reloj*** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| ***Instrucción*** | ***1*** | ***2*** | ***3*** | ***4*** | ***5*** | ***6*** | ***7*** | ***8*** |
| ***Load*** | ***IF*** | ***ID*** | ***EX*** | ***MEM*** | ***WB*** | | | |
| ***Instrucción 1*** | | ***IF*** | ***ID*** | ***EX*** | ***MEM*** | ***WB*** | | |
| ***Instrucción 2*** | | | ***IF*** | ***ID*** | ***EX*** | ***MEM*** | ***WB*** | |
| ***Instrucción 3*** | | | | ***IF*** | ***ID*** | ***EX*** | ***MEM*** | ***WB*** |

Tabla 7 - Ejecución de un "load" y una instrucción después de dos ciclos

- Riesgos de datos: Ocurre cuando se emplea como registro destino uno que es utilizado como operando en las siguientes instrucciones. La Tabla 8 recoge un ejemplo.

En estos casos se emplea la técnica del adelantamiento: el resultado de la primera instrucción se almacena en la ALU como si fuera un *latch*. Si el hardware de adelantamiento detecta que la instrucción previa ha utilizado uno de sus operandos como registro destino, la lógica de control selecciona este valor en lugar del que se encuentra en el banco de registros. Dicho procedimiento se recoge en la Figura 4 [21].

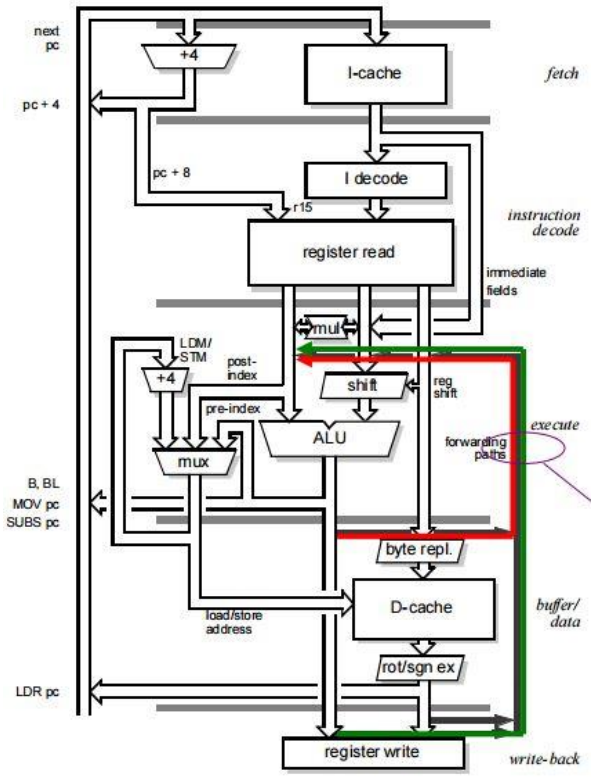



Figura 4 - Ejemplo de la técnica de adelantamiento.

| *Ciclo de reloj* | | | | | | | |
|---|---|---|---|---|---|---|---|
| | ***1*** | ***2*** | ***3*** | ***4*** | ***5*** | ***6*** | ***7*** |
| ***ADD R1, R2, R3*** | ***IF*** | ***ID*** | ***EX*** | ***MEM*** | ***WB*** | | |
| ***SUB R4, R5, R1*** | | ***IF*** | ***ID*** | ***EX*** | ***MEM*** | ***WB*** | |
| ***AND R6, R1, R7*** | | | ***IF*** | ***ID*** | ***EX*** | ***MEM*** | ***WB*** |

Tabla 8 - Ejecución de tres instrucciones en las que se repite un registro como argumento

- Riesgos de control: también denominados riesgos de *branch.* Se producen cuando el procesador no puede ejecutar la instrucción adecuada en su ciclo correspondiente, ya que la instrucción de la fase *fetch* no es la adecuada. Ocurre con las instrucciones de salto condicionales, y se suele recurrir a técnicas de predicción de saltos.

  En ARM estas técnicas resultan más complicadas que en el caso de MIPS, ya que existen una mayor cantidad de saltos y todos pueden ser condicionales. ARM implementa una pila denominada *pila de retorno,* donde se almacenan las direcciones de retorno de las subrutinas. Dichas direcciones se comparan con los resultados de las instrucciones "mov", "load" y ALU que utilizan al PC como registro destino, pudiendo predecir si dicha instrucción actúa como un salto o no y evitando riesgos de control.

  Las predicciones de salto son importantes ya que permiten reducir las pérdidas de CPI. Los riesgos de control en ARM tienen una mayor trascendencia en procesadores con mayor número de etapas. En el procesador Cortex A57 que presenta hasta 19 fases en la segmentación, estos riesgos resultan fatales.

## 2.9 *Subconjunto de instrucciones Thumb*

A continuación se explica el concepto de instrucciones *Thumb* y cómo son interpretadas y ejecutadas por el procesador ARM.

Se trata de un subconjunto de instrucciones ARM codificadas en 16 bits. *Thumb* no agrega nuevas instrucciones, todo lo que hace es una "compresión" de las instrucciones de 32 bits. En este estado, el procesador emplea un hardware de expansión para traducir los 16 bits a una instrucción de 32. El microprocesador siempre ejecuta instrucciones de 32 bits.

La entrada y salida del estado *Thumb* se realiza mediante una instrucción especial (BX). En caso de producirse una excepción durante este estado se produce un cambio automático a 32 bits, pudiendo retornar otra vez a *Thumb* una vez finalizada la excepción.

Está diseñado para optimizar el funcionamiento de ARM, permite una reducción de hasta un 40% en el tamaño del código generado y aumenta el rendimiento al reducir los *misses* en la caché de instrucciones.

El procesador de la *Raspberry Pi* soporta instrucciones *Thumb,* pero ciertos sistemas operativos, como *Raspbian* o *RISC OS* no lo tienen integrado.

La compresión se consigue mediante una reducción en la cantidad de bits disponibles para identificación de registros (solamente se emplean r0-r8), y evitando el uso de los bits de condición (que van a codificar la condición de siempre). La Figura 5 recoge un ejemplo empleando la instrucción "add".

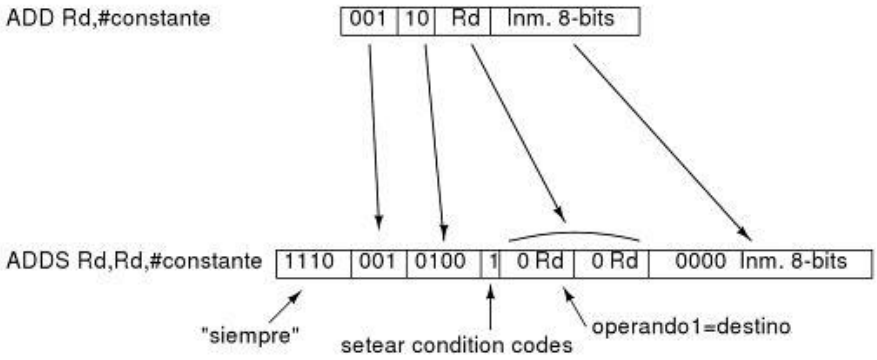



Figura 5 - Compresión de una instrucción "add" a formato *Thumb*

Como se puede observar, el registro del primer operando en el formato de 16 bits, se convierte en registro operando y destino en el formato de 32 bits. Además, las flags de condición están desactivadas, el bit "S" se encuentra siempre habilitado y el campo del inmediato se extiende con ceros a 16 bits.

La expansión de las instrucciones en el modo *Thumb* se realiza en el pipeline, en la etapa de decodificación. Simplemente existe una lógica combinatoria adicional que traduce la palabra de 16 bits a su equivalente de 32, antes de proceder a la decodificación de la instrucción. Este procedimiento se resume en la Figura 6:

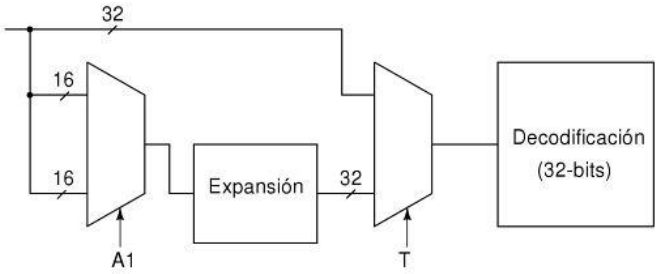



Figura 6 - Expansión de una instrucción en formato *Thumb* a formato ARM

## 2.10 *Extensión Jazelle*

La tecnología de *Jazelle* es una combinación de hardware y software para ARM. Aporta un servicio multitarea basado en máquinas virtuales Java (JVM), con el que se consigue un alto rendimiento en aplicaciones y juegos, utilizando para ello poca memoria y presupuesto.

Cuando se encuentra en estado *Jazelle* el procesador ejecuta programas "opcode", que realmente son objetos contenidos en una o más clases. El PC se va incrementando y seleccionado el siguiente "opcode" a ejecutar.

# 3 ELEMENTOS FÍSICOS DE LA *RASPBERRY PI*

La *Raspberry Pi* es un dispositivo basado en ARM y sus periféricos puede encontrarse en [5]. En él también pueden localizarse los registros encargados de controlar la generación de las interrupciones y sus diferentes funcionalidades. En este proyecto se va a hacer énfasis en el reloj del sistema. A continuación se recogen los recursos físicos que se utilizaron en los Capítulos 4 y 5 para implementar un *driver* en RISC OS y un contador siguiendo la programación *Bare Metal*.

## 3.1 *Reloj del sistema*

Este periférico es responsable de contabilizar el tiempo del sistema. Se encuentra en la dirección física *0x20003000*, en la dirección lógica *0x7E003000* y presenta los registros de 32 bits detallados en la Tabla 9.

| Offset | Nombre | Función | Lectura/Escritura |
|---|---|---|---|
| +0x00 | Control/Status (CS) | Indica el resultado de la comparación de los registros "Compare". | Ambos |
| +0x04 | Counter (CLO) | Contador que se incrementa cada 1 MHz | Lectura |
| +0x08 | Counter (CHI) | Extensión del CLO | Lectura |
| +0x0C | Compare 0 (C0) | Registro de comparación | Ambos |
| +0x10 | Compare 1 (C1) | Registro de comparación | Ambos |
| +0x14 | Compare 2 (C2) | Registro de comparación | Ambos |
| +0x18 | Compare 3 (C3) | Registro de comparación | Ambos |

Tabla 9 - Registros del reloj del sistema

*CLO* y *CHI* forman un contador de 64 bits, que se incrementa cada microsegundo y cuya resolución se denomina *ticks*. Se puede utilizar para tomar referencias temporales. Cada vez que se incrementa, se compara el registro *CLO* con los 4 registros de comparación. Si alguno tiene su mismo valor, el registro *CS* habilita el bit correspondiente dentro de su propio mapa de bits y se produce una interrupción si se encuentra habilitada.

## 3.2 *General Purpose Input Output*

Una de las razones por las que la *Raspberry Pi* se ha hecho tan popular, ha sido su habilidad para ser interconectada con electrónica externa. La GPIO (*General Purpose Input/Output*) es la encargada de realizar esta tarea. Su localización física y sus voltajes se pueden observar en las Figuras 7 y 8.

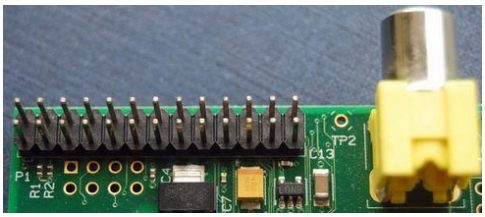



Figura 7 - GPIO

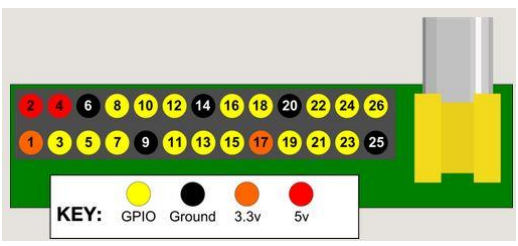



Figura 8 - Pines de la GPIO y su función

### 3.2.1 El controlador de la GPIO

La GPIO[2] tiene su propio controlador, se localiza en la dirección *0x20200000*, y está compuesto por 41 registros. Los seis primeros determinan si cada uno de los 54 pines va a ser utilizado como entrada o como salida (Tabla 10). Contienen tres bits específicos para cada uno de los pines. Si el objetivo es utilizar un pin concreto como entrada, se cargará el valor "000" en el *GPSEL* correspondiente, y si se pretende emplearlo como salida "001".

| Número de registro | Nombre | Offset | Pines asignados |
|---|---|---|---|
| 0 | GPIO Función Seleccionar 0 (GPSEL0) | #0 | 0-9 |
| 1 | GPIO Función Seleccionar 1 (GPSEL1) | #4 | 10-19 |
| 2 | GPIO Función Seleccionar 2 (GPSEL2) | #8 | 20-29 |
| 3 | GPIO Función Seleccionar 3 (GPSEL3) | #12 | 30-39 |
| 4 | GPIO Función Seleccionar 4 (GPSEL4) | #16 | 40-49 |
| 5 | GPIO Función Seleccionar 5 (GPSEL5) | #20 | 50-53 |

Tabla 10 - Registros del controlador de la GPIO (I)

Los cuatro registros siguientes son los encargados de activar y desactivar los pines. Los registros 7/8 los activan y los 9/10 los deshabilitan. Su comportamiento se puede reflejar en la Tabla 11:

| Número de registro | Nombre | Offset | Pines asignados |
|---|---|---|---|
| 7 | GPIO Pin Set 0 (GPSET0) | #28 | 0-31 |
| 8 | GPIO Pin Set 1 (GPSET1) | #32 | 32-53 |
| 9 | GPIO Output Clear 0 (GPCLR0) | #40 | 0-31 |
| 10 | GPIO Output Clear 1 (GPCLR1) | #44 | 32-53 |

Tabla 11 - Registros del controlador de la GPIO (II)

Una tensión cercana a los 3,3 voltios se considera un uno lógico, mientras que un cero lógico es una tensión cercana al cero. El voltaje umbral se encuentra cerca de 1,8 voltios, pero no está garantizado, ya que puede fluctuar. No se pueden extraer más de 50 mA del pin de 3,3 voltios ni más de 700 mA del pin de 5 voltios. Sus pines reciben nombres concretos y presentan funcionalidades diferentes, como se observa en la Figura 9 [13].

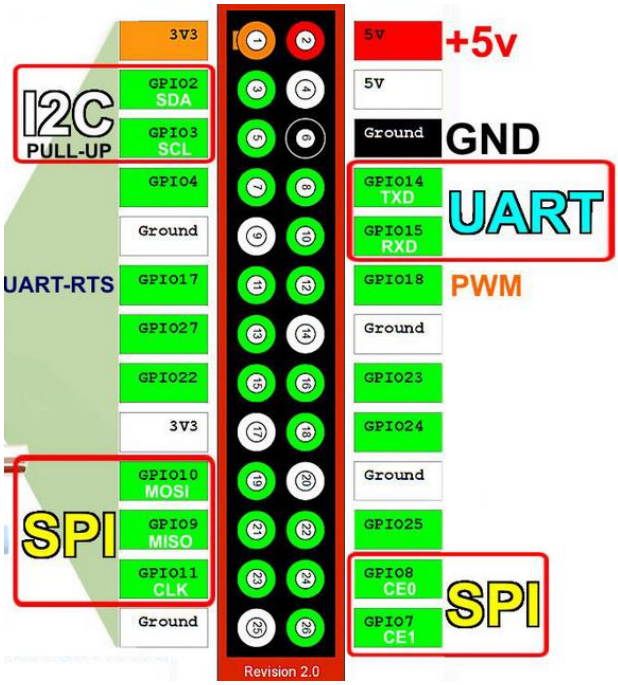



Figura 9 - Denominación común de los pines de la GPIO

[2] En el anexo existe un ejemplo concreto de la activación de uno de los pines.

Cada pin tiene unos valores por defecto (uno o cero lógicos). A su vez, existen registros controladores que permitirán al programador conocer y modificar su estado.

El conjunto de registros controladores puede apreciarse en la Figura 10. De ella, por ejemplo, en el Capítulo 5 se utilizarán los *High Level Detect Enable Regs*, que permiten seleccionar los pines que se van a controlar para recibir tensiones de subida. Los *Function Select Regs* capacitan al programador de habilitar las funciones específicas de los pines tal y como aparecen en la Figura 9.

El conjunto de controladores, así como las funciones de cada pin pueden encontrarse en la hoja de características [5].

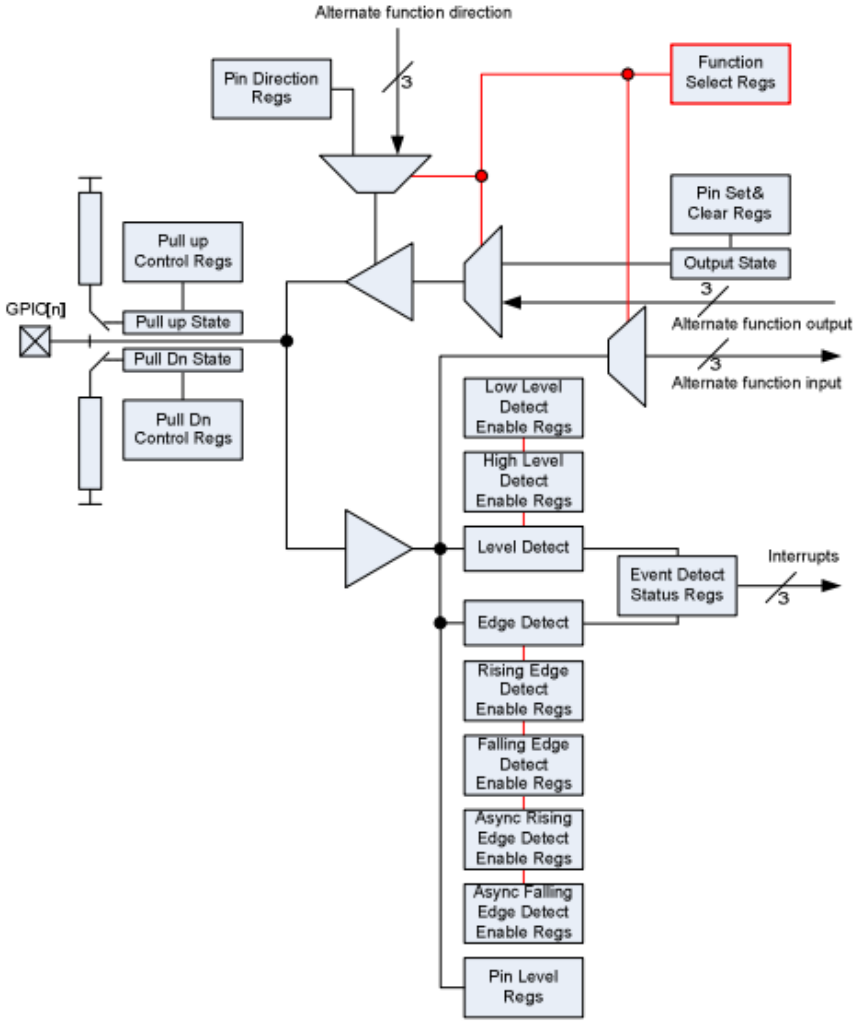



Figura 10 - Diagrama de bloques de la GPIO

Si se conectase un botón al GPIO 9 (que se encuentra en cero lógico por defecto), estando habilitadas las interrupciones mediante el GPHEN0 (*GPIO Pin High Detect Enable 0*) o por el GPREN0 (*GPIO Pin Rising Edge Detect Enable 0*) y finalmente se pulsase, pasaría a tener un valor uno lógico que sería detectado por el sistema. Para eliminar la interrupción, habría que seleccionar la posición correspondiente en el GPEDS0 (*GPIO Pin Event Detect Status 0*).

## 3.3 *Interrupciones*

El procesador ARM soporta dos tipos de interrupciones: las provenientes de los periféricos de la GPU y las de control de los periféricos de la CPU.

Para cada origen de interrupción (ARM o GPU) existe un bit de habilitación de la interrupción (lectura/escritura) y un bit de interrupción pendiente (lectura). Todas aquellas generadas por el bloque de

control ARM permanecen habilitadas hasta que se las deshabilite o el origen de la interrupción se controle.

Existen tres registros para controlar las interrupciones pendientes. Para aumentar la velocidad de procesamiento, el registro de interrupciones básicas pendientes incluye las interrupciones de la GPU más frecuentes.

Se organizan de siguiendo el esquema de la Figura 11:

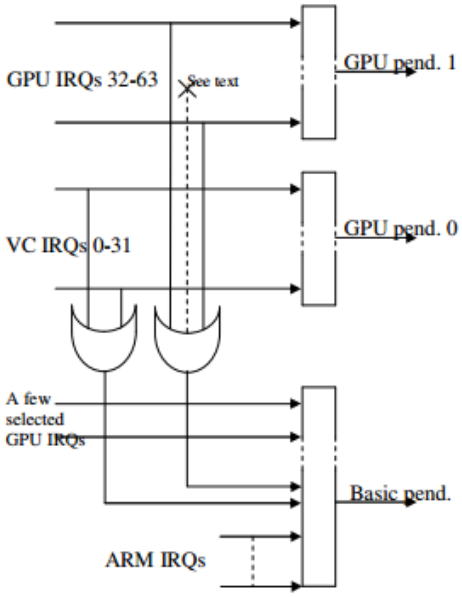



Figura 11 - Interrupt Pending Registers

El procesador ARM también soporta interrupciones *FIQ*. Las fuentes de interrupción se pueden seleccionar para ser conectadas al bus *FIQ* y existe a su vez un registro habilitador.

No existen prioridades en las interrupciones. Si cualquiera se quiere considerar más importante que las demás, deberá seleccionarse como *FIQ*. Las interrupciones restantes se procesarán realizando *polling* sobre el registro de interrupciones pendientes.

#### 3.3.1.1 Registros de interrupción

Los registros encargados de su procesamiento se encuentran en la dirección lógica *0x7E00B000* y en la dirección física *0x2000B000*. Se recogen en la Tabla 12:

| **Offset** | **Nombre** |
|---|---|
| 0x200 | IRQ basic pending |
| 0x204 | IRQ pending 1 |
| 0x208 | IRQ pending 2 |
| 0x20C | FIQ control |
| 0x210 | Enable IRQs 1 |
| 0x214 | Enable IRQs 2 |
| 0x218 | Enable Basic IRQs |
| 0x21C | Disable IRQs 1 |
| 0x220 | Disable IRQs 2 |
| 0x224 | Disable Basic IRQs |

Tabla 12 - Registros de control de interrupciones

La división en registros se debe al rango de interrupciones que abarcan. El registro uno contiene los rangos 0 -31 y el dos 32 - 63. A continuación se recogen las interrupciones que han sido testeadas en el proyecto *Embedded Xinu* [8].

La Tabla 13 resume las interrupciones utilizables:

| **Número de IRQ** | **Dispositivo** |
|---|---|
| 0 | System Timer Compare Register 0 (usada por la GPU) |
| 1 | System Timer Compare Register 1 |
| 2 | System Timer Compare Register 2 (usada por la GPU) |
| 3 | System Timer Compare Register 3 |
| 9 | USB Controller |
| 29 | Aux int |
| 43 | i2c_spi_slv_int |
| 45 | Pwa |
| 46 | Pwa1 |
| 48 | Smi |
| 49 | gpio_int[0] |
| 50 | gpio_int[1] |
| 51 | gpio_int[2] |
| 52 | gpio_int[3] |
| 53 | i2c_int |
| 54 | spi_int |
| 55 | PCM Audio |
| 57 | PL011 UART |
| 62 | SD Host Controller |

Tabla 13 - Recopilación de las interrupciones utilizables

# 4 PROGRAMACIÓN EN RISC OS

RISC OS es un sistema operativo diferente. No está basado en Linux, si no que es un sistema operativo de escritorio diseñado por los creadores del procesador ARM para el procesador ARM original. Se caracteriza por ser reducido (ocupa menos de 6MB) y por su rapidez. Fue desarrollado en una época en la que el procesador más potente era un ARM2 de 8MHz con 512KB de RAM. Es mucho más simple que otros sistemas operativos actuales, lo que lo hace más fácil de comprender.

## 4.1 *Introducción*

RISC OS es un sistema operativo cuyo funcionamiento se puede comprender de manera intuitiva. Cuenta con una versión especial para ser implementada en el dispositivo ***Raspberry Pi***, teniendo gran utilidad en la docencia. Está compuesto por un repertorio de carpetas dentro de las cuales el usuario puede acceder a manuales, tanto de programación en ensamblador, como del propio sistema operativo (denominados **Programmer's Reference Manuals**[3]).

En el Capítulo 4 se va a proceder a explicar la programación en RISC OS en ensamblador mediante BASIC, utilizando la herramienta GCC y diferentes métodos de depuración. Además se van a introducir conceptos acerca de su funcionamiento. Finalmente, se explicará una posible práctica, que en este caso fue un *driver* para controlar el reloj del sistema. Toda la información de este capítulo fue obtenida de [3], [4] y [10].

## 4.2 *Programación empleando BASIC*

RISC OS no permite de forma nativa programar en ensamblador. En su lugar, lo que se realizará en este capítulo es diseñar programas en bajo nivel embebidos en BASIC.

Para realizar esta tarea de una manera cómoda, se utilizará un editor de texto denominado **!StrongED**, disponible en el escritorio. Para que se reconozca el formato del texto, es necesario guardar el archivo con la extensión **BASIC**.

### 4.2.1 Diseño de un programa ensamblador en BASIC

Los programas ensamblador deben tener una estructura fija, debido a que están embebidos en programas BASIC. RISC OS cuenta con documentación acerca de su programación mediante BASIC en la carpeta *First Steps Programming*, en el directorio *SFDS::RISCOSpi.$.Documents.Books*. La Figura 12 se recoge su formato[4]:

```
DIM code% (100)
FOR pass =0 TO 3 STEP 3
P%=code%
[
OPT pass
.start
; comment
]
NEXT pass
END
```

Figura 12 - Estructura de código BASIC para la programación ensamblador

[3] Esta documentación se encuentra en la carpeta PRMs en el directorio *SFDS::RISCOSpi.$.Documents.Books*.

[4] Este formato es estándar, pero puede variar en función del tipo de programa que se pretenda crear. Los módulos, como se verá en la Sección 4.7, tienen otro formato.

El programa localizado entre los corchetes corresponde al programa nativo en ensamblador. El resto del código simplemente son instrucciones BASIC necesarias para su implementación.

- **DIM code% (100)** indica el tamaño en bytes del programa.
- Las instrucciones en BASIC son ejecutadas secuencialmente. El ciclo **FOR** se utiliza para que el programa pueda hacer referencia a sentencias posteriores, de esta manera el procesador es capaz de ensamblarlas y no producir error.
- **P%** es un puntero a la región de memoria donde va a ser almacenado el programa. Mediante la variable **code** se reserva el tamaño considerado.
- **OPT** es una función que controla como el programa es ensamblado. Debe de ir siempre seguido por un valor, que determina cómo se crea el código máquina. En este caso, el valor de OPT viene dado por el ciclo FOR. La función OPT 0 desactiva el reporte de errores, mientras que OPT 3 lo activa.

### 4.2.2 Ejecución del programa ensamblador

RISC OS cuenta con una ventana de comandos (denominada **Taskwindow**), de forma semejante a lo que ocurre en Linux y otros sistemas operativos. Sin embargo, el repertorio de comandos es mucho más limitado. Para acceder dicho terminal, se pulsa la combinación de teclas **Ctrl + F12**. También es posible acceder mediante las opciones del sistema, pulsando con el botón de *scroll* en el icono de *Raspberry Pi* situado en el margen inferior derecho.

A continuación, es necesario seleccionar el directorio en el cual se ha guardado el programa BASIC que se ha diseñado previamente. Si el usuario no ha realizado ninguna modificación, el programa aparece en la carpeta *SDFS::RISCOSpi.$*, perteneciente a la tarjeta SD donde se ha guardado el sistema operativo[5].

Una vez seleccionado el directorio, se carga el archivo mediante ***LOAD file_name** y se compila utilizando el comando **RUN**. Si hubiese errores de compilación, se anunciarían al usuario al finalizar este paso.

Finalmente, para ejecutar el programa se emplea el comando **CALL start**. Comenzaría a ejecutarse a partir de la directiva **.start**. Es posible mapear variables desde BASIC a ensamblador mediante este comando, que serían almacenados en un conjunto específico de registros[6].

## 4.3 *Programación mediante GNU Compiler Collection*

RISC OS dispone de multitud de paquetes adicionales que pueden ser descargados para ser utilizados en él. **!PackMan** es el programa dedicado a su descarga, se puede localizar en el escritorio.

Entre los diferentes paquetes, existe uno encargado de la compilación de programas C, C++ y ensamblador. Su nombre es **GNU Compiler Collection**[7] y su uso es semejante al de otros sistemas operativos.

Algunas de las ventajas más importantes del uso de GNU es la independencia frente a BASIC y su mayor flexibilidad, ya que permite incluir directivas que con BASIC no se podían.

Para compilar un programa utilizado la herramienta GNU, se utiliza en primer lugar el comando que se recoge a continuación, con el que se habrá convertido el programa ensamblador a programa objeto:

```
as –o file.o file.s
```

A continuación se realizará un enlace con el programa objeto y se habrá creado un archivo ejecutable:

[5] Si el usuario hubiese copiado el programa *BASIC* en otra carpeta, simplemente habría que pulsar con el botón de *scroll* dentro de ella y seleccionar la acción *Set Directory*.
[6] Este comando se procede a explicar más detenidamente en el apéndice de la memoria.
[7] La descarga e instalación de la herramienta *GCC* se recoge en el apéndice de la memoria.

```
ld –o file file.o
```

De esta manera, se podrán vincular diferentes programas en ensamblador en uno sólo (para el empleo de subrutinas, por ejemplo). Para ejecutar el programa, basta con escribir su nombre en la *TaskWindow*.

## 4.4 *Depuración*

RISC OS contiene herramientas muy útiles para realizar depuración en los programas, y con ello conocer por qué ciertas partes del código no funcionan según lo esperado. Entre ellas se pueden destacar el empleo de *breakpoints* tan comunes en otras arquitecturas y el desensamblador que traduce a código ensamblador el contenido de una dirección de memoria dada por el usuario. De esta manera, el estudiante podrá comprobar si su programa funciona de la manera correcta.

### 4.4.1 SWI 256

Ante un error de ejecución, RISC OS reacciona congelándose, en la mayoría de los casos no se podrá interactuar con él a menos que se reinicie la *Raspberry Pi*, quitándo y conectando la fuente de alimentación.

Si el programa en cuestión se congela, la tarea más importante es conocer en qué punto ocurre. Para ello, una solución bastante sencilla es mostrar por pantalla caracteres identificativos en cada sección del código. Si a partir de uno de ellos el sistema no responde, se puede conocer el motivo del problema.

En este caso se suele utilizar la instrucción *SWI 256*, ya que no requiere el empleo de ningún registro extra, simplemente un inmediato, para mostrar caracteres por pantalla. Por ejemplo, el *SWI 256+65* deberá mostrar una "A" por pantalla, si no es así el error se encuentra antes de su ejecución.

### 4.4.2 Breakpoint

Los *breakpoint* son paradas temporales en la ejecución de un programa. Provocan un "abort exception" en el estado del procesador, y permiten mostrar por pantalla el contenido de los registros normales y de los de estado. Pueden programarse tantos como se desee mediante el comando:

```
*BREAKSET <address>
```

siendo *address* la dirección de la siguiente instrucción. Para obtener la lista de los breakpoint programados:

```
*BREAKLIST
```

Para eliminarlos:

```
*BREAKCLR
```

Sin embargo, si no se desea programarlos mediante comandos desde *TaskWindow*, se pueden insertar mediante instrucciones ARM. Su formato es:

```
BKPT
```

De esta manera, para obtener una visión de todos los registros y su contenido, se empleará el siguiente comando, una vez que se haya parado la ejecución del programa mediante breakpoint:

```
*SHOWREGS
```

### 4.4.3 Manipulación de memoria

Si el programa a depurar realiza operaciones en memoria, se pueden manipular diversas secciones para conocer si opera de manera correcta sobre ella.

Un ejemplo podría ser inicializar el contenido de diversas partes de la memoria, a valores conocidos por el usuario, de manera que podría saber qué operaciones ha generado el programa sobre ella al final de su ejecución.

Para conocer el contenido en una región específica de memoria, se emplea el siguiente comando, siendo "start of memory area" la dirección de inicio y "size of memory" el tamaño de memoria que queremos estudiar en bytes:

***MEMORY (B)[8] <start of memory area> + <size of memory>**

Para fijar una sección de memoria con un valor determinado:

***MEMORYA (B) <address> <data>**

### 4.4.4 Disassembler

Es un programa que transforma el código máquina de una sección de memoria a lenguaje ensamblador. El formato del comando es el siguiente:

***MEMORYI <start_address> <final_address>**

En este caso el código máquina entre esas dos direcciones sería transformado a lenguaje ensamblador.

## 4.5 *Software vectors*

Una de las características más importantes de RISC OS es la facilidad con la que puede ser modificado y extendido. La mayor parte de su sistema está escrito mediante módulos, que pueden ser sustituidos o incluso añadidos.

La excepción a este caso es el kernel, que provee un conjunto de funcionalidades centrales necesarias para el funcionamiento de RISC OS. No se puede sustituir el kernel al completo pero sí pueden cambiarse ciertas rutinas. Esto se realiza mediante los denominados *software vectors*, almacenados en la memoria RAM. Muchas de estas rutinas realizan las funcionalidades de un *Software Interrupt*, que reciben el nombre de *SWIs vectorizados.*

### 4.5.1 Claiming vectors

Cuando se ejecuta un *SWI*, RISC OS emplea el número para decidir qué rutina se va utilizar. Si se trata de un *SWI* normal, RISC OS ejecuta el código específico localizado en la región de memoria proporcionada por el vector de excepciones del modo supervisor. Sin embargo, si se trata de un *SWI vectorizada*, esta región de memoria contiene un *branch* a otra dirección de memoria donde se localizará la subrutina a ejecutar.

Es posible cambiar dicha dirección empleando la llamada al sistema *OS_Claim (SWI &1F).* RISC OS ejecutaría la rutina seleccionada, en lugar de la proporcionada por el vector original. Utilizando esta instrucción podría realizarse una de las tres opciones siguientes:

- Reemplazar la rutina original, con lo que se estaría interceptando la llamada.
- Realizar cierto procesado antes de ejecutar la rutina original.
- Ejecutar la rutina original y posteriormente procesar los datos.

[8] B indica que se opera a nivel de byte.

#### 4.5.1.1 Ejemplo de vector claiming

Un ejemplo del apartado anterior es la llamada al sistema *OS_WriteC*. Cuando RISC OS la decodifica, entiende que se está demandando una operación de escritura de un carácter. RISC OS obtiene la dirección de un vector, denominado *WrchV*, que retornará la dirección del *SWI* estándar, permitiendo que se ejecute la rutina.

Por defecto, *WrchV* contiene la dirección de la rutina anterior. Si se sustituye esa dirección mediante la llamada al sistema *OS_Claim*, se podría ejecutar una rutina dada por el usuario. Si por otro lado se pretendiese añadir otra dirección sin sustituir la inicial, se utilizaría la llamada al sistema *OS_AddToVector (SWI &47).* Con ello se conseguiría que la subrutina cuya dirección se ha almacenado en el vector se ejecute antes de la que se encuentra por defecto.

Existen algunos vectores que no deben ser interceptados, porque la rutina por defecto puede realizar acciones importantes que deben ser ejecutadas. Un ejemplo es el vector *ErrorV*, que enlaza con el *error handler*. Si se sustituyese por otra dirección, el *error handler* nunca sería ejecutado.

## 4.6 *Software Interrupts*

RISC OS presenta un conjunto de *Software Interrupts (SWI)* específico. Su documentación puede localizarse en el directorio *SDFS::RISCOSpi.$.Documents.Books*[9]. A través de ellas se puede acceder a diferentes funciones del sistema operativo, como por ejemplo el audio.

### 4.6.1 Identificación

Las instrucciones *Software Interrupt* están codificadas empleando una numeración específica. Cada una posee su propio identificador, que está constituido por 24 bits, los cuales se explican a continuación:

- Bits 20 – 23: identifican el sistema operativo en el que se van a utilizar. En RISC OS, estos bits valen 0.
- Bits 18 – 19: identifican que parte del software del sistema implementa la instrucción. Presenta los siguientes valores: (bit 19, 18) 00: sistema operativo, 01 módulos que añaden funcionalidades al sistema operativo, 10 aplicaciones externas y 11 aplicaciones del usuario.
- Bit 17: indica si se han producido errores. Es el bit 'X'.
- Bits 6 – 16: reciben el nombre de *SWI Chunk Identification number*[10]*s.* Identifican un bloque de 64 *SWIs* consecutivas, que forman parte de una aplicación o de un módulo.
- Bits 0 – 5: identifican *SWIs* individuales dentro de un *chunk.*

### 4.6.2 Proceso de ejecución

Las instrucciones *Software Interrupt* forman parte del modo *supervisor*, de manera que en el momento en el que se ejecutan, el PC realiza un *branch* a la dirección &08[11] y la dirección *return address* es copiada en el registro *R14_svc.* Las interrupciones *normales* son deshabilitadas (bit I del CPSR) y se cambia el CPSR a modo *supervisor*. Después se resetea la flag de overflow y se ejecuta el *SWI*, cuyo valor se conoce a través de la dirección contenida en *R14_svc.* Mediante los bits 0-16 se conoce qué rutina habrá que ejecutar, pudiendo tratarse de un *SWI* vectorizado[12]. Al finalizar la ejecución, se comprueba si se han producido errores (realizando un *error handling* en caso de que los hubiese) y se retorna mediante la dirección contenida en *R14_svc.*

#### 4.6.2.1 Tratamiento de errores

Antes de que RISC OS finalice el proceso de retorno de la instrucción *Software Interrupt*, comprueba la

[9] Es necesario activar la lectura de PDP ejecutando el programa **!PDF** en el directorio *SDFS::RISCOSpi.$.Utilities*.
[10] Este identificador se explicará en el apartado de módulos.
[11] Dirección del vector de excepciones del modo *supervisor*.
[12] La definición puede encontrarse en el apartado de *Software vectors.*

bandera de overflow. Si vale cero significa que no se ha producido ningún error y se retorna de forma habitual. En caso contrario, RISC OS comprueba la instrucción *SWI* original que se utilizó:

- Si el bit 17 vale cero, RISC OS trata él mismo el error.
- Si el bit 17 vale uno, RISC OS devuelve el control directamente al programa del usuario, que conocerá que se ha producido un error por la presencia de la flag overflow y R0 contendrá un puntero a la causa del problema. El usuario deberá tratar el error él mismo.

El *error block* contiene una palabra con el número de error y la dirección posterior un mensaje de error, terminado en nulo. Los *error numbers* presentan un formato específico, de la misma manera que las instrucciones *Software Interrupt* pudiendo ser clasificados mediante rangos numéricos.

### 4.6.3 Tipos de Software Interrupt

Existen dos tipos de *SWI*, conocidas respectivamente como *error-generating* y *error-returning*. Las instrucciones *error- generating SWIs* son las que se utilizan normalmente, mientras que las *error returning* se consiguen habilitando el bit 17: sumando &20000 al número de la instrucción *SWI* que se pretende utilizar o añadiendo la letra **X** delante del nombre (por ejemplo, *XOS_WriteC*). Las *error returning* permiten en caso de error ser tratadas por el propio usuario.

### 4.6.4 Comandos SYS

Las instrucciones *Software Interrupt (SWI)* pueden ser ejecutadas a través de **BBC BASIC** empleando la ventana *TaskWindow*. Permiten conocer el estado del sistema sin necesidad de crear programas específicos. Los comandos *SYS* presentan un formato predeterminado:

**SYS Expr1 {, {exprn}….} {TO {var1} {,var2}….} {;flags} }**

- Expr1: Identifica la instrucción *Software Interrupt (SWI)*. Puede estar definida mediante su nombre completo (string) o con el número al que está asociado.
- exprn: Representa la lista de parámetros que la instrucción *SWI* va a recibir como argumento. Puede recibir un máximo de ocho parámetros y se asignarán a los registros ARM desde R0 a R7. Si el argumento es un string, se colocará en la pila **BBC BASIC** terminado con un carácter nulo y el registro correspondiente contendrá un puntero a esa dirección.
- TO: es opcional y permite retornar parámetros desde el *SWI* hasta el comando *SYS*, vinculando cada registro de salida con las variables establecidas. Si el valor a retornar es un string, se añade un carácter nulo al final y se devuelve un puntero a la dirección de memoria donde se encuentra.
- flags: son opcionales y si se especifican permiten retornar del *SWI* el resultado de las banderas de condición N, Z, C y V.

## 4.7 *Módulos*

Si el procesador ARM puede ser considerado el *corazón* para los ordenadores que lo utilizan, los módulos tienen la misma importancia para RISC OS.

Se considera como una unidad de software que cuando es cargada actúa como una extensión (añadiendo nuevas funcionalidades) del sistema operativo.

Reciben la denominación de *reubicables* porque se ejecutan en una zona de memoria conocida como *Relocatable Module Area (RMA)* y pueden ser cargados en cualquier parte de ella.

Un módulo debe cumplir los siguientes requisitos para recibir la denominación:

- Ser una extensión para RISC OS o una mejora para un módulo ya existente.
- Ser compartido por varias aplicaciones, por ejemplo la librería C.

- Tiene que ocupar memoria RAM cuando es invocado.
- Ser lo suficientemente pequeño.

### 4.7.1 Modo de empleo

El uso de los módulos se basa en el empleo de la instrucción *OS_Module*[13]. Permite la ejecución de todas las operaciones posibles que se puedan realizar sobre ellos. Entre las funciones que contiene se encuentran:

- Carga, inicialización, ejecución y borrado del módulo.
- Exámen y modificación de la cantidad de espacio RMA utilizado por el módulo.
- Petición de información.
- Modificación de las instancias de los módulos.

RISC OS dispone de diversos módulos, almacenados en su ROM. Pueden observarse mediante el comando **Modules* en la ventana *TaskWindow*.

### 4.7.2 Instancia

Un módulo puede ser instanciado más de una vez, formando *workspaces*, los cuales se definen como el área de memoria donde un módulo almacena sus datos de manera dinámica. Esto significa que mientras una sóla copia de su código es almacenada en memoria, pueden existir múltiples copias de su *workspace*, lo que permite su utilización para diferentes objetivos de manera simultánea. Cada copia del *workspace* se define como instancia y se ubica en una dirección de memoria múltiplo de cuatro[14].

En el momento en el que se instancia un módulo, se crea un puntero al comienzo de su *workspace*. Dicho puntero recibe el nombre de *private word* y es fundamental para poder acceder al *workspace* desde las diferentes partes del módulo. Cuando el módulo es invocado desde cualquier punto de acceso, RISC OS carga el registro 12 con el *private Word*, pudiendo obtener la dirección del *workspace* utilizando:

```
LDR R12, [R12]
```

### 4.7.3 Tratamiento de errores

Los módulos deben seguir ciertas normas en relación con la gestión de errores:

- Es obligatorio utilizar los *error-returning SWI* y comprobar si se han producido errores a su retorno.
- Si no se han producido errores, se deben preservar los registros siguiendo el convenio ARM y la flag de overflow debe valer 0.
- Si se han producido errores, R0 debe contener un puntero al *error block* correspondiente, se deben preservar los registros siguiendo el convenio ARM, la flag de overflow debe estar activada y se debe finalizar el módulo.

### 4.7.4 Formato de un módulo

El módulo se define con una cabecera o *module header*, el cual es una tabla con once filas, cada una de tamaño word. RISC OS lo utiliza para comunicarse con el módulo.

Las once filas que constituyen la cabecera del módulo se describen a continuación. En el caso de no utilizar alguno de los campos, se sustituirá el nombre de la etiqueta donde se encuentre el código correspondiente por un valor nulo.

[13] Las *Software Interrupts* que implementan estas funciones se detallan en el apéndice.
[14] Este hecho facilita la ejecución del módulo en situaciones de requerimientos de tiempo real, como por ejemplo manejador de interrupciones *FIQ*.

- **(&00) Código de aplicación**

  Es la parte del módulo que se ejecuta en caso de que se pretendiese iniciar una aplicación. Lo utilizan algunas aplicaciones, como es el caso de BASIC. Se ejecuta mediante el comando *RMRun o utilizando la instrucción *OS_Module 0*.

- **(&04) Código de inicialización**

  Se ejecuta en el momento en el que se carga el módulo, inicializándolo[15]. Un módulo no será invocado por ningún otro punto de acceso hasta que no se haya terminado de inicializar.

  Si se produce algún problema en su inicialización, el sistema operativo lo eliminará de la lista de módulos y se retornará siguiendo el tratamiento de errores mencionado.

  Esta parte del módulo se invoca mediante el comando[16] *RMLoad o utilizando la instrucción *OS_Module 1*.

- **(&08) Código de finalización**

  Se utiliza en el caso de que se pretenda eliminar el módulo del sistema operativo.

  Si un módulo no presenta función de finalización, RISC OS lo eliminará considerando que R12 es el puntero al *workspace*.

  Esta parte del módulo se invoca mediante el comando *RMKill o utilizando la instrucción *OS_Module 4*.

- **(&0C) Código manejador de llamadas de servicio**

  Permite al módulo gestionar una *Service Call*[17] y tratándola de manera adecuada.

  Mediante el registro R1 se especifica al módulo qué tipo de *Service call* se ha originado. Es importante que su procesado sea rápido y se rechacen aquellas notificaciones que no se deseen tratar.

- **(&10) Código del título del módulo**

  Utilizado por RISC OS para referirse al módulo. Debe ser un título corto, meramente descriptivo y terminado en un carácter nulo. Aparece cuando se ejecuta el comando *Modules.

- **(&14) Código de ayuda del módulo**

  Se trata de un string terminado en un carácter nulo que es mostrado cuando se ejecuta el comando *Help <modulename>.

  No debe de contener caracteres de control y debe seguir el siguiente formato:

**<modulename>[<TAB>(if necessary)]v.vv (DD MMM YYYY)**

- **(&18) Código de la tabla de ayuda y decodificación de comandos**

Permite generar y explicar la función de los comandos programados en el módulo. Cada uno de los comandos debe contener los siguientes campos:

- Nombre del comando (terminado en nulo).
- Etiqueta al comienzo del código del comando.
- Una palabra de información.
- Etiqueta a la dirección de descripción de la sintaxis del comando y a su dirección de ayuda.

[15] En el momento en el que se haya cargado, aparecerá en la lista de módulos (**Modules*).
[16] Existe otro comando relacionado con la liberación de memoria y la reinicialización del módulo denominado *RMTidy, pero su uso no es frecuente.
[17] Se trata de una notificación producida por RISC OS con el objetivo de que los módulos provean un servicio acorde a ella. Todas las *service calls* que RISC OS puede originar se pueden encontrar en el apartado de *OS_ServiceCall (&30)* en el **PRM** [9]

Si no se desea añadir más comandos, esta sección debe terminar con un carácter nulo.

- **(&1C) Número base de *Software Interrupt***

Contiene un número en lugar de una etiqueta. Este número se denomina el *chunk number code* del módulo y permite crear nuevos *Software Interrupt* siguiendo con su numeración. En cada módulo se pueden crear un total de 64 *SWIs.* El *chunk number code* para el usuario es el &C0000 y es posible crear *SWIs* para uso público poniéndose en contacto con los desarrolladores de RISC OS y siguiendo las instrucciones recogidas en [21].

- **(&20) Código de *Software Interrupt***

Contiene el puntero a la dirección donde comienza el código de los *Software Interrupts.* El formato a seguir está preestablecido.

- **(&24) Tabla de decodificacón de *Software Interrupt***

Permite relacionar cada *SWI* con un nombre concreto, para no depender exclusivamente del identificador numérico. El final de esta entrada debe terminar con carácter nulo, como se recoge en la Figura 13.

```
EQUS    "Shell"
EQUB    0
EQUS    "Create"
EQUB    0
EQUS    "Destroy"
EQUB    0
EQUB    0
```

Figura 13 - Ejemplo de *tabla de decodificación de SWI*

*Shell* es el prefijo de las dos *SWI.* El módulo provee en este caso *Shell_Create* y *Shell_Destroy*.

- **(&28) Offset de decodificación de *Software Interrupt***

Es utilizada por RISC OS para identificar un particular *SWI.* Si su nombre en formato string no apareciese en la *SWI* decoding table, el sistema operativo lo buscaría en esta tabla.

En general esta entrada no se utiliza y en su lugar se emplea la anterior por completo.

- **(&2C) Código de localización de archivo de mensajes**

Contiene un string terminado en nulo que identifica un directorio. En él se encuentra un archivo que contiene el texto a mostrar mediante los comandos *Help y *Command, permitiendo traducir los mensajes de ayuda a diferentes idiomas.

- **(&30) Offset de bandera de compatibilidad**

Indica el tipo de máquina para el que el módulo ha sido diseñado. Para procesadores de 32 bits este campo debe valer uno.

## 4.8 *Capa de abstracción hardware*

RISC OS está constituido por una capa intermedia entre el usuario y el hardware. Dicha capa se denomina **HAL** *(Hardware Abstraction Layer)* y realiza dos funciones principales: por un lado es responsable del arranque del sistema (como la BIOS de un PC) y por otro lado provee de APIs sencillas para el acceso al hardware.

Las APIs que propone HAL son una capa por encima del hardware. Evitan que el usuario conozca el sistema a nivel físico y el funcionamiento del kernel, permitiendo que los módulos *driver* sean independientes del hardware.

Una decisión bastante importante es el nivel de abstracción requerido a HAL: demasiado poca, y el sistema no sería lo suficientemente flexible o mucha, y su diseño sería demasiado complicado para hardware sencillo.

### 4.8.1 Lista de HAL APIs

RISC OS propone las siguientes APIs através de la capa *Hardware Abstraction Layer*, las cuales pueden encontrarse en [22]. La mayoría son de uso restringido al usuario, y existen nuevas que se encuentran en desarrollo.

- HAL interrupt handling API
- HAL timer API
- HAL counter API
- HAL NVRAM API
- HAL IIC API
- HAL Video API
- HAL matrix keyboard API
- HAL Touchscreen API
- HAL UART API
- HAL PCI API
- HAL ATA API
- HAL keyboard API
- HAL USB API

## 4.9 *Desarrollo de un driver para el control del reloj del sistema*

RISC OS permite el uso del periférico temporal del sistema. Dicho dispositivo presenta una frecuencia de funcionamiento de 1 MHz. En el ejemplo de este apartado va a proceder a diseñarse un driver específico para él siguiendo la API que propone HAL.

### 4.9.1 Modificación de la frecuencia de trabajo del sistema

Dado que la *Raspberry Pi* no posee una BIOS convencional, los parámetros de configuración que normalmente mantiene y utiliza son almacenados en un archivo denominado *config.txt.* Este documento es leído por la GPU antes que el *core* ARM sea inicializado. En RISC OS, el archivo *config.txt* es manipulado por el programa **!Boot.Loader**.

### 4.9.2 Desarrollo del módulo

A continuación se explican las secciones más importantes del módulo *driver* del reloj del sistema. Para el desarrollo del programa es fundamental cumplir con los requisitos del PRM *(Programmer's Reference Manual)* [9]. Para ello se ha utilizado la API que propone HAL. Se ha programado en BASIC dado que la herramienta GCC únicamente permite la compilación de módulos programados en lenguaje C.

- **Configuración del módulo en lenguaje BASIC**

  El formato de configuración de un módulo en lenguaje BASIC es el que se puede observar en la Figura 14:

```
DIM C% 1024
mod$="Timer"
title$="Timer"

FOR OPT%=4 TO 7 STEP 3
P%=0:O%=C%
[OPT OPT%
```

Figura 14 - Configuración en BASIC del módulo

- **Desarrollo del Module Header**

  El *driver* que se procede a implementar[18] permite al usuario interactuar con el periférico de dos maneras: mediante la ejecución de comandos desde *TaskWindow* o con el empleo de *Software Interrupts.*

  La cabecera del módulo, por tanto, es la reflejada en la Figura 15:

```
.RMhead
     DCD 0 ; Código de aplicación
     DCD init ; Código de inicialización
     DCD end ; Código de finalización
     DCD 0 ; Código de llamadas de servicio
     DCD title ; Nombre del módulo
     DCD helpstring ; Comando de ayuda del módulo
     DCD function_table ; Tabla de ayuda y decodificación de los comandos
     DCD &C0000 ; Número base de SWI para este módulo
     DCD SWIentry ; Código de los SWI
     DCD SWInames ; Tabla con los nombres de los SWI
     DCD 0 ; Tabla de decodificación de los nombres de los SWI
     DCD 0 ; Nombre de archivo para mensajes de ayuda y sintaxis de comandos
     DCD RMflags ; Bandera de compatibilidad de 32 bits
```

Figura 15 - Module Header

Como se puede apreciar, los campos que no se van a utilizar se especifican mediante un valor nulo.

- **Código de inicialización**

  El código de inicialización presenta unos patrones que se deben seguir para su correcto funcionamiento.

  - En el momento en el que se acceda a él, R11 contendrá el origen del módulo (0 si ha sido cargado del sistema de archivos), R12 la dirección del puntero al *private word* y R13 pasará a ser R13_svc.
  - A la salida, R7-R11 y R13 deben haber sido preservados. El resto, excepto la flag V (que se encarga de la tarea del tratamiento de errores) pueden ser corrompidos. R14 debe utilizarse para salir.
  - En el momento de su ejecución, el procesador se encuentra en modo *supervisor* y las interrupciones IRQ y FIQ están habilitadas.

[18] Los *Software Interrupts* que se van a utilizar en el código, se encuentran documentados en el anexo de la memoria.

La sección de inicialización del módulo realiza las siguientes funciones:

1. Obtención del identificador global del reloj disponible para uso del usuario.
2. Reserva de espacio para el *workspace*.
3. Establecimiento del período del reloj.
4. Instalación del manejador de excepciones encargado de procesar las interrupciones y determinación del tipo de *device* a diagnosticar.
5. Habilitación de las interrupciones[19] del *driver* correspondiente.

En caso de error en la ejecución de cualquier *Software Interrupt*[20], se procederá a la eliminación del *workspace* y del módulo, realizando un *jump* a la sección de finalización.

Los puntos a destacar en la programación de este apartado fueron la inicialización del *workspace* (Figura 16), instalación del manejador de excepciones para el *driver* (Figura 17) y habilitación de las interrupciones del reloj (Figura 18).

```
  ; OS_Module
       MOV R0, #6 ; Code of operation
       MOV R3, #16 ; Claim 16 bytes
       SWI "XOS Module"
```

Figura 16 - Código de inicialización (I)

```
  ; OS_ClaimDeviceVector
  ; Claim vector for Timer1
       MOV R0, R4 ; Timer device IRQ number
       ADR  R1,  timerhandler  ;  Device  driver
routine
       MOV R2, R12 ; Workspace pointer
       SWI "XOS_ClaimDeviceVector"
```

Figura 17 - Código de inicialización (II)

```
; HAL_EnableIRQ
; Enable IRQs for specific device
     MOV R0, R4 ; Timer device IRQ number
     MOV R8, #0
     MOV R9, #1
     SWI "XOS Hardware"
```

Figura 18 - Código de inicialización (III)

[19] No es necesario habilitar las interrupciones de manera general porque en el momento de su ejecución el procesador ya las activa por sí solo.

[20] Se procederá siguiendo con el proceso de tratamiento de errores explicado en la Sección 4.7.3

- **Código de finalización**

  En el momento de eliminar un módulo, se deben tener los siguientes aspectos en cuenta:

  - Al comienzo de su ejecución, R10 indica si la causa de su eliminación ha sido grave (1) o normal (0, mediante un *RMKill), R12 contiene la dirección del *private word* y R13 pasa a ser R13_svc.
  - Al final de su ejecución, R7-R11 y R13 deben haber sido preservados, el resto pueden ser corrompidos. R14 debe utilizarse para salir.
  - En el momento de su ejecución, las interrupciones IRQ no se ven alteradas y las FIQ son habilitadas. El procesador está en modo *supervisor*.

  La sección de finalización del módulo realiza las siguientes funciones:

  1. Desactivación de las interrupciones del *driver*.
  2. Desinstalación del *driver*, así como la anulación de su manejador de excepciones.
  3. Liberación de memoria del área ocupada por el *workspace*.

- **Entrada de los *Software Interrupt***

  La sección encarga de tratar los *Software Interrupts* establecidos por el módulo debe seguir las siguientes reglas:

  - Al comienzo de su ejecución, R0-R8 pueden contener parámetros necesarios para el funcionamiento del *SWI*. R11 indica su número, R12 contiene la dirección del *private word* y R13 pasa a ser R13_svc.
  - Al finalizar su ejecución, R0-R8 pueden contener parámetros retornados del *SWI* y R10-R12 pueden haber sido corrompidos.
  - En el momento de su ejecución, el procesador está en modo *supervisor*. Las interrupciones FIQ están activadas y las IRQ del mismo modo en el momento de la llamada al *SWI Entry*.

  Las funciones que se realizan en esta sección son las siguientes:

  - Comparación del número de *SWI* proporcionado al módulo con el total de *SWIs* disponibles en él. Si el número del *SWI* es mayor, el módulo no la tiene a su disposición y se produce un error que se muestra por pantalla.
  - Si el número es menor, se redirige al PC a la dirección del código del *SWI* solicitado, ejecutándose y finalizando la llamada.

  Se programaron dos *SWIs* diferentes:

  1. *SWI 0*: resetea el contador, almacenando un cero en la dirección correspondiente del workspace.
  2. *SWI 1*: retorna el valor del contador del reloj.

- **Tabla de nombres de los *Software Interrupt***

  Este apartado contiene los nombres en formato string de la sección anterior, vinculando cada número con su nombre en el mismo orden en que los *SWI* fueron programados.

  En este caso, *SWI 0* recibe el nombre *Timer_SetToZero* y *SWI 1* se denomina *Timer_ReturnTime*.

- **Tabla de ayuda y decodificación de comandos**

  Este apartado permite la creación de diferentes comandos, ejecutables desde *TaskWindow*. Para ello es necesario establecer una tabla, denominada *function table*, semejante al *module header*. A través de ella se les proporciona un nombre, se puede crear un texto de ayuda y de sintaxis y se especifica el código a ejecutar.

En este caso, se programaron dos comandos:

- Counter: retorna el valor del contador en minutos y segundos, desde que se carga el módulo.
- Restart: resetea el valor del contador.

La *tabla de ayuda y decodificación de comandos* sigue unas reglas determinadas:

- Al comienzo de su ejecución, R0 contiene un puntero al final del código, R1 el número de parámetros que se le van a proporcionar al comando, R12 contiene la dirección de un puntero al *private word*, R13 pasa a ser R13_svc y R14 contiene la dirección de retorno.
- A la salida, R0 debe contener un *error pointer*, en caso de que lo hubiera y R7-R11 deben ser preservados.
- Durante la ejecución, IRQ y FIQ se encuentran habilitados y el procesador está en modo *supervisor*.

- **Manejador de interrupciones**

Anteriormente se ha programado un *driver* para el control del reloj del sistema. Como consecuencia, se ha implementado la generación de interrupciones, que es necesario procesar mediante un manejador de excepciones específico.

En el momento en el que se produzca una interrupción del reloj del sistema, el sistema operativo permitirá que el control pase a la rutina de atención que se muestra a continuación. Esta tarea fue programada en el apartado de inicialización utilizando *OS_ClaimDeviceVector*. Dado que se van a utilizar *Software Interrupts*, es necesario almacenar ladirección de retorno R14_irq en la pila, para que el programa pueda retornar[21].

En el código se realizan tres funciones principalmente:

1. Cambio del modo *interrupt* al modo *supervisor*, guardando su estado en la pila.
2. Eliminación de la notificación de la interrupción en el registro *Control and Status* del periférico.
3. Eliminación de la interrupción del reloj del sistema.
4. Eliminación de la interrupción IRQ[22].
5. Desactivación de las interrupciones de manera general, para que no se produzcan nuevas mientras se está ejecutando el manejador de excepciones.
6. Incremento del contador contenido en el *workspace*[23]
7. Activación de las interrupciones de forma general.
8. Restauración del estado IRQ y fin del manejador de excepciones.

- **Configuración del módulo en lenguaje BASIC (II)**

Para finalizar, es necesario configurar el formato del programa. Los módulos en RISC OS tienen la extensión *FFA*, por lo que es necesario implementarlo en el código mediante BASIC. El código es el de la Figura 19:

```
      ]
      NEXT
OSCLI "SAVE "+mod$+" "+STR$~C%+"+"+STR$~O%
OSCLI "SETT. "+mod$+" FFA"
END
```

Figura 19 - Configuración en BASIC del módulo (II)

[21] Los *Software Interrupts* se ejecutan en modo *supervisor,* por lo que si no se almacenase R14_irq su valor sería sustituido por el de R14_svc.
[22] Si se ejecutase *HAL_IRQClear* antes de *HAL_TimerIRQClear,* el controlador *IRQ* vería que el reloj todavía presenta activa la interrupción, con lo que el manejador de excepciones volvería a ejecutarse, produciendo dos interrupciones por *tick* en lugar de una.
[23] En esta ocasión, R12 contiene la dirección del *private word,* ya que se le ha proporcionado al *driver* mediante la llamada al sistema *OS_ClaimDeviceVector,* no porque se lo proporcione el módulo de por sí (como en los casos anteriores).

# 5 BARE METAL

La *Raspberry Pi* ofrece una amplia variedad de posibilidades para su programación. A lo largo del desarrollo de este proyecto se descubrió la capacidad de interactuar directamente con el *hardware*, sin la supervisión de un sistema operativo. Este hecho facilita enormemente las cosas, al no tener que realizar un estudio previo del sistema para su programación, como ocurrió con RISC OS. Sin embargo, también las complica, al no poseer herramientas adecuadas para su depuración y tener que recurrir a elementos físicos. Este tipo de programación recibe el nombre de *Bare Metal*.

## 5.1 *Introducción*

*Bare Metal* consiste, tal y como su propio nombre indica, en la programación sobre el *hardware* desnudo, explotando todas sus virtudes. De esta manera el alumno podría diseñar su propio sistema operativo, en función de sus necesidades. En este capítulo se va a introducir al estudiante en dicho concepto y se le van a presentar diferentes herramientas tanto para su compilación como para su depuración, así como un ejemplo práctico. Existen varios tutoriales acerca de programación *Bare Metal* en [11], con ejemplos muy descriptivos. [12] y [18] recogen varios códigos de este tipo de programación.

## 5.2 *Compilación*

La compilación y enlace entre los diferentes programas se realiza en un entorno de compilación cruzado llamado *GNU ARM toolchain.* Para ello se diseña un archivo *Makefile* en el que se especifican los programas de entrada y el archivo *imagen* de salida, que contendrá el sistema operativo a ejecutar por la *Raspberry Pi.*

### 5.2.1 Elección de Sistema Operativo

Aunque existen paquetes especiales para la ejecución de archivos *Makefile* en *Windows*, como *Yagarto*, diferentes pruebas sobre él demostraron que no era muy cómodo y generaba bastantes problemas. Se recomienda, por tanto, el empleo de *Linux* para la realización de la compilación.

### 5.2.2 Desarrollo de un archivo *Makefile*

Los archivos *Makefile* son bastante comunes en la programación ya que facilitan enormemente el proceso de compilación, pudiendo agrupar varios comandos en un mismo documento. En ellos se especifica el tipo de herramienta de compilación y los directorios donde se encuentran los archivos de entrada y de salida.

La estructura de este tipo de archivos se puede resumir de la siguiente manera, se puede encontrar en [12].

- Las etiquetas *target* contienen *dependencies* y en la línea siguiente, código a ejecutar. Los *dependencies* hacen referencia a otros *targets*.
- En el momento que se ejecute el comando *make all* desde la ventana de comandos, el archivo *Makefile* va proceder a realizar saltos consecutivos desde el primer *target* a los *dependencies* que contiene, de tal manera que se van a ir concatenando. Cuando se alcance el último *target*, se ejecutarán los comandos que comiencen en la siguiente línea separados por una tabulación, hasta alcanzar el *target* origen.

A continuación se recoge el *Makefile* que fue empleado en este proyecto, y que permite enlazar y compilar los programas ensamblador que se diseñen. La Figura 20 recoge el código para implementar la GNU ARM.

```
# The toolchain to use.
ARMGNU ?= arm-none-eabi
```

Figura 20 - Implementación del *GNU ARM toolchain*

Para facilitar la programación, se recogen todos los nombres de archivos y carpetas mediante variables, al inicio del código, como se puede observar en la Figura 21.

```
# The intermediate directory for compiled object files.
BUILD = build/

# The directory in which source files are stored.
SOURCE = source/

# The directory in which kernel.img file is stored.
GENERATED = generated/

# The name of the output file to generate.
TARGET = kernel.img

# The name of the assembler listing file to generate.
LIST = kernel.list

# The name of the map file to generate.
MAP = kernel.map

# The name of the linker script to use.
LINKER = kernel.ld
```

Figura 21 - Archivos y carpetas a utilizar

Todo *Makefile* debe contener un comando *all* con el funcionamiento básico, se representa en la Figura 22.

```
# Rule to make everything.
all: $(TARGET) $(LIST)
```

Figura 22 - Comando *all*

La Figura 23 recoge las líneas de código que permiten la creación de las carpetas destino, el ensamblado del código fuente y su enlazado en *kernel.img*.

```
# The names of all object files that must be generated. Deduced from the
# assembly code files in source.
OBJECTS := $(patsubst $(SOURCE)%.s,$(BUILD)%.o,$(wildcard $(SOURCE)*.s))

# Rule to make the image file.
$(TARGET) : $(BUILD)output.elf $(GENERATED)
    $(ARMGNU)-objcopy $(BUILD)output.elf -O binary $(GENERATED)$(TARGET)

# Rule to make the elf file.
$(BUILD)output.elf : $(OBJECTS) $(LINKER)
    $(ARMGNU)-ld --no-undefined $(OBJECTS) -Map $(MAP) -o $(BUILD)output.elf -T $(LINKER)

# Rule to make the object files.
$(BUILD)%.o: $(SOURCE)%.s $(BUILD)
    $(ARMGNU)-as -I $(SOURCE) $< -o $@

$(BUILD):
    mkdir $@

$(GENERATED):
    mkdir $@
```

Figura 23 - Compilación y enlace de los programas

En la carpeta destino se incluye el programa *desensamblado,* la Figura 24 recoge este proceso.

```
# Rule to make the listing file.
$(LIST) : $(BUILD)output.elf
    $(ARMGNU)-objdump -d $(BUILD)output.elf > $(GENERATED)$(LIST)
```

Figura 24 - Generación del archivo *kernel.img* desensamblado

Opcionalmente, se pueden añadir otros comandos. En este caso, se incluyó uno encargado de eliminar los archivos generados y así poder trabajar de una manera más ordenada, es el comando *clean* de la Figura 25.

```
# Rule to clean all files.
clean :
    -rm -rf $(GENERATED)
    -rm -rf $(BUILD)
    -rm -f $(MAP)
```

Figura 25 - Comando *clean*

Los archivos de salida se guardarán en la carpeta *generated*, y en ella se podrán encontrar la *imagen* del sistema operativo y un archivo *desensamblado* con el código programado. Este último será bastante útil en el caso de que se enlacen varios programas, pudiendo comprobar las direcciones de salto.

### 5.2.3 Archivos requeridos durante el proceso

En el proceso de compilación, aparte del código fuente y del *Makefile*, será necesario el archivo *Linker Script*. Este archivo indica al *bootloader* las diferentes regiones del archivo *imagen.* La más importante es la sección *.init*, que especifica la dirección *0x8000* y que va a ser cargada en primer lugar. En la sección *.text* se incluirá el resto del código y en *.data* las variables a almacenar. Dicha estructura se recoge en la Figura 26.

```
SECTIONS {
.init 0x8000 : {
  *(.init)
}
.text : {
  *(.text)
}
.data : {
  *(.data)
}
/DISCARD/ : {
   *(*)
 }
}
```

Figura 26 - Secciones en el archivo *imagen*

## 5.3 *Proceso de arranque*

A continuación, se detallan los archivos requeridos durante el proceso de arranque y de inicialización y cuáles son las etapas que la *Raspberry Pi* lleva a cabo en su ejecución.

La GPU adquiere un papel fundamental, al ser la encargada de inicializar el *bootloader* y de preparar a la CPU para la ejecución del archivo *imagen* que el usuario ha programado y compilado. La Figura 27 resume dicho proceso:

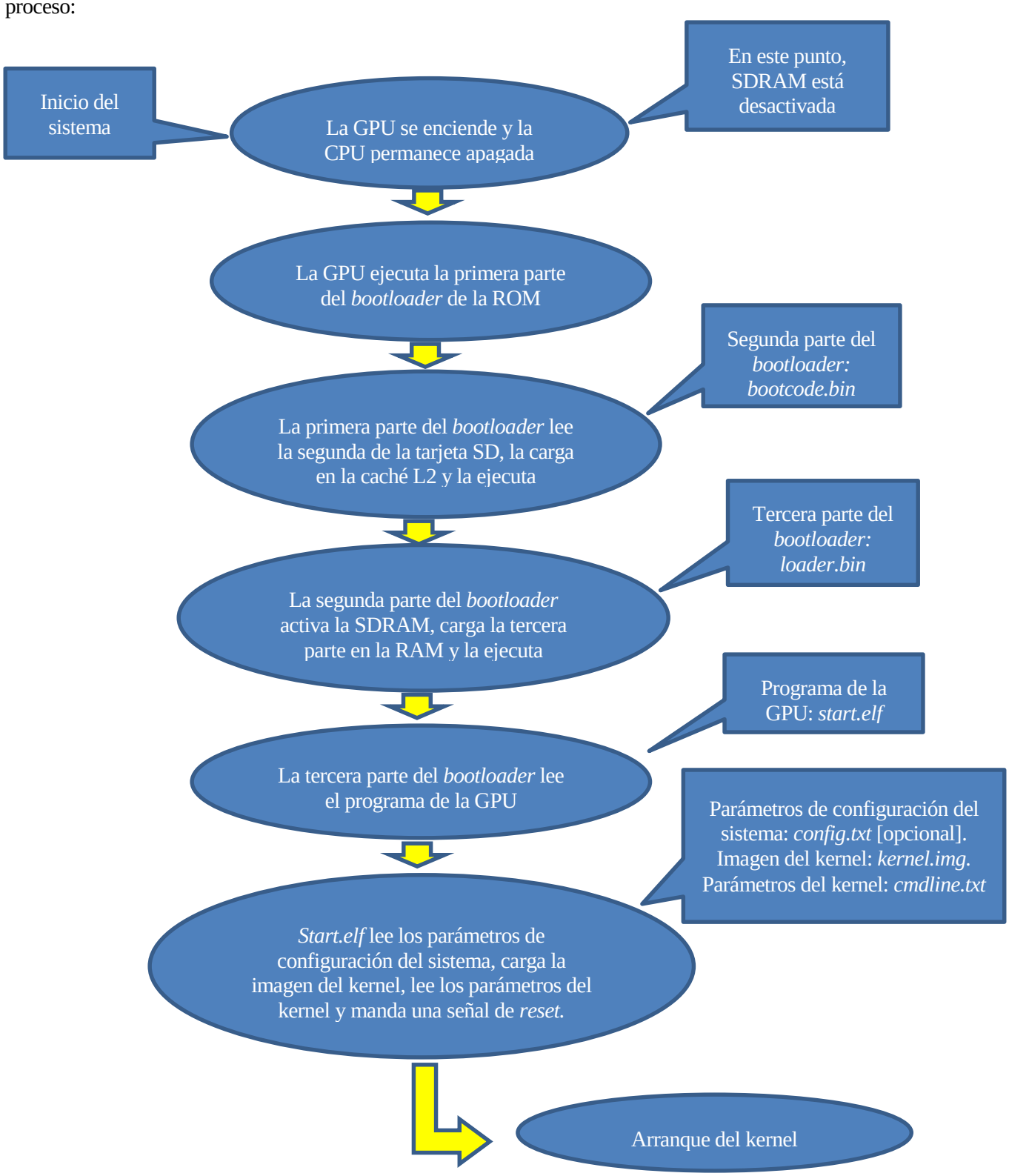



Figura 27 - Proceso de arranque de la *Raspberry Pi*

### 5.3.1 Archivos necesarios

- *Bootcode.bin*

Es el primer archivo cargado desde la tarjeta SD por la GPU. Contiene un código binario que es cargado en la caché L2 y activa la memoria SDRAM.

- *Loader.bin*

Permite cargar los archivos *elf (start.elf)*, que son de vital importancia en el proceso de arranque.

- *Start.elf*

Permite cargar un archivo denominado *kernel.img* en la memoria SDRAM en la dirección *0x8000*. Habilita ciertas opciones de configuración, presentes en el archivo *config.txt*. Algunas de estas opciones son la configuración de la resolución de vídeo y la frecuencia del procesador ARM.

- *Kernel.img*

Contiene los programas creados siguiendo el proceso de compilación de la Sección 5.2.2.

## 5.4 *Modificación de los vectores de excepción*

En el momento en el que la *Raspberry Pi* se inicia, carga el programa desde la tarjeta SD a la memoria RAM en la dirección *0x00008000*. Tal y como se comentó en el Capítulo 2 sobre la arquitectura ARM, los vectores de excepción contienen las direcciones de memoria donde localizar los manejadores de excepciones de cada tipo de excepción. Para poder modificarlos, es necesario acceder a sus posiciones de memoria en la RAM mediante el modo *supervisor* y reemplazar sus direcciones con las de las rutinas programadas por el propio usuario.

Este procedimiento se realiza con el código de la Figura 28:

```
        _start:
        ldr pc, _reset_h
        ldr pc, _undefined_instruction_vector_h
        ldr pc, _software_interrupt_vector_h
        ldr pc, _prefetch_abort_vector_h
        ldr pc, _data_abort_vector_h
        ldr pc, _unused_handler_h
        ldr pc, _interrupt_vector_h
        ldr pc, _fast_interrupt_vector_h

    _reset_h: .word   _reset_
    _undefined_instruction_vector_h: .word   undefined_instruction_vector
    _software_interrupt_vector_h: .word   software_interrupt_vector
    _prefetch_abort_vector_h: .word   prefetch_abort_vector
    _data_abort_vector_h: .word   data_abort_vector
    _unused_handler_h: .word   _reset_
    _interrupt_vector_h: .word   interrupt_vector
    _fast_interrupt_vector_h: .word   fast_interrupt_vector

    _reset_:
        mov     r0, #0x8000
        mov     r1, #0x0000
        ldmia   r0!,{r2, r3, r4, r5, r6, r7, r8, r9}
        stmia   r1!,{r2, r3, r4, r5, r6, r7, r8, r9}
        ldmia   r0!,{r2, r3, r4, r5, r6, r7, r8, r9}
        stmia   r1!,{r2, r3, r4, r5, r6, r7, r8, r9}
```

Figura 28 - Código para el reemplazo de los vectores de excepción

## 5.5 *Desarrollo de un contador*

Para demostrar el funcionamiento de esta técnica de programación de la *Raspberry Pi*, se diseñó un contador basado en interrupciones de salida del reloj del sistema y de entrada mediante un botón que lo reinicializaría. Podría dividirse en las siguientes partes:

1. Modificación del vector de excepciones específico de las excepciones *IRQ*. Se reemplazaría con la dirección de inicio del manejador de excepciones creado por el usuario.
2. Habilitación de las interrupciones del reloj del sistema y del pin de la GPIO al cual se le va a conectar el botón. Establecimiento por tanto, del período de las interrupciones del periférico temporal.
3. Habilitación de las interrupciones de manera general, con lo que las interrupciones del paso anterior comenzarán a tener vigencia.
4. Habilitación de los pines de la GPIO responsables de iluminar el *display* de siete segmentos como salida y del pin del botón como entrada.
5. Desarrollo de un manejador de excepciones *IRQ* encargado de analizar y atender las peticiones de las interrupciones. En el caso de que fuesen del reloj deberán incrementar un contador interno, fijar el siguiente período de interrupción y ejecutar una subrutina encargada de mostrar el tiempo en un *display* de siete segmentos. Si por el contrario fuesen del botón, deberá reiniciar el contador interno.
6. Desarrollo de una subrutina encargada de mostrar el valor del contador en el *display*. En primer lugar, obtendría su valor desde la posición de memoria de la sección *.data*. A continuación lo dividiría en minutos y segundos y posteriormente convertiría ambos números en las líneas correspondientes de la GPIO asociadas a los dígitos a representar. Finalmente, las activaría durante un período de tiempo determinado.

A la hora de acceder al hardware hay que destacar que se utilizarán las direcciones físicas. La MMU no se encuentra configurada por defecto, aunque existen tutoriales que explican como hacerlo en el caso de que se pretendan utilizar las direcciones virtuales. Por esta razón, para la programación se utilizó la dirección física *0x20003000* y no la dirección lógica *0x7E003000*.

El contador se basa en el fenómeno *POV (Persistence of Vision).* Trata de mantener encendidos los pines de la GPIO por un cierto tiempo (y no a la vez) y a una frecuencia tan alta que el ojo humano cree que está viendo los dígitos a la vez. El motivo de emplear esta técnica es por un lado el ahorro de hardware y por otro el evitar emplear una potencia que la *Raspberry Pi* es incapaz de proveer.

Más información acerca de este proceso se puede encontrar en [16] y [17].

El resultado de programar siguiendo los pasos descritos anteriormente, se puede observar en la Figura 29.

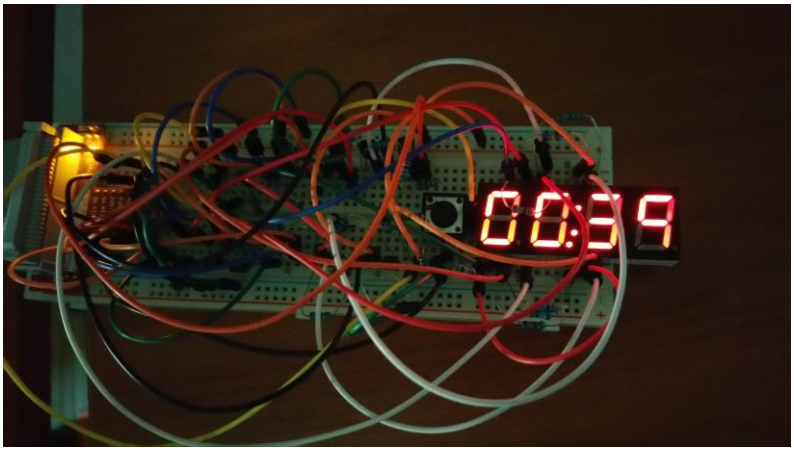


Figura 29 - Funcionamiento del contador

# 6 CONCLUSIÓN

El objetivo fundamental de este proyecto era realizar un estudio de las diferentes posibilidades para la renovación del laboratorio de microprocesadores de la Universidad de Cantabria. A partir de esta necesidad se buscaron alternativas y dispositivos capaces de satisfacerla. Las siguientes líneas tienen la finalidad de recapacitar sobre el trabajo descrito en las páginas previas.

Bautista Coto, en su proyecto fin de carrera del año 2003, argumentó el empleo de la arquitectura MIPS en el laboratorio mediante la placa desarrollada por *MyCable.* Se trataba de un dispositivo caro y limitado en cuanto a soporte y prestaciones. Sin embargo, se ajustaba a los requerimientos de nivel académico de estudiantes que procedían a iniciarse en este ámbito de la Ingeniería de Computadores.

La evolución tecnológica ha hecho mella en esta arquitectura, que se emplea cada vez menos debido al avance de otras con mayores posibilidades. El Capítulo 1 tuvo como objetivo recapacitar sobre la elección de la arquitectura *x86* ó *ARM*, siendo esta última finalmente la elegida por motivos energéticos y estadísticos respecto a su implantación en el mercado. Una vez elegida la arquitectura, se discutió sobre qué dispositivo podría ser el ideal para implementarse en el laboratorio. En la actualidad, son dos los dispositivos basados en ARM que más comúnmente se emplean para la realización de proyectos: *Raspberry Pi* y *Arduino.* Aunque ambos tienen un coste similar y el soporte no es significativamente mayor en ninguno de los dos, el primero presenta mayores ventajas en cuestión software. Dado que el fin es que los alumnos estudien y analicen el funcionamiento de esta arquitectura, fue la superioridad en el software lo que hizo a la *Raspberry Pi* la candidata perfecta para renovar el laboratorio de microprocesadores.

Asimismo, se realizó un análisis de ARM para entrever la dificultad de esta arquitectura y así poder asociarla con el nivel académico exigido por las asignaturas de computadores del Grado de Informática y del Grado en Ingeniería en Tecnologías de la Telecomunicación. En mi opinión, bajo el punto de vista de un alumno que ha cursado y estudiado la arquitectura MIPS, ARM es un salto bastante pronunciado no sólo a nivel de repertorio de instrucciones si no también en lo relacionado con la lógica interna del procesador. Es necesario una formación exhaustiva para comprender el funcionamiento de esta arquitectura.

A continuación, se buscaron dos plataformas para trabajar con este sistema. Por un lado un sistema operativo optimizado, llamado *RISC OS*, que permite compilar, ejecutar y depurar código ensamblador y C. Si bien es cierto que presenta limitaciones en lo que a la programación de su *kernel* se refiere, contiene herramientas muy útiles para una primera aproximación a ARM. Por otro lado se dispone de la posibilidad de trabajar directamente sobre el hardware, en una técnica conocida como *Bare Metal.* Este método es sin lugar a dudas el que permite un mayor aprovechamiento de los recursos de la *Raspberry Pi,* al no depender de un sistema operativo que impide, por ejemplo, acceder directamente a los periféricos y que es necesario estudiar previamente.

Para complementar los dos procedimientos, se realizaron dos ejemplos prácticos de lo que podría exigirse en una futura asignatura de la Universidad de Cantabria basada en ARM. En primer lugar, un *driver* del periférico del reloj del sistema. Permite ejecutar comandos en el terminal de *RISC OS* y programar códigos basados en sus funciones. En segundo lugar un contador, basado también en dicho periférico y que manifiesta la facilidad para acceder al hardware sin la oposición del sistema operativo. Su programación descansa sobre interrupciones tanto de entrada como de salida, relacionadas con el área de Estructura de Computadores.

Respecto a la técnica *Bare Metal*, existen multitud de tutoriales en Internet para la programación de las diferentes partes de la *Raspberry Pi.* Estos proyectos abarcan programas sencillos como encender un LED, hasta otros más complejos como la programación de la GPU o la creación de una consola semejante a la de un sistema operativo convencional. A través de estos tutoriales el alumno podría experimentar y comprender el funcionamiento de los sistemas operativos (vistos en otras asignaturas del grado) e incluso diseñar el suyo propio.

Para terminar, me gustaría concluir diciendo que este proyecto no reemplazaría el trabajo realizado por el Grupo de Arquitectura y Tecnología de Computadores. Podría formar parte de una asignatura destinada a estudiantes que deseen profundizar en las arquitecturas de procesadores más recientes. Dada la complejidad que atañe a ARM, son necesarios unos conocimientos mínimos en otras arquitecturas más sencillas, como

MIPS, para dar el paso al estudio de la que se propone en este documento. Entrando en la impartición de dicha asignatura, yo aconsejaría dividirla en tres partes. Por un lado una introducción exhaustiva a la arquitectura ARM, por otro la programación mediante *RISC OS* para compilar y depurar errores, asentando las bases de la teoría y finalmente la programación directa sobre el hardware a través de *Bare Metal.*

Una de las grandes ventajas de la *Raspberry Pi*, aparte de su soporte, material y versatilidad para la realización de proyectos de diversa índole, es su disponibilidad y su precio. Los alumnos tendrían a su disposición el dispositivo en sus casas y podrían beneficiarse de las ventajas de esta plataforma, además de disfrutar aprendiendo, que es otro de los fines, si no el más importante, del proceso formativo.

# Índice de figuras


Figura 1 - Ejemplo de *kit* de inicio de la *Raspberry Pi* 11
Figura 2 - Estructura del CPSR a nivel de bit [2] 16
Figura 3 - Formato de las instrucciones ARM 20
Figura 4 - Ejemplo de la técnica de adelantamiento. 23
Figura 5 - Compresión de una instrucción "add" a formato *Thumb* 25
Figura 6 - Expansión de una instrucción en formato *Thumb* a formato ARM 25
Figura 7 - GPIO 27
Figura 8 - Pines de la GPIO y su función 27
Figura 9 - Denominación común de los pines de la GPIO 28
Figura 10 - Diagrama de bloques de la GPIO 29
Figura 11 - Interrupt Pending Registers 30
Figura 12 - Estructura de código BASIC para la programación ensamblador 33
Figura 13 - Ejemplo de *tabla de decodificación de SWI* 41
Figura 14 - Configuración en BASIC del módulo 43
Figura 15 - Module Header 43
Figura 16 - Código de inicialización (I) 44
Figura 17 - Código de inicialización (II) 44
Figura 18 - Código de inicialización (III) 44
Figura 19 - Configuración en BASIC del módulo (II) 46
Figura 20 - Implementación del *GNU ARM toolchain* 47
Figura 21 - Archivos y carpetas a utilizar 48
Figura 22 - Comando *all* 48
Figura 23 - Compilación y enlace de los programas 48
Figura 24 - Generación del archivo *kernel.img* desensamblado 49
Figura 25 - Comando *clean* 49
Figura 26 - Secciones en el archivo *imagen* 49
Figura 27 - Proceso de arranque de la *Raspberry Pi* 50
Figura 28 - Código para el reemplazo de los vectores de excepción 51
Figura 29 - Funcionamiento del contador 52
Figura 30 - Registro *GPSEL1* 65
Figura 31 - Establecimiento del pin 16 como puerto de salida 65

Figura 32 - Activación del pin 16 65
Figura 33 - Desactivación del pin 16 65
Figura 34 - Activación de las interrupciones generales y del *System Timer Compare 1* 65
Figura 35 - Establecimiento del siguiente período de interrupción del reloj del sistema. 66
Figura 36 - Habilitación de las interrupciones de GPIO9 66
Figura 37 - Proceso de *clear interrupt* 66
Figura 38 - Proceso de identificación de la interrupción 66
Figura 39 - Conexión física para la realización del contador 67
Figura 40 - Ventana generada al descargar VNC 68
Figura 41 - Directorio con los archivos VNC 69
Figura 42 - Proceso de instalación de VNC 69
Figura 43 - Programa cliente VNC 69
Figura 44 - Guardando un archivo en formato BASIC 70
Figura 45 - Proceso de carga y compilación de un programa ensamblador 71
Figura 46 - Establecimiento de un *breakpoint* 71
Figura 47 - Ejecución de un programa ensamblador 71
Figura 48 - Muestra del contenido en memoria 71
Figura 49 - Modificación de un word en memoria desde *TaskWindow* 71
Figura 50 - Ejecución del comando desensamblador 72
Figura 51 - Ejemplo de *OS_ReadC* 72
Figura 52 - Ejemplo de *OS_WriteC* 72
Figura 53 - Ejemplo de empleo del comando *SYS* 74

# Índice de tablas


Tabla 1 - Características *Mycable XXS1500* 9
Tabla 2 - Comparación *Arduino* y *Raspberry Pi* 11
Tabla 3 - Características generales de los diferentes modelos de *Raspberry Pi* 12
Tabla 4 - Tipos de excepciones 18
Tabla 5 - Registros en ARM y su función 21
Tabla 6 - Convenio ARM 22
Tabla 7 - Ejecución de un "load" y una instrucción después de dos ciclos 23
Tabla 8 - Ejecución de tres instrucciones en las que se repite un registro como argumento 24
Tabla 9 - Registros del reloj del sistema 27
Tabla 10 - Registros del controlador de la GPIO (I) 28
Tabla 11 - Registros del controlador de la GPIO (II) 28
Tabla 12 - Registros de control de interrupciones 30
Tabla 13 - Recopilación de las interrupciones utilizables 31
Tabla 14 - Flags de condición 61

# Bibliografía


[1] *ARM Arquitecture Reference Manual*, Copyright ©1996-1998, 2000, 2004, 2005 ARM Limited.

[2] *ARM1176JZF-S™ Technical Reference Manual* Copyright © 2004-2009 ARM Limited.

[3] *Raspberry Pi, Assembly Language. RISC OS Beginners*. Bruce Smith. First Edition: 05 February 2014.

[4] *Raspberry Pi, RISC OS System. Programming Revealed.* Bruce Smith. First Edition.

[5] *BCM2835 ARM Peripherals.* © 2012 Broadcom Corporation.

[6] Página oficial de documentación ARM *http://infocenter.arm.com/*

[7] Página oficial de *Raspberry Pi* *http://www.raspberrypi.org/*

[8] Página oficial de *Embedded XINU* *http://xinu.mscs.mu.edu/*

[9] *Programmer's Reference Manual* *http://www.riscos.com/support/developers/prm/*

[10] *RISC OS Software* *http://www.heyrick.co.uk/*

[11] *Bare Metal* tutorial de Cambridge *http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/*

[12] *Bare Metal y otro software* para *Raspberry Pi* *http://www.valvers.com/open-software/raspberry-pi/*

[13] Especificaciones eléctricas de la *Raspberry Pi*

*http://www.mosaic-industries.com/embedded-systems/microcontroller-projects/raspberry-pi/gpio-pin-electrical-specifications*

[14] Información de *Arduino* y *Raspberry Pi* *http://www.xataka.com/gadgets/*

[15] Información de *Arduino* y *Raspberry Pi*

*http://hacedores.com/arduino-o-raspberry-pi-cual-es-la-mejor-herramienta-para-ti/*

[16] Tutorial de *displays* de 7 segmentos para la *Raspberry Pi* *https://projects.drogon.net/7-segment-led-display-for-the-raspberry-pi/*

[17] Tutorial de *displays* de 7 segmentos para la *Raspberry Pi* *http://devel.meruelo.com.es/blog/?p=397*

[18] *Software* para la *Raspberry Pi* *https://github.com/dwelch67/raspberrypi*

[19] Página oficial de *Arduino* *http://www.arduino.cc/*

[20] *Software* para realizar esquemáticos con la *Raspberry Pi o Arduino* *http://fritzing.org/home/*

[21] Información sobre ARM

http://access.ee.ntu.edu.tw/course/SoC_Lab_981/handout/W3_090930_ARM%20Processor%20Architecture.pdf

[22] APIs de RISC OS *https://www.riscosopen.org/wiki/documentation/show/List%20of%20HAL%20APIs*

[23] Herramienta para la creación de imágenes de SO *http://sourceforge.net/projects/win32diskimager/*

[24] Servidor *VNC* de descarga para *RISC OS* *http://www.phlamethrower.co.uk/riscos/vnc_serv.php*

[25] Cliente *VNC* de descarga para *Windows* *https://www.realvnc.com/download/vnc/*

[26] Página de compra de *Raspberry Pi* y material adicional. *http://www.ebay.es/*


*Todas las direcciones URL fueron comprobadas y validadas el 25 de junio de 2015*

# 7 APÉNDICE

## 7.1 *Instrucciones ARM*

En las siguientes páginas se procede a hacer una descripción de las instrucciones ARM más comunes, haciendo una introducción a las flags de condición (Tabla 14):

| **Sufijo** | **Significado** | **Flags** |
|---|---|---|
| EQ | Equal | Z=1 |
| NE | Not Equal | Z=0 |
| VS | Overflow Set | V=1 |
| VC | Overflow Clear | V=0 |
| AL | Always | |
| NV | Never | |
| HI | Higher | C=1 AND Z=0 |
| LS | Lower than or Same | C=0 OR Z=1 |
| PL | Plus clear | N=0 |
| MI | Minus Set | N=1 |
| CS | Carry Set | C=1 |
| CC | Carry Clear | C=0 |
| GE | Greater than or Equal | N=1, V=1 OR N=0, V=0 |
| LT | Less Than | N=1, V=0 OR N=0, V=1 |
| GT | Greater Than | N=1, V=1 OR N=0, V=0 AND Z=0 |
| LE | Less than or Equal | N=1, V=0 OR N=0, V=1 OR Z=1 |

Tabla 14 - Flags de condición

Las instrucciones más comunes se muestran a continuación clasificadas según su función:

*Instrucciones Aritmético/Lógicas*

**# ORR <R0>, <R0>, <R1>**

Realiza la operación lógica OR entre R0 y R1 y lo almacena en R0.

**# ORRS <R0>, <R0>, <R1>**

Realiza la operación lógica OR entre R0 y R1 y lo almacena en R0. Las flags de condición (N, Z, C) del CPSR se actualizan en función de los valores de esta operación ("S" está activado).

**# ORR <R0>, <R0>, <R1> LSL #1**

Realiza un desplazamiento hacia la izquierda de R1 (es decir, multiplica por dos su contenido) y posteriormente realiza la operación lógica OR entre R0 y R1 y lo almacena en R0.

**# ORRVS <R0>, <R0>, <R1>**

Realiza la operación lógica OR entre R0 y R1 y lo almacena en R0, pero únicamente si la "flag" de "overflow" (V) del CPSR está activada.

**# ORRPLS <R0>, <R0>, <R1> LSR #2**

Si la flag de condición (N) del CPSR está habilitada (el resultado de la operación anterior ha sido positiva o igual a cero), se procede a realizar un desplazamiento lógico a la derecha del contenido de R1 (dos posiciones, es decir, divide entre cuatro el contenido de R1) y posteriormente se emplea una "OR" lógica entre R0 y R1.

**# ORRMIS <PC>, <R0>, <R1> ASR #R2**

Si la flag de condición (N) del CPSR está deshabilitada (el resultado de la operación anterior ha sido negativa), se procede a realizar un desplazamiento lógico a la derecha del contenido de R1 un total de posiciones como las almacenadas en el registro R2. Posteriormente se emplea una "OR" lógica entre R0 y R1 almacena en PC. Como "S" está activado y el registro destino es el PC, se procede a hacer una copia del SPSR en el CPSR.

*Instrucciones de comparación*

**# CMP <R0>, <R1>**

Compara el contenido de R0 con el de R1. Para ello resta a R0 el contenido de R1 y actualiza el CPSR dependiendo del resultado (esto es, las flags de condición N, Z, C y V).

**# TEQEQ <R0>, <R1> LSL #1**

Solamente si la "flag" Z es cero, desplaza R1 hacia la izquierda una posición (multiplica por dos su contenido) y lo compara con R0. Realiza entre ellos una operación EOR (Rd(i) es 1 si R0(i) es diferente a R1(i)). Posteriormente actualiza las "condition flags" (N, Z, C). Esta operación se utiliza normalmente para conocer si dos valores tienen el mismo signo ("flag" N).

**# TSTAL <R0>, <R1> LSR #1**

En este caso, la etiqueta "AL" indica que la instrucción se ejecuta de forma incondicional (como si no estuviera puesta). Desplaza a R1 hacia la derecha una posición (divide entre dos su contenido) y lo compara con R0. Realiza entre ellos una operación AND. Posteriormente actualiza las flags de condición (N, Z, C). Esta operación se utiliza normalmente para conocer si un bit en concreto está activado en un registro ("flag" Z).

*Instrucciones Single Instruction Multiple Data (SIMD)*

**# SADD16 <R0>, <R1>, <R0>**

Realiza dos sumas paralelas con signo de halfs. El registro destino R0 estará constituido en su parte más significativa (bits (31:16)) por la suma de la parte más significativa de los bits (31:16) de R1 y R0. Si el resultado es mayor o igual a cero, los bits de la "flag" GE(1:0) del CPSR se actualizarán a "0b11" y si no a cero. Lo mismo ocurrirá con la suma de la parte menos significativa (bits 15:0) y la "flag" GE(3:2) del CPSR.

**# UHSUB8NE<R0>, <R1>, <R0>**

Solamente si la flag de condición Z está a cero, realiza ocho restas paralelas de cuatro bytes sin signo, y hace su media. En este caso no afecta a las "flag" GE. Las restas siguen este formato: R0(31:24) = R1(31:24) – R0(31:24), R0(23:16) = R1(23:16) - R0(23:16), R0(15:8) = R1(15:8) - R0(15:8), R0(7:0) = R1(7:0) - R0(7:0).

*Instrucciones de Multiplicación*

**# MLAS <Rd>, <Rm>, <Rs>, <Rn>**

Se trata de una multiplicación "Normal". Realizan la operación Rd = (Rm * Rs + Rn) [31:0]. Como "S" está activado se modifican las "flags" de CPSR ("N" y "Z") dependiendo del resultado.

**# MULLS <Rd>, <Rm>, <Rs>**

Se trata de una multiplicación "Normal". Solamente si la flag "C" está deshabilitado o "Z" está habilitado (condición "Unsigned lower or same"), realiza la operación Rd = (Rm * Rs)[31:0].

**# UMULL <RdLo>, <RdHi>, <Rm>, <Rs>**

Se trata de una multiplicación "Long". Realiza la multiplicación sin signo de Rm con Rs para producir un valor de 64 bits. Luego almacena los 32 bits más significativos en "RdHi" y los menos significativos en "RdLo".

**# SMLA<x><y> <Rd>, <Rm>, <Rs>, <Rn>**

Se trata de una multiplicación "Halfword". Mediante <x> seleccionamos el halfword más o menos significativo del primer operando "Rm" ("0" la menos y "1" el más) y con <y> lo mismo pero con el segundo operando <Rs>. Posteriormente los multiplica y le suma a este nuevo valor el contenido de <Rn>. El resultado lo almacena en <Rd>. Si se produce "overflow", se habilita el bit "Q" del CPSR.

*Instrucciones de uso diverso*

**# CLZ <Rd>, <Rm>**

Cuenta el número de ceros desde el bit 31 del registro <Rm> hasta el primer uno lógico y almacena el resultado en <Rd>. No actualiza las flags de condición del CPSR. Se suele emplear para normalizar el contenido de un registro, para ello después del CLZ se escribiría la siguiente instrucción: MOVS Rm, Rm, LSL Rd.

**# REV <Rd>, <Rm>**

Revierte el orden de los byte que conforman <Rm>. Sigue este formato: Rd(31:24) = Rm(7: 0), Rd(23:16) = Rm(15: 8), Rd(15: 8) = Rm(23:16), Rd(7: 0) = Rm(31:24).

**# PKHBT <Rd>, <Rn>, <Rm> {, LSL #<shift_imm>}**

Combina el halfword menos significativo del primer operando <Rn> con el más significativo del segundo operando <Rm> que se desplaza. Se trata de un shift hacia la izquierda, de una cantidad desde 0 hasta 31. El word resultante se almacena en <Rd>.

*Instrucciones sobre los Status Registers*

**# SETEND <endian_specifier>**

Modifica el bit "E" del CPSR sin alterar los demás. Si <endian_specifier> es "BE", "E" se actualiza a uno lógico y se pasa a Big Endian. Si es "LE", "E" se actualiza a cero lógico y se pasa a Little Endian como método de ordenación en memoria.

**# CPS<effect> <iflags>**

Modifica los habilitadores de interrupción "A", "I" y/o "F" del CPSR, sin alterar los demás. El campo <effect> indica la intención que se tiene sobre esos bits (habilitar/deshabilitar), y con el campo <iflags> se selecciona la "flag" de interrupción.

*Instrucciones Load/store*

**# LDREQB <Rd>, <addressing_mode>**

Solamente si el bit "Z" del CPSR está deshabilitado, carga un byte de memoria (según el modo de direccionamiento fijado) en <Rd> y lo extiende con ceros hasta completar los 32 bit del registro.

**# LDRSH <Rd>, <addressing_mode>**

Carga un halfword de la dirección de memoria fijada según el modo de direccionamiento, lo extiende el signo hasta formar 32 bit y lo almacena en <Rd>.

**# STRD <Rd>, <addressing_mode>**

Almacena un par de registros ARM a dos direcciones word consecutivas de memoria. El primer registro tiene que ser par y el segundo, el impar inmediatamente posterior (por ejemplo, R10 y R11).

*Load and Store de Múltiples registros*

**# LDM <addressing_mode> <Rn>{!}, <registers_and_pc>^**

Carga los datos contenidos en sucesivas direcciones de memoria, en registros de propósito general (especificados en la instrucción) y en el PC. Además, el SPSR del modo actual se copia en el CPSR. Al finalizar la instrucción, se produce un "branch" a la dirección cargada en el PC. Es útil para retornar desde excepciones.

## 7.2 *Raspberry Pi*

- **GPIO**

En el código de las Figuras 31, 32 y 33 se va a proceder a establecer el pin 16 como puerto de salida, para a continuación encenderlo y apagarlo, pudiendo apreciar los resultados en la *Raspberry Pi.*

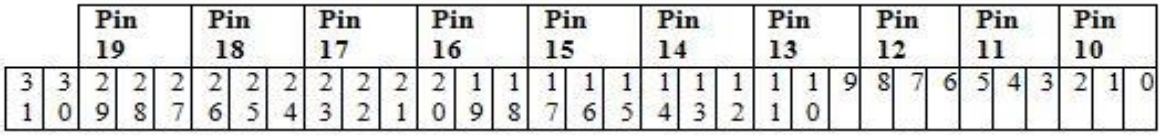



Figura 30 - Registro *GPSEL1*

```
ldr r0,=0x20200000 ; dirección de la GPIO
; Selección del PIN como puerto de salida.
mov r1,#1
lsl r2, r1, #18
str r2,[r0,#4]
```

Figura 31 - Establecimiento del pin 16 como puerto de salida

```
ldr r0,=0x20200000 ; dirección de la GPIO
; Set pin -> Lógica inversa.
mov r1, #1
lsl r1, r1, #16
str r1,[r0,#28]
```

Figura 32 - Activación del pin 16

```
ldr r0,=0x20200000 ; dirección de la GPIO
; Unset pin -> Lógica inversa.
mov r1, #1
lsl r1, r1, #16
str r1,[r0,#40]
```

Figura 33 - Desactivación del pin 16

Si a continuación se realizase un ciclo que continuamente activase y desactivase el pin 16, se podrá observar el parpadeo del LED correspondiente (*ACT*) en el dispositivo.

- **Reloj del sistema**

En la Figura 34 se pone de manifiesto el procedimiento para habilitar las interrupciones tanto del reloj del sistema como del sistema y su período.

```
    ldr r0,=0x2000B210 @ Enable_IRQs_1 */
/* Enable Compare Register 1 -> Enable_IRQs_1 */
    mov r1, #1
    lsl r1, r1, #1
    str r1,[r0]

/* Enable General Interrupts */
    mrs r0,cpsr
    bic r0,r0,#0x80
    msr cpsr_c,r0
```

Figura 34 - Activación de las interrupciones generales y del *System Timer Compare 1*

Para fijar el siguiente periodo de interrupción, es necesario obtener una referencia temporal actual y añadirla los *ticks* a los que se desee que se produzca la excepción. El código empleado fue el de la Figura 35.

```
ldr r0,=0x20003000 @ dirección del System Timer
ldrd r0,r1,[r0,#4] @ Obtener el CLO en r0
ldr r1, =1000000  @ Interval Period
add r0, r0, r1
ldr r1, =0x20003010   @ Compare Register 1
str r0,[r1]
```

Figura 35 - Establecimiento del siguiente período de interrupción del reloj del sistema.

- **Botón**

El código de las Figuras 36, 37 y 38 demuestra como habilitar las interrupciones del botón, conectado al GPIO9, y el detector de flanco de subida (este pin está a cero por defecto).

```
/* Fijo a 1 gpio_int[0] de IRQ_Enable2 */
ldr r0, =0x2000B214
mov r1, #1
lsl r1, r1, #17
str r1, [r0]

/* Fijo a 1 el GPIO9 GPHEN0 */
ldr r0, =0x20200064
mov r1, #1
lsl r1, r1, #9
str r1, [r0]
```

Figura 36 - Habilitación de las interrupciones de GPIO9

Para eliminar la interrupción una vez tratada en el manejador de excepciones *IRQ*, se aplicaron las siguientes líneas:

```
/* Elimino la interrupción en el GPEDS0 */
ldr r0, =0x20200040
mov r1, #1
lsl r1, r1, #9
str r1, [r0]
```

Figura 37 - Proceso de *clear interrupt*

Para realizar este apartado habrá que tener en cuenta los *pending registers*. En el manejador de excepciones se realizó esta comprobación con el asociado a las interrupciones de la GPIO (botón).

```
ldr r0, =0x2000B208 @ IRQ pending 2
ldr r1, [r0]
cmp r1, #0x20000 @ gpio_int[0]
bne Timer @ Si no es de botón, la interrupción es del timer
```

Figura 38 - Proceso de identificación de la interrupción

Su conexionado se puede apreciar en la Figura 39 y fue realizado con [20]:

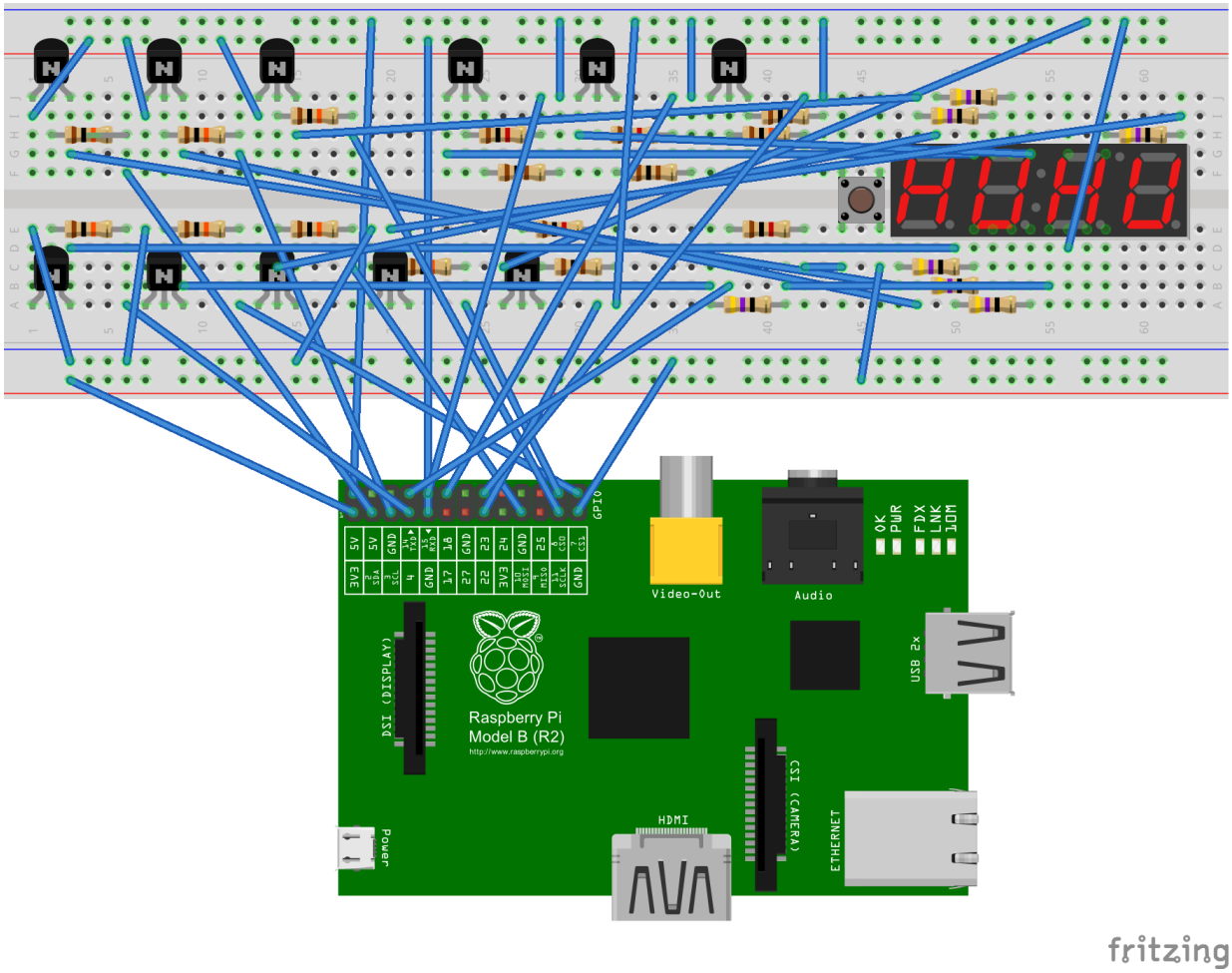



Figura 39 - Conexión física para la realización del contador

## 7.3 *RISC OS*

### 7.3.1 Instalación

Para instalar el sistema operativo en la *Raspberry Pi* es necesario seguir una serie de pasos.

1. Se requiere una tarjeta SD de mínimo 2GB de capacidad.
2. Acceder a la página oficial [7] y descargar el SO.
3. Desempaquetar el SO.
4. Descargar [23] un conversor SO a imagen, en este caso *Win32*.
5. Utilizando el archivo ejecutable *.exe* del *Win32* convertir a imagen seleccionando la tarjeta SD previamente introducida.

#### 7.3.1.1 Configuración de la conectividad a Internet

RISC OS tiene la capacidad de proveer conexión a Internet a la *Raspberry Pi*. Las conexiones se realizan mediante cable Ethernet, ya que no dispone soporte WiFi por el momento. Para dotar de este servicio, basta con seguir los siguientes pasos:

1. En el directorio inicial (*SDFS::RISCOSpi.$)* se realiza doble click en la carpeta *!Configure.*
2. Se cliquea en el icono *Network* y posteriormente en *Internet.*
3. Se selecciona la opción *Enable TCP/IP Protocol Suite*, se cierra y se pulsa en *Save.* Al guardar, aparecerá una ventana preguntando si se desea reiniciar para guardar los cambios. Se acepta y al iniciar ya se dispondrá de conexión a Internet.

#### 7.3.1.2 Instalación de VNC

A finales de 2014 fue publicada una versión de VNC *(Virtual Network Computing)* especialmente diseñada para versiones de RISC OS 3.5 o superiores. Este programa evitará el depender de un teclado, ratón y monitor, pudiendo conectar con la *Raspberry Pi* desde un ordenador. A continuación se describen los pasos para instalarlo en la *Raspberry Pi*, de manera que se inicie cada vez que el dispositivo se ponga en funcionamiento

- El primer requisito fundamental es tener conexión internet desde la *Raspberry Pi*, tal como se explicó en el apartado anterior.
- Desde el navegador *NetSurf*, se realiza la búsqueda de la dirección [24]:
- Se cliquea en *vnc_serv.zip (83,590 bytes*), lo cual iniciará la descarga del programa VNC (Figura 40).

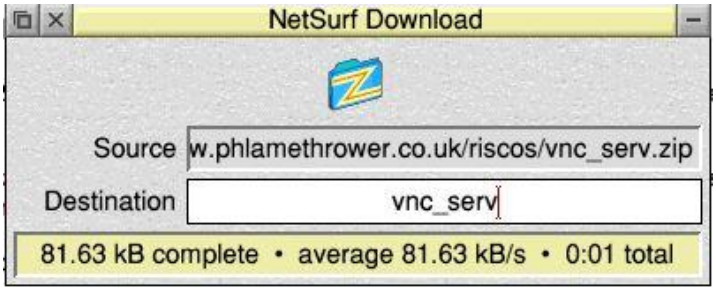



Figura 40 - Ventana generada al descargar VNC

- Los programas descargados no se almacenan en una carpeta específica como ocurre en Windows o Ubuntu. Para acceder a ellos simplemente hay que desplazarlos desde la ventana de *NetSurf Download* a la carpeta donde se pretendan conservar. Se recomienda guardarlos en la carpeta *Apps*.

- Dentro de la carpeta *Apps* se encontrarán los tres elementos principales, por un lado un módulo con el programa VNC y por otro dos archivos *Start* y *Stop*, que se van a encargar de configurar el puerto y la

contraseña, y de cargar y eliminar el módulo en el SO. Si se abre el archivo *Start*, se puede observar que vienen configurados por defecto el puerto 5900 y la contraseña 'bob'. El puerto no es necesario cambiarlo, funciona perfectamente. Los archivos se pueden apreciar en la Figura 41:

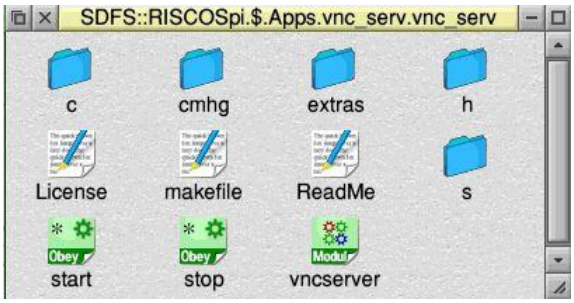



Figura 41 - Directorio con los archivos VNC

- Una vez realizados los pasos anteriores, se va a proceder a que el SO inicie el servidor cada vez que se inicie RISC OS. Para ello, dentro de la carpeta *!Boot* del directorio inicial *SDFS::RISCOSpi.$*, se abre la carpeta *Boot*, se selecciona *Run* y se arrastra el archivo *Start* dentro de la nueva pantalla que se ha generado. Finalmente se pulsa en *Set*, el proceso se recoge en la Figura 42.

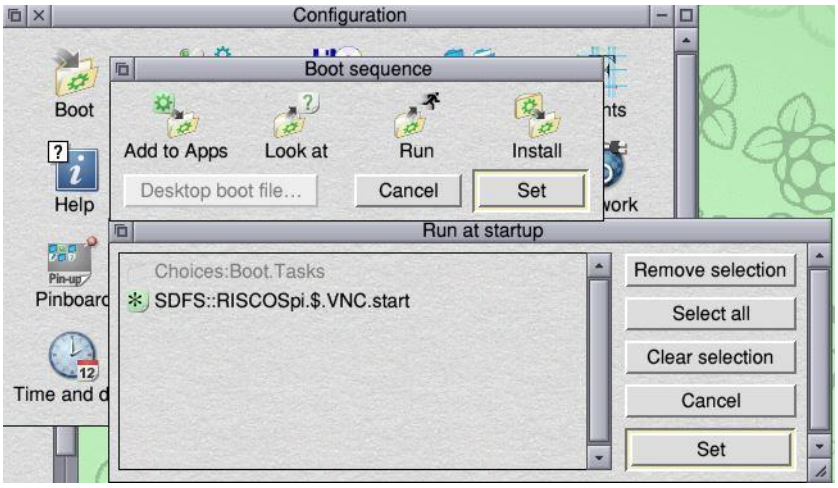



Figura 42 - Proceso de instalación de VNC

- A continuación, en la máquina controladora, es necesario descargarse un cliente VNC (*VNC Viewer)*. Para ello hay que acceder a la página oficial [25]. Existen versiones para prácticamente cualquier tipo de dispositivo, incluso hay versión Android. En mi caso descargué la compatible con Windows. En este momento, se escribe la dirección IP de la *Raspberry Pi* y tras teclear la contraseña del archivo *Start*, se establecerá la conexión y se podrá navegar por RISC OS de manera remota.

Figura 43 - Programa cliente VNC

- Para asegurar una estabilidad adecuada, se recomienda cambiar el parámetro *PointerEventInterval* de 0 a 10 dentro de las propiedades avanzadas del *VNC Viewer* (*Vnc Viewer*: opciones -> Avanzadas… -> Experto -> *PointerEventInterval).*

### 7.3.2 Instalación y compilación mediante GCC

#### 7.3.2.1 Instalación

La herramienta *GCC* se descarga[24] utilizando el programa*!PackMan*, disponible en el escritorio.

Lo principal es añadir una nueva fuente a la lista disponible en el programa. Para ello se cliquea en el icono, se pulsa *Advanced* y posteriormente *sources*, con lo que se podrán observar todos los paquetes instalados. Se pulsa en *add* y se añade el siguiente URL:

http://www.riscos.info/packages/pkg/autobuilt

Se cliquea en *OK*, *Save* y en *Close* para actualizar la lista de fuentes.

En este momento, ya se podrá descargar *GCC* desde *!PackMan.* Para ello, se abre el programa y mediante *scroll* se navega por la lista de programas hasta que se encuentre *GCC4.* Se pulsa en el nombre y en *Package GCC4>Install*. Una vez que se ha descargado, el programa aparecerá en el directorio *SDFS::RISCOSpi.$.Apps.Development.*

Antes de cerrar *!PackMan.*es necesario realizar el mismo proceso con *SharedLibs*, *SharedLibs-C* y *SharedUnixLibrary*. En el momento en el que los diferentes archivos hayan sido instalados, aparecerá un tick a la izquierda de su nombre.

A continuación, es necesario crear cuatro carpetas[25] dentro del directorio[26] *SDFS::RISCOSpi.$.Apps.Development.!GCC.Examples.C*, con los nombres *c*, *h*, *o* y *s.*

#### 7.3.2.2 Preparación para la compilación

*GCC* requiere de 6Mb de memoria para su ejecución. Para proporcionárselos, se pulsa en el icono de la *Raspberry Pi* localizado en el margen inferior derecho y se amplia la longitud de la tarea *Next* hasta superar los 6000Kb.

A continuación, es necesario activar las variables de entorno para compilar con *GCC.* Esta tarea se consigue pulsando directamente en el icono *!GCC* sin combinarlo con la tecla *shift.*

Los dos pasos previos son fundamentales para realizar la compilación, y deberán realizarse siempre que se inicie la *Raspberry Pi.*

#### 7.3.2.3 Ejemplo de depuración

A continuación, se procederá a realizar un ejemplo de depuración tal y como se explicó en el apartado correspondiente en la memoria.

En este caso se va a realizar la compilación con BASIC.

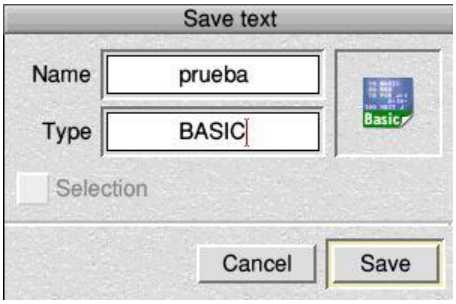



Figura 44 - Guardando un archivo en formato BASIC

[24] Para la instalación de GCC es imprescindible tener conexión a Internet, tal y como se describió en apartados anteriores.
[25] Para crear una carpeta, se cliquea en el directorio con la tecla *scroll* del ratón y se selecciona la opción *New directory*, escribiendo a continuación el nombre de la carpeta. Otra opción es escrbir el comando **cdir folder* en la ventana *TaskWindow.*
[26] Para acceder a este directorio, se debe pulsar en el icono *!GCC* del programa junto con la tecla *Shift.*

En el siguiente código el objetivo era mostrar una “P” por pantalla.

El proceso de carga y compilación se puede observar en la Figura 45:

```
)*LOAD prueba
ARM BBC BASIC V version 1.57 (C) Acorn 1989

Starting with 651516 bytes free

)RUN
00008FB4
00008FB4
00008FB4                    OPT pass
00008FB4          .start
00008FB4 E3B00050           MOVS R0, #ASC("P")
00008FB8 EF000000           SWI "OS_WriteC"
00008FBC E1A0F00E           MOV R15, R14
00008FC0                    ALIGN
```

Figura 45 - Proceso de carga y compilación de un programa ensamblador

Como prueba, se realizó un *breakpoint* después de #MOVS para comprobar que el valor ASCII de P (0x50) se cargaba correctamente en R0

Para ello se indica la dirección de memoria de la siguiente instrucción a la interesada:

```
)*BREAKSET 8FB8
```

Figura 46 - Establecimiento de un *breakpoint*

Después de ejecutar el código de la manera habitual, el resultado es el de la Figura 47:

```
)CALL start
Stopped at break point set at &00008FB8
Register dump (stored at &200093B0) is:
R0  = 00000050 R1  = 19191919 R2  = 193A1919 R3  = 45401940
R4  = 450E4516 R5  = 45450E45 R6  = 1919450E R7  = 1919193A
R8  = 00008700 R9  = 000A7FD8 R10 = 00000000 R11 = 00008100
R12 = 00008208 R13 = 000A7FC0 R14 = FC1A51DC R15 = 00008FB8
Mode USR32 flags set: nzCvqjggggeAift       PSR = 20000110
Debug*
```

Figura 47 - Ejecución de un programa ensamblador

Como se puede apreciar al realizarse el *halt* en la dirección especificada, R0 se ha cargado con el valor 50 y la “carry flag” se ha habilitado (como consecuencia de #MOVS). Los demás registros tienen valores aleatorios ya que no se han inicializado previamente.

Para mostrar el contenido de memoria a partir de la dirección *0x8FB4*, se aplica el comando de la Figura 48:

```
P)*MEMORY 8FB4

Address  :     7 6 5 4     B A 9 8     F E D C
00008FB4 :    E3B00050    EF000000    E1A0F00E
```

Figura 48 - Muestra del contenido en memoria

Para modificar el contenido de la dirección de memoria *0x8FB4* a 0x5, se aplica el comando de la Figura 49. Aparece el valor del contenido previo y el actualizado.

```
)*MEMORYA 8FB4 5
Word at &00008FB4 was &E3B00050 altered to &00000005
```

Figura 49 - Modificación de un word en memoria desde *TaskWindow*

Se va a proceder a desensamblar el código. Como se puede apreciar, aparte de mostrarse el código máquina de cada posición de memoria, se muestran los caracteres en hexadecimal y el identificador de los SWI en formato texto (no hexadecimal). Ciertas instrucciones no serán reconocidas.

```
)*MEMORYI 8FB4 8FC0
00008FB4 : P.°ã : E3B00050 : MOVS    R0,#&50          ; ="P"
00008FB8 : ...ï : EF000000 : SWI     OS_WriteC
00008FBC : .à á : E1A0F00E : MOV     PC,R14
```

Figura 50 - Ejecución del comando desensamblador

### 7.3.3 Software Interrupts (SWI)

A continuación se recogen algunas de las instrucciones *Software Interrupts* que han sido utilizadas en la realización de este proyecto.

- OS_ReadC (SWI *&04*)

  Pide y almacena el carácter pulsado por el usuario en el registro R0 y resetea la "carry flag". Si el carácter es incorrecto (ESCAPE), almacena su código en R0 y además activa la bandera de acarreo.

```
# SWI "OS_ReadC" ; Lectura de dato por teclado
# COMP R0, #&1B  ; "&1B" Escape ASCII code
# BEQ EscapePressed ; Salto a una dirección concreta
```

Figura 51 - Ejemplo de *OS_ReadC*

- OS_ReadLine (SWI *&0E*)

  Permite la recepción por teclado de un texto con una longitud máxima, su almacén en una zona específica de memoria y ciertas restricciones respecto a los caracteres aceptados. La entrada de datos finaliza con la pulsación de la tecla ENTER.

  R0 contendrá la dirección de memoria a almacenar el string, R1 el número máximo de caracteres y R2 y R3 el mínimo y máximo código ASCII aceptado respectivamente.

- OS_WriteC (SWI &0)

  Permite mostrar un único carácter por pantalla. R0 contiene el código ASCII de la tecla y la rutina no devuelve ningún valor. Se puede observar en la Figura 52.

```
# MOV R0, #ASC("0") ; Carga de un asterisco in R0
# SWI "OS WriteC" ; Muestra por pantalla del carácter
```

Figura 52 - Ejemplo de *OS_WriteC*

- OS_Write0 (SWI &2)

  Muestra un string por pantalla. Puede tener cualquier tamaño, pero tiene que finalizar en el carácter nulo. R0 contiene la dirección del string.

- OS_NewLine (SWI &3)

  Ejecuta un carriage return por pantalla.

- OS_ConvertCardinal4 (SWI &D8)

  Convierte un número a string. Recibe en R0 el valor, en R1 un puntero a un buffer donde se almacenará el string y en R2 el tamaño del buffer. A la salida, R0 contendrá la dirección de inicio del buffer, R1 la de su fin y R2 el número de bytes libres en él.

- OS_SWINumberToString (SWI &38)

  Permite convertir el número identificativo de un SWI en su nombre completo. Recibe en R0 el número de SWI, en R1 un puntero al buffer donde se almacenará el string y en R2 el tamaño del buffer. Retorna en R2 el tamaño del string en el buffer.

- OS_Hardware (SWI &7A)

  Provee acceso a la capa hardware (*HAL*). Tiene diferentes funciones dependiendo del valor que recibe a través de R8. Si es 0, permite ejecutar rutinas *HAL*. R9 indica el identificador de las subrutinas. Los resultados son retornados en R0.

  i. HAL_Timers[27] (12): indica la cantidad de timers disponibles en RISC OS para su utilización.

  ii. HAL_TimerDevice[28] (13): retorna el identificador global del número del reloj seleccionado mediante R0 para poder habilitar interrupciones.

  iii. HAL_TimerGranularity (14): indica el número de *ticks* necesarios para que el reloj indicado en R0 produzca una interrupción en un segundo.

  iv. HAL_TimerMaxPeriod (15): retorna el período máximo en ticks que es capaz de soportar el timer indicado mediante R0.

  v. HAL_TimerSetPeriod (16): establece mediante R1el período del reloj indicado en R0. Si se habilitan las interrupciones correspondientes, el reloj producirá una al final de cada período. Esta *SWI* no retorna nada.

  vi. HAL_TimerPeriod (17): indica el período actual en ticks del reloj correspondiente establecido mediante R0.

  vii. HAL_TimerReadCountdown (18): indica el número de ticks restantes para llegar al final del período del reloj especificado en R0 mediante su número, en el momento en que se ejecute esta instrucción.

  viii. HAL_IRQEnable (1): habilita las interrupciones IRQ de un dispositivo mediante su identificador global, indicado en R0. Si retorna un cero, las interrupciones de dicho dispositivo estaban desactivadas previamente.

  ix. HAL_IRQDisable (2): inhabilita las interrupciones IRQ de un dispositivo mediante su identificador global, indicado en R0. Si retorna un cero, las interrupciones de dicho dispositivo estaban desactivadas previamente.

  x. HAL_TimerIRQClear (113): desactiva las interrupciones del reloj especificado en R0 mediante su número. Para evitar efectos de latching en el manejador de excepciones del reloj deben desactivarse las interrupciones mediante HAL_TimerIRQClear y HAL_IRQDisable.

- OS_IntOn (SWI &13)

  Habilita las interrupciones de forma general, activando su posición correspondiente en el CPSR.

[27] Si se ejecuta esta *SWI* en el sistema operativo, retorna un dos. Sin embargo, únicamente el segundo estará disponible para el usuario. El primero lo emplea la GPU.
[28] Siguiendo la rutina anterior, para conocer el identificador del segundo reloj es necesario cargar en R0 un uno (el cero está vinculado para el primer reloj).

- OS_IntOff (SWI &14)

  Deshabilita las interrupciones del sistema de forma general, desactivando su posición correspondiente en el CPSR.

- OS_ClaimDeviceVector 29(SWI &4B)

  Permite instalar un driver del dispositivo seleccionado mediante su indentificador global en R0. La dirección del manejador de excepciones correspondiente se especifica en R1y la del *workspace* en R2.

- OS_ReleaseDeviceVector (SWI &4C)

  Libera al sistema operativo del driver instalado mediante OS_ClaimDeviceVector. Recibe sus mismos parámetros.

- OS_Module (SWI &1E): realiza una operación de módulo. El tipo de operación se define mediante R0.
    1. OS_Module 6: instancia un área RMA para la creación de un *workspace.* Recibe en R3 el tamaño de esta área en bytes, que debe estar alineado a word. Retorna en R2 la dirección del *workspace.*
    2. OS_Module 7: libera el área de memoria ocupada por un *workspace*, siendo R2 el puntero a su dirección de inicio. No retorna ningún parámetro.

- OS_Memory (SWI &68): permite realizar numerosas funciones en relación con la memoria. Dichas funciones se especifican mediante R0.
    1. OS_Memory 13: realiza un mapeo permanente, en tiempo de ejecución, de una dirección de memoria física establecida mediante R1y cuyo tamaño se especifica en R2. Retorna en R3 la dirección lógica asignada.
    2. OS_Memory 14: realiza un mapeo temporal en tiempo de ejecución, de la dirección física establecida en R1. R2 y R3 retornan respectivamente, la dirección lógica asignada y una llave de referencia necesaria para la liberación de memoria.
    3. OS_Memory 15: libera la memoria ocupada por *OS_Memory 14*, recibiendo en R1 la llave de referencia. No retorna ningún parámetro.

#### 7.3.3.1 SYS

A través de este ejemplo, se procederá a demostrar la practicidad del comando *SYS* para visualizar el contenido del *CLO* del reloj del sistema. Es necesario recordar la dirección física de dicho periférico (*0x2000300). S*in embargo, si se pretende acceder a él mediante esa dirección, se observará que ese espacio de memoria se encuentra vacío. Esto se debe a la labor de la MMU, que ha traducido esa dirección física a otra virtual.

Este dispositivo presenta una dirección virtual concreta (*0x7E003000),* pero se va a designar otra dirección mediante el empleo de un SWI específico, siguiendo el código de la Figura 53.

```
>DIM buffer% (20)
>SYS "OS_Memory", 13, &20003000, 32 TO buffer%
>PRINT buffer %
  F9003000
```

Figura 53 - Ejemplo de empleo del comando *SYS*

Mediante el empleo repetido del comando **MEMORY* (visto en el apartado de depuración) a la dirección retornada por el SWI *"OS_Memory"* se podrá comprobar cómo el contador se va incrementando conforme pasa el tiempo.

[29] Existe otra llamada al sistema con la misma función, denominada *OS_Claim* pero se encuentra obsoleta en el caso de IRQs.