Facultad
de
Ciencias

PORTADO DEL SISTEMA OPERATIVO M2OS A LA ARQUITECTURA RISC-V

(Porting the M2OS Operating System to the RISC-V Architecture)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Juan Romón Peña

Director: Mario Aldea Rivas

Co-Director: Héctor Pérez Tijero

Julio - 2024

Resumen

La arquitectura RISC-V ha ido adquiriendo popularidad en los últimos años. Actualmente existen dispositivos RISC-V para distintos entornos de aplicación, incluyendo sencillos microcontroladores para aplicaciones embebidas y de control.

M2OS es un sistema operativo, diseñado por el grupo ISTR de la Universidad de Cantabria, muy sencillo pensado para soportar aplicaciones concurrentes en microcontroladores con recursos de memoria muy limitados. Actualmente se encuentra disponible para microcontroladores con arquitectura ARM y AVR.

El principal objetivo de este proyecto consiste en realizar el portado del sistema operativo M2OS a la arquitectura RISC-V, en particular, a la placa Hifive1 con el SoC "Freedom E310" fabricada por SiFive.

Una parte fundamental del proyecto consistirá en modificar la capa de interfaz con el hardware de M2OS para adaptarla a la arquitectura RISC-V, así como implementar salida por consola y desarrollar un entorno de desarrollo cruzado que automatice la carga y depuración de las aplicaciones.

Palabras clave

RISC-V, Sifive, Portar ,Ada ,Emulador, Timer.

Abstract

RISC-V architecture has gained popularity in the last few years. At the moment there are multiple RISC-V devices for different application environments, including simple micro-controllers for control and embedded applications.

M2OS is an operating system developed by the ISTR group in the University of Cantabria. It is uncomplicated and intended to support concurrent applications on micro-controllers with very limited memory resources. At the moment, it is currently available for micro-controllers with ARM and AVR architectures.

The main objective of this project is to port the M2OS operating system to the RISC-V architecture, particularly to the Hifive1 board with the "Freedom E310" SoC manufactured by SiFive.

A fundamental part of the project will consist of modifying the hardware interface layer of M2OS to adapt it to the RISC-V architecture, as well as implementing console output and developing a cross-development environment that automates the loading and debugging of applications.

Keywords

RISC-V, Sifive, Port ,Ada ,Emulador, Timer.

En primer lugar, quiero expresar mi más sincero agradecimiento a mis amigos del instituto. Su amistad y apoyo incondicional han sido fundamentales a lo largo de este viaje académico.

A mis compañeros del programa Erasmus, les extiendo mi más profundo agradecimiento. La experiencia compartida en el extranjero ha sido enriquecedora tanto a nivel personal como profesional. Su compañía, colaboración y las inolvidables aventuras que vivimos juntos han dejado una huella imborrable.

A mis tutores del TFG por su ayuda y disposición.

Finalmente, quiero dedicar un agradecimiento muy especial a mi abuelo. Como ingeniero, ha sido una fuente constante de inspiración y un ejemplo a seguir de esfuerzo, perseverancia y ética de trabajo.

Contents

List of Figures

1. Introducción y objetivos

1.1 Motivación

RISC-V, una ISA abierta y libre, ha ganado una notable aceptación debido a su flexibilidad, escalabilidad y la robusta comunidad de desarrolladores que la respalda. Esta arquitectura, que permite una personalización profunda sin las restricciones de licencias propietarias, está revolucionando la forma en que se diseñan y implementan circuitos integrados desde pequeños microcontroladores a aceleradores hardware empleados para el entrenamiento de IAs.

La evolución de RISC-V ha sido rápida y significativa desde su creación en la Universidad de California, Berkeley. Su adopción ha crecido exponencialmente en diversos sectores, incluyendo la educación, la investigación y la industria. Este crecimiento es impulsado por la necesidad de soluciones de hardware eficientes, personalizables y sin los altos costes de licencia que tienen otras arquitecturas como x86 o ARM.

El IoT, que conecta dispositivos y sistemas a través de la red para mejorar la eficiencia y la toma de decisiones, requiere microcontroladores capaces de manejar múltiples tareas simultáneamente. Los sistemas embebidos, esenciales en la implementación del internet de las cosas, necesitan ser robustos, eficientes y capaces de operar en tiempo real. Aquí es donde RISC-V demuestra su potencial, ofreciendo una plataforma ideal para desarrollar soluciones embebidas complejas.

Estos sistemas embebidos o empotrados a diferencia de los computadores de propósito general, están diseñados para realizar una o pocas tareas exclusivamente. Esto implica que para cumplir con las limitaciones de energía y coste deben ser dispositivos muy simples. No obstante, estos sistemas embebidos requieren de RTOS (sistemas operativos de tiempo real), como M2OS, para facilitar su programación e interacción con el hardware.

Además de abordar las necesidades técnicas, este proyecto tiene un fuerte componente didáctico. La implementación de M2OS en RISC-V servirá como una valiosa herramienta educativa para estudiantes y desarrolladores, permitiéndoles explorar y aprender sobre sistemas embebidos y programación de bajo nivel en un entorno real. Este enfoque educativo no solo mejorará la comprensión teórica, sino que también proporcionará experiencia práctica, preparando a los futuros ingenieros para enfrentar los desafíos del desarrollo de sistemas embebidos y de tiempo real.

Figura 1.1: Logo RISC-V

1.2 Objetivos

El objetivo principal de este proyecto consiste en realizar el portado del sistema operativo M2OS a la arquitectura RISC-V, en particular, a alguna de las placas existentes en el mercado del SoC (System On a Chip) "Freedom E310" fabricado por Sifive.

El objetivo principal puede desglosarse en los siguientes objetivos secundarios:

- Configuración de un entorno de desarrollo cruzado que facilite la carga y depuración de aplicaciones, se busca proporcionar un entorno más robusto y flexible para la creación de aplicaciones embebidas y de tiempo real.
- Implementación de una salida por consola básica que utiliza la consola serie por USB.
- Portado de la capa de interfaz con el hardware de M2OS para adaptarla a esta arquitectura. Los distintos elementos que se implementarán: cambio de contexto, gestión del temporizador, gestión del stack, habilitación/deshabilitación interrupciones.

2. Herramientas, tecnologías y lenguajes

En este capítulo se exponen las herramientas, tecnologías y lenguajes empleados para el desarrollo del proyecto, su funcionalidad y su papel.

2.1 Tecnologías y lenguajes

2.1.1 Hifive1

Hifive1 es una placa de desarrollo compatible con Arduino que cuenta con el SoC SiFive Freedom E310 (FE310), lo que la convierte en plataforma adecuada para prototipar y desarrollar software para RISC-V[1]. Esta versión de la placa esta discontinuada y existe un modelo más reciente (Hifive Rev B) que añade un módulo Wifi y Bluetooth, hardware I2C para leer sensores digitales y UART adicional para comunicarse con otros periféricos[2].

Esta versión de la placa resulta muy atractiva realizar el portado de sistema operativo M2OS a la arquitectura RISC-V dado que posee algunas características interesantes:

- Coste asequible: se encuentra en la franja de lo 60€, asequible para la iniciación de los programadores en esta plataforma.
- Extensa documentación: SiFive pone a disposición de los usuarios, guías de inicio rápido, manuales, schematics y hojas de especificaciones.
- Amplia comunidad online: compuesta principalmente por foros en los que se intercambia información y dudas sobre desarrollo para RISC-V.
- Las especificaciones hardware, expuestas en el siguiente apartado, resultan idóneas para aplicaciones embebidas que permiten utilizar microcontroladores sencillos, económicos y de bajo consumo.

Características y especificaciones

El core FE310-G000 incluye un procesador RISC-V E31 de 32 bits con alto rendimiento y pipeline en orden con un pico sostenible de ejecución de una instrucción por ciclo de reloj. El core E31 tiene soporte para las extensiones estándar Multiply, Atomic y Compressed de RISC-V (RISC-V IMAC).

Cuenta con 16KB L1 de cache de instrucciones y 16KB datos SRAM. Adicionalmente, Hifive1 funciona a una frecuencia de reloj variable[3]. Además, dispone de 32 pines de los cuales, 1 puede ser usado como Wakeup, 19 pueden generar interrupciones externas, 9 pueden ser utilizados como salidas PWM, 19 como pines de e/s digital y 1 pin controlador SPI y 3 pines para chip select.

Esta placa funciona con una alimentación externa de 3.3V y 1,8V. Si bien un cable USB-A a Micro-B es necesario para la programación y comunicación, la placa pude funcionar una fuente externa de 7-12V DS, conectado a una fuente por USB, o por una batería.

Por último, la placa cuenta con varios LED. Dos están situados junto al conector USB y se encienden al conectar la placa a la corriente y un tercer LED RGB programable[4].

A nivel arquitectural el core E31 tiene soporte para dos modos de ejecución: modo máquina (M) y modo usuario (U), por defecto se opera en este primer modo a menos que se especifiquen los cambios cambios de modo [5].

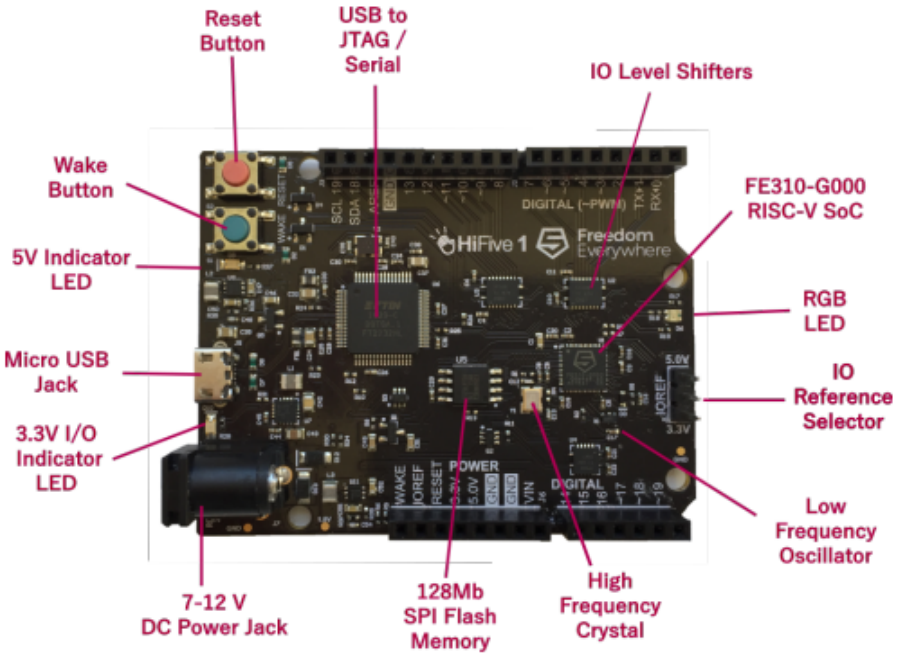

Figura 2.1: Componentes placa Hifive1

En cuanto a las interrupciones, el core E31 puede soportar interrupciones locales y (incluyendo software y timer) y globales [6].

- Las interrupciones locales se señalan directamente al hart[1] individual con un valor de interrupción específico. Las interrupciones software y del timer son locales y generadas por el Core Local Interruptor (CLINT).
- Las interrupciones globales, por otra parte, se enturan a través del Platform-Levewl Interrupt Controller (PLIC), que permite dirigir interrupciones a cualquier hart del sistema mediante una interrupción externa.

Las rutinas manejadoras de las interrupciones emplean el mismo stack que el resto de la aplicación a menos que se indique un cambio de manera explícita.

2.1.2 Ada

Inicialmente, el lenguaje de programación Ada fue creado para sistemas embebidos y en tiempo real, como es M2OS, pero con el tiempo se ha mejorado para incluir soporte a sistemas numéricos, financieros y a la programación orientada a objetos. En la actualidad, Ada se usa ampliamente en la industria aeronáutica, hardware militar, el sector aeroespacial y en cualquier sistema crítico donde un fallo en el software podría tener consecuencias graves para las personas y/o infraestructuras[7].

Ada es un lenguaje de programación con tipado fuerte y estático. Fue desarrollado por un equipo liderado por Jean Ichbiah de Honeywell Bull, bajo la solicitud del Departamento de Defensa de los Estados Unidos, con el objetivo de reemplazar los numerosos lenguajes de programación que utilizaban en esa época[8].

Para compilar los programas en Ada se emplea GNAT compiler, en la actualidad forma parte de la colección de compiladores GNU. Actualmente es mantenido por la empresa AdaCore.

[1]En RISC-V, un hart (hardware thread) es la unidad de ejecución que maneja las instrucciones del flujo de control. El core FE310-G000 cuenta con un solo hart.

Figura 2.2: Logo lenguaje Ada con eslogan

M2OS está escrito en su mayoría en este lenguaje, aprovecha sus funcionalidades para lograr que sea rápido, ligero y aproveche los recursos de los microcontroladores.

2.1.3 C

C es un lenguaje de programación de propósito general. Fue creado en la década de los 70s por Dennis Ritchie, y continúa siendo ampliamente usado e influyente.

Se trata de un lenguaje de tipo de datos estáticos, débilmente tipado, dispone de construcciones del lenguaje que permiten un control a bajo nivel pero empleando estructuras típicas de lenguajes de alto nivel. Además los compiladores ofrecen extensiones al lenguaje que posibilitan mezclar código ensamblador con código C o acceder directamente a memoria o dispositivos periféricos.

Existen muchos compiladores para C, aunque el empleado en el desarrollo de este proyecto se ha empleado el compilador cruzado de GCC para la arquitectura RISC-V [9].

Figura 2.3: Logo lenguaje C

M2OS nos permite desarrollar aplicaciones en C gracias a capa estilo Posix que implementa el sistema. Además, también se ha usado este lenguaje para la programación de dispositivos específicos como el timer empleando la librería de Freedom Metal.

2.1.4 Freedom Metal

Es una librería desarrollada por SiFive para escribir software portable para todas las RISC-V IP, RISC-V FPGA evaluation images, y placas de desarrollo de SiFive. Los programas escritos con la API de Freedom pueden ser compilados y ejecutados en todos los targets de RISC-V hechos por SiFive [10].

Freedom Metal proporciona:

- Entorno bare-metal para aplicaciones en C.
- Una API para controlar funcionalidades de la CPU, temporizador, interrupciones y periféricos.

Esto hace Freedom Metal sea adecuado para:

- Elaborar test hardware portables.
- Programación de aplicaciones bare-metal.

- Como capa de abstracción para portar sistemas operativos a RISC-V.

2.2 Herramientas

2.2.1 GDB

GDB o GNU Debugger es el depurador estándar para el compilador GNU. Es un depurador portable que se puede ejecutar en plataformas Unix y soporta múltiples lenguajes como: Ada, Ensamblador, C, C++, Fortran,etc [11].

GDB dispone de herramientas para analizar, diseccionar y modificar la ejecución de los programas con el fin de resolver errores de programación o realizar ingeniería inversa [12].

2.2.2 QEMU

QEMU es un emulador y virtualizador genérico y de código abierto. Emula el procesador de un computador mediante traducción binaria dinámica y puede correr sistemas operativos.

El uso de esta herramienta durante el desarrollo de este proyecto resulta atractivo ya que nos permite ejecutar M2OS y los programas pertinentes evitando tener que escribirlos en la memoria de la placa después de cada iteración de desarrollo del código.

2.2.3 Freedom Tools

Batería de herramientas de desarrollo embebido distribuida por SiFive para los dispositivos de la plataforma Freedom RISC-V.

Contenidos de Freedom Tools:

- RISC-V GNU NewLib Toolchain (riscv64-unknown-elf-*): conjunto de herramientas de desarrollo que incluye un compilador, enlazador, y biblioteca estándar (NewLib) para programar y construir aplicaciones para arquitecturas RISC-V.
- RISC-V OpenOCD (riscv-openocd-*): Open On Chip Debugger, es una herramienta de codigo abierto utilizada para la depuración, programación y pruebas de microcontroladores y otros dispositivos integrados.
- RISC-V QEMU (riscv-qemu-*): emulador de código abierto que permite ejecutar y probar software en una arquitectura RISC-V sin necesidad de hardware físico.
- SDK Utilities (sdk-utilities-*): son herramientas y scripts que facilitan el desarrollo, depuración y programación de aplicaciones en los microcontroladores Freedom de SiFive
- Trade Decoder (trace-decoder-*): herramienta que convierte datos de seguimiento (traza) de la ejecución de un programa en información legible y comprensible, útil para la depuración y análisis del rendimiento
- XC3SPROG (xc3sprog-*): herramienta utilizada para programar dispositivos FPGA.

2.2.4 GNAT Studio

GNAT Studio es un potente IDE desarrollado y distribuido por AdaCore que da soporte a la programación, testing y análisis de código. Por defecto incorpora soporte avanzado a los lenguajes de programación: Ada, SPARK, C, C++ y Python[13].

Además de soporte multilenguaje, también cuenta con herramientas avanzadas de navegación, organización de proyectos, Git como sistema de control de versiones.

Utilizar GNAT Studio durante el desarrollo de este proyecto resulta especialmente necesario puesto que cuenta con las características previamente mencionadas como soporte para los lenguajes Ada y C, que son los que principalmente emplearemos además de contar con herramientas de construcción de proyectos específicos de Ada, especificados en ficheros '.gpr'.

Para la edición de ficheros Makefile u otros que no son de Ada o C, se han usado de manera puntual otros editores de texto: Nano, un editor de texto para sistemas Unix ejecutado desde el terminal y Visual Studio Code, editor de texto altamente configurable desarrollado por Microsoft.

2.2.5 VirtualBox y Ubuntu

VirtualBox es un potente software de virtualización de arquitectura x86 de software abierto. Disponible para Windows, Linux y macOS. VirtualBox está en constante desarrollo con frecuentes actualizaciones.

Esta pieza software permite correr un software operativo distinto al que tengamos instalado de manera nativa en una máquina virtual. Estas máquinas virtuales pueden ser configuradas en múltiples aspectos como: interfaces de red, número de núcleos del procesador, cantidad de memoria RAM y almacenamiento, etc[14].

Ubuntu es una distribución de Linux basada en Debian, distribuida por Canonical, que fue lanzada oficialmente en 2004[15]. Incluye principalmente software libre y de código abierto, y se distribuyen versiones tanto para uso personal como para servidores.

El uso de Ubuntu en este proyecto se debe a que las herramientas utilizadas durante el desarrollo tienen soporte y especialmente Freedom E SDK es oficialmente soportado en las distribuciones para las que SiFive mantiene el Toolchain de RISC-V[16], Ubuntu y CentOS.

```
juan@tfg22:~$ uname -a
Linux tfg22 6.5.0-35-generic #35~22.04.1-Ubuntu
```

Figura 2.4: Version del kernel y Ubuntu

2.3 Sistema operativo M2OS

2.3.1 Introducción al sistema operativo

M2OS es un pequeño sistema operativo de tiempo real, que permite ejecutar aplicaciones multitarea en microcontroladores con memoria escasa. Este sistema operativo ha sido desarrollado por el departamento de Ingeniería Informática y Electrónica de la Universidad de Cantabria[17].

M20S implementa una política de planificación no expulsora que permite un alto aprovechamiento de la memoria. Específicamente, M2OS permite que el stack pueda ser compartido por todas las tareas del sistema, con lo cual, el tamaño del stack es igual al mayor tamaño de la pila necesario de entre todas las tareas. Consecuentemente, se puede aumentar el número de máximo de tareas que se pueden ejecutar en un microcontrolador [18].

En la actualidad M2OS se encuentra portado a múltiples arquitecturas:

- Arduino Uno
- Many-core Epiphany
- STM32F4

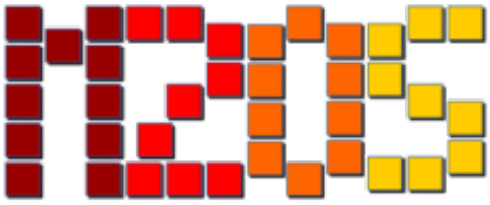

Figura 2.5: Logo de M20S

- Micro:bit

2.3.2 Estructura del sistema operativo

El modelo de capas de la implementación del sistema, se muestra en la Figura 2.6. El núcleo es la pieza de código que proporciona soporte para los threads M2OS y para las tareas de un solo disparo.

HAL define la interfaz abstracta con el hardware de M2OS. Incluye la gestión del timer, reloj, interrupciones, cambio de contexto y stack. El soporte de la funcionalidad estándar Ada lo proporciona el RTS.

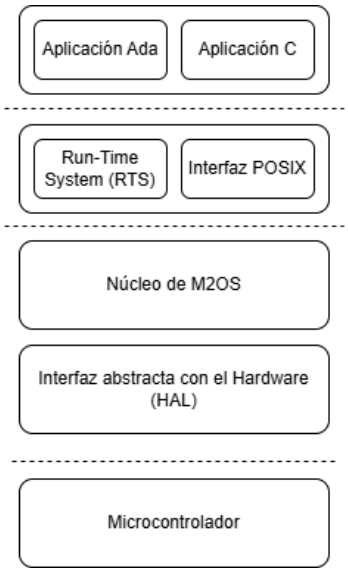

Figura 2.6: Capas de M2OS

A continuación se describen las carpetas que componen la distribución de M2OS y se expone brevemente su función dentro del sistema operativo:

2.3.3 Organización del código

- `adax/`: Directorio donde residen las extensiones a los paquetes estándar Ada definidas para M2OS.
- `arch/`: Aquí se encuentran las carpetas con el código específico de cada arquitectura. Cada carpeta corresponde a cada arquitectura contiene, al menos, los directorios hal/ y drivers/ y los ficheros de proyectos específicos de la arquitectura.
- `examples/`: Se encuentran ejemplos tanto de posix, ada_task y api_m2os cada uno en su respectivo subdirectorio.

- `gnat_rts/`: Run-time System para las distintas arquitecturas.
- `kernel/`: Código de la parte M2OS que no depende del hardware.
- `m2os_tool/`: Es una herramienta de transformación automática de código que permite convertir código multitarea a escrito en lenguaje Ada estándar (empleando la palabra reservada "`Task`") a código que utiliza la API de M2OS para crear las tareas.
- `posix/`: Contiene los ficheros de cabecera para proporcionar una capa estilo POSIX.
- `scripts/`: scripts en Python que definen los botones de ejecución en la placa y en el emulador para la arquitectura 'arduino uno'
- `tests/`: En este directorio se encuentran los test de POSIX, rendimiento, api_m2os y de tamaño.

En directorio raíz del proyecto hay una serie de ficheros esenciales para la instalación del mismo, estos son:

- `config.mk`: parámetros generales.
- `config_params.mk`: indica la arquitectura que se desea instalar y las opciones de compilación del código del usuario y del sistema operativo. Debe ser editado por el usuario.
- `rules.mk`: comandos de compilación de ficheros en los diferentes lenguajes y recetas para el borrado de ficheros generados durante la instalación
- `Makefile`: realiza la instalación o borrado de M2OS para la arquitectura indicada.

La compilación de M2OS se basa en proyectos GPR, los 3 que hay son:

- `global_switches.gpr`: se genera automáticamente por el Makefile, establece las opciones de compilación del RTS, el núcleo del sistema operativo y la aplicación.
- `shared_switches.gpr`: indica variables, librerías y compiladores para la construcción de aplicaciones.
- `m2os_riscv.gpr`: especifica los ficheros que componen el sistema operativo dentro del proyecto.

3. Freedom E SDK

Freedom E SDK es el kit de desarrollo de software mantenido y desarrollado por SiFive con el propósito de facilitar la iniciación en el desarrollo de software para las plataformas Freedom E y Freedom S Embedded RISC-V. Este SDK está desarrollado para funcionar en cualquier dispositivo soportado por las distribuciones del Toolchain de GNU RISC-V desarrollado por SiFive[19].

Este kit de desarrollo resulta esencial para el desarrollo del proyecto ya que es el conjunto de herramientas proporcionado por SiFive que nos permite compilar y ejecutar los programas a través de línea de comandos[20]. Otros métodos de desarrollo de software proporcionados por SiFive para Hifive1 son un IDE basado en Eclipse y un paquete de soporte para la placa en el IDE de Arduino, pero estos no pueden ser integrados junto a las herramientas de desarrollo para Ada.

3.1 Contenidos del SDK

Una vez instalado el SDK como se indica en el apéndice A podemos hacer uso de sus contenidos.

3.1.1 Documentación y Docker

Se proporciona un Dockerfile con el que crear una nueva imagen sobre la que instalar el las herramientas de desarrollo.

También se incluye un directorio `doc/` en el que podemos generar una documentación usando la herramienta Sphinx. Las instrucciones para generarla vienen detalladas en el fichero Readme del directorio.

Estas funcionalidades no han sido utilizadas durante el desarrollo de este proyecto.

3.1.2 BSP

Los ficheros de soporte para las placas de la librería de Freedom Metal se encuentran en un directorio independiente `bsp/target`, en nuestro caso `bsp/sifive-hifive1` y está compuesto de los siguientes ficheros.

- `design.dts`, `core.dts`: contiene la descripción DeviceTree de los dispositivos. Este fichero se emplea para parametrizar la librería Freedom Metal en el dispositivo que se va a usar.
- `metal.h`, `metal-inline.h`: ficheros de cabecera que son usados internamente para instanciar estructuras específicas de los dispositivos.
- `metal-platform.h`: cabecera que proporciona una serie de definiciones de preprocesador en C usados por Freedom Metal para indicar que la presencia de dispositivos y dotar de registros mapeados en memoria interactúan entre ellos.
- `metal.%.lds`: scripts de enlazado para el dispositivo. Los distintos scripts proporcionados permiten distintas configuraciones de memoria, durante el desarrollo del proyecto se ha utilizado el fichero por defecto `metal.default.lds` ya que es el que más se adapta a nuestras necesidades.
- `openocd.cfg`: utilizado para configurar OpenOCD para flashear y depurar el dispositivo.
- `settings.mk`: incluye una serie de parámetros que afectan a la compilación del sistema para el dispositivo como: string del ISA RISC-V, ABI seleccionado, modelo de código, etc.

3.1.3 Freedom metal

Freedom E SDK ya incorpora la librería Freedom Metal en un directorio específico `freedom-metal/` y así evita tener que instalarlo por separado. Freedom Metal hace posible que los programas de ejemplo sean portables a todos los dispositivos SiFive soportados.

Freedom Metal implementa una API para aprovechar los recursos del hardware, entre las funcionalidades proporcionadas por esta interfaz programable se encuentran:

- Gestión de interruciones: funciones para habilitar/deshabilitar interrupciones e instalar manejadores de interrupción.
- Manejo de temporizadores y reloj de tiempo real: control y configuración de temporizadores y del RTC.
- Control de E/S de propósito general: funciones para manejar pines de entrada y salida.
- Comunicación mediante SPI y UART: interfaz para la configuración y uso de los periféricos SPI y UART.
- Salida por consola por la línea serie sobre USB.

Estas funcionalidades e incluyen en las librerías `libmetal.a` y `libmetal-gloss.a` que se explican a continuación. El proceso para generar estas librerías e incorporarlos a nuestro proyecto se explica en la sección 4.2.2.

libmetal.a

Libmetal.a es una librería que forma parte de la abstracción de hardware llamada "Metal". Esta abstracción es proporcionada por el proyecto OpenAMP (Open Asymmetric Multi-Processing)[21]. La librería libmetal ofrece una interfaz de programación estándar para interactuar con el hardware, lo cual facilita la portabilidad del software entre diferentes plataformas de hardware.

libmetal.a se utiliza para:

- Abstracción hardware: Permite a los desarrolladores escribir código que puede ejecutarse en diferentes hardware sin cambios significativos, ya que libmetal maneja las diferencias de hardware.
- Gestión de recursos: Proporciona APIs para la gestión de recursos como memoria, interrupciones y dispositivos, facilitando la interacción con estos componentes del sistema.
- Soporte para multiprocesamiento asimétrico: facilita la comunicación y la gestión de recursos en sistemas que utilizan multiprocesamiento asimétrico (AMP), en nuestro caso no explotaremos esta funcionalidad.

Los contenidos de `libmetal.a` son los ficheros objeto que resultan de la compilación de los ficheros '.c' de `freedom-metal/src` y `freedom-metal/src/drivers`.

libmetal-gloss.a

`Libmetal-gloss.a`[1] es una librería que se apoya en `libmetal`, que incluye soporte para la implementación de librerías de bajo nivel en sistemas embebidos. Esto puede incluir funcionalidades específicas para facilitar la integración con otros componentes del sistema o para proporcionar soporte adicional a características específicas del hardware.

[1] Gloss es el acrónimo de "Generic Low-level Operating System Services"

Los contenidos de `libmetal-gloss`, contiene los ficheros objeto generados de `freedom-metal/gloss`.

3.1.4 Programas de ejemplo

El kit de desarrollo proporciona una batería de programas que muestran distintos ejemplos de como explotar las distintas funcionalidades del hardware, programación de timer, uso del GPIO, entre otros, y también el clásico programa 'Hello World'. Todos estos programas de prueba están almacenados en el directorio `software/`.

4. Portado de M20S

Como se indicó en la sección de objetivos, este proyecto busca portar el sistema operativo M2OS a la arquitectura RISC-V. En este capítulo cubriremos las partes fundamentales de este proyecto que son:

- Desarrollo de un entorno de desarrollo cruzado.
- Modificar la capa de interfaz con el hardware de M2OS para adaptarlo a la nueva arquitectura.
- Implementación de salida por consola para facilitar la interacción con el usuario y el desarrollo de aplicaciones.
- Procedimientos para facilitar la gestión del stack.

La parte principal del portado es la adaptación a RISC-V del paquete M2.HAL (ficheros `m2-hal.adb` y `m2-hal.ads` en el directorio `arch/riscv/hal`). Dicho paquete define la interfaz abstracta con el hardware (Hardware Abstract Layer) de M2OS, que incluye la gestión del temporizador (apartado 4.5), cambio de contexto (apartado 4.6), pila (apartado 4.7) e interrupciones (apartado 4.8)

Para cumplir estos objetivos se han usado las herramientas y tecnologías expuestas en el capítulo 2.

4.1 Entorno de desarrollo

El entrono de desarrollo que proporciona este proyecto, se compone de los siguientes elementos: compilador/enlazador de Ada para generar aplicaciones proporcionado por AdaCore ya que es el que tiene soporte para el lenguaje Ada, librerías de Freedom-metal para tener acceso al hardware, Qemu y la placa Hifive1 para la ejecución de programas, GDB para las labores de depuración y el entorno de desarrollo integrado GnatStudio para el desarrollo de aplicaciones y del proyecto.

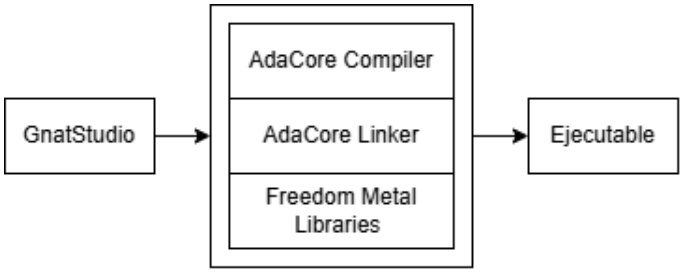

Figura 4.1: Generar un ejecutable

La Figura anterior muestra como es el proceso de generar un fichero ejecutable en el entorno, con GnatStudio editamos el código fuente y con el compilador/enlazador de AdaCore y las librerías de Freedom Metal compilamos los programas.

Para ejecutar un programa después de compilarlo, el entorno de desarrollo propone dos alternativas. La primera, en la que se graba el programa mediante OpenOCD (proporcionado en Freedom Tools) en la placa conectada al host a través de USB y observando la salida por consola serie a través de la misma conexión USB (visible a través del fichero de dispositivo `/dev/ttyUSB1` de nuestro computador de desarrollo). Esta opción (mostrada en la Figura 4.2) es poco versátil pero permite probar los programas en hardware real.

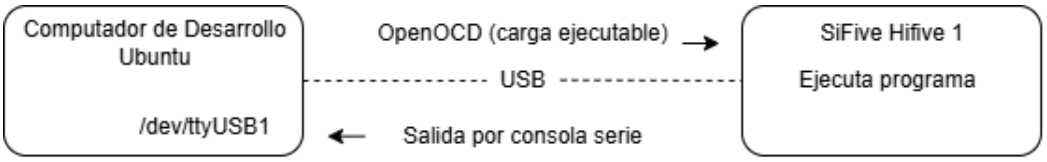

Figura 4.2: Ejecutar un programa

La segunda alternativa para ejecutar el programa, consiste en ejecutar los programas en Qemu, esta alternativa resulta interesante ya que es rápida y no requiere hardware externo y agiliza el proceso de desarrollo.

La siguiente Figura muestra los elementos empleados para la depuración, desde GnatStudio podemos ejecutar un *frontend* de GDB que se conectará al emulador de RISC-V que esté corriendo el programa que queremos depurar.

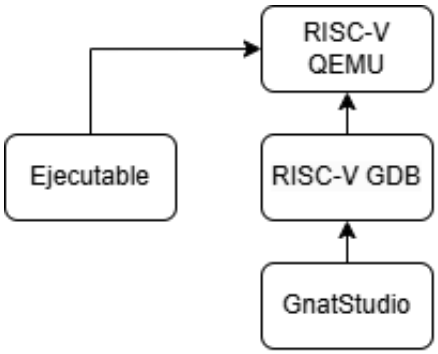

Figura 4.3: Depuración de un programa

Las instrucciones para la ejecución y depuración están indicados en el apéndice C.

4.1.1 Configuración de M2OS

Una vez las herramientas están preparadas, hay que configurar M2OS para la nueva arquitectura antes de instalar el sistema operativo y comenzar el desarrollo.

Primeramente creamos los subdirectorios `riscv/` tanto en `arch/` como en `gnatrts/`, en este primero se encontrará la capa de abstracción del hardware que tendremos que portar, en este segundo se encuentra el runtime system de Ada y el fichero de enlazado.

Añadimos una nueva arquitectura al proyecto en el fichero config.mk.

En el Makefile principal de M2OS añadimos la nueva arquitectura como opción de compilación de M2OS e indicamos las tareas específicas de la arquitectura se tiene que llevar a cabo, compilar el kernel y el RTS y descomprimir las librerías del SDK.

Indicamos la arquitectura para la que se va a construir M2OS modificando el fichero `cofig_params.mk` para que solo esté des-comentada la línea dedicada a RISC-V, también crearemos el directorio `arch/riscv/drivers`, que por el momento se mantendrá vacío hasta que implementemos la salida por consola 4.4.

La versión del RTS (runtime system) de Ada empleada en este proyecto es "light-hifive1", ya que es la única incluida en el compilador `gnat-riscv64-elf-linux64-13.1.0-1` para nuestra placa. Esta versión del runtime no incorpora soporte para tareas Ada, esto tendrá implicaciones en el

alcance de este proyecto ya que no se podrán ejecutar determinados programas que hagan uso de tareas Ada (pero sí de threads POSIX, como se verá más adelante).

Otro paso a realizar, es eliminar el código de arranque incluido en el RTS del directorio `gnat_rts/riscv/gnat/` para evitar que nos genere conflictos durante la instalación del sistema operativo.

Como ultimo paso, modificamos el fichero '`shared_switches_riscv.grp`' para configurar la depuración de programas desde el IDE GnatStudio.

```
1   package Ide is
2      for Program_Host use ":1234";
3      for Communication_Protocol use "remote";
4      for Debugger_Command use "riscv64-unknown-elf-gdb";
5   end Ide;
```

Listado 4.1: shared_switches_riscv.grp

4.1.2 Instalación de M2OS

Una vez el proyecto de M2OS está configurado para RISC-V lo instalamos utilizando el Makefile del directorio raíz del proyecto, especificamos que solo vamos a compilar e instalar el RTS y M2OS con el argumento 'install_no_tool'.

```
1    $ make intall_no_tool
```

Si la configuración es correcta, una vez M2OS se instale se mostrará el siguiente mensaje en la consola, como se muestra en la Figura 4.4.

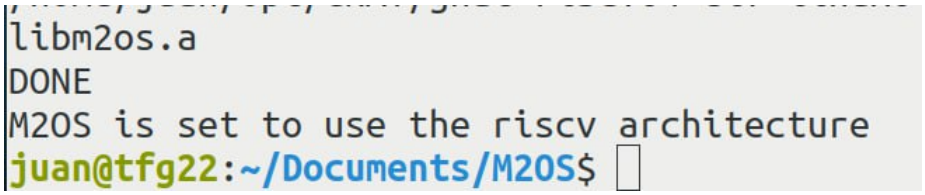

Figura 4.4: M2OS correctamente instalado

Es importante destacar que es necesario instalar M2OS de nuevo cada vez que realicemos modificaciones al kernel o a los ficheros de configuración de M2OS a lo largo del proyecto para que estos sean efectivos.

4.2 Integración con Freedom E SDK

4.2.1 Linker script

El *linker script* es un fichero cuyo principal propósito es describir como las secciones de código de los ficheros deben ser asignadas y la distribución de la memoria. De ser necesario, el *linker script* puede indicar al enlazador que debe realizar determinadas operaciones mediante comandos de enlazado [22].

El fichero de enlazado que utilziaremos para este proyecto es '`metal-default.lds`' que se encuentra en el directorio '`bsp/sifive-hifive1`' del SDK. Este fichero lo copiamos en la carpeta '`gnat_ts/riscv/ld`' y modificamos el fichero '`runtime.xml`' para especificar el *linker script* que debe utilizar.

```
1 <gprconfig>
2     <configuration>
3         <config><![CDATA[
```

```
4           /*MORE CORE*/
5           ...
6           case Loader is
7               when "ROM" =>
8                   for Required_Switches use Linker'Required_Switches &
9                       ("-T", "metal.default.lds");
10              when "USER" =>
11          end case;
12      end Linker;
13          ]]>
14          </config>
15      </configuration>
16  </grpconfig>
```

Listado 4.2: gnat_rts/riscv/runtime.xml

Al realizar este cambio, el enlazador del compilador de Gnat para RISC-V utiliza dicho script de enlazado.

4.2.2 Librerías de freedom metal

Para tener acceso a las funcionalidades del hardware y poder desarrollar aplicaciones para RISC-V con M2OS tenemos que hacer uso de los procedimientos proporcionados por Freedom Metal. Estos recursos vienen dados en las librerías `libmetal.a` y `libmetal-gloss.a` que se generan al compilar los ejemplos incluidos en el SDK.

Para facilitar la distribución de M2OS y que los futuros usuarios no tengan que descargar el SDK y generar las librerías, comprimiremos estas en un fichero .zip que añadimos al repositorio en el directorio `arch/riscv/freedom-e-sdk/`. Modificamos el `Makefile` del directorio raíz para que este fichero se descomprima a la hora de instalar el sistema operativo.

Indicaremos la ruta a estas librerías en el fichero '`shared_switches_riscv.grp`', y de esta manera serán accesibles por el *linker* al construir las aplicaciones M2OS..

```
1       ...
2       Linker_Trailing_Switches :=
3           ("-Wl,--start-group",
4           "-lm2os", "-lc" ,"-lgcc",
5            project'Project_Dir & "freedom-e-sdk/libmetal.a",
6            project'Project_Dir & "freedom-e-sdk/libmetal-gloss.a",
7            Runtime & "/adalib/libgnat.a",
8            "-Wl,--end-group"
9           );
10      ...
```

Listado 4.3: arch/riscv/shared_switches_riscv.gpr

4.3 Arranque y finalización de M2OS

Las secuencias de código que se ejecutan antes de la función principal '`main`' y después de la ejecución de un programa son proporcionadas por el SDK en distintos ficheros ensamblador incluidos en la librería "Freedom-Metal". Estos definen las rutinas de inicialización necesarias para proporcionar el entorno adecuado para la ejecución de un programa así como la finalización del mismo. Están pensados para funcionar en distintas plataformas hardware de la arquitectura RISC-V, aunque nos centraremos en aquellas funcionalidades utilizadas por el hardware empleado en este proyecto.

A continuación se explican las funcionalidades de los ficheros ensamblador utilizadas para el SoC Freedom E310.

4.3.1 scrub.S

Contiene las funciones encargadas de "limpiar" la memoria dejando todos los valores de la memoria a 0, de esta forma se evita que haya datos residuales que puedan interferir en la ejecución. También incluye una función para limpiar zonas de memoria concretas. Estas funciones se deben llamar antes de establecer el stack.

4.3.2 entry.S

El código que contiene está diseñado para ejecutarse antes del punto de entrada estándar `_start` de la biblioteca C en sistemas embebidos. Aquí se asegura de que la función `_enter`, que contiene el primer código a ejecutarse, pueda cargarse en una dirección específica.

En la función `_enter` se realizan las siguientes acciones:

- Definir un punto de entrada del código ensamblador
- Configura el registro del puntero global 'gp'. Es un registro utilizado para almacenar una dirección base que apunta a una región de datos globales en la memoria.
- Vacía la memoria (función definida en scrub.S)
- Establecer un puntero al stack.
- Llama a la función `_start` con los argumentos necesarios.

4.3.3 crt0.S

En este fichero se define la función `_start` la cuál sera ejecutada después de `_enter` y contiene una serie de rutinas que se realizan antes de la llamada al `main` del programa.

Entre estas funciones, las que se ejecutan en el SoC Freedom E310 son las siguientes:

1. Identificar el número harts del procesador.
2. Reubicar el segmento de datos a memoria RAM.
3. Copiar a la sección ITIM (Instruction Tightly Integrated Memory), esta es una memoria de solo lectura, en la que se copian secciones críticas de código desde la memoria ROM.[23]
4. Limpiar el segmento BSS (block started by symbol), que es la porción del ejecutable que tiene variables estáticas declaradas pero que aun no tienen valor. Debe ser puerto a 0 antes de la ejecución de la función main del usuario [24].
5. Establecer el puntero TLS (thread local storage pointer), para poder guardar valores deforma local a una tarea.[25].
6. En este punto, todas las funciones requeridas para la ejecución del programa se han llevado a cabo por lo que se puede proceder con la ejecución del programa.

Finalmente, el fichero `crt0.S` define la función `_exit` que incluye las acciones a realizar al finalizar la ejecución de un programa. Esta función debe ser llamada desde el paquete `M2.End_Execution` del núcleo de M2OS.

4.4 Implementación de salida por consola

Un aspecto importante para el desarrollo del proyecto y el desarrollo de aplicaciones para M2OS, es disponer de una salida por consola ya que facilita la depuración y permite que las aplicaciones puedan ser mas interactivas con usuarios.

4.4.1 Ingeniería inversa de printf

Para poder implementar una función similar al clásico `printf` primero tenemos que averiguar como poder mostrar caracteres por consola utilizado los métodos de freedom-metal. Podemos analizar los ejemplos proporcionados por Freedom E SDK para entender la implementación.

Analizamos el código ejecutable generado del ejemplo 'hello'. Como punto de partida, sabemos que la función estándar en C es `putc`[26], por lo que cabe pensar que la función correspondiente en freedom-metal seguirá una nomenclatura similar.

Para buscar está función realizaremos un análisis estático y un análisis dinámico. En el estático buscaremos si existen símbolos parecidos a `putc` sin necesidad de ejecutar el programa y en el análisis dinámico observaremos la pila de llamadas de la función `printf` con la ayuda de GDB y QEMU y así asegurarnos de que hemos encontrado el procedimiento adecuado.

Análisis estático

Durante la compilación del programa se generan 3 ficheros a parte del ejecutable:

- `hello.hex`: fichero con el contenido del ejecutable pero en hexadecimal.
- `hello.lst`: List file, contiene la lista de símbolos utilizados e información sobre el proceso de compilación.
- `hello.map`: fichero de texto que contiene información sobre el linkado del programa, información sobre los módulos utilizados y el nombre de su fichero, punto de entrada y símbolos con sus direcciones de memoria[27].

Utilizamos el fichero `hello.lst` para ver los símbolos utilizados y comprobar si se encuentra definido alguno similar a `putc`. Para ello usaremos la herramienta por línea de comandos `grep`.

```
juan@tfg22:~/opt/riscv/freedom-e-sdk/software/hello/release$ grep "putc" hello.lst
20401db2 g     F .text  0000000c metal_tty_putc
20403ad6 g     F .text  00000036 __metal_driver_sifive_uart0_putc
20401dc4 g     F .text  00000006 metal_uart_putc
20401b04:       247d                    jal     20401db2 <metal_tty_putc>
20401db2 <metal_tty_putc>:
metal_tty_putc():
20401dbc:       a021                    j       20401dc4 <metal_uart_putc>
20401dc4 <metal_uart_putc>:
metal_uart_putc():
```

Figura 4.5: Símbolos putc en hello.lst

Hemos encontrado varios símbolos que parece que implementan la función que nos interesa, `metal_tty_putc`, `metal_uart_putc` (Figura 4.5) .

En caso de no disponer de un fichero con la lista de símbolos utilizados, se puede usar una herramienta de desensamblado (`objdump`) la cual viene incluida en el *toolchain* de RISC-V. El argumento '-D' indica que se hace desensamblado del programa entero.

```
1 $ riscv64-elf-objdump -D release/hello.elf | grep "putc"
```

En la implementación de `metal_tty_putc` en el SDK (4.4) se puede ver como se llama a su vez a `metal_uart_putc`. Esto nos indica que posiblemente esta primera sea la función que estamos buscando y lo confirmaremos con el análisis dinámico.

```
int metal_tty_putc(int c) {
    return metal_uart_putc(__METAL_DT_STDOUT_UART_HANDLE, c);
}
```

Listado 4.4: freedom-metal/src/tty.c

Análisis dinámico

Con ayuda de GDB, depuraremos el programa como se indica en el apéndice C y estableciendo un punto de ruptura en '`metal_tty_putc`' para ver si cada vez que se llama a esta función se imprime un carácter por consola.

Para establecer un punto de ruptura en GDB utilizamos el comando '`break` <símbolo>' y para continuar la ejecución deteniéndonos en los puntos de ruptura usamos el comando '`continue`' o '`c`'.

```
(gdb) break metal_tty_putc
```

En la siguiente Figura (4.6) se ve cómo tras establecer un punto de ruptura, y ejecutar la función, se muestra por pantalla un carácter del string "¡Hello, World!". Esto confirma que este es el procedimiento que necesitamos para añadir salida por consola en M2OS.

Figura 4.6: Carácter "¡" mostrado durante la depuración

4.4.2 Driver de consola en M2OS

El driver de consola se define en el paquete `USB_Serial_Driver` (en el directorio `arch/riscv/drivers`) y debe seguir la estructura definida para los drivers de M2OS. La función clave de un driver de consola en M2OS es la función `Write`, que recibe como parámetros un puntero al buffer con los caracteres a imprimir y la longitud de dicho buffer (ver Listado 4.5).

En nuestra implementación se importa la función `metal_tty_putc` de la librería "Freedom-metal" la cual es llamada para cada uno de los caracteres incluidos en el buffer.

```
function Write(Buffer_ptr : System.Address ;
                       Bytes : size_t) return int is
  --build string
  Str : aliased Standard.String ( 1 .. Natural (Bytes)) with Address => Buffer_ptr
    ;
  function metal_tty_putc(c: int) return int
    with
      Import => True,
```

```
8         Convention => C;
9    Ret_val : int := 0;
10   Int_Value : Integer;
11   C_int : int;
12 begin
13   for I in Str'Range loop
14      --convert char to int
15      Int_Value := Character'Pos(Str(I));
16      --convert ADA Integer to C int
17      C_int := int(Int_Value);
18      --call metal_tty_putc
19      Ret_val := metal_tty_putc(C_int);
20   end loop;
21   return Ret_val;
22 end Write;
```

Listado 4.5: Implementación Write

4.4.3 Ejecución del programa "Hola mundo"

Con lo expuesto en este punto de la memoria, ya disponemos de un entorno que nos permite desarrollar aplicaciones sobre M2OS así como de las librerías "Freedom-metal" de acceso al hardware (que incluyen la gestión de la consola serie sobre USB). Por lo tanto, estamos en condiciones de realizar un ejemplo sencillo "Hola Mundo" que muestre la estructura de nuestras aplicaciones C sobre M2OS.

El Listado 4.6 muestra también cómo es una aplicación en C para M2OS. a diferencia de una aplicación normal, lo primero que hace es inicializar el sistema operativo (llamando a la función `m2osinit()`). Por lo demás es programa en C que imprime por pantalla "Main" hasta que el usuario decida detener el programa.

```
1  #include <stdio.h>
2  //**********//
3  //   main   //
4  //**********//
5  int main (int argc, char **argv) {
6    m2osinit();
7    while (1) {
8      puts("Main\n");
9    }
10   return 0;
11 }
```

Listado 4.6: Ejemplo Posix hello_word.c

Compilamos el programa desde GnatStudio y se creará un directorio obj/ en el que se guardará el fichero ejecutable. El siguiente paso es ejecutarlo tanto en el emulador como en la placa. En la Figura 4.7 se muestran los resultado de ambas pruebas. En el terminal de la izquierda el programa ejecutado con el emulador y en el de la derecha sobre la placa.

Este ejemplo nos confirma que cumplimos con uno del os objetivos principales del proyecto, proporcionar una entorno de desarrollo cruzado que facilite el desarrollo de aplicaciones.

4.5 Gestión del tiempo

El temporizador hardware es un dispositivo que puede ser programado para que provoque una interrupción cada cierto tiempo [28]. Estas interrupciones se usan para ejecutar los *Timing Events*

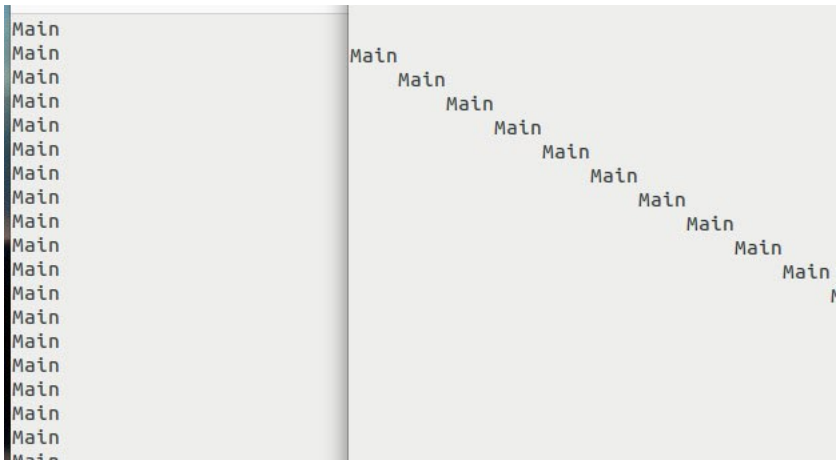

Figura 4.7: Ejemplo Posix en Qemu y Hifive1

de Ada [29]. Los *Timing Events* permiten programar la ejecución de una función definida por el programador en un instante dado de tiempo. La ejecución de dicha función se realiza en el contexto del manejador de la interrupción del temporizador.

Los *Timing Events* son ejecutados por M2OS desde la función `OS_Tick_Handler` del Listado 4.7. Esta es una función de M2OS que ya estaba implementada antes del desarrollo del portado. Esta función se exporta con el nombre `__gnat_irq_trap_rtc0` para ser utilizada por la parte del `HAL` escrita en lenguaje C.

Esta función inicializa el contador de eventos `Event_Order_Counter` utilizado por M2OS para mantener el orden FIFO sobre los eventos con el mismo instante de activación. Después, si los *Timing Events* están habilitados llama a `OS_Tick_Handler` para ejecutarlos.

```
1   ---------------------
2   -- SysTick_Handler --
3   ---------------------
4
5   OS_Tick_Handler : OS_Tick_Handler_Ac;
6   --  Handler for the timer interrupt.
7   procedure Sys_Tick_Handler;
8   pragma Export (C, Sys_Tick_Handler, "__gnat_irq_trap_rtc0");
9
10  procedure Sys_Tick_Handler is
11  begin
12     Events_Order_Counter := 0;
13     if M2.Use_Timing_Events'First then
14        OS_Tick_Handler.all (Get_HWTime);
15     end if;
16  end Sys_Tick_Handler;
```

Listado 4.7: Manejador de Ticks del sistema

4.5.1 Programación del temporizador

El temporizador de los sistemas RISC-V se maneja a través de dos registros del sistema mapeados en memoria `mtime` (machine time) y `mtimecmp` (machime time compare). Ambos registros tienen precisión de 64bits en sistemas de 32 y 64 bits.

El hardware incrementa el valor de `mtime` con una frecuencia constante generada por un RTC(real-time clock) que opera a 32.768Khz. Se produce una interrupción del timer cuando el valor en `mtime`

es mayor o igual que el valor de `mtimecmp`, por tanto, para generar interrupciones periódicas por software tenemos que ajustar el valor de `mtimecmp` [30].

La programación inicial del temporizador se realiza desde la función `Initialization` del paquete `M2.HAL` (fichero `m2-hal.adb`). En la arquitectura RISC-V, desde esta función se llama a la función C `timer_init`, implementada en el fichero `m2-hal-time.c` y mostrada en el Listado 4.9. Dicha función inicializa el core que se está utilizando, el controlador de interrupciones del core y el controlador de interrupciones del timer. La programación del temporizador se realiza con interrupciones deshabilitadas para evitar que se genere la interrupción antes de que M2OS haya finalizado su inicialización.

Por último se asigna la función manejadora de las interrupciones del timer. Esta función `time_handler`, mostrada en el Listado 4.8, llama al manejador de ticks de M2OS y a continuación, reprograma la siguiente activación del temporizador escribiendo el valor hexadecimal 0x3 en el registro `mtimecmp`, lo que equivale a un intervalo de tiempo cercano a los 0.1ms.

```
/**
 * @brief Handler function for the timer interrupt.
 * @param id The interrupt ID.
 * @param data Pointer to the CPU data.
 * */
void
timer_handler (int id, void *data)
{
  __gnat_irq_trap_rtc0();
  // 0xffff = 2s (example value)
  // 0,1ms = 3,27675 ticks (0x0003)
  // set timer
  metal_cpu_set_mtimecmp ((struct metal_cpu *)data,
                          metal_cpu_get_mtime ((struct metal_cpu *)data)
                            + 0x0003);
}
```

Listado 4.8: Función manejadora

```
#include <stdio.h>
#include <stdbool.h>

extern void __gnat_irq_trap_rtc0();

struct metal_cpu *cpu;

/**
 * @brief Initialize the timer.
 *
 * This function initializes the timer. It gets the current CPU, the CPU
 * interrupt controller, the timer interrupt controller, and the timer
 * interrupt ID. It registers the timer handler, enables the timer interrupt,
 * and enables the CPU interrupt.
 *
 * @return 0 if successful
 * @return 1 if the CPU is null
 * @return 2 if the timer is null
 * @return 3 if the CPU interrupt controller is null
 * @return 4 if the timer interrupt controller is null
 * @return 5 if the timer interrupt cannot be enabled
 * @return 6 if the CPU interrupt cannot be enabled
```

```
 */
int
timer_init (void)
{
  unsigned long long timeval, timebase;
  struct metal_interrupt *cpu_intr;
  struct metal_interrupt *tmr_intr;
  int rc, tmr_id;

  // Get the current CPU
  cpu = metal_cpu_get (metal_cpu_get_current_hartid ());
  if (cpu == NULL)
    {
      return 1;
    }

  timeval = 0;
  timebase = 0;
  // Get the CPU timer
  timeval = metal_cpu_get_timer (cpu);
  // Get the CPU timebase
  timebase = metal_cpu_get_timebase (cpu);
  if ((timeval == 0) || (timebase == 0))
    {
      return 2; // Error
    }

  cpu_intr
    = metal_cpu_interrupt_controller (cpu); // Get the interrupt controller
  if (cpu_intr == NULL)
    {
      return 3;
    }
  metal_interrupt_init (cpu_intr); // Initialize the interrupt controller

  tmr_intr = metal_cpu_timer_interrupt_controller (
    cpu); // Get the timer interrupt controller
  if (tmr_intr == NULL)
    {
      return 4;
    }
  metal_interrupt_init (tmr_intr); // Initialize the timer interrupt controller
  tmr_id = metal_cpu_timer_get_interrupt_id (cpu); // Get the timer interrupt ID

  rc = metal_interrupt_register_handler (tmr_intr, tmr_id, timer_handler, cpu);

  if (rc < 0)
    {
      return (rc * -1);
    }

  metal_cpu_set_mtimecmp (cpu, 0); // Set the timer compare register to 0
  if (metal_interrupt_enable (tmr_intr, tmr_id) == -1)
    { // Enable the timer interrupt
      return 5;
    }
  if (metal_interrupt_enable (cpu_intr, 0) == -1)
    { // Enable the CPU interrupt
```

```
81        return 6;
82      }
83    return 0;
84 }
```

Listado 4.9: Inicialización y programación del timer

4.5.2 Tiempo en M2OS

M2OS define el tipo `HWTime` para contabilizar el tiempo transcurrido desde el arranque del sistema en las unidades de cuenta del hardware. En la arquitectura RISC-V, este tipo de datos definido en `m2-hal.ads` es en realidad un entero de 64 bits sin signo.

```
1 type HWTime is new Interfaces.Unsigned_64;
```

En otras arquitecturas, la falta de precisión del timer hacía necesario el uso de una variable de tipo HWTime que se actualizase cada vez que se produjese una interrupción de timer para mantener una noción precisa del tiempo transcurrido. Dicha variable no es necesaria en nuestro caso, puesto que la arquitectura RISC-V dispone de un registro hardware en el que se van contabilizando el número de ticks del reloj de tiempo real que han transcurrido desde el arranque del sistema.

Desarrollamos una función que acceda a este registro y nos retorne el tiempo transcurrido, esta función se muestra a continuación.

```
1 /**
2  * @brief Get hardware time .
3  * The procedure gets the CPU from a global variable
4  * @return Current hardware time as unsigned long long
5  */
6 unsigned long long
7 get_hw_time ()
8 {
9   return metal_cpu_get_timer (cpu) * 2;
10 }
```

Listado 4.10: get_hw_time en C

Esta función implementada en el fichero `m2-hal-time.c` actúa como *wrapper* sobre la función proporcionada por Freedom-metal `metal_cpu_get_timer`, que nos retorna el valor almacenados en el registro del contador de tiempo real. El resultado de esta función se multiplica por 2 ya que la frecuencia de reloj es la mitad que la frecuencia del temporizador [31].

En el fichero `m2-hal.adb` la función `Get_HWTime` es la encargada de permitir que otros programas Ada puedan saber el tiempo hardware, aprovechamos la capacidad de Ada para utilizar funciones del lenguaje C [32] para llamar a la función que acabamos de implementar.

```
1    ----------------
2    -- Get_HWTime --
3    ----------------
4
5    function Get_HWTime return HWTime is
6       --SDK function wrapped to get the CPU cycle count timer value
7       function get_hw_time return HWTime
8         with
9           Import => True,
10          Convention => C;
11   begin
```

```
12        pragma Debug (DBG.Assert (Initialized));
13        return get_hw_time;
14    end Get_HWTime;
```

Listado 4.11: Get_HWTime en Ada

Como hemos dicho, `HWTime` es un tipo de dato que contabiliza el número de *ticks* del reloj de 32.768 KHz transcurridos desde el arranque del sistema y que, al tratarse de un valor entero, es muy eficiente para realizar operaciones dentro del núcleo del sistema operativo.

Sin embargo, la interfaz POSIX gestiona el tiempo como una cuenta de segundos y nanosegundos en una estructura `timespec`. Por su parte, el lenguaje Ada define el tipo real `Duration` para representar un intervalo temporal.

Para que M2OS haga las conversiones ente `HWTime` y `timespec`/`Duration`, definimos las contantes `HWTIME_HZ` en las que indicamos la frecuencia del temporizador.

Algunas de estas conversiones, han sido modificadas para la nueva arquitectura para evitar errores de redondeo provocados por operaciones con resultados decimales. Estos cambios (mostrados en el Listado 4.12) evitan que se produzcan resultados decimales y que se pierda precisión entre las operaciones.

```
1 # define TS(s, ns) {.hwtime = ((hwtime_t) (s)) * HWTIME_HZ + \
2                                 (hwtime_t) (ns) * HWTIME_HZ / NS_IN_S}
3 # define TS_NSEC(ts) (((ts).hwtime % HWTIME_HZ) * NS_IN_S / HWTIME_HZ)
4 # define TS2HWT(ts) ((hwtime_t) (ts).tv_sec * HWTIME_HZ + \
5                      (hwtime_t) (ts).tv_nsec * HWTIME_HZ / NS_IN_S)
6 # define HWT2TS(ts, hwt) \
7   (ts).tv_sec = (hwt) / HWTIME_HZ; \
8   (ts).tv_nsec = (hwtime_t) ((hwt) % HWTIME_HZ) * NS_IN_S / HWTIME_H
```

Listado 4.12: Conversiones de tiempo adaptadas

4.6 Cambio de contexto

El cambio de contexto es el mecanismo a través del cual M2OS permite se ejecuten múltiples tareas de manera alternativa sobre la misma unidad de procesamiento (CPU). El sistema operativo toma el control respecto a la tarea que se estaba ejecutando y decide si seguir ejecutando la misma tarea o una diferente, generalmente el sistema operativo retoma el control cuando se produce una interrupción, la mayoría de las veces del timer, o tras la finalización de una tarea [28].

En nuestro caso, M2 tiene una planificación no expulsora y las tareas de un solo disparo, por lo que el cambio de contexto no se produce después de una interrupción, solo se produce cuando una tarea se suspende o se bloquea. Las tareas no conservan el estado del stack entre activaciones (deben usar variables globales si quieren hacerlo).

En los casos en los que se decide cambiar de tarea, durante el cambio de contexto el estado de esta se debe de guardar para que una tarea entrante pueda ejecutarse. En los sistemas operativos de sistemas de sobremesa como Windows o Linux esta operación de cambio de contexto en la que se guardan y cargan estados de procesos puede resultar costosa. Un cambio de contexto en M2OS consiste en poner el contador del programa (PC) en la primera instrucción del cuerpo de la nueva tarea y el puntero de stack en la base del stack.

La implementación del cambio de contexto se encuentra en el fichero `m2_hal_regs.S` en el directorio `arch/riscv/hal` con el nombre `m2_hal_regs_jump_to_context` y se muestra en el siguiente bloque

de código.

```
.globl m2_hal_regs_jump_to_context

m2_hal_regs_jump_to_context:
    #a0 ->  new_task_body_address| a1 -> stack_top
    add sp, a1, zero #sp = r1+0
    jalr a0
```

Listado 4.13: Cambio de contecto M20S

La directiva `.globl` hace que la etiqueta `m2_hal_regs_jump_to_context` sea global y pueda ser referenciada desde otros archivos[33], en nuestro caso `m2-hal.adb`. Además este procedimiento recibe dos argumentos, el puntero a la primera instrucción del cuerpo de la tarea que se va a ejecutar y la dirección del stack de la tarea (stack base), en los registros `a0` y `a1` respectivamente de acuerdo a la convención de llamadas de la arquitectura RISC-V.

Escribimos en el registro Stack Pointer (SP) la nueva dirección del stack, para ello usamos la operación suma ya que esta nos permite copiar valores de un registro a otro ya que en RISC-V no contamos con una instrucción `mov`. Finalmente, saltamos a la instrucción indicada con `jalr`.

4.7 Gestión del stack

En el fichero `m2-hal.adb` se han implementado una serie de procedimientos que facilitan al programador hacer un buen uso del stack limitado del que dispone para las aplicaciones.

Los parámetros de tamaño del stack, inicio y fin del mismo son indicados por el *linker script*.

Los procedimientos relacionados con la gestión del stack en el paquete HAL de M2OS son:

- `Current_Stack_Top`: Retorna un puntero a la dirección actual del stack.
- `Global_Stack_Base`: Retorna un puntero de la dirección base del stack.
- `Get_Stack_Max_Size_In_Bytes`: Retorna el tamaño máximo útil del stack en Bytes.
- `Get_Current_Stack_Size_In_Bytes`: Retorna la cantidad de stack utilizada en Bytes.
- `Get_Current_Stack_Margin_In_Bytes`: Retorna la cantidad Bytes que quedan libres en el Stack.
- `Mark_Stack_Area`: Inicializa la memoria con valores `0xab`, a este valor hexadecimal lo llamaremos valor 'mágico' y lo usaremos para diferenciarlo de los valores introducidos por las aplicaciones en el stack. Esta función se llama durante el arranque del sistema.
- `Get_Max_Stack_Usage_In_Bytes`: Retorna la cantidad máxima de Bytes usados del stack por una aplicación. Cuenta la cantidad de Bytes del stack que no son el valor mágico. Para ello busca la posición del primer byte con un valor distinto al valor mágico. Existe la posibilidad de que el usuario escriba el valor mágico en el stack y en ese caso el número retornado de Bytes usados no sería correcto. Este procedimiento se llama durante la finalización de M2OS y se muestra el resultado al usuario por pantalla.

Para la realización de los cálculos, los procedimientos citados utilizan parámetros definidos en el *linker script* para conocer las direcciones de comienzo y fin del área destinado al stack y el tamaño del mismo.

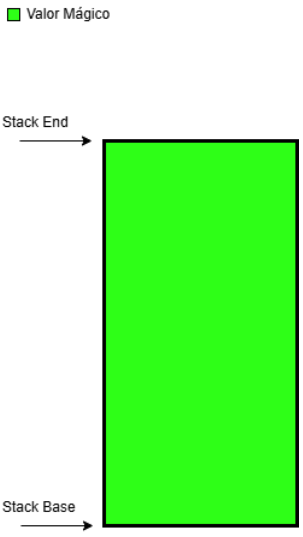

Figura 4.8: Mark_Stack_Area

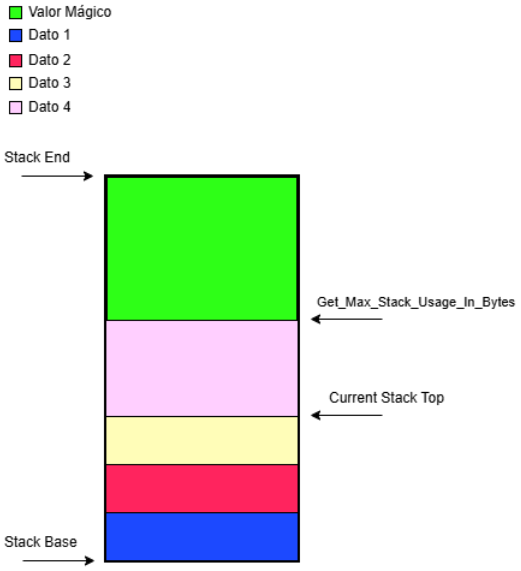

Figura 4.9: Get_Max_Stack_Usage_In_Bytes

En las Figuras 4.8 y 4.9 se muestra lo que realizan las funciones 'Mark_Stack_Area' escribiendo el stack completo con el valor mágico, y 'Get_Max_Stack_Usage_In_Bytes', retorna el valor de los bytes ocupados por por los bytes de Dato 1,2,3 y 4.

4.8 Habilitar y deshabilitar interrupciones

Generalmente, la capacidad de habilitar o deshabilitar interrupciones es esencial para que el kernel del sistema operativo pueda mantener la coherencia, seguridad y estabilidad del sistema. Permite al kernel gestionar adecuadamente las secciones críticas de código, sincronizar el acceso a recursos compartidos, y garantizar que las operaciones atómicas se completen sin interferencias, todo lo cual es crucial para el correcto funcionamiento del sistema operativo y sus aplicaciones.

M2OS es un sistema operativo mucho mas sencillo por lo que no tiene tantas necesidades estrictas de poder habilitar o deshabilitar interrupciones no obstante nos puede seguir resultando útil para momentos en lo que no queremos asegurarnos que no se produzcan interrupciones al realizar una operación especial por ejemplo, durante la inicialización del sistema operativo.

El registro `mstatus` lleva el seguimiento y controla el estado actual del procesador, incluyendo si las interrupciones están habilitadas o no [6]. En la tabla 4.1 se muestra un resumen de los campos relacionados con las interrupciones del registro mstatus.

Las interrupciones se habilitan igualando a 1 el bit MIE en mstatus, de la misma forma las interrupciones se deshabilitan cuando este bit vale 0 [34].

Para modificar el valor del registro mstatus, la ISA nos proporciona una serie de instrucciones [35]. Utilizaremos estas instrucciones directamente desde el código Ada de `m2-hal.adb` gracias a la capacidad que tiene el lenguaje de integrar código de bajo nivel [36].

Tanto para la instruccion `csr set` (csrs) como `csr clear` (csrc) indicamos el registro al que queremos acceder (`mstatus`) y el bit que queremos modificar mediante un valor inmediato, como sabemos que las interrupciones las maneja el campo MIE que ocupa el bit 3, el valor como entero inmediato es 8.

A continuación se muestra la implementación para habilitar y deshabilitar las interrupciones:

Machine Status Register			
Bits	**mstatus**		
CSR	**Field Name**	**Attr.**	**Description**
[2:0]	Reserved	WPRI	
3	MIE	RW	Machine Interrupt Enable
[6:4]	Reserved	WPRI	
7	MPIE	RW	Machine Previous Interrupt Enable
[10:8]	Reserved	WPRI	
[12:11]	MPP	RW	Machine Previous Privilege Mode

Tabla 4.1: Machine Status Register

```
1   -----------------------
2   -- Enable_Interrupts --
3   -----------------------
4
5   procedure Enable_Interrupts is
6   begin
7      System.Machine_Code.Asm("csrsi mstatus, 8", Volatile => True);
8   end Enable_Interrupts;
9
10  ------------------------
11  -- Disable_Interrupts --
12  ------------------------
13
14  procedure Disable_Interrupts is
15  begin
16     System.Machine_Code.Asm("csrci mstatus, 8", Volatile => True);
17  end Disable_Interrupts;
```

Listado 4.14: Enable and disable interrupts

Para comprobar si las interrupciones están habilitadas implementamos una función en lenguaje C que retorna un dato de tipo Bool. Se lee el valor del registro `mstatus` y se comprueba el valor del bit MIE.

```
1    /**
2     *@brief Check if interrupts are enabled.
3     * Check weather the interrupts are enabled or not. In order to use csr
     register
4     * instructions '_zicsr' need to be added to '-march=rv32imac' in runtime.xml
5     */
6    bool are_interrupts_enabled_c(void)
7    {
8      unsigned long mstatus;
9      asm volatile ("csrr %0, mstatus" : "=r" (mstatus));
10     return (mstatus & 0x8) != 0;
11   }
```

Listado 4.15: are_interrupts_enabled en C

Esta función en C es llamada desde el fichero m2-hal.adb

```
1   ----------------------------
2   -- Are_Interrupts_Enabled --
3   ----------------------------
```

```
4
5   function Are_Interrupts_Enabled return Boolean is
6      function are_interrupts_enabled_c  return Boolean
7        with
8          Import => True,
9          Convention => C;
10  begin
11     return are_interrupts_enabled_c;
12  end Are_Interrupts_Enabled;
```

Listado 4.16: Are_Interrupts_Enabled en Ada

5. Testing y pruebas

Ahora que las piezas fundamentales del portado están implementadas, en este capítulo explicaremos y detallaremos el entorno y los distinto programas de pruebas utilizados para comprobar y asegurarnos que la implementación de los elementos del portado es correcta.

5.1 Preparación del entorno

El entorno de pruebas de la implementación de M2OS está disponible en el directorio `tests/`. Los distintos programas de prueba se encuentran organizados en subdirectorios en función de la funcionalidad que se quiera probar.

Por cada grupo de test, tenemos que crear un fichero .gpr (GNAT project file) para la arquitectura RISC-V, de la misma manera que se ha hecho para la prueba de ejemplos basados en POSIX (apartado 4.4.3)

Además, para facilitar la tarea de compilación y ejecución de los programas de prueba se han Modificado los ficheros Makefile existentes, para añadir soporte a la nueva arquitectura y automatizar la ejecución de los test en el emulador. Adicionalmente se ha desarrollado un script específico para este proyecto que permite semi-automatizar el grabado y ejecución de programas en la placa Hifive1.

Por una parte los cambios realizados los ficheros Makefile, que al igual que los ficheros de proyecto hay uno por cada grupo, consisten en añadir el comando para la ejecución de un programa en el emulador y almacenar los resultados de esta ejecución en un ficheros auxiliares. Estos cambios se muestran en el Listado 5.1. Para abortar un test que se quede bloqueado se le da un margen de 10 segundos con el comando `timeout`.

```
...
else ifeq ($(M2_TARGET),riscv)
  # SIFIVE-HIFIVE1 target
  RUN_COMMAND = qemu-system-riscv32 -nographic -machine sifive_e -bios none -
    kernel
else
...
% : %.c
  @echo == Building test $< ...
  @gprbuild -P tests_*_$(M2_TARGET).gpr $< $(GPRFLAGS) -q

  @echo == Running test $@ ...
  timeout --foreground 10 $(RUN_COMMAND) $(notdir $@) 2>&1 | tee $(notdir $@).out
```

Listado 5.1: Cambios en los ficheros Makefile

Otra funcionalidad de estos ficheros, es proporcionar feedback al usuario sobre los test que han pasado y los que no utilizando el script `summary.sh` ya implementado previo al desarrollo de este proyecto.

Por otra parte, puesto que el proceso de grabar un programa en la placa y ejecutarlo manualmente resulta tedioso, conviene disponer de una herramienta que facilite esta tarea lo más posible. Por eso, para el desarrollo de este proyecto se ha desarrollado un script `run_test_on_board.sh` que convierte esta labor en un proceso semi-automático. El script se encuentra en el directorio `arch/riscv/` por

lo que se tiene que copiar a los directorios en los que se encuentren los tests. Por defecto el script ejecuta todos los ficheros ejecutables del directorio cuyo nombre termine en `_test`, en caso de querer ejecutar un test en concreto, el usuario debe indicar el nombre del test como argumento.

Cuando el script se ejecuta, se abre una ventana nueva de terminal en la que se muestra la salida por consola de los test en tiempo real, de mientras, la ventana original indica el nombre del test que se está grabado en la placa y cuando debe pulsar el botón de reset para ejecutarlo. Una vez se ha finalizado la ejecución de los tests (ver Figura 5.1), se muestra un resumen de los tests que han sido exitosos y lo que han fallado además de volcarse la salida por consola de los test en el fichero `tests_on_hardware_results.txt`.

```
====== ALL TESTS HAVE BEEN EXECUTED ======
See output logs in tests_on_hardware_results.txt
======= TESTS RESULTS =======
======= PASSED TESTS =======
periodic_thread_test
three_periodic_threads_test
timespec_hwtime_test
timespec_posix_test
two_periodic_threads_test
======= FAILED TESTS =======
```

Figura 5.1: Resultados de run_test_on_board.sh

Es preciso que antes de ejecutar de nuevo el script, el usuario haya cerrado la ventana de terminal abierta en la que se muestra la salida por consola en tiempo real.

5.2 Tests POSIX

En el subdirectorio `posix/` se encuentran una serie de programas de prueba cuyo propósito es comprobar el funcionamiento de la capa de POSIX en la nueva arquitectura.

- `timespec_hwtime_test.c`: Prueba las implementaciones de las macros de timespec definidas en time.h.
- `timespec_posix_test.c`: Prueba las implementaciones de las macros de timespec en el estándar de POSIX definidas en `time.h`, realiza las mismas conversiones que el la prueba anterior.
- `two_periodic_threads_test.c`: Prueba la implementación de la capa de POSIX de M2OS, creación y suspensión de tareas, mediante la ejecución de dos tareas periódicas, también se prueba el cambio de contexto al cambiar de tarea.
- `periodic_thread_test.c`: Ejecuta un solo thread de manera periódica y comprueba que la hora de activación de la tarea es la correcta. Es decir, en cada activación se comprueba que la hora de activación (medida con `clock_gettime`) coincide con la hora de activación que había sido programada con `clock_nanosleep`. Para que esta prueba se supere correctamente, la desviación típica de los tiempos de activación debe ser menor a 0,1ms. La primera ejecución de la tarea periódica no se tiene en cuenta ya que el tiempo de activación se ve afectado por los fallos de calentamiento en la caché de instrucciones.
- `three_periodic_threads_test.c`: Se ejecutan 3 threads con distintas prioridades y distintos períodos. El tests comprueba que la tarea que se ejecuta en cada momento es la de mayor

prioridad de las que están activas. En la tabla 5.1 se muestran los períodos de activación y tiempo de ejecución en segundos y la prioridad de cada tarea. Esta elección de parámetros permite comprobar como cuando llega una tarea de prioridad más alta mientras se ejecuta una tarea, esta se sigue ejecutando hasta que finalice y entonces se decida que tarea de las que están esperando debe ejecutarse en función de la prioridad. El diagrama temporal de la llegada y ejecución de las tareas se muestra en la Figura 5.1.

Tarea	**Período (seg)**	**Duración (seg)**	**Prioridad**
t1	2	1	Baja
t2	4	2	Media
t3	5	1	Alta

Tabla 5.1: Parámetros de planificación

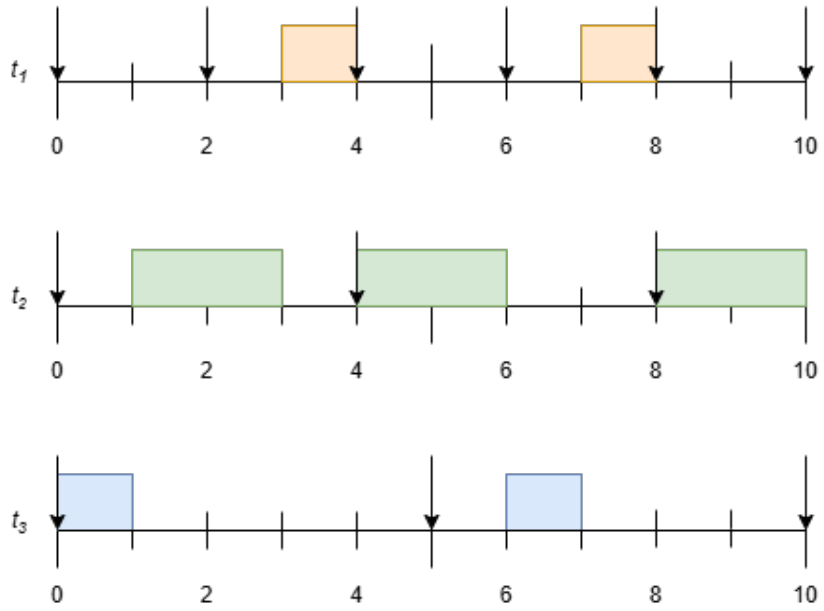

Figura 5.2: Diagrama temporal three_periodic_threads_test.c

```
 == Test summary ==
Test periodic_thread_test.out:   PASSED
Test three_periodic_threads_test.out:   PASSED
Test timespec_hwtime_test.out:   PASSED
Test timespec_posix_test.out:   PASSED
Test two_periodic_threads_test.out:   PASSED

Total tests: 5
     Passed: 5
     Failed: 0 (0 Known bugs)
----------------------------------------
```

Figura 5.3: Tests POSIX en Qemu

Ejecutamos los tests a través del Makefile y los resultados tras ejecutarlos en el emulador se muestran en la Figura anterior.

Los resultados de los tests ejecutados sobre hardware se muestran en la Figura 5.1.

Después de ejecutar los programas de prueba y que todos pasen correctamente podemos decir que

la capa POSIX funciona correctamente en RISC-V.

Los dos últimos tests se han hecho en este proyecto dado que había muy pocos tests paraprobar la interfaz POSIX de M2OS.

5.3 Otros tests de M2OS

Además de los tests de POSIX, en el directorio `tests/` disponemos pruebas adicionales organizadas en los siguientes subdirectorios `api_m2os/`, `ada_tasks/`, `performance/` y `performance_api_m2os`.

Debido a las limitaciones del *Runtime* de Ada empleado, y su falta de soporte para tareas hacen que la herramienta M2OS_tool (necesaria para las pruebas de rendimiento) y la gran mayoría de las pruebas no puedan ser compiladas, a excepción de `float_test.adb` que comprueba las operaciones suma, multiplicación y división de flotantes se realizan correctamente.

6. Conclusiones y trabajos futuros

6.1 Conclusiones

Este proyecto tenía como objetivo realizar el portado del sistema operativo M2OS desarrollado por la Universidad de Cantabria a la arquitectura RISC-V. Para ello hemos desarrollado un entorno de desarrollo cruzado que automatiza la carga y depuración de aplicaciones y hemos modificado la capa de interfaz con el hardware de M2OS para adaptarla a la nueva arquitectura.

De manera adicional se han añadido funcionalidades como la salida por consola.

La implementación desarrollada a lo largo del proyecto es completamente funcional y forma parte de la distribución de M2OS.

La creciente popularidad de RISC-V y el amplio uso de microcontroladores tanto en industria como a nivel educativo suponen oportunidad muy interesante para realizar un proyecto que mezcle incorpore ambos conceptos.

Este proyecto me ha permitido profundizar mis conocimientos adquiridos a lo largo del grado, especialmente en asignaturas como 'Sistemas de Tiempo Real' o 'Sistemas Embebidos', ambas de la mención de Ingeniería de Computadores.

Por último, haber participado en el desarrollo de un proyecto de código abierto me ha permitido ganar experiencia y perspectiva de como es la contribución a repositorios existentes de código.

6.2 Trabajos Futuros

Expongo a continuación las ideas para el avance del soporte de M2OS en RISC-V:

- Adaptación de un RTS con soporte para tareas Ada.
- Ejecución de las pruebas de `api_m2os` y de rendimiento así como de la compilación de M2OS junto a `m2os_tool`.
- Implementar funciones `Push_Function_Status` y `Pop_Function_Status` en la capa de abstracción del hardware que permite implementar la funcionalidad `Yield_To_Higher` de M2OS.

Bibliography

[1] SiFive Inc. *HiFive1 Freedom Everywhere*. https://www.sifive.com/boards/hifive1.

[2] SiFive Inc. *HiFive1 Rev B Freedom Everywhere*. https://www.sifive.com/boards/hifive1-rev-b.

[3] SiFive Inc. *SiFive FE310-G000 Manual v3p2*. 2021. Chap. 15.

[4] SiFive Inc. *SiFive FE310-G000 Datasheet v1p2*. 2021.

[5] SiFive Inc. *SiFive E31 Core Complex Manual v2p0*. 2018. Chap. 3.7.

[6] SiFive Inc. *SiFive FE310-G000 Manual v3p2*. 2021. Chap. 8.

[7] AdaCore. *Ada the languaje for safe, secure and reliable software*. https://www.adacore.com/about-ada.

[8] Wikipedia. *Ada (programming language)*. https://en.wikipedia.org/wiki/Ada_(programming_language).

[9] Wikipedia. *C (programming language)*. https://en.wikipedia.org/wiki/C_(programming_language).

[10] SiFive Inc. *Introduccion to Freedom Metal*. https://sifive.github.io/freedom-metal-docs/introduction.html.

[11] Wikipedia. *GNU Debugger*. https://en.wikipedia.org/wiki/GNU_Debugger.

[12] University of Maryland. *Using GDB for Reverse Engineering*. https://users.umiacs.umd.edu/~tdumitra/courses/ENEE757/Fall15/misc/gdb_tutorial.html.

[13] AdaCore. *GNAT Studio The simply powerful IDE*. https://www.adacore.com/gnatpro/toolsuite/gnatstudio.

[14] Oracle. *Welcome to VirtualBox.org!* https://www.virtualbox.org/.

[15] Canonical. *Ubuntu releases*. https://ubuntu.com/about.

[16] SiFive Inc. *Installing the Freedom E SDK, Supported Systems*. https://sifive.github.io/freedom-e-sdk-docs/userguide/installing.html.

[17] Universidad de Cantabria. *M2OS. RTOS with simple tasking support for small microcontrollers*. https://m2os.unican.es/.

[18] Mario Aldea Rivas and Hector Perez Tijero. "Leveraging real-time and multitasking Ada capabilities to small microcontrollers". In: *Journal of Systems Architecture* 94 (2019), pp. 32–41. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2019.02.015. URL: https://www.sciencedirect.com/science/article/pii/S1383762118302212.

[19] SiFive Inc. *Freedom E SDK Documentation*. https://sifive.github.io/freedom-e-sdk-docs/index.html.

[20] SiFive Inc. *SiFive HiFive1 Getting Started Guide 1.0.5*. 2021.

[21] OpenAMP. *OpenAMP*. https://www.openampproject.org/.

[22] HAW Hamburg. *1997*. https://users.informatik.haw-hamburg.de/~krabat/FH-Labor/gnupro/5_GNUPro_Utilities/c_Using_LD/ldLinker_scripts.html.

[23] SiFive Inc. *Instruction Tightly Integrated Memory (ITIM)*. https://sifive.github.io/freedom-metal-docs/devguide/itim.html.

[24] Wikipedia. *.bss*. https://en.wikipedia.org/wiki/.bss.

[25] FreeRTOS. *Thread Local Storage Pointers*. https://www.freertos.org/thread-local-storage-pointers.html.

[26] IBM. *putc () - putchar() - write a character*. https://www.ibm.com/docs/en/i/7.4?topic=functions-putc-putchar-write-characte.

[27] Microsoft. */MAP (Generate Mapfile)*. https://learn.microsoft.com/en-us/cpp/build/reference/map-generate-mapfile?view=msvc-170.

[28] Arpaci-Dusseau et al. "Operating systems: Three easy pieces". In: Arpaci-Dusseau Books, LLC, 2018. Chap. Mechanism: Limited Direct Execution.

[29] International Organization for Standardization. *ISO/IEC 8652:2012(E) with COR.1:2016 — Ada Reference Manual*. Timing Events. Feb. 2016. Chap. D.15.

[30] SiFive Inc. *SiFive FE310-G000 Manual v3p2*. 2021. Chap. 9.

[31] SiFive Inc. *SiFive E31 Core Complex Manual v2p0*. 2018. Chap. 4.1.

[32] AdaCore. *Interfacing with C*. https://learn.adacore.com/courses/intro-to-ada/chapters/interfacing_with_c.html.

[33] David Patterson and Andrew Waterman. "Guía Práctica de RISCV: El Atlas de una Arquitectu ra Abierta". In: *Strawberry Canyon LLC.(Primera Edición, 1.0. 5). Obtenido de http://riscvbook. com/spanish/guia-practica-de-risc* 1 (2018), p. 43.

[34] Andrew Waterman et al. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. 2021. Chap. 3.

[35] Andrew Waterman, Krste Asanović, and Sifive Inc. *The RISC-V Instruction Set Manual Volume I: User-Level ISA*. 2017. Chap. 2.8.

[36] AdaCore. *Inline Assembler*. https://docs.adacore.com/gnat_ugn-docs/html/gnat_ugn/gnat_ugn/inline_assembler.html.

[37] SiFive Inc. *Freedom E SDK Github Repository*. https://github.com/sifive/freedom-e-sdk.

[38] SiFive Inc. *Freedom Tools*. https://www.sifive.com/software.

[39] SiFive Inc. *Freedom Tools Releases*. https://github.com/sifive/freedom-tools/releases.

[40] Mario Aldea Rivas. *M2OS Repositorio oficial*. https://gitlab.com/marioaldea/M2OS.git.

A. Instalación de Freedom E sdk y Freedom-tools

Para la instalación de Freedom-e-sdk tomaremos como referencia el `README` del repositorio oficial del SDK en Github[37], si bien la guía de inicio rápido de Hifive1 ofrecida por SiFive también da instrucciones para instalar el SDK estas están obsoletas.

Para poder usar el SDK en nuestra máquina:

- GNU Make
- Git
- RISC-V GNU Toolchain
- RISC-V QEMU 4.1.0
- RISC-V OpenOCD (para usar en placas y FPGAs)
- Segger J-LINK (solo para su uso en determinadas placas de desarrollo)
- Python 3.5 o posterior
- Python Virtualenv
- Python Pip

Para descargarnos OpenOCD, el compilador cruzado de RISC-V y el emulador de QEMU, los podemos obtener tanto de la página oficial de SiFive, en el apartado de software [38], o del repositorio de Github de freedom-tools [39].

```
$git clone https://github.com/sifive/freedom-e-sdk.git
# openOCD y gcc
$cp openocd-<date>-<platform>.tar.gz /my/desired/location/
$cp riscv64-unknown-elf-gcc-<date>-<platform>.tar.gz /my/desired/location
$cd /my/desired/location
$tar -xvf openocd-<date>-<platform>.tar.gz
$tar -xvf riscv64-unknown-elf-gcc-<date>-<platform>.tar.gz
$export RISCV_OPENOCD_PATH=/my/desired/location/openocd
$export RISCV_PATH=/my/desired/location/

# qemu
$cp riscv-qemu-<version>-<date>-<platform>.tar.gz /my/desired/location
$tar -xvf riscv-qemu-<version>-<date>-<platform>.tar.gz
$export PATH=$PATH:/my/desired/location/riscv-qemu-<version>-<date>-<platform>/bin
```

Freedom E SKD cuenta con una serie de scripts en Python que son usados durante el proceso de compilación para parametrizar la compilación de Freedom Metal. Las dependencias de estos scripts están recogidas en `requirements.txt`. Freedom E SDK administra su propio entorno virtual, pero hay ciertas opciones que permiten al usuario configurar el entorno virtual para adaptarlo a sus necesidades, en nuestro caso mantendremos la configuración que viene por defecto.

Una vez cumplimos los requisitos previos podemos descargar los paquetes de Python necesarios:

```
$ make pip-cache
```

Por último ejecutamos el Makefile:

```
$ make
```

Si todo ha ido bien y no se han producido errores de dependencias el SDK está listo para usar y podemos probar a compilar y ejecutar alguno de los ejemplos como se muestra en el Apéndice B.

B. Instalación del compilador y el IDE de GNAT

Para instalar el compilador y GnatStudio utilizados en el proyecto seguiremos los siguientes pasos:

1. Instalamos el compilador para RISC-V que debemos usar puesto que tiene soporte para Ada, disponible en el apartado de releases del repositorio de GNAT-FSF-builds. Para este proyecto descargamos `gnat-riscv64-elf-linux64-13.1.0-1.tar.gz`.

2. Una vez descargado el fichero con extensión `.tar.gz`, lo descomprimimos. Añadimos la ruta de `gnat-riscv64-elf-linux64-13.1.0-1/bin` a nuestro `PATH`. en nuestro caso se encuentra en `$HOME/opt/GNAT/`

```
$ export PATH=$HOME/opt/GNAT/gnat-riscv64-elf-linux64-13.1.0-1/bin:$PATH
```

3. Descargamos el compilador de Ada nativo para poder utilizar su entorno de desarrollo (GnatStudio), utilizaremos `gnat-2021-20210519-x86_64-linux-bin` disponible en el apartado de descargas de la página oficial de AdaCore.

4. Una vez descargado, ejecutamos el fichero para comenzar la instalación. Cuando se complete, añadimos la PATH la dirección en la que hemos realizado la descarga.

```
$ export PATH=$HOME/opt/GNAT/2021/bin:$PATH
```

5. Instalamos dependencias de GnatStudio:

```
$ sudo apt install libncurses5
```

B.1 Ejecución de GnatStudio

Una vez instalados los compiladores y el entorno de desarrollo, ejecutaremos este último desde el terminal indicando la ruta al fichero `.gpr` del proyecto junto al argumento -P.

```
$ juan@tfg22:~/Desktop/M2OS-riscv$ gnatstudio -P examples/posix/
    examples_posix_riscv.gpr
```

C. Ejecución y depuración de programas

C.1 Ejecución y depuración de un ejemplo del SDK

Entre todos los ejemplos de programas proporcionados por el SDK, en este apéndice utilizaremos `hello` por simplicidad. Como ya se ha mencionado previamente en este proyecto, los comandos para compilar una aplicación están disponibles tanto en el repositorio oficial del SDK en Github y en la guía de inicio rápido. En nuestro caso hemos utilizado los indicados en el repositorio.

Para compilar una aplicación a bare-metal RISC-V:

```
$ make [PROGRAM=hello] TARGET=sifive-hifive1 [CONFIGURATION=release] software
```

`PROGRAM` indica el nombre de la aplicación que queremos compilar, `CONFIGURATION` permite especificar si queremos generar un ejecutable normal (`release`) o para depurar (`debug`).

C.1.1 Ejecución

Para correr los ficheros '.elf' podemos utilizar tanto QEMU como la propia placa.

Para ejecutar un programa en QEMU:

```
$ qemu-system-riscv32 -nographic -machine sifive_e -kernel <ruta al ejecutable>
Hello, World
```

Para ejecutar el programa directamente en la placa primero debemo grabarlo en la memoria:

```
$ make upload PROGRAM=hello BOARD=sifive-hifive1
```

También debeos disponer de algún administrador de ventanas, en este caso 'screen':

```
$sudo apt-get install screen
```

Para poder ver la salida por consola de nuestra aplicacion deberemos conectanos a la placa con el comando recién instalado y pulsar el botón de Reset (rojo) de la placa para ejecutar el programa.

```
$ sudo screen /dev/ttyUSB1 115200
```

C.1.2 Depuración

Para depurar programas combinaremos QEMU y GDB, para ellos debemos tener dos terminales abiertos. En uno de ellos lanzaremos el emulador con los flags de depuración

```
$ qemu-system-riscv32 -nographic -machine sifive_e -bios none -kernel <ruta al
    ejecutable> -S -s
```

En el otro terminal ejecutaremos `riscv64-unknown-elf-gdb` indicando la ruta del ejecutable que queremos depurar. Cuando se ejecute el depurador debemos indicarle a que puerto debe conectarse.

```
$ riscv64-unknown-elf-gdb software/hello/debug/hello.elf
(gdb) target remote localhost:1234
```

C.2 Ejecución y depuración de aplicaciones de M2OS

Para compilar los programas desarrollados a lo largo del proyecto, se utilizan los distintos Makefiles, también utilizados en otras arquitecturas y adaptados para ser utilizados en este proyecto.

C.2.1 Ejecución

Al igual que los ejemplos proporcionados por el SDK, las aplicaciones de M2OS pueden ser ejecutadas tanto en QEMU como en la propia placa, para este primero el procedimiento es idéntico:

```
$ qemu-system-riscv32 -nographic -machine sifive_e -bios none -kernel <ruta al
    ejecutable>
```

Pero para ejecutar los programas en la placa debemos primero averiguar como se graban en la memoria. Para ello, flasheamos un programa en la placa utilizando el siguiente comando que nos permitirá ver los comandos que se utilizan en el proceso.

```
$ make PROGRAM=hello TARGET=sifive-hifive1 upload -n
```

Vemos que lo último que se ejecuta es lo siguiente:

```
scripts/upload --elf /home/juan/opt/riscv/freedom-e-sdk/software/hello/debug/hello
    .elf --openocd /home/juan/Desktop/riscv-openocd-0.10.0-2020.12.1-x86_64-linux-
    ubuntu14//bin/openocd --gdb /home/juan/Desktop/riscv64-unknown-elf-toolchain
    -10.2.0-2020.12.8-x86_64-linux-ubuntu14/bin/riscv64-unknown-elf-gdb --openocd-
    config bsp/freedom-e310-arty/openocd.cfg
```

Es cuestión de utilizar este mismo comando sustituyendo el ejecutable `hello.elf` por la aplicación de M2OS que se desee para grabar el programa en la placa. Sabiendo esto, para ver y ejecutar el programa en la placa el proceso es el mismo que para los ejemplos del SDK.

C.2.2 Depuración

El proceso de depuración de aplicaciones de M2OS a través de consola de comandos es el mismo que para los ejemplos del SDK, con la diferencia de que se deben cambiar los flags de compilación que se especifican en el fichero `config_params.mk` en la raíz del proyecto.

```
#
# Compilation flags for RTS and M2OS lib
#
CONFIG_DEBUG_RTS=y              # -g -O0 flags
#CONFIG_ASSERTS_RTS=y            # -gnata -gnato flags
#CONFIG_SIZEOPTIMIZATION_RTS=y   # -Os -gnatp

#
# Compilation flags for M2OS
#
CONFIG_DEBUG_M2=y              # -g -O0 flags
#CONFIG_ASSERTS_M2=y            # -gnata -gnato flags
#CONFIG_SIZEOPTIMIZATION_M2=y   # -Os -gnatp

#
# Compilation flags for applications
#
CONFIG_DEBUG_APP=y              # -g -O0 flags
#CONFIG_ASSERTS_APP=y            # -gnata -gnato flags
#CONFIG_SIZEOPTIMIZATION_APP=y   # -Os -gnatp
```

También podemos depurar aplicaciones con GnatStudio, que nos proporciona una interfaz más cómoda con la que trabajar. Tendremos ejecutar el programa en un terminal con qemu, junto a los parámetros -S -s y desde el IDE seleccionaremos Debug, initalize y escogeremos el programa que queremos depurar.

D. Uso de Gitlab a lo largo del proyecto

Gitlab es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git. Esta platataforma ha sido utilizada a lo largo del proyecto para mantener un histórico y un control de las modificaciones realizadas en el código fuente.

El código fuente del sistema operativo se encuentra alojado en el repositorio oficial de M2OS[40], administrado por mi tutor de proyecto Mario Aldea. Cada rama del repositorio está asignada a una arquitectura o plataforma diferente, en nuestro caso realizaremos nuestras modificaciones en la rama de riscv.

Para empezar a trabajar debemos clonar el repositorio y situarnos en la rama que nos corresponde:

```
$ git clone git@gitlab.com:marioaldea/M2OS.git
$ git checkout riscv
```

Para subir los cambios al repositorio remoto de Gitlab, se deben ejecutar los siguientes comandos:

```
$ git add archivo/s o directorios
$ git commit -m "Comentario"
$ git push origin riscv
```

Es importante tener en cuenta especificar que ficheros no se quiere hacer seguimiento indicándolo en el fichero '.gitignore' y ejecutar el comando 'git status' para saber que ficheros se han modificados y cuales se quiere añadir en el commit.