Facultad
de
Ciencias

Desarrollo de un entorno de prácticas de laboratorio en ensamblador de ARM para un sistema Raspberry Pi con el sistema operativo RISC OS

(Development of a laboratory practice environment in ARM assembler for a Raspberry Pi system with the RISC OS operating system)

Trabajo de Fin de Grado
para acceder al

GRADO EN INFORMÁTICA

Autor: Elena Zaira Suárez Santamaría

Director: María del Carmen Martínez Fernández

Co-Director: Fernando Vallejo Alonso

Septiembre – 2017

RESUMEN

El estudio de la Estructura y Organización de los Computadores comprende una parte importante de la formación obligatoria de un graduado en Ingeniería Informática. Estas asignaturas pretenden desarrollar la capacidad de conocer, comprender y evaluar la estructura y arquitectura de los computadores, así como los componentes básicos que los conforman.

Para adquirir dichas capacidades, se estudia la arquitectura concreta de un procesador. Actualmente la Universidad de Cantabria, en el marco de un proyecto de innovación docente, se ha decido a realizar el cambio a la arquitectura ARM, una arquitectura cada vez más utilizada para la enseñanza en diversas Universidades.

Este proyecto, por lo tanto, está centrado en el análisis de cuál es el sistema hardware con arquitectura ARM más adecuado para sustituir los procesadores con arquitectura MIPS, que hay disponibles en el laboratorio docente. Para ello se investigará cuáles son las plataformas y dispositivos más adecuados para el aprendizaje de la arquitectura, teniendo en cuenta que el planteamiento inicial es que una misma arquitectura sea empleada en la mayor parte de los niveles de enseñanza de los fundamentos hardware de los computadores. No sólo en los niveles más bajos, sino también en aquellos aspectos del sistema operativo que están más cercanos al hardware.

Para empezar, analizaremos los diferentes modelos de Raspberry Pi disponibles y seleccionaremos el que más se aproxima a nuestros objetivos. Posteriormente se realizará un estudio del sistema operativo RISC OS, para determinar si es el adecuado para alcanzar ese doble objetivo anteriormente planteado. Se investigará cuáles son los compiladores disponibles, que nos permitan generar programas tanto en código ensamblador como C para RISC OS, así como un programa adecuado para depurar el código de nuestras prácticas.

Para asegurarnos de que todo lo anteriormente analizado cumple con lo esperado, se realizarán diversas prácticas en el nuevo laboratorio.

Palabras clave: ARM, RISC OS, arquitectura ARM, Raspberry Pi.

SUMMARY

The study of the Structure and Organization of Computers comprises an important part of the compulsory training of a graduate in Computer Engineering. These subjects aim to develop the ability to know, understand and evaluate the structure and architecture of computers, as well as the basic components that make them up.

To acquire these capabilities, we study the architecture of a processor. Currently the University of Cantabria, within the framework of a teaching innovation project, has decided to make the change to ARM architecture, an architecture increasingly used for teaching in various universities.

This project, therefore, is centered in the analysis of which is the hardware system with architecture ARM more suitable to replace the processors with architecture MIPS, that are available in the teaching laboratory. This will investigate which platforms and devices are most suitable for learning architecture, taking into account that the initial approach is that the same architecture is used in most of the teaching levels of the hardware fundamentals of computers. Not only at lower levels, but also in those aspects of the operating system that are closer to the hardware.

To begin, we will analyze the different models of Raspberry Pi available and select the one that is closest to our objectives. Subsequently, a study of the RISC OS operating system will be carried out, to determine if it is adequate to reach this double objective previously raised. We will investigate which compilers are available, which allow us to generate programs in both assembler and C code for RISC OS, as well as a suitable program to debug the code of our practices.

To ensure that everything discussed above meets what was expected, various practices will be carried out in the new laboratory.

Índice

1. INTRODUCCIÓN

La motivación principal de este proyecto es el cambio del laboratorio docente actual con procesadores MIPS, por un laboratorio centrado en el uso de arquitecturas más actuales. Permitiendo con este cambio, al igual que con los procesadores MIPS, desarrollar la capacidad de conocer y comprender la estructura y arquitectura de los computadores, así como mejorar la parte de prácticas del laboratorio, gracias al uso del sistema operativo y las herramientas disponibles para compilar y ensamblar, que permitirán al alumno centrarse principalmente en el aprendizaje de la arquitectura del computador. Para ello, se desea instalar en el nuevo laboratorio un sistema basado en Raspberry Pi, con procesadores ARM y el sistema operativo RISC OS.

1.1. OBJETIVOS

Este proyecto está centrado en el estudio de la plataforma de desarrollo y del sistema operativo que se desea implantar en el laboratorio para las asignaturas de Estructura de Computadores y Organización de Computadores. La plataforma Raspberry Pi y el sistema operativo RISC OS han sido seleccionados como los más indicados para su uso en el laboratorio. Analizaremos los diferentes modelos de Raspberry Pi disponibles, para ver cuál es el más indicado. Comprobaremos que las herramientas para compilar y depurar el código, disponibles para RISC OS, que se utilizarán en las futuras prácticas, funcionan correctamente, que el alumno no encontrará ninguna barrera y podrá centrarse en los conceptos importantes de la arquitectura del procesador. También se realizará un catálogo de los diferentes dispositivos que se podrían implementar en futuras prácticas de Entrada/Salida, así como su validación con la realización de diversos programas.

1.2. TRABAJO RELACIONADO

Actualmente hay varias universidades en España, que conozcamos, que están impartiendo clases basadas en el estudio de la arquitectura ARM, entre ellas se encuentran la Universidad Complutense de Madrid, Universidad de Cádiz, Universidad Rovira I Virgili, Universidad de Málaga, Universidad del País Vasco y Universidad Jaime I.

De las Universidades anteriores, hay tres de ellas a las que hemos podido tener acceso a información del tipo de laboratorio docente que utilizan:

La Universidad del País Vasco utiliza para realizar sus prácticas en el laboratorio, la máquina Nintendo DS, como se puede ver en su página web [32]. A parte del estudio del lenguaje ensamblador también usan el lenguaje en código C.

Por otro lado, tenemos a la Universidad Jaime I, que ha decidido el uso de un simulador de ARM (Qt ARMsim) junto con Arduino, como se puede deducir de la guía docente de la asignatura Estructura de Computadores [33].

Por último, la Universidad de Málaga, es una de las que más información disponemos en cuanto al tipo de laboratorio docente utilizado y ha sido un gran referente a lo largo de este proyecto [28]. Sus prácticas están centradas en el uso de la Raspberry Pi y usan dos entornos diferentes en función de la práctica a realizar. La programación tanto en lenguaje ensamblador como C la realizan en un entorno basado en Raspbian, este entorno no permite hacer drivers de E/S. Para las prácticas con los dispositivos de E/S utilizan programación en Bare Metal, se trabaja directamente con el hardware sin usar ningún sistema operativo ni depurador. Este uso de dos entornos diferentes y sus limitaciones, es lo que se ha tratado de evitar con la instalación de RISC OS, permitiéndonos realizar drivers de distintos dispositivos sin necesidad de cambiar de entorno, todo ello gracias al estudio de las diferentes llamadas al sistema implementadas en RISC OS.

2. ANÁLISIS DE HARDWARE

En este apartado nos centraremos en realizar un pequeño análisis de la arquitectura ARM, los registros de los que dispone, los modos de direccionamiento, instrucciones disponibles, así como los distintos modos en los que se puede encontrar el procesador. También analizaremos los diferentes tipos de Raspberry Pi disponibles, para poder más adelante, determinar cuál es el más adecuado para el laboratorio de prácticas.

2.1. ARQUITECTURA ARM

Se trata de una arquitectura de tipo RISC (Reduced Instruction Set Computing), en la que prima la eficiencia sobre el rendimiento. Muchos de los procesadores ARM se usan en sistemas embebidos y dispositivos portátiles. La información de este apartado, se trata de un resumen extraído de [27].

2.1.1. MODOS DE DIRECCIONAMIENTO

A continuación, mostraremos los modos de direccionamiento de ARM. Esta información ha sido extraída de [22].

Direccionamiento directo a registro, en este modo el operador se encuentra en el registro. Ejemplo: **`add r2, r3, r1`**

Direccionamiento inmediato, en este modo el operador se encuentra en la propia instrucción. Ejemplo: **`add r2, r3, #1`**

Direccionamiento relativo a registro con desplazamiento, en este modo la dirección efectiva del operando es una dirección de memoria que se obtiene sumando el contenido de un registro y un desplazamiento especificado en la propia instrucción. Ejemplo: **`str r2, [r3, #8]`**

Direccionamiento relativo a registro con registro de desplazamiento, en este modo la dirección efectiva del operando es una dirección de memoria que se obtiene sumando el contenido de dos registros. Ejemplo: **`ldr rd, [rb, ro]`**

Direccionamiento en las instrucciones de salto incondicional y condicional (direccionamiento absoluto), en el modo salto incondicional se usaría la instrucción **`b etiqueta`**, donde etiqueta es la dirección del salto, en el caso de salto condicional se usaría un operador condicional seguido de la etiqueta **`bxx etiqueta`**.

2.1.2. USO DE LOS REGISTROS

En ARM hay 31 registros de propósito general, más 6 registros de estado. Sólo 16 registros de propósito general son visibles en modo usuario (Tabla 2.2). Hay una convención de uso de registros llamada Procedure Call Standard (Tabla 2.1).

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Tabla 2.1: Procedure Call Standard [30].

usr sys	svc	abt	und	irq	fiq
r0					
r1					
r2					
r3					
r4					
r5					
r6					
r7					
r8					r8_fiq
r9					r9_fiq
r10					r10_fiq
r11 (fp)					r11_fiq
r12 (ip)					r12_fiq
r13 (sp)	r13_svc	r13_abt	r13_und	r13_irq	r13_fiq
r14 (lr)	r14_svc	r14_abt	r14_und	r14_irq	r14_fiq
r15 (pc)					
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Tabla 2.2: Registros en ARM de usuario y sistema [27].

El registro CPSR (Current Program Status Register) contiene bits que indican el estado del programa actual, e información acerca del resultado de operaciones anteriores. La Figura 2.1 muestra los bits del registro CPSR. Los primeros 4 bits, N, Z, C y V son los *condition flags*. La mayoría de las

instrucciones pueden modificar estas flags, y más tarde otras instrucciones usarlos para modificar sus operaciones. El significado de los *condition flags* es:

N (Negative): Este bit cambia a 1 si el resultado del signo de la operación es negativo.

Z (Zero): El bit cambia a 1 si el resultado de la operación es cero.

C (Carry): El bit cambia a 1 si se produce acarreo en el bit más significativo.

V (Overflow): Para sumas y restas, se cambia el bit si se produce desbordamiento con signo.

I: cuando se establece a 1, las interrupciones hardware normales son deshabilitadas.

F: cuando se establece a 1, las interrupciones rápidas son deshabilitadas

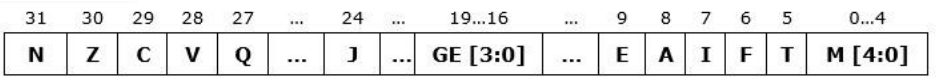


Figura 2.1: Registro CPSR [27]

El modo en el que se encuentra el procesador se selecciona escribiendo un patrón de bits en los bits del 0 al 4 del PSR (Program Status Register). Los distintos patrones se pueden ver en la Tabla 2.3.

M[4:0]	Modo	Nombre	Conjunto de registros
10000	usr	User	R0-R14, CPSR, PC
10001	fiq	Fast Interrupt	R0-R7, R8_fiq-R14_fiq, CPSR, SPSR_fiq, PC
10010	irq	Interrupt Request	R0-R12, R13_irq, R14_irq, CPSR, SPSR_irq, PC
10011	svc	Supervisor	R0-R12, R13_svc, R14_svc, CPSR, SPSR_irq, PC
10111	abt	Abort	R0-R12, R13_abt, R14_abt, CPSR, SPSR_abt, PC
11011	und	Undefined Instruction	R0-R12, R13_und, R14_und, CPSR, SPSR_und, PC
11111	sys	System	R0-R14, CPSR, PC

Tabla 2.3: Modelo Patron de bits en PSR [27]

2.1.3. ESTRUCTURA DE UN PROGRAMA EN ENSAMBLADOR

Un programa en ensamblador está formado por cuatro elementos básicos: directivas del ensamblador, etiquetas, instrucciones en ensamblador y comentarios.

Etiquetas: Las etiquetas se pueden usar en cualquier parte del programa para referirse a direcciones de datos, funciones o bloques de código. Siempre terminan con el carácter ':'.

Comentarios: Hay dos estilos, varias líneas seguidas en las que el comentario empieza con los caracteres /* y finaliza con */ o una única línea que empieza con el carácter '@'.

Directivas: Se usan principalmente para definir símbolos, asignar posiciones de almacenamiento y para controlar el comportamiento del ensamblador. Todas las directivas empiezan con un punto y las más importantes son *.data, .asciz, .text* y *.globl*

Instrucciones ensamblador: Se trata de las instrucciones que serán ejecutadas por la CPU. En el ANEXO hay un resumen de las operaciones más utilizadas.

2.2. RASPBERRY PI

La Raspberry Pi se trata de un pequeño computador que en función del modelo podran variar algunas de sus caracteristicas principales, como puede ser la disponibilidad de wifi, modelo de procesador, numero de puertos USB, etc. Actualmente disponemos en el mercado de 6 modelos diferentes de Raspberry Pi. Las caracteristicas e información de los mismos se encuentran en [26].

Raspberry Pi 1: Usa un SoC (System on Chip) Broadcom BCM2835. Hay dos modelos diferentes:

- **Modelo A+:** Remplaza el modelo original A en 2014. Dispone de 40 pines de GPIO, un adaptador para la tarjeta micro SD mejorado, menor consumo de energía, audio mejorado y un diseño de la placa mas funcional.

Figura 2.2: Raspberry Pi 1 A+ [26]

- **Modelo B+:** Remplaza el modelo B de Julio de 2014 de la Raspberry original. Dispone de:
    - Menor consumo de energía que versiones anteriores.
    - Diseño de la placa más funcional.

- 4 puertos USB
- 40 pines GPIO
- Puerto Full HDMI
- Puerto ethernet
- Conector de audio combinado de 3,5 mm y vídeo compuesto.
- Interface de camara (CSI)
- Interface display (DSI)
- Slot para micro SD
- VideoCore IV 3D graphics core
- 700MHz Broadcom BCM2835 CPU con 512MB RAM

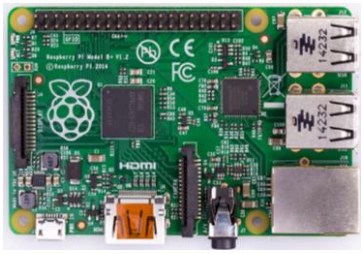


Figura 2.3: Raspberry Pi 1 B+ [26]

Raspberry Pi 2 modelo B: Se trata de la segunda generación de Raspberry Pi. Remplaza la Raspberry Pi 1 B+ en Febrero de 2015. Usa un SoC Broadcom BCM2836. En comparación con la Raspeberry Pi 1 tiene las siguientes mejoras:

- 1 CPU Cortex-A7 ARM de cuatro nucleos a 900 MHz.
- 1 Gb de RAM.

Debido a que dispone de un procesador ARMv7 puede ejecutar un amplio rango de distribuciones de GNU/Linux ARM, incluido *Snappy Ubuntu Core* así como *Microsoft Windows 10*.

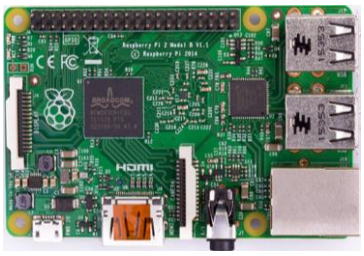


Figura 2.4: Raspberry Pi 2 B [26]

Raspberry Pi Zero: Tiene la mitad de tamaño que el modelo A+, usa un SoC Broadcom BCM2835. Dispone de:

- 1 CPU de un nucleo
- 512 Mb de RAM
- Puerto Mini-HDMI
- Puerto Micro-USB OTG
- Micro-USB power (Potencia Micro-usb)
- HAT-compatible 40-pin header
- Composite video y reset headers
- Conector cámara CSI (v1.3)

Figura 2.5: Raspberry Pi Zero [26]

Raspberry Pi Zero W: Salió al mercado a finales de febrero de 2017, tiene todas las funciones de las que dispone la Zero original pero le añade las siguientes mejoras:

- 802.11 b/g/n wireless LAN
- Bluetooth 4.1
- Bluetooth Low Energy (BLE)

Figura 2.6: Raspberry Pi Zero W [26]

Raspberry Pi 3 modelo B: Se trata de la tercera generación de Raspberry Pi. Remplaza la Raspberry Pi 2 B en Febrero de 2016. Usa un SoC Broadcom BCM2837. Comparada con el modelo 2 B, tiene las siguientes mejoras:

- 1 CPU ARMv8 de cuatro nucleos, 64 bits, a 1.2 Ghz
- 802.11n Wireless LAN
- Bluetooth 4.1
- Bluetooth Low Energy (BLE)

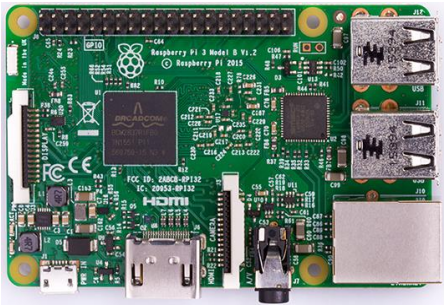


Figura 2.7: Raspberry Pi 3 B [26]

3. ANÁLISIS DE SOFTWARE

El tipo de software que se decida instalar en el laboratorio es una decisión importante, que debe de ser debidamente estudiada y analizada para evitar problemas en un futuro próximo. Para ello hemos realizado un estudio detallado del sistema operativo RISC OS, el cual consideramos es el más indicado para la realización de las prácticas en el laboratorio, ya que se trata de un sistema operativo ligero que permite desarrollar tanto prácticas básicas de nivel ensamblado (unido a C) como prácticas de E/S.

Otras herramientas software necesarias en nuestro laboratorio, son aquellas que nos permitan ensamblar nuestro código y enlazarlo correctamente para generar nuestros programas finales. Para ello analizaremos el compilador GCC disponible para RISC OS y cuáles son las herramientas de las que dispone que puedan sernos de utilidad.

También analizaremos los depuradores disponibles, su disponibilidad y utilidad a la hora de depurar los códigos generados para nuestras prácticas tanto en ensamblador, como en lenguaje C.

3.1. ANÁLISIS DEL TOOLCHAIN

Para poder realizar las prácticas correctamente será necesario el uso de un compilador. Para ello debemos encontrar un compilador que funcione correctamente, que nos permita crear programas que mezclen lenguaje C con lenguaje ensamblador ARM y que pueda ser instalado de forma rápida y sencilla en nuestro sistema operativo.

Para realizar las prácticas en el laboratorio se necesitará un ensamblador que nos permita traducir el código fuente en código objeto, permitiendo así a la máquina entender qué es lo que tiene que realizar [11]. Una vez que ya se tienen los objetos de nuestro programa, ya podemos pasar a enlazarlos para crear el ejecutable, para ello es necesario un enlazador o *linker*.

Por último, será necesario un buen depurador. Cuando programamos en lenguajes como el ensamblador, no disponer del mismo puede ser un problema, ya que trabajamos a ciegas y esto puede hacernos perder mucho tiempo innecesariamente.

3.2. RISC OS

Se ha decido instalar el sistema operativo RISC OS, ya que nos permite trabajar con programas creados tanto en lenguaje ensamblador como en lenguaje C, además de poder realizar E/S, en un entorno más amigable y didáctico. Se trata del único sistema operativo que esta íntegramente desarrollado en código ARM. Es sencillo de utilizar, recordándonos en ciertos aspectos, en cuanto a su manejo a nivel de usuario, a otros sistemas operativos más conocidos. A continuación, explicaremos unos conceptos básicos para poder empezar a trabajar en RISC OS, para una mayor información consultar [31].

3.2.1. INSTALACIÓN RISC OS

Para instalar RISC OS en la Raspberry Pi es necesario seguir los siguientes pasos. Se ha seguido como referencia en la siguiente sección el documento [13]

- Disponer de una micro SD (mínimo 2GB).
- Descargar la versión de RISC OS para Raspberry Pi en [6].
- Desempaquetar el SO.
- Descargar en [15] un conversor SO a imagen, por ejemplo, Win32.
- Utilizando el archivo ejecutable .exe del Win32, convertir a imagen seleccionando la tarjeta SD previamente introducida.

3.2.2. CONFIGURACIÓN INTERNET

Para configurar la IP (Figura 3.1):

- Doble clic en !Configure. 

- Clic en Network.
- Clic en Internet.
- Seleccionamos Interfaces, nos aparecerá una ventana donde seleccionaremos Configure. Aquí ya podremos configurar la dirección IP.

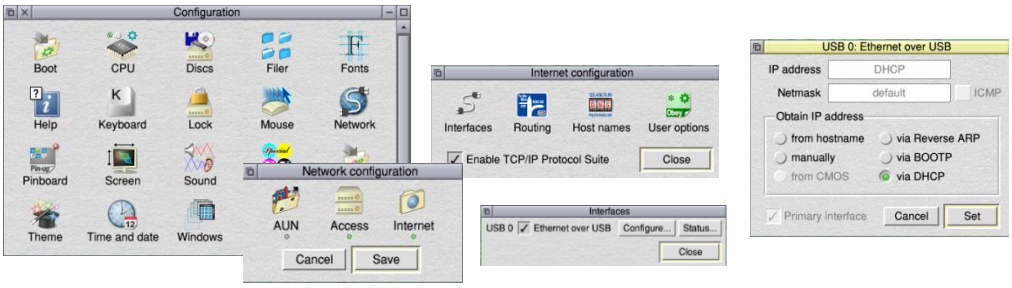
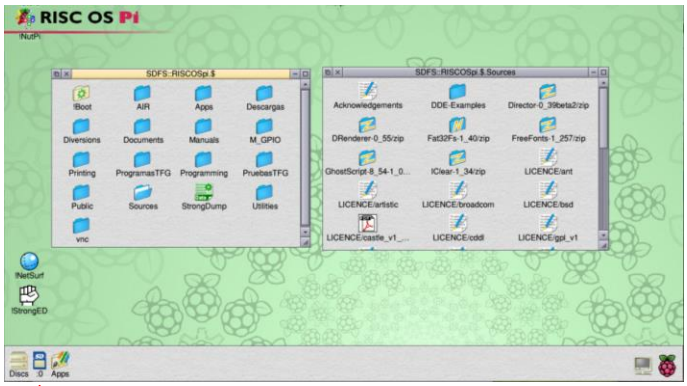


Figura 3.1: Configuración internet en RISC OS

Para cambiar Gateway seguimos los pasos indicados anteriormente, hasta la ventana de configuración de internet aquí ya podremos seleccionar ***Routing,*** que nos permitirá cambiar el Gateway.

3.2.3. DESCARGAR Y BORRAR CARPETAS

Lo mejor cuando descargamos algo es guardar el archivo zip en la SD, dentro de la carpeta Source, como se muestra en la Figura 3.2.



Figua 3.2: Escritorio RISC OS

Para borrar un directorio tendremos que pinchar botón central del ratón, nos aparecerá una ventana con distintas opciones, nos ponemos encima del nombre del directorio (sin clicar) y nos aparecerá una nueva ventana donde podremos seleccionar distintas opciones, entre ellas *Delete* (^k). Como se muestra en la Figura 3.3.

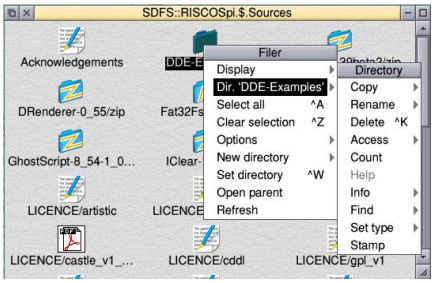


Figura 3.3: Eliminar directorios en RISC OS

3.2.4. INSTALACIÓN DE VNC

Para poder conectarnos directamente con la Raspberry Pi sin necesidad de tener que tenerla conectada a una pantalla, ratón y teclado, podemos conectarnos a través de VNC (Virtual Network Computing) con nuestro portátil. Para ello debemos seguir los siguientes pasos (Información de [13]):

- Deberemos conectar la Raspberry Pi a internet (Seguir apartado 3.2.2).

- Desde el navegador *NetSurf*, se accederá a la siguiente dirección [7] para poder descargar vnc_serv.zip (166,099 bytes), al clicar sobre el zip se iniciará la descarga que se mostrará en una ventana llamada NetSurf Download.
- Los programas que descargamos no se almacenan en una carpeta de descargas, como ocurre con otros sistemas operativos. Para acceder al archivo descargado debemos arrastrarlo a una carpeta que hayamos habilitado para ello.
- Deberemos descomprimir el fichero. Para ello, creamos una carpeta con el nombre de vnc dentro de la carpeta Apps (que se encuentra en la barra inferior a la izquierda), la abrimos, después abrimos con doble clic el archivo zip de VNC y arrastramos todo su contenido a la nueva carpeta (si no descomprimimos el archivo puede que nos de problemas al reiniciar RISC OS).
- Dentro de la carpeta vnc que nos hemos creado en Apps, se encontrarán los tres elementos principales, el módulo con el programa VNC y los archivos Start y Stop, que se encargan de configurar el puerto y la contraseña, también se encargan de cargar y eliminar el módulo en el sistema operativo. Si se abre el archivo Start, se puede ver que viene configurado por defecto el puerto 5900 y la contraseña ***bob***. El puerto no es necesario cambiarlo.
- Ahora es necesario decirle al sistema operativo que inicie el servidor cada vez que se inicia RISC OS. Para ello vamos a ***!Configure*** y abrimos la carpeta *Boot,* se selecciona *Run* y se arrastra el archivo *start* de VNC dentro de la nueva pantalla. Finalmente pulsamos Set, todo el proceso se puede ver en la Figura 3.4.

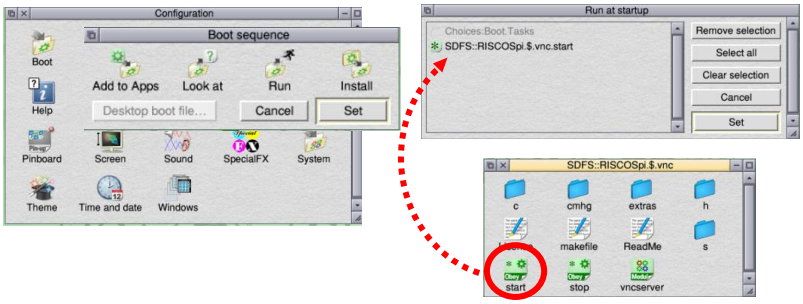


Figura 3.4:Iniciar servidor al arrancar RISC OS

- Una vez tengamos configurada la Raspberry, es necesario configurar nuestro portátil. Para ello nos descargamos en el portátil un cliente VNC (VNC Viewer), está disponible en la página oficial [8].

- Para poder conectar por ethernet el portátil y la Raspberry, tenemos que modificar la ip de la Raspberry y la del portátil. Hay que establecer las IPs como fijas e indicarle a cada uno de los equipos el Gateway (seguir el apartado 3.2.2).

  Ejemplo:
  Portátil: ip 192.168.1.1 gateway:192.168.1.2
  Rasberry: ip 192.168.1.2 gateway:192.168.1.1

- Por último, ya sólo nos queda conectar por ethernet la Raspberry Pi a nuestro portátil. Abrimos VNC Viewer en nuestro portátil y creamos una nueva conexión (File/New connection). Nos aparecerá una ventana donde deberemos indicar la IP a la que nos queremos conectar y un nombre para la nueva conexión. Una vez nos aparezca en la pantalla del VNC la nueva conexión, clicamos dos veces sobre ella y nos pedirá la contraseña para poder conectarse (si no se ha cambiado, será *bob*). Acto seguido conectará con la Raspberry mostrando el escritorio de RISC OS. Todos los pasos se pueden ver en la Figura 3.5.

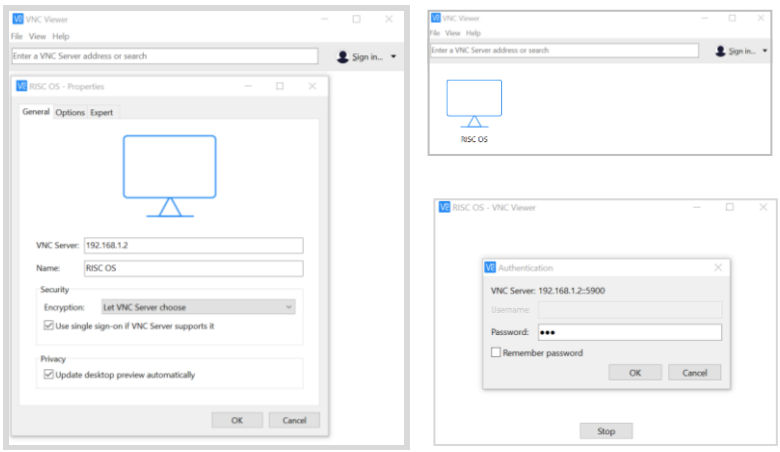


Figura 3.5: Conectarse por VNC a la Raspberry.

3.2.5. USO DEL TERMINAL

Para abrir un terminal en RISC OS pulsar *ctrl+F12*. La lista de comandos completa que se pueden ejecutar en la TaskWindow se encuentra en la siguiente web [1].

3.2.6. EDITOR !STRONGED

RISC OS dispone de un editor de texto que será necesario utilizar para realizar nuestros códigos. Para abrirlo hay que hacer doble clic sobre el icono de

!StrongED que se encuentra en el escritorio. Después de esto, en vez de aparecer la ventana del editor, aparecerá un nuevo icono en la barra inferior derecha, como se muestra en la Figura 3.6. Una vez abierto el editor, si queremos abrir algún archivo concreto solo tenemos que arrastrarlo al icono de !strongED que hay en la barra.

Figura 3.6: Barra de tareas RISC OS

Si se quiere abrir un nuevo documento, deberemos clicar una vez en el icono de !StrongED de la barra de tareas.

Clicando dos veces con la tecla shft presionada sobre el icono de ¡StrongED, aparecerá la carpeta del programa con manuales de ayuda y los ejecutables. Si abrimos ¡Help nos mostrará un menú y si seleccionamos *Key bindings* podremos ver todos los atajos de teclado del editor, como se ve en la Figura 3.7.

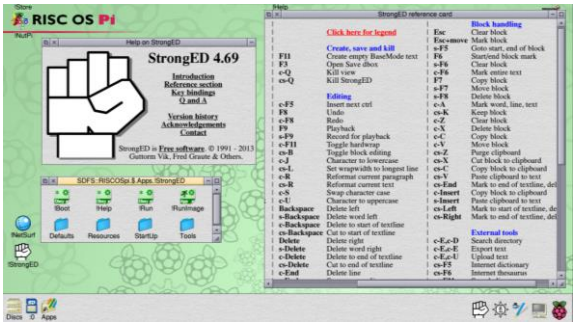


Figura 3.7: Atajos de teclado del editor StrongED

3.2.7. INSTALACIÓN GCC (GCC + AS +LD) Y PRUEBA

El compilador que usaremos de ahora en adelante se trata de GCC [16]. Es un compilador de código abierto fácil de instalar en RISC OS y nos proporcionará todas las herramientas necesarias. Una vez que lo tenemos instalado, podemos ensamblar nuestro código fuente con **as** y enlazarlo con **ld** sin problemas.

Para instalarlo, la manera más sencilla es hacerlo desde el programa *!PackMan de RISC OS*. De esta manera, PackMan se encarga automáticamente de instalar y tener cuidado con las dependencias con otros paquetes. Si se desea, también se puede instalar de forma manual, descargándose el paquete directamente desde [9], pero en este caso se deberán cargar todas las dependencias manualmente también.

Para instalar GCC desde !PackMan, primero debemos hacer doble clic sobre el icono de !PackMan que tenemos en el escritorio, lo que hará que aparezca en la parte izquierda de la barra de tareas, clicamos sobre él y nos aparecerá una ventana con todos los programas disponibles para instalar en RISC OS. Seleccionamos la lupa e indicamos que nos busque GCC. Esta búsqueda nos mostrará un listado con las instalaciones disponibles, si están ya instaladas aparecerán marcadas con un *check*. Seleccionamos GCC4 y clicamos en el botón central del ratón, aparecerá un menú en el que debemos situarnos sobre *Package 'GCC4'* e *install*. Después de esto aparecerá una ventana con los paquetes que va a instalar, para concluir la instalación de GCC seleccionamos *install*. Pasos indicados en la Figura 3.8.

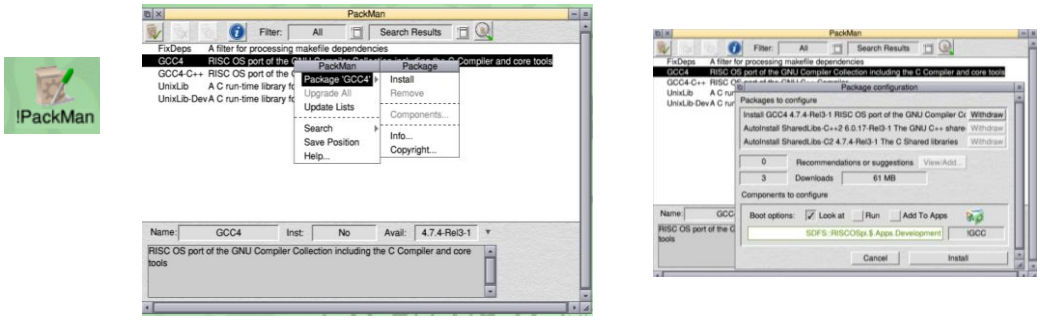


Figura 3.8: Instalación GCC con PackMan.

Para empezar a usar GCC, antes de abrir la terminal de comandos, debemos ampliar next a 16000k, esto es debido a que GCC consume mucha memoria. Para ello clicamos sobre la frambuesa de la barra de tareas y arrastramos la barra de next hasta el valor indicado, seguir Figura 3.9. Después nos vamos a la carpeta Apps (SDF::RISCOSpi.$.Apps) y hacemos doble clic en !GCC para activarlo. Ya podemos abrir la terminal (TaskWindow) y escribimos gcc -v. Si tenemos abierta la ventana de tareas y hemos ejecutado gcc -v antes de ampliar next, cerrar y volver a abrir para que compile correctamente.

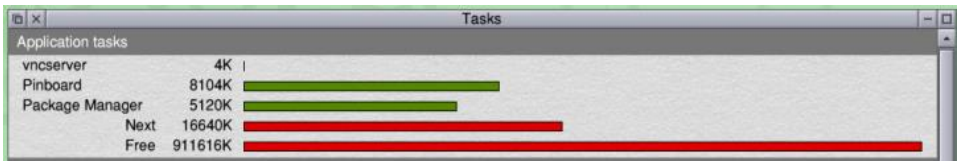


Figura 3.9: Ajustar next para que GCC disponga de memoria suficiente.

Ahora que ya tenemos todo instalado correctamente, podemos hacer una pequeña prueba con un 'Hello world'. Dentro de la carperta Apps en !GCC tenemos varios programas de prueba (para abrir el contenifo de !GCC hacemos

doble clic sobre el icono mientras presionamos la tecla shift), mas información en [29].

- Creamos una carpeta donde guardaremos los programas en C, por ejemplo *programsC* dentro de la SD.
- Dentro de esta, creamos otra carpeta llamada c, aquí es donde tendremos los códigos en *C*.
- Generamos nuestro programa en C y lo guardamos en la carpeta c.
- Abrimos la TaskWindow y ejecutamos ***gcc -v .***
- Nos situamos en la carpeta *programsC con la siguiente dirección:* ***dir sdfs::riscospi.$.programsC***
- Para listar el contenido de la carpeta usar el comando ***ex.***
- Creamos el programa en código C con el editor *!StrongED*.

```
#include <stdio.h>
int main()
{ printf("Hello World\n");
  return 0;}
```

- Compilamos el archivo con el código en C que hemos llamado helloW, nos aparecerá un ejecutable con el nombre que le indiquemos al compilar en la carpeta programsC, en este caso le hemos llamado hello como se muestra en la Figura 3.10.
- Ejecutamos hello para ver los resultados. Figura 3.10.

Figura 3.10: Compilar y ejecutar programa C.

3.2.7.1. ENSAMBLAR Y ENLAZAR

Para este proyecto necesitamos realizar partes del programa en código ensamblador ARM y en código C. Por ello es necesario ensamblar primero cada una de las partes del programa de forma individual (los archivos .s por un lado y los .c por otro). Esto se puede hacer cómodamente con el comando ***as***, disponible en GCC. Una vez que tenemos todos nuestros códigos pasados a código objeto, podemos enlazarlos y crear un solo ejecutable con el comando ***ld***. La referencia que hemos seguido para esta sección es [2].

Como ejemplo, mostraremos un pequeño programa que realiza una operación sencilla con dos enteros.

- Primero creamos un directorio que guardará los códigos del programa. Dentro de éste crearemos tres más: uno para los códigos en C, (llamado c), otro para los códigos en ensamblador (llamado s) y un último directorio donde se guardarán los objetos creados (que le llamaremos o). Estos nombres no se pueden variar.
- Generamos los códigos en C:

```
#include <stdio.h>
extern int suma(int a, int b);
int main(int argc, char *argv[])
{ printf("== sum ==\n");
  int a = 44;
  int b = 33;
  printf("%d + %d - 20 = %d\n", a, b, suma(a,b));
  return 0;}
```

- Generamos los códigos en ensamblador ARM:

```
.align 2
.global suma
suma:
  add r2, r0, r1
  sub r2, r2, #20
  mov r0,  r2
  mov pc, lr
```

- Abrimos la TaskWindow y nos situamos en el directorio del proyecto. Todos los pasos están en la Figura 3.11.
- Creamos los códigos objetos para los códigos en C:

  gcc -c <nombre_codigoC>.c
- Creamos los códigos objetos de los archivos en ensamblador ARM: ***as -o <nombre_ARM>.o <nombre_ARM>.s***
- Creamos un único ejecutable, que se creará en el directorio principal del proyecto:

  gcc -o <nombre> <nombre_codigoC>.o <nombre_ARM>.o
- Si lo que queremos es crear un único ejecutable con todos los archivos de tipo ensamblador, deberíamos usar este otro comando e lugar del anterior:

  ld -o <nombre_ejecutable> <ARM1>.o <ARM2>.o

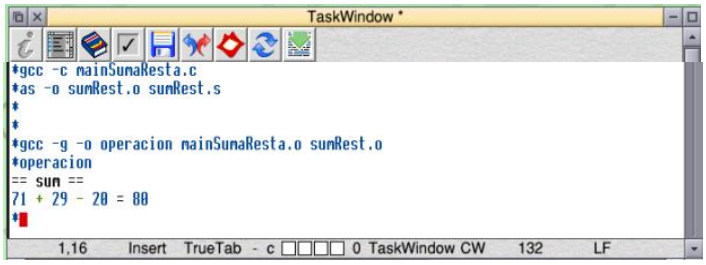


Figura 3.11: Compilar y ejecutar un programa en C con ensamblador.

3.3. DEPURADOR

Después de investigar las opciones disponibles, no hemos encontrado ningún depurador de código abierto y que realizase las tareas que eran necesarias para la realización de las prácticas de las asignaturas: Estructura de Computadores y Organización de Computadores.

A continuación, realizaremos un pequeño resumen de las opciones disponibles para depurar en RISC OS y sus inconvenientes:

1. *Debugger* de RISC OS, funciona muy bien para realizar programas con BASIC pero en nuestro caso, que utilizamos indistintamente tanto programas en código ensamblador ARM, como en código C, no nos es viable. Más información en la página web[3].
2. DDT, se trata de un *debugger* desarrollado para RISC OS, que en principio podría haber sido de utilidad. El problema es que es de pago y no hay ninguna versión de prueba disponible, se intentó ponerse en contacto con los desarrolladores para ver si era lo que se necesitaba, pero no hubo respuesta. Más información en la página web[4].
3. Breakaid, no se trata de un *debugger* cien por cien, pero si se pueden ver direcciones y estados de la memoria. Es un pequeño *debugger* desarrollado por un programador. Se puede encontrar más información en el siguiente enlace [5].
4. GDB, se intentó instalar, pero la versión que había disponible no se pudo instalar con éxito.

Actualmente se está desarrollando en la Facultad de Ciencias un debugger para RISC OS. Se le ha llamado CRISCdbg, como se indica en el artículo publicado en las Jornadas SARTECO 2017 [12]. Está centrado en las herramientas necesarias para el correcto desarrollo de las prácticas y el aprendizaje de los alumnos para depurar y codificar de forma correcta los programas tanto en lenguaje C como en ensamblador.

4. EL LABORATORIO DOCENTE

Después de toda la información anteriormente recopilada, se ha decido la instalación del sistema operativo RISC OS y del compilador GCC. En cuanto al modelo de Raspberry Pi seleccionado, se trata de la Raspberry Pi 1 modelo B+ debido a varios factores que detallaremos en los próximos apartados. Por último, también realizaremos una exposición de los dispositivos que podrían estar disponibles en el laboratorio para la realización de las distintas prácticas y la forma de uso de la GPIO disponible en nuestra Raspberry Pi, la cual nos permitirá ponernos en contacto con estos dispositivos.

4.1. MODELO RASPBERRY PI SELECCIONADO

En un principio se empezó probando la Raspberry Pi 3, se trata del modelo más actual y tiene unas mejoras que la hacen muy interesante para futuros proyectos, como por ejemplo su CPU ARMv8 de cuatro nucleos, 64 bits, a 1.2 Ghz y dispone de 802.11n Wireless LAN (apartado 2.2). Pero su reciente salida al mercado también ha tenido el inconveniente de que el sistema operativo RISC OS no estaba convenientemente actualizado para su correcto funcionamiento en ella. Ethernet no funcionaba, lo que generaba muchos problemas para actualizar el software al no poderse usar el programa !PackMan de RISC OS. Pero el problema en la instalación de las librerías para GCC (como libstdc++), así como disponer, de una CPU de 64bits que podía complicar el aprendizaje en algunas prácticas, hizo que finalmente este modelo no fuese el adecuado.

Seguidamente se realizó la mayor parte de este proyecto con Raspberry Pi 2 modelo B, pero finalmente también hubo que descartarla debido a la falta de documentación disponible y a problemas en la implementación de la terminal (TaskWindow) que impedía retornar correctamente al sistema operativo cuando se pasa a modo supervisor desde la terminal que se ejecuta pulsando Ctrl+F12. Para que todo funcionase correctamente era necesario abrir la terminal pulsando F12. Este modelo usa un procesador Broadcom BCM2836, pero solo hay disponible la documentación completa del Broadcom BCM2835[10]. Esta falta de documentación detallada, ha dado problemas a la hora de realizar las pruebas con la GPIO, ya que la dirección base no es la indicada en las diferentes documentaciones, 0x20200000 en la Raspberry o 0x7E200000, cualquiera de

estas dos direcciones deberían funcionar para poder acceder a la GPIO sin problemas, pero no es asi. Después de mucho buscar por internet, en un foro se encontro la dirección 0x3F200000 que funciona correctamente.

Por todos los motivos anteriormente citados, se ha seleccionado la ***Raspberry Pi 1 modelo B+***, dispone de amplia información y el sistema RISC OS se integra perfectamente.

4.2. INSTALACIÓN RISC OS Y GCC

Todas las Raspberry Pi del laboratorio deberán de tener instalado el sistema operativo RISC OS y el compilador GCC. Para su instalación seguir los pasos detallados en el capítulo 3 de este proyecto, en los apartados ***3.2.1.*** y ***3.2.7.***

4.3. LA GPIO

La fila de la GPIO (entrada/salida de propósito general) se encuentra en la parte superior de la Raspberry Pi, como se muestra en la Figura 4.1. Estos pins son una interfaz física entre la Raspberry Pi y el exterior.

Figura 4.1: GPIO Raspberry Pi 2 B [23]

De esos 40 pines, 26 son pins GPIO y los otros son pins power o ground (mas dos pines ID EEPROM.), como se detalla en la Figura 4.2. Para una mayor información visitar la página [23].

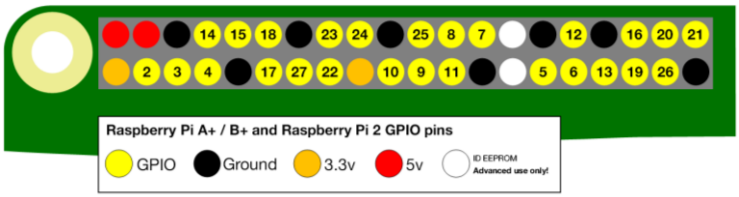


Figura 4.2: Pines GPIO Raspberry Pi A+/B+ y Raspberry Pi 2 GPIO [23]

En la Tabla 4.1 que aparece a continuación, se muestran los 41 registros de la GPIO, indicando cuál es la función de cada uno de los mismos, el tamaño y el tipo. En cuanto a la dirección que nos aparece es la dirección física en la que se encuentran situados, si quisiéramos acceder a ellos sin disponer de un sistema operativo. Pero en este caso hay que tener en cuenta la dirección base

en la que se encuentran mapeados en nuestra Raspberry Pi, que como hemos comentado en apartados anteriores se trata de la dirección base 0x20200000. El desplazamiento que aparece para el resto de los registros en la tabla se mantiene igual.

Address	Field Name	Description	Size	Read/ Write
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W
0x 7E20 0010	GPFSEL4	GPIO Function Select 4	32	R/W
0x 7E20 0014	GPFSEL5	GPIO Function Select 5	32	R/W
0x 7E20 0018	-	Reserved	-	-
0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W
0x 7E20 0024	-	Reserved	-	-
0x 7E20 0028	GPCLR0	GPIO Pin Output Clear 0	32	W
0x 7E20 002C	GPCLR1	GPIO Pin Output Clear 1	32	W
0x 7E20 0030	-	Reserved	-	-
0x 7E20 0034	GPLEV0	GPIO Pin Level 0	32	R
0x 7E20 0038	GPLEV1	GPIO Pin Level 1	32	R
0x 7E20 003C	-	Reserved	-	-
0x 7E20 0040	GPEDS0	GPIO Pin Event Detect Status 0	32	R/W
0x 7E20 0044	GPEDS1	GPIO Pin Event Detect Status 1	32	R/W
0x 7E20 0048	-	Reserved	-	-
0x 7E20 004C	GPREN0	GPIO Pin Rising Edge Detect Enable 0	32	R/W
0x 7E20 0050	GPREN1	GPIO Pin Rising Edge Detect Enable 1	32	R/W
0x 7E20 0054	-	Reserved	-	-
0x 7E20 0058	GPFEN0	GPIO Pin Falling Edge Detect Enable 0	32	R/W
0x 7E20 005C	GPFEN1	GPIO Pin Falling Edge Detect Enable 1	32	R/W
0x 7E20 0060	-	Reserved	-	-
0x 7E20 0064	GPHEN0	GPIO Pin High Detect Enable 0	32	R/W
0x 7E20 0068	GPHEN1	GPIO Pin High Detect Enable 1	32	R/W
0x 7E20 006C	-	Reserved	-	-
0x 7E20 0070	GPLEN0	GPIO Pin Low Detect Enable 0	32	R/W
0x 7E20 0074	GPLEN1	GPIO Pin Low Detect Enable 1	32	R/W
0x 7E20 0078	-	Reserved	-	-
0x 7E20 007C	GPAREN0	GPIO Pin Async. Rising Edge Detect 0	32	R/W
0x 7E20 0080	GPAREN1	GPIO Pin Async. Rising Edge Detect 1	32	R/W
0x 7E20 0084	-	Reserved	-	-
0x 7E20 0088	GPAFEN0	GPIO Pin Async. Falling Edge Detect 0	32	R/W
0x 7E20 008C	GPAFEN1	GPIO Pin Async. Falling Edge Detect 1	32	R/W
0x 7E20 0090	-	Reserved	-	-
0x 7E20 0094	GPPUD	GPIO Pin Pull-up/down Enable	32	R/W
0x 7E20 0098	GPPUDCLK0	GPIO Pin Pull-up/down Enable Clock 0	32	R/W
0x 7E20 009C	GPPUDCLK1	GPIO Pin Pull-up/down Enable Clock 1	32	R/W
0x 7E20 00A0	-	Reserved	-	-
0x 7E20 00B0	-	Test	4	R/W

Tabla 4.1: Los 41 registros de la GPIO [10]

4.3.1. USO DE LA GPIO

Los pasos que hay que seguir si queremos trabajar con un pin de la GPIO son los siguientes:

1. Determinar qué registro GPIOSEL (0-5) controla el pin que queremos usar.
2. Determinar qué bits del registro GPIOSEL serán usados.
3. Determinar qué patrón de bits deberemos usar.
4. Leer el registro GPIOSEL.
5. Limpiar los bits correctos usando la instrucción bic.
6. Aplicar el patrón correcto usando la instrucción orr. Los diferentes patrones se muestran en la Tabla 4.2.

TABLA PATRONES DE BITS	
000	GPIO pin X es una entrada
001	GPIO pin X es una salida
100	GPIO pin X toma función alternativa 0
101	GPIO pin X toma función alternativa 1
110	GPIO pin X toma función alternativa 2
111	GPIO pin X toma función alternativa 3
011	GPIO pin X toma función alternativa 4
010	GPIO pin X toma función alternativa 6

Tabla 4.2: Patrones de bits para la entrada/salida de la GPIO [28]

Por ejemplo, si quisiéramos encender un led que está situado en la GPIO9 modificaremos los puertos.

- **GPFSEL0** situado en la dirección de memoria 0x20200000
- **GPSET0** situado en la dirección de memoria 0x2020001c
- **GPCLR0** limpia la información introducida en GPSET0 situado en la dirección 0x20200028 .

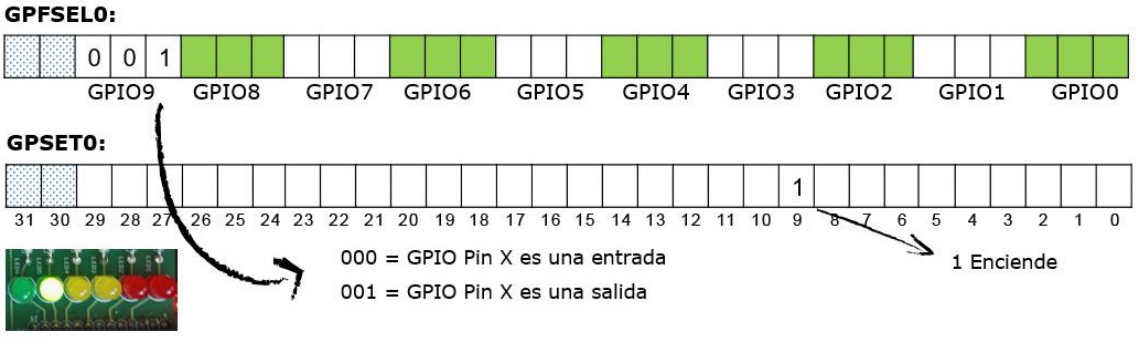


Figura 4.3: Ejemplo de uso GPIO

4.4. DISPOSITIVOS I/O

Para la realización de las prácticas, hay una gran variedad de dispositivos. Los dispositivos que mostraremos a continuación, son fáciles de encontrar en las tiendas de electrónica, así como en internet, a precios reducidos. Además, estos dispositivos son una herramienta indispensable para el desarrollo de proyectos sumamente interesantes para nuestra Raspberry Pi.

BerryClip: Placa con 6 leds, 1 buzzer y un interruptor, que se muestra en la Figura 4.4. Se vende desmontada, dispone de página web [19] con amplia información para su montaje, llegando a ser sencillo y didáctico. En la Tabla 4.3 se muestra la correspondencia entre los Leds y la GPIO.

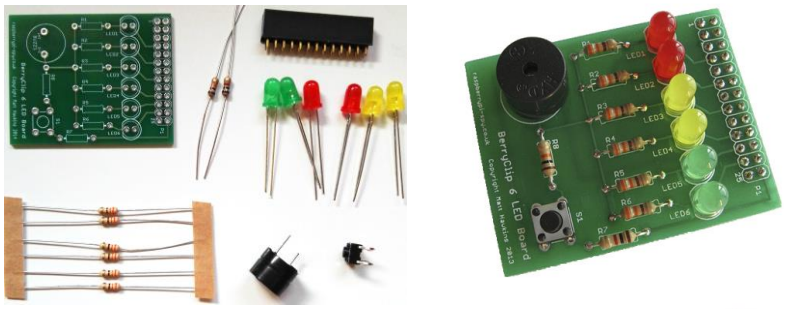


Figura 4.4: Placa BerryClip.

Nº LED	Nº PIN	GPIO
LED 1	Pin 7	GPIO4
LED 2	Pin 11	GPIO17
LED 3	Pin 15	GPIO22
LED 4	Pin 19	GPIO10
LED 5	Pin 21	GPIO9
LED 6	Pin 23	GPIO11
Buzzer	Pin 24	GPIO8
Switch	Pin 26	GPIO7

Tabla 4.3: Correspondencia LEDs y GPIO en BerryClip [19].

Sensor de temperatura y humedad DHT11: Este sensor dispone de una señal digital bien calibrada. Se trata de un sensor con un microcontrolador de 8 bits y 2 sensores resistivos encapsulados en una caja de plástico azul.

Tenemos varios modelos de sensor DHT11 disponibles, algunos de ellos vienen soldados a una placa (Figura 4.6) evitándonos tener que montar la resistencia que actúa como pull-up. Otros vienen sueltos (Figura 4.5) para usarse directamente conectados a una breadboard.

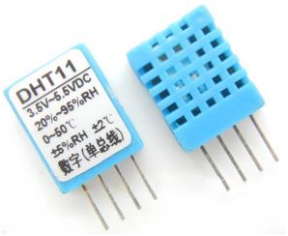


Figura 4.5: Sensor de humedad sin placa [18].

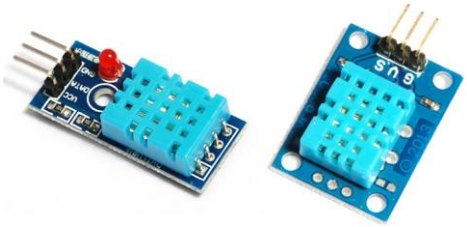


Figura 4.6: Sensor de humedad con placa [18].

El protocolo de comunicación se realiza a través de un único hilo, pudiéndolo conectar directamente a los pines GPIO de la Raspberry Pi. Figura 4.7.

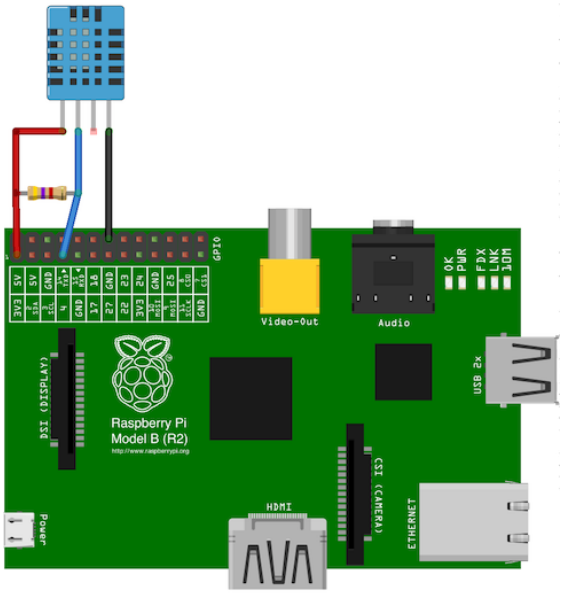


Figura 4.7: Conexión del sensor de humedad a la GPIO de Raspberry Pi [18].

La tensión de alimentación del sensor es de 3~5V, se puede conectar a los pines +5V o +3’3V y GND de la Raspberry Pi o alimentarlo desde una batería externa. Su rango de funcionamiento es de 0 a 50ºC para la temperatura y de 20% a 90% de humedad relativa. El pin de comunicaciones para realizar lecturas se puede conectar a cualquier pin GPIO y tiene la capacidad de transmitir la señal hasta 20 metros de distancia. La tasa de refresco en las lecturas es de 2 segundos. Información resumida de [18].

Placa 6 leds, buzzer y 2 interruptores: Esta es la placa (Figura 4.8) que hemos usado finalmente para realizar las pruebas de validación. Dispone de los mismos elementos que la placa BerryClip mas un interruptor extra.

Figura 4.8: Dispositivos para Raspberry Pi.

Nos hemos decido por la implementación de esta placa en concreto, debido a que es la más sencilla para programar y hay información detallada de su programación en [28]. En la Figura 4.9 se muestra la correspondencia entre los Leds y la GPIO.

Figura 4.9: Correspondencia LEDs y GPIO en BerryClip [28].

Display 16x2 LDC: Este dispositivo se trata de una pequeña pantalla alfanumérica que nos permite mostrar una cadena de caracteres. Figura 4.10.

Figura 4.10: 16x2 LCD [20].

Podremos crear programas junto con otros dispositivos, como por ejemplo el anteriormente mencionado sensor de humedad, que nos muestren la información de estos sensores a través de la pantalla LCD. Dispone de 16 pines, la salida de estos se muestra en la Tabla 4.4. Información resumida de [20].

Nº Pin	Uso
1	Ground
2	VCC (Usually +5V)
3	Contrast adjustment (VO)
4	Register Select (RS). RS=0: Command, RS=1: Data
5	Read/Write (R/W). R/W=0: Write, R/W=1: Read
6	Enable
7	Bit 0
8	Bit 1
9	Bit 2
10	Bit 3
11	Bit 4
12	Bit 5
13	Bit 6
14	Bit 7
15	LED Backlight Anode (+)
16	LED Backlight Cathode (-)

Tabla 4.4: Disposición de los pines de 16x2 LCD [20]

Sensor distancia HC-SR04: Este sensor está diseñado para detectar los objetos cercanos. Consta de un transmisor ultrasónico, un receptor y un circuito de control. (Figura 4.11). El transmisor emite un sonido ultrasónico de alta frecuencia, que rebota en cualquier objeto sólido cercano. Parte de ese ruido es reflejado y detectado por el receptor en el sensor. Esta señal recibida es entonces procesada por el circuito de control para calcular la diferencia de tiempo entre la señal que se transmite y la que recibe, de esta manera se puede determinar a qué distancia se encuentra el objeto.

Figura 4.11: Sensor distancia HC-SR04 [21]

El sensor HC-SR04 dispone de 4 pines: tierra (GND), salida de pulso eco (ECHO), entrada de pulso de activación (TRIG) y suministro de 5V (Vcc). Para encender el módulo podemos usar VCC, se conecta a tierra con GND y usando la Raspberry Pi se envía una señal de entrada a TRIG, lo que dispara el sensor para enviar un pulso ultrasónico. La onda del pulso rebota en el objeto cercano y se refleja de nuevo en el sensor. El sensor detecta estas ondas de retorno y mide el

tiempo entre la emisión y la recepción, posteriormente envía una señal de 5V en el pin ECHO.

La señal de salida del sensor (ECHO) en el HC-SR04 está clasificado en 5V. Sin embargo, el pin de entrada en el GPIO Raspberry Pi está clasificado en 3.3V. El envío de una señal de 5V en ese puerto de entrada desprotegido de 3.3V podría dañar los pines GPIO. Habría que usar un pequeño circuito de divisor de tensión, que consiste en dos resistencias, para bajar el voltaje de salida del sensor para que que la Raspberry Pi lo pueda manejar. Información resumida de [21].

5. VALIDACIÓN

Para comprobar que todo lo indicado anteriormente funciona correctamente y cumple las expectativas deseadas, hemos realizado dos prácticas en el laboratorio. Para ello nos hemos centrado principalmente en comprobar los siguientes puntos:

1. La creación de un programa íntegramente en código ensamblador que gestione una excepción.
2. El uso tanto de lenguaje C así como de ensamblador, para la creación de un único programa, permitiendo comprobar las herramientas incluidas en GCC para compilar, ensamblar y enlazar.
3. Crear un driver para la gestión de un dispositivo conectado a nuestra Raspberry Pi, permitiéndonos comprobara la gestión de Entrada/Salida.

5.1. PRÁCTICA EN ENSAMBLADOR

En esta práctica hemos creado un programa que se encarga de gestionar una excepción producida por un error de alineamiento. El programa ha sido programado en código ensamblador ARM.

Lo primero que hemos definido son dos variables, la variable ***A*** qué se trata de un variable tipo word y que guardara el número de veces que se entra en la rutina de atención, y la variable ***contenido_A*** de tipo asciz que nos permite imprimir un pequeño texto explicativo cuando mostramos el valor de la variable ***A***.

```
.data
.align 4
A: .word 0
contenido_A:
.asciz "Fallos de alineamiento: %d, \n"
```

Para que las diferentes funciones que hemos creado puedan ser usadas desde otros ficheros, las definimos como globales:

```
.text
.global main
.global inicializamos_excepcion
.global deshabilitamos_excepcion
.global rutina_atencion
```

Para inicializar la excepción hemos creado una función llamada ***inicilizamos_excepcion,*** que se encarga de inicializarla cuando se la llama. Esta función realmente lo que hace es usar una de las swi del sistema operativo disponibles en RISC OS, las cuales nos permiten trabajar con código a bajo nivel, de forma directa con nuestra Raspberry Pi, sin necesidad de tener que saltarnos el sistema operativo. La swi utilizada en este caso es ***SWI OS_ClaimProcessorVector*** [ANEXO, Tabla A.1] Antes de llamarla hay que guardar en r0, en los 8 primeros bits (bits 0-7), el número de vector a tratar (en este caso Data abort [ANEXO, Tabla A.3]) y en el bit 8 un 1 para indicar que lo habilite, esto es lo que hacemos con la instrucción ***mov r0,#0x104***, en r1 hay que carga la dirección de la rutina de atención que hemos creado para gestionar la excepción y en r2 cargamos un 0. Ahora ya podemos llamar a OS_ClaimProcessorVector con la instrucción ***swi 0x69***

```
inicializamos_excepcion:
mov r0, #0x104
ldr r1, =rutina_atencion
mov r2, #0
@ Llamamos a SWI OS_ClaimProcessorVector
swi 0x69
mov pc, lr
```

Para deshabilitar la excepción se crea otra función llamada ***deshabilitamos_excepcion.*** En este caso para OS_ClaimProcessorVector, como nos indica en la tabla [ANEXO, Tabla A.1] deberemos cambiar el bit 8 a 0.

```
deshabilitamos_excepcion:
mov r0, #0x004
ldr r2, =rutina_atencion
swi 0x69
mov pc, lr
```

Cuando se produce la excepción hemos indicado que se ejecute la siguiente rutina llamada ***rutina_atencion***. Se trata de una rutina sencilla que incrementa en 1, el valor almacenado en A.

```
rutina_atencion:
ldr r3, =A
ldr r4, [r3]
add r4, r4, #1
str r4, [r3]
subs pc, lr, #4
```

En la rutina anterior hay que tener cuidado con dos aspectos, el primero son los registros r0, r1 y r2, ya que si se desean usar es necesario guardar su valor anterior en la pila y restaurarlos al finalizar su uso. Y otro aspecto y el más importante de todos, se trata de la forma de retornar de la excepción, ya que esta debe de ser usando la instrucción ***subs pc, lr, #4*** para poder restaurar el pc, es necesario restar 4 para corregir los efectos del pipelining. Si no retornásemos con esta instrucción, el programa se bloquearía y deberíamos reiniciar el sistema operativo.

Otra de las funciones que hemos creado para que use nuestro programa principal es ***genera_excepción***, que genera un fallo de alineamiento cada vez que se la llama, dando lugar a que se produzca la excepción que hemos gestionado anteriormente.

```
genera_excepcion:
@ Genero error de alineamiento
ldr r3, =0x11223300
str r4, [r3]
movs pc, lr
```

Por último, creamos el programa principal, que se encargará de llamar a las distintas funciones y finalmente imprimir por pantalla la variable A que nos indicará si se han producido el número de excepciones deseadas.

```
main:
bl inicializamos_excepcion
bl genera_excepcion
bl genera_excepcion
bl genera_excepcion
bl deshabilitamos_excepcion
@ Imprimimos número de fallos de alineamiento producidos
ldr r2, =A
ldr r1, [r2]
stmfd sp!, {r0-r3}
ldr r0, =contenido_A
bl printf
ldmfd sp!, {r0-r3}
swi 0x11
.end
```

Ahora sólo nos quedaría ensamblar el código anteriormente generado y enlazarlo.

Para ello creamos el código objeto del archivo en ensamblador ARM: ***as -o <nombre_ARM>.o <nombre_ARM>.s***

Después crearemos el ejecutable, en este caso en vez de usar ***ld*** utilizaremos ***gcc*** para poder utilizar la función printf en el programa principal.

gcc -o <nombre> <nombre_ARM>.o

5.2. PRÁCTICA EN C Y ENSAMBLADOR

En esta práctica de laboratorio, hemos comprobado que se pueda trabajar correctamente usando tanto código en lenguaje C, como código en ensamblador para la creación de un único programa. Para ello hemos creado un pequeño juego, basado en el clásico juego Hundir la Flota.

En código ensamblador ARM, hemos creado la función ***ataca.s*** que recibe las coordenadas introducidas por el jugador y retorna el resultado obtenido.

Lo primero que haremos será definir las constantes y variables necesarias. Para ello definimos las constantes FILAS, COLUMNAS y T_ELEM que nos permiten indicar el tamaño de la matriz, así como el tamaño de los elementos que almacena dicha matriz, para poder más tarde acceder a ella. Acto seguido hemos declarado la MATRIZ con elementos de tipo word, indicando el contenido que almacenará, con las diferentes posiciones de los barcos en el tablero. Por último, se han creado 4 variables más: BarcosHundidos (en la cual almacenaremos los barcos que han sido eliminados a lo largo del juego), B1 (guardará el número de veces que debe de ser alcanzado el barco para poder hundirlo) B2 y B3 que realizan la misma función que B1 pero para sus respectivos barcos.

```
.data
.equ   FILAS, 5
.equ   COLUMNAS, 5
.equ   T_ELEM, 4
.align 4
MATRIZ:
.word 1,0,0,2,2,0,0,0,0,0,0,0,3,0,0,0,0,3,0,0,0,0,3,0,0
endMATRIZ:
.align 4
BarcosHundidos: .word 0
B1: .word 1
B2: .word 2
B3: .word 3
```

Ahora es necesario mirar en la matriz qué valor está almacenado teniendo en cuenta las coordenadas que nos ha pasado a través de los registros r0 (coordenada x) y r1 (coordenada y). Para ello accederemos a nuestra matriz y cargaremos el contenido de la posición (x,y) en el registro r3. Esto nos permite comprobar que se puede trabajar con matrices sin ningún problema para nuestras prácticas futuras.

```
.text
.global ataca
ataca:
   ldr r5, =MATRIZ    @ Carga la dirección de MATRIZ
   mov r3, r1         @ Valor coordenada y a r3
   sub r3, r3, #1
   mov r4, #COLUMNAS
   mul r6, r3, r4
   mov r4, r0         @ Valor coordenada x a r4
   sub r4, r4, #1
   add r3, r6, r4
   mov r4, #T_ELEM
   mul r6, r3, r4
   add r5, r5, r6
   ldr r3, [r5]       @ En r3 está el valor de la posición (x,y)
   mov r6, #0         @ Cargamos 0 en la posición (x,y)
   str r6, [r5]
```

Una vez que ya tenemos el valor de la matriz cargado en r3 hemos realizado un switch que nos permitirá en función de su contenido, realizar una u otra acción. Esta parte del código nos permite comprobar que podemos usar etiquetas, realizar comparaciones y saltos sin ningún problema.

```
      _if:
        cmp r3, #0
        beq _fin
        cmp r3, #1
        beq case_1
        cmp r3, #2
        beq case_2
        b case_3
        case_1:
               mov r0, #11
               b _hundido
        case_2:
               ldr r3, =B2
               ldr r2, [r3]
               sub r2, r2, #1
               str r2, [r3]
               _ifB2:
                   cmp r2, #0
                   beq _elseB2
                   mov r0, #2
```

```
                b _fincase
            _elseB2:
                mov r0, #21
                b _hundido
        case_3:
            ldr r3, =B3
            ldr r2, [r3]
            sub r2, r2, #1
            str r2, [r3]
            _ifB3:
                cmp r2, #0
                beq _elseB3
                mov r0, #3
                b _fincase
            _elseB3:
                mov r0, #31
                b _hundido
         _hundido:
                ldr r7, =BarcosHundidos
                ldr r2, [r7]
                add r2, r2, #1
                str r2, [r7]
         _fincase:
```

Por último, realizamos un if para comprobar el contenido de la variable BarconHundidos (como en el caso anterior estamos comprobando que las etiquetas, comparaciones y saltos con el código en ensamblador programado funcionan correctamente) que ha sido cargado en r2. Esto nos permite mediante el paso de parámetros, indicar únicamente con el valor almacenado en r0 que es lo que debe hacer el programa principal. Si recibe en r0 un 0 no se ha alcanzado a ningún barco, si en r0 hay un 1 es que ya se han hundido todos los barcos y si r0 mantiene el valor almacenado en el switch anterior, nos indicará que se ha tocado o hundido alguno de los barcos indicado.

```
      _ifhundidos:
         mov r3, #3
         cmp r3, r2
         bgt _termina
         mov r0, #1
         b _termina
      _fin:
          mov r0, #0
      _termina:
      mov pc, lr
```

Otro de los programas que hemos creado íntegramente en código ensamblador es el encendido de los leds, para ello hemos creado tres programas

diferentes, **leds_green.s**, **leds_yellow.s** y **leds_red.s**, los cuales se encargaran de encender cada uno de ellos los leds indicados. A continuación, pasaremos a explicar el código desarrollado para **leds_green.s**.

En esta parte del código hemos definido una serie de constantes para poder trabajar con los leds indicados de forma más rápida:

BASE_GPIO: almacena la dirección base de acceso a la GPIO.

CLEAN22_27: define los bits necesarios, para borrar la información de los puertos GPIO22 y GPIO27. Como se puede ver en la Figura 4.9, son los leds verdes L5 y L6 los que se encuentran asociados a estos puertos.

ON_OFF: pone a 1 los bits asociados a GPIO22 y GPIO27 para encenderlos o apagarlos en el momento indicado.

OUT22_27: tiene modificados los bits pertenecientes a GPIO22 y GPIO27, con el patrón necesario para indicar que se tratan de dos puertos de salida. Tabla 4.2.

```
.data
.align 4
.equ BASE_GPIO, 0x20200000
.equ ON_OFF, 0x08400000
.equ CLEAN22_27, 0x00E001C0
.equ OUT22_27, 0x00200040
.global leds_green
```

Ahora una vez que empezamos el programa, necesitamos calcular la dirección lógica en RISC OS de la dirección BASE_GPIO, mediante el uso de SWI OS_Memory [ANEXO, Tabla A.1]. Esta dirección se guardará en el registro r3.

```
leds_green:
@ Calculamos la dirección lógica en RISCO
mov r0, #13     @ Instrucción para OS_Memory
ldr r1, =BASE_GPIO
mov r2, #0x100
SWI 0x68        @ OS_Memory
mov r0, r3      @ En r3 dirección base en RISC OS
```

Para poder modificar los registros que nos permitirán encender los leds, será necesario pasar a modos supervisor, para ello disponemos de SWI OS_EnterOS [ANEXO, Tabla A.1]. Una vez que ya estamos en modo supervisor, modificaremos el registro GPFSEL2, primero limpiando los bits deseados (usando

para ello la constante CLEAN22_27) y aplicando el patrón que tenemos guardado en OUT22_27, indicando que GPIO22 y GPIO27 son una salida.

```
SWI 0x16            @ OS_EnterOS
ldr r2, [r3, #8]    @ Lee GPFSEL2 para GPIO22 y GPIO27
ldr r4, =CLEAN22_27
bic r2, r2, r4      @ Limpia bits de GPIO22 y GPIO27
ldr r4, =OUT22_27
orr r2, r2, r4      @ Aplica patrón pin de salida
str r2, [r3, #8]    @ Guarda valor en GPFSEL2
```

A continuación, encenderemos los leds guardando un 1 en la posición asignada para GPIO22 y GPIO27 del registro GPSET0 (usaremos el patrón guardado en la constante ON_OFF). GPSET0 que se encuentra en el offset 0x1c (#28), como vimos en la Tabla 4.1.

```
mov r3, r0
ldr r1, =ON_OFF
str r1, [r3, #28]    @ Enciende leds verdes
```

Permaneceremos un rato en un pequeño bucle de espera antes de apagarlos, modificando para ello el registro GPCLR0 con offset 0x28 (#40). Tabla 4.1.

```
mov r2, #0                    @ Esta un rato encendido y apaga
loop:
      cmp r2, #0x10000000
      add r2, r2, #1
      ble loop
str r1, [r3, #40]             @ Apaga leds verdes
```

Antes de salir del programa, es necesario salir del modo supervisor. Para ello disponemos de la SWI OS_LeaveOS [ANEXO, Tabla A.1].

```
SWI 0x7C     @ OS_LeaveOS
_EXIT:
mov pc,lr
```

Por último, el programa principal llamado ***mainJuego.c*** está creado en código C. Este programa se encarga principalmente, de gestionar los turnos del jugador (10 turnos), pedir en cada turno las coordenadas deseadas de ataque, llamando posteriormente a la función ***ataca(int x, int y)*** que le retornará los resultados, permitiéndole dependiendo del resultado (Agua, Tocado o Hundido) actualizar la información del tablero y llamar al programa encargado de

encender los leds correspondientes (Agua llama a ***leds_green(),*** Tocado llama a ***leds_yellow()*** y Hundido llama a ***leds_red()).***

```
#include <stdio.h>
#define M 5
extern int ataca(int x, int y);
extern leds_yellow();
extern leds_red();
extern leds_green();
void imprime(char tablero[][M]);

int main(int argc, char *argv[])
{
	int tiradas=10;
	char tablero[M][M];
	int resultado=0;
	int b=0,x=0,y=0,i,j,p;
	for(i=0;i<M;i++){
		for(j=0;j<M;j++){
			tablero[i][j]='-';
		}
	}
	while(tiradas!=0 || resultado!=1){
		imprime(tablero);
		x=0;
		y=0;
		while(x>5 || x<1 || y>5 || y<1){
			printf("Introduce coordenadas \n");
			printf("x:");
			scanf("%d",&x);
			printf("\ny: ");
			scanf("%d",&y);
		}
		resultado = ataca(x,y);
		tiradas = tiradas-1;
		if(resultado==0 || resultado==1){
			if(resultado==0){
				printf("Agua!\n");
				tablero[x-1][y-1]='A';
				leds_green();
			}else{
				printf("Ganaste!\n");
				int i;
				for(i=0;i<5;i++){
					leds_green();
					leds_yellow();
					leds_red();
				}
				tiradas=0;
			}
		}else{
			switch(resultado){
```

```
                        case 11:
                              printf("b1 hundido!\n");
                              leds_red();
                              tablero[x-1][y-1]='X';
                              break;
                        case 2:
                              printf("b2 tocado!\n");
                              leds_yellow();
                              tablero[x-1][y-1]='X';
                              break;
                        case 3:
                              printf("b3 tocado!\n");
                              leds_yellow();
                              tablero[x-1][y-1]='X';
                              break;
                        case 21:
                              printf("b2 hundido!\n");
                              leds_red();
                              tablero[x-1][y-1]='X';
                              break;
                        case 31:
                              printf("b3 hundido!\n");
                              leds_red();
                              tablero[x-1][y-1]='X';
                              break;
                        default:
                              break;
                  }
            }
      }
      if(resultado!=1){printf("Game Over\n");}
      return 0;
}
```

La siguiente función se encarga de imprimir el tablero del juego por pantalla.

```
void imprime(char tablero[][M]){
      int i;
      printf("    BARCOS  HUNDIDOS\n");
      for(i=4;i>=0;i--){
            printf("------------------------\n");
            printf("%d  || %c | %c | %c | %c | %c ||\n", i+1,
            tablero[0][i], tablero[1][i], tablero[2][i],tablero[3][i],
            tablero[4][i]);}
      printf("--|||||||||||||||||||||\n");
      printf("  || 1 | 2 | 3 | 4 | 5 ||\n");
}
```

Ahora sólo nos queda crear los códigos objetos de mainJuego.c, ataca.s, leds_green.s, leds_yellow.s y leds_red.s. Para ello seguimos los pasos definidos en el punto 3.2.7.1. de la memoria y así comprobar que todo funciona según lo esperado.

- Creamos los códigos objetos para los códigos en C:
  gcc -c mainJuego.c
- Creamos los códigos objetos de los archivos en ensamblador ARM:
  as -o ataca.o ataca.s
  as -o leds_green.o leds_green.s
  as -o leds_yellow.o leds_yellow.s
  as -o leds_red.o leds_red.s
- Creamos un único ejecutable, que se creará en el directorio principal del proyecto, con el nombre HundirLaFlota:
  gcc -o HundirLaFlota mainJuego.o ataca.o leds_green.o leds_yellow.o leds_red.o

5.3. ENTRADA / SALIDA

En este apartado se indicará como crear programas de Entrada/Salida, por encuesta o interrupción, para dispositivos conectados a la Raspberry Pi.

5.3.1. POOLING

Mediante pooling o encuesta, podemos realizar una operación programada anteriormente en función de un evento que se ha producido desde el exterior en nuestra GPIO. Para ello será necesario realizar un programa que esté continuamente comprobando cada cierto tiempo si se ha producido algún cambio en el puerto que gestiona el pin de entrada de la GPIO que queremos testear.

Para poder realizar el pooling necesitamos realizar correctamente los siguientes pasos:

1. Indicar en el puerto GPFSELn donde se encuentre la GPIOn que deseamos programar. La GPIOn se trata de una entrada (000).
2. Testear cada cierto tiempo en el puerto GPLEVn (sólo lectura), situado en la dirección de memoria 0x20200034, para ver si ha cambiado el nivel del bit deseado.
3. Si se ha producido el evento, ejecutar la función deseada.

El código realizado para validar que todo funciona correctamente, consiste en que al presionar uno de los dos pulsadores de los que disponemos en nuestra placa (GPIO3), se enciendan los dos leds rojos.

Lo primero que hemos definido, son las constantes con las direcciones y desplazamientos que usaremos de forma habitual en nuestro código. Como indicamos en la sección 4.3, para acceder a la GPIO necesitamos la dirección base indicada en la constante GPBASE. En este programa necesitaremos acceder a los registros de la GPIO llamados GPFSEL0, GPFSEL1, GPSET0, GPCLR0 y GPLEV0 las constantes que hemos definido con sus nombres hacen referencia al desplazamiento necesario para acceder a su dirección como se muestra en la Tabla 4.1.

```
.align 4
.equ GPBASE, 0x20200000
.equ GPFSEL0, 0x00
.equ GPFSEL1, 0x04
.equ GPSET0, 0x1c
.equ GPCLR0, 0x28
.equ GPLEV0, 0X34
```

En primer lugar, calcularemos la dirección lógica en RISC OS de la dirección GPBASE, mediante el uso de SWI OS_Memory [ANEXO, Tabla A.1]. Esta dirección se guardará en el registro r3.

```
.global _start
.text
_start:
@ Calcula la dirección lógica en RISC OS
mov r0, #13         @ Instrucción para OS_Memory
ldr r1, =GPBASE
mov r2, #0x100
SWI 0x68            @ OS_Memory
mov r0, r3          @ r3 dirección base en Risc OS
```

Para poder modificar los registros que nos permitirán encender los leds y configurar el switch, será necesario pasar a modos supervisor, para ello disponemos de SWI OS_EnterOS [ANEXO, Tabla A.1]. Una vez que ya estamos en modo supervisor, modificaremos los registros GPFSEL0 y GPFSEL1, guardando en los bits correspondientes a GPIO9 y GPIO10 el patrón (Tabla 4.2) que indica que se tratan de una salida. Ejemplo en Figura 4.3.

```
SWI 0x16     @ OS_EnterOS
@ Configuramos gpio 9 y 10 como salida
mov r1, #0b00001000000000000000000000000000
str r1, [r0, #GPFSEL0]
mov r1, #0b00000000000000000000000000000001
str r1, [r0, #GPFSEL1]
```

Antes de que se pueda encender los leds rojos con el pulsador, realizaremos una comprobación de que se encienden correctamente. Para ello guardaremos un 1 en cada uno de los bits que hacen referencia a la GPIO9 y GPIO10 del registro GPSET0, indicando así que se enciendan. Permanecernos un rato en un pequeño bucle de espera antes de apagarlos, modificando para ello el registro GPCLR0.

```
@ Enciende leds
mov r1, #0b00000000000000000000001100000000
str r1, [r0, #GPSET0]
@ Bucle de espera hasta apagar
mov r2, #0
loop:
     cmp r2, #0x10000000
     beq endloop
     add r2, r2, #1
b loop
endloop:
@ Apaga leds rojos
mov r1, #0b00000000000000000000001100000000
str r1, [r0, #GPCLR0]
```

Para realizar la E/S por encuesta, permaneceremos en un bucle comprobando continuamente si se ha producido una entrada en el bit del registro GPLEV0 que representa al botón en GPIO3 (esto es si hay un 0 en bit 4).

```
mov r5,#1
loop2: cmp r5, #0
       beq endloop2
       @ Consulta si se ha pulsado el botón GPIO3
       ldr r1, [r0, #GPLEV0]
       and r5,r1,#0x08
       b loop2
endloop2:
```

Al salir del bucle se encenderán los dos leds rojos como en la fase de comprobación y posteriormente se apagarán.

```
mov r1, #0b00000000000000000000001100000000
str r1, [r0, #GPSET0]
mov r2, #0   @ Está un rato encendido y apaga
loop3:
     cmp r2, #0x10000000
     beq endloop3
     add r2, r2,#1
b loop3
endloop3:
@ Apaga leds rojos
mov r1, #0b00000000000000000000001100000000
str r1, [r0, #GPCLR0]
```

Antes de salir del programa, es necesario salir del modo supervisor. Para ello disponemos de la SWI OS_LeaveOS [ANEXO, Tabla A.1].

```
fin:
SWI 0x7C @ OS_LeaveOS
_EXIT:
mov r8,#0
SWI 0X11
```

Por último, solo nos quedaría ensamblar el código para generar nuestro código objeto y enlazar para generar nuestro programa final.

as -o <nombre>.o <nombre>.s

ld -o <nombre> <nombre>.o

5.3.2. INTERRUPCIÓN

Para la gestión de las interrupciones en RISC OS se dispone de una llamada al sistema (OS_ClaimDeviceVector), más información en [25], con esta llamada, podemos asignar una rutina de atención al vector del dispositivo que gestiona la interrupción del mismo. A la hora de crear la rutina que se encargará de gestionar la interrupción, hay que tener en cuenta los siguientes puntos:

- Gestionar la interrupción.
- Parar las interrupciones generadas por el dispositivo, si es necesario.
- Retornar usando MOV PC, R14

El código realizado para validar que todo funciona correctamente, consiste en configurar el timer para que se produzca una interrupción cada segundo, esta interrupción ejecutara una rutina que incrementara en 1 un contador que tenemos definido en memoria. Con el uso de este contador podremos mantener encendidos los leds de nuestra placa el tiempo que consideremos necesario.

Al empezar el código de nuestro programa, hemos definido una serie de constantes para poder trabajar con los leds como hemos realizado en programas anteriores, en este caso encenderemos y apagaremos todos los leds a la vez (siguiendo los pasos vistos en la sección 4.3.1.), también hemos definido la variable ***segundos*** que consiste en un contador del programa que nos permitirá determinar cuantos segundos han pasado.

```
.data
segundos: .word 0
.equ BASE_GPIO, 0x20200000
.equ OUT9, 0x08000000
.equ OUT10_11_17, 0x00200009
.equ OUT22_27, 0x00200040
.equ ON_OFF, 0x08420e00
.equ CLEAN10_11_17, 0x00E0003E
.equ CLEAN9, 0x38000000
.equ CLEAN22_27, 0x00E001C0
```

Lo siguiente será crear la rutina que se ejecutará cuando se produzca la interrupción, llamada ***handler***. Esta rutina lo primero que hará será limpiar el timer, para ello usaremos la SWI OS_Hardware [ANEXO, Tabla A.1] usando el *Hardware call number* de HAL_TimerIRQClear [ANEXO, Tabla A.4]. Después incrementaremos en 1 la variable *segundos* y por último y antes de retornar, indicaremos que la interrupción ha sido atendida, usando SWI OS_Hardware y el *Hardware call number de* HAL_IRQClear.

```
.text
.global _start
handler:
stmdb sp!,{r4-r12,r14}
mrs r10, cpsr
orr r0, r10, #3      @ Modo supervisor
msr cpsr_c, r0
mov r0, #1
mov r8, #0
mov r9, #113         @ HAL_TimerIRQClear
swi 0x7a
ldr r2, =segundos
ldr r1, [r2]
add r1, r1, #1
str r1, [r2]
ldr r0, [r2, #4]
mov r8, #0
mov r9, #3           @ HAL_IRQClear
swi 0x7a
msr cpsr_c, r10      @ Restore mode
ldmia sp!,{r4-r12, pc}
```

Ahora empezará el código del programa principal, indicaremos el timer a usar con SWI OS_Hardware [ANEXO, Tabla A.1] y usando el *Hardware call number de* HAL_TimerDevice [ANEXO, Tabla A.4], después usaremos la SWI OS_CalimDeviceVector [ANEXO, Tabla A.1] para asignar la rutina *handler* al *timer1*.

```
_start:
mov r0, #1      @ Timer1
mov r8, #0
mov r9, #13     @ HAL_TimerDevice
swi 0x7a        @ OS_Hardware
mov r7, r0
ldr r1, =handler
mov r2, #0
mov r3, #0
mov r4, #0
swi 0x4B        @ OS_ClaimDeviceVector
```

Para programar el reloj se usarán junto con la SWI OS_Hardware [ANEXO, Tabla A.1] los *Hardware call number de* HAL_TimerGranularity y HAL_TimerSetPeriod [ANEXO, Tabla A.4]

```
swi 0x16        @ OS_EnterOS
mov r0, #1
mov r8, #0
mov r9, #14     @ HAL_TimerGranularity
swi 0x7A        @ OS_Hardware
mov r9, #16     @ HAL_TimerSetPeriod
mov r1, r0      @ Granurality
mov r0, #1      @ Clock
mov r8, #0
swi 0x7A        @ OS_Hardware
swi 0x13        @ OS_IntOn
mov r8, #0      @ HAL
mov r9, #1      @ HAL_IRQEnable
mov r0, r7      @ irq of timer
swi 0x7A        @ OS_Hardware
swi 0x7C        @ OS_LeaveOS
```

Ahora ya está todo preparado para que podamos indicar que se enciendan los leds de la placa, en la siguiente sección se muestra el código utilizado para el encendido de los leds, como ya vimos en la sección 5.2.

```
mov r0, #13           @ Instruccion para OS_Memory
ldr r1, =BASE_GPIO
mov r2, #0x100
SWI 0x68              @ OS_Memory
mov r0, r3            @ En r3 dirección base en RISC OS
SWI 0x16              @ OS_EnterOS
ldr r2, [r3]          @ Lee GPFSEL0 para GPIO9
ldr r5, =CLEAN9
bic r2, r2, r5        @ Limpia bits de pin9
ldr r5, =OUT9
orr r2, r2, r5        @ Aplica patrón pin de salida
str r2, [r3]          @ Guarda valor en GPFSEL0
ldr r4, [r3,#4]       @ Lee GPFSEL1
ldr r5, =CLEAN10_11_17
bic r4, r4, r5        @ Limpia bits de pin10_11_17
```

```
ldr r5, =OUT10_11_17
orr r4, r4, r5        @ Aplica patrón pin de salida
str r4, [r3, #4]      @ Guarda valor en GPFSEL1
ldr r2, [r3, #8]      @ Lee GPFSEL2 para GPIO22 y GPIO27
ldr r4, =CLEAN22_27
bic r2, r2, r4
ldr r4, =OUT22_27
orr r2, r2, r4        @ Aplico patrón pin de salida
str r2, [r3, #8]      @ Guarda valor en GPFSEL2
ldr r1, =ON_OFF
str r1, [r3, #28]     @ Enciende los leds
SWI 0x7C              @ OS_LeaveOS
```

Para ajustar el tiempo que queremos que nuestros leds permanezcan encendidos creamos un bucle en el que leemos la variable global *segundos*, del que salimos cuando dicha variable alcanza el valor deseados, en este caso queremos que nuestros leds permanezcan encendidos 15 *segundos*, cuando la variable *segundos* alcanza el valor 16 salimos.

```
stmfd sp!,{r0-r1}
ldr r0, =segundos
loop:
ldr r1, [r0]
cmp r1, #16
bne loop
ldmfd sp!,{r0-r1}
```

Una vez que salimos del bucle ya podemos apagar los leds. Antes de salir de nuestro programa, debemos deshabilitar IRQ y liberar el vector del dispositivo, para ello usaremos SWI OS_Hardware [ANEXO, Tabla A.1] junto a HAL_IRQDisable [ANEXO, Tabla A.4] y SWI OS_ReleaseDeviceVector.

```
SWI 0x16              @ OS_EnterOS
str r1, [r3, #40]     @ Apaga los leds
SWI 0x7C              @ OS_LeaveOS
mov r8, #0
mov r9, #2            @ HAL_IRQDisable
mov r0, r7            @ irq of timer
swi 0x7A              @ OS_Hardware
mov r0, r7
ldr r1, =handler
mov r2, #0            @ Mismo que claiming
swi 0x4C              @ OS_ReleaseDeviceVector
swi 0x11
.end
```

Por último, solo nos quedaría ensamblar el código para generar nuestro código objeto y enlazar para generar nuestro programa final.

6.CONCLUSIONES

El cambio de una arquitectura MIPS a ARM se trata de un cambio importante, aunque necesario. Esta nueva arquitectura permitirá acercar más al alumno a la estructura de los computadores. Permitiéndole de una forma más cercana y accesible, conocer los principales componentes de los que está compuesto un computador.

Por otro lado, el uso de la Raspberry Pi junto con el sistema operativo RISC OS que nos facilita con sus SWIs el acceso a bajo nivel, nos permitirá abarcar un mayor número de asignaturas, permitiendo al alumno ver la relación existente entre ellas de forma más clara, algo necesario si se quiere tener un conocimiento adecuado de la estructura de un computador. Además, el procesador que nos encontramos en la Raspberry Pi, basado en la arquitectura ARM de tipo RISC (tan presente en nuestros días debido a su eficiencia, bajo consumo y coste), dispone de una amplia variedad de instrucciones para programar de forma más eficiente y sencilla, lo que nos permite enseñar al alumno el lenguaje ensamblador más fácilmente.

A nivel personal creo que es un gran acierto el cambio, ya que la Raspberry Pi permite aprender de forma más amena el código ensamblador que tan difícil se hace en algunas ocasiones. Además de tener como aliciente el poder trabajar con ella en casa, gracias a su precio reducido y el uso de dispositivos externos como son las placas de leds, pantallas LCD y sensores que unido a su fácil adquisición, hacen más educativo y atractivo el estudio de las diferentes asignaturas.

ANEXO

LLAMADAS AL SISTEMA EN RISC OS

En RISCO OS disponemos de una gran cantidad de llamadas al sistema que nos permitirán realizar distintas acciones. Estas llamadas son muy importantes, ya que nos permitirán en muchos casos acceder en modo supervisor al sistema con una sencilla syscall (necesario si deseamos realizar interrupciones, o acceder a la GPIO).

Ejemplo de uso en código ensamblador, de una SWI en RISC OS para entrar en modo Supervisor al sistema:

SWI 0x16 @ Llamo a OS_EnterOS

#Realizo las operaciones necesarias en modo supervisor

SWI 0x7C @Llamo a OS_LeaveOS

En la tabla A.1, estan algunas de las llamadas que han sido usadas en este proyecto, para acceder al listado completo de llamadas disponibles ir a la siguiente web [24]:

OS SWI Calls	Instrucción	Descripción	Entrada	Salida
OS_AddToVector	SWI &47	Esta llamada asigna la rutina indicada al vector seleccionado.	**R0** número de vector (tabla A.2). **R1** puntero a la dirección de la rutina a la que llama el vector. **R2** Valor que sera pasado en R12 cuando la rutina es llamada.	**R0** Preservado. **R1** Preservado. **R2** Preservado.
OS_ClaimProcessorVector	SWI &69	El propósito de esta llamada es declarar un vector de procesador.	**R0** bits 0-7: num. vector bit 8: 0 deshabilitar, o 1 para habilitar bits 9-31: reservado. **R1** Dirección de la rutina de remplazo. **R2** Si esta liberado: Dirección de tu rutina.	**R0** Preservado **R1** Si se habilita, dirección de la rutina original, else preserved. **R2** Preservado
OS_ClaimDeviceVector	SWI &4B	El propósito de esta llamada es declarar un vector de dispositivo.	**R0** Número de dispositivo y flag en bit 31 **R1** Dirección de la rutina del driver del dispositivo	**R0** Preservado **R1** Preservado **R2** Preservado **R3** Preservado

			R2 Valor que se pasará en R12 cuando se llame al driver **R3** Dirección del estado de interrupción si R0 = podule "IRQ" o "FIQ como IRQ" en la entrada **R4** Máscara de interrupción, si R0=podule "IRQ" o "FIQ como IRQ" en la entrada	**R4** Preservado
OS_ReleaseDeviceVector	SWI &4C	El propósito de esta llamada es eliminar un driver de las lista de llamadas del vector del dispositivo.	**R0** Número de dispositivo **R1** Dirección de la rutina del driver del dispositivo **R2** Valor que se pasará en R12 cuando se llame al driver **R3** Dirección del estado de interrupción si R0 = podule "IRQ" o "FIQ como IRQ" en la entrada **R4** Máscara de interrupción, si R0=podule "IRQ" o "FIQ como IRQ" en la entrada	**R0** Preservado **R1** Preservado **R2** Preservado **R3** Preservado **R4** Preservado
OS_EnterOS	SWI &16	Esta llamada se encarga de pasar a modo supervisor.	**-**	Todos los registros preservados.
OS_LeaveOS	SWI &7C	Esta llamada se encarga de pasar a modo usuario.	**-**	**-**
OS_Memory	SWI &68	El propósito de esta llamada es realizar varias operaciones para la gestión de memoria.	**R0** Código orden que se desea realizar. (Información en la web [20]) (bits 0–7). Flags (bits 8 - 31) que son específicas del código de orden. Todos los demás registros dependen del código de orden.	**R0** Preservado. Los demas registros dependen del código de orden introducido.
OS_WriteC	SWI &00	Imprime el carácter pasado a R0 por pantalla	**R0** Carácter a escribir	**R0** Preservado.
OS_Hardware 0	SWI &7A	Llama una rutina HAL. Las rutinas HAL internamente son ATPCS, por lo que R0-R3 se pasan como a1-a4, y R4-R7 se pasa a la pila. Los a1-a4 al salir de la rutina se pasan de nuevo a R0-R3. Si la rutina HAL no está disponible, se devuelve un error. Las rutinas HAL reales no devuelven errores RISC OS -	**R0-R7** Parámetros para la rutina hardware **R8** 0 **R9** Hardware call number	**R0-R3** Actualizados por llamada **R4-R9** Preservado

		cualquier posible fallo se indicará de una manera específica de la llamada.		
OS_IntOn	SWI &13	Esta llamada habilita las interrupciones.	-	-
OS_IntOff	SWI &14	Esta llamada deshabilita las interrupciones.	-	-

Tabla A.1: OS SWI Calls RISC OS

VECTORES HARDWARE

Offset	Name
&00	Reset
&04	Undefined instruction
&08	SWI
&0C	Prefetch abort
&10	Data abort
&14	Address Exception/Hypervisor trap
&18	IRQ
&1C	FIQ

Tabla A.2: Vectores Hardware

#	Vector
0	Branch through zero
1	Undefined instruction
2	SWI executed
3	Prefetch abort
4	Data abort
5	Address exception (ARMv2), Hypervisor trap (ARMv7VE, not used by RISC OS)
6	IRQ

Tabla A.3: Número vectores procesador

HARDWARE CALL NUMBER

En la tabla A.4, estan algunos de los *Hardware call number* que han sido usados en este proyecto, para acceder al listado completo ir a la siguiente web [14]:

HAL	HAL entry	Descripción	Entrada	Salida
HAL_TimerIRQClear void HAL_TimerIRQClear(int32_t timer)	#113	Esta entrada fue añadida originalmente para soportar una plataforma donde había varios *timers*, todos multiplexados a través de la misma interrupción en el controlador de interrupción principal.	**timer** número del *Timer*	-
HAL_IRQClear void HAL_IRQClear(int device)	#3	Esta llamada tiene un doble propósito: 1- Para borrar el estado de interrupción de las fuentes de interrupción bloqueadas. Algunos (no todos) puertos hardware pueden utilizar HAL_TimerIRQClear para este propósito. 2- Para activar el controlador de interrupciónes y comenzar a buscar nuevas fuentes de interrupción. Debido al punto 2, todas las rutinas de servicio de interrupción deben llamar a esta rutina después de dar servicio al dispositivo. Las versiones actuales del kernel no llamarán HAL_IRQClear - es su responsabilidad hacerlo.	**device** número del dispositivo a borrar el estado de la interrupción	-
HAL_TimerDevice int HAL_TimerDevice(int timer)	#13	Devuelve el número de dispositivo del *timer* n. El número de dispositivo IRQ para llamadas de interrupción.	**timer** número del *Timer* a consultar	Retorna el número de dispositivo (número IRQ) del *timer*
HAL_TimerGranularity unsigned int HAL_TimerGranularity(int timer)	#14	Devuelve la granularidad básica del *timer* n en ticks por segundo.	**timer** número del *Timer* a consultar	Devuelve la granularidad básica del *timer* n en ticks por segundo.
HAL_TimerSetPeriod void HAL_TimerSetPeriod(int timer, unsigned int period)	#16	Establece el período del *timer* n. Si periodo>0, el *timer* generará interrupciones cada (período / granularidad) segundos. Si período=0, el *timer* puede ser detenido. Esto en algúnos hardware puede no ser posible, por lo que la interrupción debe ser enmascarado además de llamar a esta función con período 0. Si periodo> maxperiod, el	**timer** número del *Timer* a modificar **Period** Periodo del *Timer*, en unidades de granularidad	-

		comportamiento es indefinido.		
HAL_IRQEnable int HAL_IRQEnable(int device)	#1	Modifica el controlador de interrupción para que el ARM reciba una IRQ cuando el dispositivo genere una interrupción. Para algunos controladores de interrupción, como IOMD, cualquier línea de interrupción dada sólo puede funcionar como un IRQ o un FIQ, y los números de dispositivo se asignan de forma independiente para cada uno. En estos casos, esta llamada simplemente desenmascara la interrupción en el controlador de interrupción. Otros controladores, tales como OMAP, soportan la dirección de cada línea de interrupción entre IRQ e FIQ independientemente, y un espacio numérico se utiliza para números de dispositivo en ambos tipos de interrupción. En estos casos, esta llamada configura adicionalmente la interrupción para activar un IRQ en lugar de un FIQ. Nota: El comportamiento no está definido si se intenta habilitar tanto IRQ como FIQ para el mismo dispositivo.	**device** número de dispositivo para habilitar las interrupciones	**0** Las interrupciones fueron deshabilitadas anteriormente para este dispositivo **Distinto de 0** Las interrupciones se habilitaron anteriormente para este dispositivo
HAL_IRQDisable int HAL_IRQDisable(int device)	#2	Modifica el controlador de interrupción de modo que ARM no recibe una IRQ cuando el dispositivo genera una interrupción. En el caso de los controladores de interrupción que requieren activar la búsqueda de nuevas causas de interrupción, si el dispositivo está actualmente interrumpiendo, esta llamada también debe activar el controlador de interrupción para comenzar a buscar nuevas interrupciones (asegurándose de que RISC OS no recibirá una interrupción desde este dispositivo Hasta que se vuelva a activar)	**device** número de dispositivo para deshabilitar las interrupciones	**0** Las interrupciones fueron deshabilitadas anteriormente para este dispositivo **Distinto de 0** Las interrupciones se habilitaron anteriormente para este dispositivo

Tabla A.4: Hardware call number

INSTRUCCIONES ENSAMBLADOR

Instrucciones ensamblador: Se trata de las instrucciones que serán ejecutadas por la CPU. Algunas de ellas se encargan de mover datos de un sitio a otro. Otras se encargan de realizar sumas, restas y otros tipos de operaciones de computo. Otros tipos de instrucciones se usan para realizar comparaciones y controlar que parte del programa se ejecutará después. A continuación, pasaremos a hacer un pequeño resumen de las operaciones más utilizadas.

- **Operaciones de comparación:** Cada una de las siguientes operaciones realiza una operación aritmética, pero los resultados son descartados. Solo modifican las *carry flags* del CPSR.

  Sintaxis:

  ```
  <op> {<cond>}  Rn, Operend2
  ```

  <op> puede ser cmp, cmn, tst o teq

  La que más usaremos cmp, que compara el contenido de los registros y cmn que compara negativos.

  <cond> puede ser cualquiera de los siguientes códigos:

<cond>	English Meaning
al	Always (this is the default <cond>)
eq	Z set (=)
ne	Z clear (≠)
ge	N set and V set, or N clear and V clear (≥)
lt	N set and V clear, or N clear and V set (<)
gt	Z clear, and either N set and V set, or N clear and V set (>)
le	Z set, or N set and V clear, or N clear and V set (≤)
hi	C set and Z clear (unsigned >)
ls	C clear or Z (unsigned ≤)
hs	C set (unsigned ≥)
cs	Alternate name for HS
lo	C clear (unsigned <)
cc	Alternate name for LO
mi	N set (result < 0)
pl	N clear (result ≥ 0)
vs	V set (overflow)
vc	V clear (no overflow)

Tabla A.5: Modificadores condicionales ARM [27]

- **Operaciones aritméticas:** Disponemos de varias operaciones aritméticas. Todas ellas necesitan de dos operandos y un registro de destino.

  Sintaxis:

  ```
  <op>{<cond>}{s}   Rd, Rn, Operand2
  ```

  <op> puede ser add, adc, sub, sbc, rsb o rsc

  Las que mas usaremos son add que suma (Rn + operand2) y sub que resta (Rn -operand2).

  La s es opcional, si la ponemos la instrucción modificara los bits del CPSR.

  <cond> puede ser cualquiera de los códigos que se encuentran en la Tabla A.5.

- **Operaciones lógicas:** Constan de dos operandos y un registro destino.

  Sintaxis:

  ```
  <op>{<cond>}{s}   Rd, Rn, Operand2
  ```

  <op> puede ser and, eor, orr, orn o bic

  Las que mas usaremos son and que realiza la operación and bit a bit y guarda el resultado en el bit destino y orr que realiza la operación or bit a bit.

  La s es opcional, si la ponemos la instrucción modificara los bits del CPSR.

  <cond> puede ser cualquiera de los códigos que se encuentran en la Tabla A.5.

- **Operaciones de movimiento de datos:** Copia los datos de un registro a otro.

  Sintaxis:

  ```
  <op>{<cond>}{s}   Rd, Operand2
   movt{<cond>}    Rd, #immed16
  ```

  <op> puede ser mov o mvn

mov copia operand2 a Rd, mvn copia el contenido del operand2 negado Rd y movt copia el inmediato de 16 bits dentro de los 16 bits mas significativos de Rd.

La s es opcional, si la ponemos la instrucción modificara los bits del CPSR.

<cond> puede ser cualquiera de los códigos que se encuentran en la Tabla A.5.

- **Operaciones de multiplicación:** Realizan las operaciones de multiplicación.

  Sintaxis:

```
mul{<cond>}{s}   Rd, Rm, Rs
mla{<cond>}{s}   Rd, Rm, Rs, Rn
```

  La instrucción mul multiplica y mla multiplica y acumula.

  La s es opcional, si la ponemos la instrucción modificara los bits del CPSR.

  <cond> puede ser cualquiera de los códigos que se encuentran en la Tabla A.5.

BIBLIOGRAFÍA

[1] Comandos disponibles para ejecutar en la terminal de RISC OS: https://www.riscosopen.org/wiki/documentation/show/*Commands

[2] Compilar ensamblador ARM y C: http://rpiplus.blogspot.com.es/2013/06/compilar-ensamblador-arm-y-c.html

[3] Debugger de RISC OS: http://www.riscos.com/support/developers/prm/debugger.html

[4] Debugger DDT: http://www.riscos.com/support/developers/dde/ddtold.html

[5] Debugger Breakaid: http://www.heyrick.co.uk/software/breakaid/

[6] Descarga RISC OS: https://www.riscosopen.org/content/downloads/raspberry-pi

[7] Descarga VNC para RISC OS: http://www.phlamethrower.co.uk/riscos/vnc_serv.php

[8] Descarga cliente VNC para Windows: https://www.realvnc.com/download/vnc/

[9] Descarga de GCC: http://www.riscos.info/downloads/gccsdk/latest/

[10] Documentación BCM2835: https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/BCM2835-ARM-Peripherals.pdf

[11] Ensamblar y enlazar: https://elpistolerosolitario.wordpress.com/2015/08/31/hola-mundo-en-ensamblador-x86-para-gnulinux/

[12] Enseñanza Práctica de Estructura y Organización de Computadores con Raspberry Pi. Cristóbal Camarero, Elena Zaira Suárez, Esteban Stafford, Fernando Vallejo, Carmen Martínez. Artículo publicado en las Jornadas Sarteco 2017.

[13] Evaluación de la plataforma Raspberry Pi para la docencia de Microprocesadores. Adrián Barredo Ferreira. Universidad de Cantabria.

[14] Hardware call number: https://www.riscosopen.org/wiki/documentation/show/HAL%20entry%20points%20by%20number

[15] Herramienta para la creación de imágenes de SO: https://sourceforge.net/projects/win32diskimager/

[16] Información GCC para RISC OS: http://www.riscos.info/index.php/GCC_for_RISC_OS

[17] Información OS_Memory: https://www.riscosopen.org/wiki/documentation/show/OS_Memory#reason

[18] Información Sensor Humedad: http://fpaez.com/sensor-dht11-de-temperatura-y-humedad/

[19] Información montaje Berryclip: http://www.raspberrypi-spy.co.uk/berryclip-6-led-add-on-board/berryclip-6-led-add-on-board-instructions/

[20] Información LCD: http://www.raspberrypi-spy.co.uk/2012/07/16x2-lcd-module-control-using-python/

[21] Información sensor HC-SR04: https://www.modmypi.com/blog/hc-sr04-ultrasonic-range-sensor-on-the-raspberry-pi

[22] Información modos direccionamiento en ARM: http://lorca.act.uji.es/libro/practARM/CAP04_ebook.pdf

[23] Introducción GPIO en Raspberry Pi: https://www.raspberrypi.org/documentation/usage/gpio-plus-and-raspi2/README.md

[24] Llamadas al sistema implementadas en RISC OS: https://www.riscosopen.org/wiki/documentation/show/OS%20SWI%20Calls

[25] Llamada al sistema OS_ClaimDeviceVector: https://www.riscosopen.org/wiki/documentation/show/OS_ClaimDeviceVector

[26] Modelos Raspberry Pi: https://www.raspberrypi.org/products/

[27] Modern Assembly Language Programming With The Arm Processor. Larry D. Pyeatt. Elsevier. ISBN: 978-0-12-803698-3

[28] Prácticas de Ensamblador Basadas en Raspberry Pi. Antonio José Villena Godoy, Rafael Asenjo Plaza, Francisco J. Corbera Peña. Departamento de Arquitectura de Computadores. Universidad de Málaga / Manuales.

[29] Primeros pasos con GCC en RISC OS: http://www.riscos.info/index.php/GCC_tutorial

[30] Procedure Call Estándar: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf

[31] Raspberry Pi, Assembly Language, Risc Os Beginners. Bruce Smith. First edition: 05 Febrero 2014 ISBN-13: 978-0-9923916-2-1.

[32] Universidad País Vasco: Guía docente de la asignatura, Estructura de Computadores.

[33] Universidad Jaime I, guía docente Estructura de Computadores: https://e-ujier.uji.es/pls/www/gri_www.euji22883_html?p_curso_aca=2016&p_asignatura_id=EI1004&p_idioma=es&p_titulacion=225

Todas las direcciones URL fueron comprobadas y validadas el 12 de julio de 2017.