Universidad de Cantabria

Facultad
de
ciencias

LIBRERÍA DE SOPORTE DE TAREAS ADA PARA MARTE OS

(MaRTE OS Ada task support library)

Trabajo de Fin de Grado
para acceder al
GRADO EN INGENIERÍA INFORMÁTICA

Autor: Ismael Escalada Diego

Tutor: Mario Aldea

Co-Tutor: Héctor Pérez

Agosto – 2025

Índice general

Glosario

`null` Valor especial que indica la ausencia de un valor o puntero válido. 1, 23

AdaCore Empresa que desarrolla y mantiene herramientas y *runtimes* para el lenguaje Ada (por ejemplo, el compilador *GNAT*) y ofrece soporte profesional, documentación y utilidades para el desarrollo de software seguro y de tiempo real. 1, 18, 33, 35

ARM Una arquitectura de conjunto de instrucciones (ISA) basada en el diseño RISC (Reduced Instruction Set Computer). Fue desarrollada originalmente por Acorn Computers en la década de 1980 y actualmente es ampliamente utilizada en sistemas embebidos, dispositivos móviles y plataformas de bajo consumo energético debido a su eficiencia y bajo coste.. 1, 5, 6, 8, 9, 11, 12, 14, 20, 32, 35

búfer Área de memoria utilizada temporalmente para almacenar datos durante operaciones de entrada/salida.. 1

delay Función para suspender la ejecución de tareas por un intervalo de tiempo. 1, 14

GDB (GNU Debugger) Herramienta de depuración que permite a los programadores examinar y controlar la ejecución de un programa. 1, 4, 11, 25, 26

GNARL Acrónimo de *GNAT Run-Time Library*. Es la librería en tiempo de ejecución del compilador GNAT, escrita en Ada. GNARL proporciona las primitivas necesarias para soportar la concurrencia, el manejo de tareas y la sincronización, elementos fundamentales del lenguaje Ada en sistemas embebidos y de misión crítica. 1, 5, 6, 8, 12, 20, 21

GNAT Es el compilador libre de Ada desarrollado inicialmente por el proyecto GNU y actualmente mantenido por AdaCore. GNAT forma parte de la cadena de herramientas GNU y permite compilar, depurar y ejecutar programas en Ada. Además, incluye utilidades adicionales como `gnatmake`, `gnatbind` y `gnatlink`, que facilitan la gestión completa del proceso de construcción. 1, 5, 8, 12, 20, 21, 24, 35

GPIO General Purpose Input/Output. Se refiere a los pines digitales programables en un microcontrolador o procesador, utilizados para recibir o enviar señales eléctricas. 1, 25, 33

IDE Entorno de Desarrollo Integrado, software que agrupa herramientas para facilitar la programación, como editor, compilador y depurador. 1, 12, 35

MaRTE OS Sistema operativo embebido para aplicaciones en tiempo real. 1, 8, 12, 13, 16, 18–20, 22, 28, 30, 32, 35

periférico Dispositivo de hardware que se conecta a un dispositivo para ampliar su funcionalidad. Ejemplos de periféricos: impresoras, teclados, ratones, escáneres, entre otros.. 1, 13, 28, 30

POSIX POSIX (acrónimo de Portable Operating System Interface) es un conjunto de estándares definidos por IEEE (estándar IEEE 1003), cuyo objetivo es garantizar portabilidad del software entre sistemas operativos compatibles.. 1, 8, 12, 16, 17

QEMU (siglas de Quick Emulator) emulador de máquinas y virtualizador. 1, 11, 25, 36

rendezvous En el lenguaje de programación **Ada**, un *rendezvous* es el mecanismo mediante el cual una tarea cliente sincroniza y se comunica con una tarea servidora. El cliente realiza una llamada a un procedimiento protegido denominado *entry*, mientras que el servidor debe estar preparado para aceptar dicha llamada. El encuentro (o "rendezvous") ocurre cuando ambas partes están listas, permitiendo así la sincronización y el intercambio de datos de manera segura y controlada. Este concepto es fundamental en el modelo de concurrencia de Ada. . 1, 8, 10

RISC-V Una arquitectura de conjunto de instrucciones (ISA) abierta y libre, basada en el diseño RISC (Reduced Instruction Set Computer). Fue desarrollada inicialmente en la Universidad de California, Berkeley, con el objetivo de ser una plataforma flexible, extensible y adecuada tanto para investigación como para aplicaciones comerciales.. 1, 33

RTS Runtime System, el sistema en tiempo de ejecución para Ada. 1, 7, 8, 11–14, 16, 18–20, 22, 32, 35, 43

RunTime System Conjunto de componentes y servicios que proporciona un entorno de ejecución para un programa, gestionando aspectos como la memoria, llamadas al sistema operativo, planificación de tareas y otras funcionalidades necesarias durante la ejecución del código [1].. 1, 8, 11, 14, 16, 18–22, 28, 32, 33

semihosting Mecanismo que permite a un programa en ejecución en un microcontrolador comunicarse con el entorno de desarrollo (host) para operaciones de entrada/salida, como leer/escribir archivos o imprimir mensajes de depuración. Generalmente se realiza a través de GDB, sin necesidad de un sistema operativo completo en el dispositivo embebido.. 1, 9, 21, 25, 26

STM32F4 Familia de microcontroladores de 32 bits basados en ARM Cortex-M4.. 1, 5–9, 19, 28, 32

wrapper Patrón o estructura de programación que actúa como envoltorio de otra función, clase o módulo, con el fin de modificar o extender su comportamiento sin alterar su implementación original. 1, 9, 25–27, 31, 37, 40

Resumen/Abstract

La programación concurrente en sistemas embebidos de tiempo real es fundamental para cumplir restricciones de temporalidad. Ada [2], un lenguaje de programación orientado a sistemas críticos, ofrece un constructo de alto nivel denominado *tarea* (*task*), que representa un hilo independiente de ejecución. De este modo, Ada facilita la implementación de aplicaciones paralelas seguras y deterministas.

MaRTE OS (Minimal Real Time Operating System), desarrollado por el Grupo de Ingeniería de Software y Tiempo Real de la Universidad de Cantabria, es un sistema operativo de tiempo real diseñado para aplicaciones embebidas, basado en el subconjunto POSIX.13 de tiempo real mínimo. Está mayormente programado en Ada, con partes en C y ensamblador. Permite el desarrollo cruzado de software en Ada y C utilizando los compiladores GNU GNAT y GCC. También es compatible con depuración remota a través de gdb. La biblioteca de tiempo de ejecución de Gnat (GNARL) ha sido adaptada para funcionar con este kernel.[3]

El objetivo principal de este Trabajo Fin de Grado es el diseño e implementación de una *librería de soporte de tareas* en el lenguaje de programación Ada, orientada al sistema operativo MaRTE OS. Esta librería proporcionará un conjunto de primitivas de concurrencia —incluyendo la creación y gestión de tareas, mecanismos de sincronización mediante semáforos u objetos protegidos, y herramientas de comunicación entre tareas— con el fin de facilitar el desarrollo de aplicaciones paralelas de forma sencilla, robusta y eficiente.

La implementación estará específicamente dirigida a plataformas embebidas basadas en la arquitectura ARM, utilizando como entorno de pruebas la plataforma STM32F4. De este modo, se busca extender las capacidades de MaRTE OS para soportar de manera efectiva la ejecución de aplicaciones concurrentes en sistemas en tiempo real sobre arquitecturas ARM, ampliamente utilizadas en el ámbito de la computación embebida.

Abstract

Concurrent programming in real-time embedded systems is essential to meet timing constraints. Ada [2], a programming language designed for safety-critical systems, provides a high-level construct called *task*, which represents an independent thread of execution. This feature allows Ada to support the development of parallel, safe, and deterministic applications.

MaRTE OS (Minimal Real Time Operating System), developed by the Software and Real-Time Engineering Group at the University of Cantabria, is a real-time operating system for embedded applications, based on the POSIX.13 minimal real-time subset. It is mainly written in Ada, with some components in C and assembly. It supports cross-development in Ada and C using the GNU GNAT and GCC compilers, and provides

remote debugging through gdb. The Gnat runtime library (GNARL) has been adapted to run on this kernel.[3]

The main objective of this Final Degree Project is the design and implementation of a *task support library* in the Ada programming language, specifically targeted at the MaRTE OS operating system. This library will provide a set of concurrency primitives —including task creation and management, synchronization mechanisms through semaphores or protected objects, and inter-task communication tools— to simplify the development of parallel applications in a robust and efficient way.

The implementation will be focused on embedded platforms based on the ARM architecture, using the STM32F4 platform as a testing environment. In this way, the goal is to extend the capabilities of MaRTE OS to effectively support the execution of concurrent applications in real-time systems running on ARM architectures, which are widely used in the field of embedded computing.

Capítulo 1

Introducción

El desarrollo de software para sistemas embebidos de tiempo real constituye un área de gran relevancia dentro de la ingeniería informática, dado que este tipo de sistemas se caracteriza por la necesidad de cumplir estrictos requisitos temporales y de fiabilidad. En este contexto, resulta imprescindible disponer de entornos de ejecución que garanticen un comportamiento determinista, eficiente y seguro.

El lenguaje de programación Ada, ampliamente empleado en aplicaciones críticas como la aeronáutica, la industria ferroviaria o la defensa, ofrece un conjunto de características orientadas a la robustez y la seguridad, entre las que destacan el fuerte tipado, el manejo estructurado de excepciones y, especialmente, el soporte nativo para concurrencia mediante tareas y mecanismos de sincronización. La correcta ejecución de programas escritos en Ada requiere del uso de un RunTime System (RTS), un conjunto de bibliotecas y servicios que proporcionan las funciones esenciales para el funcionamiento del lenguaje sobre una plataforma concreta. El STM32F4 se encarga de gestionar aspectos como la inicialización del programa, la planificación y coordinación de tareas, la comunicación entre procesos y la interacción con el sistema operativo subyacente, actuando como puente entre el código generado por el compilador y el hardware de destino. AdaCore clasifica y define los RTS de la siguiente forma:

- **Standard Run-Time**: Sistema de ejecución completo de GNAT Pro, destinado a plataformas con sistema operativo.
- **Embedded Run-Time**: Subconjunto del Standard para sistemas bare-metal o RTOS limitados.
- **Light Run-Time**: Orientado a sistemas embebidos pequeños y certificables.
- **Light-Tasking Run-Time**: Variante del Light con soporte de multitarea, mediante los perfiles *Jorvik* (por defecto) o *Ravenscar* (activable)
- **Configurable Run-Time Facility**: Permite definir un RTS reducido o adaptado a necesidades específicas.

Para más información, consúltese la sección 4.2, *Tipos de librerías para RunTime Systems*

En este marco, MaRTE OS (Minimal Real-Time Operating System), desarrollado por el Grupo de Ingeniería de Software y Tiempo Real de la Universidad de Cantabria, constituye un sistema operativo de tiempo real minimalista, basado en el perfil POSIX.13. Está orientado a aplicaciones embebidas que requieren predictibilidad temporal, ofreciendo un soporte ligero y flexible que lo convierte en un candidato idóneo para la integración con Ada y su RTS.

La plataforma hardware seleccionada para este trabajo es la STM32F4, basada en la arquitectura ARM Cortex-M4, ampliamente extendida en el ámbito de los sistemas embebidos por su eficiencia energética y capacidad de procesamiento. La combinación de MaRTE OS, Ada y ARM permite explorar una solución completa para el desarrollo de aplicaciones concurrentes en entornos de tiempo real.

GNAT es el compilador libre y de código abierto para el lenguaje Ada, inicialmente desarrollado en la Universidad de Nueva York en colaboración con el proyecto GNU y actualmente mantenido por la empresa AdaCore. Forma parte de la cadena de herramientas GNU, lo que le permite integrarse de manera natural con otros compiladores como GCC, y proporciona un conjunto completo de utilidades que facilitan la construcción, depuración y despliegue de aplicaciones en Ada.

El compilador GNAT no solo traduce el código fuente escrito en Ada a código ejecutable, sino que también incluye herramientas auxiliares como gnatmake (gestión de dependencias y compilación automática), gnatbind (vinculación de unidades Ada y preparación del entorno de ejecución) y gnatlink (enlazado final del ejecutable con las librerías del RTS). Estas utilidades permiten gestionar proyectos complejos, automatizando tareas que en otros compiladores requieren una mayor intervención manual.

GNARL (GNAT Run-Time Library) es la librería de ejecución asociada al compilador GNAT y constituye una parte esencial del *Run-Time System (RTS)* de Ada. Su propósito principal es proporcionar la implementación de las primitivas que permiten dar soporte a las características del lenguaje relacionadas con la concurrencia, la planificación de tareas, la sincronización y el manejo de excepciones.

Mientras que GNAT traduce el código fuente Ada a instrucciones ejecutables, GNARL ofrece el conjunto de servicios necesarios en tiempo de ejecución para que dicho código se ejecute conforme al modelo del lenguaje. Entre estos servicios se incluyen:

- **Gestión de tareas (tasking)**: creación, inicialización, suspensión y finalización de tareas concurrentes.
- **Sincronización**: mecanismos como *rendezvous*, objetos protegidos y semáforos que garantizan exclusión mutua y comunicación segura entre tareas.
- **Planificación**: adaptación de las políticas de planificación a los sistemas subyacentes (FIFO, Round Robin, EDF, prioridades dinámicas).
- **Interacción con el sistema operativo**: delegación de operaciones de bajo nivel, como la creación de hilos POSIX, la gestión de temporizadores o el control de interrupciones.
- **Soporte de tiempo real**: provisión de mecanismos deterministas que cumplen los requisitos temporales definidos en el estándar Ada.

En el caso de MaRTE OS, se dispone de una adaptación de un RTS "Standard", aunque se trata de una versión antigua y limitada exclusivamente a la arquitectura `x86`. En cambio, para la arquitectura `ARM` únicamente se ha desarrollado un RTS "light", el cual carece de soporte para la gestión de tareas, lo que restringe sus capacidades en comparación con la versión para `x86`.

Objetivos del Trabajo:

El presente Trabajo de Fin de Grado tiene como objetivo principal el diseño e implementación de una librería de soporte de tareas en Ada para el sistema operativo MaRTE OS para microcontroladores ARM, lo que implica portar y adaptar un RunTime System .^estándar"de GNAT a la plataforma seleccionada MaRTE OS. Para alcanzar este fin se plantean los siguientes objetivos específicos:

- Portado del RTS de Ada a MaRTE OS, garantizando la correcta integración del entorno de ejecución con los servicios del kernel.
- Implementación de mecanismos de concurrencia y sincronización, incluyendo la gestión de tareas, el soporte de objetos protegidos y la provisión de funciones de temporización (delay relativo y delay absoluto).
- Adaptación de funciones de entrada/salida, mediante la definición de primitivas faltantes y el desarrollo de un wrapper que permita utilizar de forma transparente tanto el semihosting como la comunicación por UART en la placa STM32F4.
- Validación funcional del sistema, mediante pruebas en entornos de emulación y sobre el hardware real, evaluando la capacidad de ejecución de aplicaciones concurrentes en Ada.
- Diseño modular y portable, que facilite la mantenibilidad del código y su futura extensión hacia nuevas arquitecturas, placas de desarrollo u otras versiones de MaRTE OS.

La consecución de estos objetivos permitirá dotar a MaRTE OS de una librería de Ada actualizada, capaz de soportar aplicaciones concurrentes en plataformas ARM, contribuyendo así al avance en el desarrollo para sistemas embebidos.

Capítulo 2

Tecnologías y herramientas utilizadas

Para desarrollar la librería es necesario contar con un conjunto de tecnologías que permitan no solo escribir el código, sino también compilarlo, probarlo y depurarlo en un entorno similar al del hardware final. Cada herramienta seleccionada cumple un papel específico dentro de este proceso: desde el lenguaje de programación con el que se construye la solución, hasta los compiladores, emuladores y depuradores que facilitan su validación y puesta a punto. En las siguientes secciones se describen las principales tecnologías que se emplearán y la función que desempeñan dentro del proyecto.

- Lenguaje Ada:

  Ada es un lenguaje de programación de propósito general, fuertemente tipado, diseñado con un enfoque en la seguridad, la confiabilidad y la mantenibilidad del software. Fue desarrollado por un equipo liderado por Jean Ichbiah por encargo del Departamento de Defensa de los Estados Unidos, y su primera versión fue estandarizada en 1983 [4]. Ada incluye soporte nativo para concurrencia a través de tareas, sincronización mediante rendezvous. soportando mecanismos de sincronización entre tareas, y manejo robusto de excepciones, lo que lo hace especialmente adecuado para sistemas embebidos y de misión crítica. Ha sido ampliamente utilizado en sectores donde la precisión y la tolerancia a fallos son esenciales, como la aviación, los sistemas militares, el control del tráfico aéreo, la industria ferroviaria y la exploración espacial. Además, su sintaxis clara y su fuerte tipado facilitan la detección temprana de errores, lo que contribuye a la creación de software confiable y seguro.

  Todo el código del Runtime System (RTS) —véase la sección 4.2, «Diseño del sistema: Tipos de librerías para Runtime Systems»— de Ada está programado íntegramente en el propio lenguaje Ada, lo que refuerza su coherencia interna y facilita el mantenimiento y la evolución del sistema. Esta característica es especialmente relevante en el contexto de los sistemas críticos, ya que permite aplicar las mismas garantías de seguridad, robustez y fiabilidad del lenguaje al núcleo de su propia ejecución.

- GPRBuild:

  GPRbuild, desarrollado por AdaCore, es una herramienta de construcción multilenguaje. Está diseñada para compilar proyectos grandes que integren Ada, C, C++, Fortran, etc. Usa un archivo de proyecto (.gpr) para analizar dependencias y automatizar las fases de compilación, vinculación y post-compilación, compilando solo lo necesario.

  Esta será la herramienta principal encargada de compilar el código fuente escrito en Ada y de gestionar todo el proceso de construcción del proyecto de manera eficiente

y automatizada. GPRbuild se encarga de coordinar las distintas etapas de compilación, análisis de dependencias, generación de objetos intermedios, vinculación de bibliotecas y ejecución de tareas posteriores a la compilación, como la instalación o la generación de documentación. Gracias a su integración con archivos de proyecto .gpr, permite una configuración precisa y flexible del entorno de construcción, lo que facilita el manejo de proyectos complejos con múltiples unidades, dependencias externas y configuraciones específicas.

- Compilador cruzado Arm:

  Un compilador cruzado es un tipo de compilador que genera código ejecutable para una arquitectura de hardware distinta de aquella en la que se ejecuta el propio compilador. Se utiliza cuando se necesita desarrollar software en una arquitectura, como x86, pero que debe ejecutarse en otra, como ARM. Esto permite crear programas para plataformas con diferentes conjuntos de instrucciones sin necesidad de compilar directamente en ellas, lo cual es especialmente útil cuando la arquitectura de destino no dispone de los recursos necesarios para compilar el código por sí misma.

  El RTS(RunTime System) se implementará sobre la plataforma STM32F4, la cual presenta la arquitectura ARM, por lo que de cara a la implementación durante el desarrollo utilizaremos un compilador cruzado para poder trabajar sobre nuestro equipo(que presenta una arquitectura x64)

- Qemu:

  QEMU [5] es un software libre y de código abierto que actúa como emulador de máquinas y virtualizador. Utiliza traducción dinámica de binarios para ejecutar código de una arquitectura en otra y puede funcionar en modo emulación de sistema completo (emulando CPU, memoria y dispositivos) o en modo de emulación de usuario (ejecutando programas de otra arquitectura)

  Gracias a QEMU se dispondrá de la capacidad de emular el hardware de la placa STM32F4 y verificar que funcione correctamente sin necesidad de utilizar el hardware real, acelerando el desarrollo del software y facilitando la realización de las pruebas de este. En nuestro caso, utilizaremos el comando Gnatemu en lugar de usar el programa QEMU de forma manual. Gnatemu se encarga de iniciar QEMU de manera automática y configurada para imitar el hardware de la placa STM32F4. De esta forma podemos probar y depurar el software como si estuviéramos trabajando directamente sobre la placa real, pero sin necesidad de contar físicamente con ella.

- GDB (GNU Debugger):

  GDB es un depurador desarrollado por el proyecto GNU en 1986 para ayudar a los programadores a encontrar y corregir errores en sus programas. Desde entonces, se ha convertido en una herramienta fundamental en el mundo del software libre, permitiendo ejecutar programas paso a paso, inspeccionar variables y controlar su flujo para facilitar el diagnóstico de fallos y mejorar la calidad del código.

  Se usará de cara a la resolución de problemas o errores que se detecten durante la ejecución de las pruebas del RTS. Permitiendo observar el avance de la ejecución verificando que esta se realice de la forma correcta y esperada.

- Git y Github:

  Git es un sistema de control de versiones distribuido que permite a los desarrolladores gestionar y registrar los cambios en el código fuente de sus proyectos de manera eficiente y segura. Facilita el trabajo colaborativo, permitiendo que varias personas trabajen simultáneamente sin perder el historial de modificaciones. GitHub es una

plataforma en línea que utiliza Git para alojar repositorios de código. Además de almacenamiento, ofrece herramientas para la colaboración, como control de versiones, gestión de incidencias, revisión de código y documentación, haciendo que el desarrollo en equipo sea más organizado y transparente.

La herramienta Git permitirá disponer de un control de versiones para el desarrollo, permitiendo explorar distintas rutas para la implementación mediante el uso de ramas. Además de mantener a salvo el trabajo de modificaciones que rompan el funcionamiento durante el proceso de adaptación del RTS. GitHub será de gran utilidad para almacenar el código fuera del equipo garantizando la disponibilidad de este. Se dispone del repositorio en el Anexo A.4.

- MaRTE OS:

  MaRTE OS (Minimal Real-Time Operating System for Embedded Applications) [3] es un kernel de tiempo real minimalista, diseñado principalmente en Ada (con componentes críticos en C y ensamblador), que implementa el perfil Minimal Real-Time POSIX.13 para equipos embebidos: ofrece servicios POSIX (por ejemplo, pthreads, mutexes, variables de condición, señales, relojes y temporizadores de CPU y de pared, suspensiones relativas y absolutas) con respuesta temporal acotada, todo ello ejecutándose en un espacio de direcciones único que comparten la aplicación multihilo y el kernel; fue desarrollado por el Grupo de Computación y Tiempo Real de la Universidad de Cantabria y se distribuye bajo la licencia GPL v2. Su entorno está basado en la herramienta GNU (GNAT para Ada y gcc para C), integrando una versión adaptada de la librería de ejecución GNARL para ejecutarse sobre el planificador de tareas de MaRTE, lo que permite el desarrollo cruzado de aplicaciones Ada-C y la depuración remota por gdb sobre conexión serie o Ethernet desde un host GNU/Linux hacia un PC objetivo, ya sea en modo bare-metal o como ejecutable Linux. Admite aplicaciones en Ada, C, C++, garantizando una planificación global coherente que integra tareas Ada y hilos POSIX en C, y soporta los mecanismos de tiempo real Ada, incluida la determinación dinámica de prioridades, sincronización por techo o herencia y políticas de planificación como EDF o Round-Robin.

  En el contexto del proyecto MaRTE OS será el sistema operativo base sobre el que se ejecutará el RTS, dado que al ser un sistema POSIX permite comunicarse de forma sencilla con el runtime.

- Placa de desarrollo STM32f4:

  La placa STM32F4(Figura 2.1) es una plataforma diseñada para poner sobre la mesa todas las capacidades de los microcontroladores STM32F4 de STMicroelectronics, centrada en un potente núcleo ARM Cortex-M4 de 32 bits, con unidad de coma flotante (FPU) y soporte de instrucciones DSP para procesamiento de datos en tiempo real. Funciona a una frecuencia de hasta 168 MHz y utiliza la arquitectura ARM para ofrecer eficiencia y velocidad. Incluye un programador/-depurador ST-LINK/V2-A integrado, lo que facilita la depuración directa desde STM32CubeIDE o IDEs comerciales, con ejemplos del paquete STM32CubeF4 o bibliotecas legacy como STSW-STM32068.

Esta es la placa objetivo sobre la que deberán ejecutar la combinación del RTS portado y MaRTE OS. Creando así un entorno de desarrollo que se beneficia de las entradas para periféricos que nos incluye la placa de desarrollo. Pudiendo así realizar sistemas de tiempo real basados en mediciones de sensores, por ejemplo. Ha sido seleccionada dada su versatilidad, dispositivos, soporte inicial.

Figura 2.1: Placa STM32F4

Capítulo 3

Especificación de requisitos

3.1. Requisitos Funcionales

- **RF1 - Portado del RTS de Ada a MaRTE OS:**
  El sistema debe portar correctamente el RTS (RunTime System)de Ada, debe adaptarse una de sus variantes(véase la sección 4.2, *Tipos de librerías para RunTime Systems*) acorde los requisitos definidos a continuación(RF2, RF3, RF4) para que funcione sobre el sistema operativo MaRTE OS.
- **RF2 - Soporte para objetos protegidos:**
  La implementación debe incluir soporte para objetos protegidos (protected objects) [6], garantizando la sincronización y exclusión mutua siguiendo así el estándar Ada.
- **RF3 - Función de delay relativo:**
  El sistema debe proporcionar una función de delay relativo que permita suspender la ejecución de una tarea por un tiempo determinado relativo al instante actual.
- **RF4 - Función de delay absoluto:**
  El sistema debe proporcionar una función de delay absoluto que permita suspender la ejecución de una tarea hasta un instante específico en el tiempo.
- **RF5 - Integración con MaRTE OS:**
  El RTS portado debe integrarse adecuadamente con MaRTE OS, aprovechando sus servicios y APIs disponibles.

3.2. Requisitos No Funcionales

- **RNF1 - Plataforma:**
  La implementación debe estar diseñada para ejecutarse en la plataforma STM32F4 y el sistema operativo MaRTE OS, considerando sus limitaciones y características específicas, tales como arquitectura ARM Cortex-M4, memoria disponible, y servicios del sistema operativo.
- **RNF2 - Fiabilidad:**
  El sistema debe ser robusto y garantizar la correcta ejecución de las tareas y sincronizaciones bajo condiciones normales.
- **RNF3 - Portabilidad:**
  El código portado debe ser modular y fácilmente adaptable a futuras versiones de MaRTE OS o diferentes placas compatibles con MaRTE OS.
- **RNF4 - Mantenibilidad:**
  El código debe realizar las modificaciones mínimas al RTS de Ada original, haciendo así su posterior actualización más liviana.

- **RNF5 - Tiempo real:**
  La implementación debe cumplir con los requisitos de tiempo real, asegurando que las funciones de delay y la gestión de objetos protegidos respondan en los tiempos esperados. Siendo comprobado mediante mediciones usando entornos controlados, predecibles y diseñados previamente con tal objetivo.

Capítulo 4

Diseño del sistema

4.1. Estructura de la librería

Este sistema adopta una arquitectura en capas (como se muestra en la Figura 4.1). En el nivel superior se encuentra la definición de la tarea; a continuación, se sitúa la capa de planificación y gestión, donde interviene MaRTE OS; y, finalmente, en el nivel más bajo se encuentra el hilo POSIX. Esta última capa proporciona la estructura fundamental para la creación y ejecución de código en Ada sobre el sistema operativo MaRTE OS, que actúa como intermediario antes de acceder a la interfaz POSIX.

El lenguaje de programación Ada se caracteriza por su robustez y por ofrecer mecanismos que facilitan la creación de tareas concurrentes. Asimismo, garantiza el cumplimiento de estándares relevantes en el ámbito de los sistemas en tiempo real, como la correcta gestión de prioridades.

Con el fin de ilustrar el funcionamiento de la arquitectura propuesta, en los siguientes apartados se empleará un ejemplo basado en la definición y gestión de una tarea en Ada. A partir de este caso, se detallará el papel que desempeña cada una de las capas. En la Figura 4.2 se refleja cómo interactúan entre sí estas capas.

Al definir una tarea en Ada, durante el proceso de compilación se generan automáticamente una serie de bibliotecas proporcionadas por el RTS. Estas bibliotecas forman parte del soporte en tiempo de ejecución del lenguaje y contienen tanto el código como las estructuras de datos necesarias para implementar el comportamiento concurrente de las tareas. Entre otras cosas, permiten la creación, inicialización y planificación de tareas, así como la gestión eficiente de su ciclo de vida. Además, incluyen mecanismos para manejar objetos protegidos, controlar el paso de mensajes, gestionar excepciones, aplicar bloqueos y sincronización, y coordinar correctamente la ejecución paralela. Todo esto se realiza de manera transparente para el programador, lo que facilita el desarrollo de aplicaciones concurrentes, seguras y confiables en Ada.

El RTS establece una comunicación bidireccional con MaRTE OS mediante un conjunto de funciones exportadas desde MaRTE OS hacia el RunTime System. Esto significa que el RTS pone a disposición de MaRTE OS ciertas interfaces o puntos de entrada que este último puede invocar para interactuar con el entorno de ejecución de Ada. Esta arquitectura permite que el RTS delegue en MaRTE OS la gestión de aspectos de bajo nivel, como la planificación de tareas, el manejo de interrupciones o el control de temporización. A su vez, MaRTE OS hace uso de estas funciones proporcionadas por el RTS para llevar a cabo dichas operaciones, garantizando así una coordinación eficiente entre ambas capas del sistema.

En lo que respecta a MaRTE OS, este será el sistema operativo subyacente a través de

la interfaz POSIX, empleando las primitivas necesarias para la creación y gestión de hilos POSIX. De este modo, se asegura que la planificación y ejecución de las tareas definidas en el código de alto nivel se realice conforme a los criterios y políticas especificadas.

Figura 4.1: Esquema de las capas que intervienen en el RTS

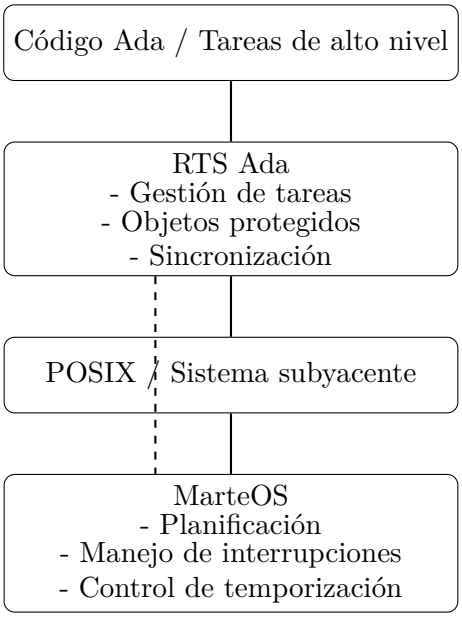

Figura 4.2: Interacción entre el RTS de Ada, MarteOS y POSIX en la arquitectura de capas.

4.2. Tipos de librerías para RunTime Systems

En cuanto a la clasificación de las librerías de Ada, AdaCore soporta las siguientes:[7]

- **Standard Run-Time**: Es el sistema de ejecución completo de GNAT Pro ("Standard"), destinado a plataformas con sistema operativo (como Linux, Windows, VxWorks, RTEMS). Ofrece soporte total del lenguaje Ada, multitarea, manejo de excepciones, temporizadores y servicios del sistema operativo.
- **Embedded Run-Time**: Subconjunto del Standard para sistemas bare-metal o RTOS limitados. Permite multitarea solo mediante perfiles Jorvik o Ravenscar, entrada/salida serial básica, sin red ni capacidades avanzadas. Este perfil garantiza un comportamiento determinista y predecible, lo que lo hace ideal para aplicaciones de tiempo real y sistemas críticos que requieren análisis de planificabilidad y cumplimiento de restricciones temporales estrictas
- **Light Run-Time**: No multitarea, enfoque para sistemas embebidos pequeños y certificables. Soporta tipos numéricos, aritmética de punto flotante, cadenas, excepciones locales, pero sin propagación de excepciones ni liberación de memoria.
- **Light-Tasking Run-Time**: Basado en Light, pero con soporte de multitarea según los perfiles Jorvik (por defecto) o Ravenscar (activable), sin soporte para hilos externos.
- **Configuración personalizable (Configurable Run-Time Facility)**: Permite definir un RTS extremadamente reducido o adaptado para necesidades específicas, seleccionando solo los elementos necesarios del lenguaje, útil en entornos con restricciones extremas o certificaciones particulares.

El RTS objetivo de este trabajo es un RTS Standard, es decir una implementación completa del mismo

Sin embargo, según el punto y estado actual del proyecto, se encontraría bajo la categoría de Light-Tasking Run-Time, dado soporte para multitarea sencillo, dado que solo se dispondría de la presencia de las tareas y su sincronización entre ellas en la forma más básica del estándar. Dado que existen funcionalidades que no están desarrolladas en el propio MaRTE OS y, por tanto, no pueden ser compatibles con las versiones actuales de MarTE, véase por ejemplo las funciones relacionadas con sockets (paquetes de red).

4.3. Estructura del proyecto

El proyecto se encuentra dividido en varias secciones, en las cuales cada una presenta un uso concreto o los archivos contenidos en ella mantienen una relación. Se explica a continuación cómo se estructura la carpeta que contiene el RTS. Como se observa en la Figura 4.3

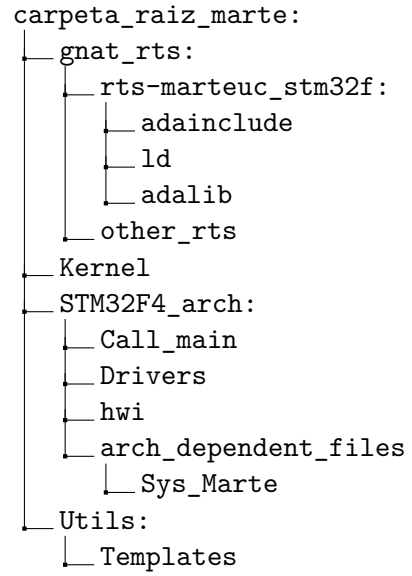

Figura 4.3: Estructura de carpetas del proyecto

A continuación, se describe brevemente el propósito de cada carpeta y algunos de sus archivos más relevantes o a destacar:

- **gnat_rts**: Contiene los runtimes de Ada utilizados por el sistema. Destaca `rts-marteuc_stm32f`, que es el RTS que se ha portado para la placa STM32F4.
  - `adainclude`: Archivos fuente del RTS.
    Estos son los archivos principales para la librería. De aqui parten las funcionalidades de Ada.
    Cabe destacar el archivo `libada.gpr`, el cual define las directrices que debe seguir la herramienta `GPRbuild` para compilar el código del proyecto (véase la sección 2, *Tecnologías y herramientas*).
  - `ld`: Contiene archivos de la arquitectura STM32F4.
    Aunque la librería sea genérica es necesario contar con cierta información de enlace para poder cargar y ejecutarse en la placa objetivo. Es por eso que aquí de se almacenan os scripts de enlazado `(linker scripts), RAM, ROM y memory-map` de la placa STM32F4
  - `adalib`: Bibliotecas compiladas del RTS.
    Donde se almacenan el resultado de la compilación.
- **Kernel**: Contiene todo el código de MaRTE OS. Este es el punto de unión entre el RunTime System y Marte.
- **STM32F4_arch**: Agrupa archivos específicos para la arquitectura STM32F4, como controladores, rutinas de interrupción y el punto de entrada principal. En ella se agregarán aquellos ficheros que solo la placa precise para su correcto funcionamiento.
  - `Call_main`, `Drivers`, `HWI`, `arch_dependent_files Sys_Marte`: Módulos funcionales organizados según su responsabilidad dentro del sistema.
- **Utils**: Incluye los comandos y herramientas de creación del proyecto independiente de la arquitectura.
  - `Templates`: Esquemas de cómo debe ser cada archivo de `Utils` para una arquitectura concreta. Al usar el comando `"msetcurrentarch"` se cambiarán los archivos de `Utils` siguendo lo marcado en los de la carpeta `Templates`

Capítulo 5

Implementación

La implementación del proyecto que aquí se detalla tiene como objetivo portar el sistema de ejecución del lenguaje Ada (RunTime System, RTS) al sistema operativo en tiempo real MaRTE OS, específicamente sobre una plataforma hardware STM32F4. Esta tarea implica una integración profunda entre tres elementos fundamentales: el compilador GNAT de Ada y su infraestructura interna (GNAT + GNARL), el sistema operativo MaRTE OS, y la arquitectura basada en ARM Cortex-M4. La complejidad del trabajo reside en adaptar un entorno originalmente pensado para arquitecturas con mayores recursos a un entorno embebido con recursos limitados, restricciones de tiempo real y características de bajo nivel propias del hardware.

Este capítulo presenta una visión global del proceso de implementación antes de entrar en el desarrollo detallado de cada parte. Para ello, se describen las estrategias adoptadas, los módulos desarrollados y las decisiones clave que han guiado el portado.

A lo largo de las siguientes secciones, se detallará cada uno de estos bloques:

- La generación automática de archivos de dependencias
- La compatibilidad del enlazado
- La implementación de funciones faltantes de entrada/salida(E/S)
- El desarrollo del sistema de salida de consola adaptable

Esta visión general permite enmarcar el conjunto del trabajo y entender cómo cada parte encaja dentro del objetivo mayor de ejecutar aplicaciones Ada sobre MaRTE OS en una plataforma embebida real.

El primer paso ha consistido en analizar y seleccionar las bibliotecas necesarias para que el compilador Ada pueda generar código ejecutable en nuestra plataforma. Dado que el RTS depende de una gran cantidad de unidades del sistema GNAT y GNARL, se ha desarrollado una herramienta que automatiza la creación de archivos de dependencias (`.lst`) a partir de las librerías ya compiladas, seleccionando solo aquellas que realmente están presentes en la arquitectura destino. Esta automatización es esencial para garantizar portabilidad y escalabilidad en futuros desarrollos.

Posteriormente, se han abordado los problemas relacionados con el enlazado del sistema. Muchas funciones que el RTS espera encontrar no están implementadas por defecto en MaRTE OS. Para solventar esto, se ha creado un paquete Ada que exporta explícitamente dichas funciones, incluso si su implementación aún no está completa. Esto permite completar el proceso de compilación y enlazado, manteniendo la posibilidad de extender la funcionalidad más adelante.

En paralelo, se ha detectado la ausencia de funciones básicas de entrada/salida como `fwrite`, `fread` y `fputc`, que normalmente forman parte de la biblioteca estándar en

entornos más convencionales. Para resolver esta carencia, se han reimplementado estas funciones en C, ajustándolas al entorno hardware específico y haciendo uso de primitivas proporcionadas por una capa de bajo nivel integrada con la placa STM32F4.

Asimismo, se ha diseñado y desarrollado un *wrapper* en Ada llamado `Stm32f4_TextIO`, compatible con la interfaz de `Ada.Text_IO`, que permite cambiar dinámicamente entre dos modos de salida: el modo semihosting (útil para depuración con `GDB`) y la salida por UART (útil para ejecución autónoma). Este diseño busca la máxima flexibilidad sin necesidad de recompilar el sistema, proporcionando una experiencia de desarrollo más fluida y robusta.

Todo el diseño de esta implementación se ha guiado por principios de modularidad, portabilidad y mantenimiento a largo plazo. Se ha intentado minimizar la dependencia entre módulos y mantener interfaces limpias que faciliten futuras extensiones, ya sea hacia otros entornos de ejecución o para integrar nuevos servicios del sistema operativo o del compilador.

5.1. Script generador de .lsts

La nueva arquitectura del RunTime System (RTS) utiliza dos archivos esenciales: `libgnat.lst` y `libgnarl.lst`. Estos archivos enumeran las librerías que deben incluirse durante el proceso de compilación:

- `libgnat.lst`: Agrupa las librerías pertenecientes al componente **GNAT**, responsables de la funcionalidad secuencial del lenguaje Ada.
- `libgnarl.lst`: Incluye las librerías asociadas a **GNARL**, que proporcionan soporte al modelo de multitarea (tasking) de Ada.

La biblioteca GNAT contiene una serie de paquetes de propósito general y específico. Representa una funcionalidad que los desarrolladores de GNAT han considerado útil y que se pone a disposición de los usuarios de GNAT. Los paquetes descritos aquí están totalmente soportados y se mantendrá la compatibilidad hacia futuras versiones, por lo que puede utilizar estas funcionalidades con la confianza de que estarán disponibles en futuras versiones.

En GNAT, los servicios de tareas de Ada se implementan sobre GNARL, una capa independiente de la plataforma y del sistema operativo que gestiona la creación de tareas, el rendezvous y las operaciones protegidas.

GNARL descompone estas funciones en operaciones de bajo nivel —crear hilos, establecer prioridades, ceder el procesador, manejar cerrojos, etc-.

Para poder conocer qué paquetes eran requeridos en los .lst, se realizó un análisis en el que se encontró que en las librerías libgnat.a y libgnarl.a estaban listadas las dependencias de cada uno de ellos. Para ello, se utilizó el comando `ar -t`.

Dado que se parte de librerías ya compiladas como base para conocer cuáles son los paquetes requeridos para esta implementación —pero como estas provienen de otras arquitecturas y no de la STM32F4— se deben añadir únicamente aquellos que estén presentes en la carpeta `adainclude`.

Como uno de los objetivos es que el portado de futuras librerías sea lo más sencillo e independiente posible con respecto a la implementación actual, se desarrolló un script que automatiza el proceso de creación de los archivos .lst, buscando las librerías coincidentes y añadiéndolas al archivo.

Su uso es sencillo: se le indica el .a del que debe tomar las librerías y el nombre final del archivo de dependencias; en nuestro caso: `libgnat/gnarl.lst`. Además, al finalizar, el script genera un archivo `notfound.txt` con todas aquellas dependencias que no se hayan encontrado, para poder analizarlas y determinar si realmente son necesarias o si son

específicas del .a del RTS base.

Se muestra aquí el código del script utilizado:

```
#!/bin/bash
#Uso: $1 nombre de libreria $2 nombre del lst a generar
if [ "$#" -ne 2 ]; then
    echo "Usage: $0 <library_name> <lst_name>"
    exit 1
fi

# Limpieza y creacion de las listas
rm -f notfound.lst
rm -f $2

touch notfound.lst
touch $2

# Buscar cada dependencia de ar
for file in $(ar -t $1); do
    name="${file%.*}"
    dependencies=$(find ../adainclude/ -maxdepth 1  -name "$name.*")

    if  [[ -z "${dependencies//[[:space:]]/}" ]]; then
        echo "$name" >> notfound.lst
    else
        for dependency in $dependencies; do
            nameExtension=$(basename "$dependency")
            echo "$nameExtension" >> $2
        done
    fi
done

echo "Procesado de la libreria $1 terminado, los archivos no
   encontrados estan en notfound.lst"
```

Figura 5.1: list_gen.sh: script para generar lista de dependencias

5.2. Compatibilizando el enlazado entre Marte, el RunTime y el código

Una vez se ha conseguido compilar tanto el RunTime System como MaRTE OS (véase sección A.1, «Anexos: Compilar el Proyecto»), al realizar el enlazado con `mgnatmake` u otras herramientas, se producirán errores en los que se indican funciones no definidas en el mismo, esto se debe a que no se está realizando una implementación completa del sistema. Para dar solución a este problema se ha desarrollado un paquete adicional para el RTS incluido en `adainlude`, denominado `unimplemented_functions`.

La función de este paquete simple es exportar todas estas funciones no definidas que se piden para realizar la compilación. En la Figura 5.2 se ejemplifica cómo es la estructura de las funciones exportadas, en todas se hace uso del paquete `Ada.Text_IO` para dejar un mensaje que informe al usuario que dicha función no ha sido desarrollada e integrada al momento en el que se está usando la librería. Existe una excepción importante para ciertas funciones, cosa que ha quedado reflejada en la documentación del código, ciertas funciones son llamadas por la inicialización del paquete `Ada.TexIO` (Estas funciones no se presentan utilidad en la fase actual del proceso de portado, aún no se han alcanzado, para realizar alguna operación. Sin embargo, a la hora de construir los paquetes que `Ada.TexIO` necesita

para su funcionamiento, se llama a las funciones, creando así un error de dependencias ya que estas funciones que se están incluyendo para el paquete `Ada.TexIO` usan el mismo. Dado que las funciones no son utilizadas en esta etapa, se definen como `null` para permitir así que compile y dándose por conocido que existe un subconjunto de funciones que no enviarán un mensaje avisando que no están implementadas si se realizan llamadas a dichas funciones.

```
with Ada.Text_IO; use Ada.Text_IO;

package body Unimplemented_Functions is
   procedure Atomic_Load_8;

   --  Exportar las funciones que son requeridas por el enlazado
   pragma Export (C, Atomic_Load_8, "__atomic_load_8");

   procedure Atomic_Load_8 is
   begin
      Put_Line ("Function '__atomic_load_8' not implemented.");
   end Atomic_Load_8;

   -- Example of function used by TextIO
   procedure Atoi is null;
   pragma Export (C, Atoi, "atoi");

   --  More function definitions

end Unimplemented_Functions;
```

Figura 5.2: ejemplo de definición de funciones no implementadas en ada

5.3. Definición de funciones de entrada/salida faltantes

Se detectó la ausencia de las funciones `fwrite`, `fput` y `fread`. Por esta razón, se implementaron e incorporaron en el archivo:
`stm32f4_arch/arch_dependent_files/IO_functions.c`

Las funciones `fwrite`, `fput` y `fread` son necesarias en nuestra implementación debido a que muchas operaciones básicas de entrada/salida en bajo nivel dependen de ellas para interactuar correctamente con el sistema. Aunque en ciertos entornos pueden estar proporcionadas por la biblioteca estándar, en nuestro caso particular —dado que se está trabajando sobre una implementación personalizada o un entorno limitado— estas funciones no estaban disponibles por defecto, lo que generaba errores en tiempo de ejecución.

`fwrite` es esencial para escribir bloques de datos en un flujo (stream), lo cual permite, por ejemplo, imprimir cadenas completas o volcar estructuras a archivos o salidas estándar.

`fput` (o `fputc`) se utiliza para escribir caracteres individuales, y muchas funciones de nivel superior como `printf` dependen de ella internamente para emitir texto carácter por carácter.

`fread` permite leer bloques de datos desde un flujo, lo que es indispensable para manejar entradas, cargar archivos o recibir datos de manera estructurada.

Para realizar estas funciones se han importado las siguientes funciones que se describen a continuación:

- **`write_newlib_bb`**: Es la implementación bare-metal de la llamada al sistema `write()` que la biblioteca `newlib` requiere. Recibe como parámetros un descriptor de archivo

(`fd`), un puntero al búfer de datos (`buf`) y el número de bytes a escribir (`nbytes`). Devuelve la cantidad de bytes efectivamente escritos (o un valor negativo en caso de error).

- **`read_newlib_bb`**: Implementa la llamada al sistema `read()` para `newlib`. Sus parámetros son el descriptor de archivo (`fd`), un donde almacenar los datos leídos (`buf`), y el número máximo de bytes a leer (`count`). Devuelve la cantidad de bytes leídos (o `0` si se alcanza fin de archivo, `-1` en caso de error). En sistemas con GNAT, esta función típicamente obtiene datos desde una entrada estándar simulada (como la consola serie o el monitor de depuración). Permite que primitivas de Ada como `Get_Line` funcionen correctamente al apoyarse en la infraestructura de `newlib`.

Dado que estas funciones actúan como una capa base sobre la cual se construyen otras funciones más complejas, su implementación fue necesaria para garantizar el soporte funcional completo de operaciones de E/S dentro del sistema.

En la Figura 5.3 se muestran las funciones desarrolladas para la entrada/salida.

```c
#include <stddef.h>  // Para size_t
#include <unistd.h>  // Para write y read
#include <sys/stat.h>  // Para struct stat

// Declaraciones de funciones de newlib-bb.c
extern int write_newlib_bb(int fd, char *buf, int nbytes);
extern int read_newlib_bb(int fd, char *buf, int count);

// Redefinición de fwrite
size_t fwrite(const void * ptr, size_t size, size_t count, void *
    stream) {
    int fd = (int)(size_t)stream;  // Interpretamos el stream como
        descriptor de archivo (entero)
    int total_bytes = size * count;

    int written = write_newlib_bb(fd, (char *)ptr, total_bytes);
    if (written <= 0) return 0;
    return written/size;
}

// Redefinición de fputc
int fputc(int c, void * stream) {
    int fd = (int)(size_t)stream;  // Interpretamos el stream como
        descriptor de archivo (entero)
    char ch = (char)c;

    int written = write_newlib_bb(fd, &ch, 1);
    if (written == 1) {
        return c;  // Devuelve el carácter escrito
    } else {
        return -1;  // Error
    }
}

// Redefinición de fread
size_t fread(void * ptr, size_t size, size_t count, void * stream) {
    int fd = (int)(size_t)stream;  // Interpretamos el stream como
        descriptor de archivo (entero)
    int total_bytes = size * count;

    int read = read_newlib_bb(fd, (char *)ptr, total_bytes);
    if (read <= 0) return 0;
    return read/size;
}
```

Figura 5.3: IO_functions.c: redefinición de funciones de entrada/salida básicas

5.4. wrapper para permitir la salida por consola en la placa de desarrollo

Una vez se entró en la fase de pruebas sobre el hardware real en lugar del emulador QEMU, la salida por consola quedó disponible únicamente a través de la opción semihosting, es decir solo cuando la placa se configuraba en modo debug y se realizaba una conexión con la herramienta GDB(véase la sección 2, *Tecnologías y herramientas*).

Dado que la placa cuenta con pines GPIO, se agregó siguiendo el esquema de la Figura 5.4 una conexión puerto serie UART, gracias a ella se podrá disponer de una forma alternativa al semihost que evitará tener una conexión permanente entre el dispositivo maestro

y la placa ejecutando el software compilado para la misma.

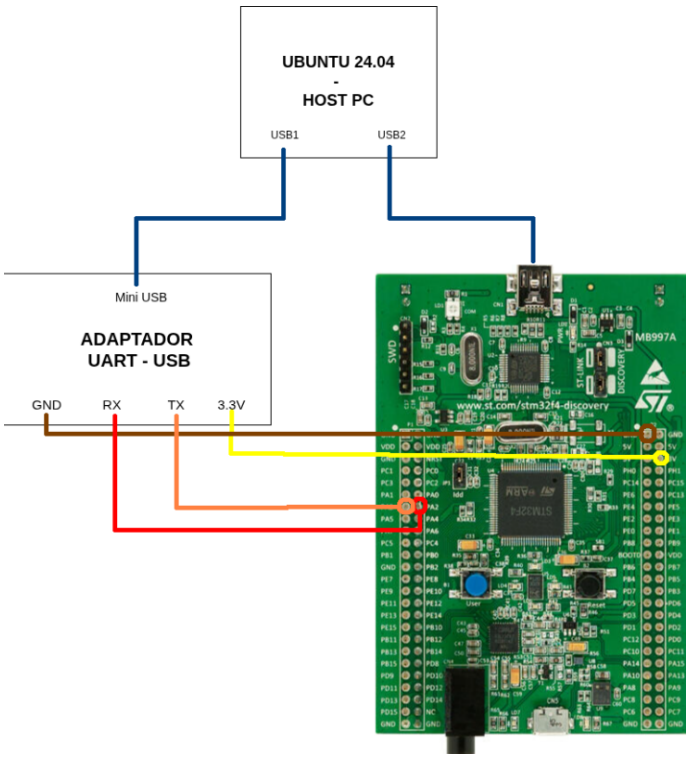

Figura 5.4: Esquema de conexión para el uso del puerto serie

Para que el uso de las salidas de texto por la placa sea cómodo y no se tenga que recompilar el código cada vez que se desee alternar entre la opción de semihost y la salida uart, se optó por el desarrollo de un nuevo paquete para el sistema: `stm32f4_TextIO`, este paquete presenta los mismos métodos y funciones que `Ada.Text_IO`, facilitando así su reemplazo en el código y haciendo así que sea totalmente amigable a nuevos usuarios. Se muestran las cabeceras definidas para el paquete en la Figura 5.5

El paquete `Ada.Text_IO`, perteneciente a la librería estándar del lenguaje Ada, proporciona los mecanismos básicos de entrada y salida de texto, permitiendo la lectura y escritura de caracteres, cadenas y números en formato textual tanto por consola como mediante ficheros; incluye procedimientos como `Put`, `Put_Line` y `Get`, que constituyen la base para la interacción del programa con el usuario y para el manejo de archivos de texto de manera estructurada.

El motivo detrás de esta modificación radica en las limitaciones del semihosting y la necesidad de disponer de una salida de depuración más flexible. El semihosting permite que la placa envíe información de depuración a través de la conexión de depuración (GDB), pero esto requiere que la placa esté constantemente conectada al depurador, lo que puede interferir con el comportamiento normal del software y limita la independencia del hardware.

Para el desarrollo del paquete se tomaron como ejes centrales la modularidad, es decir, que se puedan incluso añadir nuevas formas de salida de texto por consola de forma rápida y sencilla, por eso en el código se utiliza una función más interna que contiene la selección de los métodos más primitivos de forma que solo hay que modificar estos wrappers internos, en este caso particular estas primitivas fueron `Output` para salida estándar y `OutPut_error` para salida de error.

Otra de las primitivas fue hacer transparentes todas las necesidades nuevas que pueda suponer el paquete como es el caso de la inicialización de las salidas por consola, siendo que,

aunque el paquete cuenta con un método de inicialización `Initialize()`, este es llamado de forma interna cuando se detecta que no ha sido inicializado evitando que el usuario de la librería tenga que hacerlo explícitamente. Si bien es cierto que se tomó este patrón de diseño, existen 2 funciones propias y específicas de este paquete que no están presentes en la librería que se pretende imitar (`Ada.Text_IO`): `Set_Mode()` y `Get_Active_Mode()`

`Set_Mode()`: Permite el cambio de modo, se dispone de un enumerado en el que están las opciones `UART` y `SEMIHOST`. Esto, además, hace que su uso sea muy sencillo, como se puede observar en el test 6.1.

`Get_Active_Mode()`: Esta función nos permite conocer qué modo está activo, de cara a ejecuciones condicionales o incluso detección de errores, además de que al tratarse de una librería compartida, otras funciones pueden haber seleccionado un modo de consola no deseado y debemos conocerlo para su posterior restauración segura.

También fue necesaria la modificación de uart_console.c A.2, para ajustar la velocidad del reloj y así permitir que la comunicación se realizara usando los parámetros correctos y necesarios.

```
--  Stm32f4_TextIO.ads
--  Text_IO-like Ada wrapper for STM32F4 UART console, modeled after
    MaRTE.Direct_IO

with System;

package Stm32f4_TextIO is
   pragma Preelaborate;

   procedure Put (Str : in String);
   procedure Put (N : in Integer; Base : in Positive := 10);
   procedure Put_Line (Str : in String);
   procedure New_Line;
   procedure Initialize;
   procedure Error (Msg : in String; Fatal : in Boolean := False);
   procedure Put_Error (Msg : in String; Fatal : in Boolean := False);

   type Output_Mode is (SEMIHOST, UART);

   --  control output mode
   procedure Set_Mode (New_Mode : in Output_Mode);
   --  0 for \gls{semihosting} (marteIO), 1 for UART console
   function Get_Active_Mode return Output_Mode;

private
   -- Internal state
   Is_Initialized : Boolean := False;  -- True if the text IO is
      initialized
   Mode : Output_Mode := UART;  -- 0 para \gls{semihosting}, 1 para
      UART
   -- Internal error functions
   procedure Output (Str : in String);
   procedure Output_Error (Str : in String);

end Stm32f4_TextIO;
```

Figura 5.5: stm32f4_textIO.ads: Cabeceras del paquete wrapper para salida por consola

Capítulo 6

Pruebas y validación

El proceso de pruebas tuvo como objetivo verificar que el sistema desarrollado no solo compilara y enlazara correctamente, sino que además ofreciera las funcionalidades esperadas en un entorno de tiempo real embebido. Las pruebas se diseñaron con un doble propósito: por un lado, detectar errores durante la fase de desarrollo; y por otro, demostrar que la librería resultante es válida y utilizable para programas reales que se ejecutan sobre MaRTE OS en la placa STM32F4.

6.1. Fases del proceso de validación

El proceso completo se estructuró en tres fases principales:

1. **Compilación y enlazado**. Esta etapa consumió gran parte del tiempo, ya que se partía de los archivos del RunTime System. Fue necesario comprender en detalle las herramientas de compilación y depuración disponibles, así como determinar los parámetros de compilación y enlazado adecuados. El enlace con MaRTE OS resultó especialmente complejo, pues requirió múltiples iteraciones para identificar qué componentes eran imprescindibles y cómo debían integrarse.

2. **Compilación de los programas de prueba**. Una vez establecida la infraestructura, se desarrollaron pequeños programas (tests) que ejercitaban las primitivas clave de la librería. Estos programas estaban diseñados para ser simples, repetibles y fáciles de interpretar.

3. **Ejecución y validación**. Inicialmente, los tests se ejecutaron en un emulador, lo que permitió depurar paso a paso y detectar fallos tempranos. Posteriormente, se repitieron en la placa STM32F4, validando así la interacción con periféricos reales y el comportamiento en condiciones cercanas a un entorno de producción.

6.2. Clasificación y descripción de las pruebas

A continuación, se describen los distintos tests diseñados, agrupados según el aspecto que pretendían validar.

6.2.1. Pruebas básicas de compilación y salida por consola

- `tests/hello/hello_world.adb` Este test consiste en la impresión de un mensaje sencillo por consola. Aunque pueda parecer trivial, resulta fundamental como primera validación: confirma que el entorno de compilación, el enlazado y la ejecución básica funcionan correctamente. Además, sirve como punto de partida para estudiantes y futuros desarrolladores, pues ofrece un ejemplo mínimo de programa Ada ejecutándose sobre MaRTEOS.

6.2.2. Pruebas de temporización

En sistemas de tiempo real, la temporización es un aspecto esencial. Es necesario garantizar que una tarea puede ser suspendida durante un intervalo específico o hasta un instante absoluto del reloj del sistema. Estas pruebas validaron que las primitivas de temporización implementadas funcionan como se espera.

- `tests/simple_tasking/simple_delay` Este test comprueba que la instrucción de *delay relativo* funciona correctamente. En términos prácticos, significa que una tarea puede "dormir" durante un número de milisegundos y, al despertar, continuar con su ejecución sin interferencias.
- `tests/simple_tasking/delay_until.adb` En este caso, se valida la semántica del *delay absoluto*, que suspende la tarea hasta un instante de tiempo exacto marcado por el reloj del sistema. Esta funcionalidad es crítica en sistemas donde deben cumplirse plazos temporales estrictos (por ejemplo, tareas que deben ejecutarse cada 100 ms de forma periódica).

6.2.3. Pruebas de concurrencia básica

Un sistema multitarea debe ser capaz de ejecutar varias tareas en paralelo sin necesidad de sincronización explícita. Estas pruebas buscan confirmar que el planificador reparte de forma adecuada el tiempo de CPU entre tareas independientes.

- `tests/simple_tasking/no_sync_tasks.adb` Se crean dos tareas completamente independientes, sin variables compartidas ni mecanismos de sincronización. El objetivo es verificar que ambas pueden alternarse en la ejecución, demostrando así que el planificador del sistema operativo funciona correctamente incluso en escenarios sencillos.

6.2.4. Pruebas de concurrencia avanzada: objetos protegidos

Ada ofrece una abstracción denominada *objeto protegido*, que permite encapsular datos compartidos y controlar su acceso concurrente.

Objeto protegido

Un **objeto protegido** es una estructura que combina *datos* y *operaciones* sobre esos datos dentro de una misma entidad, garantizando que solo una tarea pueda acceder a la sección crítica a la vez. Esto se logra mediante un mecanismo interno de *exclusión mutua*, de manera que el programador no necesita gestionar explícitamente semáforos u otros mecanismos de bajo nivel.

Semáforos

Para entender mejor la ventaja de los objetos protegidos, conviene recordar qué es un **semáforo**. Un semáforo es un mecanismo clásico de sincronización en sistemas concurrentes que permite controlar el acceso a recursos compartidos. Funciona como un contador:

- Cuando una tarea quiere acceder al recurso, debe "tomar" el semáforo.
- Si el semáforo está disponible, la tarea continúa y decrementa el contador.
- Si no está disponible (contador = 0), la tarea se bloquea hasta que otro proceso "libere" el semáforo incrementando el contador.

Aunque los semáforos son poderosos, su uso puede ser propenso a errores como *interbloqueos* (conocidos como *deadlocks*) o *condiciones de carrera* , porque la gestión de adquisición y liberación depende completamente del programador.

Ventajas de los objetos protegidos

Los objetos protegidos de Ada abstraen este manejo, proporcionando operaciones llamadas *procedimientos* y *funciones protegidas*:

- **Procedimiento protegido:** Permite modificar los datos internos de manera exclusiva, garantizando que solo una tarea acceda a la vez.
- **Función protegida:** Permite leer los datos internos y, dependiendo de la configuración, garantiza la exclusión mutua o permite acceso concurrente seguro.

De esta manera, los objetos protegidos simplifican la programación concurrente, mejoran la seguridad frente a errores de sincronización y permiten diseñar sistemas embebidos más fiables sin necesidad de gestionar manualmente semáforos.

- `tests/protected/test_protected.adb` En este test se definió un objeto protegido sencillo que era accedido por dos tareas diferentes. Se comprobó que:
  - nunca se producía acceso simultáneo al recurso,
  - las tareas bloqueadas esperaban correctamente cuando el objeto estaba ocupado,
  - la ejecución continuaba en cuanto el recurso quedaba libre.

  Este test es especialmente importante porque valida el correcto soporte de sincronización en la librería desarrollada, una de las bases del modelo de concurrencia en Ada.

6.2.5. Pruebas sobre hardware específico

Además de las validaciones en emulación, fue necesario comprobar que la librería funcionaba sobre el hardware real, interactuando con los periféricos de la placa STM32F4.

- `examples/stm32f4/blink_led.adb` Este test utiliza las primitivas de temporización para hacer parpadear los LEDs integrados en la placa. Aunque conceptualmente es muy simple, demuestra que el sistema operativo es capaz de controlar periféricos reales respetando los plazos temporales configurados.
- `examples/stm32f4/testio_wrapper.adb` Descrito en detalle en la Sección 6.4, este test valida la librería de entrada/salida `Stm32f4_TextIO`, asegurando que los mensajes pueden redirigirse tanto al puerto serie como al depurador mediante semihosting.

6.3. Resultados de las pruebas

En conjunto, las pruebas permitieron validar los siguientes aspectos:

- El entorno de compilación y enlazado funciona correctamente y permite ejecutar programas Ada sobre MaRTE OS.
- Las primitivas de temporización (`delay`, `delay until`) cumplen con la semántica esperada, tanto en emulación como en hardware real.
- El planificador es capaz de ejecutar múltiples tareas de manera concurrente.
- Los objetos protegidos garantizan exclusión mutua y sincronización correcta entre tareas.
- La interacción con el hardware (puerto serie, LEDs) funciona sin errores y de forma estable.

Gracias a las pruebas, en concreto aquellas que implicaban la interacción entre varias tareas, se detectó un error menor: Cuando una tarea finaliza su ejecución impide que las demás continúen de forma normal, provocando la terminación de todo el programa. Esto nos obliga a que estas funciones se ejecuten dentro de bucles con esperas prolongadas, de modo que se permita el cambio de contexto, dado que contamos con una placa de un solo núcleo (single-core). De manera similar, la función principal del sistema de tasking debe incluir esta espera para que las tareas definidas puedan ejecutarse. Se ha considerado un fallo menor dado que en la mayoría de sistemas embebidos programados en Ada es más habitual planificar funciones que se ejecuten de manera continua, por lo que no es común que las tareas finalicen durante la operación normal. Dadas las restricciones temporales presentes, este problema queda marcado como trabajo a futuro.

6.4. Test wrapper textio

Para comprobar el correcto funcionamiento de la nueva librería, que permitía al usuario imprimir tanto por la salida por el puerto serie como por la consola de las herramientas de análisis y depurado, se desarrolló el test que se muestra a continuación en la Figura 6.1.

```
with Stm32f4_TextIO; use Stm32f4_TextIO;

procedure TestIO_Wrapper is
begin
    -- Estas funciones apareceran por el puerto serie
   Put("Prueba mensaje regular");

   Put_Error("testando error");

   Set_Mode(SEMIHOST);

   -- Estas apareceran por el debug
   Put_Line("Prueba mensaje regular");

   Put_Error("testando error");


end TestIO_Wrapper;
```

Figura 6.1: TestIO_Wrapper.ads: Test para comprobar el correcto funcionamiento de la librería stm32f4_textIO

Capítulo 7

Conclusiones y trabajo futuro

Gracias al desarrollo de este trabajo se ha conseguido obtener una versión funcional del RunTime System de Ada, integrada con el sistema operativo MaRTE OS. Se han realizado modificaciones mínimas en el código externo de Ada, cosa que ha supuesto una mayor complejidad al ser necesario mayor cuidado y comprensión del RTS, este sistema permite mantener la posibilidad de actualización gracias a su diseño modular basado en *imports*, junto con los *scripts* desarrollados para facilitar futuros portados. Además, la disponibilidad de este RTS permite ejecutar aplicaciones concurrentes sencillas utilizando mecanismos estándar de Ada en una plataforma empotrada como la STM32F4

Este resultado es fruto de un proceso largo, complejo y exigente, que ha requerido una inversión significativa de tiempo, esfuerzo y dedicación. Desde las fases iniciales, se ha llevado a cabo una labor intensiva de exploración, comprensión y adaptación de componentes clave, enfrentando múltiples desafíos tanto a nivel técnico como metodológico. El hecho de trabajar sobre una arquitectura distinta (ARM) y con un sistema operativo no estándar como *MaRTE OS*, ha supuesto una barrera adicional, ya que no se contaba con documentación extensa ni con comunidades de soporte, como ocurre en entornos más comunes como *Windows*, *MacOS* o distribuciones *Linux*.

A lo largo del desarrollo, ha sido necesario realizar múltiples pruebas, identificar errores difíciles de localizar y emplear herramientas de depuración de forma sistemática. Estos procesos no solo implicaron la detección de fallos, sino también su análisis detallado y la posterior implementación de soluciones efectivas, lo que ha exigido una comprensión profunda tanto del funcionamiento interno del RTS como de su interacción con el sistema operativo. En paralelo, se han diseñado y aplicado mecanismos para facilitar la trazabilidad del código y acelerar tareas repetitivas, con el objetivo de optimizar el flujo de trabajo y minimizar errores en futuras iteraciones.

Las pruebas presentaron dificultades tanto en las herramientas de emulación como en el hardware real. En el caso de las herramientas de emulación, fue necesario investigar su funcionamiento y desarrollar comandos específicos que permitieran un proceso de desarrollo más rápido y eficiente. Por otro lado, en lo que respecta al hardware, es decir, la propia placa, resultó aún más complejo depurarla y determinar el origen de los errores. Además, la obtención de archivos compilables supuso un reto adicional, ya que en una primera fase solo eran compatibles con el emulador.

En resumen, se ha llevado a cabo un trabajo minucioso, continuado y altamente técnico, superando obstáculos significativos que, sin una dedicación constante y una actitud proactiva frente a los problemas, habrían imposibilitado la consecución de un RTS plenamente funcional y adaptado a las características específicas del entorno de destino.

Existen ciertas limitaciones en el sistema actual, las tareas en su finalización impiden que el resto continue de forma regular, lo cual nos restringe a que estas funciones terminen en bucles con esperas de gran duración para que permitan el cambio de contexto dado que disponemos de una placa single-core. Lo mismo ocurre con la función principal en tasking que debe contener esa espera para que las tareas definidas puedan ejecutar, aunque cabe

destacar que el uso más habitual de Ada en sistemas embebidos es tener las funciones a ejecutar planificadas y no es habitual que estas terminen.

7.1. Trabajo a futuro

- **Añadir nuevas funcionalidades**: Al igual que se ha desarrollado un paquete de textIO alternativo, se podría proveer desde marte de librerías para el control de la GPIO desde alto nivel o de cualquier tipo de sistema de entrada/salida que posea la placa.
- **Full RTS**: convertirse en un RunTime System completo (`Full RTS`(véase la sección 4.2, *Tipos de librerías para RunTime Systems*)), esto se debería hacer en conjunto con el desarrollo y evolución de Marte dado que existen ciertas características del estándar que define AdaCore. Pero se debería realizar un análisis previo para añadir aquello que soporte nuestra arquitectura.
- **Soporte para nuevas arquitecturas**: Gracias a los conocimientos adquiridos en este portado, tomar las estrategias y métodos que han funcionado y llevarlos a nuevas arquitecturas como RISC-V, una arquitectura que comienza a ser utilizada a nivel europeo y que cuenta con el apoyo de grandes centros como el BSC(Barcelona Supercomputing Centre) donde se están diseñando procesadores con dicha arquitectura, por tanto, en un futuro puede que veamos sistemas embebidos sobre RISC-V.
- **Soporte para nuevas placas**: Lo mismo aplica para otras placas de desarrollo como raspberry Pi u cualquiera que marte tenga soporte desarrollado.

Bibliografía

[1] "Runtime system." https://en.wikipedia.org/wiki/Runtime_system. consultado en julio de 2025.

[2] AdaCore, "Ada." https://https://www.adacore.com/products/languages, 2025.

[3] https://marte.unican.es/about.htm, "Marteos." https://marte.unican.es/, 2000-2017.

[4] "Estandarización ada." https://es.wikipedia.org/wiki/Ada_(lenguaje_de_programaci%C3%B3n)#Estandarizaci%C3%B3n. consultado en julio de 2025.

[5] Qemu, "Qemu: Generic and open source machine emulator and virtualizer." https://www.qemu.org/.

[6] AdaCore, "Protected units and protected objects." https://docs.adacore.com/live/wave/arm05/html/arm05/RM-9-4.html.

[7] AdaCore, "Predefined gnat pro run-times," 2023. Resumen de tipos de RTS disponibles.

Apéndice A

Anexos

A.1. Compilar el Proyecto

Para poder compilar el proyecto completo debemos cumplir las siguientes dependencias A.1.1. Y continuar con los pasos:

1. Añadir al `PATH` la carpeta `utils` del proyecto.
2. Ejecutar ./minstall en la carpeta raíz de MaRTE OS
3. Seleccionar arquitectura

```
msetcurrentarch stm32 f4
```

4. compilar rts y marte

```
mkrtsmarteuc && mkmarte
```

En este punto ya tendríamos disponibles el RTS y MaRTE OS y con las herramientas del IDE o con los comandos incluidos en utils como `mgnatmake` podremos compilar nuestros programas para usar la librería y el sistema sobre la placa.

A.1.1. Dependencias

- **GNAT Studio**: IDE proporcionado por AdaCore, recomendado para la programación en Ada.
- **GPRBuild**: herramienta de construcción de proyectos en Ada, utilizada para compilar múltiples unidades y gestionar dependencias.
- **Alire (alr)**: gestor de paquetes y proyectos en Ada que permite instalar dependencias y gestionar entornos fácilmente.
- **GNAT Native 14.2**: compilador nativo de Ada, necesario para compilar el código en la plataforma anfitriona. Se puede obtener usando el comando:

```
alr get gnat_native~14.2
```

- Compilación cruzada:
  - **gnat-arm-elf-linux64-x86_64-14.2.0-1**: toolchain oficial de AdaCore para compilación cruzada en sistemas embebidos ARM.

A.2. Guía de desarrollo

Una vez dispongamos del sistema compilado, como se ha mostrado en el apartado A.1, podremos hacer uso de las librerías a través de un emulador o de forma nativa subiendo el compilado a la placa y viendo su ejecución.

A.2.1. Uso en emulador

Para emular usaremos el comando `arm-eabi-gnatemu` incluido en la versión del compilador cruzado utilizado en las versiones antiguas de arm-eabi, dado que, debido a su salida como producto comercial, se optó por la retirada del compilador QEMU. En el caso de este desarrollo, se utilizó solo para la emulación:

```
GNATEmulator 2018 (20180524) for \gls{arm} Bareboard (Cortex-M3),
using: qemu-system-arm
 QEMU emulator version 2.8.1(AdaCore-2.8.1-qemu-20180524)
 Copyright (c) 2003-2016 Fabrice Bellard and the QEMU Project developers
```

Con las dependencias cubiertas, se debe compilar el código haciendo uso del comando `mgnatmake`, es importante tener en el $PATH la carpeta `Utils` del proyecto para poder hacer uso de los comandos mencionados. Una vez se obtenga el archivo ejecutable, se podrá hacer uso del emulador.

```
arm-eabi-gnatemu --board=stm32f4 <ejecutable>
```

A.2.2. Uso en placa de forma nativa

Se debe disponer de Opencd y GnatStudio Se debe lanzar GnatStudio con las siguientes opciones:

```
gnatstudio -Xmtarget=stm32f4  -P <proyecto_deseado>.gpr &
```

Y pulsar en el botón que se muestra en la Figura A.1 remarcado en color rojo.

Con eso, una vez subidos los archivos a la placa por parte de gnatstudio tendremos cargado y ejecutando el programa sobre la misma.

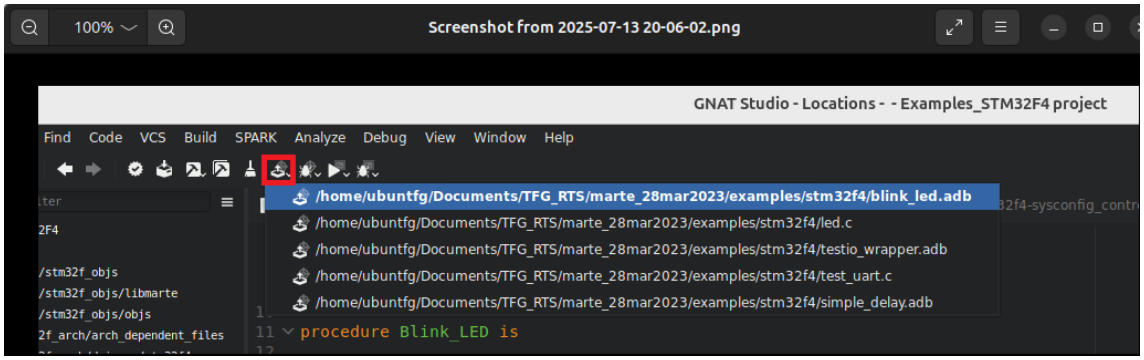

Figura A.1: Botón para compilar y subir un programa a la placa en la interfaz de gnatstudio

A.3. Código desarrollado para la salida por consola

Como se muestra en las Figuras A.2 y A.3. Se ha desarrollado un paquete para poder seleccionar la salida por consola más adecuada según indique el usuario. —véase la sección 5.4, «Implementación: wrapper para permitir la salida por consola en la placa de desarrollo»—

```
--  Stm32f4_TextIO.adb
--  Ada Text_IO-like wrapper for STM32F4 UART console

with Interfaces.C;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;

package body Stm32f4_TextIO is

   ------------------------------------------------------------
   --  Imports of  functions for text ios                    --
   ------------------------------------------------------------
   -- Import the C UART initialization function
   procedure uart_console_init (baudrate : Interfaces.C.int);
   pragma Import (C, uart_console_init, "uart_console_init");

   -- Import the C UART write function (string)
   procedure uart_print_console (msg : Interfaces.C.char_array);
   pragma Import (C, uart_print_console, "uart_print_console");

   -- Import the C UART write function (char)
   procedure uart_console_putchar (c : Interfaces.C.char);
   pragma Import (C, uart_console_putchar, "uart_console_putchar");

   -- Import the C UART error write function (string)
   procedure uart_print_error (msg : Interfaces.C.char_array);
   pragma Import (C, uart_print_error, "uart_print_error");

   -- Marte Direct_IO-like interface for STM32F4 UART console

   ----------------------------------
   --  Basic_Stdout_Initialization --
   ----------------------------------
   procedure Basic_Stdout_Initialization;
   pragma Import (Ada, Basic_Stdout_Initialization,
                  "basic_stdout_initialization");


   ----------------------------------
   --  Basic_Stderr_Initialization --
   ----------------------------------
   procedure Basic_Stderr_Initialization;
   pragma Import (Ada, Basic_Stderr_Initialization,
                  "basic_stderr_initialization");

   --
      ------------------------------------------------------------------------

   --  Functions for C Code
      ---------------------------------------------------
   --
      ------------------------------------------------------------------------


   ------------------------------
```

```
   --  Direct_Write_On_Stdout  --
   -------------------------------
   procedure Direct_Write_On_Stdout
     (Buffer_Ptr : in System.Address;
      Bytes      : in Interfaces.C.size_t);
   pragma Import (Ada, Direct_Write_On_Stdout, "direct_write_on_stdout"
       );

   -------------------------------
   --  Direct_Write_On_Stderr  --
   -------------------------------
   procedure Direct_Write_On_Stderr
     (Buffer_Ptr : in System.Address;
      Bytes      : in Interfaces.C.size_t);
   pragma Import (Ada, Direct_Write_On_Stderr, "direct_write_on_stderr"
       );

   ----------------------------
   -- Direct_Read_From_Stdin --
   ----------------------------
   function Direct_Read_From_Stdin return Interfaces.C.unsigned_char;
   pragma Import (Ada, Direct_Read_From_Stdin, "direct_read_from_stdin"
       );


   -- Internal error message
   Str_Internal_Error : constant String :=
     Standard.ASCII.CR & "  MaRTE OS INTERNAL ERROR: ";
   Str_Fatal_Error : constant String := " (Fatal Error) ";

   -------------------------------
   --    Functions Text_IO      --
   -------------------------------
   procedure Initialize is
   begin
      uart_console_init(115200);
      Basic_Stdout_Initialization;
      Basic_Stderr_Initialization;
   end Initialize;

   -- Generic call that change output console
   procedure Set_Mode (New_Mode : Output_Mode) is
   begin
      Mode := New_Mode;
   end Set_Mode;

   function Get_Active_Mode return Output_Mode is
   begin
      return Mode;
   end Get_Active_Mode;

   ----------------------
   -- Out wrappers     --
   ----------------------
   procedure Output (Str : in String) is
   begin
      if not Is_Initialized then
         Initialize;
         Is_Initialized := True;
      end if;
```

```
      if Mode = UART then
         -- Use UART console
         uart_print_console (Interfaces.C.To_C (Str));
      else
         -- Use direct write
         Direct_Write_On_Stdout (Str'Address, Interfaces.C.size_t (Str'
            Length));
      end if;
   end Output;

   procedure Output_Error (Str : in String) is
   begin
      if not Is_Initialized then
         Initialize;
         Is_Initialized := True;
      end if;

      if Mode = UART then
         -- Use UART console error output
         uart_print_error (Interfaces.C.To_C (Str));
      else
         -- Use direct write
         Direct_Write_On_Stderr (Str'Address, Interfaces.C.size_t (Str'
            Length));
      end if;
   end Output_Error;



   ------------------------------------
   ----    Text_IO Procedures      ----
   ------------------------------------
   procedure Put (Str : in String) is
   begin
      Output(Str);
   end Put;

   procedure New_Line is
      C : Character := Standard.ASCII.LF;
   begin
      Output(String'(1 => C));
   end New_Line;

   procedure Put (N : in Integer; Base : in Positive := 10) is
      C : Character;
      Mag : Positive;  -- Magnitude of the number
      Num_Abs : Natural;
      Is_Negative : Boolean := False;
      Digit : Natural;
   begin
      if N < 0 then
         Num_Abs := -N;
         Is_Negative := True;
      else
         Num_Abs := N;
      end if;

      Mag := 1;
      while Num_Abs / Mag >= Base loop
         Mag := Mag * Base;
      end loop;
```

```
      if Is_Negative then
         C := '-';
         Output(String'(1 => C));
      end if;

      loop
         Digit := Num_Abs / Mag;
         if Digit <= 9 then
            C := Character'Val (Character'Pos ('0') + Digit);
         else
            C := Character'Val (Character'Pos ('a') + Digit - 10);
         end if;

         Output(String'(1 => C));

         exit when Mag = 1;

         Num_Abs := Num_Abs - Digit * Mag;
         Mag := Mag / Base;
      end loop;
   end Put;

   procedure Put_Line (Str : in String) is
   begin
      Put(Str);
      New_Line;
   end Put_Line;

   -----------
   -- Error --
   -----------
   --
   --  MaRTE_Internal_Error
   procedure Error (Msg   : in String;
                    Fatal : in Boolean := False) is
   begin
      Put_Error (Str_Internal_Error);
      Put_Error (Msg, Fatal);
   end Error;

   ---------------
   -- Put_Error --
   ---------------
   procedure Put_Error (Msg : in String; Fatal : in Boolean := False)
       is
      procedure Exit_Process (Status : in Interfaces.C.int);
      pragma Import (C, Exit_Process, "exit");
   begin
      Output_Error(Msg);
      if Fatal then
         Output_Error(Str_Fatal_Error);
         -- Exit process if fatal
         Exit_Process(1);
      end if;
   end Put_Error;

end Stm32f4_TextIO;
```

Figura A.2: stm32f4_textIO.adb: Paquete wrapper para salida por consola

```c
/*
 * uart_console.c
 *
 *  Created on: Nov 17, 2021
 *      Author: Kunal
 *      Modified by: Juan Román Peña
 *
 * This file has been simplified to be part of the M2OS project
    implementation.
 * Uses USART2: RX->PA2, TX->PA3
 */

#include "uart_console.h"
#include "GPIO.h"
#include "stm32f407xx.h"
#include "stm32f4xx.h"
#include "system_stm32f4xx.h"

USART_TypeDef *port = USART2;

void
uart_console_init (int baudrate)
{
  RCC->APB1ENR |= RCC_APB1ENR_USART2EN;
  GPIO_Pin_Setup ('A', 2, ALTERNATE_FUNCTION_OUTPUT_PUSHPULL, USART2_TX
     );
  GPIO_Pin_Setup ('A', 3, ALTERNATE_FUNCTION_OUTPUT_OPENDRAIN,
     USART2_RX);
  port->CR1 |= USART_CR1_UE;

  // Correct baud rate calculation for STM32F4
  // For USART2 on APB1: BRR = fCK / baudrate
  // Where fCK is the APB1 clock (typically SystemCoreClock for default
      config)
  port->BRR = SystemCoreClock / baudrate;

  port->CR1 |= USART_CR1_TE;
  port->CR1 |= USART_CR1_RE;
}

void
uart_print_console (char *msg)
{
  while (*msg != '\0')
    {
      port->DR = *msg++;
      while (!(port->SR & USART_SR_TXE))
        ;
    }
}

void
uart_console_putchar (char c)
{
  while (!(port->SR & USART_SR_TXE)) {
  }

  port->DR = c;
}
```

```
// Function to write multiple bytes to UART console
// Compatible with MaRTE Direct_IO write function signature
int uart_console_write(int fd, const void *buffer, size_t bytes) {
  const char *buf = (const char *)buffer;
  size_t i;

  for (i = 0; i < bytes; i++) {
    uart_console_putchar(buf[i]);
  }

  return (int)bytes; // Return number of bytes written
}

void uart_print_error(const char *msg) {
  char* error_header = "ERROR: ";
  uart_print_console(error_header);
  uart_print_console((char *)msg);
  uart_print_console("\n");

}
```

Figura A.3: uart_console.c: Código para la comunicación por puerto UART @author Kunal

A.4. Repositorio con el código del proyecto

Se añade el repositorio donde se encuentra almacenado el código de todas las fases del desarrollo del RTS

`https://github.com/iescalada884/TFG_RTS`

Existen varias Realeses dentro del mismo con hitos importantes del mismo.

Si se desea una usabilidad más cómoda y enfocada al RTS en lugar del proyecto completo(donde se incluyen los scripts y archivos de investigaciones realizadas). Se dispone del script `install.sh` en el directorio `utility_Scripts`, este script cuenta con la opción **-h, –help** para mostrar una ayuda sobre su uso.

Con el script se puede instalar el RTS con MaRTEOS de una release concreta, además cuenta con funciones para ver un registro de las distintas releases publicadas.