Facultad de Ciencias

PORTADO DE M2OS AL MICROCONTROLADOR RP2040

(M2OS PORT TO THE RP2040 MICROCONTROLLER)

Trabajo de Fin de Grado

para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Carlos Mediavilla Martínez

Director: Mario Aldea Rivas

Co-Director: Hector Perez Tijero

Junio - 2024

Resumen

El objetivo de este proyecto es portar M2OS, el sistema operativo de tiempo real para placas con recursos restringidos desarrollado por la Universidad de Cantabria, al microcontrolador RP2040, integrado en la placa Raspberry Pi Pico.

Para ello es necesario entender cómo funciona un sistema operativo —independientemente de sus requerimientos— teóricamente y en código. Esto incluye un análisis de ciertas partes del funcionamiento de M2OS y Ada, el lenguaje en el que está escrito. Paralelamente, hay que analizar el entorno de desarrollo de software de la Raspberry Pi Pico, que es la manera más fácil y eficiente de interactuar con el hardware de la placa, escrito en C, siendo necesario integrar partes en ambos lenguajes.

Dicha integración de M2OS con el RP2040 involucra la modificación y expansión de ciertos módulos del sistema operativo, permitiendo que este acceda y controle los recursos del hardware de manera eficiente a través de funciones del entorno de desarrollo. Este proceso incluye la implementación del timer del sistema mediante el hardware de la RP2040, así como la inicialización del hardware y la gestión de la multitarea en un entorno de doble núcleo para realizar ciertas tareas básicas que se pueden expandir más adelante.

Palabras clave

Tiempo Real, Microcontroladores, M2OS, Raspberry Pi Pico, PicoSDK, RP2040

Abstract

This project aims to port M2OS, a real-time operating system developed for scarce-resource boards by Universidad de Cantabria, to the RP2040 microcontroller integrated in the Raspberry Pi Pico board.

It is thus necessary understand how Operating Systems work —no matter its requirements— This includes a detailed analysis of certain operational aspects of M2OS and its programming language, Ada. Simultaneously, a comprehensive understanding of the Raspberry Pi Pico's software development kit is required. This kit, written in C, represents the most efficient and straightforward method to interface with the board's hardware. Therefore, integrating components written in both Ada and C is a necessary part of this process.

The integration of M2OS with the RP2040 microcontroller involves modifying and expanding certain modules of the operating system. This enables the system to access and control the hardware resources efficiently through the development kit's functions. Key tasks in this integration include implementing the system timer using the RP2040's hardware clock, initializing the hardware, and managing multitasking in a dual-core environment to fulfil basic tasks that can be expanded upon later.

Key Words

Real Time, Microcontrollers, M2OS, Raspberry Pi Pico, PicoSDK, RP2040

1. Introducción

1.1 Motivación

Los sistemas embebidos son una parte integral de la vida cotidiana, presentes en una multitud de dispositivos y aplicaciones. Si bien muchos sistemas embebidos, como los encontrados en dispositivos móviles, consolas de videojuegos o sistemas de telecomunicaciones, no requieren estrictas restricciones temporales, existe una amplia gama de aplicaciones críticas que sí lo hacen. Estas aplicaciones, que incluyen sectores como el automovilístico, la aviónica y los sistemas industriales, demandan respuestas computacionales en tiempo real para mantener su funcionamiento eficiente y seguro. Un desafío común en estos sistemas es operar con recursos limitados, como energía y memoria, lo que exige soluciones optimizadas para manejar estas restricciones.

M2OS, un sistema operativo de tiempo real diseñado por la Universidad de Cantabria específicamente para sistemas con memoria limitada [1]. Escrito en Ada, un lenguaje de programación que ofrece robustez y seguridad, con alto soporte para aplicaciones concurrentes y modular, lo que permite dividir programas complejos en unidades manejables y portables [20]

M2OS se beneficia de las características notables de este y su capacidad de interactuar sin problemas con otros lenguajes, incluido C mediante interfaces [2]. Desarrollado en GNATStudio, un entorno de desarrollo integrado que usa el compilador de Ada, GNAT, y desarrollado por la misma entidad que desarrolla el lenguaje en sí, AdaCore, todas las ventajas del lenguaje pueden ser explotadas en este.

M2OS actualmente solo es compatible con tres tipos de placas, de las cuales solo una ofrece soporte multinúcleo [1]. Dadas las capacidades y ventajas de M2OS, es razonable y beneficioso buscar expandir su soporte a más hardware, especialmente a aquellos con capacidades multinúcleo. Tal expansión no solo ampliaría su aplicabilidad en varios sectores críticos, sino que también potenciaría su utilidad en sistemas que requieren respuestas de tiempo real bajo restricciones de recursos en las placas más usadas.

La Raspberry Pi Pico es una placa de desarrollo de costo accesible, tamaño compacto, portando el microcontrolador RP2040 en su interior con una serie de características que lo hacen destacar entre otros microcontroladores similares; pensado para aplicaciones de recursos escasos, el microprocesador cuenta con dos núcleos de procesamiento. lo que lo convierte en una opción ideal para aplicaciones que requieren un equilibrio entre rendimiento y restricciones de memoria, tanto para desarrolladores profesionales como para proyectos amateur.

El uso del entorno de desarrollo picoSDK es la forma en la que programas hechos en C pueden interactuar con la placa. Elegido por ser más eficiente que otras alternativas y junto con la inclusión de periféricos integrados, hacen del RP2040 una plataforma versátil y fácil de usar, haciendo que, junto con su bajo costo, la integración con el sistema operativo de tiempo real M2OS pueda dar lugar a soluciones potentes y económicas para aplicaciones de tiempo real.

1.2 Objetivos

El objetivo de este proyecto radica en poder ejecutar una serie de tareas básicas bajo entorno de M2OS en la placa Raspberry Pi Pico, además de la capacidad de explotar el entorno de doble núcleo para tareas con restricciones de tiempo real. Para ello hay que completar los siguientes puntos:

- Comprender las partes relevantes de la placa Raspberry Pi Pico y de su microcontrolador RP2040
- Comprender el entorno de desarrollo de aplicaciones de la placa, denominado picoSDK, las librerías que utiliza y como genera aplicaciones que se ejecutan en la placa mediante archivos objeto de dichas librerías
- Entender el sistema operativo M2OS, su inicialización y las partes relevantes para que se inicie correctamente.
- Entender el entorno de programación de Ada GNAT, como compila y construye los componentes y el entorno de depuración
- Portar e integrar las librerías proporcionadas por picoSDK en el entorno de M2OS.
- Desarrollar un prototipo funcional para la ejecución multinúcleo de tareas de M2OS con periféricos de E/S.

1.3 Palabras y términos clave

- I/O: son las siglas de Input/Outpt, traducido también al español como E/S, (Entrada/Salida), se refiere al acto de comunicación, bidireccional o no, entre un sistema de procesamiento de información y el “exterior”, pudiendo ser este el entorno u otro sistema de procesamiento de información, como un periférico.
- PIO (Programmable I/O, Entrada/Salida Programable): Técnica de comunicación de un dispositivo con periféricos en el que la CPU se encarga de todo el proceso de conexión o transferencia: en el caso de la placa usada en este proyecto, existen bloques PIO independientes al procesador para el manejo determinista de E/S.
- SPI, UART, I2C: diferentes protocolos de comunicación.
- PWM (Pulse Width Modulation, Modulación por Anchura de Pulso): técnica para la modulación (o generación) de señales analógicas mediante cambios en la altura de una onda
- GPIO (General Purpouse I/O, E/S de Propósito General): un tipo de entrada o pin en una placa que puede programarse para más de un tipo de función, usar más de un tipo de protocolo o tener diferentes tipos de características.
- SWD (Serial Wire Debug, Depuración por Cable Serie): protocolo de depuración por dos cables en serie para el acceso directo a la arquitectura del procesador ARM Cortex M0+.
- Wrapper: código que “envuelve” o “encapsula” la funcionalidad de cierto software para simplificarlo o adaptarlo a la funcionalidad que se requiere.

2. Herramientas y tecnologías usadas

En este apartado se detallan las tecnologías y herramientas de software y hardware utilizadas en el proyecto, desde hardware específico hasta entornos de desarrollo y lenguajes de programación.

2.1 Hardware

Para el desarrollo de este proyecto se han usado tres periféricos que se detallan a continuación:

2.1.1 Raspberry Pi Pico

La Raspberry Pi Pico es una placa de desarrollo creada por *The Raspberry Pi Foundation,* conocidos por crear la serie de ordenadores monoplaca Raspberry Pi.

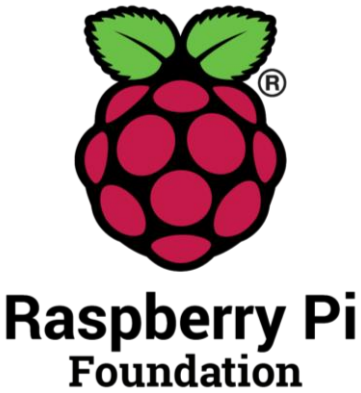


FIG. 1 - LOGOTIPO DE LA FUNDACIÓN RASPBERRY PI

Al contrario que su conocida gama de placas principal, la Raspberry Pi Pico es un microcontrolador, y no está diseñado para funcionar como un ordenador tradicional debido a que carece de componentes críticos para este propósito, como la salida de vídeo o almacenamiento interno significativo. En su lugar, este se centra en proporcionar un entorno eficiente para aplicaciones embebidas y de tiempo real, con el énfasis en la interacción más directa con el hardware [3]. Pese a su tamaño, tiene la misma cantidad de pines de entrada/salida que las Pi regulares, incluyendo GPIO, SPI, I2C, y UART [4], aunque de fábrica esta placa no tiene los pines conectores necesarios para integrar tarjetas de expansión o periféricos. Cuenta además con 264 KB de SRA y su placa hermana, la Pico W, cuenta con capacidades de comunicación inalámbricas como Wifi y Bluetooth.

Basada en el chip RP2040, diseñado por la ya mencionada fundación, este microcontrolador posee un procesador de dos núcleos ARM Cortex-M0+. Este núcleo de procesamiento es el más eficiente de la familia Cortex-M de ARM Holdings, diseñados para sistemas empotrados y optimizados para aplicaciones de bajo costo y consumo de energía, siendo este una versión más potente y de menor consumo energético del Cortex-M0, el más básico de esta familia.

La Raspberry Pi Pico ha sido la placa elegida para el proyecto, para la que ya se habían adaptado ciertas partes críticas del sistema operativo M2OS, aprovechando su flexibilidad.

La placa tiene un pin dedicado para el LED de la placa, 28 pines de propósito general que soportan varios protocolos de comunicación como I2C, UART o SPI, PIO, moduladores de tipo PWM, además de alimentación y E/S mediante un micro USB-B y tres pines extra para la interfaz SWD, optimizada para dar acceso y depurar, el microcontrolador [Fig. 2].

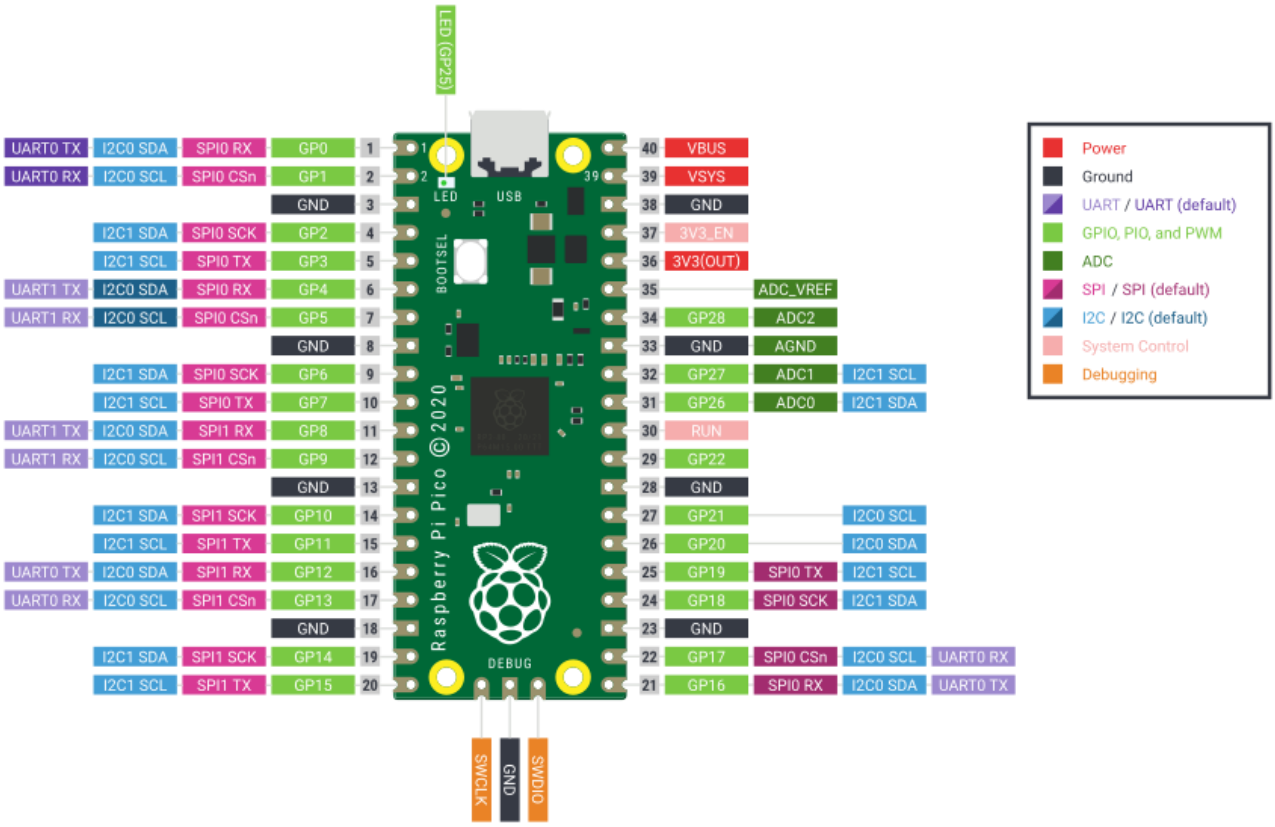


FIG. 2 – ESQUEMA DE LOS PINES DE LA RASPBERRY PI PICO

El microcontrolador RP2040

Es necesario entender la diferencia entre la placa Raspberry Pi Pico y el RP2040: el Microcontrolador RP2040, el primero de la marca Raspberry Pi, está pensado para brindar una alta eficiencia a un bajo costo con una gran facilidad de uso [5]. Tiene una alta flexibilidad respecto a las líneas de E/S que soporta, relojes de varias precisiones, tanto internos como para conexiones externas, y hardware de apoyo, como un interpolador y un divisor independientes, entre otros.

Este chip puede ser integrado en una infinidad de placas de E/S. En este caso, este es usado en la placa Raspberry Pi Pico, placa que añade otra serie de funcionalidades como el botón de selector de arranque “BOOTSEL” y soporte Flash mediante SPI, un LED de estado, una línea de depuración externa, y alimentación y entrada de datos mediante una toma micro USB-B o los 40 pines de E/S para conectar mediante diferentes tipos de conexiones a

diferentes periféricos, creando una interfaz fácil e intuitiva para que el microcontrolador los maneje. Lo que se entiende como “Raspberry Pi Pico” es el conjunto de elementos de Hardware propios de una implementación específica del microcontrolador RP2040.

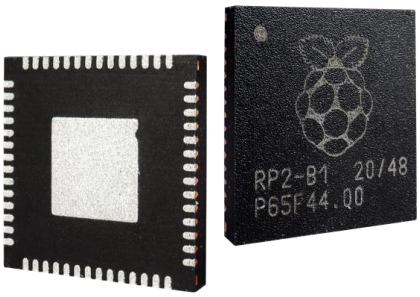


FIG. 3 - MICROCONTROLADOR RP2040

FIG. 4 - PLACA RASPBERRY PI PICO CON EL MICROCONTROLADOR INTEGRADO EN SU CENTRO, EN EL MODELO SIN PINES SOLDADOS.

Los pines periféricos que dan su capacidad de comunicación a la placa conectan directamente con el microcontrolador: 28 pines GPIO de propósito general, tanto para señales digitales como analógicas convertibles a digital en 3 de los pines. Además, el procesador tiene varias líneas independientes para la carga en memoria de programas mediante SPI. Además, tiene líneas dedicadas para el soporte USB y depuración en serie independiente [6].

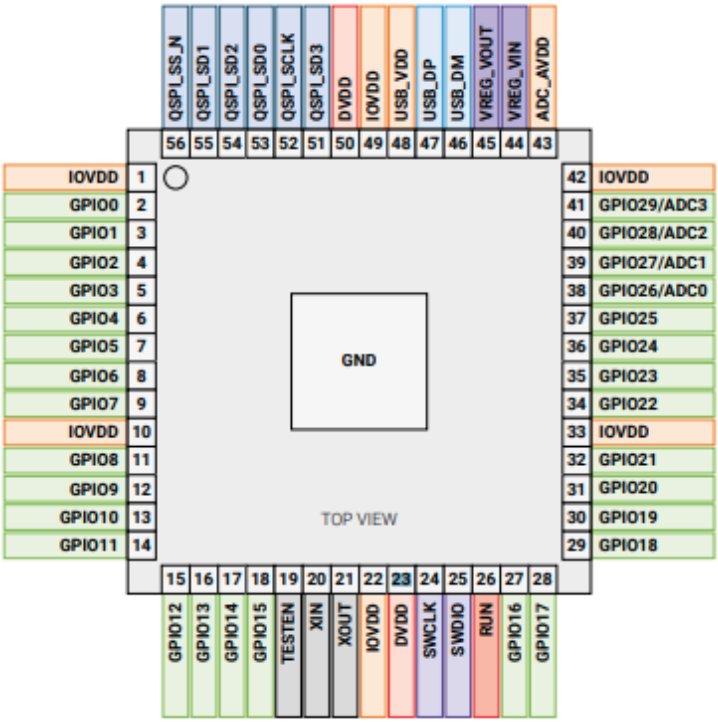


FIG. 5 - ESQUEMA DE LAS SEÑALES DE ENTRADA AL MICROCONTROLADOR

Estos, mayoritariamente, se corresponden con las entradas presentes en la placa, con la salvedad de que el controlador no distingue entre diferentes protocolos, que son tratados "fuera" del microcontrolador, en la placa de la que este forme parte.

Subsistema de procesamiento y Cortex-M0+

Mirando aún más de cerca la placa, el procesador Cortex M0+ opera a una frecuencia base de 48MHz y una máxima de hasta 133 MHz. Tiene dos núcleos de procesamiento y es la contraparte un tanto más potente del modelo de procesador más básico de la gama M de ARM, el cortex M0. Vistos como una alternativa con capacidad de procesamiento y gran eficiencia energética a los antiguos procesadores de 8 bits, esta gama de procesadores está pensadas para aplicaciones "altamente embebidas" y bare-metal, ya que carecen de algunos de los componentes necesarios para la operación de programas de alto nivel como un sistema operativo de propósito general como puede ser la unidad de manejo de memoria virtual. Además, incluye soporte SPI, I2C, y UART [4].

Mientras que para las tareas a realizar los detalles del microcontrolador en sí no son tan importantes, una cosa para tener en cuenta es el subsistema de procesamiento de la Raspberry Pi Pico: este abarca la completitud del hardware, los dos núcleos de procesamiento del Cortex M0+, además de los sistemas de comunicación extra: interfaz de bus del sistema, interfaz de interrupciones y depuración y el bloque SIO, que también incluye la comunicación inter-core o internúcleo [7].

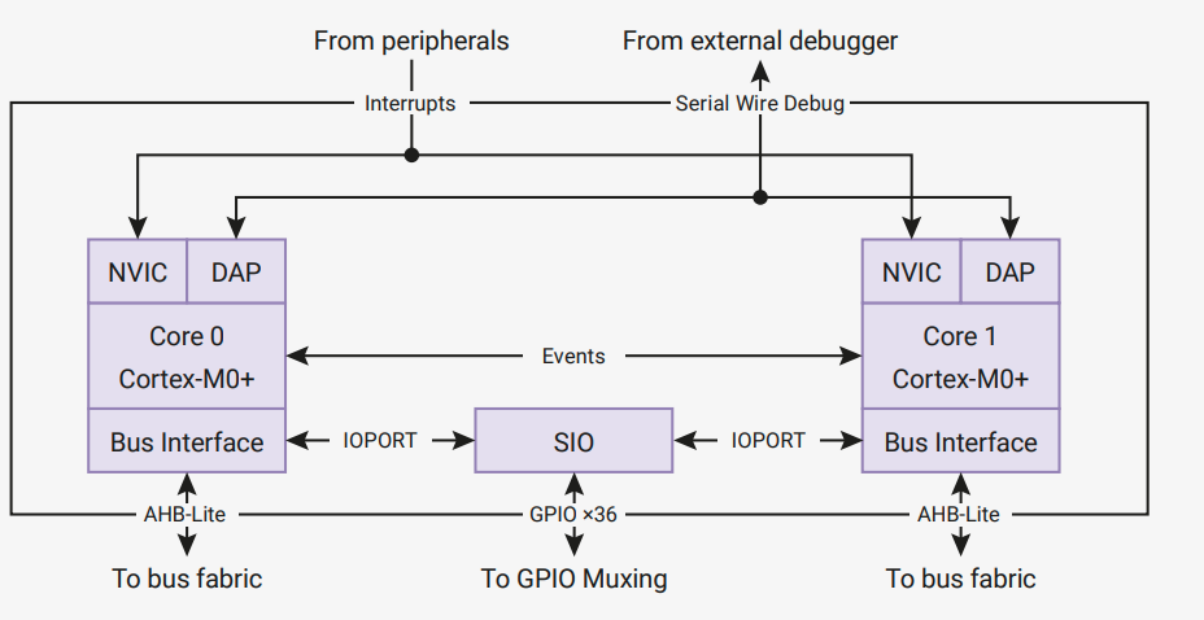


FIG. 6 - ESQUEMA DE CONEXIÓN DEL SUBSISTEMA DE PROCESAMIENTO CON EL EXTERIOR.

En primer lugar, la interfaz de bus de sistema conecta los procesadores con el resto del sistema mediante dos líneas de bus de protocolo AHB-lite que da acceso a la memoria y a cualquier periférico mapeado en esta [7].

La interfaz de interrupciones es un Controlador de Interrupciones Vectoriales Anidadas standard de ARM (ARM-NVIC). Existen 32 líneas de interrupción independientes para cada núcleo (*bank* 0 para el núcleo 0 y *bank* 1 para el núcleo 1), lo que significa que pueden existir interrupciones independientes para cada núcleo dependiendo del *bank* al que la línea llegue.

De estas 32 líneas, las primeras 26 están conectadas de fábrica al NVIC, y las restantes (de la 26 a la 31), a 0, lo que significa que nunca causarían una interrupción, aunque pueden dispararse manualmente activando el bit necesario en el Registro de Interrupciones Pendientes de Activación (ISPR) del NVIC [8].

Las líneas de interrupción tienen dos prioridades: una dinámica, en la que las líneas se colocan en 4 grupos, de grupo 0 al 3, en orden, de 0 a 31. Las líneas de un grupo más alto (las líneas de valor más bajo) tienen prioridad sobre el resto. La segunda prioridad es fija para las interrupciones en el mismo nivel dinámico, que siguen el mismo principio en el que la línea de valor más bajo del grupo es más prioritaria.

Además, el procesador puede entrar en un estado de baja energía esperando señales o eventos externos.

El bus de depuración utiliza la interfaz SWD para acceder directamente a los núcleos mediante las entradas dedicadas (SWDIO y SWDCLK) y dota, mediante un dispositivo de depuración externo, de acceso al estado y arquitectura del procesador, control del flujo de ejecución y carga de software [9].

El bloque de entrada-salida de ciclo único (Single cycle Input/Outpt o SIO) es el último de los tres sistemas de comunicación con señales exteriores del procesador. La diferencia principal con el bus y las interrupciones es que, como su propio nombre indica, las operaciones dentro

de este bloque se deben de hacer en un ciclo, lo que implica una limitación de tiempo que no se cumple con las señales del bus ni con el overhead generado con el manejo de interrupciones.

El bloque incluye un divisor y dos interpoladores para cada núcleo para hacer de manera eficiente dichas operaciones. Este bloque también cuenta con 32 spinlocks de hardware compartidos entre los dos núcleos, de los cuales la mayoría son accesibles desde las aplicaciones de usuario.

Incluye también dos colas unidireccionales (o mailboxes) para la comunicación entre núcleos, con un tamaño de 8 entradas de 32 bits por cola. Cada una solo puede ser leída por un núcleo y escrita por el otro y, en caso de estar activadas, tienen una interrupción especial de prioridad alta (denominada “evento”) que salta cada vez que una de las colas recibe datos [10]. Esta es la manera que tiene la placa de dotar a los núcleos de comunicación da nivel de hardware entre ellos.

Por último, tiene varios registros: el registro de identificación de núcleo (CPUID), es un registro que identifica a cada core, tanto entre ellos como para el exterior.

También, el banco de registros GPIO para rápido acceso y control de los pines de E/S designados como tal: todos estos componentes están conectados a dos buses dedicados por núcleo (diferentes del bus de sistema) para conseguir una rápida comunicación [Fig. 7].

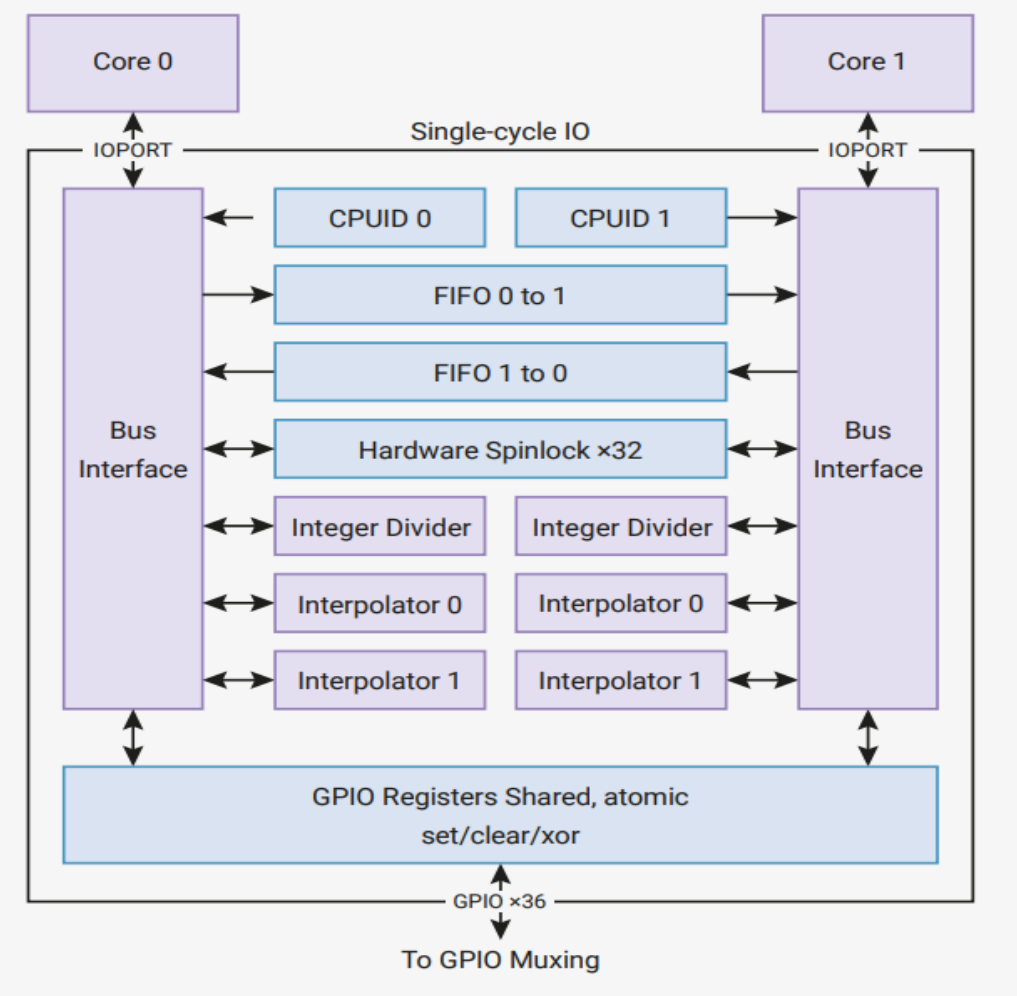


FIG. 7 - ESQUEMA DEL BLOQUE SIO DE LA RASPBERRY PI PICO

2.1.2 Placa de expansión de E/S

La placa Raspberry Pi Pico es usable en el estado en el que viene de fábrica. Como ya se ha mencionado, dependiendo de cual se compre, esta viene con o sin pines soldados. Además, dependiendo también de la aplicación en la que se vaya a usar la placa, es más fácil y cómodo desglosar los pines de la Raspberry, que debido a su compacto tamaño comparten varios usos [Fig2], o dotan a la placa de funcionalidades extra.

En este proyecto se utiliza la placa de expansión y desarrollo de la empresa DFRobot llamada "Pico Gravity Board". Esta es un buen punto de partida para cualquiera interesado en desarrollar con ella. En las dos líneas de puertos hembra más cercanas al centro de la placa, la Raspberry Pico va encajada, mientras que las dos, más exteriores, conectan directamente a la placa: los pines GPIO de la Raspberry Pi Pico.

Además, tiene pines extra, etiquetados con las interfaces o protocolos a los que atienden: entradas digitales y analógicas (que hacen uso del convertidor analógico-digital de la placa) y entradas SPI, UART e I2C completamente separadas entre ellas para mayor facilidad. La placa también cuenta con una interfaz de pantalla análoga a la que las Raspberry Pi normales tienen, y un puerto de voltaje de hasta 5.5V, interfaces con las que el microcontrolador no cuenta de fábrica.

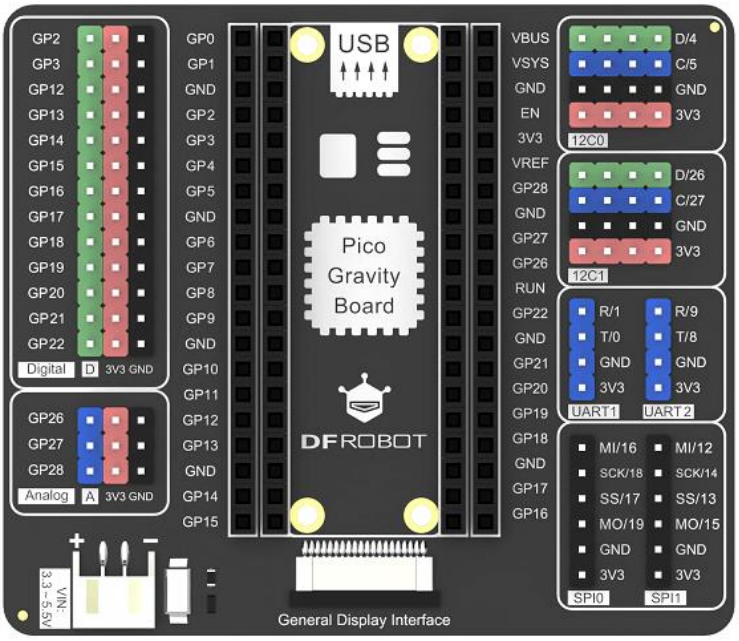


FIG. 8 – ESQUEMA DE PINES DE LA PICO GRAVITY BOARD

Por último, hay que tener en cuenta que, aunque se encuentren separadas, hay algunas entradas que pueden compartir pin. Por ejemplo, las entradas UART 0 y UART 1 se corresponden con los pines GPIO 0-1,6-7,11-12 y 19-22. A su vez, se puede acceder a algunos pines mediante las entradas genéricas (las líneas de conexión hembra) o en las entradas especializadas (como por ejemplo los pines digitales) [Fig. 2].

Esto significa que hay que tener cuidado de no usar dos pines de la Gravity Board cuyas señales entren por el mismo pin GPIO de la placa, o actuar en consecuencia cuando sea posible.

2.1.3 PicoProbe

Existen alternativas tanto de hardware como de Software para depurar la placa. Las de software se basan en emuladores que pueden cargar un subconjunto de funciones propias de la placa para ciertos sistemas, como el emulador rp2040js [18] o el depurador on-board de doble núcleo pico-debug [19]

El problema de estos es que pueden llegar a ser imprecisos dependiendo del sistema donde se ejecuten, en el caso de los emuladores o acaparan ciertos recursos de la placa, en el caso de depuradores on-board.

Es por esto por lo que es mejor opción usar depuradores externos. Raspberry Pi detalla dos métodos principales: Usar una Raspberry Pi 3 o 4 como servidor de depuración, usando los pines SWD y software específico cargado en esta o usar una pequeña placa basada en el chip RP2040 especializada para la depuración.

La *Raspberry Pi debugging probe,* traducido al español como sonda de depuración de Raspberry Pi, y también llamada *PicoProbe*, es una placa especializada en la depuración de dispositivos basados en ARM, especialmente placas Raspberry Pi, que soporta una variedad de *hosts*, incluidos Linux y Windows.

Ha sido elegida debido a que ofrece una interfaz más sencilla que otras soluciones y es fácil de programar. Portando también el microcontrolador RP2040, tiene un puente de UART (lado del hardware a depurar) a USB (lado de conexión al host) y es compatible con el estándar CMSIS-DAP compatible con programas de depuración mediante conexión serial como OpenOCD y depuración mediante GDB.

FIG. 9 – SONDA DE DEPURACIÓN DE RASPBERRY PI

2.2 Software

El Software utilizado en este proyecto abarca varias áreas: los lenguajes de programación usados, los entornos de desarrollo necesarios y el entorno en el que se ha hecho el trabajo, incluyendo Sistema Operativo, scripts, programas de depuración y compiladores:

2.2.1 Lenguajes

Ada

El lenguaje Ada (llamado así en honor a Ada Lovelace), es un lenguaje orientado a objetos, concurrente y de tipado fuerte creado en 1980 e inspirado en lenguajes como C++, Java y Pascal.

Como lenguaje de programación, además de ofrecer las mismas características que la mayoría de los lenguajes ampliamente usados, embebidas y de tiempo real [21]. Con documentación para aprender el lenguaje y soportado en una amplia gama de procesadores [12].

FIG. 10 - LOGOTIPO DEL LENGUAJE ADA CON EL LEMA "IN STRONG TYPING WE TRUST"

En este lenguaje de programación está escrito el Sistema Operativo que se requiere portar, debido en parte a las prestaciones tanto de rendimiento como de seguridad: chequeos en tiempo de ejecución, mecanismos de modularización de código y paralelismo e interfaces hacia otros lenguajes de programación [2].

C

C es un lenguaje de programación imperativo estructurado de propósito general creado en 1972. Es uno de los lenguajes más antiguos y por lo tanto más estables y populares.

FIG. 11 - LOGOTIPO DEL LENGUAJE C.

Este principalmente es usado por el kit de desarrollo de la Raspberry Pi Pico (PicoSDK). Es necesario entender cómo se desarrollan los wrappers que actuarán de comunicación entre M2OS y picoSDK, además de cómo es su interfaz con Ada.

2.2.2 Kits de desarrollo

Para el proyecto se han usado dos kits de desarrollo (SDK) distintos. GNAT Studio, el IDE orientado a Ada, y PicoSDK, orientado a las librerías de C de la Raspberry Pi Pico.

GNAT Programming Studio

El Estudio de programación GNAT, (GNAT Programming Studio, GPS, escrito a veces como GNATStudio) es un entorno de desarrollo integrado (IDE) multilenguaje desarrollado por Adacore, entidad principal encargada de mantener Ada y que crea soluciones comerciales para este programa.

Este entorno de desarrollo integrado es multiplataforma y multilenguaje, soportando, además de Ada, C, C++, Python y SPARK [13] y con una interfaz fácil de comprender. La razón principal de su uso es su fácil configuración tanto para ejecución y compilación como para depuración de programas escritos en Ada. El sistema operativo por desarrollar tiene integración completa con este software, haciendo la elección bastante evidente. La versión comunitaria es suficiente para desarrollar un proyecto con estas características.

FIG. 12 - LOGOTIPO DEL IDE GNATSTUDIO.

Gracias a la modularización de Ada y a GNATSTudio, se pueden usar varios archivos .gpr (GNATStudio PRoject) para añadir diferentes partes necesarias para M2OS. Se pueden distinguir tres proyectos para el correcto funcionamiento del sistema operativo:

- M2OS_RP2040 contiene todos los archivos de GNAT y Ada relativos al sistema operativo en sí: El HAL, el kernel, los drivers y los wrappers.
- Switches: Son parámetros configurados que se pasarán tanto al compilador como al linker a la hora de generar ejecutables del un proyecto de GNATStudio. Se distinguen dos:
    - Global switches: Se aplican a todos los proyectos añadidos
    - Shared switches: Se aplican a un subconjunto de proyectos. En este caso, son los switches para la compilación en el microcontrolador RP2040, y distingue, además, entre configuraciones para Ada y para C.

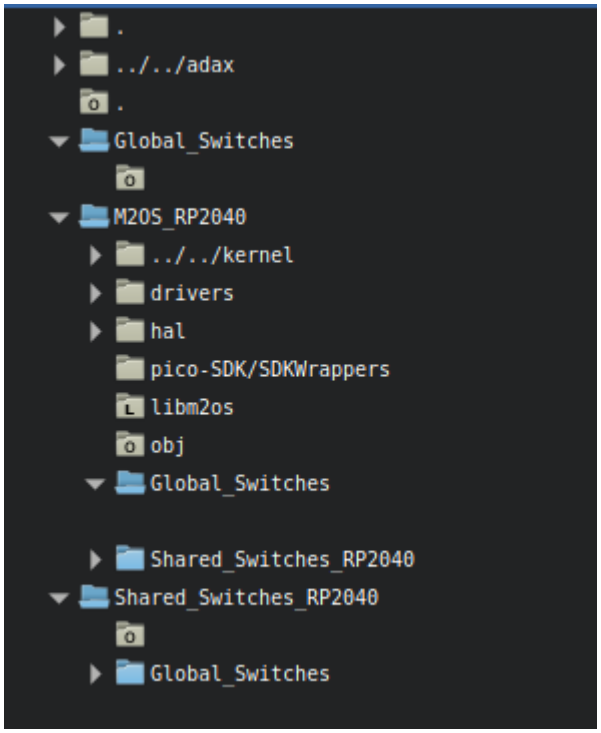


FIG. 13 – VISTA BÁSICA DE LOS TRES PROYECTOS PRINCIPALES DE CUALQUIER PROGRAMA HECHO PARA M2OS EN EL MICROCONTROLADOR RP2040, DESDE GNATSTUDIO

PicoSDK

PicoSDK es el conjunto de librerías, cabeceras y sistema de generación de código necesarios para desarrollar programas para dispositivos basados en el RP2040 en C, C++ y ensamblador. Presenta varias interfaces (APIs) con un gran soporte para todos los sistemas compatibles con el microcontrolador como USB, DMA, IRQs y PIOs entre otros [14].

La elección de este entorno de desarrollo de software (SDK) se debe a la cantidad de documentación y ayuda comunitaria que existe y a la facilidad para acceder a las prestaciones del hardware del RP2040. Al trabajar sobre lenguajes compilados (C, C++) es mucho más rápido y eficiente que la alternativa (MicroPython/CircuitPython) que son lenguajes interpretados. Esto es importante para el objetivo de M2OS, un sistema operativo de tiempo real.

Además, otorga al desarrollador control completo sobre las partes a usar, incluidas aquellas que se han creado pensando en conseguir más sencillez sacrificando algo de eficiencia, como es el caso del manejo de errores, pudiendo editarse o completarse de ser necesario.

El sistema de generación de código (build) está basado en *CMake*, que es altamente configurable para muchas plataformas y sistemas de build (como *make*) y es fácil e intuitivo de usar en su forma más básica.

2.2.3 M2OS

M2OS es un sistema operativo de tiempo real diseñado por la Universidad de Cantabria para sistemas con memoria limitada [1],

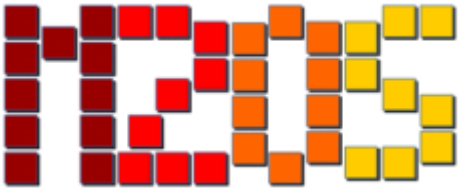


FIG. 14 - LOGOTIPO DE M2OS

Este sistema operativo tiene un sistema de planificación no expulsor de tareas de una sola ejecución (one-shot) para obtener una huella de memoria pequeña. Pensado para soportar multitarea y múltiples procesadores, utilizando también las características notables de Ada, el lenguaje en el que está escrito, como su robustez y seguridad, y su capacidad de interfaz con otros lenguajes [2] [20].

Este es el lenguaje sobre el que se realizará el trabajo de fin de grado. Su código, que es abierto, se encuentra en el repositorio de Gitlab con el mismo nombre. La rama sobre la que se desarrolla el contenido de este trabajo se denomina RP2040 [17].

M2OS se compone de varios submódulos: Principalmente, el kernel, unos drivers esenciales, como el controlador USB y la E/S directa, y los wrappers.

Aquí, hay que destacar el papel de dos interfaces: La API del kernel, que son métodos que exponen ciertas funcionalidades del kernel a las aplicaciones de usuario de M2OS, y el HAL. Este último, es una capa de abstracción que proporciona acceso a funcionalidad de ciertas partes del hardware a través de funciones.

La modificación de estas dos interfaces es necesaria para la adaptación de M2OS al microcontrolador. Ciertas funciones del HAL tienen que usar métodos disponibles a través de picoSDK para poder acceder a recursos de la placa.

La API de usuario también deberá ser modificada acorde si el usuario necesitara acceder explícitamente a funcionalidades añadidas durante el desarrollo del proyecto.

2.2.4 Compiladores

Los dos compiladores usados son GCC (GNU C Compiler) para C y GNAT (GNU NYU Ada Translator) para Ada.

Se necesita un compilador cruzado, dado que la plataforma de origen y la de destino no son iguales: en este caso, el compilador cruzado de GNAT de linux64 a ARM con el formato ELF.

Este es el compilador utilizado para poder crear un ejecutable para el RP2040 (microcontrolador de arquitectura ARM) desde Linux, en este caso de 64 bits.

Además, se necesita soporte de compilación nativo, tanto de Ada (GNAT) como de C (GCC), que forman parte de la colección de compiladores de GNU (GCC).

FIG. 15 – LOGOTIPO DE LA COLECCIÓN DE COMPILADORES GNU.

2.2.5 Otras herramientas

El proyecto se desarrolló en una máquina ejecutando Linux basado en Debian de 64 bits. Otras herramientas, mientras que no son estrictamente necesarias para el trabajo, han sido utilizadas para la automatización, gestión y depuración del software:

Automatización y herramientas de generación

Se han utilizado tres herramientas para la automatización del proyecto: la primera es la línea de comandos: el intérprete de comandos bash. Se han automatizado las tareas de creación de la librería partiendo de los ficheros objeto, además de un script para lanzar un ejecutable a la PicoProbe mediante OpenOCD.

El script que guía al usuario por todos los pasos para crear su propia versión de la librería también está escrito como un script de terminal bash.

En Python se ha creado el script que rellena el CMake del proyecto con los headers seleccionados de la librería PicoSDK para generar el archivo estático necesario en M2OS.

Por último, como parte de los sistemas de compilación de M2OS y PicoSDK, se han usado dos herramientas para generar parte del código utilizado:

- *CMake*, abreviación de "*Cross-platform Make"* es un sistema de automatización que se usa para crear scripts de generación de código ejecutable (build) mediante archivos *CMakeLists.txt*. PicoSDK lo utiliza para crear los archivos necesarios para compilar el ejemplo o código que se quiere lanzar en la placa usando la librería.
  M2OS lo utiliza como parte de su sistema de generación para preparar para su compilación todas las áreas del sistema operativo. Esta herramienta es potente y puede usarse para generar scripts de compilación en una variedad de entornos, entre ellos make.
- *Make* es el programa encargado de generar código ejecutable a partir del código fuente que se lista en archivos *makefile* y puede ser creado manualmente o a partir de herramientas como CMake.

Es decir, *CMake* modulariza la creación de scripts de generación de código ejecutable, tanto de *Make* como de otros programas de función similar, mientras que el propio *Make* es el programa que se encarga de convertir, de manera automática, el código especificado en código ejecutable con todas sus dependencias usando las herramientas necesarias, como compiladores.

FIG. 16 – LOGOTIPO DEL PROGRAMA CMAKE.

Gestión del proyecto y control de versiones

Para el control de versiones y seguimiento, se han usado dos implementaciones del software *Git*: *GitLab* y *GitHub*. *Git* es un sistema de desarrollo de software y control de versiones centralizado basado en entregas (*commits*). Fue creado en 2008 para el Kernel de Linux y se convirtió rápido en una herramienta usada por ser de las primeras de su tipo en ser desarrolladora.

GitHub es una plataforma de hosting de repositorios basados en Git, centrado en el desarrollo open-source colaborativo. En este proyecto se usa para toda la parte de desarrollo de la librería y compatibilidad con el PicoSDK, ***RPIPICO-M2OSPORT-TFG*** [16]. Parte de los archivos contenidos son la base de algunas de las tareas de automatización del proyecto final, y algunas pruebas, aunque este sea independiente del producto final del proyecto.

Por otro lado, *GitLab* es una plataforma, también basada en *Git*, pero más orientada al *DevOps*: test, desarrollo y despliegue de software, en la nube y localmente. *GitLab* se usa para alojar el código fuente de M2OS. Todos los cambios, adiciones y archivos que se explicarán a continuación se pueden ver en la rama RP2040 del repositorio de *GitLab* M2OS.

FIG. 17 – LOGOTIPOS DE GIT, GITHUB Y GITLAB RESPECTIVAMENTE

Herramientas de depuración

Por último, se ha efectuado la depuración del proyecto durante su desarrollo, especialmente en la fase de implementación de la librería PicoSDK y los diferentes componentes en la placa Raspberry Pi Pico:

Para la depuración de bajo nivel, se ha utilizado *Open On-Chip Debugger* (*OpenOCD*) configurado específicamente para trabajar con la Raspberry Pi Pico mediante la PicoProbe. *OpenOCD* establece una interfaz de comunicación entre host y el microcontrolador RP2040, permitiendo así operaciones de programación y depuración en tiempo real para sistemas embebidos.

Integrado con *OpenOCD*, el *GNU Debugger* (*GDB*) proporciona las capacidades de depuración necesarias para controlar la ejecución del firmware, inspeccionar el estado del sistema, y alterar el flujo de ejecución. *GDB* es esencial para identificar y solucionar problemas específicos durante el desarrollo de los componentes del sistema operativo M2OS, permitiendo la ejecución paso a paso y el análisis detallado del comportamiento del código en la Raspberry Pi Pico.

Por último, *Minicom* se ha utilizado como interfaz de terminal para facilitar la comunicación serie con la Raspberry Pi Pico. Esto permite observar los registros de salida y los mensajes de error o información generados por M2OS o el programa que ejecuta, ofreciendo una vista directa y en tiempo real del estado de la ejecución en el sistema. El uso de *Minicom* verifica el correcto funcionamiento del sistema y que las salidas son las esperadas.

3. Preparación del entorno de desarrollo de M2OS

La preparación del entorno de desarrollo del sistema M2OS requiere generar una librería de archivos objeto para el control de la placa, así como los compiladores y el depurador. Los pasos se detallan a continuación:

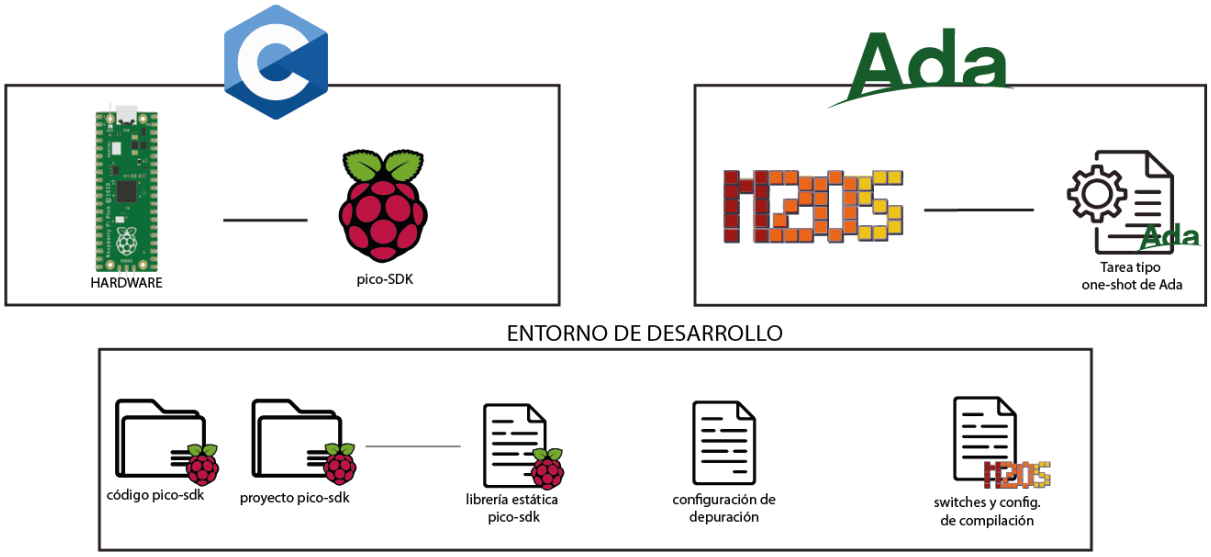


FIG. 18 – DIFERENTES PARTES DEL PROYECTO QUE HAY QUE ESTUDIAR O MODIFICAR. DEBAJO, LAS DIFERENTES PARTES DEL ENTORNO DE DESARROLLO

3.1 PicoSDK: entorno, compilación e integración para M2OS

PicoSDK (Kit de desarrollo de software para Raspberry Pi Pico) es usado para generar una librería estática. Esta librería se utilizará durante la ejecución de M2OS, para tener acceso a las funciones del kit de desarrollo. Estas, a su vez, son las encargadas de interactuar con las diferentes partes del sistema. De este modo, se pueden hacer llamadas a estas funciones desde M2OS, que, al contrario que el PicoSDK, está escrito en Ada.

Sin esta, como ya se ha explicado, habría que interactuar con el hardware de otra manera, ya fuera de manera directa mediante código ensamblador o usando otras alternativas menos eficientes como MircroPython. Faltando esta librería en el entorno actual, se harían llamadas a funciones no disponibles en el entorno de ejecución de M2OS, por lo que el código no podría compilar.

Para llegar a generar esta librería, es necesario entender cómo se crea un proyecto de PicoSDK, como se genera el ejecutable, y como este ejecutable tiene disponibles las funciones que se encargarán de interactuar con los componentes del dispositivo:

Creando un proyecto de PicoSDK – ejemplo mínimo compilable

Tanto el código fuente como las instrucciones para desplegar un ejemplo sencillo se encuentran en el repositorio de Github de PicoSDK [15]. Se necesitan, también, unas dependencias, incluido CMake y el compilador cruzado ya mencionadas.

En este caso, se proceden a crear dos carpetas alojadas en la misma ubicación. Una es producto de clonar el repositorio de Pico-SDK y tendrá este nombre. La otra tendrá un nombre

arbitrario e incluirá los archivos del provecto del SDK de prueba en el que se desarrollará esta parte del proyecto [16]. La ubicación es arbitraria, pero tenerlas en el mismo directorio facilitará el referenciarse la una a la otra y será más fácil entender el proceso de compilación y el código generado.

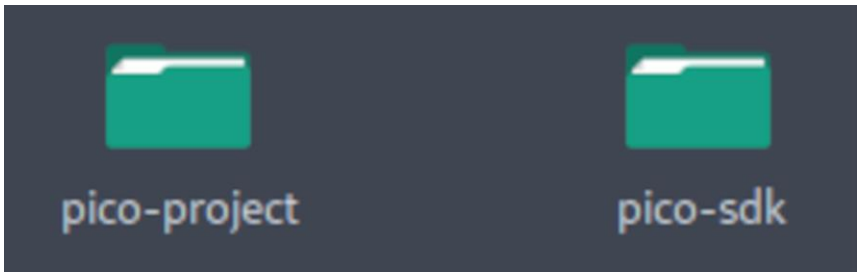


FIG. 18 – CARPETAS DEL SDK DE LA RASPBERRY PI PICO Y EL PROYECTO DE PRUEBA "PICO-PROJECT" ALOJADAS EN EL MISMO DIRECTORIO.

Una vez creada la carpeta del proyecto de prueba y estando en esta, creamos el *CMakeLists.txt* copiando su contenido de las instrucciones del repositorio de pico-sdk.

El archivo C sería, en un proyecto de picoSDK normal, el código que se ejecutaría en la placa y se serviría de las librerías de picoSDK, importadas mediante directivas `#include`, para acceder al hardware.

En nuestro caso, este archivo puede solamente contener un punto de entrada vacío, ya que solo se usará de plantilla para generar el resto de los archivos y, como se explicará a continuación, este archivo no afecta a qué partes de las librerías se compilan o no.

Generar la carpeta build es el último paso. Dentro de esta, se creará, además del archivo final, todos los archivos intermedios generados o usados durante el proceso de construcción del ejecutable.

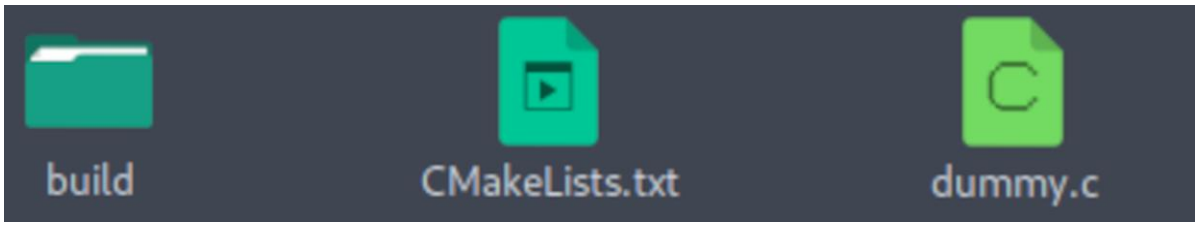


FIG. 19 – ELEMENTOS MÍNIMOS DENTRO DE LA CARPETA DEL PROYECTO DE PRUEBA

Utilizas las herramientas `cmake` y `make` como se relata en el ejemplo, es el último paso necesario para comenzar el proceso de generación del ejemplo en cuestión.

Entendiendo el comportamiento del generador de código.

La compilación crea un ejecutable de extensión *elf* que después es convertido por el propio proceso de compilación de pico-SDK en un archivo de extensión *uf2*. Este es un archivo apto para ser cargado en memoria de la Raspberry Pi Pico mediante carga o *flash* USB.

Pero nuestro objetivo no es ejecutar el código generado en la placa, si no valernos del código de picoSDK que su compilación genera para usarlos en M2OS como librería.

Mediante la salida del comando de generación y estudiando un poco la estructura del ejemplo, se puede saber que de la carpeta Pico-SDK se extraen los códigos y cabeceras de las librerías usadas y sus funciones según la ruta suministrada a la variable de sistema *PICO_SDK_PATH,* la cual debe apuntar al repositorio que hemos clonado. Estos archivos se encuentran residiendo ya dentro de la propia carpeta del proyecto, más concretamente dentro de la carpeta *build.*

Generar una librería estática

En un proyecto de pico-SDK, las librerías compiladas son independientes del archivo C que las usa y las directivas que incluya. Estás serán configuradas en el archivo CMakeLists.txt donde se listarán las librerías utilizadas para cada ejecutable a compilar, así como otros ajustes, incluido los lenguajes y versiones a compilar, entre otros.

El ejemplo mínimo que se puede copiar de las instrucciones tiene este aspecto:

```
cmake_minimum_required(VERSION 3.13)


# initialize the SDK based on PICO_SDK_PATH
# note: this must happen before project()
include(pico_sdk_import.cmake)

project(my_project)

# initialize the Raspberry Pi Pico SDK
pico_sdk_init()

add_executable(hello_world
  hello_world.c
)

# Add pico_stdlib library which aggregates commonly used features
target_link_libraries(hello_world pico_stdlib)

# create map/bin/hex/uf2 file in addition to ELF.
pico_add_extra_outputs(hello_world)
```

La variable de entorno PICO_SDK_PATH debe de estar rellena con la ubicación del pico-sdk clonado para referenciar el cmake de entrada (`pico_sdk_import.cmake`), así como al método de inicialización `pico_sdk_init()`. `pico_add_extra_outputs`, añadirá formatos extra a la salida en formato *elf*, como el formato *uf2* que se usa para cargar el ejecutable en la placa.

Los métodos interesantes respecto a la compilación son dos: add_executable y *target_link_libraries.* La primera clausula otorga un identificador a uno o varios objetos de los que se creará el ejecutable final.

La segunda especifica que librerías accesibles se usarán para compilar dichos ejecutables. En este caso, se especifican los sistemas de la placa cuyos archivos fuente se cargarán con el ejecutable.

De estos archivos fuente, se crean archivos objeto de extensión "*.o"*. Estos archivos contienen el código interpretable por la plataforma que equivale al código legible del que proviene. Una vez se han generado todos estos archivos, se ensambla el ejecutable que se puede ejecutar en la placa con todas las definiciones y referencias

Al ejecutar el comando *cmake ..*, se crean ciertas carpetas con otros archivos CMake necesarios para generar el código. Acto seguido, el *make* genera todos los objetos y archivos dictados en los archivos CMake dentro de la carpeta build.

Por convención del entorno de desarrollo, cada archivo del que generar un ejecutable tiene su propia carpeta donde se alojan todos los archivos objetos necesarios para su compilación. Esta carpeta se encuentra dentro de una carpeta denominada *CMakeFiles*, que a su vez se encuentra dentro de la carpeta build. La carpeta para cierto ejecutable tiene como nombre *nombre_add-executable.dir,* que es el mismo que se le dio en la cláusula *add_executable* del *CMakeLists*.

A modo de ejemplo, imaginemos que tenemos un archivo *dummy.c* del que queremos crear un ejecutable. Este puede tener código funcional o no. En el *CMakeLists.txt*, existirá una cláusula parecida a:

```
add_build_executables(dummyfile
      ./dummy.c
)
```

Donde *dummyfile* es el identificador dado al CMake (y no tiene necesariamente que coincidir con el nombre del archivo de código), y *./dummy.c* es la ruta al archivo que se quiere compilar.

También existirá la cláusula *target_link_libraries*, que será parecida algo como:

```
target_link_libraries(dummyfile
    pico_stdlib
    hardware_timer
    pico_multicore
    hardware_gpio
)
```

Donde *dummyfile* es el identificador del archivo al que añadir los objetos de la librería pico-SDK. En este caso, se añaden las funciones estándar (*pico_stdlib)*, las funciones de control de timer (hardware_timer), las funciones para controlar el segundo núcleo (*pico_multicore)*, y las funciones para controlar periféricos mediante el GPIO estándar (hardware_gpio). Se pueden quitar o añadir muchas otras que controlan diferentes aspectos o sistemas de la placa.

Sabiendo esto, los objetos relacionados con estas sublibrerías, al generar el ejecutable, se encontrarán en

/ruta/a/proyecto/.../build/CMakeFiles/dummyfile.dir/

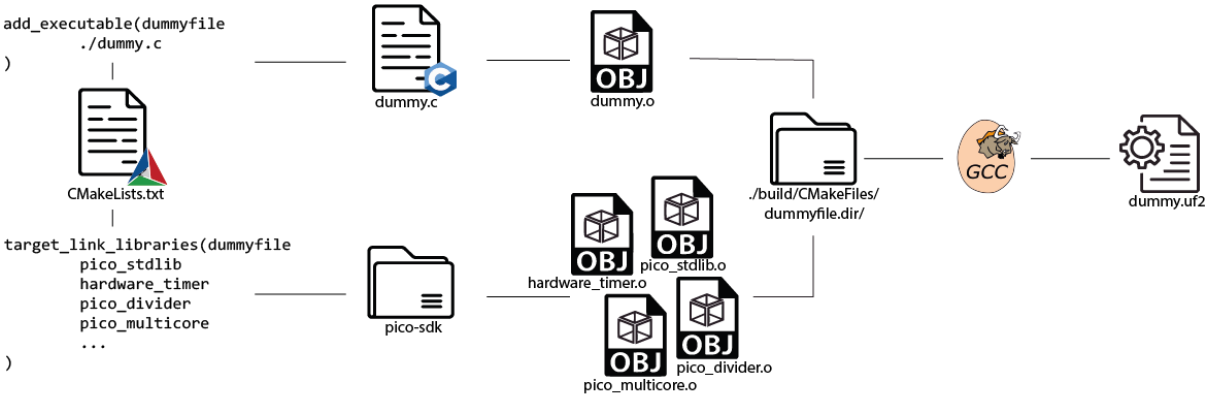


FIG. 20 – GRÁFICO SIMPLIFICADO DE GENERACIÓN DE UN EJECUTABLE DE PICO-SDK.

Ahora, solo es cuestión de generar una librería estática con el comando del entorno de compilación correspondiente, en nuestro caso, ***arm-none-eabi-ar,*** pasando una lista con los objetos, ya sea de manera manual o mediante otros comandos como *find*.

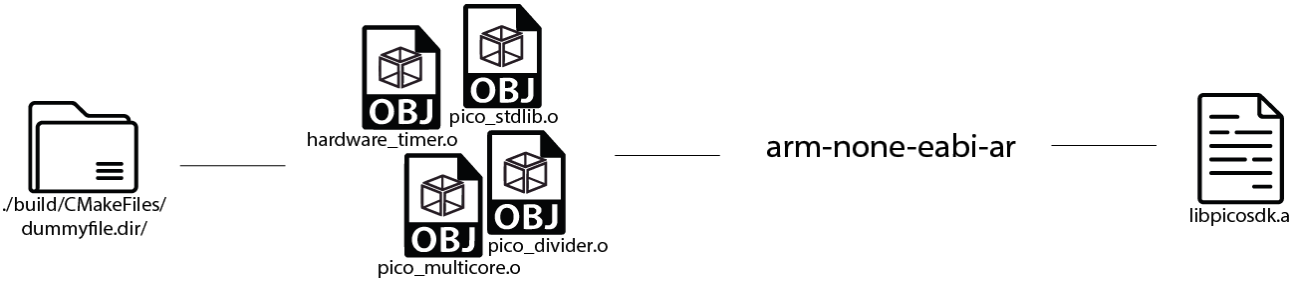


FIG. 21 – LA LIBRERÍA SE HACE A PARTIR DE LOS OBJETOS EN LA CARPETA BUILD DEL PROYECTO GENERADO

Otras carpetas dentro de *build*, generan otros archivos de apoyo, como por ejemplo el programa *elf2uf2* que crea un archivo en formato *uf2* a partir del archivo ejecutable *elf* compilado que no son necsarias para nuestro propóstio.

Símbolos de punto de entrada: conflicto con M2OS.

Solamente hay que tener una última cosa en cuenta: los símbolos de entrada. Cada programa, dando igual el lenguaje en el que esté escrito, tiene un “entry point” o punto de

entrada. Es decir, por dónde el programa se empezará a ejecutar cuando se cargue en memoria. En el código objeto, durante la compilación y la vinculación, ese punto se representa, como muchas otras cosas del código, mediante símbolos.

Por defecto, a no ser que se indique lo contrario, el símbolo de entrada se denomina `main` y coincide con la función de entrada del mismo nombre.

De este símbolo en concreto, solo puede haber uno por ejecutable. En caso de querer hacer un proyecto del picoSDK, no sería necesario hacer nada más. Sin embargo, dado que esta librería será usada en M2OS, que ya tiene su propio punto de entrada *main*, generaría un conflicto a la hora de compilar y ejecutar. Es por esto, que se necesita cambiar el nombre del símbolo de entrada del objeto del código del que se genera el ejecutable.

Además, hay un conflicto con dos librerías del divisor de hardware: ***pico_divider.h*** y ***hardware_divider.h***. Ya que ambos usan código específico común, solo se puede usar uno. Utilizar ambos resultaría en un fallo de compilación por tener, de nuevo, referencias y símbolos duplicados.

Pico_divider es un superset que incluye ***hardware_divider*** y otro conjunto de rutinas, por lo que para arreglar este problema solo hay que obviar (o en este caso, borrar) el objecto hardware_divider.h para que la librería solo incluya ***pico_divider.***

Aplicación en el proyecto.

La rama rp2040 de M2OS, viene con dos librerías listas “comprimidas”, que se explicarán más adelante. Dependiendo de una opción alojada en el archivo *config_params.mk*, llamada ***CONFIG_RP2040_SERIAL,*** se elegirá una de estas al generar los archivos del sistema operativo, generando un archivo ***picosdk.a*** de manera rápida que contiene todas las librerías estándar de pico-sdk para controlar todo el hardware de la placa. La diferencia radica en que en una librería está activada la salida por serie y en la otra por USB, dado que es incompatible tenerlas ambas activas.

Además, se puede regenerar la librería con cierto grado de personalización ejecutando un script llamado ***picosdk-init.sh.*** Este script llevará al usuario por todos los pasos de generación de la librería y la creará automáticamente. Los pasos incluyen:

1. Descargar una copia del PicoSDK de github, si no hubiera ya una presente
2. Cargar los submódulos de PicoSDK, necesarios para ciertas librerías del entorno
3. Generar una carpeta, picosdk-library, de la que se cogerán los objetos, como se ha explicado más arriba, con los importes y el archivo C vacío.
4. Preguntar al usuario, cuales de las librerías disponibles en el SDK se quieren usar en esta instancia de M2OS.
5. Generar un CMakeLists.txt de acuerdo con lo introducido en el paso anterior.
6. Preguntar al usuario si se creará la librería con símbolos para la depuración
7. Crear la carpeta build, llamar a los comandos cmake .. y make
8. Copiar los objetos, crear la librería y cambiar el nombre del símbolo del punto de entrada
9. Copiar la librería generada a la ubicación final en .../arch/rp2040/pico-SDK
10. Limpiar las carpetas si el usuario así lo decidiera.

Un pseudocódigo aproximado sería:

ENTRAR A DIRECTORIO ./picosdk-init

CLONAR pico-sdk

INICIALIZAR submódulos de pico-sdk

ESCRIBIR VARIABLE PICO_SDK_PATH

CLONAR pico-extras

CREAR directorio picosdk-library

ENTRAR A DIRECTORIO ./picosdk-library

CREAR picosdk-library.c vacío

COPIAR archivos de soporte necesarios, si se necesitaran

EJECUTAR cmakelist_lib_generator.py para generar el CMakeLists.txt

CREAR directorio build

ENTRAR A DIRECTORIO build

PREGUNTAR AL USUARIO si la librería debería tener símbolos de depuración o no

EJECUTAR create_libpicosdk.sh para generarla con o sin símbolos de depuración

COMPROBAR QUE libpicosdk.a se creó correctamente

PREGUNTAR AL USUARIO si quiere limpiar los directorios creados durante el script

BORRAR directorios dependiendo de la confirmación.

La librería ***picosdk.a*** creada será referenciada desde el entorno de M2OS para poder usar las funciones del PicoSDK desde Ada.

3.2. GNATSTUDIO: compilación cruzada y depuración

El entorno de desarrollo GNATStudio se utiliza para el resto del desarrollo de este proyecto, dado que está escrito en lenguaje Ada. Para este proyecto, se necesitan los compiladores cruzados ya mencionados, para poder compilar código para la Raspberry desde un ordenador. A diferencia de los compiladores cruzados necesarios para picoSDK, que son de C, estos son para el lenguaje Ada. También es importante activar la depuración durante el desarrollo del sistema operativo, y también dar la opción al usuario final de activarla para poder arreglar cualquier problema en su propia aplicación corriendo en M2OS

Compilación cruzada

Hay que descargar una versión específica del compilador cruzado que no es de las más recientes, pero se evita así tener problemas de compatibilidad tanto con la versión de GNATStudio elegida como con el código de M2OS. En este caso, es la versión 12.2.0-1 del compilador cruzado proveniente de la colección de compiladores de GNU, que es además compatible con la versión usada de GNATStudio (GNATStudio 2021).

Proceso de compilación de M2OS

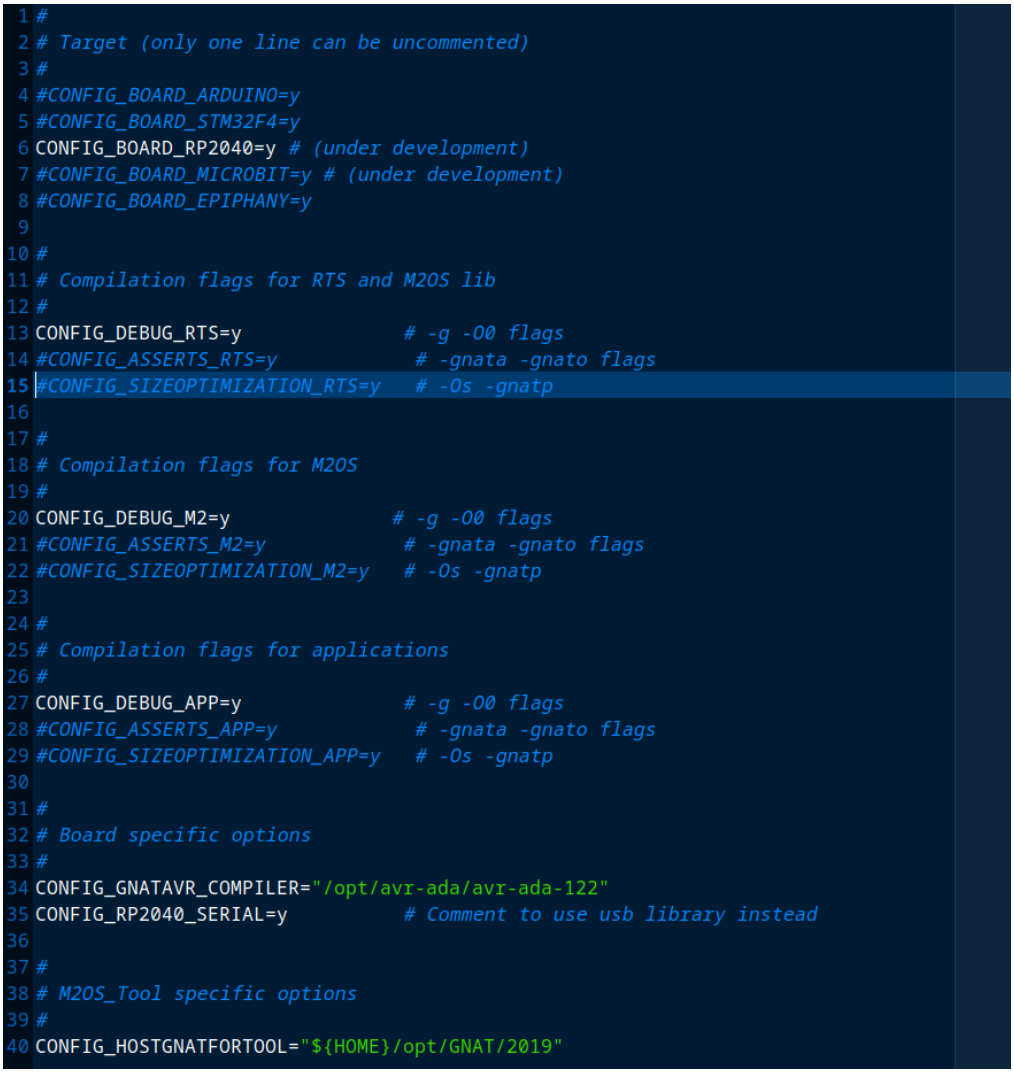
El make en la raíz del proyecto se utiliza para generar todos los archivos de M2OS. El archivo *makefile* y el archivo ***rules.mk*** proporcionan el código de generación. Además, existen dos archivos más, ***config_params.mk*** y ***config.mk.***

config_params.mk es un archivo editable con el que se modifican ciertas variables globales de compilación. Especialmente importante es la variable ***CONFIG_BOARD*** que dicta el sistema objetivo de la compilación.

Los siguientes tres grupos de opciones dictan las flags para generar el entorno de ejecución de GNAT (gnatRTS), M2OS y las aplicaciones para M2OS.

El siguiente grupo tiene opciones para placas específicas: la ruta para el compilador de chips AVR (cuando aplique, no en este caso) y la flag para elegir la librería por defecto para el microchip RP2040.

Las opciones descomentadas son las usadas. De cada grupo, excepto de las configuraciones específicas, los dos últimos grupos, solo puede haber una. De haber varias podría haber comportamientos inconsistente o erróneo.



```
#
# Target (only one line can be uncommented)
#
#CONFIG_BOARD_ARDUINO=y
#CONFIG_BOARD_STM32F4=y
CONFIG_BOARD_RP2040=y # (under development)
#CONFIG_BOARD_MICROBIT=y # (under development)
#CONFIG_BOARD_EPIPHANY=y

#
# 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

#
# Board specific options
#
CONFIG_GNATAVR_COMPILER="/opt/avr-ada/avr-ada-122"
CONFIG_RP2040_SERIAL=y          # Comment to use usb library instead

#
# M2OS_Tool specific options
#
CONFIG_HOSTGNATFORTOOL="${HOME}/opt/GNAT/2019"
```

FIG. 22– CONFIG_PARAMS.MK MODIFICADO PARA EL DESARROLLO Y DEPURACIÓN DE M2OS PARA LA RP2040

config.mk toma las opciones sin comentar y escribe ciertas variables que el makefile utilizará durante su ejecución para cumplir los ajustes establecidos.

Uso de libpicoSDK en Ada

Para que el entorno de Ada pueda encontrar la librería e interpretar sus objetos, hay que usar un script de enlace (*link script*), un archivo de texto que dicta como manejar las distintas fuentes de código para generar el ejecutable final.

`memmap_default.ld`, disponible en ...***/M2OS/arch/rp2040/pico-SDK/,*** se encarga de esto. Para añadirlo a Ada, hay que modificar los switches de enlazado de M2OS, explicados más arriba. Estos se encuentran en el archivo presente en la ruta ...***/M2OS/arch/rp2040/ pico-SDK/shared_switches_rp2040.gpr***. Aquí, en el paquete ***Default_Swtiches***, hay que añadir la carpeta donde se encuentra el script y marcarlo para su uso: así vemos las líneas

```
"-L" & project'Project_Dir & "pico-SDK",
"-Tmemmap_default.ld");
```

Y en el paquete ***Trailing_Swtiches***, además, se añade para su uso con las flag.

`"-Wl,--whole-archive", "-lpicosdk".`

La flag ***whole-archive*** indica que se compilará toda la librería y no solo las referencias usadas por M2OS, ya que es imposible saber que partes de la picoSDK usará el usuario en cada momento. En caso contrario, habría que recompilar M2OS con cada nuevo ejemplo o aplicación.

Ambos paquetes han de tener este cambio, tanto para C como para Ada.

```
   for Default_Switches ("Ada") use
     Wrap_Switches
     & ("-Wl,--gc-sections",
        "-Wl,-Map,map.txt",
        "-L" & project'Project_Dir & "libm2os",
        "-L" & project'Project_Dir & "pico-SDK",
        --project'Project_Dir & "pico-SDK/irq_handler_chain.S.obj",
        --project'Project_Dir & "pico-SDK/crt0.S.obj",
        "-Tmemmap_default.ld");
   for Default_Switches ("C") use
     Wrap_Switches
     & ("-Wl,--gc-sections",
        "-Wl,-Map,map.txt",
        "-L" & project'Project_Dir & "libm2os",
        "-L" & project'Project_Dir & "pico-SDK",
        "-Tmemmap_default.ld");

   for Trailing_Switches ("Ada") use
     ("-Wl,--start-group",
      Runtime & "/adalib/libgnat.a", "-lm2os", "-lgcc", "-lm2os",
      "-Wl,--whole-archive", "-lpicosdk", "-Wl,--no-whole-archive",
      "-lc",
      "-Wl,--end-group",
      project'Project_Dir & "pico-SDK/bs2_default_padded_checksummed.S");
   for Trailing_Switches ("C") use
     ("-Wl,--start-group",
      Runtime & "/adalib/libgnat.a", "-lm2os", "-lgcc", "-lm2os",
      "-Wl,--whole-archive", "-lpicosdk", "-Wl,--no-whole-archive",
      "-lc",
      "-Wl,--end-group",
      project'Project_Dir & "pico-SDK/bs2_default_padded_checksummed.S");
end Linker;
```

FIG. 23 – SHARED_SWITCHES_RP2040 CON LAS FLAGS AÑADIDAS.

Depuración en M2OS y el entorno de ejecución (RTS)

Para poder activar la depuración, primero, es necesario modificar el archivo ***config_params.mk***. Dentro de este archivo, se debe comentar la línea que define ***CONFIG_SIZEOPTIMIZATION*** en las secciones correspondientes a ***M2***, ***APP*** y ***RTS***.

Las flags que activa esta opción están pensadas para la ejecución en entorno final, ya que reducen el tamaño del archivo y todos los checks en tiempo de ejecución que no sean explícitamente pedidos por el usuario (flags de compilador ***–Os*** y ***–gnatp*** respectivamente), además de descomentar la línea ***CONFIG_DEBUG*** en los tres apartados. Esto asegura que las optimizaciones de tamaño se desactiven y se habilite la generación de información de

depuración (flags de compilador ***–O0*** y *–**g*** respectivamente), para el sistema operativo, las aplicaciones creadas, y el entorno de ejecución o runtime.

Es importante tener en cuenta que el runtime de Ada (RTS) no toma automáticamente las flags de configuración definidas en ***config_params.mk.*** Esto es debido a que M2OS corre bajo el perfil *Ravenscar*. Este perfil es un conjunto de restricciones y reglas para el lenguaje Ada, diseñado para aplicaciones de tiempo real con requisitos críticos de seguridad y confiabilidad.

Por lo tanto, hay que editar directamente el archivo en ***.../M2OS/gnat_rts/rp2040/Ravenscar_build.gpr.*** En este archivo, se debe acceder a ***target_options.gpr*** y agregar la opción de depuración -g a la variable ***ADAFLAGS***. Esto asegura que el compilador GNAT genere la información de depuración necesaria en todos los archivos generados dentro de GNATSudio.

Si se presta atención, una manera más fácil para el usuario de hacerlo sería usar la variable **BUILD** externa. Esta se lee en el archivo y, si fuera **DEBUG**, y no el valor por defecto **PRODUCTION**, entonces entraría a una sección que definiría automáticamente la flag de debug "***-g***". Sin embargo, ya que parece que en esta rama no se toma en cuenta y, habría que hacer varios ajustes al entorno de generación CMake de M2OS, que están fuera del objetivo principal de este proyecto y se pueden optimizar después, se ha añadido la flag directamente a las flags generales en el bloque "*PER LANGUAGE FLAGS*".

En cualquier caso, es importante, una vez se desarrolle dicha parte del sistema de generación de código, retirar la flag de depuración de las flags generales, ya que estas podrían provocar problemas de optimización en un entorno de producción.

```
--  their own version of this file.

abstract project Target_Options is

   type Build_Type is ("Production", "Debug", "Assert", "Gnatcov");
   Build : Build_Type := external ("BUILD", "Production");

   --  COMFLAGS here is common flags (used for C and Ada).
   COMFLAGS := ("-fcallgraph-info=su,da",
         "-ffunction-sections",
         "-fdata-sections");
   COMGNARLFLAGS := ("");

   --  Per language flags (COMFLAGS will be added later)
   ADAFLAGS := ("-g","-gnatg", "-nostdinc", "-fno-delete-null-pointer-checks");
   ASMFLAGS := ("");
   CFLAGS := ("-g","-DIN_RTS",
         "-Dinhibit_libc", "-Werror", "-Wall");
   case Build is
      when "Production" =>
         --  Optimize
         COMFLAGS := COMFLAGS & ("-O2");
         ADAFLAGS := ADAFLAGS & ("-gnatp", "-gnatn2");
      when "Debug" =>
         --  Disable optimization and add debug symbols
         COMFLAGS := COMFLAGS & ("-O0", "-g");
         ASMFLAGS := ASMFLAGS & ("-g");
      when "Assert" =>
         --  Possibly enable assertions. This might use too much memory on
         --  some systems or could be too slow.
         COMFLAGS := COMFLAGS & ("-O");
         ADAFLAGS := ADAFLAGS & ("-gnata");
      when "Gnatcov" =>
         --  For coverage
         COMFLAGS := COMFLAGS & ("-O0", "-g", "-fdump-scos",
                                 "-fpreserve-control-flow");
   end case;
```

FIG. 24 – ARCHIVO TARGET_OPTIONS.GPR CON LA VARIABLE **ADAFLAGS** Y EL BLOQUE CASE QUE SE PODRÍA AJUSTAR AL MODO DEBUG, CON LA VARIABLE "-G" YA AÑADIDA.

Una vez realizadas estas modificaciones, se deben eliminar todos los archivos generados previamente y recompilar, asegurando que las nuevas configuraciones sean aplicadas correctamente. Después, se ejecuta el comando ***make install_no_tool*** para volver a instalar el proyecto sin herramientas adicionales, asegurando que todas las librerías y componentes se generen con las nuevas configuraciones.

Por otro lado, las dos librerías que vienen por defecto con M2OS tienen las flags de depuración activadas. Además, el usuario puede elegir si activarlas al hacer la librería mediante el script. Sin estas, aunque activáramos las flags de depuración para M2OS, no habría información de depuración en la librería ya que es un archivo externo al entorno de M2OS y NO se genera con el compilador de GNAT.

Depuración en proyectos de M2OS mediante GNATStudio

Ahora, se han de activar las opciones de depuración en programas ejecutados usando M2OS, como por ejemplo los test y las demos, y hacer esas opciones accesibles en GNATStudio.

Generalmente, cada proyecto de GNATStudio viene con una serie de botones rápidos para tareas como compilar o depurar. Sin embargo, para la RP2040 la dificultad radica tanto en la adición de la librería externa como el acceso al entorno de depuración de la placa mediante OpenOCD y Minicom.

Para activar la depuración en el proyecto GPR utilizado para los tests u otro proyecto específico para M2OS, se deben seguir estos pasos:

1. Modificar el script de compilación de GNATStudio

   El botón *build all* en la barra de herramientas de GNATStudio no tiene la opción de depuración añadida por defecto. En su lugar, se debe ir al menú build > project > build all en la barra superior.
   Al hacer clic, se abrirá una ventana con opciones de compilación, el cual habrá que modificar para añadir la flag de depuración "-g" al comando original, quedando así:

%builder -g -d %rbt %rd %eL -P%PP %config %autoconf %X -largs -Wl,-Map=map.txt.

   Este comando placeholder es traducido por GNATStudio para adaptarse al entorno específico:

   - **%builder**: representa el comando de compilación a usar (gprbuild)
   - **-g, -d:** representan las flags de depuración
   - **%rbt**: representa el objetivo de compilación (--target=arm-eabi)
   - **%rd, %eL**: representan el path a un directorio de compilación específica y flags extra de a añadir, respectivamente (no usadas)
   - **-P%PP**: flags que denotan el path al proyecto a compilar (archivo formato gpr)
   - %config, %autoconf: archivos de configuración y scripts a añadir
   - **%X**: flags de compilación extra.
   - -largs -Wl,-Map=map.txt: generar un mapa de linkado en el archivo map.txt

En resumidas cuentas, este placeholder genera el comando de compilación para todo el proyecto que se quiera compilar, con el target adecuado, y con flags de compilación y con el mapa de símbolos guardado en un archivo de texto.

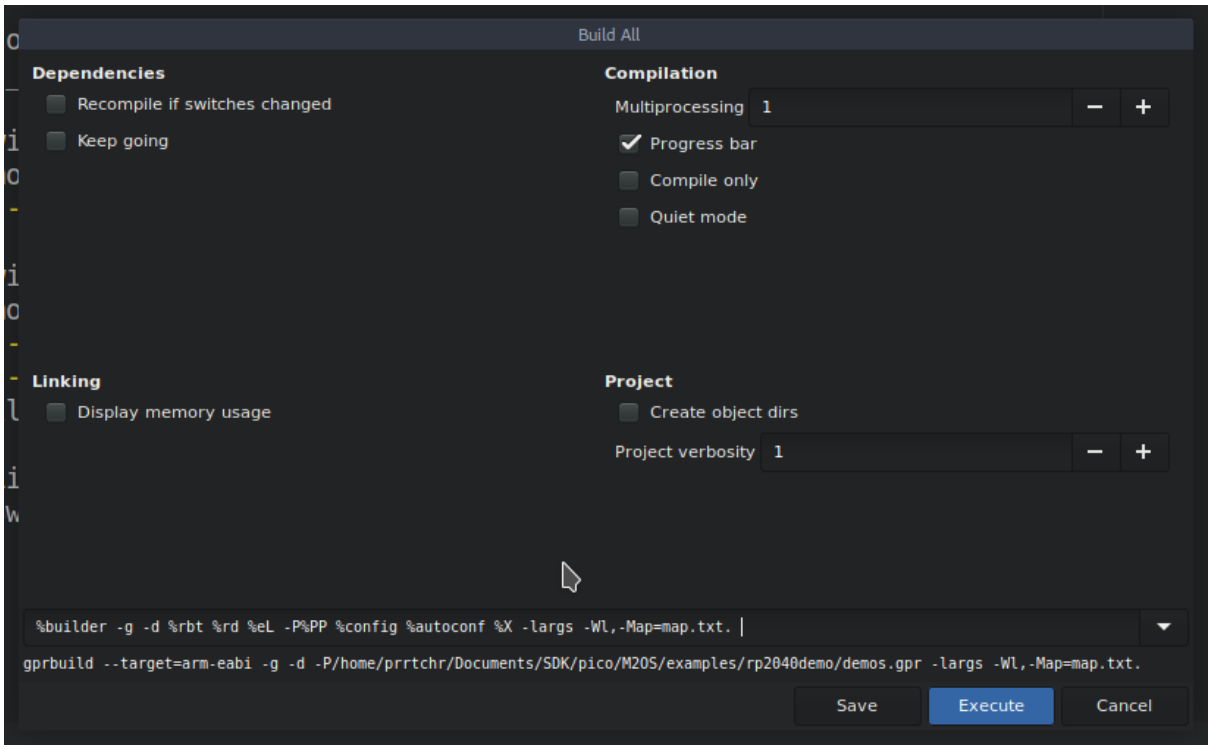


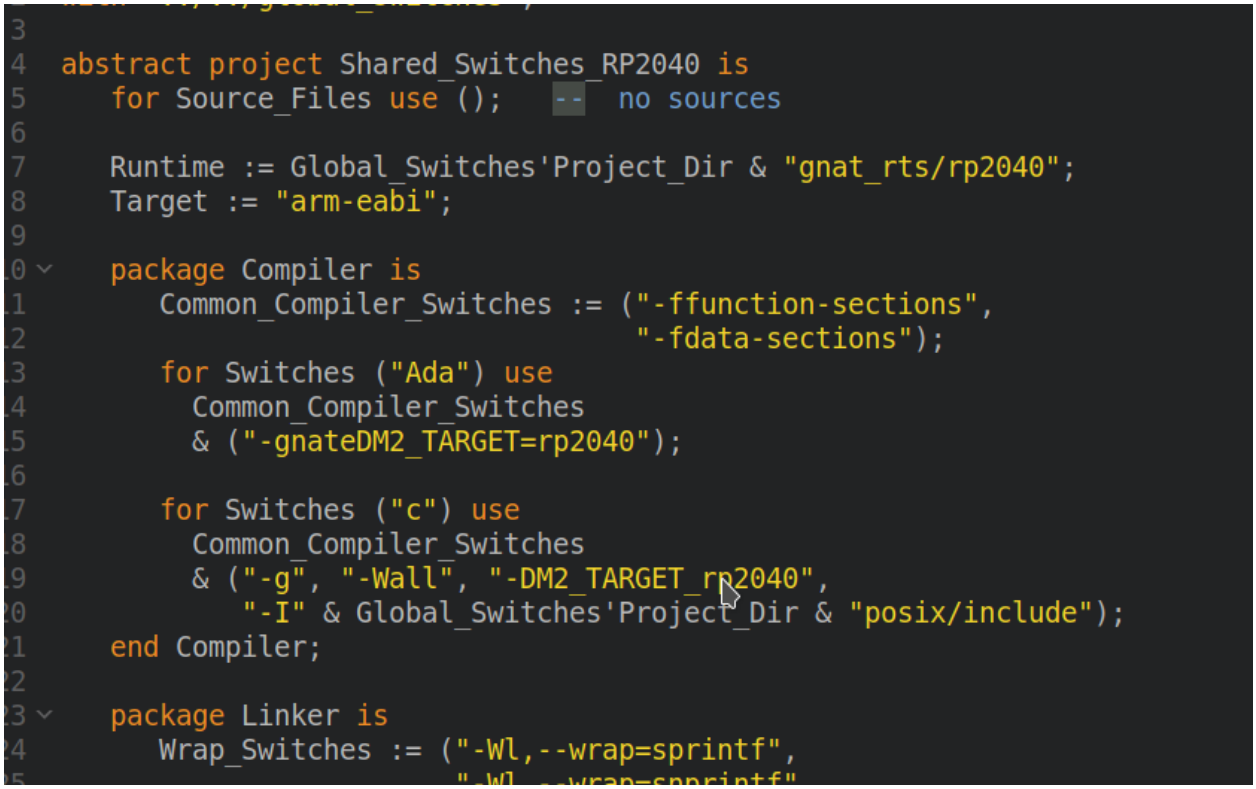
FIG. 25 – MENÚ BUILD ALL CON EL COMANDO MODIFICADO. DEBAJO SE PUEDE VER LA "TRADUCCIÓN" DEL COMANDO PARA EL PROYECTO ACTUAL.

También cabe aclarar que en las opciones de GNATStudio, se puede crear o modificar el botón build all para que ejecute el comando con la flag de depuración incluida. Esto permite lanzar directamente la construcción con depuración desde la barra de herramientas.

2. Activar depuración en tests escritos en C

Para las pruebas escritos en C, se debe añadir la opción ***"-g"*** a los switches del compilador C en el paquete ***Compiler*** dentro del archivo ***shared_switches_rp2040*** para el lenguaje C, accesible desde cualquier proyecto que use M2OS.

Como ya se ha explicado, los switches generan las flags y opciones para todo o ciertas partes del proyecto a generar de manera programática. En este caso, esto hace que los archivos C que se compilen bajo GNAT (más específicamente los wrappers que se mencionarán más adelante) tengan las flags de compilación puestas ya que la opción -g del comando de build / build all en GNATStudio solo se aplica a los archivos Ada.



```
3
4  abstract project Shared_Switches_RP2040 is
5     for Source_Files use ();    --  no sources
6
7     Runtime := Global_Switches'Project_Dir & "gnat_rts/rp2040";
8     Target := "arm-eabi";
9
10    package Compiler is
11       Common_Compiler_Switches := ("-ffunction-sections",
12                                    "-fdata-sections");
13       for Switches ("Ada") use
14         Common_Compiler_Switches
15         & ("-gnateDM2_TARGET=rp2040");
16
17       for Switches ("c") use
18         Common_Compiler_Switches
19         & ("-g", "-Wall", "-DM2_TARGET_rp2040",
20            "-I" & Global_Switches'Project_Dir & "posix/include");
21    end Compiler;
22
23    package Linker is
24       Wrap_Switches := ("-Wl,--wrap=sprintf",
```

FIG. 26 – SHARED SWITCHES DE LA RP2040 CON LA FLAG "-G" AÑADIDA A LOS SWITCHES DE C EN EL PAQUETE COMPILER.

Lanzar un programa con depuración

Para lanzar un test y realizar la depuración de un programa en GNATStudio utilizando una Raspberry Pi con picoprobe, openOCD, GDB y Minicom.

Hay dos maneras de iniciar una sesión de depuración. Manualmente o usando GNATStudio.

De manera manual, hay que usar dos comandos: GDB para iniciar una sesión de depuración y un script que lanza el ejecutable especificado a esta: ***launch_to_pico_probe.sh.***

Se utiliza minicom para la salida de texto serial. OpenOCD se usa para establecer la conexión con la placa (o la PicoProbe). En caso de ejecución normal se programa directamente la Raspebrry Pi Pico con el ejecutable. En caso de la depuración, esperar a recibir datos de una sesión de GDB sin programarlo ya que de la carga se encargará el depurador.

Un pseudocódigo aproximado sería:

COMPROBAR LA SINTAXIS (ejecutable valido, argumento run o debug)

LANZAR TERMINAL SERIAL mediante minicom

SI EL ARGUMENTO ES RUN:

LANZAR OPENOCD CON EL EJECUTABLE A UN RP2040

SI EL ARGUMENTO ES DEBUG:

LANZAR SOLO OPENOCD A UN RP2040 Y ESPERAR ENTRADA SERIAL

La sintaxis es: ***launch_to_pico_probe.sh. Nombre_de_ejeutable_a_lanzar debug***

Tras ejecutar este script, solo queda ejecutar GDB y "engancharlo" a la dirección del servidor de GDB local creado por OpenOCD

```
gdb
```

```
(gdb) file /path/to/compiled/ada/project
```

```
(gdb) target extended-remote localhost:3333
```

Es decir, iniciar GDB, marcar el archivo ejecutable como target, acción que también carga sus símbolos automáticamente, y luego marcar como objetivo el servidor GDB abierto por OpenOCD, en este caso en el puerto 3333 por defecto.

En GNATSudio, mediante la GUI, se puede cargar cualquier demo o programa de M2OS, que básicamente es una manera más visual de ejecutar los pasos mostrados arriba

1. Conectar con la placa: menú ***build > bareboard > debug on board*** (***launch_to_pico_probe.sh. debug)***
2. Inicializar el depurador de GNATStudio: menu ***debug > initialize*** (***gdb, target)***
3. Cargar el Archivo en la Pico Probe en el menú de depuración: menú ***debug > load file (gdb file)***

Tras estos pasos, se puede comenzar con la depuración en GNATStudio.

4. M2OS: adaptación del HAL y el kernel.

Tras preparar satisfactoriamente el entorno, se procede a la realización de la adaptación.

La modularización de este proyecto está hecha mediante varios archivos "*.gpr*" (GNAT PRoject). En cualquier proyecto de Ada que use M2OS se pueden ver las partes usadas para su correcta ejecución. La mayor parte del sistema subyacente de M2OS funciona sin cambios. El trabajo necesario para adaptar M2OS requiere que ciertas partes del sistema interactúen con el hardware de la placa objetivo:

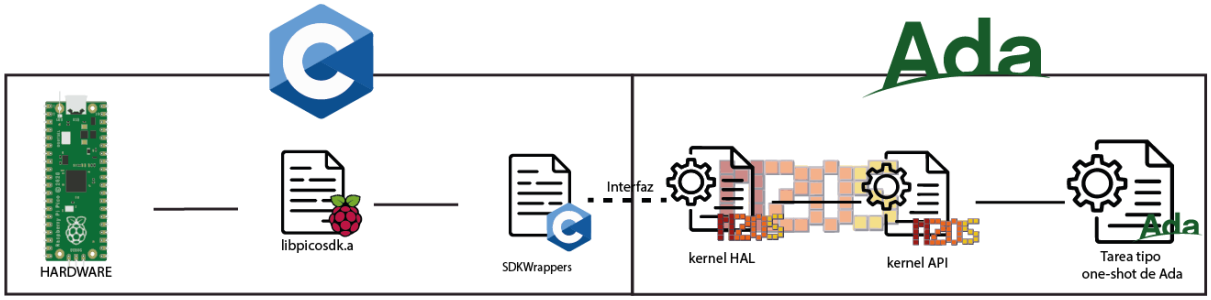


FIG. 27 – DIAGRAMA SIMPLE QUE MUESTRA LA INTERACCIÓN ENTRE COMPONENTES.

4.1 - Jerarquía de M2OS: kernel API y HAL.

Las dos partes principales del sistema que requieren de modificación son la API del Kernel y el HAL. El Kernel de M2OS encapsula las definiciones necesarias para la ejecución del sistema, como el planificador y la cola de tareas, la implementación del bloque de control de hilos (TCB), operaciones básicas de drivers y entrada salida directa...

Como el sistema operativo funciona en varias placas con arquitectura y diseño diferentes, este conjunto de definiciones tiene que ser pequeño. La implementación de servicios críticos, como el timer de sistema, se hace en la capa de abstracción de hardware (HAL). La interfaz encargada de, esencialmente, conectar el hardware de la placa usada con las diferentes partes del sistema operativo M2OS.

Se definen también una serie de parámetros: ***HWTime_Hz*** marca, en hertzios, la frecuencia de actualización del reloj del sistema, así como ***HWTime_Interval_us*** lo marca en microsegundos, que es la resolución de la Raspberry Pi Pico. ***HWTime*** marca el valor del contador del timer en un momento dado.

Por otro lado, la API del kernel expone ciertas partes del sistema a las aplicaciones de usuario. Es necesario exponer solo las partes explícitamente necesarias, ya que el kernel se encarga de las tareas de privilegio elevado. Esta interfaz de cara al usuario se adaptará para ciertas funciones relacionadas tanto con el timer como con la ejecución multinúcleo.

El programa llega, durante la inicialización, a la función *Initialization* del HAL. En esta, se tienen que hacer varias tareas: inicializar el timer y, si se necesitara, cualquier otro subsistema de la placa en ***Initialize_Board***, que en este caso no se usa, y la dirección base del stack del sistema. Es crucial que todos los métodos en esta función se ejecuten satisfactoriamente y realicen sus tareas para la correcta inicialización y funcionamiento de M2OS.

```
   procedure Initialization (OS_Tick_Handler : OS_Tick_Handler_Ac)is

      -- Timer init from m2_hal_timer
      procedure M2_HAL_Init_Timer  (Period_Us : Interfaces.Integer_64;
      M2_Handler    : access procedure);
      pragma Import(C, M2_HAL_Init_Timer, "m2_hal_init_timer");

      function To_Unsigned_32 is
        new Ada.Unchecked_Conversion (System.Address, Interfaces.Unsigned_32);

   begin
      pragma Debug
        (Debug.Assert (System.Address'Size = Interfaces.Unsigned_32'Size));

      HAL.Disable_Interrupts; --Disable interrupts during init

      Initialize_CPU;  --CPU related init (inter-core IRQ)
      --Initialize_Board; --  Initialize and start timer (Microbit)

      HAL.OS_Tick_Handler := OS_Tick_Handler; --Link to the OS tick handler

      M2_HAL_Init_Timer(Period_Us  => HWTime_Interval_us,
                        M2_Handler => Sys_Tick_Handler'Access); --Init system timer via PICOSDK wrapper

      Stack_Base := To_Integer (Stack_Base_Linker_Script'Address); --Addres of the base of the stack

      Initialized := True; --End initialization

      --HAL.Enable_Interrupts; --Re-enable interrupts?
   end Initialization;

end M2.HAL;
```

FIG. 28 – FUNCIÓN DE INICIALIZACIÓN DEL HAL

4.2 - Interacción entre Ada y C: picoSDK wrappers.

Una parte esencial de la integración es entender como llamar a las funciones de la librería creada *picosdk.a* desde Ada. Esto implica dos pasos:

Para poder invocar las funciones del PicoSDK desde Ada, se utiliza una directiva. Las directivas (*pragmas* en inglés) son instrucciones directas para el compilador desde el código. Estas marcan cómo debe comportarse el compilador ante cierta parte del código. En este caso, se utiliza la directiva ***pragma import*** en Ada. Este ***pragma*** se usa para indicar que un método usado en el código no está directamente disponible en el entorno local, sino que es una función externa y normalmente escrita en otro lenguaje distinto. Se necesita una función declarada equivalente en C, con el mismo tipo de retorno y argumentos. El pragma tiene el siguiente formato:

pragma import (lenguaje a importar, función local, "función a importar")

Para el desarrollo de M2OS, permite definir una función en Ada que se refiera a una función escrita originalmente en C. Esta función tiene que estar disponible para que el compilador la vincule correctamente, y en nuestro caso se encuentra en la librería de PicoSDK creada.

```
procedure M2_HAL_Init_Timer  (Period_Us : Interfaces.Integer_64;
M2_Handler    : access procedure);
pragma Import(C, M2_HAL_Init_Timer, "m2_hal_init_timer");
```

FIG. 29 – EJEMPLO DE IMPORT- EL PROCEDIMIENTO EQUIVALENTE DE ADA SEGUIDO DEL PRAGMA IMPORT DE LA FUNCIÓN EN C.

Además de las llamadas directas mediante ***pragma import*** se utilizan wrappers o envoltorios. Los wrappers son código en C que encapsula de manera sencilla las funciones requeridas del PicoSDK. Estos se encuentran en

.../localización/de/M2OS/arch/rp2040/pico-SDK/SDKWrappers

Aunque la mayoría de las funciones pueden ser llamadas directamente desde Ada utilizando ***pragma import*** , se crean métodos de envoltura para prácticamente todas las funciones del PicoSDK. Esto se hace por varias razones:

- Conveniencia: los wrappers simplifican la llamada a las funciones desde Ada, especialmente cuando se necesita realizar una serie de pasos repetitivos o establecer configuraciones comunes antes de la llamada a la función original del PicoSDK
- Manejo de Complejidad: algunas funciones del PicoSDK pueden tener argumentos complejos o estructuras que no son directamente compatibles con Ada. Los wrappers permiten transformar estos argumentos complejos en formas más sencillas o compatibles con Ada. Por ejemplo, cuando se usan estructuras de C para definiciones del bloque de control de hilos (TCB) o del temporizador, replicar estas estructuras en Ada puede ser muy tedioso y propenso a errores. Los wrappers manejan esta traducción de manera eficiente al ser estas estructuras creadas en C, el idioma nativo de la librería, en vez de hacer que el desarrollador tenga que crear estructuras altamente complejas en Ada que podrían fallar o no ser compatibles.

También hay una situación en la que se necesita lo contrario: llamar a una función de Ada desde C. Para esto, se utiliza la directiva ***pragma export*** en Ada, que como su propio nombre indica, “exporta” el método al lenguaje deseado y bajo el nombre de método deseado. El formato es:

pragma export (lenguaje objetivo, función local a exportar, “función en el lenguaje exportado”)

Tras eso, se crea una cabecera ***ada2c.h*** para que el compilador de C que crea la librería tenga unas "referencias" (aunque en el momento de la creación de la librería esas referencias no lleven a ningún sitio) para que, cuando se ejecute en M2OS, estas queden vinculadas.

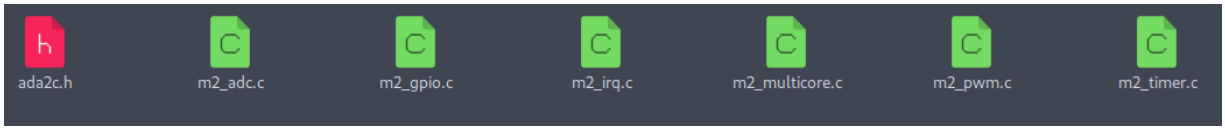


FIG. 30 – CONTENIDOS DE LA CARPETA SKDWRAPPERS JUNTO CON LA CABECERA ADA2C.H

```
--Export the function to C as __gnat_sys_tick_trap to work with the m2_hal_timer wrapper
pragma Export (C, Sys_Tick_Handler, "__gnat_sys_tick_trap");
```

FIG. 31 EJEMPLO DE EXPORT- LA REFERENCIA EN LOS WRAPPERS SE HARÁ A TRAVÉS DE LA CABECERA ADA2C.H

Por último, hay que asegurarse que los argumentos de métodos cruzados entre lenguajes son compatibles. Se asegura, por ejemplo, que los tipos de argumento usados en los wrappers sean compatibles con el código en Ada utilizando tipos de datos definidos en el paquete estándar de Ada ***Interfaces.C.***

Los wrappers se compilan junto con la librería para que el compilador de Ada los reconozca en los ***pragma import.*** Y se utilizan en lugar de las llamadas directas a las funciones de PicoSDK disponibles en la librería creada.

Respecto a esto, hay que tener en cuenta que las partes de PicoSDK que tienen wrappers tienen que tratarse de distinta manera: si intentáramos añadir un wrapper a la librería sin tener su librería correspondiente cargada, la librería no se podría usar.

Es decir que, a la hora de crear la librería, cuando se usen (o no) estas partes del PicoSDK, hay que también añadir o quitar sus wrappers para evitar problemas de compilación de la librería.

4.3 - Timer de sistema e integración

El timer del sistema es una pieza fundamental en cualquier sistema operativo, actuando como el "reloj del sistema" y proporcionando la base para medir el tiempo y ejecutar tareas periódicas. A continuación, se ofrece una explicación detallada y clara de cómo se implementa y utiliza el timer del sistema en el RP2040 con M2OS.

El reloj del sistema es un contador que aumenta a intervalos regulares, usando una fuente constante de pulsos para medir el tiempo. En el caso del RP2040, se emplea un contador de hardware fijo presente en la placa. Este contador de hardware permite que el sistema mantenga una medición precisa del tiempo.

En cuanto al código, el reloj del sistema se maneja como un timer de repetición normal. Sin embargo, hay dos funciones específicas para inicializar y cancelar el reloj del sistema. El archivo ***m2_timer.c*** contiene una estructura que mantiene la información del timer del sistema, actuando como referencia y punto de acceso para el timer si es necesario.

Para configurar el timer del sistema, se utiliza la función ***add_repeating_timer_us*** dentro del wrapper. Esta función programa un timer que llama a una función específica, introducida como puntero, cada vez que expira el tiempo configurado en microsegundos. La función llamada por el timer es ***m2_tick_handler_wrapper***, una función del wrapper en C. Esta función, a su vez, llama a una función de Ada encargada de aumentar el contador del sistema M2OS.

La función en Ada que se encarga de manejar los ticks del sistema se llama ***Sys_Tick_Handler***. Cada vez que el timer expira, ***m2_tick_handler_wrapper*** llama a ***Sys_Tick_Handler***, que actualiza el contador del sistema de M2OS, manteniendo el tiempo del sistema correctamente. Para hacer esto, ***Sys_Tick_Handler*** utiliza otra función del SDK, **time_us_64**, que devuelve el valor del contador de hardware en microsegundos en un entero sin signo de 64 bits. Este valor se usa para actualizar el valor del timer de M2OS.

Respecto al periodo, debe ser un número que permita la ejecución de tareas de manera fluida, y que a la vez permita llevar al timer un conteo que puedan usar otros elementos del Sistema Operativo de manera fluida. En nuestro caso, el timer expira cada 0.1 milisegundos, 12500 veces más lento que la frecuencia nominal de la placa, 125Mhz (8 nanosegundos).

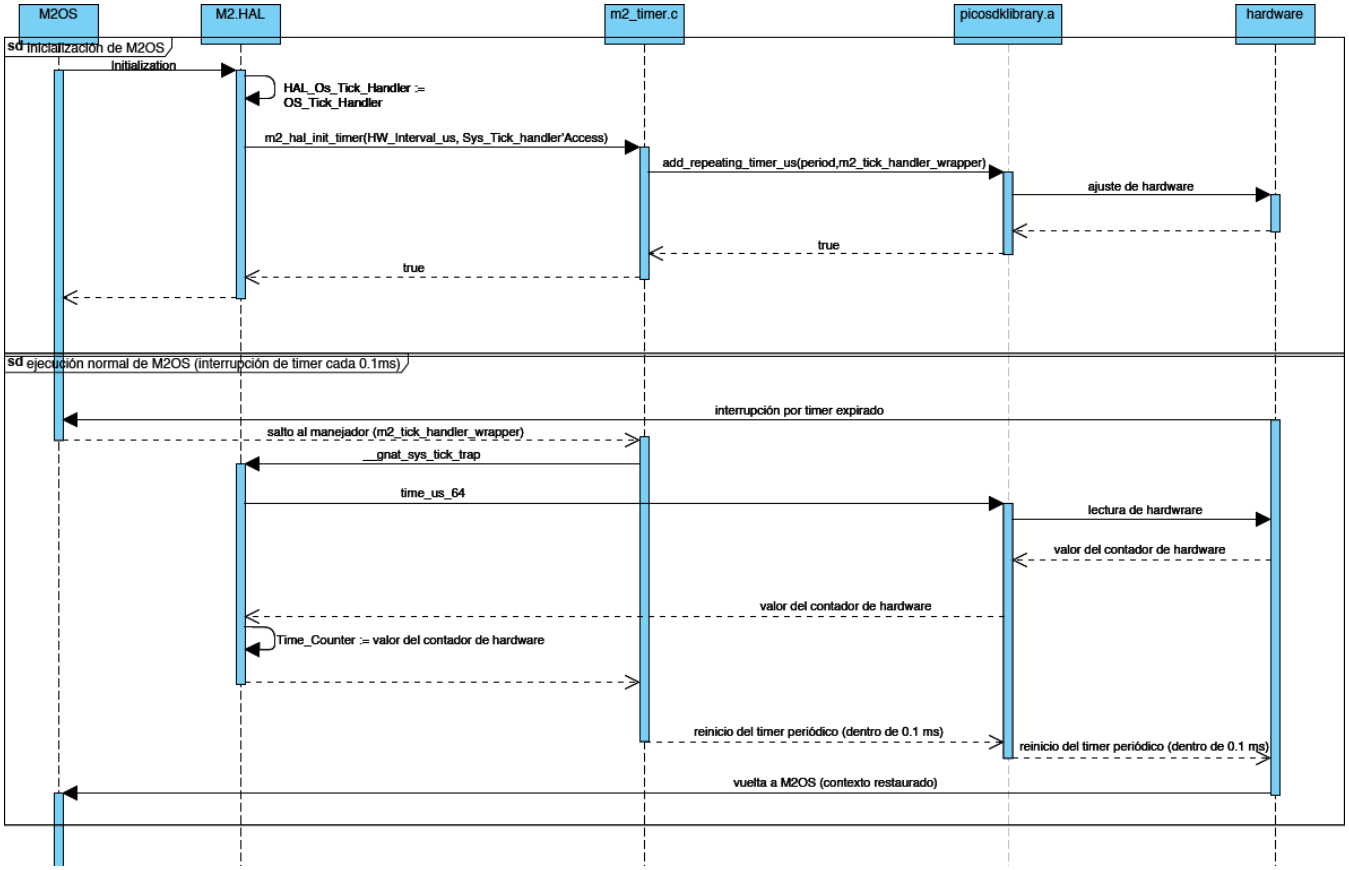


FIG. 32 – DIAGRAMA DE SECUENCIA QUE MUESTRA LA INICIALIZACIÓN Y UNA INTERRUPCIÓN DEL TIMER DEL SISTEMA.

Existe un problema relacionado con la resolución del timer de M2OS: el timer devuelve el tiempo en microsegundos, pero el contador de hardware tiene una resolución más alta. Para solucionar este problema, se debe ajustar el número de ticks del sistema en **s-osinte.*Ticks_Per_Second*** en el archivo ***M2OS/gnat_rts/rp2040/gnarl_user***, para que coincida con ***HWTime_Hz*** en el HAL. Esto asegura que la frecuencia de los ticks del sistema esté alineada con la resolución del hardware.

En la API, se expone la función ***Get_HWTime*** del HAL a través del método ***Get_Time*** para el usuario.

4.4 - Entrada/salida, pines e interacción con periféricos.

Gracias a los wrappers, la interacción con los pines, las entradas analógicas y el resto del hardware de control de periféricos es tan fácil como importar las funciones necesarias como si de un archivo C se tratase.

Se han creado wrappers para algunas funciones de entrada-salida: lectura analógica, lectura y escritura digital y modulación PWM. No es necesario pasar por los módulos de drivers o entrada salida directa, del kernel ya que estos se encargan de la conexión con periféricos no estándar y tareas más complejas. La mayoría de los dispositivos se pueden controlar mediante el manejo de pines, de manera idéntica a como se haría en un archivo C con picoSDK de manera directa. Un ejemplo de su uso se detalla en el apartado 6.

Por cada sistema periférico que se quiera usar hay que asegurarse de que la librería tiene cargada dicho código: por ejemplo, para usar los pines GPIO añadir ***hardware_gpio***, para añadir soporte para el uso de modulación por amplitud de pulso usar ***hardware_pwm.***.

Siguiendo la documentación del PicoSDK para cada tipo de periférico y sabiendo el funcionamiento de cada uno es suficiente para poder realizar aplicaciones que las utilicen

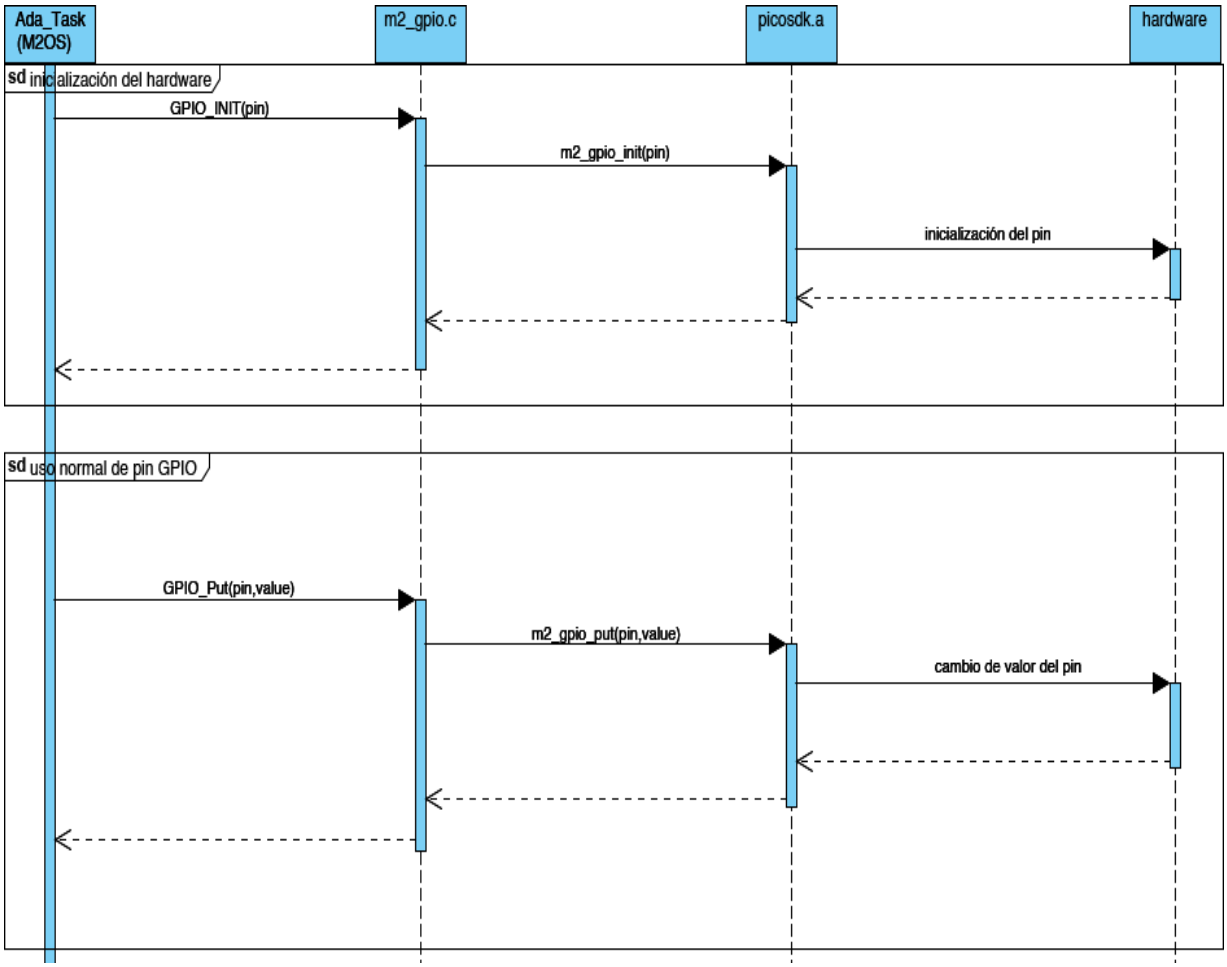


FIG. 33 – DIAGRAMA DE SECUENCIA DE INTERACCIÓN CON PERIFÉRICOS - EJEMPLO CON GPIO

5. Sincronización, comunicación e implementación en entorno de doble núcleo.

El trabajo detallado hasta ahora permite que programas compatibles con M2OS –tareas de tipo one-shot escritas en Ada- en un solo core.

El microprocesador RP2040 es un sistema de núcleo doble, denominados núcleo 0 y núcleo 1. Es decir, tiene dos unidades físicas funcionales para la ejecución de tareas. Por defecto, en todas las librerías estándar de la Raspberry Pi Pico, incluida PicoSDK, sólo se utiliza un núcleo, el 0, para ejecutar todo el código.

Se ha desarrollado el control básico del segundo núcleo, para ejecutar tareas en este, con dos modos de comunicación entre los núcleos del RP2040, del que solo se utiliza uno. Además, se ha implementado el uso de spinlocks como primitiva de sincronización a partir de las funciones disponibles en el PicoSDK.

NUM_CORES es un parámetro definido en el HAL que guarda el número de cores del sistema en cuestión: 2. Esto está pensado para poder expandirse a otros microprocesadores similares que tengan más cores en el futuro.

Ahora se necesita Inicializar lo referente a la CPU, como pueden ser las colas de mensajes internúcleo o las interrupciones no estándar, si se utiliza la comunicación inter-núcleo por hardware. Es necesario, además, reiniciar el estado del core 1 a un estado conocido antes de su uso, y verificar que ciertos parámetros, como el número de núcleos, son ajustados correctamente para la placa, ya que ciertas funciones multinúcleo dependen del valor de este.

```
procedure Initialize_CPU;
procedure Initialize_CPU is
   -- Set an exclusive handler and enable for interrupt line 15 (core 0)
   procedure M2_HAL_set_and_enable_intercore0_IRQ;
   pragma Import(C, M2_HAL_set_and_enable_intercore0_IRQ, "m2_set_and_enable_core0_intercore_IRQ");

   -- Set an exclusive handler and enable for interrupt line 16 (core 1)
   procedure M2_HAL_set_and_enable_intercore1_IRQ;
   pragma Import(C, M2_HAL_set_and_enable_intercore1_IRQ, "m2_set_and_enable_core1_intercore_IRQ");

   -- Reset core 1 state to default knowns tate
   procedure M2_HAL_multicore_reset_core1;
   pragma Import(C, M2_HAL_multicore_reset_core1, "multicore_reset_core1");

    -- Drain the read FIFO of the calling core (0 For this function)
   procedure M2_HAL_multicore_fifo_drain;
   pragma Import(C, M2_HAL_multicore_fifo_drain , "m2_multicore_fifo_drain");

   -- Clear any pending IRQ for the calling core intercore FIFOs
   procedure M2_HAL_multicore_fifo_clear_irq;
   pragma Import(C, M2_HAL_multicore_fifo_clear_irq , "m2_multicore_fifo_clear_irq");
begin
    --NUM_COUNT CORRECTEDNESS CHECK
   pragma Compile_Time_Error (NUM_CORES < 2,
                              "Core count error. Please use at least 2 cores to run M2OS :).");

   --Multicore (more than 2) implementation unavaialble. Remove this pragma once its available
   pragma Compile_Time_Warning (NUM_CORES > 2,
                           "Multicore architecture not supported yet. This version will only use 2 cores");

   --Set core 1 to known state. Clear core 0 FIFO states
   M2_HAL_multicore_reset_core1;

   --Boilerplate for possible IRQ implementation
   --M2_HAL_multicore_fifo_drain;
   --M2_HAL_multicore_fifo_clear_irq;
   --On this block, initialize inter-core interrupts via PICOSDK hardware_irq
   --M2_HAL_set_and_enable_intercore0_IRQ;
   --M2_HAL_set_and_enable_intercore1_IRQ;

   pragma Compile_Time_Warning (True, "Initialize_CPU");
end Initialize_CPU;
```

FIG. 34 – MÉTODO INITIALIZE_CPU LLAMADO DESDE INITIALIZE, QUE HACE LAS COMPROBACIONES Y REINICIA EL CORE 1, USANDO VARIOS MÉTODOS DEL WRAPPER M2_MULTICORE.C

Se necesita cargar en la librería las partes pertinentes de picoSDK: ***pico_multicore*** para las funciones multinúcleo estándar, y ***hardware_irq***, ***hardware_sync*** y ***hardware_claim*** para el desarrollo de la sincronización internúcleo.

5.1 - Wrapper de funciones multinúcleo

En este proyecto, M2OS se ejecuta en el núcleo 0, mientras que el núcleo 1 se dedica a ejecutar una única tarea específica. Es importante destacar que, si por alguna razón el núcleo 0 termina de ejecutar M2OS o el kernel “entra en pánico” o se cuelga, la tarea en el núcleo 1 pasará a estar en un estado desconocido hasta que se reinicie la placa. Al reiniciarse, M2OS también reiniciará el núcleo 1 mediante el método de inicialización.

Las funciones utilizadas para la implementación de control multinúcleo se encuentran en el wrapper ***m2_multicore.c.***

Los dos métodos básicos son ***`m2_launch_task_on_core1,`*** que toma un puntero a una dirección de memoria desde la que el core 1 empezará a ejecutar código. En el HAL se traduce al método ***`Sys_task_launch_core`***, que toma un punto de entrada y un identificador. Este método, solo puede mover ejecuciones al core 1, es decir que no hay manera de poder mover código de vuelta al 0, ya que eso interrumpiría la ejecución de M2OS. En la API de usuario, esto se traduce en dos métodos, en el que uno es la función equivalente del HAL, ***`Launch_To_Core`***, con argumentos de puntero a código y núcleo en el que lanzarlo, y ***`Move_To_Core`***, que solo tomo de argumento el core, y lanzará el punto de entrada de la tarea actual en el core especificado.

Por otro lado, la función de wrapper ***`m2_get_core_num`***, devuelve el identificador del núcleo donde se ejecuta la llamada (0 o 1), leyendo el valor del registro *CPUID* en el bloque *SIO* del microprocesador.

En M2OS, lo más propicio es tener dos métodos separados para las ejecuciones en cada uno. La tarea tendrá un método de inicialización en el que se enviará el método deseado al core 1.

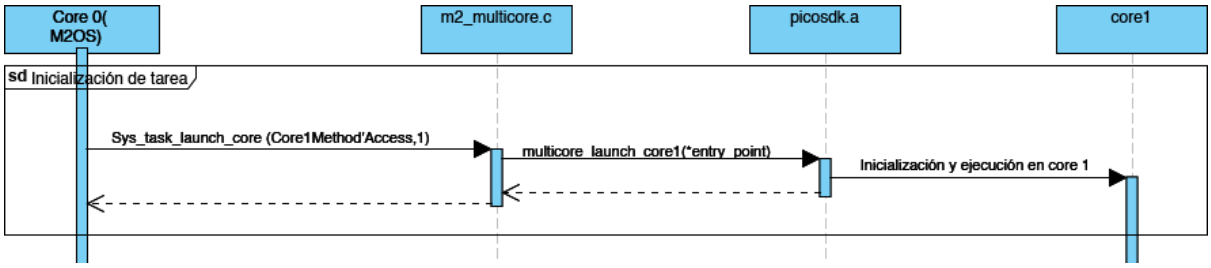


FIG. 35 – DIAGRAMA DE SECUENCIA CON LA INICIALIZACIÓN DE UNA TAREA MULTICORE.

5.2 - Comunicación inter-núcleo mediante hardware

La arquitectura de la placa Raspberry Pi Pico otorga métodos de hardware para la comunicación inter-nucleo.

Una forma de comunicación entre los núcleos del RP2040 es a través de interrupciones y colas "mailboxes" inter-núcleo. Estas hacen uso de las dos colas FIFO unidireccionales disponibles en el bloque SIO del subsistema de procesamiento. Estas colas utilizan interrupciones para el manejo inmediato de datos en las colas.

La implementación sería la siguiente: existen dos métodos en el HAL para el manejo de interrupciones generadas por la llegada de datos a la cola respectiva de cada núcleo: ***Sys_intercore0_IRQ_handler***, cuando el núcleo 1 deje datos en la cola del core 0 y ***Sys_intercore1_IRQ_handler*** cuando el core 0 deje datos en la core del core 1. Estas son exportadas mediante el ***pragma export*** y utilizadas en el archivo de wrapper ***m2_irq.c.***

Durante la fase de inicialización, se activarían los cores llamando a ***m2_set_and_enable_core0_intercore_IRQ y m2_set_and_enable_core1_intercore_IRQ*** respectivamente. Estas funciones crearían un manejador exclusivo para cada core utilizando las líneas de interrupción específicas para las interrupciones de cola inter-nucleo del core correspondiente. Que sea exclusivo significa que ningún otro manejador puede utilizar ya estas dos líneas a no ser que el manejador actual las libere.

El puntero a las funciones que se instalarían como manejadoras, serían las ya mencionadas dos funciones del HAL exportadas. No tienen exposición en la API porque son funciones de control del sistema y el usuario no debería necesitar acceder a ellas.

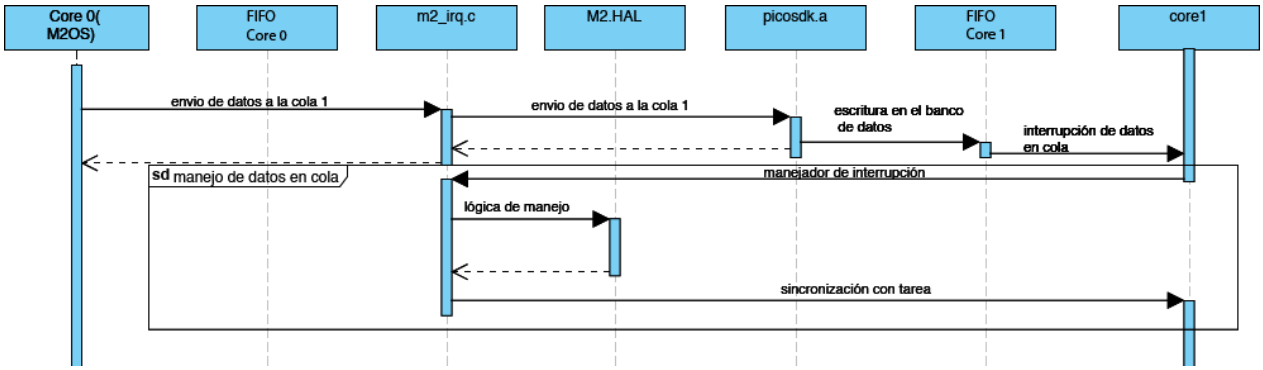


FIG. 36 – DIAGRAMA DE FLUJO EJEMPLIFICANDO UNA POSIBLE IMPLEMENTACIÓN NO BLOQUEANTE DE LA SINCRONIZACIÓN MULTINÚCLEO POR HARDWARE

Cada hilo puede ser completamente independiente, lo que significa que mantienen stacks (porciones de memoria) completamente separadas entre sí; Hay ciertas funciones del picoSDK en las que se puede especificar una dirección y tamaño del stack para el código lanzado en el core 1, aunque por simplicidad, no se usan en el proyecto. Además, las colas son atómicas e inmediatas.

Sin embargo, esta forma de comunicación requiere la creación de un framework de soporte para manejar las interrupciones y enmascarar el resto durante el proceso de tratamiento de manera manual. Ya que las colas multinúcleo son también usadas de manera independiente a la ejecución de M2OS por el PicoSDK, sería necesario usar enmascaramientos selectivos o cambiar los ajustes de M2OS para evitar esto, derivando en una serie de potenciales problemas.

Desactivar por completo o enmascarar las interrupciones puede llevar a comportamientos del sistema no deseados, lo que hace que su implementación sea más compleja y se necesitaría hacer un uso muy preciso de las máscaras de interrupciones para un correcto funcionamiento de este método

Actualmente, no se usa este método, aunque se ha dejado el código pertinente, como el código de inicialización de las líneas de interrupción en la función ***Initialize_CPU*** y el wrapper ***m2_irq.c*** para un posible uso futuro, además de las funciones exportadas de Ada, que están vacías.

Por último, ***m2_multicore_fifo_drain*** limpia las colas FIFO de ambos núcleos y ***m2_multicore_fifo_clear_irq*** elimina cualquier interrupción pendiente en estos.

5.3 - Comunicación inter-núcleo mediante software (Ada)

Otra forma de comunicación es la proporcionada por el lenguaje de programación en el que se desarrolla el programa de M2OS, Ada, junto con el funcionamiento de ciertos métodos multinúcleo de M2OS. Como ya se ha explicado, la función para lanzar una tarea al núcleo 1, por defecto, hace que ambas tareas compartan la misma pila o stack a no ser que se especifique lo contrario. Esto significa que una variable declarada como global (accesible desde las tareas ejecutadas en núcleos separados) permite la comunicación entre núcleos.

Esta forma de comunicación es la elegida debido a su simplicidad. La desventaja más obvia es que los núcleos acceden a datos no consistentes. Además, ambos núcleos deben gastar parte de su tiempo de ejecución en hacer polling de la variable o variables en cuestión, aunque permita más flexibilidad a la hora de leerlas y sea una implementación más sencilla.

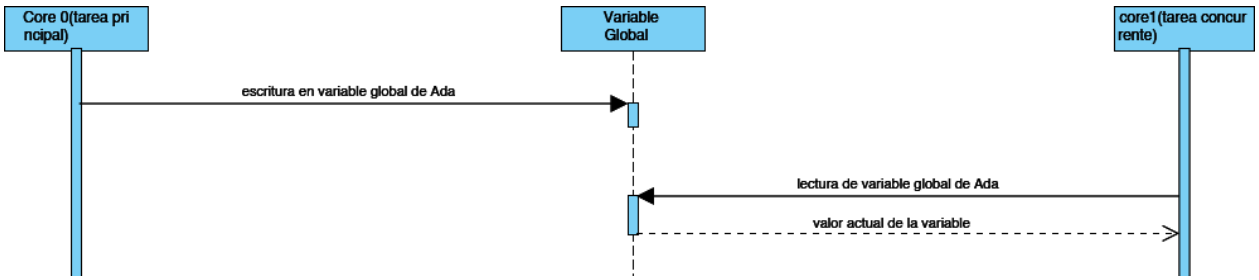


FIG. 37 – DIAGRAMA DE FLUJO CON VARIABLE COMPARTIDA. EL USUARIO DEBE DE ASEGURAR LA EXCLUSIÓN MUTUA DE LOS CORES AL ACCEDER A LA VARIABLE

El usuario ha de asegurar que el acceso a las variables compartidas sea mutuamente excluyente: es decir, hay que asegurar que mientras una de las tareas está leyendo o escribiendo en la variable, la otra no accede a la vez debido a la concurrencia de las tareas. Ignorar esto podría derivar en problemas debido a que el valor de uno de los núcleos podría estar desactualizado respecto del otro.

5.4 - Primitiva de sincronización básica: spinlock.

Para asegurar que le acceso concurrente a la variable compartida cumpla el principio de exclusión mutua, hay que desarrollar alguna primitiva de sincronización que bloquee a un core si el otro se encuentra dentro de una sección crítica protegida por dicha primitiva.

PicoSDK tiene funciones para manejar unas primitivas sencillas: los spinlock. Estos spinlocks están directamente vinculados con los 32 spinlocks de hardware que existen dentro del bloque SIO del subsistema de procesamiento.

Un spinlock se usa para el acceso exclusivo a una sección crítica que protege: cuando un hilo, ejecutándose en un núcleo, “reclama” un spinlock sin bloquear, entonces lo bloquea, cambiando el valor de una variable de control y pasa a la sección crítica, cambiando el valor de la variable al salir de esta. Si un segundo hilo intentase “reclamar” el spinlock estando ya bloqueado, no podría, y pasaría a hacer una espera activa, “girando” o “haciendo spin” en un bucle esperando a que el primer hilo saliese, cambiando el valor de la variable a su valor original, indicando que la sección crítica vuelve a estar libre.

Debido a que gastan tiempo de ejecución del core bloqueado y que se desactivan las interrupciones durante la sección crítica, estas secciones críticas protegidas por spinlocks deberían ser pequeñas y no muy extensas.

Los spinlocks de hardware son preferibles a los exclusivos de software porque tienen menor overhead y latencia, y manejan mejor las contenciones entre núcleos. El RP2040 dispone de 32 spinlocks de hardware. Algunos están reservados para el sistema y la placa, y otros están disponibles para el usuario, según su identificador:

Los 14 primeros, con identificadores 0-13, son de uso exclusivo por el SDK y otras librerías internas. Estos NO se deben usar en ninguna circunstancia.

El 14 y el 15 son de uso exclusivo para la capa inmediatamente superior a la librería: un sistema operativo. Estos se pueden usar para operaciones atómicas de M2OS, pero no deberían ser usados por el usuario.

El resto, del 16-31 son utilizable por el usuario: hasta el 23 son de tipo “stripped”, que son reclamables en modo “round-robin” para ser accedidos de manera múltiple para ciertos tipos de implementaciones, y a partir del 23, los reclamables de manera exclusiva.

En el wrapper ***m2_multicore.c,*** nos fijamos en las funciones que reclaman spinlocks de manera exclusiva (es decir, solo un thread puede acceder a ellos a la vez) mediante ***m2_claim_unused_spinlock,*** que reclama el primer spinlock exclusivo que encuentre, y ***m2_unclaim_used_spinlock,*** que declama el spinlock con el ID introducido.

Una vez tengamos el ID del spinlock reclamado, también necesitamos guardar su instancia, que es la que se utiliza para bloquearlo y desbloquearlo, en un puntero con ***m2_get_spinlock_instance_from_id***. También se puede reclamar directamente un spinlock y recoger solo su instancia con ***m2_claim_and_get_spin_lock.***

Una vez tengamos la instancia (que debe ser recibida por todos los hilos que lo usen) solo es cuestión de bloquearlo y desbloquearlo: ***m2_lock_spinlock*** hace una llamada bloqueante al spinlock. Otros tipos de llamadas son el intento de bloqueo con timeout en el que solo se queda “girando” por un tiempo antes de “rendirse” y seguir con su ejecución.

m2_unlock_held_spinlock desbloquea un spinlock que el núcleo llamante esté bloqueando.

Adicionalmente, para usar estas llamadas, hay que tener en cuenta el estado de las interrupciones antes de bloquear el spinlock, ya que el picoSDK se encarga de desactivarlas durante la sección crítica. Al llamar a un método de bloqueo, este devuelve en forma de un entero sin signo de 32 bits, el estado de las interrupciones.

Este número de 32 bits debe de ser guardado por el usuario y suplido a la función de desbloqueo para que las interrupciones vuelvan a su estado original y no ocurran problemas.

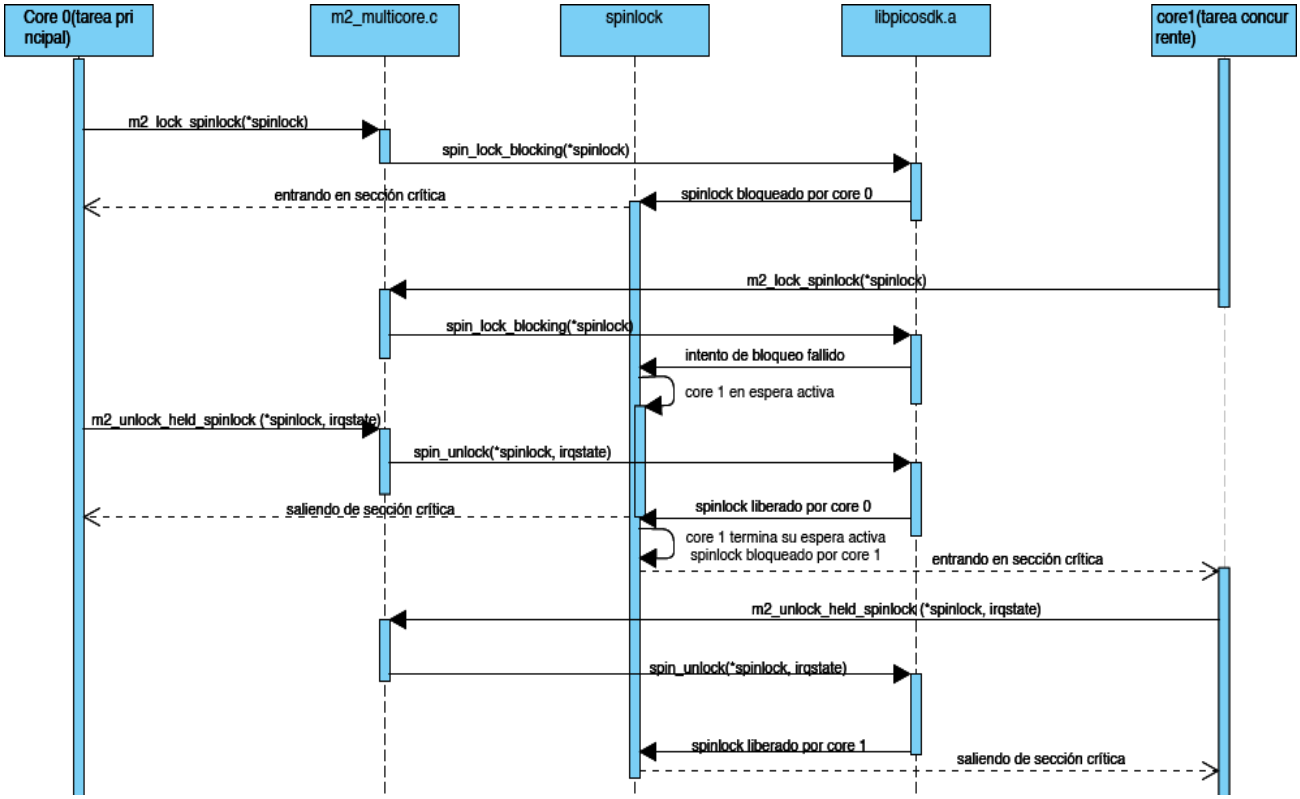


FIG. 38 – DIAGRAMA DE SECUENCIA QUE MUESTRA EL USO BÁSICO DE UN SPINLOCK COMO MÉTODO DE CONTROL DE EXCLUSIÓN MUTUA. EL EFECTO CON EL ORDEN DE BLOQUEO CONTRARIO SERÍA EL MISMO, PERO CON EL CORE 0 ESPERANDO.

Nota: otras primitivas de sincronización.

Gracias a los spinlocks de hardware, picoSDK tiene también otras primitivas de sincronización más complejas como mutexes, semáforos y secciones críticas complejas que también pueden implementarse a futuro ya que quedan fuera del alcance de este proyecto.

6. Tests y demostración

6.1 - Tests de API y POSIX.

Los tests presentes en .../***M2OS/tests/api_m2os*** y .../***M2OS/tests/posix*** ayudan en la tarea de verificar si el timer del sistema funciona correctamente. Uno de los objetivos del desarrollo de este proyecto es completar dichos test, que prueban un conjunto mínimo básico de tareas que M2OS ha de realizar para poder usarse para su propósito como sistema operativo de tiempo real, además de usar spinlocks en algunos de los test.

Los test en ***/M2OS/tests/posix*** verifican el uso de threads de POSIX, incluido threads periódicos. Esta sección prueba que la aplicación de los hilos del modelo POSIX es correcta. De este conjunto, se pasan todos los test.

Los test en ***/M2OS/tests/api_m2os*** prueban la API prueban diferentes tipos de tareas comunes para el sistema operativo, a través de la API del kernel. Pasan todos salvo uno, ***timing_events_and_tasks_test,*** en el que salta un "known bug". En general, esto muestra que la implementación del timer funciona y la inicialización es correcta, así como la implementación del spinlock.

Los programas en ***/M2OS/examples/*** son ejemplos con los que se puede probar, de manera manual ciertas partes del sistema. Por ejemplo, la prueba ***rp2040_easy_spinlock*** en ***/M2OS/examples/api_m2os*** usa un spinlock en el programa con el que se puede comprobar la atomicidad de las operaciones usando spinlocks.

6.2 - Demostración sencilla: control de periféricos dividido por núcleos.

La demostración final se encuentra en ../M2OS/examples/demo/demos.gpr bajo los nombres ***`multicoredemo_task.adb`***, ***`multicore_demo.adb`***, y ***`pmulticore_demo.ads`***. El resto de los archivos prueban los periféricos usados en la demo final de manera separada.

En esta demostración, se ponen a trabajar los dos cores para controlar, de manera sincronizada pero concurrente, el led interno de la placa, un LED RGB, un microservo controlado por pasos y un potenciómetro.

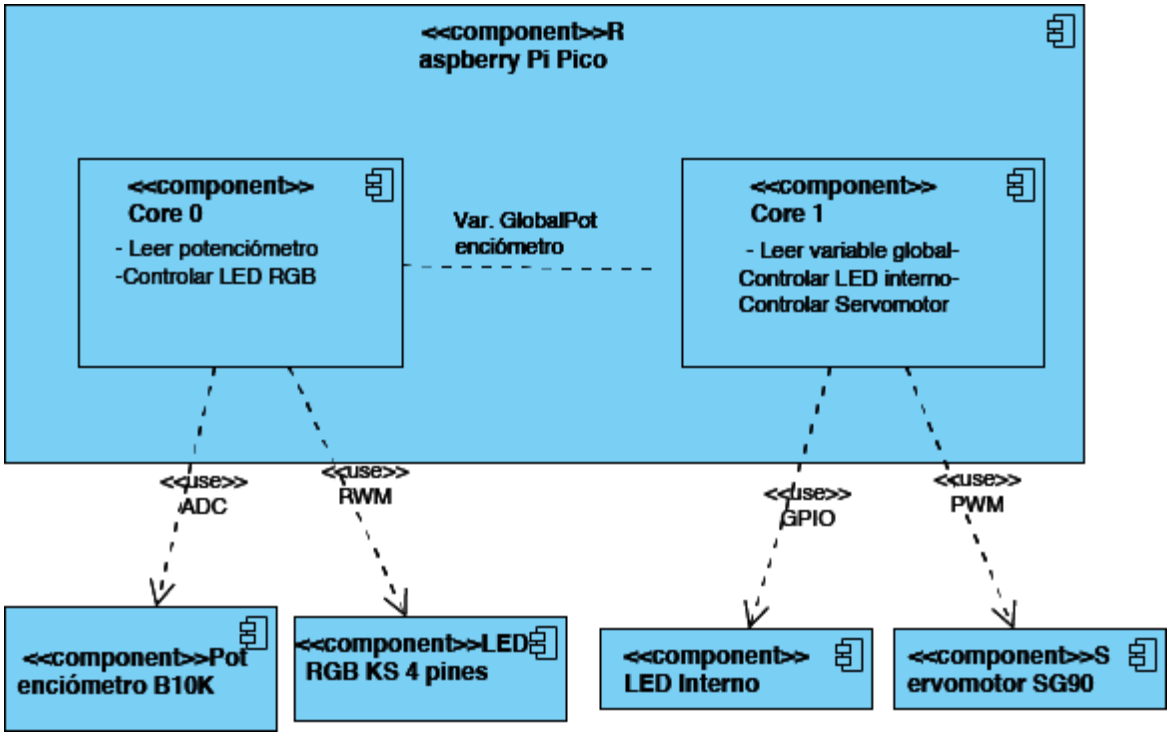


FIG. 39 – DIAGRAMA DE COMPONENTES BÁSICO DEL EJEMPLO.

Siguiendo los pasos del apartado 3, se puede lanzar en una raspberry pi pico que esté equipada con los periféricos necesarios: una gravity board, una picoprobe, un led RGB de cuatro pines, un servomotor de pulso, como el SG90, y un potenciómetro de tres pines.

multicoredemo_task.adb es el punto de entrada para M2OS declarada como una tarea de tipo one-shot con las políticas de planificación requeridas por el sistema operativo. Es la unidad de ejecución referenciada por el planificador de M2OS.

Esta tarea salta directamente a ***multicore_demo,*** que tiene un archivo de especificación (ads) con todas las importaciones y variables necesarias para su funcionamiento, y un archivo de cuerpo de paquete (adb) donde se alojan las funciones a ejecutar.

Los pines a los que conectar cada periférico se pueden ver en el archivo de especificación y cambiar acorde, teniendo en cuenta que no todos los pines sirven para todas las funciones, junto con otras variables y los métodos de control de los periféricos importados, para no tener que “gastar espacio” en el archivo del cuerpo del programa: ADC, PWM y GPIO.

Por ejemplo, la variable ***POTENTIOMETER_PIN*** está ajustada al valor 27. Esto significa que, por ejemplo, en la gravity board, los tres pines del potenciómetro tendrán que ir conectados a los pines GP27 del bloque analógico de la gravity board. El potenciómetro solo puede funcionar en los pines GP26, GP27 y GP28 ya que son los únicos equipados con el convertidor analógico-digital necesario para leer el potenciómetro.

Para el pin RGB, ya que se usan directamente los pines digitales, ***RED_LED_PIN, GREEN_LED_PIN***, ***BLUE_LED_PIN***, la única limitación es en la distribución física de los pines, que van juntos, incluido el cable de tierra. Además, se crean variables para guardar el canal y la “slice” de cada color, necesarias para el control mediante PWM:

En la Raspberry Pi Pico, todos los pines tienen capacidad PWM y tiene 8 generadores de PWM; que se denominan canales. Cada canal tiene dos “tiras” o “slices” usables. Cada grupo

de pines GPIO está vinculado a un generador PWM distinto. Y cada pin puede acceder independientemente a las dos tiras de dicho generador.

El “pin” correspondiente al LED interno de la placa siempre es el mismo, el 25, guardado en ***PICO_DEFAULT_LED_PIN.***

SERVO_PIN indica el pin para el servo que ha de ir con sus tres pines conectados al bloque digital de la gravity board, en este caso a la línea 18. En este, además de crear el canal y el “slice”, se tienen que calcular los valores para ciertas posiciones del servo: ***Left_Duty***, ***Center_Duty***, ***Right_Duty***. Estas son dependientes del valor de envoltura (***ServoWrap***) y la frecuencia (***ServoFrequency***), parámetros que se usan para, dado el ciclo de trabajo (duty cicle) del motor, enviar los pulsos correctos para poder girar el motor a una posición correcta.

Cada generador iniciará un contador, desde 0 hasta el valor de envoltura. Dependiendo del valor de porcentaje de canal, más o menos parte de este recorrido tendrá la señal a un nivel o a otro. Por ejemplo, con un valor de envoltura de 100 y un valor de porcentaje de canal de 50, la mitad del pulso (es decir, de 0 a 49) tendrá la señal en valor alto y la otra mitad (de 0 a 99) tendrá la señal en valor bajo. El valor de envoltura puede tener cualquier valor de 1 a 65565 y el valor de porcentaje de canal puede tener valores de 25, 50 y 75, siendo 50 el valor por defecto.
La rapidez de aumento en el contador, es decir, lo rápido o despacio que cambiará la anchura del ciclo, viene dado por la frecuencia del reloj del que los generadores PWM toman señal: el PLL del sistema. Como esta frecuencia es muy grande para la mayoría de las aplicaciones, se puede aplicar un divisor de frecuencia para que el generador PWM tome una división entera de la frecuencia total.

En el cuerpo del programa, el método de entrada es el método de inicialización ***Multicore_Task_Init.*** Dentro de este método se preparan los pines para cada función; GPIO para el LED interno, PWM para el motor y el LED RGB, ADC para el potenciómetro y se lanzará la función pertinente, `Core1_Task_Body`, al core 1. Al terminar esta inicialización, se saltará directamente a la siguiente función, en este caso `Core0_Task_Body`, a seguir ejecutando en el core 0.

Ahora tenemos ***Core0_Task_Body*** ejecutando en el core 0 y ***Core1_Task_Body*** ejecutando en el core 1 tras haber inicializado correctamente todos los periféricos.

En el core 0, se leerá el valor del potenciómetro gracias a los convertidores analógico-digitales de la placa. Este valor se asignará a una variable que será compartida con el método que se ejecuta paralelamente en el core 1. Además, este valor se utilizará para computar el valor de **la intensidad** del LED RGB, que hará un ciclo por varios colores, mezcla de uno o más componentes de color de dicho LED: cuanto más bajo sea el valor del potenciómetro (más “cerrada” esté la rueda de este), más tenue será la luz y viceversa.

Para escribir el valor en la variable del potenciómetro, se usa un spinlock. Además, para la salida de DEBUG en pantalla (texto mediante Direct I/O de Ada) se usa otro spinlock para evitar que las entradas de ambos núcleos se “solapen” produciendo texto ilegible en la pantalla.

El core 1 se encargará de controlar el LED interno de la placa y el motor. Para que se vea que ambos cores trabajan de manera independiente, el valor del potenciómetro, leído siempre desde la variable compartida, se computa al contrario que en el core 0: cuanto más bajo sea el valor, más rápido será tanto el parpadeo del LED de la placa como el giro del servomotor en sus tres posiciones y, al contrario, cuanto más alto el valor, más despacio irán ambas cosas.

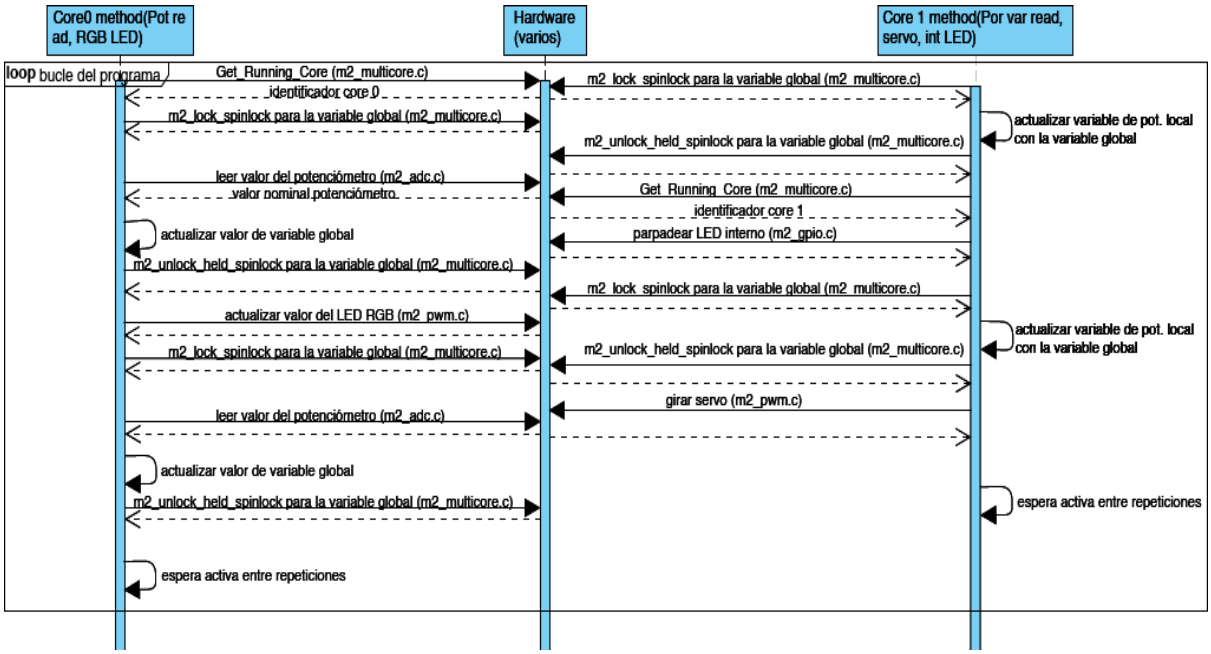


FIG. 40 – DIAGRAMA SIMPLIFICADO DE LA DEMO. EN ESTE EJEMPLO SE OBVIAN LAS LLAMADAS INTERMEDIAS Y LOS BLOQUEOS DEL SPINLOCK YA EXPLICADOS EN APARTADOS ANTERIORES.

Este ejemplo demuestra eficientemente todo lo desarrollado: el timer del sistema, en el que se basan las esperas activas entre acciones de los núcleos, utilizando la función que devuelve el tiempo del sistema para hacer pasar el tiempo, los spinlocks, debido a que no hay fallos de lectura/escritura en la variable compartida ni texto solapado en la salida, y el uso correcto de ambos núcleos, especialmente ya que se usa la función que devuelve el identificador del núcleo y se puede ver que en cada método devuelve el respectivo identificador, además de verse claramente que los dos grupos de periféricos trabajan de manera independiente e incluso de manera concurrente unos con otros, cosa que no podría ocurrir si el código fuera completamente secuencial.

7. Conclusión y trabajos futuros.

En conclusión, este trabajo demuestra la correcta y eficiente implementación de un conjunto de tareas básicas de M2OS en la placa Raspberry Pi Pico usando el microprocesador RP2040.

Para conseguirlo, ha sido necesario entender parte del hardware de la placa, así como los varios componentes de software que trabajan juntos para ejecutar tareas de Ada. Esto incluye:

- Integración de PicoSDK en el proyecto: Cómo se generan los archivos necesarios para el control de hardware y como juntarlos en una librería usable por M2OS
- Desarrollo de M2OS para el control de hardware de la placa, que incluye:
    - Adaptar la compilación de M2OS para usar la librería creada mediante PicoSDK
    - Crear wrappers que faciliten el acceder a funciones de la librería desde M2OS
    - Crear un timer de sistema para las tareas periódicas del kernel de M2OS
    - Controlar el segundo núcleo, lanzar tareas y comunicarse con el primer núcleo.
    - Crear una primitiva de sincronización básica para usar en tareas concurrentes.
    - Control de periféricos mediante M2OS

Además, se ha conseguido crear un marco para la depuración de M2OS y sus tareas en el chip y se ha modificado la compilación del sistema operativo para añadir opciones que facilitan su ajuste en el chip RP2040.

Otros trabajos podrían centrarse en expandir los wrappers de entrada/salida e implementar correctamente la sincronización inter-núcleo de hardware, lo que dotaría al sistema incluso de más flexibilidad que actualmente.

Podrían adaptarse las primitivas de sincronización más complejas de PicoSDK a M2OS, como mutexes y semáforos, para llevar a cabo otras tareas de tiempo real bajo M2OS, o incluso optimizar partes del sistema operativo, para que trabajen más estrechamente con la librería.

También, optimizar el sistema de compilación de M2OS para facilitar la preparación del entorno de desarrollo, como por ejemplo a la hora de añadir o retirar las flags de compilación del entorno de Ada o buscar una manera de integrar los archivos objeto del picoSDK en el proceso de compilación de GNAT, para no tener que ejecutar scripts externos y facilitar la preparación del entorno.

Todas estas mejoras facilitarían el crear tareas más complejas para M2OS en el RP2040, ayudando a su desarrollo y explotación en diferentes entornos y placas en las que el microprocesador forme parte.

Bibliografía

1. M2OS, "RTOS with simple tasking support for small microcontrollers," [Online]: https://m2os.unican.es.
2. M2OS, "M2OS characteristics," [Online]: https://m2os.unican.es/characteristics/.
3. Raspberry Pi Foundation, "Meet the Raspberry Pi Pico," [Online]: https://projects.raspberrypi.org/en/projects/getting-started-with-the-pico/.
4. Raspberry Pi Foundation, "Rp2040 specifications," [Online]: https://www.raspberrypi.com/products/rp2040/specifications/.
5. Raspberry Pi Foundation, "RP2040 Product Brief," [Online]: https://datasheets.raspberrypi.com/rp2040/rp2040-product-brief.pdf.
6. Raspberry Pi Foundation, "About the Raspberry Pi Pico pinout," [Online]: https://www.raspberrypi.com/documentation/microcontrollers/raspberry-pi-pico.html.
7. Raspberry Pi Foundation, "Raspberry Pi Pico Processor subsystem," [Online]: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf#page=28.
8. Raspberry Pi Foundation, "Interrupt architecture," [Online]: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf#page=61.
9. Raspberry Pi Foundation, "Serial Wire Debug line," [Online]:
10. Raspberry Pi Foundation, "Inter-core mailboxes," [Online]: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf#page=31.
11. Raspberry Pi Foundation, "Raspberry Pi Debug Probe," [Online]: https://www.raspberrypi.com/products/debug-probe/.
12. AdaCore, "About Ada," [Online]. Available: https://www.adacore.com/about-ada.
13. AdaCore, "GNATStudio," [Online]: https://www.adacore.com/gnatpro/toolsuite/gnatstudio.
14. Raspberry Pi Foundation, "An introduction to PicoSDK," [Online]: https://www.raspberrypi.com/documentation/pico-sdk/index_doxygen.html.
15. Raspberry Pi Foundation, "PicoSDK repository," [Online]: https://github.com/raspberrypi/pico-sdk.
16. NullCharx, "Pico port repository," [Online]: https://github.com/NullCharx/RPIPICO-M2OSPORT-TFG.
17. M. Aldea, "M2OS Repository," [Online]: https://gitlab.com/marioaldea/M2OS/-/tree/rp2040.
18. wokwi, "rp2040.js," [Online]: https://github.com/wokwi/rp2040js.
19. majbthrd, "pico-debug," [Online]: https://github.com/majbthrd/pico-debug.
20. Mario Aldea Rivas and Héctor Pérez Tijero. “Leveraging real-time and multitasking Ada capabilities to small microcontrollers. Journal of Systems Architecture", vol. 94, pp. 32-41, March 2019.
21. Ada Reference Manual, Language and Standard Libraries. International Standard ISO/IEC 8652:2023