TRABAJO PRÁCTICO PROFESIONAL

INGENIERÍA ELECTRÓNICA

Desarrollo e Implementación de una Computadora de Abordo para un Satélite CubeSat, utilizando un Sistema Operativo en Tiempo Real y Bibliotecas de Código Abierto

Autores:
Facundo Nicolás Arballo
Faustino Ferrari
Francisco Spaltro

Director:
Ing. Joaquin Gaspar Ulloa

Codirector:
Ing. Sebastián García Marra

Jurados:
Ing. Martín Echarri
Ing. Fernando V. Filippetti
Ing. Ricardo A. Veiga

Ciudad Autónoma de Buenos Aires
Mayo de 2025

Resumen

Este trabajo presenta el desarrollo de un sistema embebido para una computadora de abordo (OBC) basada en un microcontrolador STM32, capaz de gestionar sensores, memorias y comunicaciones en un entorno satelital simulado. Se integraron periféricos mediante UART, SPI e I^2C, y se llevaron a cabo pruebas funcionales mediante un entorno de validación automatizado. Además, se implementó una interfaz gráfica para el monitoreo y control remoto del sistema desde una Raspberry Pi.

Agradecimientos

Agradezco sinceramente a todas aquellas personas que formaron parte de este camino y me acompañaron a lo largo de este proceso.

A mi familia, gracias por ser mi base y mi mayor apoyo. Su soporte incondicional, su paciencia y su confianza fueron lo que me impulsó en los momentos más difíciles.

A mis compañeros y colegas, por estar presentes en los desafíos, los logros y las horas de trabajo. Su apoyo y amistad los volvió una parte fundamental de mi vida.

A mis amigos y personas más cercanas, gracias por estar presentes, por comprender mis ausencias y por brindar siempre ánimo o un momento de distracción cuando más necesité.

A mis tutores, gracias por su guía, su tiempo y sus aportes. Su experiencia y dedicación fueron clave para el desarrollo y finalización de este proyecto.

A mi abuelo Hugo, que desde donde esté, sé que se siente orgulloso y que más que nadie habría querido acompañarme hoy.

A todos gracias por estar. Este trabajo también es su logro.

Facundo Nicolás Arballo

Quiero agradecer profundamente a quienes me acompañaron durante todo este recorrido.

A mis tutores, por estar siempre presentes, brindando su ayuda y compromiso en cada etapa del proyecto.

A mis compañeros de trabajo, por ser un apoyo constante, por compartir momentos de entusiasmo y motivación que hicieron más llevadero este proceso.

A mis amigos, porque sin su compañía y presencia incondicional, este camino hubiese sido mucho más difícil. Gracias por estar siempre, incluso en los momentos más complejos.

Y, por último, a mi familia, que me apoyó en cada idea, proyecto o situación que surgió. Gracias por estar ahí en todo momento. Este logro también es de ustedes.

Faustino Ferrari

A María Silvia, mi madre, por su amor inagotable, su apoyo incondicional y por enseñarme a perseguir siempre mis sueños.

A Alfredo, mi padre, por su fe en mí y por enseñarme el don de la perseverancia.

A ustedes les debo lo que soy y lo que puedo llegar a ser.

A Juli y Lelé, mis hermanas, por haberme acompañado (y aguantado) todos estos años.

A Mili, mi novia y compañera, por la paciencia y el amor.

A toda mi familia, por acompañarme incluso desde lejos, por su cariño constante y por hacerme sentir cerca a pesar de la distancia.

A Facu y Fausti, mis compañeros en este trabajo, y a Nacho, por ser los tres parte fundamental de este camino.

A mis amigos, por sostenerme siempre.

A todos los profesores que me apoyaron y cultivaron en mí la pasión por la docencia.

A Joaco y Seba, por su guía: no podríamos haber elegido mejores directores.

A la Facultad de Ingeniería de la Universidad de Buenos Aires, educación pública, gratuita y de calidad, por haberme formado y haberse convertido en mi segundo hogar.

A todos ustedes: **gracias totales**.

Francisco Spaltro

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

En los últimos años, los satélites de tipo CubeSat han ganado protagonismo como plataforma accesible y versátil para el desarrollo de tecnología espacial. Su bajo costo relativo, combinación con *hardware* de código abierto y posibilidad de integración en programas educativos los convierte en una herramienta clave tanto para la formación de ingenieros como para la investigación aplicada.

En este contexto, la Facultad de Ingeniería de la Universidad de Buenos Aires impulsa el desarrollo del proyecto ASTAR, una iniciativa institucional cuyo objetivo es diseñar y construir un CubeSat desde el ámbito universitario. En el marco de esta iniciativa, el presente trabajo se enfoca en el desarrollo de un subsistema fundamental: la Computadora de Abordo (OBC, por sus siglas en inglés, *On-Board Computer*), encargada de coordinar el funcionamiento del satélite.

Dentro del proyecto ASTAR, nuestro desarrollo comenzó desde cero, sin un marco preexistente para guiar la implementación de la OBC. A lo largo del trabajo, definimos progresivamente la arquitectura, seleccionamos las herramientas adecuadas y estructuramos un *framework* que permitiera estandarizar y optimizar el desarrollo. El objetivo no fue solo construir una solución funcional para esta etapa del satélite, sino también establecer una base sólida que facilite futuras iteraciones. La modularidad, la documentación detallada y la integración de herramientas de pruebas automatizadas aseguran que este trabajo pueda ser ampliado y reutilizado en las siguientes fases de ASTAR, contribuyendo a su evolución y mejora continua.

1.1. Estándar Cubesat

Un CubeSat es un tipo de satélite miniaturizado que sigue un estándar definido originalmente en 1999 por la Universidad Politécnica del Estado de California (Cal Poly) y la Universidad de Stanford. Este estándar fue creado con el objetivo de facilitar el acceso al espacio, particularmente para instituciones educativas, mediante una plataforma de bajo costo y alta estandarización.

El formato más básico es el de 1 U (una unidad), que corresponde a un cubo de 10×10×10 cm y una masa de entre 1 y $1,33$ kg. A partir de esta unidad se han estandarizado tamaños mayores como $1,5$ U, 2 U, 3 U, 6 U y hasta 12 U, permitiendo mayor flexibilidad en la carga útil y subsistemas del satélite [1].

La estandarización del tamaño y la interfaz mecánica ha permitido el desarrollo de componentes comerciales reutilizables, reduciendo costos y complejidad de integración. Además, existen sistemas dispensadores compatibles que permiten lanzar múltiples CubeSats de forma segura y eficiente como cargas secundarias en lanzamientos principales, o incluso desde la Estación Espacial Internacional (ISS, del inglés *International Space Station*).

Su uso se ha expandido más allá del ámbito académico: agencias gubernamentales, empresas privadas y consorcios internacionales han adoptado el formato CubeSat para misiones científicas, tecnológicas, educativas e incluso comerciales. La reducción de costos, el

diseño modular y la facilidad para iterar en ciclos cortos han convertido a los CubeSats en una herramienta clave para la innovación en el espacio.

Un CubeSat típico está conformado por varios subsistemas que trabajan de manera conjunta para cumplir los objetivos de la misión. Entre los módulos principales se encuentran:

- **Estructura:** proporciona soporte mecánico y protege los componentes internos. Debe cumplir con las dimensiones estándar del formato U.
- **Energía:** incluye paneles solares, reguladores de tensión y baterías. Es responsable de generar, almacenar y distribuir energía eléctrica.
- **OBC:** coordina la operación general del satélite. Ejecuta las tareas críticas, administra la comunicación con sensores y sistemas, y toma decisiones en base a eventos.
- **Subsistema de comunicaciones:** se encarga del envío y recepción de datos hacia y desde la estación terrestre. Puede incluir radios UHF/VHF o bandas más altas como S-band.
- **Control de actitud y órbita (ADCS):** mide y ajusta la orientación del satélite mediante sensores (como giroscopios y magnetómetros) y actuadores (como ruedas de reacción y *magnetorquers*).
- **Carga útil:** corresponde al conjunto de instrumentos o experimentos que justifican la misión. Puede ser una cámara, un sensor especializado o un módulo de comunicaciones.

Cada uno de estos módulos debe integrarse de forma compacta y eficiente, considerando las severas restricciones de espacio, peso y consumo propias del formato CubeSat.

1.2. Objetivos

El objetivo general de este trabajo es diseñar e implementar un prototipo funcional de la OBC para un satélite tipo CubeSat. Esta OBC debe ser capaz de interactuar con sensores y periféricos utilizando protocolos de comunicación ampliamente empleados en sistemas similares, como UART, I^2C y SPI. Además, debe gestionar tareas concurrentes en tiempo real y permitir su verificación mediante un entorno automatizado.

Entre los objetivos particulares se incluyen:

- La selección del microcontrolador y las bibliotecas de *software* adecuadas.
- El diseño e implementación de controladores para la comunicación con sensores comunes en entornos satelitales (GPS, sensores de temperatura, humedad, acelerómetros).
- La integración de estas funciones bajo FreeRTOS, asegurando una arquitectura robusta y escalable.
- La construcción de un entorno de pruebas basado en integración continua (CI/CD), utilizando herramientas actuales como Docker y GitLab CI.
- La generación de una documentación exhaustiva que permita la replicabilidad y evolución del sistema.

1.3. Alcance del proyecto

El alcance de este trabajo incluye el desarrollo completo del *software* embebido de la OBC, junto con su validación mediante pruebas unitarias e integrales. Es importante destacar que todas las pruebas del *software* desarrollado se ejecutaron sobre ***hardware* real**, mediante un sistema de *testing* automatizado que permite verificar su funcionamiento en condiciones controladas y reproducibles. El sistema debe gestionar sensores a través de *buses* de comunicación, almacenar datos en memorias no volátiles y transmitir información por interfaces serie.

Capítulo 2

Arquitectura

El diseño de la arquitectura de la OBC constituye un eje central en el desarrollo de un CubeSat. Para ello, deben considerarse diversas reglas de diseño relacionadas con el consumo energético, la robustez frente a condiciones adversas del entorno espacial, la capacidad de procesamiento y la compatibilidad con los demás subsistemas del satélite. A fin de abordar estos desafíos, se definieron una serie de requisitos mínimos, tomando como referencia experiencias previas y lineamientos técnicos extraídos de proyectos satelitales universitarios y colaborativos.

En cuanto a la elección del microcontrolador, se buscó un dispositivo que cumpliera con los requisitos funcionales clave en entornos espaciales, como bajo consumo, amplio soporte de *buses* de comunicación, buena disponibilidad de herramientas de desarrollo y, especialmente, pertenencia a una familia de microcontroladores que incluya versiones con tolerancia a radiación. Esta decisión permite realizar pruebas de desarrollo en dispositivos más económicos y fácilmente accesibles, sin comprometer la posibilidad de extrapolar el diseño a modelos aptos para vuelo.

Respecto a los protocolos de comunicación, se analizaron las interfaces más utilizadas en misiones CubeSat, priorizando estándares robustos y ampliamente soportados en componentes comerciales, con especial atención a aquellos puntos donde suelen producirse fallas. Esto permitió definir una arquitectura de comunicación tolerante a fallos, interoperable con múltiples dispositivos y adecuada al ecosistema del satélite.

Para la gestión del *software*, se evaluaron diferentes alternativas de implementación, incluyendo programación en *bare metal*, sistemas operativos de tiempo real (RTOS) y sistemas embebidos basados en Linux. Cada enfoque presenta ventajas y limitaciones en cuanto a consumo de recursos, complejidad, portabilidad y capacidad de respuesta ante eventos en tiempo real. Asimismo, se consideró el uso de bibliotecas de código abierto, no solo por su disponibilidad y comunidad activa, sino también por su alineación con el espíritu colaborativo de los proyectos universitarios, facilitando la transferencia de conocimientos y la evolución del sistema en futuras iteraciones.

2.1. Selección del microcontrolador

2.1.1. Entorno espacial

La órbita baja terrestre (LEO, por sus siglas en inglés), que se extiende entre los 160 km y los 2.000 km de altitud, ha sido históricamente la región más utilizada para la operación de CubeSats [2]. Aunque su acceso resulta relativamente sencillo en comparación con otras órbitas, el entorno espacial en LEO presenta una serie de desafíos ambientales que deben ser cuidadosamente considerados al seleccionar los componentes de una plataforma satelital.

Según Lu et al. (2019) [3], los factores más relevantes son los siguientes:

- Oxígeno atómico (AO): presente principalmente entre los 200 km y los 800 km de altitud, este elemento altamente reactivo puede deteriorar las superficies de los materiales del satélite al generar erosión a nivel microscópico debido a impactos de alta energía.
- Plasma ionosférico: puede provocar la acumulación de carga eléctrica en la superficie del satélite, dando lugar a descargas electrostáticas no deseadas, pérdida de eficiencia en paneles solares y errores en la transmisión de señales.
- Radiación ionizante: tanto los protones atrapados en el cinturón de radiación interno como los eventos solares pueden causar efectos como errores lógicos (SEU), disparo de corrientes anómalas (SEL) y deterioro gradual de componentes electrónicos (TID).
- Partículas macroscópicas (basura espacial y micrometeoritos): debido a la alta densidad de objetos orbitando en LEO, existe un riesgo considerable de colisiones a velocidades extremadamente altas, que pueden comprometer la integridad del satélite.
- Campo geomagnético: si bien ofrece cierta protección frente a partículas solares, su interacción con el plasma puede complejizar el entorno electromagnético, especialmente en regiones cercanas a los polos.
- Variaciones térmicas extremas: la constante alternancia entre luz solar directa y sombra genera oscilaciones térmicas significativas (desde –101 °C hasta +93 °C), lo que puede afectar la estabilidad estructural y funcional del satélite.

Algunas de estas condiciones ambientales afectan principalmente a elementos estructurales y superficiales del satélite, como los micrometeoroides o el oxígeno atómico, que degradan recubrimientos y materiales expuestos. Otros factores como el plasma ionosférico impactan sobre los arreglos solares y superficies cargadas, mientras que el campo geomagnético influye especialmente en la calibración de sensores sensibles, como magnetómetros o instrumentos de navegación. Además, algunas de estas amenazas requieren sistemas de defensa específicos, como escudos o estrategias de evasión basadas en simulaciones orbitales.

Por estas razones, en la elección del microcontrolador nos centraremos en aquellos factores que inciden directamente sobre su funcionamiento eléctrico y lógico, principalmente los niveles de radiación ionizante y variaciones térmicas que el dispositivo debe ser capaz de soportar en condiciones reales de operación en LEO.

2.1.2. Radiación ionizante

La radiación ionizante en LEO puede provocar diferentes efectos sobre los componentes electrónicos, particularmente en microcontroladores que forman parte de la OBC. Estos efectos pueden clasificarse en dos grandes categorías: acumulativos y transitorios [4].

Los efectos acumulativos incluyen la dosis total de radiación ionizante (TID, por sus siglas en inglés), que se mide en Gy (Gray) y representa la cantidad de energía ionizante absorbida por unidad de masa del material (joules por kilogramo). Esta dosis puede causar degradación funcional o paramétrica en el microcontrolador, como desplazamientos en el umbral de tensión, aumento de las corrientes de fuga o pérdida gradual de funcionalidad. En la figura 2.1, se puede observar que, a la izquierda, se modelan los efectos inducidos por radiación en un transistor MOS, representando tanto el desplazamiento del voltaje umbral (V_T) como la aparición de una corriente de fuga entre *drain* y *source*. A la derecha, se muestra la curva característica I_{DS} en función de V_{GS}, donde se evidencia un corrimiento de las curvas tras la exposición a una dosis de radiación, lo cual indica un aumento en el voltaje

umbral y un incremento en la corriente de fuga en la región subumbral. Simulaciones realizadas mediante herramientas como SPENVIS[1] indican que, para órbitas típicas de CubeSats en LEO y con blindaje mínimo (1 mm de aluminio), la TID acumulada es de $17,2$ Gy durante un año, 45 Gy durante tres años y $67,3$ Gy durante 5 años [5].

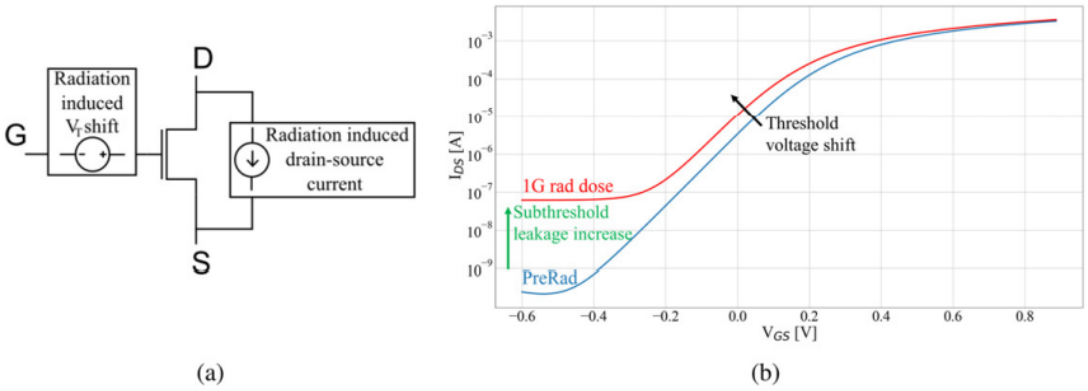

FIGURA 2.1. Desplazamiento del umbral de tensión en un MOS producto de la TID

Por otro lado, los efectos transitorios o eventos de partícula única (SEE, del inglés *Single Event Effects*) comprenden fenómenos que ocurren cuando una partícula individual, como un protón o un ion pesado, impacta directamente sobre una región sensible del circuito. Entre ellos se destacan dos tipos principales: los *Single Event Upsets* (SEU), que corresponden a cambios lógicos no destructivos en celdas de memoria, y los *Single Event Latch-up* (SEL), que ocurren cuando una partícula cargada activa componentes parásitos presentes en el sustrato, tal como sucede con los TBJs que se forman en la celda que ilustra la figura 2.2, generando un cortocircuito que puede inducir estados de alta corriente destructivos si no se detecta y controla a tiempo.

En el caso de los SEU, la mitigación se basa principalmente en técnicas de diseño de *firmware* y arquitectura del sistema, incluyendo la implementación de *watchdog timers* para recuperación automática, uso de códigos de corrección de errores (ECC, por sus siglas en inglés), verificaciones cíclicas (CRC, por sus siglas en inglés), y redundancia lógica. Para los SEL, que representan un riesgo mayor al poder comprometer permanentemente el funcionamiento del microcontrolador, se emplean supervisores de corriente externos, monitoreo de consumo, o reinicios forzados mediante *hardware* en caso de detección de estados anómalos.

2.1.3. Temperatura

El entorno térmico en la órbita baja terrestre presenta variaciones extremas de temperatura debido a la alternancia cíclica entre la exposición directa a la radiación solar y el paso por regiones de sombra orbital. Este ciclo ocurre aproximadamente cada 90 minutos [6], que es el período orbital típico de un satélite en LEO, lo que intensifica los efectos térmicos sobre la plataforma, al producir transiciones bruscas y frecuentes entre condiciones de iluminación y oscuridad. Estas oscilaciones pueden generar gradientes térmicos muy pronunciados en las superficies externas del satélite, alcanzando diferencias de temperatura de hasta –101 °C a +93 °C [3].

[1] del ingles *Space Environment Information System*, es un *software* operativo desarrollado por la Agencia Espacial Europea (ESA, por sus siglas en inglés) que proporciona una interfaz web para evaluar el entorno espacial y sus efectos sobre sistemas espaciales y tripulaciones.

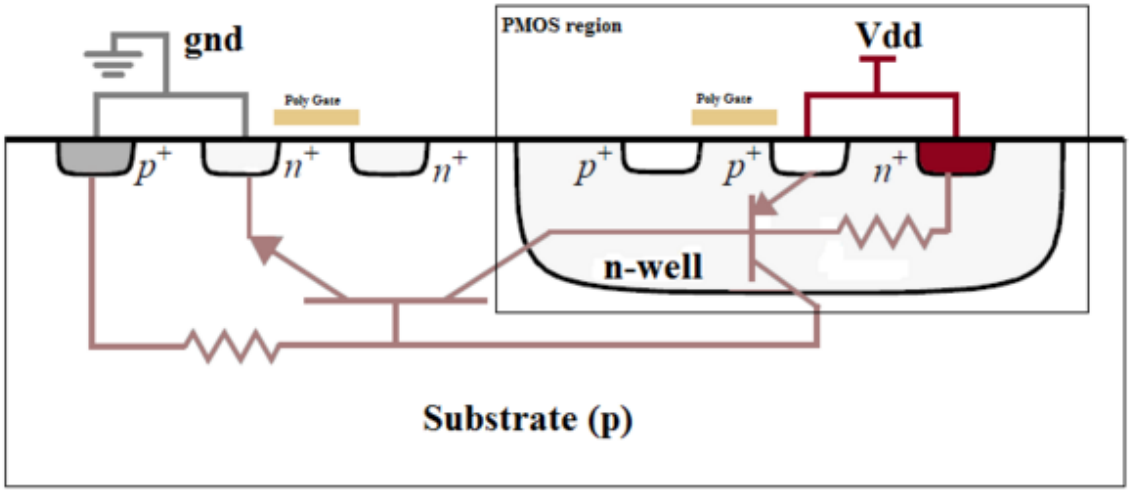

FIGURA 2.2. Componentes parásitos que pueden inducir un evento de *Latch-Up*.

Además, en el entorno espacial no existe convección que permita distribuir el calor de forma uniforme, por lo que el lado del satélite expuesto al Sol puede alcanzar temperaturas muy elevadas, mientras que el lado opuesto permanece extremadamente frío. Esto provoca la existencia de fuertes gradientes térmicos incluso dentro de la estructura del satélite, lo que representa un desafío significativo para el diseño térmico, especialmente en lo que respecta a la selección de materiales, disipación de calor y estabilidad estructural.

No obstante, los sistemas embebidos que forman parte de la OBC suelen ubicarse en el interior del satélite, donde el ambiente térmico se encuentra regulado mediante técnicas de aislamiento, conducción térmica y, en algunos casos, calefacción activa. En estos casos, los componentes electrónicos deben estar preparados para operar normalmente dentro de un rango de –20 °C a +60 °C, y resistir, sin daño irreversible, exposiciones entre –40 °C y +75 °C [7] en condiciones extremas de misión.

2.1.4. Características técnicas del microcontrolador

Uno de los factores más relevantes en el proceso de selección del microcontrolador para la OBC es el consumo energético. Dado que la OBC opera con una fuente limitada de energía —proporcionada por el sistema de generación y almacenamiento del satélite—, es indispensable contar con un procesador eficiente en términos de potencia, sin comprometer sus capacidades funcionales. Como referencia, se ha documentado que una OBC completamente funcional puede requerir alrededor de 1 W en modo activo, con consumos significativamente menores en modos de bajo consumo como *sleep* [8].

Otro criterio fundamental es la capacidad de procesamiento. La OBC debe ejecutar múltiples tareas en tiempo real, tales como la adquisición y almacenamiento de datos, la ejecución de algoritmos de control y la gestión de protocolos de comunicación con los distintos subsistemas. Para ello, se consideró necesario contar con una arquitectura optimizada para operaciones concurrentes, con soporte nativo para interrupciones y temporización precisa, así como un conjunto de instrucciones que permitiera ejecutar código de forma eficiente.

También se evaluó la compatibilidad con interfaces de comunicación estándar, como UART, I^2C y SPI, que permiten la conexión con sensores, actuadores y módulos auxiliares. Adicionalmente, se valoró la posibilidad de expansión mediante memoria externa, un

aspecto clave para el almacenamiento de telemetría, registros de eventos y respaldo de datos.

Otro aspecto importante es la madurez del ecosistema de desarrollo. La disponibilidad de documentación técnica, herramientas de depuración, bibliotecas de código abierto y una comunidad activa facilita las etapas de diseño, validación y mantenimiento. Asimismo, contar con soporte continuo del fabricante y acceso a una familia de dispositivos escalables resulta beneficioso para el ciclo de vida del proyecto.

Con base en estos criterios, se consultaron antecedentes de otras misiones CubeSat. En el estudio *A Survey to Select Microcontroller for Sathyabama Satellite's On Board Computer Subsystem* [9], se comparan diversas arquitecturas y fabricantes en función de su consumo energético, capacidad de cómputo y disponibilidad de periféricos. Dicho trabajo explora múltiples opciones ampliamente documentadas en aplicaciones satelitales educativas, incluyendo microcontroladores de Atmel (AT91SAM7A1, ATmega128), Texas Instruments (MSP430), Freescale (MAC7111, MAC7131) y STMicroelectronics (STM32F1, STM32F2) [9]. Estas alternativas fueron evaluadas con base en sus prestaciones técnicas, disponibilidad en el mercado, experiencia de vuelo previa y compatibilidad con arquitecturas de *software* en tiempo real.

Además, se prestó especial atención a la capacidad de almacenamiento del sistema. En la mayoría de los casos, la memoria interna disponible en los microcontroladores resulta insuficiente para gestionar de forma segura los datos generados por los sensores de *payload*, los registros del sistema y la telemetría. Por ello, los diseños CubeSat suelen incorporar mecanismos de expansión externa mediante *buses* como SPI o I^2C, permitiendo almacenar no solo datos de misión, sino también versiones del *software* de vuelo enviadas desde Tierra. Esta capacidad resulta fundamental para implementar esquemas de actualización remota (OTA, del ingles *Over-the-air*) y para contar con copias de respaldo del *firmware* que permitan recuperar el sistema en caso de fallos.

La arquitectura de memoria asociada a la OBC cumple, en este contexto, funciones críticas: preservar información operativa, retener datos científicos y facilitar la actualización remota del *firmware*. Lumbwe [8] recomienda que un microcontrolador adecuado para este tipo de sistema debe contar con una cantidad significativa de memoria interna para código, y ofrecer una arquitectura escalable que permita integrar memorias externas de manera fiable. Esta flexibilidad es esencial para adaptarse a distintos perfiles de misión y asegurar la tolerancia a fallos mediante redundancia en el almacenamiento.

A su vez, se ha documentado una transición progresiva desde microcontroladores de 8 y 16 bits hacia arquitecturas de 32 bits, motivada por la creciente demanda de procesamiento, la eficiencia alcanzada en términos de consumo energético y la posibilidad de reutilizar código en lenguajes de alto nivel. En trabajos previos se señala que, si bien en los primeros CubeSats predominaban dispositivos de menor capacidad, el avance tecnológico y la aparición de aplicaciones más exigentes han impulsado la adopción de núcleos más potentes, como los ARM Cortex-M. Incluso se destaca que, en misiones como NCube2 y CanX-1, ya se implementaron microcontroladores de 32 bits con éxito en tareas críticas de comando y manejo de datos, utilizando configuraciones con hasta 512 kB de SRAM y 32 MB de memoria flash externa.

2.1.5. Elección del microcontrolador

Teniendo en cuenta los requerimientos funcionales de la OBC, las condiciones ambientales de la órbita baja terrestre y los criterios de selección previamente expuestos, se optó por utilizar un microcontrolador ARM Cortex-M3, en particular el modelo STM32F103C8T6 fabricado por STMicroelectronics. Esta elección se fundamenta en múltiples factores.

En primer lugar, este microcontrolador ofrece una arquitectura de 32 bits, ampliamente utilizada en sistemas embebidos con requerimientos de tiempo real. También presenta un equilibrio favorable entre capacidad de procesamiento (72 MHz) y bajo consumo energético.

En segundo lugar, cuenta con un ecosistema de desarrollo maduro, con abundante documentación, herramientas accesibles y soporte comunitario activo, lo que facilita tanto el desarrollo inicial como la depuración, pruebas y mantenimiento del sistema.

En cuanto a recursos internos, el microcontrolador cuenta con 64 kB de memoria flash para almacenamiento de programa y 20 kB de SRAM para datos. Aunque no dispone de memoria EEPROM integrada, su compatibilidad con *buses* SPI e I^2C permite una expansión flexible mediante memorias externas no volátiles.

Respecto al consumo energético, el dispositivo opera típicamente a $3,3$ V, con una corriente aproximada de 50 mA en modo de ejecución a 72 MHz, a 105 °C, con *clock* externo y todos los periféricos encendidos, lo que implica un consumo de potencia de 165 mW. En modos de bajo consumo, el STM32F1 reduce significativamente sus requerimientos: en modo *sleep* con todos los periféricos desactivados, el consumo desciende a $26,4$ mW [10].

Desde el punto de vista ambiental, el STM32F1 fue sometido a ensayos de radiación ionizante total (TID) en condiciones representativas de una misión en órbita baja (LEO), con un blindaje de 1 mm de aluminio. En estos ensayos, el microcontrolador se mantuvo funcional durante más de 5 años de exposición simulada. A medida que se acumula la dosis de radiación, comienzan a observarse pequeñas degradaciones en la frecuencia de reloj y en la memoria flash, hasta fallar al alcanzar una dosis acumulada de 1070 Gy. Estos resultados respaldan su idoneidad para entornos espaciales con niveles moderados de radiación.

En cuanto a su tolerancia térmica, el microcontrolador está especificado para operar de forma estable en un rango de –40 °C a +85 °C en su versión estándar, correspondiente al grado industrial. Asimismo, existen versiones de la misma familia que extienden este rango hasta +105 °C, lo que ofrece un margen adicional para aplicaciones con condiciones térmicas más exigentes. Estas temperaturas se encuentran dentro del rango descrito anteriormente.

Además, es importante destacar que el STM32F103 posee herencia de vuelo documentada. En el informe *State-of-the-Art of Small Spacecraft Avionics* [4] publicado por NASA en 2024, se detalla que este modelo fue utilizado como núcleo del sistema CDH-110 desarrollado por *Berlin Space Technologies*, validado en misiones reales en órbita baja y con tolerancia a radiación comprobada hasta 300 Gy a nivel de placa de circuito impreso. Esta evidencia refuerza la viabilidad del STM32F1 en contextos espaciales, incluso más allá del uso estrictamente educativo.

Finalmente, esta elección responde a una estrategia de diseño orientada al prototipado accesible y escalable. El STM32F103C8T6 es una opción de bajo costo y amplia disponibilidad, ideal para el desarrollo inicial, validación funcional y ensayos en banco. Posteriormente, el diseño puede ser migrado hacia versiones radiotolerantes o encapsuladas de la misma familia ARM Cortex-M, aprovechando la continuidad de arquitectura y compatibilidad a nivel de código, lo que simplifica el proceso de transición hacia entornos más exigentes sin rediseños sustanciales.

2.1.6. Biblioteca utilizada para el desarrollo

En el contexto de un proyecto educativo basado en código abierto, se adoptó la decisión estratégica de prescindir de bibliotecas propietarias, como la HAL de STM32Cube, así como de entornos de desarrollo integrado (IDEs, por sus siglas en inglés) específicos de fabricantes. Esta determinación se fundamenta en los objetivos de lograr un conocimiento profundo del *hardware* a nivel de registro, mantener un control absoluto sobre la inicialización y gestión de periféricos, y promover el uso de herramientas de *software* libre multiplataforma.

Por este motivo, se seleccionó una biblioteca de bajo nivel que facilitara el acceso directo a los registros del microcontrolador, evitando capas de abstracción que pudieran enmascarar aspectos críticos de configuración. En este escenario, **libopencm3** emergió como la solución idónea, al ofrecer un equilibrio entre control de *hardware* y portabilidad, manteniendo la adhesión a los principios de código abierto y transparencia técnica.

Libopencm3 se presenta como biblioteca de *hardware* de bajo nivel de código abierto para microcontroladores ARM Cortex-M3 (aunque también se admiten M0 y M4). Este proyecto, antes conocido como **libopenstm32**, tiene como objetivo crear una biblioteca de *firmware* libre o de código abierto (LGPL v3, o posterior) para varios microcontroladores ARM Cortex-M0(+)/M3/M4, incluyendo ST STM32, Ti Tiva y Stellaris, NXP LPC 11xx, 13xx, 15xx, 17xx, Atmel SAM3, Energy Micro EFM32 y otros [11].

Un factor determinante en la adopción de dicha biblioteca es su escalabilidad y flexibilidad. Al ofrecer soporte unificado para múltiples familias de microcontroladores Cortex-M, la biblioteca permite la migración eficiente de proyectos entre dispositivos de diferentes capacidades (por ejemplo, desde un STM32F1xx hacia un STM32F4xx o STM32H7xx). Esta portabilidad se consigue mediante simples ajustes en las configuraciones de compilación y asignación de pines, preservando la lógica de control fundamental sin requerir modificaciones sustanciales en el código base.

Como recurso fundamental para la implementación de esta biblioteca, el libro *Beginning STM32: Developing with FreeRTOS, libopencm3 and GCC* de Warren Gay [12] resultó invaluable, al proporcionar un marco de referencia completo para el desarrollo con STM32. La obra no solo facilitó la comprensión de los fundamentos arquitectónicos y el uso del *toolchain* GCC, sino que también cubrió aspectos prácticos clave como la programación de los microcontroladores mediante OpenOCD y la integración eficiente de FreeRTOS en el flujo de trabajo. Su enfoque didáctico permitió trasladar los conceptos teóricos a implementaciones concretas, optimizando significativamente el proceso de desarrollo.

2.2. Configuración

La arquitectura de la OBC se organizó en torno a un esquema centralizado, con un microcontrolador principal que actúa como nodo maestro encargado de gestionar tanto la adquisición de datos como las comunicaciones con el entorno interno y externo del satélite. La selección de los periféricos y de los protocolos de comunicación se realizó en función de criterios de eficiencia, compatibilidad, robustez y disponibilidad de controladores en la arquitectura seleccionada (STM32F103C8T6).

2.2.1. Protocolo UART

El *bus* UART se empleó principalmente por su simplicidad de implementación, bajo consumo de recursos y su amplia compatibilidad con módulos externos. Este protocolo cumple con cuatro funciones críticas dentro del sistema:

- **Comunicación con la estación terrestre**: mediante un enlace con un microcontrolador secundario o transceptor dedicado, UART permite transmitir y recibir datos de telemetría y comandos en tiempo real. Esta solución fue consultada con el equipo encargado del desarrollo del sistema de comunicaciones, a fin de asegurar compatibilidad y simplicidad en la integración.
- **Actualización del *firmware***: se habilita la reprogramación del sistema en vuelo a través de una línea UART, utilizando un *bootloader* compatible.

- **Adquisición de datos de GPS**: el receptor GPS utilizado transmite su información en formato NMEA mediante UART, lo que permite su integración directa con el microcontrolador.
- **Interfaz de prueba con Raspberry Pi**: durante la etapa de validación funcional en banco, UART proporciona una vía directa de comunicación con sistemas de análisis y validación como la Raspberry Pi.

2.2.2. Protocolo I²C

El *bus* I²C fue seleccionado por su capacidad de conectar una gran cantidad de dispositivos utilizando únicamente dos líneas físicas (SDA y SCL), lo cual resulta especialmente ventajoso en sistemas embebidos donde los recursos de pines y espacio son limitados. Esta topología permite integrar de forma sencilla múltiples sensores y periféricos sin complejidades adicionales en el cableado.

Este protocolo es ideal para incorporar sensores con bajo requerimiento de ancho de banda, como aquellos utilizados para monitorear variables ambientales y de orientación del satélite. En particular, permite conectar:

- **Sensores de temperatura y humedad**, como el HTU21D
- **Sensores de aceleración y giro**, como la MPU6050

Además de sensores, el *bus* I²C admite la integración de **memorias no volátiles** como las EEPROM (por ejemplo, AT24C256), que pueden emplearse para almacenar configuraciones críticas, registros de eventos o datos persistentes del sistema. Aunque su uso específico puede variar según los requerimientos de misión, estas memorias representan una opción confiable para el almacenamiento de bajo volumen con mínima demanda energética.

2.2.3. Protocolo SPI

Para dispositivos que requieren mayor velocidad de transferencia o almacenamiento de grandes volúmenes de datos, se recurrió al *bus* SPI, que ofrece mayor rendimiento que los protocolos mencionados anteriormente, a costa de requerir más líneas físicas por dispositivo esclavo.

Los periféricos conectados mediante SPI incluyen:

- **Tarjeta SD**: destinada al almacenamiento de telemetría, datos continuos de sensores y eventos de misión, utilizando un sistema de archivos FAT/exFAT accesible desde tierra. Si bien su uso requiere inicialización y manejo del sistema de archivos, su alta capacidad de almacenamiento lo justifica.
- **Memoria NOR SPI (Winbond W25Q)**: empleada como *buffer* circular de alta velocidad o almacenamiento redundante para datos binarios en bruto. Su uso permite almacenar información crítica incluso cuando la tarjeta SD no se encuentra montada, y presenta ventajas en términos de velocidad y resistencia ante apagones frente a sistemas basados en archivos.

2.2.4. Otros recursos disponibles del microcontrolador

Además de los *buses* de comunicación utilizados para vincular periféricos y módulos de almacenamiento, el microcontrolador STM32F 03C8T6 ofrece una serie de recursos adicionales que amplían sus capacidades de control y adquisición, permitiendo futuras expansiones del sistema.

- **Entradas/salidas digitales (GPIO)**: el microcontrolador dispone de varias líneas GPIO programables, de las cuales muchas permanecen disponibles tras la asignación de los *buses* principales. Estas pueden ser utilizadas para futuras integraciones como control de actuadores, lectura de pulsadores o interconexión con nuevos sensores digitales.
- **Conversores analógico-digitales (ADC)**: el STM32F1 cuenta con hasta 10 canales ADC de 12 bits, aptos para la digitalización de señales analógicas. Estos permiten incorporar sensores de tensión, monitorear el nivel de batería o realizar diagnósticos internos como lectura de temperatura de referencia o medición de voltajes críticos.
- **Temporizadores (*timers*)**: el microcontrolador incluye múltiples temporizadores de propósito general y específicos, que pueden emplearse en tareas como la generación de señales PWM, temporización de eventos, captura de tiempos de vuelo, o control de servomecanismos.
- **Interrupciones externas (EXTI)**: ofrece líneas dedicadas para responder de forma inmediata a eventos físicos, como flancos de señal provenientes de sensores, pulsadores o interrupciones generadas por otros módulos.
- **Reloj de tiempo real (RTC)**: el STM32F103 incorpora un RTC de 32 bits que mantiene un contador de segundos, respaldado por el pin VBAT para conservar el tiempo aun durante reinicios o modos de bajo consumo. Aunque no dispone de registros específicos de fecha (día, mes, año), estos valores pueden calcularse por *software* a partir del número de segundos acumulado, permitiendo registrar eventos en forma temporal o mantener sincronización relativa dentro de la misión.

La disponibilidad de estos recursos sin utilizar en la configuración actual garantiza una arquitectura flexible y modular, capaz de adaptarse a futuras etapas de desarrollo, integración de subsistemas adicionales o pruebas en condiciones específicas de misión.

2.2.5. *Watchdog*

En entornos embebidos críticos como los sistemas de control de un CubeSat, resulta esencial garantizar la capacidad del sistema de recuperarse de manera autónoma frente a bloqueos o errores de ejecución. Una de las fuentes más comunes de fallos transitorios en el espacio son los Single Event Upsets (SEU), tal como se comentó anteriormente.

Frente a este tipo de fenómenos, una de las primeras líneas de defensa del sistema es la implementación de mecanismos de detección y recuperación automática. En este proyecto, se integró un sistema de vigilancia mediante el watchdog independiente (IWDG) incluido en el microcontrolador STM32F103C8T6.

El IWDG es un temporizador que funciona en paralelo al núcleo principal y es alimentado por un oscilador interno (LSI), lo cual lo hace inmune a los bloqueos del reloj principal y confiable incluso ante errores internos severos. Su función es provocar un reinicio forzado del sistema si este deja de enviar señales de "vida" periódicamente, lo que normalmente ocurriría ante un bloqueo por error de *software* o un SEU que afecte el flujo de ejecución.

La implementación consta de:

- La función `watchdog_setup()`, que inicializa el IWDG y define el período de vigilancia mediante la constante `WATCHDOG_PERIOD`.
- La tarea periódica `taskResetWatchdog()`, ejecutada bajo FreeRTOS, que reinicia el contador del watchdog cada `WATCHDOG_PERIOD/2` milisegundos, asegurando que el sistema reinicie solo en caso de fallos reales.

- La función `check_iwdg_reset_flag()`, que permite detectar si el reinicio anterior fue causado por el *watchdog*, y que limpia la bandera correspondiente en `RCC_CSR`.

2.3. Sistema operativo

El diseño del *software* para la OBC de un CubeSat implica una cuidadosa selección del sistema operativo. Las opciones más comunes incluyen sistemas operativos de tiempo real (RTOS), distribuciones de Linux embebido, y programación *bare-metal*. Cada una de estas alternativas presenta ventajas y limitaciones que deben evaluarse según los requisitos de la misión, los recursos del microcontrolador y la arquitectura general del satélite.

2.3.1. Arquitecturas del *software*

Programación *Bare-metal*

Consiste en programar directamente sobre el *hardware* sin sistema operativo. Esta modalidad se basa en estructuras simples de control como bucles e interrupciones, con una planificación implícita según la estructura del código. El control de los periféricos conectados se realiza mediante *polling* o mediante la gestión de interrupciones, lo que da lugar a una secuencia de ejecución dependiente del flujo del programa y de los eventos externos. Esta estrategia resulta eficiente para resolver problemas simples que implican pocas tareas y no requieren planificación avanzada.

Sin embargo, una de sus principales desventajas es que el código no es escalable. Esto se manifiesta tanto en términos de recursos del sistema como en la capacidad de gestionar aplicaciones más complejas, lo que se refleja en dos tipos de escalabilidad:

- **Escalabilidad vertical:** implica mejorar los recursos de un solo procesador, como usar CPUs más rápidas, mayor RAM o mayor capacidad de almacenamiento.
- **Escalabilidad horizontal:** se refiere al uso de múltiples núcleos, procesadores o sistemas distribuidos para manejar cargas de trabajo crecientes.

Estas limitaciones hacen que la programación *bare-metal* sea poco adecuada para aplicaciones más complejas o con múltiples módulos [13].

Linux embebido

Requiere un procesador que disponga de una Unidad de Gestión de Memoria (MMU por sus siglas en inglés), como ARM Cortex-A. Aunque Linux embebido proporciona un ecosistema robusto de herramientas y controladores, su núcleo (*kernel*) está diseñado para optimizar el rendimiento general mediante la multitarea y el acceso compartido a recursos, en lugar de priorizar tiempos de respuesta estrictos. Como consecuencia, el tiempo de reacción ante eventos externos no es determinístico: la ejecución de tareas puede variar dependiendo de la carga del sistema, el orden de planificación o el manejo interno de interrupciones.

Sistemas operativos de tiempo real

Proporciona una solución intermedia ideal para CubeSats. Incorpora un planificador que permite ejecutar tareas concurrentes con distintos niveles de prioridad, soporte para preempciones[2] , mecanismos de sincronización como semáforos y *mutexes*, y un conjunto de

[2]en inglés, *preemptive*. Se refiere a la capacidad de un sistema operativo de interrumpir una tarea en ejecución para dar paso a otra de mayor prioridad.

APIs que facilitan la gestión eficiente de recursos. Su uso está particularmente recomendado cuando se requiere garantizar comportamiento determinista, respuesta en tiempo real y facilidad de mantenimiento en sistemas que ejecutan múltiples tareas [13].

Ventajas específicas del uso de RTOS

- Garantía de rendimiento en tiempo real.
- Soporte para multitarea y concurrencia mediante planificadores.
- Manejo de comunicación y sincronización entre tareas.
- Facilita la escalabilidad vertical y horizontal del *software*.

2.3.2. Elección del sistema operativo

En el contexto de este proyecto, se optó por utilizar un sistema operativo en tiempo real, específicamente FreeRTOS, como sistema operativo para la OBC debido a una combinación de razones estratégicas, técnicas y operativas que lo posicionan como una solución especialmente adecuada para plataformas de tipo CubeSat con recursos limitados.

Desde un enfoque estratégico, FreeRTOS es un proyecto de código abierto, lo cual permite su integración, modificación y auditoría sin restricciones comerciales. A esto se suma una comunidad activa y extensa, con abundante documentación, ejemplos prácticos y soporte para una amplia gama de plataformas y microcontroladores.

Además, su éxito global se debe a una propuesta de valor clara: FreeRTOS es desarrollado profesionalmente, sometido a rigurosos controles de calidad, robusto, mantenido y completamente libre de ambigüedades sobre propiedad intelectual. Su licencia tipo MIT[3] permite su uso en aplicaciones comerciales sin la necesidad de divulgar código propietario. Desde 2016, se encuentra bajo el respaldo de Amazon Web Services (AWS), que garantiza continuidad, soporte a largo plazo, revisión formal, verificación de seguridad, pruebas de memoria, y una comunidad de desarrollo amplia y diversa. Todo esto se ofrece sin imponer herramientas específicas, prácticas restrictivas o costos, manteniéndose como un proyecto abierto y neutral respecto al *hardware*, herramientas de desarrollo y servicios en la nube [14].

FreeRTOS destaca por su bajo uso de memoria, ocupando típicamente entre 6K y 12K de ROM, y su alta portabilidad entre arquitecturas, lo que lo hace ideal para sistemas basados en ARM Cortex-M como los empleados en misiones espaciales educativas. Su integración con herramientas actuales de desarrollo embebido facilita una curva de aprendizaje accesible, incluso para equipos de ingeniería con recursos humanos limitados. Estas características lo convierten en una alternativa eficaz frente a sistemas operativos más pesados como Linux embebido, eCOS o uCLinux, los cuales requieren mayores recursos computacionales y generalmente una unidad de gestión de memoria, ausente en muchos microcontroladores de bajo consumo [13].

Este sistema operativo permite dividir la aplicación del OBC en tareas independientes que se ejecutan simultáneamente, cada una a cargo de un subsistema o función específica, lo cual resulta crucial para misiones con múltiples sensores, subsistemas de comunicación, almacenamiento y control de actitud. Además, su arquitectura basada en tareas favorece un diseño modular, escalable y desacoplado, ideal para mantener la robustez del sistema ante fallos.

A continuación se detallan algunos beneficios prácticos que refuerzan esta elección:

[3] La licencia MIT (del inglés, *Massachusetts Institute of Technology*) permite usar, modificar, distribuir y copiar el *software* incluso con fines comerciales, requiriendo únicamente conservar el aviso de *copyright* y la licencia en las distribuciones.

- **Eficiencia energética:** Las tareas pueden bloquearse en espera de eventos, evitando el uso continuo del CPU mediante técnicas de *polling*. Esto permite que el microcontrolador entre en modos de bajo consumo (*sleep* o *deep sleep*), en caso de ser requerido, reduciendo significativamente el gasto energético cuando no hay procesamiento activo.
- **Medición de carga del sistema:** La presencia de una tarea *idle*[4] permite estimar el uso real del procesador y tomar decisiones sobre la planificación de tareas o gestión energética.
- **Diseño desacoplado:** El uso de colas, semáforos y otros mecanismos de comunicación entre tareas facilita la separación de responsabilidades y mejora la mantenibilidad del sistema.
- **Reutilización de código:** El enfoque por tareas permite encapsular funciones específicas que pueden ser reutilizadas en otros sistemas o versiones futuras del satélite.
- **Soporte para múltiples esquemas de ejecución:** FreeRTOS admite la combinación de procesamiento periódico, continuo y basado en eventos en un mismo sistema.
- **Manejo eficiente de interrupciones:** El uso de tareas para procesar eventos complejos, delegados desde interrupciones, permite mantener las rutinas de interrupción (ISR) cortas y eficaces, mejorando la latencia y la estabilidad del sistema.

2.3.3. Fundamentos operativos de FreeRTOS

El núcleo de FreeRTOS proporciona un planificador determinista basado en prioridades, que permite ejecutar múltiples tareas de forma concurrente. A pesar de que un procesador de núcleo único solo puede ejecutar una tarea a la vez, el sistema operativo logra una aparente simultaneidad mediante cambios de contexto controlados y rápidos entre tareas. Esta capacidad es esencial en aplicaciones con múltiples funciones paralelas, tales como la adquisición de sensores, comunicaciones, registro en memoria y control de actitud.

En esta sección se describen los principios fundamentales de funcionamiento de FreeRTOS, sus mecanismos de planificación, sincronización y comunicación entre tareas, y su implementación particular en la OBC desarrollada en el marco de este trabajo.

2.3.4. Esquemas de planificación en FreeRTOS

Antes de abordar los algoritmos de planificación implementados por el sistema operativo, es fundamental comprender el concepto de **tarea** dentro del sistema operativo. En FreeRTOS, una tarea representa una unidad independiente de ejecución, análoga a un hilo en sistemas operativos de propósito general. Cada tarea cuenta con su propio contexto, pila, y conjunto de instrucciones, y puede interactuar con otras tareas o con el entorno mediante mecanismos de sincronización y comunicación.

El núcleo de FreeRTOS organiza y selecciona la ejecución de estas tareas en función de su estado actual y de las prioridades asignadas. Por lo tanto, conocer los posibles estados que una tarea puede adoptar resulta imprescindible para entender cómo opera el planificador y cómo se comporta el sistema en su conjunto.

[4] *Idle* se refiere al estado en que el procesador no ejecuta tareas activas, permaneciendo a la espera de nuevas operaciones.

2.3.5. Estados de una tarea en FreeRTOS

En FreeRTOS, cada tarea puede encontrarse en uno de los siguientes estados: *Running*, *Ready*, *Blocked* o *Suspended*. Estos estados definen si una tarea puede ejecutar inmediatamente o si debe esperar un evento o acción externa, como se muestra en la Figura 2.3.

- **Running:** Es el estado en el que se encuentra la tarea que está ejecutándose activamente en el procesador. En sistemas de un solo núcleo, sólo puede haber una tarea en estado *Running* a la vez.
- **Ready:** Tareas en estado *Ready* están listas para ejecutarse pero están esperando a que el planificador las seleccione. El planificador elige siempre la tarea de mayor prioridad en estado *Ready* para que pase a *Running*.
- **Blocked:** Una tarea pasa al estado *Blocked* cuando espera un evento para continuar su ejecución. Existen dos tipos principales de eventos que pueden provocar este bloqueo:

  1. Eventos temporales: como un retardo generado por funciones como `vTaskDelay()`, o una espera hasta alcanzar una marca de tiempo específica.
  2. Eventos de sincronización: como la espera por datos en una cola, un semáforo, o una notificación directa. Estos eventos son originados por otras tareas o interrupciones.

  FreeRTOS permite además establecer un *timeout*, de modo que una tarea pueda desbloquearse si no ocurre el evento esperado en un tiempo determinado.
- **Suspended:** Tareas en este estado están explícitamente suspendidas debido a la llamada a `vTaskSuspend()`. No pueden ser seleccionadas por el planificador bajo ninguna condición, a menos que se reanuden mediante `vTaskResume()` o `xTaskResumeFromISR()` en el caso de ejecutarse desde una interrupción. Este estado se utiliza en situaciones específicas, y la mayoría de las aplicaciones no lo emplean frecuentemente.

Tareas basadas en eventos vs. procesamiento continuo

En las primeras etapas de desarrollo es común implementar tareas que procesan de forma continua. Estas tareas nunca se bloquean, y, por lo tanto, permanecen siempre en estado *Ready*, impidiendo que tareas de menor prioridad puedan ejecutarse. Para evitar esta situación, FreeRTOS promueve el diseño de tareas orientadas a eventos, que solo pasan al estado *Ready* cuando un evento lo justifica, permaneciendo en *Blocked* el resto del tiempo. Esto no solo mejora la eficiencia del procesador, sino que también permite una asignación más justa y funcional de las prioridades entre tareas.

2.3.6. Sincronización y coordinación entre tareas

En un sistema multitarea como FreeRTOS, donde múltiples tareas pueden ejecutarse de forma concurrente y en distintos momentos del tiempo, es necesario contar con mecanismos que permitan coordinar su interacción. La sincronización entre tareas resulta esencial para asegurar la consistencia de los datos compartidos, evitar condiciones de carrera, y establecer relaciones causales entre eventos distribuidos en el tiempo.

FreeRTOS ofrece un conjunto de herramientas específicas para lograr dicha sincronización, entre las que se destacan las colas, los semáforos binarios, los *mutexes* con prioridad heredada, y las notificaciones directas entre tareas. Estos mecanismos permiten que una tarea espere por un evento generado por otra, que múltiples tareas accedan ordenadamente a

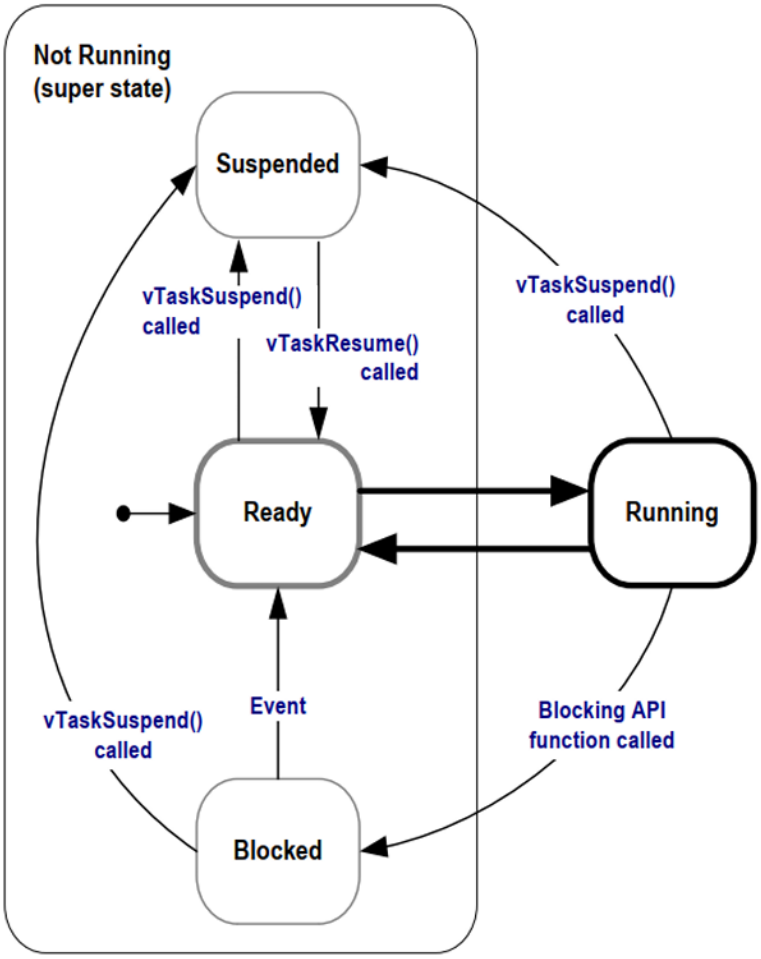

FIGURA 2.3. Diagrama de estados de una tarea en FreeRTOS [14]

un recurso compartido, o que una interrupción pueda desencadenar una acción determinada sin generar inconsistencias.

A continuación se describen en detalle los distintos mecanismos de sincronización y manejo de recursos compartidos utilizados a lo largo del desarrollo de la OBC:

- **Semáforo binario.** Puede conceptualizarse como un *token* que se utiliza para indicar que cierto evento ha ocurrido. Una tarea puede quedar a la espera de dicho *token*, y solo cuando otro componente del sistema lo libera, dicha tarea puede continuar su ejecución. Este mecanismo es especialmente útil para establecer una relación de dependencia temporal entre tareas, garantizando que una no avance hasta que la otra haya cumplido con una condición determinada. El *token*, en este caso, actúa únicamente como señalización, y no requiere ser devuelto una vez utilizado.

- ***Mutex.*** También puede entenderse como un *token*, pero en este caso asociado a un recurso compartido. Una tarea que desee acceder a dicho recurso debe primero obtener el *token* correspondiente. Mientras lo posea, se asume que el recurso está en uso exclusivo. Finalizada la operación, la tarea debe devolver el *token*, permitiendo que otras tareas puedan acceder al mismo recurso. Es importante destacar que este mecanismo no impone restricciones físicas sobre el acceso: técnicamente, una tarea podría utilizar el recurso en cualquier momento, incluso sin haber obtenido el *mutex*. Sin embargo, el diseño del sistema se basa en una convención estricta: cada tarea acuerda no acceder

al recurso a menos que haya obtenido el *token* de forma legítima, lo cual garantiza la correcta operación del sistema y evita condiciones de carrera o interferencias.

- **Colas.** Si bien su función principal es la transferencia segura de datos entre tareas o entre tareas e interrupciones, las colas también actúan como mecanismos de sincronización. Al permitir que una tarea se bloquee mientras espera la llegada de un dato, se establece una dependencia temporal entre el productor y el consumidor de dicho dato. Este comportamiento sincroniza el flujo de ejecución de las tareas involucradas, además de encapsular el dato transmitido y evitar condiciones de carrera sobre variables compartidas.

Herencia de prioridad

La **herencia de prioridad** es un mecanismo exclusivo de los *mutexes* que mitiga los efectos negativos de la inversión de prioridades, una condición donde una tarea de alta prioridad se ve bloqueada por una de baja prioridad que retiene un recurso necesario. En este caso, FreeRTOS eleva temporalmente la prioridad de la tarea que sostiene el *mutex* al nivel de la tarea bloqueada más prioritaria, garantizando que dicha tarea pueda liberar el recurso lo antes posible.

Este mecanismo impone varias consideraciones importantes:

- Una tarea puede heredar prioridades múltiples si toma varios *mutexes* sin liberarlos.
- La prioridad heredada se mantiene hasta que la tarea libera todos los *mutexes* retenidos.
- La prioridad heredada no se revierte aunque las tareas que esperan por el *mutex* agoten su tiempo de espera.

Dado que la herencia de prioridad afecta la planificación del sistema y complica el análisis temporal, no se recomienda depender de ella como mecanismo primario de diseño. Además, los *mutexes* no deben ser utilizados dentro de rutinas de interrupción, ya que su comportamiento depende del estado de las tareas y la planificación del sistema, no siendo segura en contextos asincrónicos.

Consideraciones sobre bloqueos y *mutexes* recursivos

Al utilizar *mutexes*, es fundamental tener en cuenta dos posibles escenarios problemáticos que pueden surgir si no se diseñan correctamente los accesos a los recursos compartidos.

- ***Deadlock* (bloqueo mutuo):** ocurre cuando dos o más tareas quedan bloqueadas permanentemente al esperar recursos que están siendo retenidos por las otras. Esta situación, conocida también como abrazo mortal, puede prevenirse asegurando un orden consistente en la adquisición de *mutexes* y, sobre todo, evitando esperas indefinidas. Por esta razón, se recomienda utilizar tiempos de espera que no sean infinitos en las funciones de toma de *mutexes*, permitiendo que la función retorne en caso de no poder obtener el recurso y facilitando así la detección y reporte de posibles errores de diseño antes de que afecten el sistema.
- **Problema de auto-bloqueo por llamadas anidadas:** en ciertos diseños, una tarea puede invocar funciones que, a su vez, intentan tomar el mismo *mutex* que la tarea ya posee. Esto puede suceder, por ejemplo, cuando una función externa con protección de exclusión accede a un recurso compartido y es llamada desde otra función que ya

había tomado dicho *mutex*. En estos casos, el uso de *mutexes* tradicionales puede llevar a un auto-bloqueo, ya que la tarea intentará tomar un recurso que ya está en su posesión, generando una espera indefinida.

Para resolver esta clase de situaciones, FreeRTOS proporciona *mutexes* recursivos, los cuales permiten que una misma tarea tome múltiples veces un mismo *mutex*, siempre y cuando lo libere exactamente la misma cantidad de veces.

Colas como mecanismo de sincronización

Además de su rol fundamental en la transferencia de datos, las colas en FreeRTOS cumplen una función clave como mecanismo de sincronización entre tareas, tal como se explica en la sección 2.3.9. El simple acto de enviar o recibir un dato mediante una cola puede establecer una relación de dependencia temporal entre tareas, donde una espera que la otra produzca cierta información antes de continuar con su ejecución.

Desde una perspectiva conceptual, una cola puede entenderse como un canal a través del cual las tareas se comunican y se sincronizan. Cuando una tarea lectora queda bloqueada esperando un dato y solo se reactiva cuando dicho dato es colocado por otra tarea, se produce un efecto de sincronización implícita: no solo se transfiere información, sino también un control sobre el flujo de ejecución.

En el código 2.1, se presenta una tarea utilizada en el sistema de validación automática del CubeSat, cuya responsabilidad es ejecutar pruebas en función de la cantidad de pulsos recibidos desde una entrada externa. La sincronización entre la fuente de pulsos (que puede ser una interrupción o una tarea auxiliar) y esta tarea se realiza mediante una cola denominada `xPulseQueue`.

```
void taskProcessPulses(void *params) {
    TickType_t delay = pdMS_TO_TICKS((int)params);
    int local_pulse_count;

    for (;;) {
        if (xQueueReceive(xPulseQueue, &local_pulse_count, portMAX_DELAY)
   ) {
            ejecutar_test(local_pulse_count);

            while(UART_available_txdata(USART1) > 0) {
               taskYIELD();
            }
            vTaskDelay(delay);
            gpio_set(TEST_INDICATOR_PORT, TEST_INDICATOR_PIN);
            vTaskDelay(delay);
            gpio_clear(TEST_INDICATOR_PORT, TEST_INDICATOR_PIN);
            vTaskDelay(delay);
        }
        taskYIELD();
    }
}
```

CÓDIGO 2.1. Tarea que procesa los pulsos y ejecuta el test correspondiente

Esta implementación permite que la tarea permanezca en estado *Blocked* hasta que haya datos disponibles en la cola, lo que ocurre cuando la entidad encargada del conteo de pulsos envía el valor correspondiente mediante `xQueueSend()`. Cabe destacar que este mecanismo podría haber sido implementado utilizando un semáforo binario para activar la tarea, junto con una variable global compartida para almacenar la cantidad de pulsos. Sin embargo, el uso de una cola simplifica la lógica de diseño, encapsulando simultáneamente la señal

de activación y el dato asociado. Esta solución resulta más robusta y escalable, al eliminar el riesgo de condiciones de carrera sobre variables compartidas.

2.3.7. Tipos de planificación disponibles

FreeRTOS permite elegir entre tres esquemas de planificación principales para microcontroladores de un solo núcleo:

1. **Planificación cooperativa** (en inglés, *Cooperative scheduling*): En este esquema, el sistema operativo no interrumpe automáticamente la tarea en ejecución. La tarea debe ceder voluntariamente el control del procesador, ya sea llamando a `taskYIELD()`[5] o entrando en estado *Blocked* o *Suspended*.

   Ventajas: mayor simplicidad en el manejo de recursos compartidos.
   Desventajas: menor capacidad de respuesta a eventos críticos.

2. **Planificación con prioridad fija y preempción** (en inglés, *preemptive scheduling*): Cuando una tarea de mayor prioridad pasa al estado *Ready*, interrumpe de inmediato a la tarea en ejecución. Sin embargo, si hay varias tareas con la misma prioridad en estado *Ready*, no se alternan automáticamente: la tarea que obtiene el procesador continúa ejecutándose hasta que se bloquee o se suspenda.

   Ventajas: mayor determinismo y menor sobrecarga del planificador.
   Desventajas: posibilidad de *starvation* (una tarea puede monopolizar la CPU).

3. **Planificación con prioridad fija y segmentación temporal** (en inglés, *time slicing*): Además de permitir la preempción, este esquema reparte equitativamente el tiempo del procesador entre tareas con la misma prioridad que se encuentren en estado *Ready*. Cada tarea recibe una porción de tiempo para ejecutarse, determinada por el intervalo del *tick* del sistema, alternándose en rondas sucesivas (*time slicing*).

   Ventajas: ejecución más equitativa entre tareas del mismo nivel.
   Desventajas: mayor cantidad de cambios de contexto y ligera sobrecarga del planificador.

Cuando se desactiva el *time slicing*, el planificador sólo selecciona una nueva tarea para ejecutar si:

- Una tarea de mayor prioridad pasa al estado *Ready*, o
- La tarea en ejecución entra en estado *Blocked* o *Suspended*.

Este comportamiento reduce la frecuencia de cambios de contexto, disminuyendo la carga del planificador sobre el sistema. Sin embargo, también implica que si existen varias tareas en estado *Ready* con la misma prioridad, una sola de ellas puede monopolizar el uso del procesador, mientras que las demás permanecen sin ejecutar. Esta situación puede provocar un reparto desigual del tiempo de CPU entre tareas equivalentes y, en el peor de los casos, generar un bloqueo indefinido.

Por esta razón, ejecutar el planificador sin *time slicing* se considera una técnica avanzada, recomendada únicamente para usuarios experimentados que puedan garantizar una correcta distribución de CPU por otros medios (como ceder el control voluntariamente o dividir la ejecución en fragmentos breves) [14].

[5] `taskYIELD()` es una función del API de FreeRTOS que permite a una tarea ceder voluntariamente el control del procesador, forzando al planificador a seleccionar otra tarea en estado *Ready*.

2.3.8. Elección del esquema en este proyecto

En esta implementación se optó por el esquema de planificación **preemptivo con *time slicing***, configurando las siguientes macros en el archivo `FreeRTOSConfig.h`:

```
#define configUSE_PREEMPTION        1
#define configUSE_TIME_SLICING      1
```

Este esquema permite una respuesta inmediata ante eventos críticos mediante la **preempción**, y garantiza una distribución equitativa del tiempo de CPU entre tareas en estado *Ready* con igual prioridad. Esta característica es fundamental en sistemas CubeSat, donde múltiples tareas (como la adquisición de datos de sensores, escritura en memoria, control de actitud y comunicación) deben ejecutarse concurrentemente y mantener niveles de prioridad diferenciados.

Como contrapartida, y tal como se detallará en los capítulos siguientes, se implementaron mecanismos de exclusión mutua provistos por el sistema operativo para evitar el acceso concurrente a recursos compartidos —como variables globales, *buses* I^2C o la memoria SD— ante un posible cambio de contexto. Estos mecanismos permiten asegurar la coherencia de los datos incluso cuando una tarea es interrumpida en medio de una operación crítica. Para ello se utilizaron semáforos, *mutexes* y técnicas de sincronización segura desde interrupciones.

2.3.9. Concepto y funcionamiento de las colas

Como se mencionó antes, FreeRTOS proporciona un mecanismo robusto de comunicación entre tareas, interrupciones y otros componentes del sistema mediante estructuras denominadas *colas* (*queues*). Las colas permiten transferir datos de manera segura y asincrónica, implementando un búfer del tipo *First In, First Out* (FIFO). Cada cola puede almacenar un número finito de ítems de tamaño fijo, definido en el momento de su creación.

Las colas pueden ser accedidas por múltiples tareas o incluso desde interrupciones, lo que las convierte en una herramienta fundamental para la sincronización y comunicación entre tareas concurrentes. El sistema operativo proporciona mecanismos de bloqueo tanto al escribir como al leer, permitiendo que una tarea espere durante un tiempo determinado si la cola está llena o vacía, respectivamente.

Un aspecto fundamental a destacar en la arquitectura de colas es que el sistema utiliza por defecto un modelo de transmisión denominado *queue by copy*. En este modelo, el contenido que se envía a una cola se copia directamente dentro de la memoria interna de la cola, a diferencia del modelo alternativo *queue by reference*, en el cual sólo se envía un puntero al dato original. Este enfoque de copia aporta ventajas significativas en términos de robustez, simplicidad y portabilidad.

A continuación se detallan los motivos por los cuales esta decisión de diseño resulta favorable:

- El envío por copia no impide utilizar colas también para enviar punteros, lo cual resulta útil cuando el tamaño del dato hace que copiarlo sea ineficiente.
- Es posible enviar una variable local (en *stack*) a una cola, incluso si la función que la declaró ha finalizado, ya que el contenido es copiado en la cola antes de que la variable deje de existir.
- No es necesario reservar previamente un *buffer* para el dato que se enviará; la cola se encarga internamente de almacenar el contenido.

- La tarea emisora puede reutilizar de inmediato el *buffer* o la variable utilizada, ya que su contenido ha sido copiado.
- El diseño desacopla completamente la tarea emisora de la receptora: no hay dependencia sobre la propiedad ni la gestión del ciclo de vida de los datos transmitidos.
- El sistema operativo es responsable de la memoria interna de la cola, lo que simplifica la gestión general.
- En sistemas con protección de memoria, enviar punteros puede ser problemático si la memoria apuntada no es accesible por ambas tareas. El modelo por copia permite cruzar estos límites sin comprometer la integridad.

Comunicación entre tareas

En lugar de compartir variables globales, FreeRTOS permite que las tareas intercambien información a través de colas. Cada tarea puede insertar o extraer datos de una cola, delegando al *kernel* la responsabilidad de sincronizar y gestionar el acceso concurrente de forma segura.

Una característica destacada de este mecanismo es que las tareas pueden permanecer en estado *Blocked* mientras esperan la llegada de datos a la cola, reanudando su ejecución únicamente cuando los datos estén disponibles. Esto permite una planificación eficiente, evitando el uso innecesario del procesador.

Asimismo, como se mencionó anteriormente, el modelo de operación por copia permite que los datos enviados a la cola provengan del *stack* de la tarea emisora, eliminando riesgos de accesos inválidos o corrupción de memoria posterior.

Caso de uso: transmisión UART

Un caso representativo de uso de colas en este proyecto es la interfaz UART. En dichos controladores se diseñó un mecanismo de transmisión basado en una tarea dedicada que se inicia una vez que se configura el periférico USART correspondiente. Esta tarea, `taskUART_transmit()`, es responsable de extraer datos desde una cola asociada al canal de transmisión (`txq`) y enviarlos a través del periférico serie correspondiente.

Una característica relevante de esta implementación es que la tarea no realiza un sondeo continuo (*polling*) para verificar si hay datos disponibles en la cola. En cambio, se aprovechan las capacidades del *kernel* de FreeRTOS para mantener a la tarea en estado *Blocked* mientras no haya elementos en la cola. Esta condición permite liberar el procesador para otras tareas y mantener un sistema más eficiente desde el punto de vista del uso de CPU.

El evento que despierta a la tarea de su estado bloqueado es la llegada de un nuevo dato a la cola, lo cual ocurre cuando otra parte del sistema invoca la función `UART_puts()` o `UART_write()`. Esta función encapsula el envío de cadenas de caracteres y coloca los datos en la cola de transmisión utilizando `xQueueSend()`. Una vez que se encola un nuevo elemento, la tarea `taskUART_transmit()` pasa automáticamente al estado *Ready*, quedando disponible para ser seleccionada por el planificador en cuanto tenga asignado un intervalo de ejecución.

Dentro del ciclo principal de la tarea, se emplea una llamada a `xQueueReceive()` con un tiempo de espera definido (`TIMEOUT_TRANSMIT`), que permite procesar todos los datos disponibles en la cola de forma secuencial. La transmisión real del carácter por USART se realiza sólo cuando el registro de transmisión se encuentra vacío. En caso contrario, la tarea cede el control del procesador mediante `taskYIELD()`, evitando un ciclo de espera activa (*busy-waiting*) e incrementando la eficiencia general del sistema.

A continuación se presenta el código de dicha tarea:

```
void taskUART_transmit(void *param) {
    uint32_t usart_id = (uint32_t)param;

    uart_t *uart = get_uart(usart_id);
    if (uart == NULL) return;

    uint16_t ch;
    for (;;) {
        // Esperar y recibir datos desde la cola de transmision
        while (xQueueReceive(uart->txq, &ch, TIMEOUT_TRANSMIT) == pdPASS)
    {
            // Esperar a que el registro de transmision este vacio
            while (!usart_get_flag(uart->usart, USART_SR_TXE))
                taskYIELD(); // Ceder la CPU hasta que este listo

            // Enviar el byte por USART
            usart_send(uart->usart, (uint16_t)ch);
        }

    }
}
```

CÓDIGO 2.2. Tarea encargada de la transmisión UART mediante cola

Comunicación desde múltiples tareas hacia una única receptora

En muchas aplicaciones embebidas es común que una misma tarea necesite recibir datos provenientes de múltiples fuentes. Una solución eficiente recomendada por FreeRTOS consiste en definir una estructura que contenga tanto el dato como su identificador de origen, y utilizar una única cola compartida por todas las tareas emisoras. La tarea receptora interpreta el contenido en base al identificador recibido, permitiendo una arquitectura escalable y desacoplada.

Este enfoque evita la necesidad de múltiples colas o del uso de conjuntos de colas (*queue sets*), que presentan mayor complejidad y sobrecarga. La Figura 2.4 ilustra este patrón, donde tres tareas independientes escriben en una misma cola una estructura `Data_t` que incluye un campo de identificación.

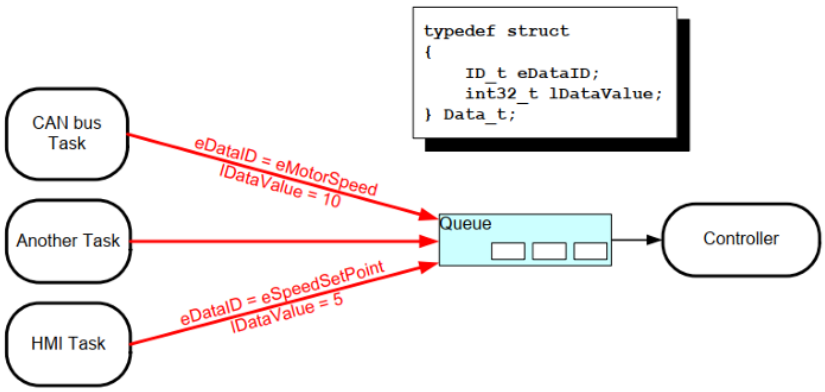

FIGURA 2.4. Comunicación desde múltiples tareas utilizando una única cola [14]

Dicho patrón fue utilizado en el monitor de recursos, el cual se encarga de la adquisición de datos de los distintos sensores, el procesamiento de dichos datos y su posterior almacenamiento. Se recomienda su uso en futuras implementaciones del sistema en el CubeSat, ya que ofrece una solución simple y eficiente para la recepción centralizada de datos provenientes de múltiples tareas y sensores.

2.3.10. Gestión de memoria dinámica en FreeRTOS

FreeRTOS permite la creación de objetos del *kernel* mediante asignación de memoria estática o dinámica. Cada estrategia presenta ventajas y limitaciones específicas, y su elección impacta directamente en el diseño, el aprovechamiento de recursos y la robustez del sistema embebido.

Un aspecto importante a destacar es que el término **asignación dinámica**, tal como lo utiliza FreeRTOS, hace referencia a la creación de objetos del sistema operativo durante la ejecución del programa. Esto contrasta con la **asignación estática**, en la que todos los recursos se reservan en tiempo de compilación. Sin embargo, es fundamental aclarar que esta asignación "dinámica" se realiza sobre un bloque de memoria que ha sido previamente reservado de forma estática.

Este bloque, conocido como *heap*, consiste en un arreglo estático de tipo `uint8_t`, cuyo tamaño está definido en tiempo de compilación mediante la macro `configTOTAL_HEAP_SIZE`. Tanto su ubicación como su extensión son conocidas desde el enlazado del programa, por lo que forma parte del segmento de datos del sistema. Durante la ejecución, ese bloque se administra como un *heap* dinámico: la función `pvPortMalloc()` permite subdividir el bloque en porciones más pequeñas cada vez que se solicita memoria, y la función `vPortFree()`, disponible en algunos esquemas, permite liberar dicha memoria. La forma en que se realiza esta gestión depende del esquema de *heap* seleccionado (por ejemplo, `heap_1`, `heap_4`, etc.). Este enfoque híbrido proporciona flexibilidad en tiempo de ejecución sin renunciar a un control preciso sobre el uso de memoria, lo cual resulta esencial en sistemas embebidos con recursos restringidos y requisitos de comportamiento determinista.

FreeRTOS utiliza memoria dinámica o estática para administrar todos sus objetos internos, incluyendo colas, semáforos, *mutexes*, temporizadores y tareas. Cada uno de estos elementos mantiene estructuras internas que permiten gestionar su estado y funcionamiento. Por ejemplo, las colas requieren espacio para almacenar los datos transmitidos entre tareas, mientras que los semáforos y *mutexes* conservan información sobre las tareas que esperan acceder a recursos compartidos. Esta infraestructura, aunque eficiente, consume RAM, lo que obliga a realizar un dimensionamiento cuidadoso en sistemas con recursos limitados.

En el caso de las tareas, FreeRTOS asigna memoria para dos componentes principales: la pila de ejecución (*stack*) y el bloque de control de tarea, conocido como *Task Control Block* (TCB). Esta última es una estructura que encapsula toda la información necesaria para que el planificador del sistema pueda administrar la tarea. Incluye punteros al *stack*, la prioridad asignada, el nombre de la tarea y referencias que permiten insertarla en listas de planificación o espera de eventos [15]. Además, si está habilitado el mecanismo de herencia de prioridad, también guarda la prioridad base para poder restaurarla una vez que se libera el recurso compartido.

La pila de cada tarea, por su parte, almacena tanto variables locales como el contexto del procesador necesario para realizar cambios de tarea. Su tamaño debe elegirse con cuidado: un *stack* demasiado pequeño puede causar errores difíciles de detectar, mientras que uno excesivamente grande desperdicia memoria valiosa. Para facilitar esta tarea, FreeRTOS ofrece una función, `uxTaskGetStackHighWaterMark()`, que permite inspeccionar el uso máximo alcanzado por el *stack* durante la ejecución, lo cual es útil para ajustar márgenes de seguridad y optimizar la asignación de recursos, como se detalla en el Anexo J.1.

2.3.11. Criterios de selección del esquema de asignación dinámica en FreeRTOS

La asignación dinámica de memoria es un concepto fundamental en la programación en C, que permite reservar memoria en tiempo de ejecución según las necesidades del sistema. Este enfoque es especialmente útil cuando la cantidad de memoria requerida no puede determinarse de antemano o cuando se desea gestionar los recursos de manera más flexible y eficiente. Sin embargo, en sistemas embebidos con recursos limitados, su uso conlleva desafíos significativos: las funciones estándar `malloc()` y `free()` no siempre están disponibles, pueden consumir espacio de código considerable, carecen de seguridad en entornos multitarea, no son deterministas y pueden sufrir fragmentación de memoria, lo cual compromete la estabilidad del sistema.

En el contexto de sistemas RTOS, estos desafíos adquieren mayor relevancia. La predictibilidad, eficiencia y confiabilidad son aspectos críticos, por lo que FreeRTOS no depende de las funciones estándar de asignación dinámica. En su lugar, emplea un conjunto de funciones propias: `pvPortMalloc()` y `vPortFree()`, que presentan la misma interfaz que las funciones estándar pero están implementadas de forma específica para sistemas embebidos.

FreeRTOS delega la responsabilidad de la gestión de memoria dinámica a la capa portable, permitiendo adaptar la implementación a los requerimientos del *hardware* y del sistema. Para ello, proporciona cinco esquemas de asignación de memoria ubicados en los archivos `heap_1.c` a `heap_5.c`, cada uno con características particulares en cuanto a complejidad, eficiencia, soporte para liberación de memoria, y manejo de fragmentación. Esta flexibilidad permite al desarrollador seleccionar el esquema más apropiado según las restricciones y objetivos del proyecto, garantizando un equilibrio entre simplicidad y robustez.

A continuación, se describen en detalle las características, ventajas y limitaciones de cada uno de los esquemas de gestión de memoria dinámica que ofrece FreeRTOS.

2.3.12. Esquemas de gestión de *heap* disponibles

heap_1: asignación sin liberación

El esquema de memoria `heap_1` implementa la versión más simple de `pvPortMalloc()`, y no incluye una implementación de `vPortFree()`. Esto significa que la memoria solicitada no puede liberarse una vez asignada, lo cual lo convierte en una estrategia determinista y libre de fragmentación. En estos casos, la memoria se asigna al inicio de la ejecución y se mantiene reservada durante toda la vida útil del sistema.

La función `pvPortMalloc()` en `heap_1.c` trabaja dividiendo un arreglo de tipo `uint8_t` predefinido (el *FreeRTOS heap*) en bloques más pequeños cada vez que se solicita memoria.

Esta estrategia garantiza una ejecución determinista y consistente, eliminando los riesgos asociados al uso de *heap* en tiempo real. No obstante, en aplicaciones donde la necesidad de memoria es transitoria —es decir, donde ciertos bloques de memoria solo son requeridos durante intervalos limitados de tiempo—, la imposibilidad de liberar memoria asignada puede conducir a una ocupación permanente e innecesaria de recursos. Esta situación también puede derivar en complicaciones en la gestión de recursos conforme el sistema experimenta variaciones en la carga de trabajo o incrementa su complejidad a medida que el proyecto escala.

heap_2: esquema con liberación simple

El archivo `heap_2.c` implementa `pvPortMalloc()` y `vPortFree()`, permitiendo liberar memoria previamente asignada. Utiliza una estrategia básica de lista libre enlazada,

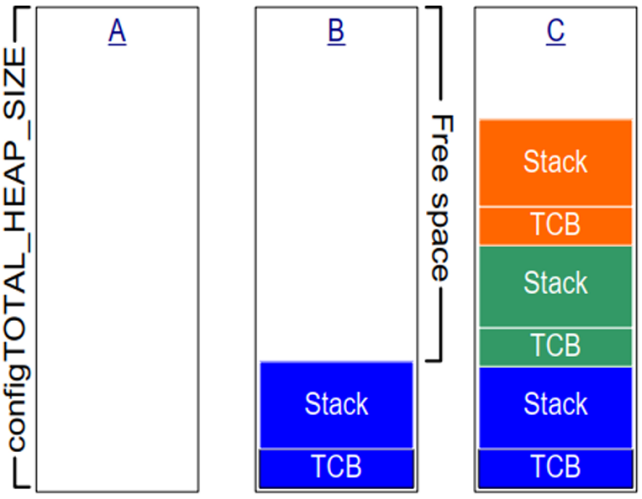

FIGURA 2.5. Esquema de asignación de memoria con `heap_1.c` [14].

lo cual lo hace más flexible que `heap_1`, pero a costa de introducir cierta complejidad y posibilidad de fragmentación.

Sin embargo, este esquema ha sido superado por `heap_4`, el cual ofrece mayor robustez, menor fragmentación y mejor rendimiento en la gestión de memoria dinámica. Por esta razón, `heap_2` se mantiene en la distribución oficial de FreeRTOS solo por compatibilidad hacia atrás, y no se recomienda su uso en nuevos diseños [14].

heap_3: asignación delegada al sistema

El esquema `heap_3` implementa `pvPortMalloc()` y `vPortFree()` redirigiendo ambas llamadas a las funciones estándar `malloc()` y `free()` de la biblioteca del lenguaje C. En consecuencia, el tamaño y comportamiento del *heap* dependen del entorno de ejecución del compilador.

Para evitar condiciones de carrera durante el acceso concurrente a la memoria dinámica, dado que las funciones estándar no son intrínsecamente *thread-safe*, FreeRTOS suspende temporalmente el planificador cada vez que se invoca una de ellas. Si bien esta estrategia permite una integración sencilla con aplicaciones existentes y bibliotecas externas que ya utilizan el *heap* del sistema, no ofrece control sobre la fragmentación de memoria ni sobre los tiempos de ejecución.

heap_4: gestión con coalescencia de bloques libres

El esquema `heap_4` es uno de los más recomendados por FreeRTOS cuando se requiere asignación y liberación dinámica de memoria en tiempo de ejecución. Al igual que `heap_1` y `heap_2`, utiliza un arreglo estático de tipo `uint8_t`, cuyo tamaño es definido por la macro `configTOTAL_HEAP_SIZE`, y que actúa como el área de *heap* del sistema. Este arreglo es parte del segmento de datos del sistema y, por lo tanto, está reservado en tiempo de compilación.

La principal característica de `heap_4` es que implementa un algoritmo de asignación *first fit* (primer ajuste), y que además realiza coalescencia[6] de bloques libres. Esto significa que:

[6] Propiedad de las cosas de unirse

- Al solicitar memoria con `pvPortMalloc()`, el sistema recorre los bloques de *heap* disponibles en orden y selecciona el primero que sea suficientemente grande para satisfacer la solicitud.
- Si existen bloques libres contiguos, `heap_4` los combina (*coalescing*) en un único bloque mayor. Esta estrategia minimiza la fragmentación de memoria interna y externa.

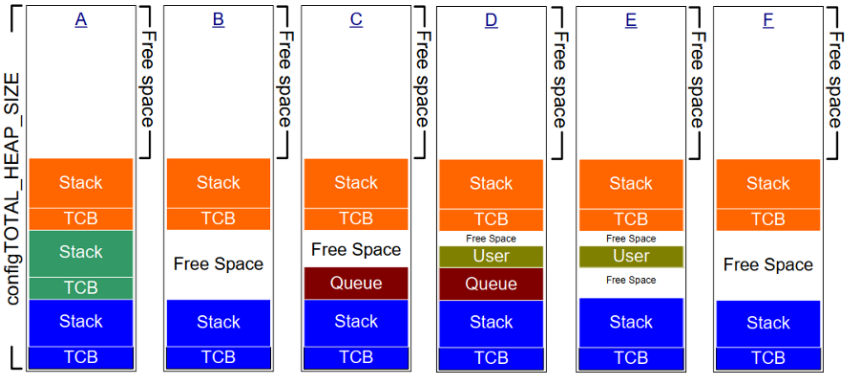

FIGURA 2.6. Ejemplo de funcionamiento de `heap_4.c` [14].

heap_5: gestión de múltiples regiones de memoria no contiguas

El esquema de gestión de memoria `heap_5` extiende las capacidades de `heap_4` al permitir la utilización de múltiples regiones de memoria físicamente no contiguas como un único *heap* lógico. Al igual que `heap_4`, emplea un algoritmo de asignación del tipo *first-fit* con coalescencia de bloques libres, lo que reduce la fragmentación interna y mejora la eficiencia en escenarios donde se liberan y reasignan bloques de distintos tamaños.

La principal diferencia radica en que `heap_4` se limita a gestionar memoria dentro de un único bloque contiguo definido por `configTOTAL_HEAP_SIZE`, mientras que `heap_5` permite distribuir el *heap* en varias secciones de memoria separadas. Esta funcionalidad resulta especialmente útil en plataformas embebidas donde la RAM no se presenta como un espacio continuo en el mapa de memoria, como puede ocurrir en dispositivos con RAM interna y externa, o en arquitecturas segmentadas.

Para habilitar esta funcionalidad, el programador debe definir explícitamente las regiones de memoria a utilizar mediante un arreglo de estructuras `HeapRegion_t`, especificando la dirección base y el tamaño de cada región. Posteriormente, se debe invocar la función `vPortDefineHeapRegions()` para registrar estas regiones antes de realizar cualquier asignación dinámica.

Si bien el uso de `heap_5` implica una mayor complejidad de configuración, proporciona la máxima flexibilidad en sistemas que requieren un aprovechamiento avanzado y controlado del espacio de memoria disponible.

2.3.13. Elección del gestor de memoria dinámica

La selección del esquema de gestión de memoria dinámica es un componente fundamental en el diseño de sistemas embebidos basados en FreeRTOS, especialmente en entornos críticos como una OBC de un CubeSat. En este proyecto se ha optado por utilizar el esquema `heap_4`, considerando su adecuado equilibrio entre flexibilidad, eficiencia en el uso del *heap* y soporte para funciones avanzadas de diagnóstico en tiempo de ejecución.

El esquema `heap_4` permite realizar asignaciones y liberaciones dinámicas de memoria, además implementa coalescencia de bloques libres, lo que permite minimizar así el riesgo de fragmentación de memoria. Estas características lo convierten en una solución apta para sistemas que requieren gestión dinámica de recursos durante su operación, pero sin los niveles de complejidad adicionales presentes en esquemas más avanzados como `heap_5`.

Aunque `heap_4` no garantiza un comportamiento completamente determinista —ya que el tiempo necesario para recorrer el *heap* puede variar en función del estado actual de la memoria—, se destaca por ofrecer un desempeño eficiente y adecuado para la mayoría de los escenarios. Tal como lo expresa el manual oficial de FreeRTOS, "`heap_4` no es determinista, pero es más rápido que la mayoría de las implementaciones estándar de `malloc()` y `free()`" [14], lo que respalda su uso en sistemas donde se requiere la utilización de memoria dinámica, manteniendo al mismo tiempo criterios de eficiencia temporal y control sobre la fragmentación del *heap*.

Es importante aclarar que la elección de `heap_4` como gestor de memoria dinámica no excluye el uso de memoria estática. FreeRTOS permite habilitar simultáneamente la asignación estática y dinámica mediante las macros `configSUPPORT_STATIC_ALLOCATION`[7] y `configSUPPORT_DYNAMIC_ALLOCATION`, permitiendo así que objetos críticos o de uso conocido sean reservados de forma estática para asegurar su disponibilidad, mientras que aquellos de naturaleza más transitoria o condicional pueden gestionarse dinámicamente mediante el *heap*.

Otro aspecto clave que motivó la elección de `heap_4` fue su compatibilidad con funciones avanzadas de monitoreo de uso de memoria, que resultan esenciales en las etapas de diseño, validación y operación de sistemas embebidos. Estas funciones permiten obtener métricas de uso del *heap* en tiempo real, facilitando la detección de comportamientos anómalos, el ajuste fino de los parámetros del sistema y la verificación de que la memoria disponible es suficiente bajo distintas condiciones de carga.

Entre las funciones más relevantes se incluyen:

- `xPortGetFreeHeapSize()`: permite consultar en tiempo real la cantidad total de memoria libre disponible en el heap. Aunque no proporciona información sobre fragmentación, es útil para monitorear el estado general del sistema.
- `xPortGetMinimumEverFreeHeapSize()`: devuelve la menor cantidad de memoria libre que ha existido desde el arranque del sistema. Esta métrica es clave para validar que el *heap* fue correctamente dimensionado y que no se han alcanzado condiciones críticas.
- `vPortGetHeapStats()`: completa una estructura de tipo `HeapStats_t` con estadísticas detalladas sobre el estado del heap, incluyendo el tamaño del bloque libre más grande y más pequeño, la cantidad total de bloques libres, y contadores de asignaciones y liberaciones exitosas.

Estas herramientas resultan especialmente útiles durante el diseño y validación del sistema, ya que proporcionan estadísticas en tiempo de ejecución que permiten ajustar de forma precisa los tamaños de memoria asignados en la etapa de compilación, optimizando así el uso global de recursos en la OBC.

[7] Si se habilita `configSUPPORT_STATIC_ALLOCATION`, el usuario debe proporcionar manualmente la memoria para cada objeto del *kernel*. Ver sección 3.4.2 del manual *Mastering the FreeRTOS Real-Time Kernel* [14].

Capítulo 3

Framework

Este capítulo describe el desarrollo de un entorno de pruebas automatizado que permite verificar de manera sistemática el funcionamiento del *firmware* implementado para la OBC del CubeSat. Dado que la validación de módulos embebidos involucra tanto aspectos de *software* como de *hardware*, fue necesario diseñar una infraestructura capaz de integrar ambos mundos de forma flexible, reproducible y escalable.

El objetivo principal de este entorno es permitir la ejecución de pruebas funcionales sobre la placa STM32F103C8T6, facilitando la detección de errores, la calibración de sensores, y el análisis del comportamiento del sistema ante distintos escenarios. Para lograr esto, se implementó una arquitectura distribuida y automatizada que permite compilar, cargar, ejecutar y evaluar pruebas en la STM32, a partir de un conjunto de herramientas integradas entre sí.

Todo el trabajo, incluyendo el código del *firmware*, *scripts* y configuración del entorno, se encuentra disponible públicamente en:

https://gitlab.com/fspaltro/FiubaSAT

Entre los componentes principales del entorno de pruebas se destacan los siguientes:

- **Herramientas de *software*:**
  - **Docker:** garantiza un entorno de compilación homogéneo, portátil y sin dependencias externas.
  - **Git:** sistema de control de versiones utilizado para gestionar el código fuente y colaborar en el desarrollo.
  - **GitLab Runner:** ejecuta los *pipelines* de integración continua. Cada vez que se realiza un *push* al repositorio, se activa un flujo de pruebas en la placa real.
  - **Tailscale:** proporciona una red privada virtual (VPN) para acceder de manera remota y segura a la Raspberry Pi desde cualquier ubicación, facilitando el desarrollo y monitoreo sin necesidad de acceso físico.
- **Herramientas de *hardware*:**
  - **Raspberry Pi 5:** actúa como nodo de control principal. Se encarga de compilar el *firmware*, programar la STM32, ejecutar las pruebas y validar los resultados. También hospeda el *runner* de GitLab.
  - **Analizador lógico:** permite capturar señales digitales clave para verificar el comportamiento del sistema a nivel de *hardware*, especialmente útil en pruebas de protocolos como I^2C o SPI.

- **Interruptor de corriente inteligente:** permite controlar remotamente el encendido de la Raspberry Pi.

Este entorno fue fundamental para poder validar el correcto funcionamiento de los controladores desarrollados, así como también para asegurar que las rutinas de inicialización, configuración de sensores y adquisición de datos funcionen de forma predecible en una arquitectura real.

Cabe destacar que, a lo largo del proyecto, se optó por evitar herramientas propietarias como `STM32CubeIDE` u otros entornos cerrados, y en su lugar se priorizó el uso de herramientas de código abierto y ampliamente soportadas por la comunidad. Esta decisión no solo permitió una integración más fluida con el sistema automatizado de pruebas, sino que también garantiza la sostenibilidad del proyecto a largo plazo. Al basarse en entornos de compilación y grabación estándar, el sistema puede ser reutilizado, extendido o adaptado sin quedar atado a versiones específicas de *software* propietario ni a licencias restrictivas. Esto asegura que el entorno de pruebas pueda seguir evolucionando aún si cambia el equipo de desarrollo, las plataformas utilizadas o el ecosistema de herramientas disponibles.

3.1. Docker

Docker es una plataforma que permite crear, desplegar y gestionar aplicaciones dentro de contenedores (entornos ligeros, portátiles y autocontenidos que agrupan todo lo necesario para ejecutar una aplicación, incluyendo el código fuente, bibliotecas, herramientas y configuraciones del sistema). Su principal ventaja radica en ofrecer un entorno de ejecución consistente, independiente del sistema operativo anfitrión y fácilmente replicable en distintos dispositivos.

En el contexto de este proyecto, el uso de Docker se justifica desde tres enfoques principales:

1. El desarrollo sobre sistemas embebidos requiere la instalación de compiladores, toolchains y bibliotecas específicas, cuya gestión puede afectar al sistema operativo principal o interferir con otros proyectos. Docker permite aislar completamente estas dependencias, evitando conflictos y asegurando que el entorno se mantenga estable y reproducible a lo largo del tiempo, independientemente de las modificaciones que puedan hacerse en el equipo del desarrollador.

2. Al contener todo el entorno de desarrollo en una imagen Docker, todos los miembros del equipo pueden trabajar sobre una base idéntica. Esto reduce drásticamente los errores derivados de configuraciones divergentes y garantiza que cualquier modificación o avance pueda ejecutarse de forma uniforme en todos los entornos. Además, facilita la incorporación de nuevos integrantes al equipo sin necesidad de replicar manualmente cada ajuste del entorno.

3. Docker permite capturar un *snapshot* completo del entorno de desarrollo, incluyendo versiones exactas de todas las dependencias, configuraciones, herramientas y variables de entorno. Esta capacidad es fundamental para proyectos a largo plazo como este, ya que garantiza que las pruebas y funcionalidades desarrolladas puedan seguir funcionando de forma idéntica en el futuro, independientemente de actualizaciones o cambios de entorno. En este sentido, Docker actúa como un complemento práctico a la documentación.

Dentro del ecosistema de Docker, dos conceptos resultan esenciales: las imágenes y los contenedores [16]. Una imagen de Docker es un paquete inmutable que contiene todo lo

necesario para ejecutar una aplicación: el código fuente, las bibliotecas y dependencias requeridas, herramientas auxiliares, variables de entorno, scripts de inicialización y configuraciones específicas del entorno de ejecución. Esta imagen, que puede entenderse como una fotografía estática del sistema de archivos y el entorno de ejecución en un estado dado, no se ejecuta por sí sola, sino que sirve como base para generar instancias funcionales.

Un contenedor es precisamente esa instancia en ejecución de una imagen. Cuando se lanza un contenedor, Docker toma la imagen como base y crea un entorno aislado que hereda el sistema de archivos definido en la imagen, pero con su propio espacio de ejecución. Un contenedor puede modificarse en tiempo de ejecución, generar archivos temporales o interactuar con el exterior, pero estos cambios no afectan a la imagen original.

A diferencia de las máquinas virtuales tradicionales, los contenedores no requieren un sistema operativo completo dentro de cada instancia, ya que comparten el núcleo (*kernel*) del sistema operativo del *host*. El *kernel* es el componente central encargado de gestionar el *hardware* y coordinar la ejecución de procesos.

Esta arquitectura permite que los contenedores sean considerablemente más livianos, se inicien en menos tiempo y utilicen los recursos del sistema de forma más eficiente. Además, al encapsular únicamente el espacio de usuario necesario para la aplicación, los contenedores simplifican el despliegue, reducen la complejidad operativa y facilitan el escalado de instancias en entornos distribuidos.

3.1.1. Dockerfile

Una vez comprendido el rol de las imágenes y contenedores dentro de Docker, el siguiente paso es entender cómo se construyen dichas imágenes. Para facilitar este proceso, Docker permite generar imágenes de forma automática a partir de un archivo de instrucciones denominado **Dockerfile**.

Un Dockerfile es un archivo de texto que contiene una serie de comandos que Docker ejecuta de manera secuencial para ensamblar una imagen personalizada. Estos comandos definen, por ejemplo, qué sistema base utilizar, qué paquetes instalar, qué archivos copiar, qué variables de entorno establecer y qué instrucciones ejecutar al iniciar el contenedor [17]. Cada instrucción dentro del Dockerfile describe una acción específica, y Docker las ejecuta secuencialmente durante el proceso de construcción. Si bien las instrucciones no son sensibles a mayúsculas o minúsculas, por convención se escriben en mayúsculas para diferenciarlas claramente de sus argumentos.

Requerimientos

Para diseñar un entorno de desarrollo y pruebas adecuado para la OBC del CubeSat, fue necesario identificar todas las herramientas y dependencias que intervienen en el proceso de compilación, programación, ejecución y validación del firmware desarrollado para la placa STM32. Estas herramientas debían incluirse dentro de la imagen Docker de manera que permitieran una ejecución reproducible, automatizada y libre de conflictos con el sistema operativo anfitrión.

Los requisitos principales fueron los siguientes:

- *Toolchain* para ARM Cortex-M: se requiere el compilador `gcc-arm-none-eabi`, compatible con la arquitectura de la STM32, para compilar el código fuente en C del *firmware*.
- Herramientas de programación: es necesaria la inclusión de **OpenOCD** (*Open On-Chip Debugger*), una herramienta fundamental para establecer comunicación con la placa a

través de interfaces como SWD (del inglés, *Serial Wire Debug*), mediante el uso de programadores como ST-Link, para cargar el *firmware* en la memoria del microcontrolador.

- Python y entorno de pruebas: el sistema de validación automatizada está basado en scripts escritos en Python, por lo que se debe incluir `python3`, junto con `pip` y `venv` para gestionar entornos virtuales y dependencias como `pytest`.
- Herramientas auxiliares: se incorporan paquetes como `git` (para clonar repositorios si fuera necesario), `make` (para sistemas de construcción), `usbutils` (para diagnóstico de dispositivos conectados por USB) y editores de texto básicos como `nano`, que pueden ser útiles para inspecciones o ajustes rápidos dentro del contenedor.
- Configuración del entorno de trabajo: se define un directorio de trabajo común (`/usr/src/`) y se ajustan las variables de entorno necesarias para que el entorno virtual de Python quede disponible de forma predeterminada, facilitando la ejecución de los *scripts* de test sin pasos adicionales por parte del usuario o del *pipeline*.

La selección de estos componentes responde a la necesidad de contar con un entorno que no solo permita compilar el *firmware*, sino también ejecutar pruebas funcionales, gestionar la programación sobre la placa real y analizar el comportamiento del sistema bajo diferentes condiciones. La encapsulación de este entorno dentro de una imagen Docker asegura coherencia entre entornos de desarrollo locales y servidores de integración continua, lo cual es especialmente relevante en proyectos de sistemas embebidos donde las configuraciones pueden ser sensibles a pequeñas variaciones.

El contenido detallado del Dockerfile y una explicación más profunda se presentas en el Anexo A.

3.2. Gitlab Runner

En el desarrollo de software, especialmente en proyectos con requerimientos de alta confiabilidad como una OBC para CubeSats, resulta fundamental contar con mecanismos que automaticen la validación, prueba e implementación del código. En este contexto, la metodología de CI/CD ((*Continuous Integration / Continuous Delivery* o *Continuous Deployment*) se convierte en un factor fundamental en el flujo de trabajo.

La CI/CD es una práctica que busca integrar de forma continua los cambios de código en un repositorio central, para luego construir, validar e implementar de manera automatizada [18]. Esta cadena de pasos, conocida como *pipeline*, permite que el *software* se valide desde etapas tempranas, reduciendo los errores, acelerando los ciclos de desarrollo y facilitando la entrega de nuevas funcionalidades.

Esta metodología se compone de dos etapas principales:

- **Integración continua (CI):** Consiste en integrar frecuentemente los cambios al repositorio principal y validar automáticamente su funcionamiento mediante compilaciones y pruebas. Esto permite detectar errores temprano, evitar conflictos entre ramas y mantener un código base estable.
- **Despliegue continua (CD):** Automatiza las etapas finales del desarrollo, dejando el *software* listo para ser desplegado en cualquier entorno en cualquier momento. Incluye la validación completa, el empaquetado y, si se desea, el aprovisionamiento de la infraestructura necesaria.

Un *pipeline* de CI/CD es un conjunto de etapas automatizadas que se ejecutan ante cada cambio en el código (como un *push* al repositorio). Este *pipeline* se puede definir como código

(por ejemplo, en un archivo `.gitlab-ci.yml`) y se ejecuta automáticamente cada vez que se detecta un cambio en el repositorio.

En este trabajo, se diseñó un *pipeline* CI/CD específico para la validación del firmware de la OBC del CubeSat. Cada vez que se introduce un cambio en el repositorio, el pipeline:

1. Compila el *firmware* utilizando una imagen Docker preconfigurada.
2. Programa automáticamente la placa STM32 conectada a la Raspberry Pi.
3. Ejecuta pruebas funcionales sobre el *firmware* cargado.
4. Analiza los resultados y valida el comportamiento esperado.

Para poder ejecutar el *pipeline* de CI/CD diseñado para este proyecto, fue necesario instalar y configurar un GitLab Runner en la Raspberry Pi 5 [19], ya que esta actúa como entorno de pruebas físico y ejecuta las tareas de compilación, programación y validación sobre la STM32. Un GitLab Runner es un agente que escucha instrucciones del servidor de GitLab y ejecuta los trabajos (*jobs*) definidos en el archivo `.gitlab-ci.yml`.

Para más información acerca de cómo configurar el Gitlab Runner y cómo se definió el *pipeline* de este trabajo, consultar los Anexos B y C, respectivamente.

3.3. Herramientas

Luego de haber detallado la configuración y uso de Docker y Gitlab CI/CD, en esta sección se detallan los componentes principales del entorno de pruebas automatizado, previamente enumerados. Cada herramienta desempeña un rol específico dentro de la arquitectura del sistema, ya sea en la etapa de compilación, ejecución de pruebas, monitoreo o control remoto.

Para más información acerca de como configurar las herramientas aquí detalladas, consultar el Anexo D.

3.3.1. Raspberry Pi 5

La **Raspberry Pi** es una computadora de placa reducida (SBC, por sus siglas en inglés) desarrollada por la Fundación Raspberry Pi con el objetivo de promover la enseñanza de ciencias de la computación y facilitar el desarrollo de proyectos de automatización y control. A pesar de su tamaño compacto y bajo consumo energético, ofrece una capacidad de procesamiento considerable, conectividad flexible y una comunidad de soporte muy activa, lo que la ha convertido en una herramienta clave en entornos educativos, industriales y científicos.

En este trabajo se utilizó específicamente la Raspberry Pi 5, la versión más reciente y potente hasta el momento. Equipada con un procesador ARM Cortex-A76 de cuatro núcleos a 2,4 GHz y arquitectura de 64 bits, esta versión ofrece un rendimiento de CPU entre dos y tres veces superior al de la Raspberry Pi 4 [20]. Además, incorpora una GPU VideoCore VII de 800 MHz, soporte para doble salida de video 4Kp60, y un nuevo chip RP1 que mejora significativamente el desempeño de periféricos (USB, interfaces MIPI, tarjetas SD y PCIe).

Durante el desarrollo, la Raspberry Pi 5 actuó como nodo central de pruebas, lo que permitió mantener una experiencia fluida incluso bajo carga. Se utilizó de forma simultánea para compilar *firmware*, ejecutar *scripts*, programar la STM32, capturar datos por UART y vía USB, y mantener múltiples conexiones SSH. A pesar de esta carga, el sistema se comportó de manera estable y eficiente, facilitando el trabajo remoto y la integración continua sin interrupciones.

Conexiones físicas y esquema de prueba

Para implementar un entorno de pruebas automatizado y robusto, fue necesario establecer una forma confiable de comunicación entre la Raspberry Pi y la placa STM32. Esta última se encuentra conectada a uno de los puertos USB de la Raspberry Pi mediante un programador `ST-LINK`, lo que permite compilar y grabar el *firmware* de manera remota desde scripts ejecutados por el *GitLab Runner*.

Además de la conexión de programación, se definieron una serie de líneas digitales conectadas a pines GPIO de la Raspberry Pi con el fin de controlar qué prueba se ejecuta, habilitar su ejecución y detectar cuándo finaliza. Este mecanismo de control evita depender de interfaces como UART, I^2C o SPI, que podrían estar siendo evaluadas en las propias pruebas, lo cual eliminaría posibles interferencias o ambigüedades.

Las conexiones físicas utilizadas en el sistema se detallan a continuación:

- **GND**: conexión de referencia común entre la Raspberry Pi y la STM32.
- **ST-LINK V2 (vía USB)**: utilizado para programar y depurar la STM32 a través de la interfaz SWD.
- **UART**: líneas de transmisión y recepción para comunicación serie. Se utilizan tanto para la recepción de mensajes durante los tests como para la grabación de firmware mediante bootloader.
- **BOOT0 y NRST**: pines controlados desde la Raspberry Pi para automatizar el ingreso al bootloader serial y realizar resets programáticos.
- **TEST SELECTOR**: línea de entrada hacia la STM32. Se utiliza para enviar una cantidad determinada de pulsos digitales; el número total representa el identificador del test a ejecutar.
- **TEST ENABLE**: línea de entrada que habilita la recepción de pulsos. Si esta línea no está activa, la STM32 ignora cualquier pulso en `TEST SELECTOR`, previniendo interferencias por ruido eléctrico.
- **TEST INDICATOR**: línea de salida desde la STM32. Se activa una vez finalizado el test, permitiendo a la Raspberry Pi detectar con precisión cuándo concluyó la prueba.

La lógica general del sistema de pruebas es la siguiente:

1. Cada módulo de código posee un archivo de tipo `C` con sus funciones, su correspondiente `.h`, y un `main.c` específico que define la lógica del *test*.
2. Un mismo módulo puede tener múltiples pruebas definidas, por lo que se requiere una forma clara de seleccionar cuál debe ejecutarse. Para esto, la Raspberry Pi activa la línea `TEST ENABLE` y luego envía `N` pulsos por la línea `TEST SELECTOR`, donde `N` indica el número del *test* a ejecutar.
3. La STM32 cuenta los pulsos recibidos mientras la línea está activa, y ejecuta la prueba correspondiente.
4. Una vez finalizado el *test*, la STM32 genera un pulso en la línea `TEST INDICATOR` para informar a la Raspberry Pi que puede continuar con la siguiente etapa del pipeline.

Este esquema garantiza una comunicación robusta, completamente independiente de los periféricos internos de la STM32 que pudieran estar siendo evaluados. Además, permite una sincronización precisa con herramientas de captura como analizadores lógicos, asegurando la validez de cada prueba.

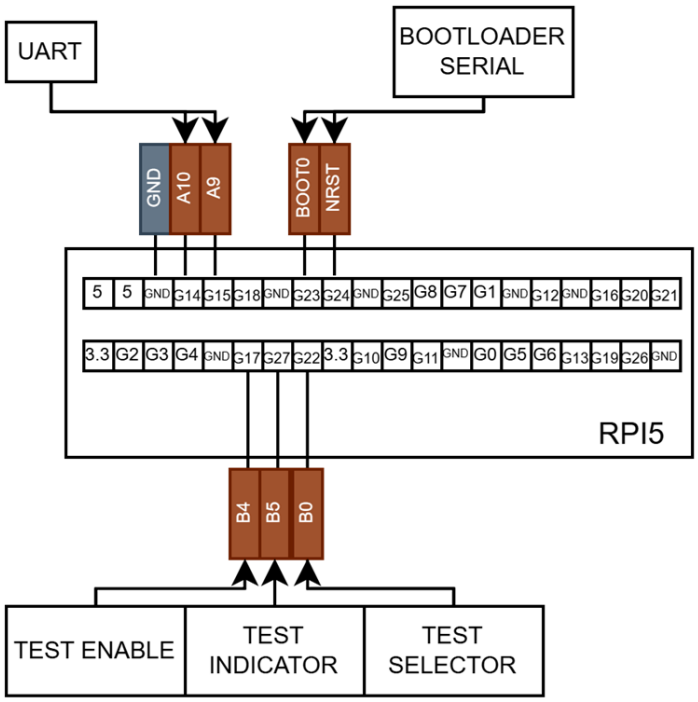

FIGURA 3.1. Conexiones entre la Raspberry Pi 5 y la placa STM32 utilizadas para pruebas y grabación.

Conexión SSH con Visual Studio Code

Existen múltiples formas de interactuar con la Raspberry Pi:

- Acceso directo mediante periféricos conectados (monitor, teclado y *mouse*).
- Acceso remoto a través de **Raspberry Pi Connect**, una herramienta oficial que permite:
  - Compartir pantalla (*Screen Sharing*).
  - Abrir una terminal remota (*Remote Shell*).
- Establecer una conexión SSH entre la Raspberry Pi y la máquina local

Si bien el acceso directo mediante periféricos es útil en ciertas situaciones, puede volverse incómodo y limitar la ubicación del dispositivo, especialmente en entornos colaborativos. Por otro lado, aunque el uso de *Screen Sharing* permite acceder visualmente al entorno gráfico, esta opción suele demandar muchos recursos y resulta poco práctica en redes con ancho de banda limitado. Como alternativa, una solución mucho más liviana y eficiente consiste en utilizar el acceso remoto por **SSH** en combinación con el editor **Visual Studio Code (VSCode)**, mediante la extensión *Remote - SSH*.

3.3.2. Tailscale

Tailscale es una herramienta que permite crear redes privadas virtuales (*VPNs*) de forma sencilla, segura y escalable. Su configuración es prácticamente automática y no requiere el manejo manual de direcciones IP, puertos o certificados. Esta solución es particularmente útil en contextos donde se necesita acceder de forma remota a dispositivos como la Raspberry Pi 5, sin exponerlos directamente a internet ni depender de configuraciones complejas de red.

En este proyecto, se utilizó Tailscale para facilitar la conexión remota segura desde cualquier ubicación, permitiendo desarrollar, monitorear y controlar las pruebas del sistema sin necesidad de acceso físico al entorno de pruebas.

3.3.3. Interruptor inteligente

Como complemento a la plataforma de acceso remoto, el uso de un **interruptor de corriente inteligente** resulta especialmente útil. Este dispositivo, conectado a la red Wi-Fi, permite controlar el encendido y apagado de la Raspberry Pi de forma remota, evitando que el sistema deba permanecer encendido de manera continua y facilitando la gestión energética del banco de pruebas.

En este proyecto se empleó como ejemplo el modelo **Pronext TI-SM05**, que ofrece compatibilidad con asistentes virtuales como Alexa y Google Assistant, además de poder ser controlado mediante la aplicación móvil *Smart Life*. También dispone de un temporizador programable para encender o apagar en un horario definido. Esta funcionalidad permitió activar o desactivar la alimentación de la Raspberry Pi desde cualquier ubicación, lo que resulta de gran utilidad para tareas de mantenimiento, reinicios programados o control remoto en contextos donde no se dispone de acceso físico directo.

3.3.4. Analizador lógico

Durante el desarrollo del entorno de pruebas para la OBC, uno de los desafíos principales fue la validación de los controladores de comunicación implementados para interactuar con sensores y periféricos a través de protocolos como I^2C, SPI y UART. Para asegurar la confiabilidad de estas implementaciones, se exploraron distintas alternativas para simular, monitorear y verificar las comunicaciones.

Una primera propuesta consistió en utilizar la propia Raspberry Pi 5 como dispositivo esclavo I^2C y SPI. La idea era que la STM32 enviara solicitudes de lectura, y la Raspberry respondiera con datos predefinidos, actuando como si fuera un sensor real. Esto permitiría comparar lo enviado con lo recibido y, de esta manera, validar el correcto funcionamiento del controlador de la OBC bajo el supuesto de que los controladores de Linux era confiable y libre de errores.

Sin embargo, esta solución se descartó por razones técnicas. Los controladores del sistema operativo Linux no permiten operar como esclavo de forma nativa. Existen librerías como `pigpio` que permiten emular un esclavo a través de técnicas de *bit-banging*, pero esto introduce limitaciones importantes:

- El *bit-banging* no utiliza el *hardware* del periférico de la Raspberry Pi, lo cual reduce la precisión en la temporización de las señales.
- Las respuestas simuladas pueden presentar comportamientos no estándar o inconsistentes en comparación con sensores reales.
- Cualquier error observado en la comunicación podría ser causado por limitaciones del entorno de emulación, y no por fallos en el código de la STM32.

Ante estas limitaciones, se adoptó una estrategia más robusta y confiable: utilizar un **analizador lógico** como herramienta de monitoreo externo, conectándolo a las líneas de comunicación entre la STM32 y los dispositivos periféricos. Esto permitió observar directamente las señales eléctricas transmitidas en tiempo real, sin intervenir en la comunicación.

En este trabajo se utilizó un analizador lógico genérico de 8 canales con una frecuencia máxima de muestreo de 24 MHz, ampliamente compatible con las herramientas del ecosistema `Sigrok`. Su uso fue esencial tanto para la detección de errores en etapas de desarrollo como para la validación completa de los controladores una vez corregidos. Gracias a esta herramienta, fue posible verificar que las tramas transmitidas cumplieran con las especificaciones de cada protocolo y que las respuestas de los sensores fueran interpretadas correctamente por la OBC.

El analizador puede emplearse mediante diferentes herramientas:

- **Logic 2**: *software* oficial de Saleae, utilizado en Windows.
- **PulseView**: interfaz gráfica multiplataforma basada en Sigrok, útil para análisis manual desde la Raspberry Pi.
- **sigrok-cli**: herramienta de línea de comandos para automatizar capturas. Esta fue la opción utilizada dentro de las pruebas.

Una de las características más destacadas de estas herramientas es su capacidad para decodificar múltiples protocolos. A partir de las transiciones lógicas capturadas, los programas pueden reconstruir direcciones, comandos, datos y confirmaciones, permitiendo una interpretación directa y de alto nivel de la comunicación observada.

Para realizar una captura de un número `NUM_MUESTRAS` de muestras, se puede ejecutar el siguiente comando desde la Raspberry Pi:

```
$   sigrok-cli -driver fx2lafw -channels D0,D1 -samples NUM_MUESTRAS
```

CÓDIGO 3.1. Medición de una cantidad fija de muestras

Donde:

- `-driver fx2lafw` especifica el controlador del dispositivo USB.
- `-channels D0,D1...` selecciona los canales a utilizar.
- `-samples NUM_MUESTRAS` indica la cantidad de muestras a capturar.

Opcionalmente, se puede agregar `-o captura.sr` para guardar los valores en un archivo. Estas capturas pueden luego abrirse en PulseView para un análisis visual detallado, con decodificadores aplicados sobre los datos adquiridos.

Es importante destacar que la frecuencia de muestreo debe seleccionarse cuidadosamente: si es demasiado baja (por ejemplo, por debajo de 20 kHz), los bordes rápidos de protocolos como I^2C pueden perderse, resultando en tramas mal interpretadas. Se recomienda utilizar, como mínimo, una frecuencia de 1 MHz.

Estas herramientas también permiten decodificar las tramas en protocolos conocidos, agregando el término `-P PROTOCOLO:LINEA1=CANAL:LINEA2=CANAL...`. Por ejemplo, para decodificar una comunicación I2C donde la línea `SCL` está conectada al canal D6 y la línea `SDA` al canal D4, se debe agregar al comando

```
    -P i2c:scl=D6:sda=D4
```

También es posible realizar capturas continuas en lugar de limitarse a un número fijo de muestras. Para ello, puede reemplazarse el parámetro

```
    -samples
```

por

```
    -config samplerate=FRECUENCIA -continuous
```

donde `FRECUENCIA` especifica la tasa de muestreo deseada, por ejemplo `1M` para 1 MHz. Esta modalidad resulta útil en pruebas prolongadas o cuando se espera una actividad específica en un intervalo no determinado con precisión.

Esta etapa de validación fue crucial para ganar confianza en el sistema antes de avanzar hacia pruebas automatizadas más complejas, donde el rol del analizador fue reemplazado, por ejemplo, por la interfaz UART.

3.4. Pytest

Para automatizar la verificación de resultados y mantener una estructura de pruebas clara y escalable, se empleó el framework **`pytest`**. Este entorno de pruebas, ampliamente utilizado en la comunidad de desarrollo en **Python**, permite escribir tests de forma sencilla y legible, pero también cuenta con herramientas poderosas para escalar hacia escenarios más complejos, incluyendo validaciones funcionales completas sobre aplicaciones o bibliotecas [21].

En el caso de este proyecto, `pytest` resultó particularmente útil para validar los resultados obtenidos tras la ejecución de cada test en la STM32. Cada prueba sigue, generalmente, una estructura común, basada en una secuencia de pasos que se repiten en la mayoría de los casos:

1. Lanzamiento del analizador lógico en segundo plano (en los casos en que es necesario capturar señales digitales).
2. Inicialización de un hilo dedicado a la lectura del puerto UART, encargado de recolectar los mensajes enviados por la STM32.
3. Envío de la instrucción correspondiente desde la Raspberry Pi hacia la STM32, indicando qué test debe ejecutarse.
4. Espera activa del pulso generado por la línea `TEST INDICATOR`, lo cual marca la finalización del test por parte de la STM32.
5. Espera adicional breve para permitir que se completen los procesos de captura de datos.
6. Terminación del proceso del analizador lógico (si se utilizó), y recolección de su salida.
7. Finalización del hilo de lectura UART y recolección de todos los mensajes recibidos.
8. Procesamiento de los datos capturados (ya sea por UART o por el analizador) y ejecución de las correspondientes validaciones mediante `asserts`.

Para lograr esta paralelización, se hace uso de herramientas de concurrencia del lenguaje Python. En particular, el módulo `threading` permite lanzar hilos secundarios (objetos `Thread`) que ejecutan funciones en paralelo al hilo principal. Esto es clave para poder escuchar la UART mientras se espera el pulso de finalización, sin que una tarea bloquee a la otra.

Además, se utilizan colas (`queue.Queue`) como mecanismos seguros de intercambio de datos entre hilos, evitando condiciones de carrera y simplificando la comunicación asincrónica.

En los casos donde se utiliza el analizador lógico, este se lanza como un subproceso independiente utilizando `subprocess.Popen`, lo que permite iniciar herramientas como `sigrok-cli` desde el entorno Python, capturar su salida y luego analizarla como parte del test.

La función de validación central de cada prueba se encuentra encapsulada en una función que comienza con el prefijo `test_`, lo cual es detectado automáticamente por `pytest`. Al finalizar, se utilizan sentencias `assert` para comparar los resultados obtenidos con los valores esperados y determinar si el test pasó o falló.

Todas estas pruebas son ejecutadas de forma automática por el *job* correspondiente del *pipeline*, detallado en la sección C. Su documentación, como así también la de las funciones codificadas para su uso, puede encontrarse en la carpeta `test_ci_cd` del repositorio.

3.5. Entorno de prueba local

Además del uso de *pipelines* automatizados mediante GitLab CI/CD, resulta fundamental contar con herramientas que permitan ejecutar pruebas de forma local durante el desarrollo. Esto facilita una iteración más rápida, especialmente en etapas tempranas, donde es necesario validar funcionalidades de manera inmediata sin esperar a la ejecución del *pipeline* remoto. Para ello, se desarrollaron dos herramientas principales: un *script* automatizador (`pipeline.sh`) y un ejecutable de prueba personalizada (`manual_pulse.py`).

3.5.1. Script de automatización: `pipeline.sh`

Este *script* escrito en **bash** permite centralizar y automatizar una serie de tareas comunes durante el desarrollo del sistema, sin depender del *pipeline* remoto. Entre sus funcionalidades se destacan:

- Configuración del entorno virtual mediante la función `configurar_venv`, que crea y activa un entorno Python aislado, instalando dependencias necesarias (recopiladas en el archivo `requirements.txt` del repositorio)
- Compilación cruzada: utilizando un contenedor Docker preconfigurado con la *toolchain* apropiada, se compila el *firmware* directamente desde el *host* sin requerir configuración adicional.
- Grabación del dispositivo, tanto usando Docker (a través de `make flash`) como mediante UART utilizando un script en Python.
- Ejecución de tests automatizados invocando el comando `testear`, que ejecuta los *tests* definidos con `pytest` directamente sobre el entorno local.
- Reinicio y limpieza: incluye opciones para aplicar un *reset* por *software* o borrar los archivos generados por la compilación.

La necesidad de este *script* radica en encapsular todas las operaciones necesarias para compilar y probar el proyecto en una sola herramienta, facilitando su uso por parte del equipo de desarrollo sin requerir conocimientos avanzados sobre la *toolchain* o configuración del entorno.

Algunos ejemplos de su uso son:

- Compilar el proyecto:

```
$   ./pipeline.sh compilar
```

- Compilar el proyecto con análisis de errores:

```
$   ./pipeline.sh debug
```

- Grabar el dispositivo por USB:

```
$   ./pipeline.sh flashear
```

- Grabar el dispositivo por UART (usando *bootloader*):

```
$   ./pipeline.sh flashear_uart
```

- Ejecutar los *tests* automatizados de `pytest`:

```
$   ./pipeline.sh testear
```

3.5.2. Pruebas manuales interactivas: `manual_pulse.py`

Este archivo en **Python** permite realizar pruebas controladas de forma manual sobre el *hardware*, con opciones para registrar la actividad mediante UART o un analizador lógico. Sus características principales incluyen:

- Envío de instrucciones personalizadas al dispositivo bajo prueba, indicando la cantidad de pulsos a generar.
- Lectura en paralelo del puerto UART, capturando posibles errores o mensajes durante la ejecución del test.
- Activación de un analizador lógico en caso de requerir análisis temporales precisos de las señales digitales generadas.
- Sincronización por eventos mediante `pulse_device.wait_for_press()`, asegurando que el test se complete antes de procesar resultados.
- Visualización de los datos capturados

Este *script* permite un nivel más fino de control durante las pruebas, útil especialmente cuando se investigan errores o se requiere validar con precisión el comportamiento del *hardware*. Su diseño modular y configurable lo hace ideal para ensayos rápidos sin necesidad de redefinir el flujo del *pipeline* automatizado.

Algunos ejemplos de su uso son:

- Ejecutar una prueba manual con 10 pulsos y lectura por UART:

```
$   python3 manual_pulse.py 10 --uart
```

- Ejecutar una prueba manual con 5 pulsos, lectura por UART y con analizador lógico:

```
$   python3 manual\_pulse.py 5 --uart --analizador
```

3.6. Compilación y grabación del firmware

El desarrollo del firmware para la STM32F103C8T6 se organiza en módulos independientes, cada uno con su propio entorno de pruebas. Para compilar estos módulos se emplea un `Makefile` personalizado, que automatiza los pasos de compilación, enlazado, conversión a binario, grabación y limpieza.

El diseño modular del Makefile permite al usuario seleccionar qué archivos fuente y cabecera incluir en cada prueba, facilitando la reutilización de código común y optimizando los tiempos de compilación. También se definen distintos *targets* como `debug`, `flash` o `clean`, según el flujo deseado. La configuración completa del Makefile y sus *flags* se detalla en el Anexo E.

Para grabar el firmware compilado en el microcontrolador se implementaron dos métodos:

1. **ST-LINK (SWD)**: utilizado durante el desarrollo, mediante la herramienta OpenOCD. Este método permite programación rápida y depuración directa a través del puerto SWD.

2. **UART (bootloader)**: pensado para el entorno espacial, donde no se dispone de acceso físico. Se utiliza el bootloader serial incorporado en los STM32 originales, que permite grabar firmware a través de USART1. El procedimiento se controla mediante un script Python (`Flash.py`) que manipula los pines `BOOT0` y `NRST` y ejecuta la herramienta `stm32loader`.

Ambos métodos están integrados al flujo de trabajo y pueden ser utilizados desde el entorno de desarrollo basado en Docker o desde la Raspberry Pi validación. Los comandos y configuraciones completas de grabación se encuentran en el Anexo F.

Capítulo 4

Protocolos

La interacción eficiente entre un microcontrolador y sus periféricos es fundamental en sistemas embebidos de alta confiabilidad, especialmente en aplicaciones espaciales como la computadora de abordo de un satélite. En este capítulo se analiza el desarrollo e implementación de *drivers* para tres protocolos de comunicación esenciales: UART, I2C y SPI. Mientras que en secciones anteriores se abordaron los fundamentos de comunicación en sistemas embebidos, aquí se profundiza en su implementación práctica integrada con un sistema operativo en tiempo real, en particular FreeRTOS, con el objetivo de garantizar robustez y eficiencia operativa.

El análisis cubre las diferentes alternativas evaluadas para cada protocolo, destacando sus ventajas, limitaciones e integración con los mecanismos del *kernel* de FreeRTOS, como colas, semáforos y notificaciones. Se presenta inicialmente una implementación básica de UART mediante *polling* dentro de una tarea, seguida de su optimización mediante interrupciones, evaluando su interacción con el *scheduler* y los mecanismos de sincronización del RTOS.

La implementación de estos protocolos debe considerar no solo parámetros técnicos como ancho de banda, configuración de *hardware* y eficiencia energética, sino también su operación en entornos exigentes, con posibles interferencias electromagnéticas y restricciones de tiempo real. Este capítulo explora las estrategias de implementación adaptadas a la OBC, con especial atención en el manejo de errores y la optimización del rendimiento mediante las capacidades nativas de FreeRTOS.

4.1. UART

El protocolo **UART (del inglés *Universal Asynchronous Receiver-Transmitter*)** es un estándar de comunicación serial asincrónico ampliamente utilizado en sistemas embebidos y microcontroladores. A diferencia de protocolos sincrónicos como I^2C o SPI, UART no requiere una señal de reloj compartida entre los dispositivos, lo que simplifica la conexión física y reduce la cantidad de líneas necesarias. Fue desarrollado originalmente para la transmisión de datos en computadoras y terminales, y desde entonces se ha consolidado como una solución robusta y versátil en una gran variedad de aplicaciones electrónicas.

Una de las principales ventajas de UART radica en su simplicidad de implementación y su compatibilidad casi universal: como se observa en la Figura 4.1, basta con dos líneas —una de transmisión (**TX**) y otra de recepción (**RX**)— para establecer una comunicación punto a punto entre dos dispositivos. Esta arquitectura simplificada lo convierte en una opción ideal para enlaces directos entre microcontroladores, módulos GPS, sensores inteligentes, transmisores, entre otros periféricos.

En una comunicación con este protocolo, ambos dispositivos deben acordar previamente parámetros como la velocidad de transmisión (*baud rate*), la cantidad de bits de datos, la presencia de bits de paridad y los bits de parada. Estos parámetros deben coincidir exactamente para garantizar una comunicación fiable, ya que la ausencia de una señal de reloj

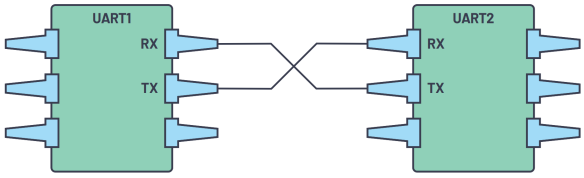

FIGURA 4.1. Dos interfaces UARTs interconectadas [22]

compartida implica que la sincronización se basa únicamente en los tiempos establecidos por ambos extremos.

A pesar de ser un protocolo de tipo *full-duplex*, es decir, capaz de transmitir y recibir simultáneamente, UART no permite la conexión directa de múltiples dispositivos en el mismo canal sin circuitos adicionales como multiplexores. No obstante, su bajo consumo, alta fiabilidad y facilidad de uso lo hacen especialmente adecuado para sistemas embebidos con recursos limitados.

La velocidad de transmisión puede variar desde unos pocos cientos de bits por segundo hasta varios megabits por segundo, dependiendo del microcontrolador, la longitud del cable y la calidad del enlace. Por ejemplo, la STM32F103C8T6 puede operar UARTs a velocidades superiores a 1 Mbit/s con la configuración adecuada del reloj y las líneas físicas correctamente diseñadas.

Actualmente, casi todos los microcontroladores modernos incluyen periféricos UART dedicados, como el bloque `USARTx` en la familia STM32, que permiten gestionar automáticamente tareas como la detección de errores de paridad, el control de flujo por *hardware* (*RTS/CTS*, del inglés *Request to Send / Clear to Send*) o la generación de interrupciones al recibir nuevos datos, lo que mejora la eficiencia del sistema al evitar el *polling* constante.

En el presente proyecto se hará uso intensivo del protocolo UART para establecer comunicación entre la computadora de abordo del CubeSat y periféricos externos como el módulo GPS y otros microcontroladores, además de su posible uso en futuros subsistemas o sensores. Se detallarán tanto los aspectos teóricos del protocolo como su implementación práctica utilizando la biblioteca `libopencm3` sobre la plataforma STM32.

4.1.1. Trama UART

La comunicación UART se basa en la transmisión asincrónica de tramas estructuradas, donde cada paquete de datos incorpora elementos esenciales para garantizar la sincronización, integridad y delimitación de la información. A continuación se detalla la composición de una trama típica, analizando desde el bit de inicio —que marca el comienzo de la transmisión— hasta los bits de parada, pasando por el campo de datos, el bit de paridad y los mecanismos que permiten la detección de errores.

FIGURA 4.2. Paquete UART [22]

Una trama típica comienza con un **bit de inicio** (*start bit*). Este bit es generado por el transmisor al llevar la línea **TX** desde un estado de reposo (nivel alto) a un estado activo (nivel bajo), lo que indica al receptor que se aproxima una nueva secuencia de datos. Este flanco descendente es utilizado por el receptor para iniciar el muestreo del resto de la trama, basado en su reloj interno.

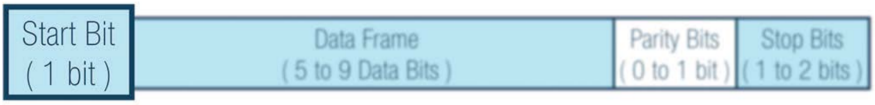

FIGURA 4.3. Bit de inicio [22]

A continuación del bit de inicio, se transmiten los **bits de datos** (*data frame*). El número de bits por palabra de datos suele ser configurable entre 5 y 8, si se usa bit de paridad y 9 si es que no se usa, aunque los valores más comunes son 8 o 7 bits. En la mayoría de los casos, los datos se transmiten en formato **little-endian**, es decir, el bit menos significativo (*LSB*) se transmite primero.

FIGURA 4.4. Bits de datos [22]

Luego del *data frame*, entra en juego el **bit de paridad** (*parity bit*). Este dígito describe el carácter par o impar de los datos para que el receptor sepa si algún dato cambió durante la transmisión. Esto puede deberse a varios motivos, como por ejemplo radiación electromagnética (muy común en entornos espaciales), velocidades de transmisión desajustadas o transferencias de datos a largas distancias. Esta última razón no suele ser el principal problema dentro de sistemas embebidos o desarrollos compactos, pero sí en comunicaciones entre dispositivos que se encuentran muy separados entre sí.

Una vez que el receptor UART ha capturado la trama de datos, procede a contar cuántos bits tienen el valor lógico alto (1). Luego, verifica si esa cantidad es par o impar. En caso de que se haya configurado paridad par (bit de paridad en 0), el número total de bits en 1 dentro de la trama debería ser par. Por el contrario, si se usa paridad impar (bit de paridad en 1), entonces la cantidad de bits en 1 debe ser impar. Si el valor del bit de paridad concuerda con el recuento de bits altos, la UART asume que la transmisión se realizó sin errores. Sin embargo, si el bit de paridad es 0 pero la cantidad de bits en 1 es impar, o si el bit de paridad es 1 pero el total resulta ser par, la UART detecta que ocurrió un error en los datos transmitidos.

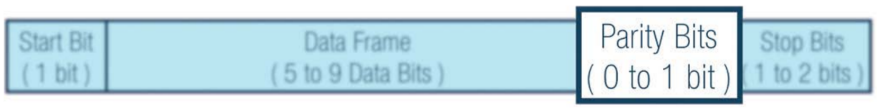

FIGURA 4.5. Bits de paridad [22]

Finalmente, la trama se completa con uno o dos **bits de parada** (*stop bits*) dependiendo de la configuración, que se transmiten en nivel alto. Estos bits sirven para delimitar claramente el final de la trama y para garantizar un mínimo de tiempo de inactividad entre dos transmisiones consecutivas, permitiendo que el receptor resincronice su lógica de muestreo si es necesario [22].

FIGURA 4.6. Bits de parada [22]

Una vez establecida la configuración, la comunicación UART es completamente **bidireccional** y puede operar en modo *full-duplex*. Es decir, un dispositivo puede estar recibiendo datos por su línea RX mientras transmite simultáneamente por su línea TX. No obstante, a diferencia de I^2C, UART no incluye mecanismos nativos de direccionamiento ni de confirmación de recepción (ACK/NACK). Por esta razón, en aplicaciones donde se requiere confiabilidad, suelen implementarse protocolos de nivel superior sobre UART o protocolos personalizados que incluyan sumas de verificación y respuestas del receptor.

4.1.2. UART vs USART

En el contexto del desarrollo con microcontroladores es habitual encontrar referencias tanto a UART como a **USART (del inglés *Universal Synchronous/Asynchronous Receiver/-Transmitter*)**, lo que puede llevar a cierta confusión entre ambos términos. Aunque suelen usarse indistintamente en la práctica, especialmente cuando se trabaja en modo asincrónico, es importante establecer una distinción clara entre ambos conceptos y comprender por qué se suele optar por UART en ciertas ocasiones por sobre una comunicación sincrónica.

USART es una interfaz más versátil, ya que puede operar tanto en modo asincrónico (como UART) como en modo sincrónico, donde se transmite además una señal de reloj junto con los datos. En este modo, el receptor puede sincronizarse directamente con el transmisor usando esta señal de reloj, permitiendo velocidades potencialmente más altas y una sincronización más precisa.

La elección entre UART y USART depende del contexto y de las necesidades específicas del sistema embebido. En muchos casos se prefiere UART por razones prácticas:

- Simplicidad del hardware: UART requiere solo dos líneas (TX y RX) para funcionar, lo que reduce el número de pines y simplifica la topología del sistema, algo muy valorado en diseños con restricciones de espacio o recursos limitados.
- Compatibilidad extensa: La mayoría de sensores, módulos periféricos (como módulos GPS, Bluetooth, WiFi, etc.) y microcontroladores disponibles comercialmente usan UART como su forma principal o única de comunicación serie.
- Adecuado para comunicaciones punto a punto: En muchos sistemas embebidos, los enlaces UART son punto a punto (ej. microcontrolador - sensor), donde el uso de un reloj compartido no ofrece ventajas significativas.
- Uso eficiente de recursos: Dado que el UART no necesita una señal de reloj adicional, el uso de líneas GPIO y recursos del sistema (como DMA[1]. o interrupciones) es más eficiente. Esto es relevante en ciertos contextos donde se busca optimizar el uso de periféricos y energía. UART puede funcionar con DMA o interrupciones sin necesidad de gestionar un reloj adicional. Esto libera ciclos de CPU para otras tareas. En cambio, el USART puede requerir más configuración para coordinar el reloj con las interrupciones, aumentando la carga computacional.

Sin embargo, hay escenarios en los que el modo sincrónico de USART es preferible o incluso necesario:

- Velocidades de transmisión más altas y precisas: El modo sincrónico permite sincronizar los datos con una señal de reloj compartida, lo que reduce el margen de error en la

[1]DMA (*Direct Memory Access*) es una característica de *hardware* que permite a ciertos periféricos del microcontrolador (como UART) leer o escribir directamente en la memoria RAM sin que la CPU tenga que intervenir en cada transferencia de dato

sincronización bit a bit, especialmente a velocidades altas. Ejemplo: si se está diseñando un sistema de adquisición de datos a muy alta velocidad (por ejemplo, mayor a 4 Mbps), el modo sincrónico puede ser más fiable.

- Transmisión más estable y con menos *jitter*: Como el receptor sigue el mismo reloj que el transmisor, hay menos dependencia de la precisión de los osciladores en ambos extremos. Esto elimina el problema típico del UART que es el error acumulado por diferencias de reloj en transmisor y receptor (*baud rate mismatch*).
- Comunicación con dispositivos que requieren sincronía: Algunos periféricos digitales avanzados, ASICs [2] o protocolos industriales personalizados requieren una señal de reloj durante la transmisión, lo cual solo es posible con USART en modo sincrónico. En sistemas tiempo-real o críticos, como automatización industrial, la sincronía garantiza que cada bit llegue exactamente en el momento esperado.
- Diseños con múltiples receptores (*broadcast*) En sistemas donde un maestro transmite datos a varios esclavos simultáneamente, usar USART en modo sincrónico garantiza que todos los receptores lean los bits al mismo tiempo gracias al reloj compartido.

4.1.3. USART modo sincrónico en STM32

El modo sincrónico se selecciona escribiendo el bit CLKEN del registro `USART_CR2` a 1. En modo sincrónico, los siguientes bits deben mantenerse despejados:

- bit LINEN en el registro `USART_CR2`,
- bits SCEN, HDSEL e IREN en el registro `USART_CR3`.

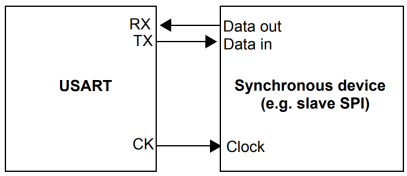

FIGURA 4.7. USART ejemplo de transmisión sincrónico [23]

USART permite establecer una comunicación serie síncrona bidireccional en modo maestro, utilizando el pin CK como salida de reloj. La emisión de pulsos en CK se omite durante el bit de arranque, el bit de parada y en las fases de espera, preámbulo o envío. La presencia de un pulso en el último bit de datos depende del bit LBCL, mientras que los bits CPOL y CPHA configuran la polaridad y fase del reloj, respectivamente, a través del registro `USART_CR2`.

En modo sincrónico, el transmisor USART funciona exactamente igual que en modo asincrónico pero como CK está sincronizado con TX (según CPOL y CPHA), los datos en TX son sincrónicos.

Si bien el funcionamiento por defecto que se planteó para las pruebas y conexiones es con la interfaz en modo asincrónico, se implementó la posibilidad para que el usuario pueda configurar al microcontrolador en modo sincrónico maestro, utilizando la señal del clock para marcar la transmisión por `Tx`.

[2] del inglés *Application-Specific Integrated Circuit* (circuito integrado de aplicación específica), es decir, un circuito personalizado diseñado para realizar funciones concretas.

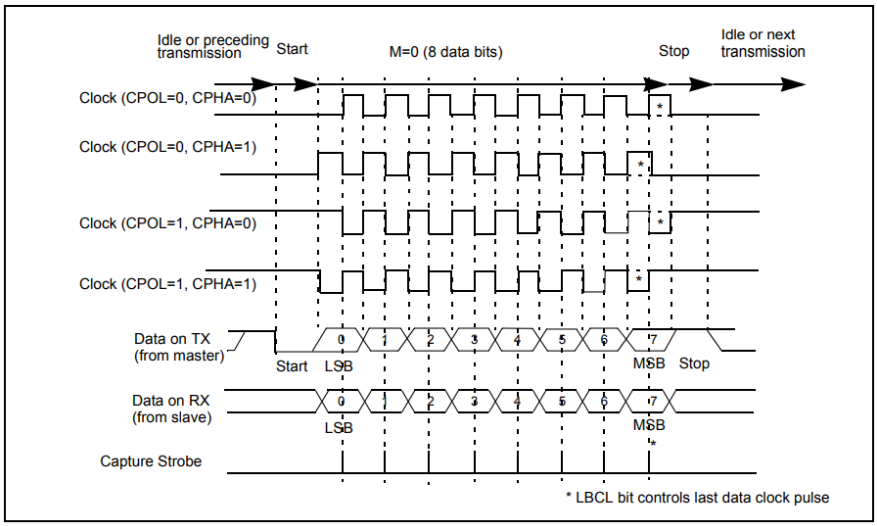

FIGURA 4.8. USART diagrama y datos del muestreo con clock. M = 0.

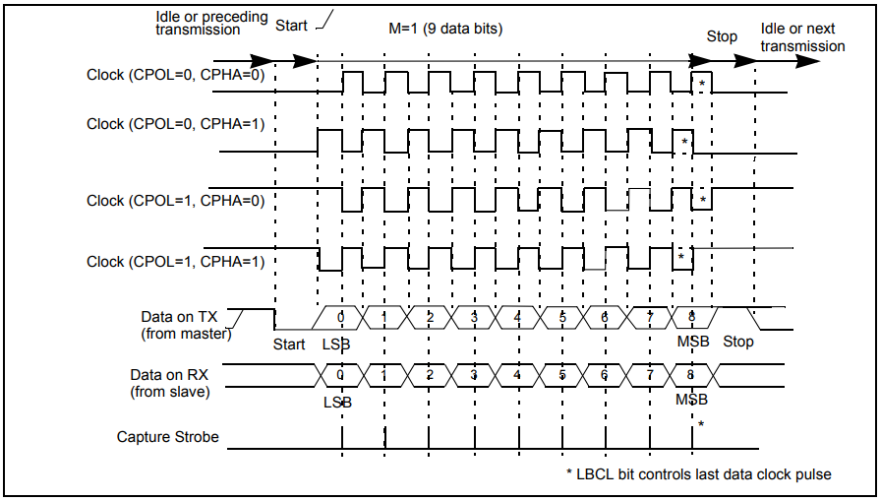

FIGURA 4.9. USART diagrama y datos del muestreo con clock. M = 1.

Esta elección se basa en que la configuración de la Rapsberry del entorno de pruebas para que funcione en modo USART esclavo requería una capa de lógica de control extra y tampoco se podía hacer una conexión en modo *loopback* sobre la misma STM32. Por este motivo, se realizaron mediciones con el analizador lógico para el caso sincrónico. La figura 4.10 muestra el envío de 4 *bytes* de datos y se observa que el reloj no envía señal en los bits de *start* ni *stop*, pero sí en los de datos.

En las Figuras 4.11 y 4.12, se observa la diferencia que existe en los tiempo de transmisión para la misma interfaz USART, utilizando dos *baudrates* distintos. En el primer escenario, la configuración de 9600 bps implica un tiempo de transmisión de aproximadamente 27 ms para la trama completa, mientras que en el segundo caso, al incrementar la velocidad a 921600 bps, la misma información se transmite en apenas 0,84 ms, reduciendo el tiempo de ocupación del *bus* en un 96,9 %.

Aunque esta diferencia podría parecer marginal en sistemas con recursos abundantes, adquiere especial relevancia en arquitecturas de un único procesador, donde el mismo núcleo debe gestionar de forma concurrente tareas críticas como la medición de sensores, el procesamiento de datos y la generación de respuestas en tiempo real. En tales entornos, un

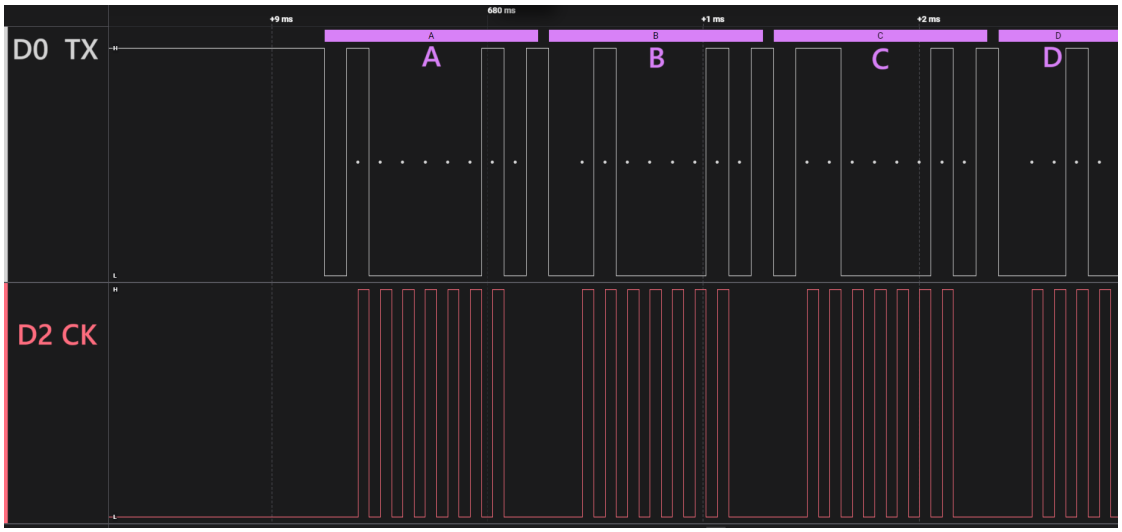

FIGURA 4.10. Transmisión USART sincrónico

retraso acumulado en las comunicaciones puede provocar que las tareas de menor prioridad sufran *starvation*, al verse desplazadas por procesos bloqueantes asociados al manejo de interfaces serie.

Un ejemplo ilustrativo es el caso de un satélite que emplea una cámara de imágenes para transmitir datos a la estación terrestre. Si el *payload* generado requiere varios minutos para su transmisión —incluso a velocidades elevadas—, cualquier ineficiencia en la capa física (como el uso de *baudrates* subóptimos) agrava exponencialmente los cuellos de botella, comprometiendo la ventana temporal disponible para otras operaciones. Este escenario demuestra por qué la selección de velocidades de transmisión adecuadas es un factor crítico en sistemas donde la latencia y el aprovechamiento de recursos son determinantes para el cumplimiento de los requisitos operativos.

Las tramas se capturan utilizando un analizador lógico.

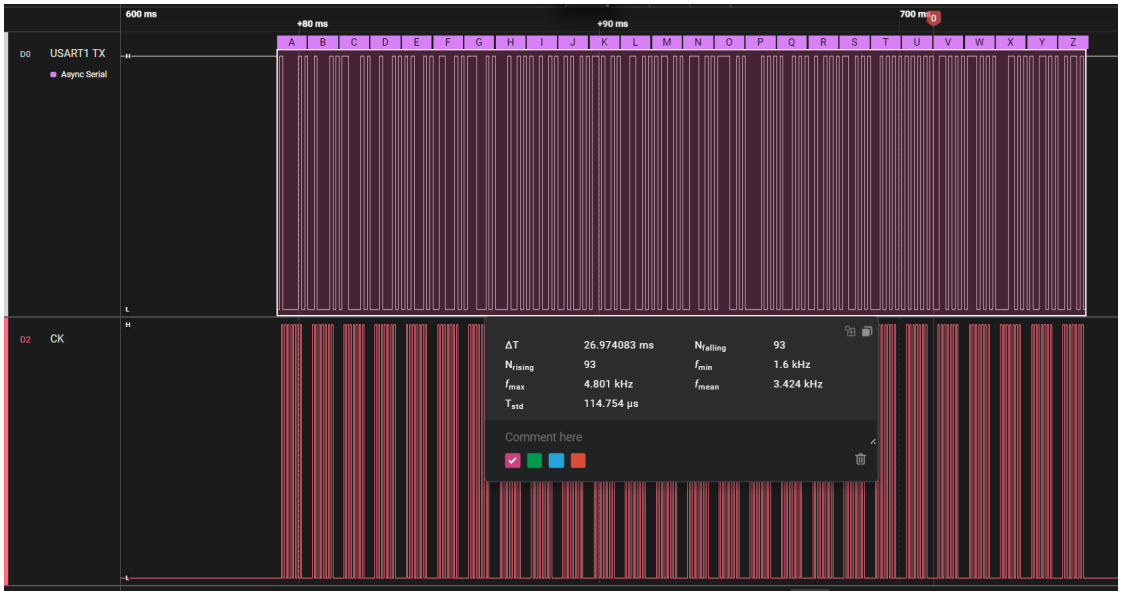

FIGURA 4.11. Transmisión USART sincrónico 9600 baudios

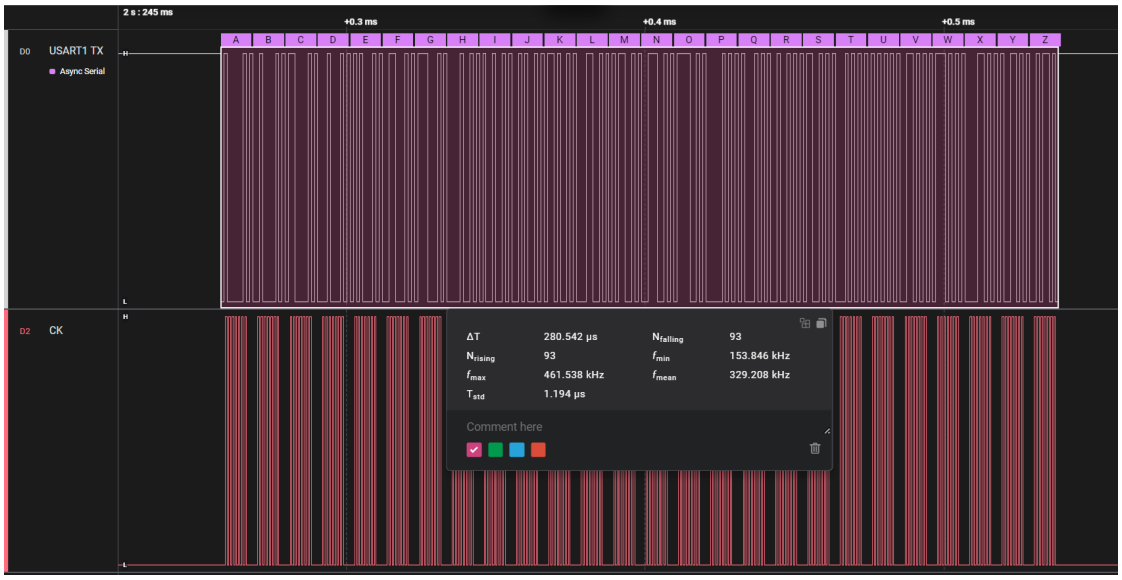

FIGURA 4.12. Transmisión USART sincrónico 921600 baudios

4.1.4. Importancia de la implementación del protocolo UART en satélites

En el diseño de sistemas embebidos para satélites pequeños, la interfaz **UART** se utiliza ampliamente como medio de comunicación entre módulos internos, gracias a su simplicidad, bajo consumo de recursos y alta confiabilidad en entornos hostiles. Estas características la hacen ideal para conectar componentes como la OBC, sensores, cámaras, almacenamiento y subsistemas de telemetría [24].

Como se mencionó, al no requerir de una línea de reloj compartida entre los dispositivos transmisor y receptor, esta característica no solo reduce el número de líneas físicas necesarias —lo cual es especialmente valioso en plataformas donde el espacio y los pines de E/S son limitados— sino que además simplifica el diseño de las placas y disminuye el consumo de energía. Estas ventajas lo convierten en una solución altamente viable para la mayoría de los enlaces internos del satélite, donde la sincronización precisa puede lograrse mediante una correcta configuración del baud rate.

Una de las razones clave para su adopción generalizada en aplicaciones espaciales es su integración directa en la mayoría de los microcontroladores modernos, como los basados en arquitecturas ARM Cortex-M (por ejemplo, el STM32 utilizado en este proyecto y muchos otros [25]), los cuales incorporan múltiples periféricos USART (que soportan ambos modos de operación), facilitando así la transmisión y recepción de datos sin necesidad de *hardware* adicional.

En sistemas satelitales, este protocolo se utiliza principalmente para:

- Transmitir datos de sensores al OBC.
- Enviar comandos desde el OBC a subsistemas como módulos de control de orientación o cargas útiles como lo podría ser una cámara.
- Establecer comunicación entre el OBC y el subsistema de telemetría o el transceptor de radiofrecuencia.
- Facilitar la depuración y el monitoreo durante la fase de integración y pruebas funcionales en tierra.

Desde el punto de vista de la arquitectura de software, UART permite una integración eficiente con sistemas operativos en tiempo real, como FreeRTOS, ya que su naturaleza asincrónica se adapta bien al uso de colas de mensajes, interrupciones y mecanismos como DMA, lo que optimiza el rendimiento sin bloquear el procesador [26].

La tolerancia del protocolo a pequeñas variaciones de sincronización y su baja complejidad hacen que sea especialmente robusto en ambientes ruidosos o en condiciones críticas, como aquellas encontradas en órbita baja terrestre (LEO). Asimismo, la ausencia de necesidad de direccionamiento complejo o mecanismos de arbitraje lo vuelve ideal para topologías simples y deterministas, donde la confiabilidad es prioritaria sobre la flexibilidad topológica. En otras palabras, aplica para los casos que se necesita un entorno y arquitectura confiable tratando de reducir al mínimo la complejidad de conexiones.

Si bien existen protocolos más avanzados (como CAN, SPI o I^2C) para aplicaciones específicas, UART sigue siendo una piedra angular en el diseño de sistemas de comunicación internos para plataformas satelitales pequeñas debido a su comprobada eficacia, tanto en tierra como en condiciones operacionales reales en órbita [27].

4.1.5. Interfaz USART en la STM32

El microcontrolador STM32F103C8T6 dispone de tres módulos USART (USART1, USART2 y USART3), que permiten comunicación full dúplex en modo asincrónico y salida de reloj para transmisión síncrona. USART1 opera sobre el *bus* APB2, lo que le permite alcanzar velocidades de hasta $4,5$ Mbit/s, mientras que USART2 y USART3, sobre el *bus* APB1, alcanzan hasta $2,25$ Mbit/s.

Entre sus características destacadas se encuentran: generación fraccionada de velocidad en baudios mediante el registro `USART_BRR`, soporte para tramas de 8 o 9 bits, configuración de 1 o 2 bits de stop, almacenamiento en SRAM usando DMA, y habilitación independiente de transmisor y receptor.

En estos microcontroladores[23], los periféricos USART integrados permiten configurar todos los parámetros descritos anteriormente mediante registros como `USART_CR1` (*Control register 1*), `USART_CR2` (*Control register 2*) y `USART_BRR` (*Baud rate register*) entre otros. Además, estos módulos permiten generar interrupciones ante eventos como la recepción de un nuevo byte con RXNE (*Rx buffer not empty*), la habilitación de la transmisión con TE (*transmit enable*), la finalización de la transmisión con TC (*transmission complete*), o la detección de errores (ORE - *Overrun error* , PE - *Parity error*, FE - *Frame error*), facilitando el diseño de sistemas eficientes basados en interrupciones o DMA.

4.1.6. Configuración del puerto USART

El microcontrolador `STM32F103C8T6` de la familia STM32 de STMicroelectronics, ofrece tres módulos **USART**. Estos permiten la implementación de interfaces en modo asincrónico o en modo sincrónico, adaptándose a una amplia variedad de aplicaciones embebidas. A continuación, se detallan las configuraciones y conexiones relacionadas a cada uno.

4.1.7. Implementación

El periférico USART en los microcontroladores STM32 ofrece una implementación flexible que admite tanto modo sincrónico como asincrónico, permitiendo adaptarse a diversos escenarios de comunicación serial. En su configuración básica, el modo asincrónico (UART) es el más utilizado, ya que requiere únicamente dos líneas (TX y RX) para la transmisión de datos, mientras que el modo sincrónico añade una señal de sincronización (CK) para entornos donde el *timing* debe ser más estricto. Además, STM32 permite habilitar señales

de control de flujo *hardware* (RTS y CTS), útiles para evitar pérdida de datos en sistemas con diferencias significativas en las velocidades de procesamiento entre el transmisor y el receptor.

Mientras que el modo asincrónico es ideal para comunicaciones simples con dispositivos periféricos, el modo sincrónico resulta ventajoso en entornos donde la integridad temporal de los datos es prioritaria, como en interfaces con sensores de alta velocidad o sistemas maestro-esclavo. La elección entre una implementación basada en *polling*, interrupciones o DMA dependerá de los requisitos de eficiencia y capacidad de respuesta del sistema, siendo el DMA la opción preferida para minimizar la carga de CPU en aplicaciones de alto rendimiento.

Envío de Datos: Modo Bloqueante vs No Bloqueante

En los sistemas embebidos, la transmisión de datos a través de interfaces como USART puede realizarse utilizando funciones bloqueantes o no bloqueantes. La elección entre una u otra afecta directamente al comportamiento del sistema, el rendimiento en tiempo real y la complejidad de implementación.

De una u otra manera, `libopencm3` ofrece una abstracción para hacer el envío de datos, la cual internamente hace el manejo de los registros e información a enviar.

```
/**
 * @brief USART Send a Data Word.
 *
 * @param usart Direccion base del periferico USART (por ejemplo, USART1)
    .
 * @param data Dato de 8 o 9 bits a transmitir.
 */
void usart_send(uint32_t usart, uint16_t data)
{
    /* Send data. */
    USART_DR(usart) = (data & USART_DR_MASK);
}
```

CÓDIGO 4.1. usart_send en libopencm3

Transmisión Bloqueante

Una función bloqueante detiene la ejecución del programa hasta que la operación de transmisión se completa. Este enfoque es común en implementaciones simples debido a su facilidad de uso, pero puede ser ineficiente en aplicaciones donde el procesador debe atender múltiples tareas.

```
void usart_send_blocking(uint32_t usart, uint16_t data)
{
        usart_wait_send_ready(usart);
        usart_send(usart, data);
}
```

CÓDIGO 4.2. Ejemplo de transmisión bloqueante

En este ejemplo, la función `usart_send_blocking` hace uso de la `usart_wait_send_ready` la cual espera a que el registro de transmisión esté listo antes de enviar el carácter, y no retorna hasta que el byte haya sido transmitido correctamente.

La implementación de UART mediante polling destaca por su sencillez de implementación, ya que no requiere configurar interrupciones ni gestionar buffers complejos. Además,

garantiza que el dato ha sido enviado completamente antes de continuar con la ejecución del programa, lo que evita problemas de sincronización en operaciones secuenciales.

Sin embargo, este enfoque es ineficiente en sistemas multitarea o con requerimientos de tiempo real, ya que el procesador queda bloqueado esperando la finalización de la transmisión, impidiendo la ejecución de otras tareas durante ese período.

Transmisión No Bloqueante

Una función no bloqueante intenta realizar la transmisión sin detener el flujo del programa. Si el periférico no está listo, el programa continúa su ejecución normal, permitiendo una mayor eficiencia y concurrencia, especialmente en sistemas con un sistema operativo en tiempo real (RTOS) o uso intensivo de interrupciones.

En el código de 2.2, el programa verifica si el periférico está listo para transmitir mediante el flag `TXE` (*Transmitter Empty*). Si el flag está activo, se transmite el carácter; en caso contrario, se omite la operación o se difiere para otro momento.

Entre las principales ventajas del uso de funciones no bloqueantes se encuentra la mejora en la eficiencia del sistema, ya que permiten que el procesador continúe ejecutando otras tareas mientras se completa la transmisión. Este enfoque resulta especialmente útil en arquitecturas concurrentes o cuando se utiliza un sistema operativo en tiempo real. Sin embargo, también presenta ciertas desventajas como la de requerir una lógica adicional para gestionar adecuadamente la disponibilidad del periférico (ya sea mediante un *mutex* o un semáforo) y no garantiza que el dato sea transmitido de forma inmediata, lo que puede implicar la necesidad de mecanismos de control o verificación adicionales a nivel de *software*.

Opciones analizadas

Como fue mencionado en la sección 2.3.9, para la implementación del protocolo no solo es necesario el correcto uso de las funciones del microcontrolador, sino también su integración con objetos propios del *kernel*.

En este caso, se definen tres estructuras `uart_t`, una por cada interfaz (`USART1`, `USART2`, `USART3`) con su respectivo inicializador que se ejecuta al momento de la configuración (`USART_setup`).

```
// Definicion de estructuras UART
static uart_t uart1;
static uart_t uart2;
static uart_t uart3;
```

CÓDIGO 4.3. Estructuras `USART`

Dentro de cada una de estas se encuentra un identificador (propio de `libopencm3`) para reconocer a qué interfaz se refiere, dos estructuras de tipo Cola (`xQueue`) para el almacenamiento de datos tanto en envío como en recepción, y dos señalizadores del tipo *mutex* y semáforo, (`xSemaphoreMutex` y `xSemaphoreBinary` respectivamente), para la protección de acceso a recursos y señalización de datos.

```
typedef struct {
    uint32_t usart;  // Identificador del USART.
    QueueHandle_t txq;  // Cola para la transmision de datos.
    QueueHandle_t rxq;  // Cola para la recepcion de datos.
    SemaphoreHandle_t mutex;  // Mutex para proteger el acceso a recursos
     compartidos.
    SemaphoreHandle_t semaphore; // Semaph para avisar disponibilidad de
   datos en rxq.
```

```
} uart_t;
```

CÓDIGO 4.4. Definición estructura para periférico USART

Se comienza con un identificador `uint32_t usart` que se utiliza para identificar la estructura y asignarla a una interfaz particular. Este dato se completa con la etiqueta nativa de la librería `USART1, USART2, USART3` y se utiliza la misma para que, al momento de invocar una función por fuera del archivo en cuestión, se pueda llamar dinámicamente enviando esta etiqueta como parámetro y de esa forma poder elegir con qué canal trabajar sin necesidad de repetir código.

En la cola de transmisión `txq` se guardan los datos que se desean enviar por la interfaz correspondiente. Para esto, existe a su vez una tarea `taskUART_transmit` que, cada vez que sea seleccionada por el *scheduler*, se encarga de desencolar los datos a transmitir y enviarlos utilizando las funciones nativas de `libopencm3`. Por esto último es importante una buena gestión de la prioridad de la tarea, para que se logre una transmisión fluida y sin errores.

Este parámetro de prioridad permite una gestión dinámica de recursos por parte del *scheduler*. Por ejemplo, en el caso de la cola de transmisión, la tarea encargada de desencolar datos puede iniciar con una prioridad base, pero, a medida que el *buffer* se aproxima a su capacidad máxima, el *scheduler* puede elevar su prioridad para acelerar el procesamiento y evitar *overflow*. Por el contrario, si el *buffer* se encuentra vacío y no demanda recursos inmediatos, el sistema puede reducir la prioridad de dicha tarea, optimizando así la asignación de capacidad de procesamiento.

En la cola de recepción `rxq` se guardan los datos entrantes. Para esto se desarrolla una lógica de recepción por interrupción, la cual tiene una declaración y definición genérica, pero se invoca desde cualquiera de las tres interrupciones (`usart1_isr`, `usart2_isr`, `usart3_isr`). A continuación se muestra la lógica para que un dato recibido se encole en el buffer `rxq` correspondiente:

Recepción por interrupción

La implementación de recepción por interrupción de USART, como la mostrada a continuación, presenta beneficios por sobre el enfoque de *polling* al resolver problemas críticos que este último presenta. Mientras el *polling* requiere consultas constantes al periférico -desperdiciando ciclos de CPU y energía-, las interrupciones permiten un manejo eficiente donde el procesador solo actúa cuando hay datos disponibles. Esto es particularmente importante en dispositivos con restricciones de tiempo y recursos de procesamiento.

Un riesgo clave del *polling* es la pérdida de datos: al no contar con un búfer automático por parte del *hardware* más allá del registro de datos (DR) y el bit de estado RXNE, en muchos microcontroladores básicos si llegan dos palabras consecutivas entre ciclos de consulta, la segunda puede sobrescribir a la primera antes de ser leída. Las interrupciones eliminan este problema al atender cada byte inmediatamente.

La solución presentada combina las interrupciones *hardware* con las primitivas de FreeRTOS. Usando colas y semáforos desde la ISR (con las APIs _FromISR), se logra un puente eficiente hacia las tareas de aplicación. El parámetro `xHigherPriorityTaskWoken` optimiza este proceso, gestionando los cambios de contexto solo cuando es estrictamente necesario.

```
void usart1_isr(void) {
    usart_generic_isr(USART1);
}
void usart2_isr(void) {
    usart_generic_isr(USART2);
}
```

```
void usart3_isr(void) {
    usart_generic_isr(USART3);
}

static void usart_generic_isr(uint32_t usart_id) {
    uart_t *uart = get_uart(usart_id);
    if (uart == NULL) return;

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    while (usart_get_flag(uart->usart, USART_SR_RXNE)) {
        uint16_t data = usart_recv(uart->usart); // No usar version
    blocking en ISR

        if (xQueueSendToBackFromISR(uart->rxq, &data, &
    xHigherPriorityTaskWoken) == pdTRUE) {
            xSemaphoreGiveFromISR(uart->semaphore, &
    xHigherPriorityTaskWoken);
        }
    }

    // Si una tarea de mayor prioridad fue despertada, forzar cambio de
    contexto
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

CÓDIGO 4.5. Esquema de recepción de datos `USART`

El parámetro `xHigherPriorityTaskWoken` es un mecanismo fundamental en FreeRTOS para gestionar eficientemente las prioridades de las tareas cuando se ejecutan operaciones desde una rutina de servicio de interrupción. Su función principal es indicar si una tarea de mayor prioridad ha sido despertada, es decir, si ha pasado del estado *Blocked* al estado *Ready*, como consecuencia de una operación realizada dentro de la ISR. Este mecanismo es crucial para optimizar el rendimiento del sistema, ya que evita cambios de contexto innecesarios y asegura que las tareas de mayor prioridad se ejecuten lo antes posible.

Al inicio de una ISR, `xHigherPriorityTaskWoken` se inicializa con el valor **pdFALSE**, lo que significa que, en ese momento, no se ha despertado ninguna tarea de mayor prioridad. A medida que la ISR ejecuta operaciones, como enviar datos a una cola mediante `xQueueSendToBackFromISR` o liberar un semáforo con `xSemaphoreGiveFromISR`, estas funciones pueden modificar el valor de `xHigherPriorityTaskWoken` a **pdTRUE** si la operación despierta una tarea cuya prioridad es superior a la de la tarea que se estaba ejecutando antes de que ocurriera la interrupción.

Al finalizar la ISR, se verifica el valor de la variable. Si este es **pdTRUE**, se invoca la función `portYIELD_FROM_ISR()`, la cual fuerza un cambio de contexto inmediato. Esto permite que la tarea de mayor prioridad, que fue despertada durante la ejecución de la ISR, comience a ejecutarse sin demora. Este enfoque es especialmente importante en sistemas de tiempo real, donde la capacidad de responder rápidamente a eventos críticos es esencial para garantizar el determinismo y la eficiencia del sistema.

4.1.8. Entorno de pruebas

Con el objetivo de desarrollar un entorno de validación para el controlador del protocolo UART implementado, se seleccionaron dos dispositivos de prueba representativos: un módulo GPS NEO-6M y la Raspberry Pi 5. Esta selección no solo permite verificar la correcta

transmisión y recepción de datos en distintos escenarios, sino que también refleja situaciones prácticas de interconexión que son habituales en sistemas embebidos y satelitales.

Como la estructura de trama de UART es sencilla y no se admiten múltiples dispositivos en un mismo *bus*, las pruebas se enfocaron principalmente en garantizar la correcta sincronización de los parámetros de comunicación (*baudrate*, bits de datos, paridad y bits de parada), así como en validar la robustez ante errores comunes como pérdida de datos o *framing errors*.

Para el primer caso de prueba, se utilizó un módulo GPS NEO-6M configurado para transmitir sentencias NMEA estándar a una tasa de 9600 baudios. El flujo continuo de datos generado por el GPS permitió verificar la recepción asincrónico de información, la integridad de las tramas, y la gestión correcta de la cola de recepción implementada en FreeRTOS. Además, la naturaleza constante del flujo NMEA ofreció un excelente escenario para evaluar el desempeño del sistema en condiciones de tráfico de datos sostenido.

En paralelo, se estableció una comunicación UART con una Raspberry Pi 5, configurada para actuar como estación de prueba. Desde esta, se enviaron secuencias de caracteres de control, bloques de datos de prueba y comandos especiales, permitiendo así evaluar el comportamiento del módulo UART en escenarios de transmisión y recepción bidireccional. Esta metodología no solo facilitó la detección de errores en tiempo real mediante la visualización de datos de error en la Raspberry Pi, sino que también sirvió como entorno flexible para realizar pruebas automatizadas y repetibles.

Dado que el protocolo UART no requiere un mecanismo explícito de control de *bus* ni direccionamiento de dispositivos, no fue necesario implementar una arquitectura de separación de líneas o expansores de comunicación como en el caso de I^2C. Sin embargo, para detectar problemas a nivel de señal y *timing*, se empleó un analizador lógico conectado a las líneas TX y RX. Esta herramienta resultó fundamental durante el proceso de depuración, permitiendo observar directamente la estructura de las tramas, la alineación de los bits y los tiempos de respuesta del sistema.

Otro aspecto a resaltar también es que UART fue el primer protocolo implementado y probado en este proyecto, por lo que fue utilizado como herramienta de *debug* durante el desarrollo de los demás protocolos de comunicación, e inclusive se utiliza en algunas pruebas automatizadas de estos para realizar validación de resultados.

Pruebas utilizando Raspberry

Con el objetivo de validar el correcto funcionamiento del controlador UART desarrollado, se diseñó un conjunto de pruebas que involucran la transmisión, recepción, manejo de condiciones de sobrecarga y estrés de datos. Estas pruebas se ejecutaron comunicando la STM32F103C8T6 con una Raspberry Pi mediante enlaces UART, simulando escenarios reales de operación.

- **Transmisión básica de mensaje de prueba**
  - `test_tx(usart_id)`
  - Configuración: Se inicializa el periférico UART correspondiente.
  - Solicitud: se transmite una cadena formateada indicando el número de UART.
  - Objetivo: Verificar la correcta transmisión de datos por UART.
- **Recepción y reenvío de datos**
  - `test_rx(usart_id)`
  - Configuración: Se inicializa el periférico UART correspondiente.

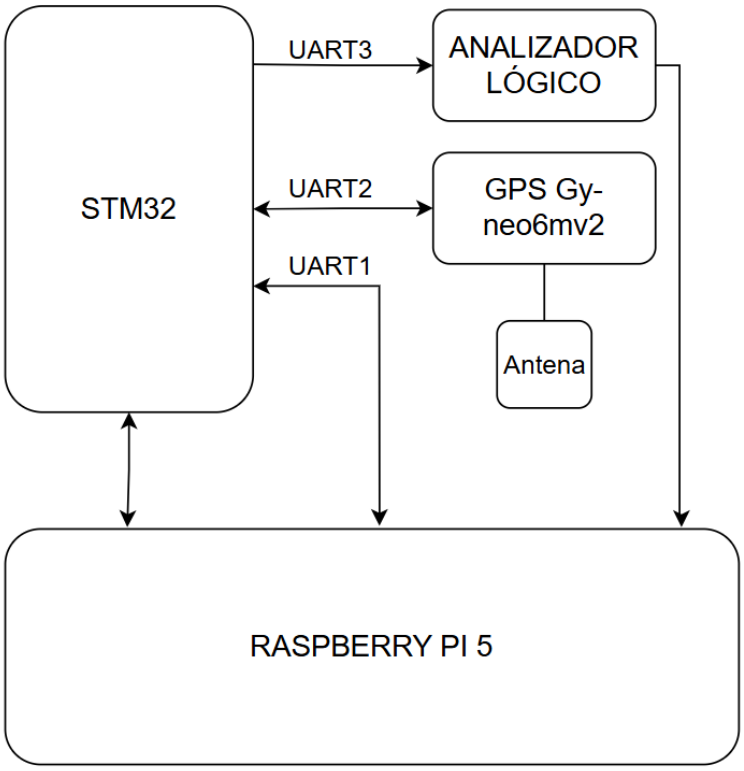

FIGURA 4.13. Banco de pruebas del protocolo UART

- Solicitud: se espera la llegada de datos y se reenvían por el mismo canal.
- Objetivo: Validar que la UART es capaz de recibir y reenviar datos correctamente.

- **Prueba de recepción ante desborde de buffer**
  - `test_overflow(usart_id)`
  - Configuración: Se inicializa el periférico UART correspondiente.
  - Solicitud: se simula la llegada de un bloque de datos mayor al tamaño de *buffer*.
  - Objetivo: Comprobar el comportamiento del sistema ante un posible desborde.
- **Transmisión de datos binarios**
  - `test_tx_binary(usart_id)`
  - Configuración: Se inicializa el periférico UART correspondiente.
  - Solicitud: se transmite una secuencia binaria ascendente.
  - Objetivo: Evaluar la capacidad de enviar datos no ASCII a través de UART.
- **Prueba de estrés con múltiples lecturas**
  - `test_stress()`
  - **Configuración:** Inicializa el periférico UART correspondiente.
  - **Funcionamiento:** Recibe múltiples valores por UART, realiza su suma y envía el resultado.
  - **Objetivo:** Verificar la estabilidad del sistema ante una carga sostenida y lectura continua.

4.1.9. GPS

El GPS (**Global Positioning System**) constituye un componente fundamental dentro de la arquitectura de un satélite, ya que proporciona información precisa de posición, velocidad y tiempo. Estos datos son esenciales para la determinación orbital, la sincronización de eventos y la navegación autónoma, capacidades críticas en misiones espaciales modernas, particularmente en satélites pequeños como los CubeSats [28].

La determinación de la órbita no sólo permite conocer la ubicación en tiempo real, sino que también habilita funciones como el control de actitud, la planificación de maniobras de evasión ante riesgos de colisión, y la optimización de enlaces de comunicaciones. En situaciones específicas, como la necesidad de ajuste de órbita para mejorar la cobertura o evitar desechos espaciales, contar con información de navegación fiable resulta vital para la seguridad de la misión [29].

El uso de módulos GPS en satélites debe considerar desafíos únicos asociados al entorno espacial, especialmente para satélites en órbita terrestre baja (LEO, *Low Earth Orbit*). A diferencia de los receptores convencionales diseñados para su uso terrestre, los receptores espaciales deben operar bajo condiciones atípicas:

- Altas velocidades relativas, superiores a los 7,km/s, que introducen efectos de Doppler significativos sobre la frecuencia de las señales recibidas.
- Captación de señales GPS desde altitudes donde la geometría de los satélites visibles es limitada y la intensidad de señal puede ser baja.
- Resistencia a la radiación espacial y robustez ante perturbaciones electromagnéticas.
- Requerimientos estrictos de bajo consumo energético y volumen reducido, adecuados a las restricciones de masa y potencia de los CubeSats.

La mayoría de los módulos GPS utilizados en satélites CubeSat han sido variantes adaptadas de módulos comerciales (COTS, *Commercial Off-The-Shelf*), optimizados para el entorno espacial [29][28].

La comunicación entre el módulo GPS y la computadora de abordo generalmente se realiza a través de interfaces serie, siendo UART una de las opciones más utilizadas debido a los factores mencionados en secciones anteriores. Este protocolo permite una transmisión robusta y eficiente de los mensajes NMEA (National Marine Electronics Association), el formato estándar utilizado para reportar datos GPS en misiones espaciales.

Finalmente, cabe destacar que la correcta elección del módulo GPS y su integración mediante interfaces como UART no sólo impactan en el desempeño de la misión, sino que también definen la capacidad del satélite para operar de manera autónoma, reduciendo la dependencia de redes terrestres de seguimiento y control, y aumentando significativamente la flexibilidad de operación de futuras misiones espaciales.

GPS NEO-6MV2

Para la fase de desarrollo y pruebas del CubeSat, se ha seleccionado el módulo GY-NEO6MV2, basado en el receptor u-blox NEO-6M[30]. Esta elección se debe a su bajo costo y facilidad de integración, lo que permite validar la interfaz de comunicación UART y los protocolos de manejo de datos GPS dentro del sistema de la computadora de abordo. Sin embargo, este módulo no es apto para condiciones espaciales, por lo que no será utilizado en la versión final del satélite una vez en órbita.

Protocolo de Comunicación y Tramas NMEA

El NEO-6M transmite datos en el estándar NMEA 0183, que es el formato más común en receptores GPS. Cada mensaje NMEA es una línea de texto ASCII estructurada en sentencias que contienen información de posicionamiento, tiempo y velocidad, entre otros.

De acuerdo con el estándar NMEA, el formato de salida de la latitud y longitud se muestra en grados, minutos y fracciones decimales de minutos.

Ejemplo de Trama GPRMC Se obtuvo la siguiente trama NMEA durante pruebas en marzo de 2025:

```
GPRMC,152430.00,A,3407.1234,N,11815.6789,W,000.5,054.7,250325,,,A*6C
```

Esta trama contiene los siguientes campos:

CUADRO 4.1. Desglose de campos GPRMC

Campo	Descripción
GPRMC	Identificador de mensaje
152430.00	Hora UTC (15:24:30)
A	Estado válido (A)
3407.1234,N	Latitud: 34°07,1234′ Norte
11815.6789,W	Longitud: 118°15,6789′ Oeste
000.5	Velocidad: 0,5 nudos
054.7	Rumbo: 54,7°
250325	Fecha: 25/03/2025
A	Modo autónomo
*6C	Checksum de verificación

Además de GPRMC, el receptor GY-NEO6MV2 emite otras tramas útiles para la misión. Algunos ejemplos son:

- `GPGGA`: Datos fundamentales de posición y altitud
- `GPGSA`: Precisión y satélites activos
- `GPGSV`: Satélites visibles y calidad de señal
- `GPVTG`: Velocidad y rumbo sobre terreno

Estas tramas permiten validar la funcionalidad del GPS en tierra y verificar su correcta integración con la computadora de abordo.

Por defecto, el receptor no va a mostrar datos inválidos, sino que en esos casos va a devolver campos vacíos:

```
GPGLL,,,,,124924.00,V,N*42  (Ejemplo de reporte de posición inválida
                             (pero tiempo válido))

GPGLL,,,,,,V,N*64          (Si el tiempo no se conoce, ejemplo en
                            un cold-start)
```

donde `V` indica datos no válidos y los campos vacíos representan información no disponible.

En el anexo G.5 se detalla información sobre las posibles configuraciones de este dispositivo.

4.2. I²C

El *bus* **I²C** (del inglés *Inter-Integrated Circuit*) es un estándar de comunicación serial sincrónica desarrollado por *Philips* en la década de 1980. Fue diseñado para facilitar la interconexión de múltiples dispositivos integrados mediante un número mínimo de líneas físicas. Gracias a su simplicidad, bajo consumo y capacidad para manejar varios dispositivos en un mismo canal, I²C se ha convertido en una de las opciones más utilizadas en sistemas embebidos, microcontroladores y sensores, estableciéndose como un estándar global [31].

Una de sus principales ventajas frente a otros protocolos como SPI es que requiere solo dos líneas —una para datos (**SDA**) y otra para reloj (**SCL**)— para comunicar un dispositivo maestro con múltiples esclavos. Esto es posible porque cada dispositivo conectado al *bus* posee una dirección única, utilizada por el maestro para iniciar una transacción con el esclavo deseado. Esta simplicidad de interconexión reduce la complejidad del hardware, minimiza el uso de pines del microcontrolador y permite diseños de PCB más compactos.

No obstante, I²C también presenta ciertas limitaciones: al compartir un único canal, la velocidad de comunicación es menor que en protocolos como SPI, y puede verse afectada por la capacitancia del bus, especialmente en sistemas con muchos nodos o líneas largas. Además, emplea el mecanismo de *clock stretching*, mediante el cual un esclavo puede mantener SCL en bajo para ganar tiempo de procesamiento. Si este mecanismo no está bien gestionado, puede causar bloqueos del bus.

En contextos como el de un CubeSat —donde los trayectos de señal son cortos y el entorno electromagnético está razonablemente controlado— estas limitaciones pueden mitigarse con un diseño eléctrico adecuado, selección apropiada de dispositivos y prácticas robustas de enrutamiento de PCB.

I²C utiliza dos líneas bidireccionales: SDA (*Serial Data*) y SCL (*Serial Clock*), ambas con resistencias de *pull-up* a una fuente positiva. Los dispositivos deben implementar salidas *open-drain* o *open-collector*, lo que permite que cualquier nodo pueda forzar la línea a bajo, pero no a alto. Esto genera un esquema de "AND cableado" donde un solo nivel bajo domina el valor lógico.

Si las resistencias de *pull-up* están mal dimensionadas o alimentadas desde una fuente inadecuada, pueden producirse flancos de subida lentos, errores de comunicación o bloqueo del bus. Además, si los dispositivos están alimentados desde distintas fuentes, puede ocurrir una situación de *back-powering* (alimentación involuntaria de un dispositivo a través de los pines SDA/SCL), lo cual puede dañar componentes o interferir con la señal. Para evitarlo, en este trabajo se utilizó un switch (PCF8574) para desconectar sensores del *bus* cuando no están en uso.

Transmisión de datos

Los datos se transmiten en bloques de 8 bits (un byte), sincronizados con pulsos de reloj en SCL. Durante la transmisión, SDA debe mantenerse estable mientras SCL está en alto; cualquier transición en SDA debe ocurrir sólo cuando SCL está en bajo. Luego de cada byte, el receptor debe emitir un bit de reconocimiento (ACK) en el noveno pulso de reloj. Si SDA se mantiene en alto durante ese pulso, se interpreta como un NACK (*not acknowledge*).

Toda comunicación inicia con una condición de *start*, generada al llevar SDA a bajo mientras SCL está en alto. A continuación, el maestro envía la dirección del esclavo (7 bits, *big-endian*) y un bit que indica si se trata de una lectura (1) o escritura (0). Cada byte transmitido o recibido va seguido de un ACK o NACK.

Para finalizar la comunicación, el maestro genera una condición de *stop*, cambiando SDA de bajo a alto mientras SCL está en alto. En algunos casos es posible emitir una condición de

repeated start en lugar de una de parada, lo que permite encadenar operaciones sin liberar el bus. Esto es útil, por ejemplo, al acceder a memorias EEPROM.

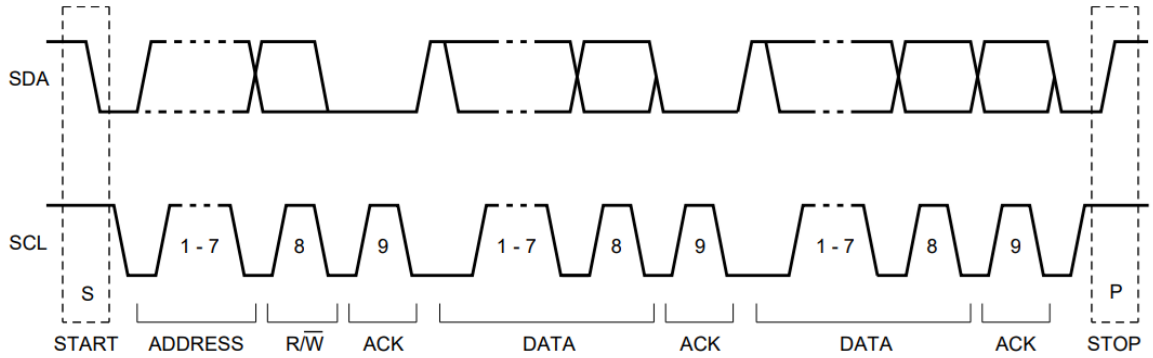

FIGURA 4.14. Trama típica de una comunicación I²C [31]

El protocolo define distintos modos de operación según la velocidad:

- Modo estándar: hasta 100 kbits/s
- Modo rápido: hasta 400 kbits/s
- Modo rápido plus: hasta 1 Mbit/s
- Modo de alta velocidad: hasta 3,4 Mbit/s

El número de dispositivos está limitado principalmente por la capacitancia total del *bus* y la necesidad de que cada dirección sea única. Aunque en este trabajo se utiliza un único maestro (la OBC del CubeSat), el protocolo también permite configuraciones multi-maestro mediante arbitraje.

4.2.1. I²C en los CubeSat

La elección de I²C refleja una realidad ampliamente documentada en el ámbito aeroespacial: este protocolo, a pesar de sus limitaciones, se ha convertido en un estándar de facto en muchos diseños de CubeSats. Entre el 71 % y el 81 % de los CubeSats lanzados o en etapa de diseño emplean I²C como uno de sus *buses* de comunicación principales [32]. Esta elección se debe en gran medida a las ventajas que ofrece el protocolo en contextos con restricciones severas de tamaño, peso, consumo y presupuesto, como los CubeSats: I²C requiere solo dos líneas de comunicación (SCL y SDA), permite la conexión de múltiples dispositivos mediante direccionamiento, y se encuentra ampliamente soportado por una gran variedad de componentes COTS (*Commercial Off-The-Shelf*), los cuales son frecuentemente utilizados para reducir costos y acelerar el desarrollo de misiones.

Sin embargo, el uso de I²C en el espacio no está exento de desafíos. A diferencia de otros protocolos, I²C no implementa mecanismos intrínsecos de verificación de integridad de datos ni de recuperación frente a fallos. En entornos espaciales, donde la radiación puede inducir *bitflips* en registros internos o causar malfuncionamientos transitorios en dispositivos, esta falta de tolerancia a fallos convierte al *bus* en un posible punto único de falla. Se han documentado casos concretos de fallos asociados al *bus* I²C en misiones como CP4 (California Polytechnic State University), Delfi-C3 y Delfi-n3Xt (TU Delft), donde se han reportado *bus lockups*, errores persistentes de transmisión y fallas catastróficas por corrupción de buffers o esclavos defectuosos [33]. Estadísticas en misiones reales indican que el 43 % de los

problemas en el *bus* I²C involucraron bloqueos del *bus* de más de un minuto, un 21 % bloqueos menores, además de errores de transmisión, fallos catastróficos confirmados (3 %) y fallos catastróficos probables (7 %) [32] [33].

En este trabajo se proponen algunas alternativas para mitigar dichas limitaciones. Una de ellas es la adopción una arquitectura híbrida que incorpora un expansor de E/S I²C (PCF8574) junto con interruptores analógicos (74HC4066) para permitir la desconexión física de esclavos en caso de fallo. Esta solución no solo permite evitar bloqueos del *bus* provocados por un esclavo defectuoso —por ejemplo, uno que mantenga SDA en bajo o que entre en un estado indeterminado— sino que también posibilita la reconexión controlada por software, sin necesidad de un reinicio completo del sistema. Esta arquitectura refleja prácticas documentadas en la literatura, como el uso de multiplexores o buffers bidireccionales con líneas de habilitación para aislar secciones del bus, pero con una implementación más flexible y de bajo costo, acorde al objetivo pedagógico del trabajo.

Adicionalmente, si bien el protocolo I²C no contempla mecanismos de verificación de integridad de datos en su especificación original, es posible incorporar estrategias a nivel de aplicación para mitigar este punto débil. En este proyecto se ha implementado verificación por CRC-8 en sensores que lo permiten, como el HTU21D, lo cual asegura que los datos recibidos han sido transmitidos correctamente a través del bus.

En dispositivos que no ofrecen mecanismos de verificación nativos, se adoptaron estrategias adicionales, como la lectura posterior al ciclo de escritura en memorias EEPROM para confirmar que los datos grabados coincidan con los esperados. Estos ejemplos refuerzan la importancia de abordar la integridad de los datos desde el diseño del software, adaptando las estrategias de verificación a las capacidades específicas de cada dispositivo, y delegando esta responsabilidad al controlador del dispositivo correspondiente.

4.2.2. Implementación

El protocolo I²C posee una especificación bien definida, pero su implementación puede variar según el microcontrolador que se esté utilizando. En el caso de los STM32F1, el periférico I²C requiere un control detallado de su estado interno para funcionar correctamente, en especial cuando actúa como maestro receptor. En este caso, a diferencia de otros periféricos más autónomos, el *software* debe anticipar con precisión cómo finalizará la transacción antes de que ocurran eventos como la recepción del último byte.

Estas restricciones hacen que la secuencia de operaciones varíe según la cantidad de bytes que se desee recibir. Por ejemplo, al leer un único byte, el maestro debe emitir una condición de parada (`STOP`) incluso antes de que el dato esté disponible en el registro de datos. Esta necesidad, que a primera vista parece contradecir la lógica del protocolo, se debe al comportamiento interno del periférico: el dato es primero capturado en un *shift register* y luego transferido al registro de datos sólo cuando se cumplen ciertas condiciones. Por lo tanto, señales como `NACK` y `STOP` deben ser anticipadas con precisión para que el flujo de recepción funcione correctamente. Este mismo principio se aplica a las lecturas de dos o más bytes, aunque con variantes específicas en el orden y sincronización de eventos.

Dado que el comportamiento del periférico depende fuertemente del contenido de registros como `CR1`, `SR1` y `SR2`, se diseñó un controlador personalizado en C, con lógica basada en *polling* y controlada mediante FreeRTOS. Esta implementación permite mantener el flujo de comunicación sin bloquear el sistema, monitoreando *flags* clave (`RxNE`, `BTF`, `ADDR`) con *timeouts* y llamadas a `taskYIELD()` para permitir la ejecución de otras tareas del sistema.

Para detalles precisos sobre las secuencias especiales requeridas por la STM32F1, se remite al Anexo H.

Opciones analizadas

Para la implementación de los controladores de I^2C para la STM32 se abordaron distintas alternativas. En un primer enfoque, se propuso la utilización de dos estructuras:

- `i2c_t`: representa el *bus* I^2C, con identificador de instancia (I2C1, I2C2), colas de transmisión y recepción (TXQ, RXQ) y un *mutex* para acceso concurrente.
- `msg_t`: Contiene la dirección del esclavo, el buffer de datos, la longitud del mensaje y un *flag* `request` para distinguir entre envío y solicitud de datos

En esta implementación, se utilizó una tarea que corría en segundo plano, encargada de la transmisión y recepción de datos. De esta forma, cuando una función quería enviar un mensaje o comando, solo cargaba los datos en la cola de transmisión y solicitaba la respuesta, en caso de corresponder, en la cola de recepción. El procesamiento se realizaba como indica la figura 4.15

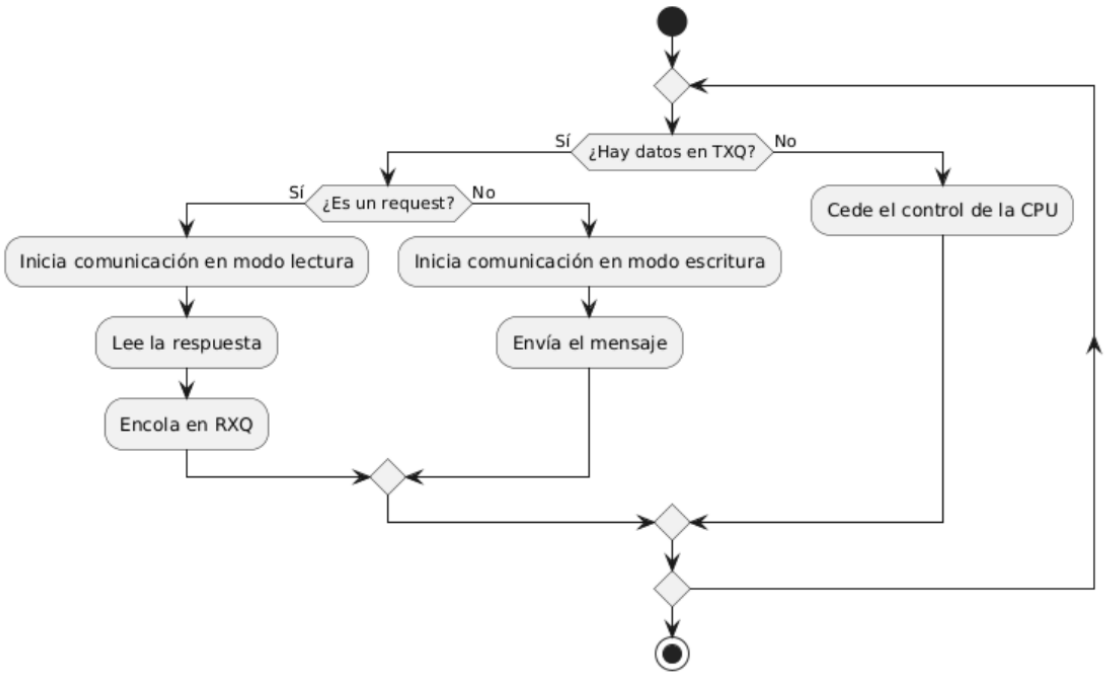

FIGURA 4.15. Enfoque inicial

Si bien centralizar la transmisión de datos simplifica el diseño, dificulta el manejo de múltiples solicitudes simultáneas de distintos dispositivos, en especial la asociación de cada solicitud con su respuesta.

Estos motivos llevaron a profundizar un segundo enfoque que eliminó la tarea y cola de transmisión, delegando esta actividad en las funciones particulares que lo requieran, pero manteniendo la cola de recepción. Se crearon funciones separadas para escritura, solicitud de lectura y combinaciones de ambas (escritura-lectura). También se incluyó un identificador numérico único (ID) para cada estructura `msg_t`, lo cual permitía rastrear un mismo mensaje entre una función y otra. Si bien esto trajo una mayor modularidad y redujo la complejidad de asociar respuestas con solicitudes, mantuvo la complejidad al utilizar múltiples dispositivos (debe recordarse que la lectura del dato puede hacerse en otro momento distinto a la petición).

Algunas soluciones propuestas fueron reemplazar la cola por un arreglo (estático en principio) o por una lista enlazada. En el arreglo estático, cada mensaje recibido, con la dirección del esclavo en su estructura, se almacenaba en una posición determinada, pudiendo

ser esta posición una función de la dirección del esclavo que permita encontrar fácilmente un ID. Sin embargo, reorganizar los elementos para evitar huecos podía resultar lento. La lista, por su parte, permitía una eliminación más eficiente, aunque la búsqueda seguía siendo un punto a optimizar.

Finalmente, se decidió simplificar la estructura del bus, que actualmente consiste únicamente en una instancia con un identificador de 32 bits y su *mutex* asociado. Se eliminó la estructura `msg_t` y la cola de recepción. En su lugar, se proporciona al usuario el acceso directo a funciones de lectura, escritura y control del bus, incluyendo la toma y liberación del *mutex*. Este esquema permite que cada tarea gestione sus propios tiempos de espera y sincronización, adaptándose a las necesidades específicas de cada dispositivo, y liberando el CPU mediante llamadas como `taskYIELD()` durante operaciones de espera.

Aunque este nuevo enfoque podría implementarse directamente utilizando dos *mutex* independientes —uno por cada puerto I²C—, se optó por mantener la estructura `i2c_t` como contenedor. Esta decisión responde a la intención de agrupar en una única entidad los recursos y parámetros asociados a cada *bus*, facilitando su extensión futura y abriendo la posibilidad de instrumentar registros de diagnóstico, estadísticas o mecanismos de recuperación ante fallos, sin alterar la interfaz pública del controlador.

Controlador

Como se explicó en la sección 2.1.6, se descartó el uso de la biblioteca HAL provista por STMicroelectronics debido a sus limitaciones en sistemas con FreeRTOS y su menor portabilidad. En su lugar, se empleó la biblioteca de código abierto `libopencm3`, cuya interfaz liviana y enfoque directo sobre los registros permite una integración más controlada dentro de un sistema operativo de tiempo real. Algunas de sus funciones se utilizaron para configurar el periférico y manejar señales básicas del protocolo. Sin embargo, se evitó el uso de funciones de más alto nivel como `i2c_transfer7`, ya que encapsulan toda la transacción sin permitir un control preciso sobre errores, tiempos de espera o secuencias particulares exigidas por la arquitectura STM32.

En cambio, se desarrolló una implementación propia con lógica explícita para cada paso del protocolo, y soporte de temporización mediante primitivas de FreeRTOS (como `taskYIELD()` y `vTaskDelay()`). Este diseño permitió manejar correctamente los distintos modos de lectura (1, 2 y 3 o más bytes), y garantizar una ejecución no bloqueante compatible con otras tareas concurrentes.

La lista completa de funciones utilizadas de `libopencm3` puede consultarse en su documentación oficial [34].

El controlador desarrollado expone una interfaz modular orientada a tareas concurrentes en FreeRTOS. Se diseñó un conjunto de funciones para enviar y recibir datos, tomar y liberar el mutex del bus, inicializar el periférico, y controlar la conexión de sensores a través del expansor PCF8574.

Todas las funciones aplican verificación explícita de errores y cuentan con tiempos máximos de espera para evitar bloqueos permanentes en el sistema. La estructura de errores utilizada es del tipo `i2c_status_t`, con valores que permiten distinguir fallos por *timeout*, problemas de sincronización (*mutex*) o parámetros inválidos.

Adicionalmente, el controlador incluye funciones auxiliares internas (declaradas como `static`) para encapsular la lógica de control de *flags* (`SB`, `ADDR`, `RxNE`, por ejemplo), envío de direcciones, y validación de parámetros. Este diseño permitió cumplir con las secuencias de operación requeridas por el periférico I²C de STM32 en sus distintos modos, sin bloquear el planificador del sistema operativo.

Con el fin de incrementar la robustez del sistema y permitir una validación más flexible, se definió una arquitectura que separa los dispositivos conectados al *bus* en dos grupos: en el

puerto 1 se ubica un conjunto compuesto por un expansor de E/S PCF8574 y un conjunto de interruptores analógicos SN74HC4066; mientras que todos los demás periféricos se conectan al puerto 2. Esta configuración brinda control individual sobre las líneas SDA y SCL de cada dispositivo conectado al segundo bus, permitiendo desconectarlos selectivamente en caso de que alguno de ellos esté bloqueando la comunicación al mantener en bajo una de las líneas.

Adicionalmente, se implementó una rutina específica para recuperar el *bus* I²C (`liberar_bus_i2c()`), que actúa como mecanismo de seguridad en caso de detección de un *bus* bloqueado. Este procedimiento reconfigura temporalmente las líneas SDA y SCL como salidas digitales y genera manualmente nueve pulsos de reloj en la línea SCL, una técnica recomendada en situaciones donde un esclavo I²C haya quedado en un estado inconsistente esperando un décimo pulso de reloj para completar la transmisión de datos. Luego, se intenta forzar una condición de parada (STOP), liberando el bus.

La descripción detallada de las funciones públicas y privadas del controlador puede consultarse en el repositorio del trabajo.

4.2.3. Entorno de pruebas

Con el objetivo de desarrollar un entorno de validación para el controlador del protocolo I²C implementado, se seleccionaron diversos dispositivos esclavos compatibles, tales como el sensor de humedad y temperatura HTU21D, la unidad de referencia inercial MPU6050 y la memoria EEPROM AT24C256. Estos periféricos no solo permiten verificar la correcta implementación del protocolo en diferentes escenarios de lectura y escritura, sino que además constituyen una selección representativa de funcionalidades frecuentemente utilizadas en aplicaciones espaciales. En cada caso se abordaron funcionalidades específicas de interés, tales como la lectura de datos sensados, la escritura controlada en memoria o la identificación del dispositivo. El desarrollo se centró en aquellas operaciones que resultan más relevantes para validar el comportamiento del *bus* I²C y construir una base de ejemplos que puedan ser reutilizados o ampliados en futuras etapas del proyecto.

La incorporación de estos módulos busca replicar situaciones reales que podrían surgir en una misión, por lo que se implementaron únicamente algunas de sus funcionalidades más relevantes. Este enfoque selectivo permite construir una base de ejemplos concreta que sirva como punto de partida para el desarrollo futuro de controladores específicos en el contexto de una computadora a bordo (OBC), facilitando la integración de nuevos sensores o memorias según las necesidades del proyecto. El código correspondiente a estas funciones se encuentra documentado en el repositorio del proyecto y puede utilizarse como referencia para la integración de nuevos dispositivos en futuras etapas.

Dado que el soporte como esclavo I²C no está incluido de forma nativa en el *kernel* de Linux, se optó por emplear un analizador lógico como *sniffer* de la comunicación y un microcontrolador ATmega328P configurado como esclavo. Esta decisión evitó la necesidad de implementar soluciones no estándar en la Raspberry Pi 5, lo que podría introducir ambigüedad en el diagnóstico de errores, especialmente aquellos atribuibles al controlador implementado en la STM32.

El analizador lógico no solo permitió realizar pruebas detalladas a nivel de señal, sino que también resultó fundamental durante el proceso de *debugging*, facilitando la detección de errores sutiles en la lógica del controlador.

El banco de pruebas resultante puede observarse en la figura 4.16.

Los detalles de la implementación de los dispositivos esclavos utilizados en este trabajo pueden consultarse en el Anexo H.6.

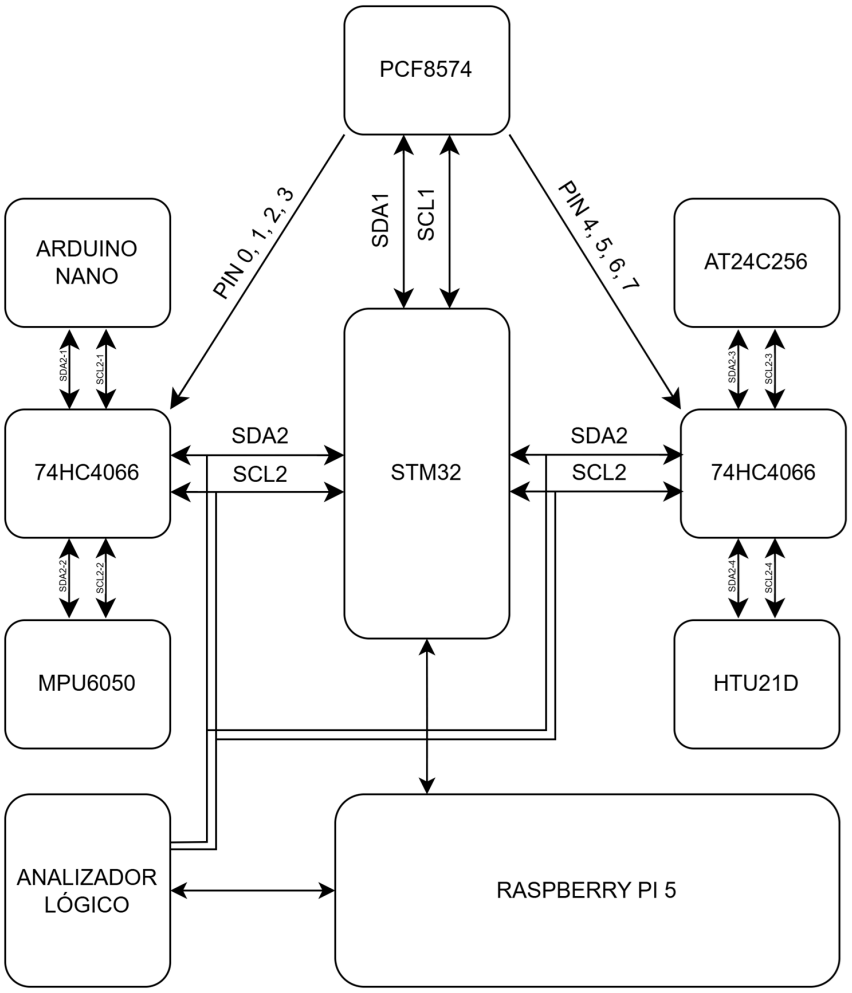

FIGURA 4.16. Banco de pruebas del protocolo I^2C

PCF8574 y SN74HC4066

El PCF8574 es un expansor de puertos de entrada/salida (I/O) bidireccional, controlado a través del *bus* I^2C y diseñado para operar con tensiones entre $2,5$ V y 6 V [35]. Los SN74HC4066, por su parte, son interruptores analógicos cuádruple basado en tecnología CMOS, capaz de manejar señales tanto analógicas como digitales de hasta 6 V pico en ambas direcciones [36]. Cada sección del switch tiene su propia entrada de control (C): cuando C está en alto, la llave se cierra y cuando C está en bajo, la llave se abre.

Pruebas

- **Secuencia de mensajes con todos los dispositivos conectados**
  - `reading_mix(false)`
  - Configuración: Se habilita la conexión con todos los esclavos
  - Solicitud: temperatura y humedad (HTU21D), aceleración y giro (MPU6050), envío de los datos serie
  - Objetivo: Verificar que todos los esclavos están correctamente conectados.
- **Secuencia de mensajes con un dispositivo desconectado**

- `reading_mix(true)`
- Configuración: Se habilita la conexión con todos los esclavos, a excepción del HTU21D
- Solicitud: temperatura y humedad (HTU21D), aceleración y giro (MPU6050), envío de los datos por serie
- Objetivo: Verificar que el *bus* no se corrompa al solicitar datos a un esclavo inexistente y que pueda realizar con éxito las demás peticiones.

Con el objetivo de robustecer la operación del *bus* I²C, se implementó la tarea `vTaskCheckSensors`. Esta tarea se ejecuta en segundo plano y, cada cierto intervalo de tiempo fijo, verifica si alguno de los sensores ha sido desconectado debido a un error anterior.

En caso de detectar un sensor en dicha condición, se procede a conectarlo temporalmente y enviarle un mensaje para verificar si responde con un ACK a su dirección I²C. Si el sensor responde correctamente, se considera que está nuevamente operativo y se lo reconecta permanentemente al puerto correspondiente. Este proceso puede observarse en la figura 4.17.

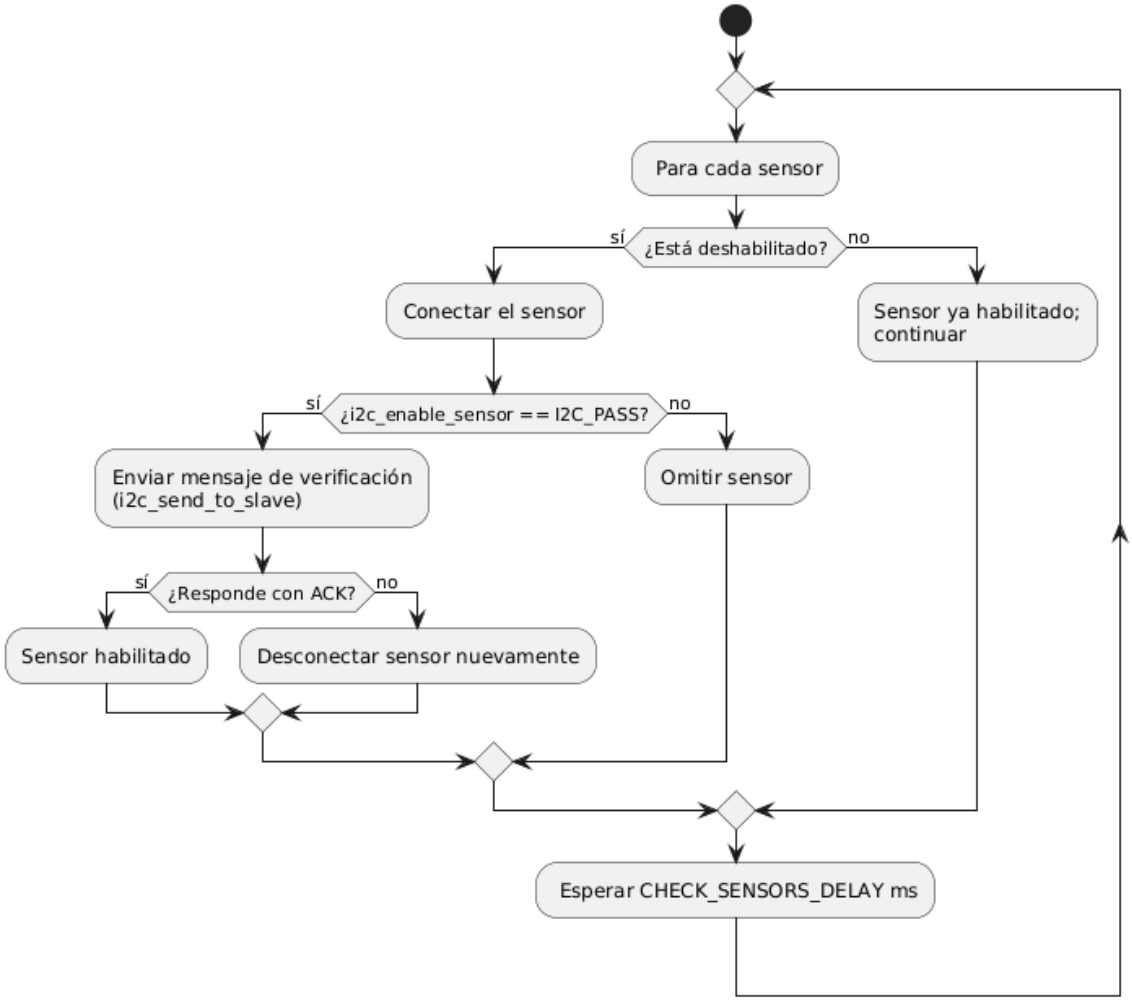

FIGURA 4.17. Tarea monitora de sensores

- **Desconexión de un dispositivo con la tarea monitora corriendo**
  - `test_measurement(true)`

  - Configuración: Se habilita la conexión con todos los esclavos, a excepción del HTU21D
  - Solicitud: temperatura y humedad
  - Objetivo: Verificar que el dispositivo está desconectado.

- **Solicitud de temperatura unos instantes después**

  - `test_measurement(true)`
  - Configuración: No se realiza ninguna configuración previo a la solicitud de datos
  - Solicitud: temperatura y humedad
  - Objetivo: Verificar que el dispositivo fue conectado automáticamente y se recibieron correctamente los datos

ATmega328P

El ATmega328P es un microcontrolador de 8 bits perteneciente a la familia megaAVR [37], ampliamente utilizado en placas como Arduino UNO y Nano.

En este trabajo se emplea como dispositivo esclavo dentro del entorno de pruebas para el protocolo I^2C, particularmente en la validación de funciones de transmisión y recepción de datos. Para ello, se conectaron los pines PC4 (SDA) y PC5 (SCL) del ATmega328P al puerto I2C2 de la STM32, a través del interruptor analógico correspondiente.

El microcontrolador ejecuta un programa (incluido en la carpeta `Arduino/i2c` del repositorio) diseñado específicamente para responder a distintos comandos enviados por el maestro I^2C. La dirección del dispositivo está definida por la constante `SLAVE_ADDRESS`, cuyo valor por defecto es `0x04`.

El protocolo implementado en el ATmega permite seleccionar diferentes modos de operación enviando un primer byte a modo de encabezado, el cual determina el comportamiento del dispositivo ante futuras interacciones. Las acciones definidas son las siguientes:

- **Encabezado `0x00`**: el esclavo recibe los datos enviados por el maestro y los retransmite por el puerto serie. Este modo se utiliza para probar la función de transmisión implementada en la STM32.
- **Encabezados `0x01`, `0x02`, ..., `0x05`**: ante una posterior petición de lectura, el esclavo responde enviando 1, 2, 3, 4 o 5 bytes, respectivamente, comenzando desde el valor `0x00` y aumentando de forma incremental. Este mecanismo permite verificar la correcta recepción de datos y validar el tratamiento de los distintos casos particulares (1, 2 y 3 o más bytes) en el maestro.

Pruebas

El objetivo de las pruebas con el ATmega328P es poder analizar exclusivamente las funciones de envío y recepción de datos, es decir, el controlador I^2C, de forma aislada. A su vez, es importante verificar que, en caso de un error, se pueda comunicar de forma adecuada. Los códigos se encuentran en las carpetas `test/test_i2c` y `test_ci_cd/test_i2c`.

Para este apartado, se propusieron 10 situaciones diferentes. Las dos primeras pruebas consisten en transmisiones bajo condiciones normales. Luego, se evalúa el comportamiento ante el envío a una dirección inexistente. A continuación, se transmite un bloque de datos de longitud variable con una desconexión intencional durante la comunicación. Posteriormente, se introduce una segunda transmisión mientras una anterior sigue en curso. Finalmente, se ejecutan lecturas de 1 a 5 bytes para verificar el manejo correcto de cada caso particular en el STM32.

- **Envío de un byte**
  - `test_send_msg_1()`
  - Configuración: Se habilita la conexión con el ATmega 328p
  - Mensaje: hexadecimal correspondiente a “a”
  - Objetivo: Verificar el correcto envío del byte.
- **Envío de datos en dos comunicaciones secuenciales**
  - `test_send_msg_2()`
  - Configuración: Se habilita la conexión con el ATmega 328p
  - Mensaje: hexadecimales correspondientes a “Prueba extensa numero 1” y “Prueba extensa numero 2” para la primera y segunda comunicación, respectivamente
  - Objetivo: Verificar el correcto envío de ambos mensajes.
- **Envío de datos a una dirección inexistente**
  - `send_wrong_direction()`
  - Configuración: Se habilita la conexión con el ATmega 328p y se escribe otra dirección en la función que envía los datos
  - Mensaje: hexadecimales correspondientes a "Nueva direccion"
  - Objetivo: Verificar que el *bus* no se corrompe al intentar enviar un mensaje a una dirección inexistente
- **Envíos de datos con una interrupción en la comunicación**
  - `send_msg_with_disconnect()`
  - Configuración: Se habilita inicialmente la conexión con el ATmega 328p
  - Mensaje: cadena de longitud y contenido aleatorio que, en un momento al azar, sufre una desconexión del esclavo
  - Objetivo: Verificar que el *bus* no se corrompe al no poder enviar un dato en medio de una comunicación y que finaliza con una señal de Stop
- **Intento de una nueva comunicación cuando aún no finalizo la anterior**
  - `send_msg_with_collision()`
  - Configuración: Se habilita la conexión con el ATmega 328p
  - Mensaje: hexadecimales correspondientes a "TEST COLLISION"
  - Objetivo: Verificar que el mensaje se envía correctamente pese a que, en el medio, se intenta iniciar una nueva comunicación. Recibir correctamente los errores correspondientes a los *mutex* del periférico
- **Recepción de datos**
  - `receive_xx_bytes()`
  - Configuración: Se habilita la conexión con el ATmega 328p
  - Mensaje: [0x00], [0x00, 0x01], [0x00, 0x01, 0x02], [0x00, 0x01, 0x02, 0x03], [0x00, 0x01, 0x02, 0x03, 0x04] (según el valor de xx)
  - Objetivo: Verificar la correcta recepción de los datos para los casos particulares de la implementación de STM32 (1, 2 y 3 o más bytes)

HTU21D

El HTU21D es un sensor digital de humedad y temperatura con salida en formato I^2C y bajo consumo de energía [38]. La prueba para el sensor de humedad y temperatura se encuentran en las carpetas `test/test_htu21d` y `test_ci_cd/test_htu21d`.

Pruebas

- **Solicitud de humedad y temperatura**
  - `test_measurement()`
  - Configuración: Se habilita la conexión con el HTU21D
  - Solicitud: temperatura y humedad
  - Objetivo: Verificar el correcto envío del comando, la recepción de datos dentro de los márgenes esperados y su validación por CRC-8.

MPU6050

La MPU6050 es una unidad de medición inercial (IMU, por sus siglas en inglés de *Inertial Measurment Units*) que contiene un acelerómetro y un giroscopio, ambos de 3 ejes. Las pruebas que verifican el uso de la unidad de referencia inercial y ejemplifican su uso se encuentran en las carpetas `test/test_mpu6050` y `test_ci_cd/test_mpu6050`

Pruebas

- **Comunicación con el dispositivo**
  - `test_who_am_i()`
  - Configuración: Se habilita la conexión con el MPU6050
  - Solicitud: valor del registro `WHO_AM_I` (dirección del dispositivo)
  - Objetivo: Verificar la correcta comunicación con el dispositivo y la recepción del valor esperado.
- **Solicitud de la aceleración con ajuste de desvío**
  - `test_accel()`
  - Configuración: Se habilita la conexión con el MPU6050
  - Solicitud: aceleración
  - Objetivo: Verificar la recepción del valor con y sin ajuste de la aceleración, junto a la mejora esperada
- **Solicitud del giro con ajuste de desvío**
  - `test_gyro()`
  - Configuración: Se habilita la conexión con el MPU6050
  - Solicitud: aceleración
  - Objetivo: Verificar la recepción del valor con y sin ajuste del giro, junto a la mejora esperada

AT24C256

El AT24C256 es una memoria EEPROM de 256 kilobits (32 kilobytes), organizada en 512 páginas de 64 bytes cada una. Utiliza una interfaz I²C para la comunicación, lo que permite realizar operaciones de lectura y escritura tanto de forma aleatoria como secuencial. Esta memoria es ampliamente usada en sistemas embebidos por su bajo consumo, persistencia de datos y simplicidad de integración. Las pruebas para este módulo se encuentran en las carpetas `test/test_at24c256` y `test_ci_cd/test_at24c256`.

Pruebas

- **Borrado completo de la memoria**
  - `test_erase_all()`
  - Configuración: se habilita la conexión con el AT24C256
  - Solicitud: escritura del valor 0x00 en todos los bytes de la EEPROM
  - Objetivo: ejecutar la acción y leer una posición al azar para comprobar que su valor sea igual a 0x00
- **Escritura de un byte en una posición determinada**
  - `test_write_byte()`
  - Configuración: se habilita la conexión con el AT24C256
  - Solicitud: escritura del valor 0x55 en la página 2, byte 16
  - Objetivo: verificar que la acción se complete correctamente
- **Lectura de un byte en una posición determinada**
  - `test_read_byte()`
  - Configuración: se habilita la conexión con el AT24C256
  - Solicitud: lectura del valor almacenado en la página 2, byte 16
  - Objetivo: verificar que el valor sea igual al escrito en la prueba anterior (0x55).
- **Escritura secuencial de una página**
  - `test_escritura_secuencial()`
  - Configuración: se habilita la conexión con el AT24C256
  - Solicitud: escritura secuencial de un conjunto de números entre 0 y 63 en la página 3.
  - Objetivo: verificar que la operación finalice correctamente
- **Lectura secuencial de una página**
  - `test_lectura_secuencial()`
  - Configuración: se habilita la conexión con el AT24C256
  - Solicitud: lectura secuencial de la página 3.
  - Objetivo: verificar que los valores leidos sean iguales a los escritos en la prueba anterior.

4.3. SPI

El protocolo **SPI** (del inglés *Serial Peripheral Interface*) es un estándar de comunicación serial sincrónica que fue desarrollado por Motorola en la década de 1980. Desde entonces, se ha convertido en un estándar de facto para la interconexión de microcontroladores con dispositivos periféricos, tales como memorias Flash, sensores y conversores digital-analógico (DAC), entre otros. A diferencia de otros protocolos como I^2C o CAN, SPI permite comunicación *full-dúplex* y alcanza velocidades de transferencia considerablemente elevadas, lo que lo convierte en una opción ideal para aplicaciones que requieren baja latencia y alta eficiencia temporal.

Una de las principales ventajas de SPI frente a otros *buses* como I^2C es que sus tasas de transferencia no están restringidas por modos de operación predefinidos y pueden ser varios órdenes de magnitud superiores. No obstante, esta flexibilidad impone una exigencia adicional: el dispositivo esclavo debe ser capaz de manejar el ritmo de transmisión impuesto por el maestro, ya que el protocolo no incorpora mecanismos de *clock stretching*. Por lo tanto, una falta de sincronización o de capacidad de procesamiento en el esclavo puede derivar en pérdida de datos. En este sentido, la correcta elección del divisor de frecuencia (*prescaler*) resulta crítica para no superar las especificaciones máximas de cada periférico y así garantizar una transmisión confiable y estable.

Si bien el protocolo SPI no incluye mecanismos integrados de *handshaking* ni confirmación de recepción (*acknowledgment*), el microcontrolador empleado incorpora soporte para el cálculo de **CRC** (del inglés *Cyclic Redundancy Check*) por *hardware*, una técnica utilizada para la detección de errores en la transmisión o almacenamiento de datos. Esta funcionalidad puede activarse según sea necesario, y permite configurar distintos polinomios, adaptándose así al utilizado por cada dispositivo esclavo. De este modo, se habilita una detección eficiente de errores en la transmisión sin requerir validaciones adicionales por *software*.

Una limitación importante del protocolo SPI en su configuración ***full-duplex*** es la necesidad de utilizar cuatro señales: dos líneas de datos (`MOSI` y `MISO`), una línea de reloj (`SCK`) y una línea de control para la selección del esclavo ($\overline{\texttt{CS}}$). Además, en arquitecturas multi-esclavo, cada dispositivo requiere su propia línea de selección, lo que incrementa significativamente el uso de pines del microcontrolador. En el caso particular del microcontrolador utilizado en este proyecto, se dispone únicamente de una línea $\overline{\texttt{CS}}$ gestionada por el *hardware*. Por este motivo, y como se detallará en secciones posteriores, fue necesario implementar líneas de selección adicionales mediante pines de propósito general (GPIO) configurados por *software*.

El presente capítulo tiene como objetivo describir en detalle el funcionamiento del *bus* SPI, abordando su topología, modos de operación y su comportamiento como un conjunto de registros de desplazamiento. En este proyecto, se desarrolló un controlador SPI en modo **maestro**, ya que la OBC actúa como unidad central del sistema y no como periférico esclavo. Asimismo, se implementó el modo ***full-duplex***, por tratarse de la modalidad más completa y de mayor rendimiento disponible, y por ser el requerido por las memorias Flash utilizadas durante el desarrollo.

Finalmente, cabe destacar que durante el diseño del controlador se priorizó la robustez del sistema, implementando mecanismos de protección de recursos compartidos mediante herramientas proporcionadas por **FreeRTOS**, así como funciones de envío y recepción con verificación por **CRC**, con el fin de garantizar la integridad de los datos transmitidos. En las secciones siguientes se analizan las ventajas, limitaciones y consideraciones prácticas del protocolo SPI en entornos embebidos.

4.3.1. SPI en los CubeSats

En los sistemas CubeSat, la selección del *bus* de datos tiene implicancias directas en la robustez del sistema, la eficiencia de las transferencias y la posibilidad de integrar periféricos críticos. Entre los protocolos más utilizados se encuentran I^2C, RS-232 y SPI, siendo este último una de las opciones preferidas cuando se requieren altas velocidades de transmisión y una comunicación más estable entre módulos.

Según el estudio de Bouwmeester et al. [33], el *bus* SPI fue implementado en el **50 % de los CubeSats lanzados** y en el **43 % de los CubeSats en desarrollo** . Estos porcentajes reflejan una adopción significativa del protocolo en misiones CubeSat.

En términos de confiabilidad, el *bus* SPI evidenció un desempeño notable. De los CubeSats que lo implementaron, el **94 % no reportó ningún tipo de fallo en órbita**, mientras que únicamente el **6 % indicó la ocurrencia de un bloqueo del *bus* (*bus lockup*) superior a un minuto**, en contraste, por ejemplo, con lo mencionado para I^2C en la Sección 4.2.1.

Estas diferencias se deben en parte a la arquitectura de SPI: se trata de un *bus* sincrónico y full-duplex, con líneas de selección dedicadas para cada dispositivo esclavo, lo cual minimiza las interferencias y elimina los conflictos de arbitraje presentes en I^2C. Aunque SPI no incorpora tolerancia a fallos a nivel físico de forma nativa, su simplicidad eléctrica y topológica contribuye a una operación más confiable.

Una de las principales motivaciones para adoptar SPI en este proyecto fue la necesidad de contar con un sistema de almacenamiento fiable y de alta velocidad. En el contexto de una misión espacial, es fundamental registrar datos provenientes del *payload* científico (como mediciones ambientales o datos experimentales) y, eventualmente, almacenar imágenes, coordenadas, registros de estado del sistema o incluso actualizaciones de firmware. Con este fin, se optó por la utilización de dos tipos de memorias conectadas mediante SPI: **memorias NOR Flash** (como la Winbond W25Q32) y **tarjetas SD**, para las cuales se implementaron controladores que permiten realizar operaciones básicas de lectura, escritura y borrado, y que fueron validados, como se detalla en la Sección 4.3.7.

4.3.2. Topología del *bus* SPI

SPI sigue una arquitectura de tipo **maestro-esclavo**. Esto significa que un dispositivo actúa como maestro (generalmente un microcontrolador) y controla la comunicación con uno o más dispositivos esclavos. El maestro es responsable de generar la señal de reloj, seleccionar los dispositivos y coordinar el flujo de datos.

Los cuatro líneas principales del *bus* SPI son:

- **MOSI** (*Master Out Slave In*): línea por la que el maestro envía datos al esclavo.
- **MISO** (*Master In Slave Out*): línea por la que el esclavo devuelve datos al maestro.
- **SCK** (*Serial Clock*): señal de reloj generada por el maestro que sincroniza la transferencia.
- **$\overline{\text{CS}}$ o NSS** (*Chip Select o Negative Slave Select*): línea activa en bajo que selecciona el esclavo.

En la Figura 4.18, se ilustra un sistema SPI con un único esclavo conectado al maestro. Sin embargo, es posible conectar múltiples esclavos, cada uno con su línea individual de ($\overline{\text{CS}}$), lo que permite al maestro seleccionar cuál desea activar en un momento dado.

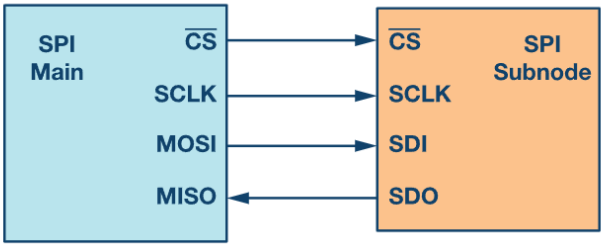

FIGURA 4.18. Topología de SPI con un maestro y un esclavo [39].

4.3.3. El rol de la linea *Chip Select*

La línea *Chip Select* ($\overline{\texttt{CS}}$), también conocida como *Negative Slave Select* (NSS), cumple la función de delimitar una transacción SPI. Se trata de una señal activa en bajo: cuando ($\overline{\texttt{CS}}$) se encuentra en nivel lógico bajo, el esclavo interpreta que la comunicación está activa y debe responder al maestro.

- El maestro debe poner ($\overline{\texttt{CS}}$) en bajo antes de iniciar la transmisión del primer bit.
- La línea debe volver a nivel alto una vez finalizada la transferencia.
- En algunos dispositivos, como memorias Flash, esta secuencia es estrictamente requerida para la ejecución de comandos críticos (por ejemplo, escritura o borrado de sectores).

En topologías donde existe más de un dispositivo esclavo conectado al mismo *bus* SPI, como se muestra en la Figura 4.19, la línea ($\overline{\texttt{CS}}$) cumple un rol fundamental en la selección del dispositivo al cual se le desea transmitir datos. Para cada esclavo se utiliza una línea ($\overline{\texttt{CS}}$) dedicada, y sólo aquel cuya línea se encuentra en nivel bajo responderá a la transacción, mientras que los demás permanecerán inactivos.

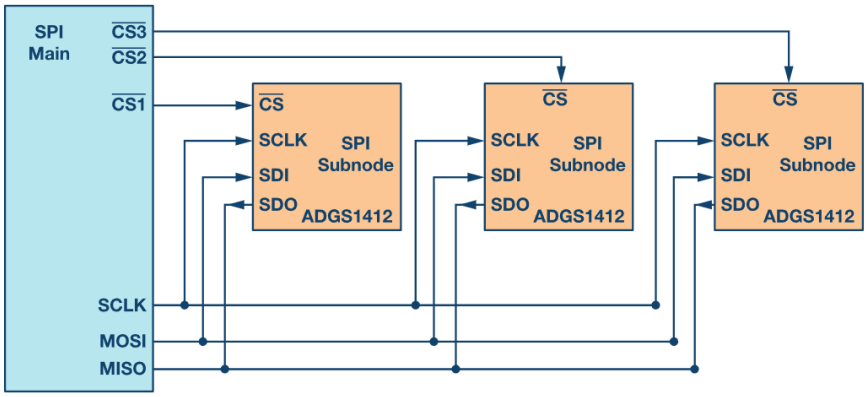

FIGURA 4.19. Topología SPI con múltiples esclavos [39].

Incluso en configuraciones con un único esclavo, el uso correcto de ($\overline{\texttt{CS}}$) es fundamental para garantizar la integridad de la comunicación. El esclavo necesita una referencia clara de inicio y fin de transacción; de lo contrario, podría perder la sincronización a causa de ruido en la línea de reloj o datos. Esto puede derivar en consecuencias graves, como la ejecución incorrecta de comandos sensibles, entre ellos el borrado accidental de memoria.

Como se mencionó en la introducción, el microcontrolador utilizado posee solo una línea de selección de chip nativa del periférico. En su configuración por defecto dicho pin es

manejado por el *hardware* activándose cuando se inicia la comunicación y volviendo al estado alto una vez que se deshabilita el periférico. Es por ello que se desarrolló la opción de manejo de las líneas CS mediante pines GPIO. Para ello el controlador brinda un archivo de cabecera *spi_config.h* donde se deben declarar los pines necesarios y donde se definen las etiquetas para cada esclavo los cuales serán utilizados por una función interna del controlador para configurarlos en su correspondiente modo.

4.3.4. Transmisión de datos

Uno de los aspectos más característicos del *bus* SPI es su método de comunicación. A medida que el maestro envía bits de datos por la línea MOSI, el esclavo devuelve simultáneamente bits al maestro por la línea MISO. Esta operación se realiza de forma paralela y sincronizada, como se ilustra en la Figura 4.20, donde ambos extremos se comportan como un conjunto de registros de desplazamiento conectados entre sí.

La señal de reloj (SCK), que siempre es generada por el maestro, determina el momento en que los bits son desplazados y muestreados hacia los registros receptores de ambas partes. Una vez que se completa la transferencia de un dato (por ejemplo, 8 bits), tanto el maestro como el esclavo pueden acceder y leer el contenido recibido en sus respectivos registros.

Durante cada pulso de reloj generado por el maestro:

- El maestro envía un bit al esclavo a través de MOSI.
- Simultáneamente, el esclavo envía un bit al maestro por MISO.
- Ambos datos son desplazados dentro de sus respectivos registros receptores.

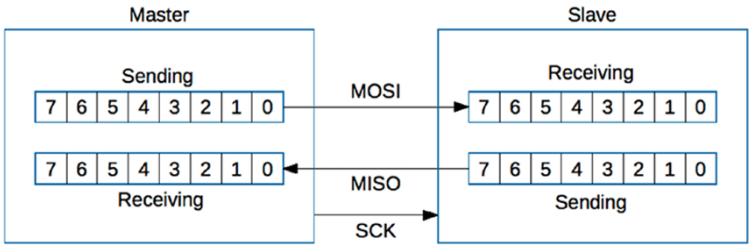

FIGURA 4.20. Maestro y esclavo SPI como conjunto de registros de desplazamiento [12]

El periférico SPI del microcontrolador STM32 está diseñado para manejar dicha lógica mediante un registro de datos (`SPI_DR`), un *buffer* de transmisión y un *buffer* de recepción, además de un registro de desplazamiento interno (*shift register*).

La descripción de esta secuencia se basa en la documentación oficial del fabricante, específicamente en el manual de referencia RM0008 para la familia STM32F1 [23].

Transmisión

La secuencia básica de transmisión comienza cuando se escribe un dato en el *buffer* de transmisión. Este dato es cargado en paralelo desde el *bus* interno hacia el registro de desplazamiento (*shift register*) durante el primer ciclo de reloj SPI. A partir de ese momento, el dato se transmite bit a bit por la línea MOSI, comenzando por el bit más significativo (MSB) o el menos significativo (LSB), según lo determine el bit `LSBFIRST` del registro `SPI_CR1`.

La generación de la señal de reloj SCLK está directamente controlada por el maestro y se activa únicamente durante una operación de transmisión. Cada flanco de reloj corresponde a un bit desplazado desde el maestro hacia el esclavo. Por lo tanto, la escritura en el registro de datos no solo inicia la transmisión de bits por la línea MOSI, sino que también activa la generación de pulsos de reloj en la línea SCLK.

Una vez que el dato es transferido desde el *buffer* de transmisión al registro de desplazamiento, se activa el flag `TXE` (*Transmit Buffer Empty*), indicando que el *buffer* está libre para cargar un nuevo dato. Si el bit `TXEIE` del registro `SPI_CR2` está habilitado, este evento genera una interrupción que puede ser utilizada para continuar la transmisión sin bloquear el procesador.

Recepción

En el receptor, cuando finaliza la transmisión de un byte, el contenido del registro de desplazamiento se transfiere al *buffer* de recepción, activando el *flag* `RXNE` (*Receive Buffer Not Empty*). Este *flag* indica que hay un dato disponible para ser leído. Si el bit `RXNEIE` en `SPI_CR2` está habilitado, se genera una interrupción. De lo contrario, el *software* debe verificar periódicamente el estado del *flag* mediante *polling*. El *flag* `RXNE` se borra automáticamente al leer el contenido del registro `SPI_DR`.

A continuación, se muestra un diagrama de la comunicación SPI 4.21 en **modo 0** (CPOL = 0, CPHA = 0), donde los datos son muestreados en el primer flanco de reloj, específicamente en el flanco de subida (cuando la señal `CLK` cambia de nivel bajo a nivel alto). La comunicación se inicia cuando el maestro coloca la línea ($\overline{\texttt{CS}}$) en nivel bajo, lo que habilita al esclavo seleccionado para participar en la transacción. A partir de ese momento, el maestro genera los pulsos de reloj y transmite los bits del dato por la línea `MOSI`, mientras el esclavo responde simultáneamente por la línea `MISO`, completando así una transferencia en modo *full-duplex*.

El inicio y el final de la transmisión están indicados por las líneas discontinuas verdes. Los flancos de muestreo, en los que los datos son capturados por los receptores, están marcados con líneas naranjas, mientras que los flancos de desplazamiento, donde se actualizan los bits en las salidas de los registros, están señalados en azul. Este esquema facilita la comprensión del funcionamiento interno del protocolo como un conjunto de registros de desplazamiento sincronizados.

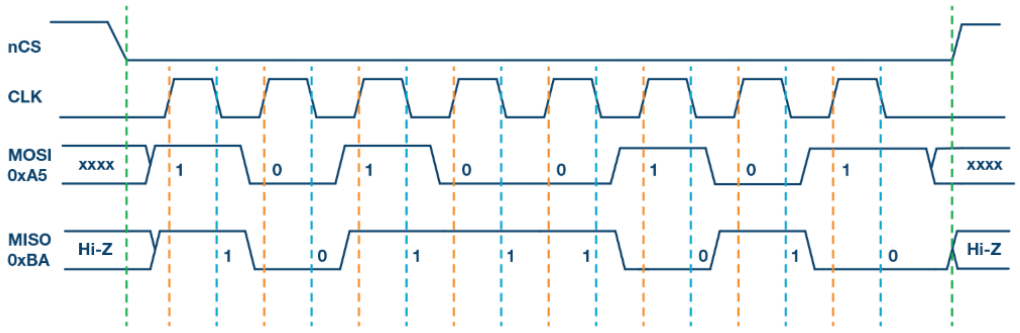

FIGURA 4.21. Ejemplo de transacción SPI *full-duplex* en modo 0 [39].

Es importante remarcar que, dado su funcionamiento, la transmisión de datos siempre implica una operación simultánea de recepción. En consecuencia, cuando se desea únicamente recibir datos, como en las funciones `spi_receive()`, es necesario colocar un valor *dummy* (por ejemplo, `0xFF`) en el registro de transmisión. Esto permite generar la señal de reloj necesaria para que el esclavo envíe los bits requeridos a través de la línea MISO.

Como consecuencia, durante la primera transacción la información que transmite el esclavo de manera simultánea en la línea `MISO` carece de validez. Para obtener una respuesta útil, se realiza una segunda transacción en la que el maestro transmite un dato ficticio (comúnmente una palabra *dummy*, con el propósito de generar los ciclos de reloj necesarios para que el esclavo pueda colocar su respuesta en la línea `MISO` y esta sea leída por el maestro.

Este mecanismo de doble transacción es fundamental en operaciones de lectura, especialmente al interactuar con dispositivos que requieren primero recibir una dirección de memoria o un código de operación antes de poder devolver los datos solicitados.

4.3.5. Configuración y modos del periférico SPI

El protocolo contempla cuatro modos de temporización que determinan cómo se interpretan los flancos del reloj y en qué instante deben muestrearse o transmitirse los datos. Estos modos están definidos por dos bits de configuración:

- `CPOL` (*Clock Polarity*): determina el nivel lógico inactivo de la línea de reloj (0 = bajo, 1 = alto).
- `CPHA` (*Clock Phase*): indica si los datos se capturan en el primer o en el segundo flanco del reloj.

Modo	CPOL	CPHA	Descripción
0	0	0	Reloj en reposo bajo, datos capturados en flanco de subida
1	0	1	Reloj en reposo bajo, datos capturados en flanco de bajada
2	1	0	Reloj en reposo alto, datos capturados en flanco de bajada
3	1	1	Reloj en reposo alto, datos capturados en flanco de subida

CUADRO 4.2. Modos SPI definidos por CPOL y CPHA.

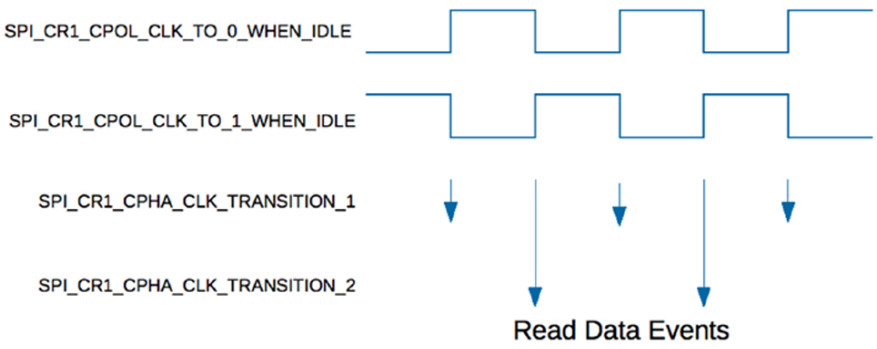

FIGURA 4.22. Representación gráfica de los modos SPI según CPOL y CPHA [12]

Seleccionar el modo correcto es esencial, ya que el maestro y el esclavo deben coincidir exactamente en la configuración de `CPOL` y `CPHA`; de lo contrario, los datos podrían interpretarse incorrectamente. Para resolver esta necesidad, los controladores desarrollados incluyen la función:

```
spi_status_t spi_set_mode(uint32_t spi_id, uint8_t mode);
```

que permiten cambiar el modo de temporización que fue configurado por defecto, con el fin de que el maestro pueda adaptarse a las necesidades del esclavo.

Otros de los parámetros configurables son el formato y orden de los bits transmitidos. El periférico SPI permite configurar el formato del *data frame*, incluyendo tanto el tamaño de la trama como el orden en que se transmiten los bits. Por defecto, el periférico fue configurado para operar con tramas de 8 bits y para transmitir los datos en orden **MSB** (más significativo primero), que es el formato estándar adoptado por la mayoría de los dispositivos SPI, incluyendo las memorias utilizadas en este desarrollo. Sin embargo, estas configuraciones pueden modificarse en tiempo de ejecución mediante la función:

```
spi_status_t spi_set_dff(uint32_t spi_id, uint8_t data_size, uint8_t
    bit_order);
```

Adicionalmente, el controlador desarrollado permite configurar el **prescaler** del reloj SPI, lo cual es fundamental para adaptar la velocidad de transferencia a las especificaciones eléctricas y temporales de cada dispositivo esclavo. Esta configuración determina la frecuencia de la señal `SCK` mediante un divisor preestablecido, cuyos valores típicos son: 2, 4, 8, 16, 32, 64, 128 y 256. La función encargada de modificar esta configuración es:

```
spi_status_t spi_set_prescaler(uint32_t spi_id, uint8_t prescaler);
```

Modificar dinámicamente la configuración del periférico SPI durante la ejecución puede ser necesario en sistemas donde múltiples dispositivos esclavos, con distintos requerimientos de comunicación, comparten el mismo *bus*. Por esta razón, se implementaron funciones específicas que permiten adaptar la configuración del periférico en tiempo de ejecución de manera controlada y segura.

Cabe destacar que tanto estas funciones de configuración como el resto de las operaciones del controlador fueron diseñadas para ser *thread-safe*[3]. Esta característica no solo es esencial para garantizar el funcionamiento correcto del *bus* SPI, sino que también constituye un principio de diseño transversal aplicado a todos los módulos del sistema. Por ello, se introduce este concepto, ya que será recurrente a lo largo de esta sección al analizar la interacción entre tareas y periféricos en el contexto de un sistema operativo en tiempo real.

En dicho contexto, la concurrencia entre tareas puede provocar conflictos si no se aplican mecanismos adecuados de protección del *bus* y su configuración. A continuación, se describen tres situaciones típicas que pueden generar errores en ausencia de sincronización:

1. **Conflicto entre tareas:** Una tarea inicia una transferencia SPI y, antes de finalizarla, se produce un cambio de contexto. Si otra tarea intenta acceder al mismo esclavo, puede generarse una colisión que provoque corrupción de datos.

2. **Selección incorrecta de esclavo:** Una tarea comienza una transacción con un determinado esclavo, pero es interrumpida por otra que intenta comunicarse con un esclavo diferente. Si la línea `NSS` del primer dispositivo no fue liberada correctamente, la segunda transmisión podría dirigirse al esclavo equivocado.

3. **Configuración inconsistente del periférico:** Algunos dispositivos esclavos requieren configuraciones específicas, como un modo SPI particular (por ejemplo, modo 0), un tamaño de trama distinto o un orden de bits determinado. Si una tarea modifica la configuración mientras otra realiza una transacción activa, se compromete la integridad de la comunicación.

Por ejemplo, la función `spi_set_prescaler()` incorpora validaciones internas para asegurar que el *bus* SPI no se encuentre ocupado durante el cambio de configuración, ya

[3]Una función *thread-safe* puede ser ejecutada por múltiples tareas o hilos de manera concurrente sin producir condiciones de carrera ni corrupción de datos, ya que maneja correctamente el acceso a recursos compartidos.

que alterar el valor del prescaler durante una transmisión activa podría provocar comportamientos indeseados sobre la línea de reloj SCK, afectando la integridad de la comunicación. Además, la función bloquea temporalmente el acceso al periférico por parte de otras tareas hasta que la nueva configuración haya sido aplicada por completo.

4.3.6. Implementación

Las funciones de transmisión y recepción de datos fueron desarrolladas con el objetivo de extender y robustecer las capacidades provistas por la biblioteca *libopencm3*, incorporando mecanismos para el manejo seguro de errores y asegurando su correcto funcionamiento en entornos con múltiples tareas, mediante el control del acceso a recursos compartidos. Para ello, se integraron mecanismos de exclusión mutua que indican cuándo el *bus* se encuentra en uso, junto con verificaciones de estado y mecanismos de salida por *timeout*, con el fin de evitar bloqueos indefinidos ante fallos en la comunicación o mal funcionamiento de los dispositivos conectados.

Con el fin de organizar estas capacidades y facilitar la escalabilidad del sistema, se definió la siguiente estructura de configuración para cada interfaz SPI:

```
typedef struct {
    uint32_t spi_id;              // Identificador del periferico SPI
    SemaphoreHandle_t mutex;      // Mutex para acceso exclusivo desde
   multiples tareas
    const slave_t *slaves;        // Arreglo de esclavos disponibles para
   esa interfaz
    uint8_t data_size;            // Largo de palabra (8 o 16 bits)
} spi_t;
```

CÓDIGO 4.6. Estructura de configuración para periféricos SPI

Esta estructura permite encapsular toda la configuración relevante de cada instancia del periférico SPI, otorgando independencia entre ellas. Al incluir un `mutex` propio por interfaz, se garantiza la protección del acceso concurrente desde múltiples tareas en FreeRTOS. Asimismo, el campo `data_size` permite configurar dinámicamente el tamaño de palabra de transmisión y recepción, permitiendo adaptar las funciones de transmisión y recepción según sea necesario.

Gracias a este diseño modular, la implementación resulta fácilmente escalable, permitiendo futuras extensiones como el agregado de configuraciones personalizadas por interfaz, sin alterar la lógica central del controlador.

Otra mejora relevante es la gestión robusta de los recursos compartidos, en particular el acceso al *bus* SPI. Tal como se describió en la sección 4.3.3, el control de la línea $\overline{\text{CS}}$ es esencial para evitar corrupciones durante la transmisión de datos. Para ello, el controlador implementa las funciones `spi_select_slave()` y `spi_deselect_slave()`, que no solo informan al esclavo el inicio y fin de la comunicación, sino que además, al momento de seleccionar el esclavo, toman el *mutex* correspondiente a la interfaz SPI involucrada. Esto genera un bloqueo que impide el uso concurrente del *bus* por parte de otras tareas o funciones hasta que se libere.

Tal como se mencionó en la sección 2.3.6, desde el punto de vista técnico, una tarea o función podría intentar acceder al recurso en cualquier momento, incluso sin haber obtenido previamente el *mutex*. Sin embargo, el diseño del controlador se basa en una convención estricta: para activar y desactivar las líneas $\overline{\text{CS}}$, es obligatorio utilizar las funciones proporcionadas (`spi_select_slave()` y `spi_deselect_slave()`). De lo contrario, no se garantiza la ausencia de condiciones de carrera durante la transmisión.

Para facilitar el acceso a los distintos esclavos conectados al *bus* SPI, se define una estructura que representa a cada uno de ellos. Esta estructura incluye un identificador único para

cada esclavo —el cual debe ser definido por el usuario (por ejemplo, `SLAVE_1` o un nombre descriptivo como `SD_CARD`)—, junto con el puerto GPIO y el pin correspondientes a la línea CS, utilizando las definiciones de la biblioteca `libopencm3`.

```
// Definiciones de slaves predefinidos para SPI1
/**

@brief Lista de slaves predefinidos para el bus SPI1.

Este arreglo contiene los slaves que estan conectados al bus SPI1.

Cada elemento de la lista tiene un identificador de slave, el puerto GPIO
    y el pin para controlar el CS.

La lista termina con el valor SLAVE_END.
*/
static const slave_t spi1_slaves[] = {
{SLAVE_1, GPIOA, GPIO4}, /**< SLAVE_1 conectado al puerto GPIOA, pin 4.
   */
{SLAVE_2, GPIOA, GPIO7}, /**< SLAVE_2 conectado al puerto GPIOA, pin 7.
   */
SLAVE_END /**< Final de la lista de slaves. */
};
```

De esta forma, cada vez que se desee seleccionar un esclavo, se utilizará la función proporcionada de la siguiente manera:

```
spi_select_slave(SPI1, SD_CARD);
```

Esta metodología no solo organiza y simplifica la definición de los pines asociados a cada esclavo, sino que también facilita la portabilidad del código. En caso de que el proyecto escale o sea necesario migrar a otro microcontrolador, el cambio de pines puede gestionarse de manera centralizada y clara, sin necesidad de modificar el código funcional distribuido en el resto del sistema.

Implementación por *polling*

Una consideración fundamental en este proyecto es que el microcontrolador opera como **maestro SPI**, lo que implica que tiene control total sobre el inicio, la duración y la frecuencia de las transferencias. En este contexto, implementar la transmisión y recepción de datos mediante *polling* no significa realizar un monitoreo continuo del periférico, sino ejecutar esperas muy acotadas y puntuales, únicamente cuando el sistema decide transmitir o recibir datos. Al ser el maestro quien inicia la comunicación, el *polling* no representa una carga constante para la CPU, sino una operación breve que ocurre justo antes y después de colocar un dato en el registro de transmisión. Esta situación puede observarse claramente en el anexo I.1, donde se ilustra el flujo completo de una transacción SPI.

Por este motivo, y considerando además que el uso de interrupciones implicaría cambios de contexto entre la rutina de servicio (ISR) y las tareas —introduciendo una sobrecarga adicional sobre el *scheduler* de FreeRTOS—, se optó por la estrategia de *polling* como solución más eficiente y controlada. Esta decisión resulta especialmente razonable en función de las capacidades del microcontrolador utilizado: el periférico SPI1 se encuentra conectado al *bus* APB2 (hasta 72MHz), mientras que SPI2 está ligado al APB1 (hasta 36MHz). Ambos permiten divisores mínimos de 2, habilitando velocidades de hasta 36MHz y 18MHz respectivamente. En este contexto, el tiempo de bloqueo asociado al *polling* es despreciable frente al tiempo total de ejecución del sistema.

Además de su eficiencia, esta estrategia aporta simplicidad al código, facilita la comprensión secuencial del flujo de datos y mejora considerablemente la depuración durante las fases de desarrollo y validación.

Sobre esta base, se implementaron mejoras con respecto a la biblioteca ofrecida por *libopencm3*, incluyendo el uso de *timeouts* para detectar condiciones de error y evitar bloqueos prolongados que puedan comprometer la estabilidad de las tareas en ejecución.

Como se explicó en la Sección 4.3.4, el funcionamiento del protocolo SPI implica que la transmisión y la recepción de datos ocurren simultáneamente. Este comportamiento es importante destacarlo, ya que si bien se dispone de funciones como `spi_transmit()`, `spi_receive()` y `spi_transmit_receive()`, todas ellas comparten una lógica interna común basada en este principio de operación.

La diferencia principal entre estas funciones radica en su finalidad específica y en cómo se utilizan los datos de entrada y salida. Por ejemplo:

- `spi_transmit()` realiza una escritura en el registro de transmisión y, para limpiar el *flag* RXNE, también realiza una lectura del registro de recepción, aunque el dato recibido se descarta.
- `spi_receive()` requiere enviar datos ficticios (*dummy*) en el registro de transmisión con el objetivo de generar los pulsos de reloj necesarios para recibir datos desde el esclavo.
- `spi_transmit_receive()` contempla ambas operaciones de forma simétrica, considerando tanto los datos enviados como los recibidos como relevantes.

Por este motivo, en el anexo I.1 se incluye la implementación detallada de la función `spi_transmit_receive()`, utilizada como base para las operaciones SPI en las que ambos flujos de datos tienen importancia funcional.

Verificación de integridad mediante CRC

El código de redundancia cíclica (CRC) es una técnica ampliamente utilizada para detectar errores en la transmisión o el almacenamiento de datos. Consiste en calcular un valor de verificación (*checksum*) a partir del contenido del mensaje, aplicando una operación matemática basada en álgebra polinómica. Este valor se transmite junto con los datos y permite al receptor verificar si la información fue alterada durante su envío, ya sea por ruido, fallas eléctricas o condiciones inesperadas del sistema.

En este proyecto, se incorporó soporte para verificación mediante CRC utilizando el periférico SPI del microcontrolador STM32. Esta funcionalidad está implementada por *hardware* y puede activarse en cada transmisión pasando el valor `true` en el parámetro `crc_enable` de las funciones desarrolladas.

Cuando está habilitada, el periférico calcula automáticamente el CRC del flujo de datos transmitido y lo adjunta al final de la transmisión. Simultáneamente, el receptor realiza un cálculo equivalente sobre los datos recibidos, lo que permite comparar el valor recibido con el calculado localmente. En caso de discrepancia, se activa el flag `CRCERR` en el registro `SPI_SR`, permitiendo así detectar errores de transmisión. Esta verificación se realiza dentro de las funciones de transmisión, y ante una falta de coincidencia se devuelve el código de error `SPI_CRC_ERROR`. Todas las funciones implementadas en este desarrollo retornan un código que refleja el estado de la operación, y el listado completo de posibles errores puede consultarse en el archivo de cabecera *spi_driver.h*.

La configuración del cálculo de CRC se realiza en función del tamaño de los datos transmitidos, utilizando un polinomio de 8 o 16 bits. Además, se provee la función `spi_set_crc`

`_polynomial()`, que permite establecer manualmente el polinomio a utilizar, en caso de que se requiera una configuración específica.

En la figura 5.8 se puede observar la captura de una transmisión SPI en modo 0 (CPOL = 0, CPHA = 0), en la cual el maestro envía la secuencia de bytes 0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF, 0xAC. Esta última palabra corresponde al valor del CRC calculado por *hardware*, utilizando un polinomio de 8 bits igual a `0x07`.

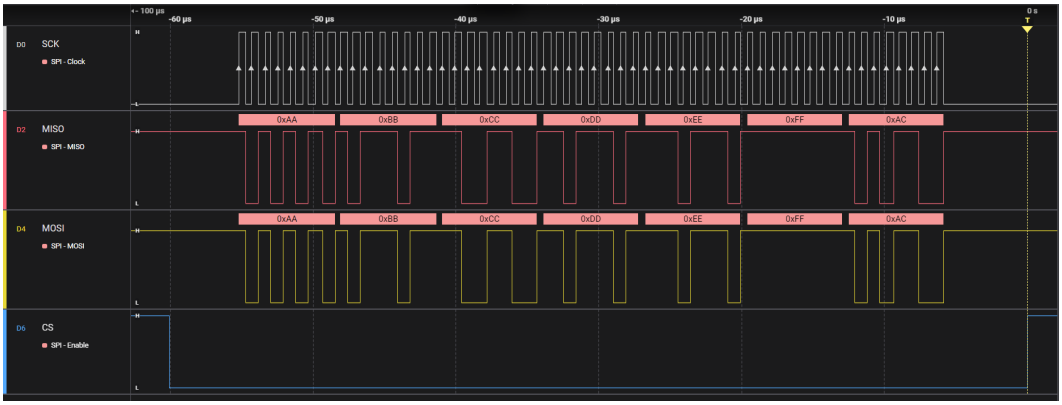

FIGURA 4.23. Captura de envio de datos trama con CRC

4.3.7. Entorno de pruebas

Durante el desarrollo y validación de los controladores SPI implementados, se diseñó un entorno de pruebas en múltiples etapas. Inicialmente, se utilizó un microcontrolador STM32 como maestro, conectado a un microcontrolador ATmega328p (proveniente de una placa Arduino) configurado como esclavo. Esta configuración permitió verificar la transmisión de datos y observar la respuesta del sistema mediante una interfaz UART.

A medida que avanzó el desarrollo, se complementó el banco de pruebas con un analizador lógico, el cual permitió observar en tiempo real las señales del *bus* SPI y decodificar los mensajes transmitidos. Sin embargo, esta topología presentaba una limitación significativa: el módulo SPI del ATmega328p está restringido a tramas de 8 bits, lo que impedía validar correctamente la lógica implementada para transmisiones de 16 bits.

Frente a esta restricción, se evaluaron otras alternativas, como el uso del puerto SPI de la Raspberry Pi 5 empleada en otras pruebas del trabajo. No obstante, al igual que otros sistemas basados en Linux, esta plataforma no ofrece soporte nativo para operar como esclavo SPI. Se consideró también una solución propuesta por la comunidad basada en *bit-banging* para simular un esclavo SPI mediante el control manual de los pines GPIO. Si bien esta técnica es viable para aplicaciones simples, no resulta adecuada para tareas críticas o de alta velocidad, debido a que el sistema operativo no garantiza tiempos de respuesta precisos. Por lo tanto, se descartó su uso, ya que validar un controlador utilizando una implementación no oficial e imprecisa podría llevar a conclusiones erróneas.

Como resultado, se decidió dividir la validación del controlador SPI en dos etapas claramente diferenciadas: una primera fase utilizando el periférico en modo *loopback*, y una segunda fase con pruebas realizadas sobre periféricos reales.

Validación mediante conexión *loopback* y analizador lógico

La primera técnica utilizada es una metodología ampliamente conocida: la conexión *loopback*[4] entre las líneas MISO y MOSI. Esta conexión permite que los datos enviados por el microcontrolador retornen por la misma interfaz, posibilitando la verificación del funcionamiento básico del controlador SPI sin necesidad de un dispositivo esclavo externo. Adicionalmente, dicha conexión se encuentra conectada a un analizador lógico, el cual permite observar en detalle todas las señales involucradas en la comunicación.

Una de las limitaciones de la configuración *loopback* es que, por sí sola, no permite validar el correcto funcionamiento de las líneas de reloj (SCK) y de selección de chip ($\overline{CS}$), ya que no se requiere una respuesta activa por parte de un dispositivo esclavo para completar la comunicación. En consecuencia, es posible que la transmisión se efectúe de manera aparente sin que dichas señales cumplan con las condiciones eléctricas y temporales requeridas.

Por esta razón, se decidió complementar el banco de pruebas con el uso de un analizador lógico. Este instrumento permite capturar y decodificar las señales involucradas en la comunicación SPI, y se integra al sistema de validación mediante scripts en la Raspberry Pi. Además de facilitar la inspección visual de las tramas mediante su interfaz gráfica, el analizador lógico garantiza que las líneas SCK y CS estén funcionando correctamente, dado que una decodificación válida sólo es posible si ambas señales presentan un comportamiento conforme al protocolo SPI.

En la Figura 4.24 se muestra una captura de pantalla correspondiente a una transmisión correctamente decodificada, mientras que en la Figura 4.25 se presenta un caso de prueba en el cual se forzó un error al mantener la línea CS inactiva. Como resultado, el analizador no logra identificar una trama válida, reflejando el carácter crítico del control de dicha señal para la operación del sistema.

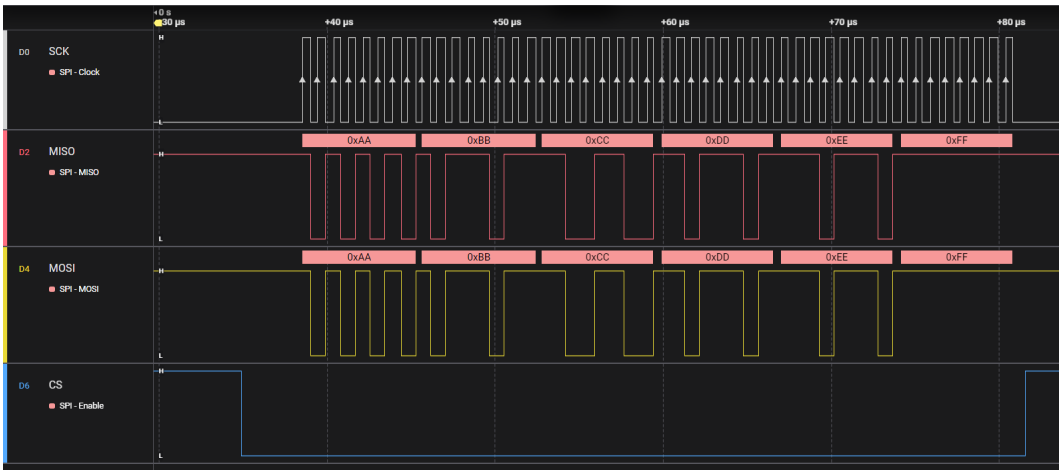

FIGURA 4.24. Captura de trama SPI en modo 8 bits obtenida con analizador lógico.

Este comportamiento es clave para la validación, ya que en las pruebas automatizadas realizadas en la Raspberry Pi los datos SPI no se analizan a través de la interfaz gráfica, sino a partir de un archivo de texto plano generado por el analizador lógico. Dicho archivo contiene la transcripción completa de la transmisión y es procesado mediante un *script* que evalúa el contenido decodificado.

En este contexto, es fundamental que las señales de reloj (SCK) y de selección de chip funcionen correctamente, ya que cualquier fallo en estas líneas impedirá que el analizador

[4]Consiste en conectar físicamente la salida de transmisión (MOSI) con la entrada de recepción (MISO) del mismo dispositivo, permitiendo validar el funcionamiento del periférico sin requerir un dispositivo externo.

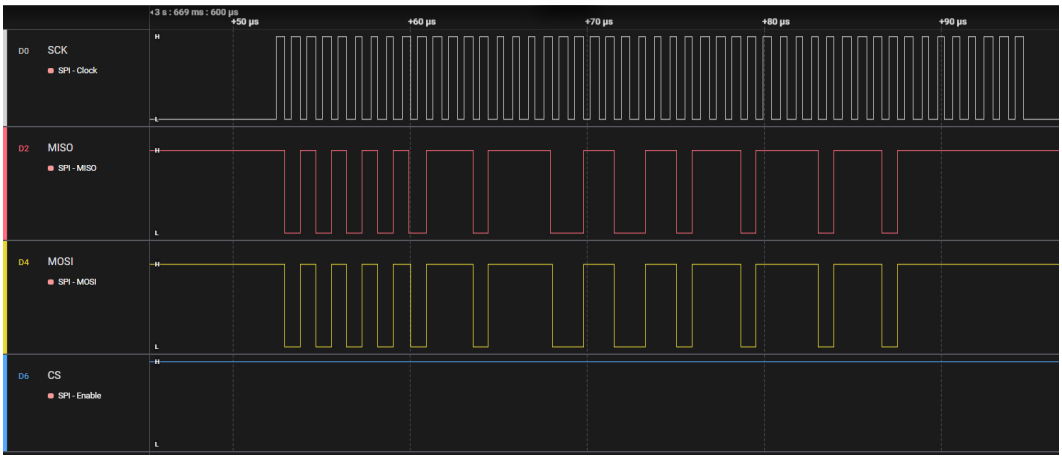

FIGURA 4.25. Captura de trama no decodificada debido a $\overline{\text{CS}}$ inactivo durante la transmisión.

pueda decodificar la trama. Esto convierte a la validación estructural de las señales en un mecanismo indirecto pero robusto para detectar fallas en la transmisión, incluso antes de analizar los datos recibidos. Del mismo modo, parámetros como la polaridad y fase del reloj (CPOL y CPHA), el tamaño de palabra (8 o 16 bits) y el orden de los bits (MSB o LSB primero) deben coincidir exactamente con la configuración real del *bus*, ya que cualquier discrepancia en estos aspectos imposibilitará la decodificación correcta por parte del analizador lógico.

Como se verá en la siguiente sección, este comportamiento fue aprovechado intencionalmente para verificar que las funciones de configuración del controlador SPI apliquen correctamente los parámetros establecidos, ya que la imposibilidad de decodificación ante una configuración incorrecta actúa como una señal clara de fallo funcional.

La Figura 4.7 muestra un fragmento representativo del archivo de salida:

```
spi-1: mosi-bit: "1"
spi-1: mosi-bit: "0"
...
spi-1: mosi-data: "AAAA"
spi-1: miso-data: "AAAA"
...
spi-1: mosi-transfer: "AAAA BBBB CCCC DDDD EEEE FFFF"
spi-1: miso-transfer: "FFFF FFFF FFFF FFFF FFFF FFFF"
```

CÓDIGO 4.7. Fragmento de la salida decodificada del analizador lógico.

En este archivo, las líneas con sufijo `-bit` representan los valores individuales capturados en cada flanco de reloj SPI. A partir de estos bits, el analizador reconstruye los valores transmitidos por palabra, que aparecen como `mosi-data` o `miso-data`. Finalmente, las líneas `mosi-transfer` y `miso-transfer` corresponden a las secuencias completas decodificadas durante una sesión de comunicación SPI, es decir, desde la activación hasta la desactivación de la línea $\overline{\text{CS}}$. Estas últimas son las que se utilizan como referencia para las validaciones automáticas realizadas por los *scripts* en la Raspberry Pi.

Si bien el analizador lógico no actúa como un esclavo real (ya que no retorna datos por la línea MISO), su integración con la configuración *loopback* brinda una herramienta completa y confiable para la prueba funcional del periférico SPI implementado.

Pruebas de validación del periférico SPI. Cada una de las pruebas fue verificada mediante el análisis en *python* de las de las tramas recibidas en el analizador lógico y/o mediante

comparaciones internas entre *buffers* transmitidos y recibidos. Esto permitió validar las funciones de configuración de los modos de operación, el tamaño de palabra, el orden de bits y el correcto cálculo del CRC en *hardware*.

- **Transmisión de bytes fijos en modo 8 bits**
  - `test_transmit_8bits()`
  - Configuración: Modo 0, 8 bits, MSB primero, sin CRC.
  - Datos transmitidos: `0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF`
  - Objetivo: Validar que el analizador lógico decodifique la misma trama y que ésta coincida con lo esperado mediante una validación en `pytest`.
- **Transmisión de palabras de 16 bits**
  - `test_transmit_16bits()`
  - Configuración: Modo 2, 16 bits, MSB primero.
  - Datos transmitidos: `0xAAAA, 0xBBBB, 0xCCCC, 0xDDDD, 0xEEEE, 0xFFFF`
  - Objetivo: Verificar la lógica de funcionamiento para transmisión de datos de 16 bits, validando la salida del analizador lógico con `pytest`.
- **Transmisión y recepción en modo 8 bits**
  - `test_transmit_receive_8bits()`
  - Configuración: Modo 0, 8 bits, MSB primero.
  - Descripción: Se transmite un vector de datos y se compara internamente en la STM32 con el vector de recepción. Aunque se espera una coincidencia por el modo *loopback*, el objetivo principal es verificar la lógica de recepción.
  - Resultado: Se transmite un mensaje de éxito o fallo que sera validado por la Raspberry Pi.
- **Transmisión y recepción de 16 bits**
  - `test_transmit_receive_16bits()`
  - Configuración: Modo 2, 16 bits, MSB primero.
  - Descripción: Se transmite un vector de datos y se compara internamente en la STM32 con el vector de recepción. Aunque se espera una coincidencia por el modo *loopback*, el objetivo principal es verificar la lógica de recepción de 16bits.
  - Resultado: Se transmite un mensaje de éxito o fallo que sera validado por la Raspberry Pi.
- **Transmisión en orden LSB primero**
  - `test_transmit_16bits_lsb()`
  - Configuración: Modo 2, 16 bits, LSB primero.
  - Objetivo: Validar la correcta configuración del orden de los bits comparando los bits decodificados por el analizador con el resultado esperado.

Antes de ejecutar las pruebas de recepción, se modificó el valor por defecto utilizado como dato *dummy* (anteriormente `0xFFFF`), con el objetivo de facilitar la detección de posibles errores asociados a líneas de datos flotantes. En este contexto, se definieron los valores `0xCD` para recepciones de 8 bits y `0xABCD` para recepciones de 16 bits, lo que

permite distinguir con mayor claridad si un dato leído igual a `0x00` o `0xFF` proviene de un estado forzado de las líneas MISO (en bajo o alto) o si se trata efectivamente del patrón transmitido por el maestro.

De esta manera, se espera que el vector recibido durante las pruebas de recepción esté compuesto exclusivamente por los valores *dummy* definidos. Cualquier desviación respecto a estos patrones se interpreta como un fallo en la línea de transmisión o en el periférico SPI.

- **Recepción de bytes**
  - `test_receive_8bits()`
  - Configuración: Modo 0, 8 bits, MSB primero.
  - Descripción: Se realiza la lectura de 6 bytes utilizando un valor *dummy* de `0xCD` para generar los pulsos de reloj. Se verifica si el vector recibido corresponde al patrón esperado.
- **Recepción de palabras de 16 bits**
  - `test_receive_16bits()`
  - Configuración: Modo 0, 16 bits, MSB primero.
  - Descripción: Se realiza la lectura de 6 palabras utilizando el valor *dummy* `0xABCD`. Se valida que el vector recibido coincida con los valores enviados.

  Como se explicó en la Sección 4.3.6, las funciones `spi_transmit()` y `spi_receive()` comparten una lógica interna común, basada en la operación simultánea de transmisión y recepción del protocolo SPI. Sin embargo, su propósito difiere, así como también los datos considerados relevantes. Las limitaciones del modo *loopback* para validar funciones de recepción motivan que estas pruebas se complementen en la Sección 4.3.7, utilizando dispositivos esclavos reales.
- **Transmisión aleatoria con CRC-8**
  - `test_transmit_8bits_crc()`
  - Configuración: Modo 0, 8 bits, CRC habilitado con polinomio `0x07`.
  - Descripción: Se genera un vector de hasta 64 bytes con contenido aleatorio. La trama incluye automáticamente el CRC calculado por *hardware*.
- **Transmisión aleatoria con CRC-16**
  - `test_transmit_16bits_crc()`
  - Configuración: Modo 0, 16 bits, CRC-16-CCITT con polinomio `0x1021`.
  - Descripción: Se transmite un vector aleatorio de hasta 64 palabras, incluyendo el valor de CRC generado por el periférico.

El objetivo principal de estas dos últimas pruebas es validar el funcionamiento correcto de la lógica de cálculo de CRC en el periférico SPI. Se utiliza el mismo polinomio tanto en el transmisor como en el *script* que analiza los datos desde la Raspberry Pi. El analizador lógico permite verificar que el CRC transmitido por *hardware* coincida con el que es reconstruido en *software*, validando tanto la configuración del polinomio como la ejecución de la operación en tramas de 8 y 16 bits.

Validación con periféricos reales: memoria Winbond y tarjeta SD

La segunda etapa de la validación consistió en el uso de periféricos reales: una **memoria flash NOR *Winbond W25Q32*** y una **tarjeta de memoria SD**. Ambos dispositivos, además de aportar funcionalidades útiles para el proyecto (almacenamiento de datos), permiten validar el protocolo SPI en condiciones reales, utilizando esclavos nativos que responden a los comandos según sus especificaciones técnicas. Esto evita las posibles incongruencias derivadas de validar un controlador maestro con un esclavo desarrollado por el mismo programador.

La memoria **W25Q32** se comunica mediante el protocolo SPI y ofrece múltiples comandos para su operación, incluyendo escritura de páginas, lectura secuencial, borrado sectorial y borrado completo. Además, permite obtener el identificador del fabricante y del dispositivo, lo cual fue utilizado durante las pruebas iniciales para verificar la comunicación básica con el periférico.

Como parte del presente trabajo, se desarrollaron controladores básicos para este módulo, con el objetivo de facilitar pruebas automatizadas de lectura, escritura y borrado sobre distintas direcciones de memoria. Estas funciones encapsulan las secuencias requeridas por el protocolo de la memoria, y permitieron validar tanto el funcionamiento de la interfaz SPI como la integridad de los datos escritos.

El diseño de estos controladores se basó en la documentación oficial del fabricante [40], la cual detalla la estructura de comandos, tiempos de programación y características eléctricas del dispositivo.

Las tarjetas SD requieren una inicialización más compleja y el uso de comandos definidos por la especificación SD. Para su manejo, se integró el sistema de archivos *FatFs*, ampliamente utilizado en sistemas embebidos, que proporciona una capa de abstracción sobre las operaciones de bajo nivel [41]. La integración de dicho sistema de archivos con el middleware necesario para su funcionamiento en el STM32 se basó en una implementación desarrollada por un miembro de la comunidad, la cual fue modificada para adaptarse a las bibliotecas y funciones utilizadas en este proyecto [42].

El sistema *FatFs* permite crear, leer, escribir y borrar archivos utilizando funciones similares a las de cualquier sistema operativo, lo que facilita el desarrollo de pruebas funcionales. La correcta integración del controlador SPI con la memoria SD fue un indicador clave de la robustez de la implementación, ya que cualquier falla en la temporización, en la selección del dispositivo o en la transferencia de datos se manifestaría directamente como errores en la apertura, lectura y escritura de los archivos.

A diferencia de la técnica *loopback*, el uso de estos dispositivos reales permitió validar completamente el protocolo SPI, asegurando que las transacciones se desarrollen según lo especificado y que el controlador implemente correctamente todos los requisitos del protocolo.

Pruebas funcionales sobre la memoria flash Winbond W25Q32

Con el objetivo de validar la correcta operación de la memoria flash externa W25Q32, se desarrolló un conjunto de pruebas automatizadas orientadas a verificar tanto la integridad de los comandos de identificación como las operaciones fundamentales de escritura, lectura y borrado. Todas las pruebas se ejecutaron desde una tarea dedicada de FreeRTOS y utilizan la interfaz SPI previamente validada.

Durante la primera inicialización, se ejecuta un borrado completo del chip (`Chip Erase`) con el fin de eliminar posibles datos residuales de ejecuciones anteriores.

- **Test 1: Identificador de fabricante y dispositivo**

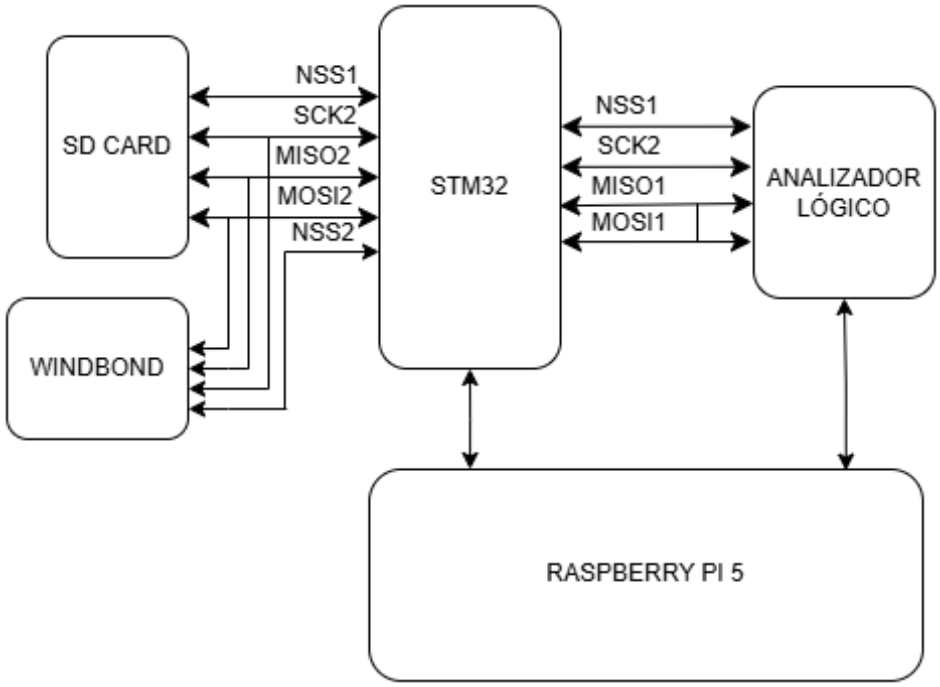

FIGURA 4.26. Banco de pruebas controlador SPI.

- **Comando**: `0x90`
- **Objetivo**: Leer el identificador de 16 bits proporcionado por el fabricante.
- **Validación**: El valor leído se envía a la Raspberry Pi a través de la interfaz UART para ser comparado con el valor esperado, previamente conocido.

- **Test 2: Identificador JEDEC**
  - **Comando**: `0x9F`
  - **Objetivo**: Leer el identificador JEDEC de 24 bits.
  - **Validación**: El valor se envía a la Raspberry Pi, donde se compara con el identificador esperado del modelo W25Q32.
- **Test 3: Número de identificación único (UID)**
  - **Comando**: `0x4B`
  - **Objetivo**: Leer el UID de 64 bits único por dispositivo.
  - **Validación**: El UID obtenido se transmite a la Raspberry Pi, que lo compara con el valor previamente almacenado como referencia.
- **Test 4: Escritura de cadena aleatoria**
  - **Acción**: Se genera una cadena alfanumérica aleatoria de entre 200 y 399 bytes.
  - **Dirección**: Elegida aleatoriamente, alineada a una página de 256 bytes.
  - **Validación**: Se transmite a la Raspberry Pi tanto la dirección como el contenido escrito, para su almacenamiento y futura comparación.
- **Test 5: Lectura del último bloque escrito**
  - **Acción**: Se lee desde la dirección utilizada en el Test 4.

  - **Validación**: Los datos leídos se envían a la Raspberry Pi, donde son comparados con el contenido original previamente recibido.

- **Test 6: Borrado de sectores afectados y verificación parcial**
  - **Acción**: Se identifican y borran los sectores de 4 KB donde se realizó la escritura en el Test 4.
  - **Verificación**: Se seleccionan *offsets* aleatorios dentro de cada página del sector y se verifica que su contenido sea `0xFF`.
  - **Validación**: Se envía a la Raspberry Pi un mensaje que indica el éxito o el fallo de la operación.

- **Test 7: Escrituras aleatorias + Chip Erase + verificación**
  - **Acción**:
    - Escritura de cinco páginas con datos aleatorios en direcciones aleatorias.
    - Ejecución del comando `Chip Erase`.
    - Lectura completa de las páginas previamente escritas.
  - **Validación**: Se espera que todos los bytes leídos después del borrado contengan el valor `0xFF`, indicando que el borrado fue exitoso.

Estas pruebas permiten validar de forma exhaustiva el funcionamiento de la memoria flash en cuanto a identificación, escritura, lectura, borrado por sectores y borrado completo. Adicionalmente, de forma inherente al uso de dicha memoria, se verifica la integridad de la comunicación SPI, tanto en la transmisión como en la recepción de datos.

Pruebas funcionales sobre la memoria flash SD

Con el fin de verificar el correcto funcionamiento de la memoria microSD y su integración con el sistema de archivos, se implementó un conjunto de pruebas automáticas utilizando funciones encapsuladas en una API basada en la biblioteca FatFs.

- **Test 1: Montaje del sistema de archivos**
  - **Objetivo**: Validar que la tarjeta microSD sea detectada correctamente y que el sistema de archivos se monte de forma exitosa.
  - **Validación**: Se transmite un mensaje por UART indicando el éxito o el fallo del montaje inicial.

- **Test 2: Creación de archivo con contenido aleatorio**
  - **Acción**: Se genera un nombre de archivo aleatorio de 8 caracteres seguido de la extensión `.txt`, y se escribe un contenido aleatorio de 200 caracteres en dicho archivo.
  - **Validación**: Se confirma la correcta creación del archivo y se reporta tanto su nombre como su contenido vía UART.

- **Test 3: Lectura del archivo creado**
  - **Acción**: Se accede al archivo creado en el Test 2 y se recupera su contenido.
  - **Validación**: El nombre y contenido del archivo leído se envían por UART para ser comparados con los datos generados previamente, asegurando que la operación de lectura fue exitosa y sin corrupción de datos.

- **Test 4: Borrado del archivo creado**
  - **Acción**: Se elimina el archivo previamente creado en el Test 2 utilizando su nombre guardado. A continuación, se solicita un listado de los archivos presentes en la raíz de la tarjeta.
  - **Validación**: Se transmite por UART un mensaje de confirmación del borrado, seguido del listado actualizado de archivos. Este listado es utilizado para verificar que el archivo efectivamente ya no se encuentra presente en el sistema de archivos.

Estas pruebas permiten validar la funcionalidad esencial del sistema de archivos: montaje, escritura y lectura. Como se detalla en la Sección 5.4, esta memoria se utiliza posteriormente como unidad de almacenamiento persistente para registrar los datos generados por los sensores y eventos del sistema durante la operación nominal del CubeSat.

Capítulo 5

Resultados

Este capítulo presenta los principales resultados obtenidos a partir del desarrollo e implementación del sistema propuesto. Se incluye una ejecución completa del *pipeline* de integración continua, junto al desarrollo completo de los datos que recibe la Raspberry Pi para determinar si una prueba fue exitosa. Además, se contrasta el estado inicial del prototipo —utilizado para validaciones preliminares— con la versión final más estable y organizada del hardware, y se incluye una vista general de la interfaz gráfica desarrollada para visualizar los datos en tiempo real.

5.1. Resultados por protocolo de comunicación

En esta sección se presentan ejemplos representativos de las pruebas desarrolladas para cada protocolo de comunicación implementado en el sistema. Cada caso muestra el mensaje recibido por la Raspberry Pi que le permite decidir si una prueba fue exitosa, ya sea mediante lectura UART directa o por verificación con herramientas externas como un analizador lógico.

5.1.1. UART

Dado que el protocolo UART tiene la ventaja de poder conectar directamente la STM32 con la Raspberry Pi, la verificación puede realizarse de forma inmediata, sin necesidad de decodificación o interpretación adicional: si la cadena recibida coincide exactamente con la esperada, la prueba se considera exitosa.

Prueba 1: Transmisión de texto por USART1 Esta prueba consiste en enviar un mensaje de texto formateado a través del periférico USART1. El mensaje identifica el número de USART utilizado y permite verificar que la transmisión básica de caracteres ASCII se realiza correctamente.

Para la verificación, se recibe a través del puerto USART en la Raspberry Pi la cadena enviada por la STM32 y se comprueba que sea lo que se esperaba.

```
>>  Esta es una prueba de transmision por UART 1
```

Prueba 2: Recepción y eco de datos por USART1 En esta prueba, la STM32 recibe un bloque de datos provenientes de la Raspberry Pi, almacenándolos en el *buffer* de recepción. Posteriormente, el contenido del *buffer* es transmitido nuevamente (eco) hacia la Raspberry Pi, permitiendo validar la correcta captura y posterior reenvío de datos.

```
>>  Test: Recepcion por USART 1
```

Prueba 3: Manejo de condiciones de *overflow* en recepción Esta prueba fuerza la sobrecarga del *buffer* de recepción, enviando un volumen de datos que excede la capacidad esperada. El objetivo es evaluar el comportamiento del sistema ante situaciones de *overflow*.

Se realiza el envío de una cadena larga de caracteres desde la Raspberry Pi hacia la STM32, la cual devuelve lo recibido y luego se compara con lo esperado.

En esta prueba también incide el tamaño máximo del *buffer* `rxq` de la STM32.

```
>>  Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus nec
   est sit amet nisl hendrerit porttitor. Nulla facilisi. Nulla erat
   risus, maximus sed nisi id, auctor interdum lacus. Suspendisse in
   accumsan tortor. Maecenas molestie ante ac ullamcorper faucibus. Donec
    hendrerit luctus ultrices. Nullam at tellus et leo posuere bibendum.
   Donec faucibus efficitur mauris, quis elementum mauris finibus in.
   Donec non ipsum ex. Phasellus pharetra in risus in elementum. Fusce
   dignissim diam ac justo suscipit orc
```

Prueba 4: Transmisión de texto por USART3 De manera análoga a la Prueba 1, se realiza una transmisión de un mensaje formateado, pero utilizando el periférico USART3. Esta prueba valida la correcta configuración y operación de múltiples instancias de UART de manera independiente y su validación se realiza mediante el analizador lógico.

```
>>  ['45', '73', '74', '61', '20', '65', '73', '20', '75', '6E', '61',
   '20', '70', '72', '75', '65', '62', '61', '20', '64', '65', '20',
   '74', '72', '61', '6E', '73', '6D', '69', '73', '69', '6F', '6E',
   '20', '70', '6F', '72', '20', '55', '41', '52', '54', '20', '33', '0A
   ']
```

Prueba 5: Transmisión de datos binarios Se realiza el envío de una secuencia de bytes binarios desde 0 hasta 99 a través de USART1. Esta prueba permite verificar que la transmisión UART soporta no sólo caracteres imprimibles, sino cualquier tipo de dato binario, evaluando la transparencia de la comunicación para distintos valores de byte.

```
>>  ['00', '01', '02', '03', '04', '05', '06', '07', '08', '09', '0a', '0
   b', '0c', '0d', '0e', '0f', '10', '11', '12', '13', '14', '15', '16',
   '17', '18', '19', '1a', '1b', '1c', '1d', '1e', '1f', '20', '21',
   '22', '23', '24', '25', '26', '27', '28', '29', '2a', '2b', '2c', '2d
   ', '2e', '2f', '30', '31', '32', '33', '34', '35', '36', '37', '38',
   '39', '3a', '3b', '3c', '3d', '3e', '3f', '40', '41', '42', '43',
   '44', '45', '46', '47', '48', '49', '4a', '4b', '4c', '4d', '4e', '4f
   ', '50', '51', '52', '53', '54', '55', '56', '57', '58', '59', '5a',
   '5b', '5c', '5d', '5e', '5f', '60', '61', '62', '63']
```

Prueba 6: Prueba de estrés de recepción Se crea una tarea dedicada (`vStressTask`) que monitorea la llegada de datos en USART1 de manera continua. La tarea acumula los valores recibidos y, tras múltiples lecturas, transmite la suma total de los datos al *host*. Esta prueba simula una condición de tráfico sostenido de datos, evaluando el comportamiento del sistema ante cargas elevadas y la correcta sincronización entre recepción e interpretación de datos.

```
>>  Suma de datos: 1000
```

Prueba 7: Prueba del mutex Esta prueba se basa en el correcto funcionamiento del objeto interno *mutex* del protocolo UART. En esta ocasión se prueba tomar el mismo y luego liberarlo correctamente. La validación se hace internamente en la STM32 y se envía la confirmación a la Raspberry Pi por USART1.

```
>>  Test mutex OK
```

Prueba 8: Prueba de la función UART_write Como bien se mencionó en capítulos anteriores, otros protocolos y parte del entorno de desarrollo hacen uso de la interfaz UART tanto para manejo de errores como para comunicación en las pruebas. Es por eso que se incluye una prueba de validación de la firma de la función de transmisión de este protocolo, para que ante algún cambio no se impacte en otros módulos.

Se envía una cadena con el uso de esta función a probar desde la STM32 hacia la Raspberry Pi, en donde se hace la validación.

```
>>  Test funcion UART_write
```

GPS

Debido a la naturaleza de este periférico, que requiere una conexión directa entre la antena y los satélites para obtener datos, no fue posible incluirlo en las pruebas de integración continua. Esto se debe a que, en muchos casos, la ubicación del *hardware* no permitía garantizar la recepción de señal GPS confiable. No obstante, se desarrolló una rutina específica que, en intervalos definidos, accede al buffer de recepción de `USART1` —interfaz a la cual está conectado el módulo GPS— para realizar la lectura de los datos disponibles.

En primer lugar, se verifica la presencia de datos en el buffer. Si este se encuentra vacío, se emite un mensaje indicando *No se encontraron datos*. En caso contrario, se procede al análisis de la trama NMEA recibida. Esta puede ser válida o inválida, dependiendo de si el módulo GPS logró establecer comunicación con los satélites. Si la trama no es válida, se notifica mediante el mensaje *Error GPS*.

Cuando se recibe información válida, se decodifican los mensajes NMEA del tipo GGA y RMC utilizando las funciones `decodeGGA` y `decodeRMC`, respectivamente. Estas funciones asignan los valores obtenidos a los campos correspondientes de la estructura definida en el archivo `NMEA.h`. Una vez completado este proceso de forma satisfactoria, se genera un mensaje con el siguiente formato:

Latitud - Longitud - Día/Mes/Año Hora:Min:Seg

En la Figura 5.1, se presenta un ejemplo ilustrativo del comportamiento descrito. En este caso, la tarea de procesamiento del GPS se encuentra en ejecución desde el inicio, comenzando con el periférico desconectado. Posteriormente, se conecta el módulo GPS a la STM32, lo que da lugar a la recepción de datos inválidos. Finalmente, una vez que la antena logra captar correctamente la señal de los satélites, se obtiene y decodifica exitosamente una trama NMEA válida.

5.1.2. I^2C

PCF8574/SN74HC4066

Prueba 1 Para esta prueba, se intentan realizar mediciones con el sensor HTU21D y la unidad inercial MPU6050. La primera etapa del procedimiento consiste en:

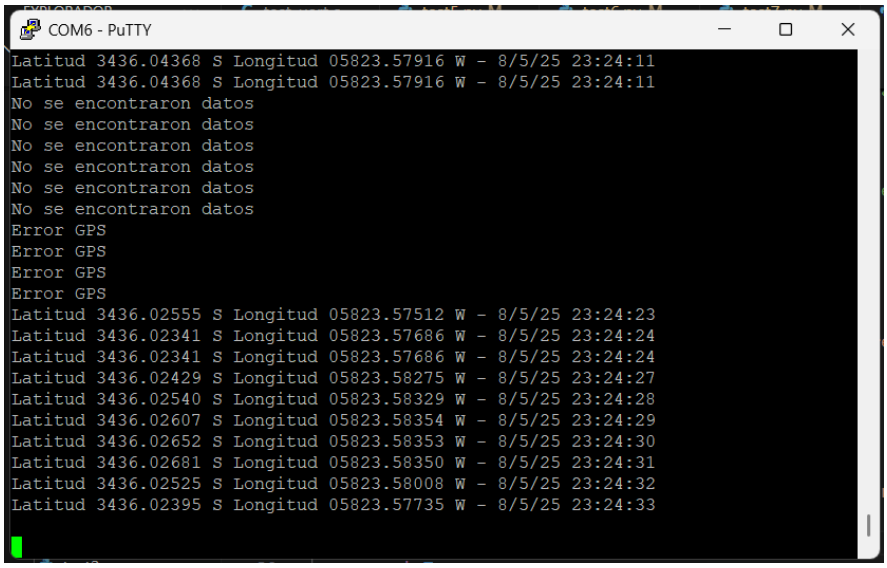

FIGURA 5.1. Resultados tarea decodificación GPS

1. Habilitar todas las conexiones
2. Reiniciar el HTU21D
3. Solicitar una medición de temperatura
4. Solicitar una medición de humedad
5. Validar los CRC correspondientes
6. Activar la MPU6050
7. Configurar la escala y realizar la auto-calibración de la MPU6050
8. Si se recibió, enviar el valor de temperatura por serie
9. Leer la aceleración y, si se recibió, enviar cada eje por serie
10. Si se recibió, enviar el valor de humedad por serie
11. Leer el giróscopo y, si se recibió, enviar cada eje por serie
12. Poner la MPU6050 en modo `SLEEP`

Resultado. Los mensajes validados por la Raspberry Pi son, por ejemplo:

```
>>  Temp: 24.98 C
>>  Acel en X: 0.001709 g
>>  Acel en Y: 0.002075 g
>>  Acel en Z: 1.006409 g
>>  Hum: 67.85 %
>>  Giro en X: -0.061069 dps
>>  Giro en Y: 0.053435 dps
>>  Giro en Z: -0.106870 dps
```

La Raspberry Pi extrae los valores obtenidos y comprueba que los mismos se encuentren dentro de rangos válidos, lo que sirve como mecanismo adicional para validar que fueron adquiridos y traducidos correctamente.

Prueba 2 Se utilizaron los sensores HTU21D y MPU6050, con el objetivo de verificar que los errores derivados de la desconexión de un dispositivo se comuniquen correctamente, y que el sistema continúe funcionando de forma parcial con los sensores que permanecen accesibles.

El procedimiento de prueba es similar al caso anterior. La única modificación es que, en el primer paso, se habilitan todas las conexiones con excepción del HTU21D.

Resultado. Los mensajes validados por la Raspberry Pi son, por ejemplo:

1. Errores correspondientes a intentar resetear el sensor HTU21D. Por implementación, se realiza 3 veces antes de determinar el fallo. Como el sensor esta desconectado, el error ocurre en la primera instancia de la comunicación, que es en la dirección al intentar enviar el comando con `i2c_send_to_slave()`

```
>>  NACK recibido en i2c_wait_until_address
>>  Error en i2c_wait_until_address en i2c_send_address
>>  Error en i2c_send_address en i2c_send_to_slave

>>  NACK recibido en i2c_wait_until_address
>>  Error en i2c_wait_until_address en i2c_send_address
>>  Error en i2c_send_address en i2c_send_to_slave

>>  NACK recibido en i2c_wait_until_address
>>  Error en i2c_wait_until_address en i2c_send_address
>>  Error en i2c_send_address en i2c_send_to_slave

>>  Error en reset_htu21d
```

2. Errores correspondientes a intentar solicitar datos de temperatura del HTU21D (se permitió acceder incluso si el *reset* falló para visualizar los errores)

```
>>  NACK recibido en i2c_wait_until_address
>>  Error en i2c_wait_until_address en i2c_send_address
>>  Error en i2c_send_address en i2c_send_to_slave
>>  Error al solicitar temperatura en request_htu21d
```

3. Análogamente, se reciben los errores al intentar solicitar los datos de humedad

```
>>  NACK recibido en i2c_wait_until_address
>>  Error en i2c_wait_until_address en i2c_send_address
>>  Error en i2c_send_address en i2c_send_to_slave
>>  Error al solicitar humedad en request_htu21d
```

4. Aunque hayan ocurrido errores con otros dispositivos, el *bus* debe seguir disponible y se deben recibir los datos de la MPU6050 (que está conectada)

```
>>  Acel en X: -0.002014 g
>>  Acel en Y: 0.000977 g
>>  Acel en Z: 1.007935 g
>>  Giro en X: 0.015267 dps
>>  Giro en Y: 0.412214 dps
>>  Giro en Z: -0.091603 dps
```

En este escenario, al igual que en muchas de las pruebas que siguen, la Raspberry Pi verifica que los errores ocurran en el orden y forma esperados, según una secuencia previamente definida. Esto permite confirmar que el flujo de control ante fallos se está ejecutando correctamente.

Prueba 3 La prueba comienza con la desconexión de todos los sensores. Al intentar realizar un reinicio del HTU21D, se producirá un error. La STM32 enviará entonces los mensajes correspondientes por puerto serie, finalizando la rutina.

Resultado. Los mensajes validados por la Raspberry Pi son:

```
>>  NACK recibido en i2c_wait_until_address
>>  Error en i2c_wait_until_address en i2c_send_address
>>  Error en i2c_send_address en i2c_send_to_slave

>>  NACK recibido en i2c_wait_until_address
>>  Error en i2c_wait_until_address en i2c_send_address
>>  Error en i2c_send_address en i2c_send_to_slave

>>  NACK recibido en i2c_wait_until_address
>>  Error en i2c_wait_until_address en i2c_send_address
>>  Error en i2c_send_address en i2c_send_to_slave

>>  Error en reset_htu21d
```

De esta manera, se verifica que el sensor fue desconectado y que se intentó la comunicación sin éxito en tres oportunidades (por implementación).

Antes de concluir, se activa la tarea `vTaskCheckSensors`.

ATmega328p

En este apartado, se detallan las pruebas de envío y recepción de datos que verifican el funcionamiento exclusivo del controlador I^2C.

Debe tenerse en cuenta que, por la implementación del código del ATmega328P, el envío de datos posee un encabezado adicional (`0x00`) que le indica al dispositivo que debe actuar como receptor.

Prueba 1 En esta prueba, se envía el caracter ASCII 'a' al ATmega328P y se verifica, mediante el analizador lógico, si la secuencia de señales es la esperada.

Resultado. La Raspberry Pi debe recibir todos los componentes de la comunicación:

```
>>  ['Start', 'Write', '04', 'ACK', '00', 'ACK', '61', 'ACK', 'Stop']
```

donde `04` corresponde al valor hexadecimal de la dirección asignada al ATmega328P, `00` al encabezado, `61` al valor ASCII del caracter 'a', todos ellos con su ACK correspondiente. La Raspberry Pi compara esta cadena con una almacenada previamente y, si son iguales, declara el test como exitoso.

Prueba 2 Se envian las dos cadenas "Prueba extensa numero 1" y "Prueba extensa numero 2", ambos en sus caracteres ASCII).

Resultado. Los datos validados por la Raspberry Pi son:

```
>>  ['Start', 'Write', '04', 'ACK', '00', 'ACK', '50', 'ACK', '72', 'ACK
    ', '75', 'ACK', '65', 'ACK', '62', 'ACK', '61', 'ACK', '20', 'ACK',
    '65', 'ACK', '78', 'ACK', '74', 'ACK', '65', 'ACK', '6E', 'ACK', '73',
     'ACK', '61', 'ACK', '20', 'ACK', '6E', 'ACK', '75', 'ACK', '6D', 'ACK
    ', '65', 'ACK', '72', 'ACK', '6F', 'ACK', '20', 'ACK', '31', 'ACK', '
    Stop', 'Start', 'Write', '04', 'ACK', '00', 'ACK', '50', 'ACK', '72',
    'ACK', '75', 'ACK', '65', 'ACK', '62', 'ACK', '61', 'ACK', '20', 'ACK
    ', '65', 'ACK', '78', 'ACK', '74', 'ACK', '65', 'ACK', '6E', 'ACK',
    '73', 'ACK', '61', 'ACK', '20', 'ACK', '6E', 'ACK', '75', 'ACK', '6D',
     'ACK', '65', 'ACK', '72', 'ACK', '6F', 'ACK', '20', 'ACK', '32', 'ACK
    ', 'Stop']
```

Se identifica la dirección del ATmega328P, seguida por el encabezado correspondiente al modo de prueba. Luego se transmite la primera secuencia convertida a su representación ASCII, seguida por una señal de parada. A continuación, se observa una nueva señal de inicio que marca el comienzo de la segunda secuencia, también en ASCII, finalizando con una segunda señal de parada.

Prueba 3 Se envía un conjunto de bytes a la dirección `0x23`, la cual no corresponde a ningún dispositivo conectado al bus.

Resultado. Los mensajes validados por la Raspberry Pi son:

```
>>  ['Start', 'Write', '23', 'NACK', 'Stop']
>>  NACK recibido en i2c_wait_until_address
>>  Error en i2c_wait_until_address en i2c_send_address
>>  Error en i2c_send_address en i2c_send_to_slave
```

Se verifica tanto la secuencia de señales mediante el analizador lógico como la aparición de los errores esperados por el puerto serie. La traza del analizador muestra un intento de comunicación en modo escritura con el esclavo `0x23`, que no respondió (`NACK`). Por su parte, los mensajes de error reflejan correctamente que el `NACK` fue detectado al esperar la confirmación de dirección, dentro de la función que gestiona la transmisión.

Prueba 4 Esta prueba valida el tratamiento adecuado de una interrupción en medio de una comunicación. El procedimiento consiste en los siguientes pasos:

1. Se crea una cadena de caracteres de 32 bytes de extensión (capacidad máxima del *buffer* de I^2C del ATmega328P en una única comunicación).
2. Se selecciona un largo aleatorio para este vector de datos y se rellena con caracteres ASCII seleccionados de forma aleatoria.
3. Se comunica, por puerto serie, el mensaje que se enviará por I^2C.
4. Se envía el vector utilizando la función de envío de datos bajo el modo de prueba 1. En este modo, la función tiene la capacidad de seleccionar un valor aleatorio, comunicarlo por puerto serie y desconectar el ATmega328P en el momento elegido.

Resultado. Los datos validados por la Raspberry Pi son los siguientes:

```
>>  ['Start', 'Write', '04', 'ACK', '00', 'ACK', '6C', 'ACK', '63', 'ACK
    ', '61', 'ACK', '62', 'ACK', '77', 'ACK', '67', 'ACK', '6D', 'ACK',
    '73', 'ACK', '63', 'ACK', '72', 'ACK', '6E', 'ACK', '6F', 'NACK', '
    Stop']
```

```
>>  Mensaje: lcabwgmscrnoiaftlfpcuqffaxozqe
>>  Random: 12
>>  Timeout esperando el bit de TXE en i2c_send_to_slave
```

En este punto, se observa que el mensaje a enviar originalmente tenía 30 caracteres. El valor aleatorio donde se corta la comunicación es el número 12, lo que implica que el duodécimo dato recibido (sin contar el encabezado) no fue recibido, ya que se muestra un NACK. Esto ocurre porque nunca se pudo liberar el *buffer* de transmisión, lo que genera un *timeout*. A pesar de este error, la señal de *stop* es enviada al bus, signo de un correcto tratamiento de errores.

Al finalizar esta prueba, se solicita a la STM32 reconectar el ATmega328P para restaurar el estado original y continuar con las pruebas.

Prueba 5 Se verifica el funcionamiento del *mutex* del controlador. Para ello, se envía un mensaje al ATmega328P utilizando la función correspondiente en el modo de prueba número 2. Esto implica que, al enviar el primer byte, también se intenta enviar un mensaje al HTU21D por el mismo puerto I^2C.

Resultado. Los datos validados por la Raspberry Pi son los siguientes:

```
>>  Valor capturado: ['Start', 'Write', '04', 'ACK', '00', 'ACK', '54', '
   ACK', '45', 'ACK', '53', 'ACK', '54', 'ACK', '20', 'ACK', '43', 'ACK',
    '4F', 'ACK', '4C', 'ACK', '4C', 'ACK', '49', 'ACK', '53', 'ACK',
   '49', 'ACK', '4F', 'ACK', '4E', 'ACK', 'Stop']
>>  Error tomando el semaforo en i2c_take_mutex
>>  Error tomando el semaforo en i2c_send_to_slave
```

Se puede observar que el mensaje enviado al ATmega328P fue transmitido correctamente, y la petición de enviar un dato al HTU21D fue rechazada de forma adecuada. Esto ocurrió porque no se pudo tomar el *mutex*, ya que en ese momento el *bus* estaba ocupado.

Pruebas 6-10 En estas pruebas se verifica la correcta codificación de la función que recibe datos por el puerto I^2C, especialmente en lo que respecta al manejo de casos con 1, 2 o más bytes de datos, debido a la implementación en STM32.

Para cada prueba, se envía un byte de datos al ATmega328P que contiene el encabezado, el cual indica cuántos datos se requerirán, activando así el modo de prueba correspondiente. A continuación, se ejecuta la función de lectura y, utilizando un analizador lógico, se verifica que la salida sea la esperada.

Resultados. Los datos validados por la Raspberrry Pi son los siguientes:

```
>>  ['Start', 'Write', '04', 'ACK', '01', 'ACK', 'Stop', 'Start', 'Read',
    '04', 'ACK', '00', 'NACK', 'Stop']
>>  ['Start', 'Write', '04', 'ACK', '02', 'ACK', 'Stop', 'Start', 'Read',
    '04', 'ACK', '00', 'ACK', '01', 'NACK', 'Stop']
>>  ['Start', 'Write', '04', 'ACK', '03', 'ACK', 'Stop', 'Start', 'Read',
    '04', 'ACK', '00', 'ACK', '01', 'ACK', '02', 'NACK', 'Stop']
>>  ['Start', 'Write', '04', 'ACK', '04', 'ACK', 'Stop', 'Start', 'Read',
    '04', 'ACK', '00', 'ACK', '01', 'ACK', '02', 'ACK', '03', 'NACK', '
   Stop']
>>  ['Start', 'Write', '04', 'ACK', '05', 'ACK', 'Stop', 'Start', 'Read',
    '04', 'ACK', '00', 'ACK', '01', 'ACK', '02', 'ACK', '03', 'ACK',
   '04', 'NACK', 'Stop']
```

En cada una de las pruebas se verifica que se reciben correctamente 1, 2, 3, 4 o 5 bytes, según corresponda. Cada secuencia indica dos comunicaciones: una de escritura en la que se le indica el modo de prueba y una de lectura, en la que se reciben los datos esperados. Durante la lectura, se responde con ACK para todos los datos, excepto para el último byte, donde se responde con NACK.

HTU21D

Se realiza una lecto-escritura de datos para verificar el funcionamiento del controlador implementado para el sensor de temperatura y humedad.

Prueba Los pasos de la prueba implementada son:

1. Hacer un *reset* del HTU21D.
2. Ejecutar y obtener la medición de la temperatura.
3. Ejecutar y obtener la medición de la humedad.
4. Realizar el chequeo CRC-8 y, si falla, comunicar por UART y cortar la transmisión.
5. Comunicar los datos obtenidos (temperatura y humedad) por puerto serie.

Resultado. Los datos validados por la Raspberry Pi son, por ejemplo:

```
>>  24.76 C
>>  64.00 %
```

De esta forma, la Raspberry Pi valida que los valores obtenidos estén dentro de un rango esperado. En este caso, la temperatura debe estar entre 0 y 40 °C, y la humedad relativa entre 0 y 100 %.

MPU6050

Se evalúa el controlador de la unidad de medición inercial, desde su configuración hasta la obtención de valores con y sin desvío.

Prueba 1 La primera prueba consiste en verificar el registro `WHO_AM_I`, mediante la función correspondiente, y enviarlo por puerto serie.

Resultado. El mensaje que la Raspberry Pi espera recibir es:

```
>>  Who am I: 0x68
```

De esta forma, compara la cadena con una almacenada previamente y revisa que no haya llegado ningún mensaje de error.

Prueba 2 Se evalúa el funcionamiento del proceso de auto-calibración de la MPU6050. Para ello, se realiza una secuencia de pasos que permite obtener mediciones de aceleración y velocidad angular tanto con como sin la aplicación del *offset* calculado automáticamente. La secuencia de ejecución es la siguiente:

1. Configurar el reloj interno de la MPU6050 para encender el dispositivo.

2. Configurar la escala de medición, el filtro digital pasa bajos (DLPF) y ejecutar la rutina de auto-calibración.
3. Realizar `N_SAMPLES_TEST` mediciones de aceleración en los tres ejes, sin aplicar el *offset*.
4. Convertir cada valor crudo (entero de 16 bits) a su correspondiente representación decimal en unidades físicas.
5. Aplicar el *offset* calculado previamente mediante la función `mpu6050_get_offsets()`.
6. Enviar los datos obtenidos por puerto serie.
7. Configurar la MPU6050 en modo `SLEEP` para reducir el consumo energético.

Resultado. Los datos validados por la Raspberry Pi son, por ejemplo:

```
>>  Acel en X: 0.112915 g
>>  Acel en X (offset): -0.000977 g
>>  Acel en Y: -0.019043 g
>>  Acel en Y (offset): -0.000305 g
>>  Acel en Z: 0.924805 g
>>  Acel en Z (offset): 0.996887 g
    ...
>>  Acel en X: 0.115601 g
>>  Acel en X (offset): 0.001709 g
>>  Acel en Y: -0.018799 g
>>  Acel en Y (offset): -0.000061 g
>>  Acel en Z: 0.930542 g
>>  Acel en Z (offset): 1.002625 g
```

La Raspberry Pi verifica, en primer lugar, que no se haya detectado ningún error durante la medición. Luego, valida que los valores se encuentren dentro de los rangos esperados, según la configuración seleccionada: para el acelerómetro, entre -2 g y 2 g, mostrando que la obtención y traducción del dato se realizó de forma correcta.

Posteriormente, calcula la suma total de los desvíos absolutos entre las mediciones y los valores ideales en reposo (aceleraciones en X e Y cercanas a 0 g, y en Z cercana a 1 g), tanto en el caso sin compensación como con el *offset* aplicado, verificando que la suma de los errores con compensación sea menor a la suma de errores sin compensación, utilizando múltiples muestras para evitar que el resultado se vea afectado por lecturas atípicas.

Prueba 3 Esta prueba es similar a la anterior, pero con los valores del giroscopio.

Resultado. Los mensajes esperados por la Raspberry Pi son, por ejemplo:

```
>>  Giro en X: -1.030534 dps
>>  Giro en X (offset): 0.007634 dps
>>  Giro en Y: 0.190840 dps
>>  Giro en Y (offset): 0.038168 dps
>>  Giro en Z: -0.931298 dps
>>  Giro en Z (offset): 0.015267 dps
    ...
>>  Giro en X: -1.022901 dps
>>  Giro en X (offset): 0.015267 dps
>>  Giro en Y: 0.190840 dps
>>  Giro en Y (offset): 0.038168 dps
```

```
>>  Giro en Z: -0.931298 dps
>>  Giro en Z (offset): 0.015267 dps
```

El análisis es análogo al caso anterior, evaluando que los datos estén entre `-250°/s` y `250°/s` y con los desvíos calculados respecto al 0 °/s en todos los ejes.

AT24C256

Se evalúa el controlador de la memoria EEPROM, utilizando los distintos tipos de escritura y lectura (un byte o secuencial).

Prueba 1 Los pasos correspondientes a esta prueba son:

1. Ejecutar el borrado completo
2. Seleccionar una dirección aleatoria dentro de la memoria
3. Leer el valor almacenado en dicha posición
4. Enviar los datos de la dirección y el valor por puerto serie

Resultado. El mensaje esperado por la Raspberry Pi es, por ejemplo:

```
>>  Pagina: 46, Byte: 16
>>  Dato leido: 0
```

La Raspberry Pi verifica que no haya ocurrido ningún error y que el dato leido sea igual a cero.

Prueba 2 Se verifica la escritura de un byte en una dirección específica. La STM32 escribe un dato en una posición de la memoria (ambos predefinidos) y comunica el resultado de la operación por puerto serie.

Resultado. Los mensajes validados por la Raspberry Pi son, por ejemplo:

```
>>  NACK recibido en i2c_wait_until_address
>>  Error en i2c_wait_until_address en i2c_send_address
>>  Error en i2c_send_address en i2c_send_to_slave
>>  NACK recibido en i2c_wait_until_address
>>  Error en i2c_wait_until_address en i2c_send_address
>>  Error en i2c_send_address en i2c_send_to_slave
>>  NACK recibido en i2c_wait_until_address
>>  Error en i2c_wait_until_address en i2c_send_address
>>  Error en i2c_send_address en i2c_send_to_slave

>>  Byte escrito correctamente
```

Es importante destacar que una ejecución correcta de la prueba no implica la ausencia de errores previos a la escritura del byte. La aparición de mensajes de NACK durante la espera de la dirección se debe al sondeo que debe realizarse hasta que la memoria finalice la escritura y son contemplados por la Raspberry Pi, quien verifica que el mensaje del resultado haya sido exitoso.

Prueba 3 Se lee el contenido de la dirección utilizada en la prueba anterior y se envía el valor almacenado por puerto serie.

Resultado. El mensaje validado por la Raspberry Pi es:

```
>>  Dato leido: 85
```

La Raspberry Pi verifica que el valor contenido sea igual a `0x55` (decimal 85).

Prueba 4 Se realiza la escritura secuencial de una página completa y se envía un mensaje por puerto serie con resultado de la operación.

Resultado. El mensaje validado por la Raspberry Pi es:

```
>>  Escritura secuencial finalizada
```

La Raspberry Pi compara esta cadena con la almacenada para verificar que sea el mensaje esperado.

Prueba 5 Se realiza una lectura secuencial de la misma página utilizada en la prueba anterior y se envían los valores almacenados por puerto serie.

Resultado. El mensaje validado por la Raspberry Pi, en forma acotada, es:

```
>>  0
>>  1
>>  2
    ...
>>  63
```

La Raspberry Pi verifica que los datos recibidos sean 64 e iguales a los números esperados.

5.1.3. SPI

Validación del periférico SPI

A continuación se describen paso a paso las pruebas realizadas sobre el bus SPI. Cada prueba configura el periférico, ejecuta una operación concreta y transmite los resultados, los cuales se verifican mediante un analizador lógico o validación interna.

Prueba 1

1. Se configura SPI en modo 0.
2. Palabras de 8 bits, orden MSB first.
3. Se transmite la secuencia fija: `0xAA, 0xBB, 0xCC, 0xDD, 0xEE, 0xFF`.
4. El analizador lógico captura la transmisión para su validación.

```
MISO:  ['AA', 'BB', 'CC', 'DD', 'EE', 'FF']
```

Prueba 2

1. Se configura SPI en modo 2.
2. Palabras de 16 bits, orden MSB first.
3. Se transmite la secuencia: `0xAAAA, 0xBBBB, ..., 0xFFFF`.
4. El analizador lógico verifica la salida.

```
MISO:  ['AAAA', 'BBBB', 'CCCC', 'DDDD', 'EEEE', 'FFFF']
```

Prueba 3

1. Se configura SPI en modo 0, 8 bits, MSB first.
2. Se transmite una cadena de texto y se recibe simultáneamente.
3. STM32 compara buffers transmitido y recibido.
4. Si coinciden, se transmite por SPI un mensaje de validación.

```
MOSI: ['54', '65', '73', '74', '20', '74', '72', '61', '6E', '73', '6D',
   '69', '74', '5F', '72', '65', '63', '65', '69', '76', '65', '20',
   '70', '61', '72', '61', '20', '38', '62', '69', '74', '73', '00']
MOSI: ['54', '45', '53', '54', '20', '74', '72', '61', '6E', '73', '6D',
   '69', '74', '5F', '72', '65', '63', '65', '69', '76', '65', '20',
   '38', '42', '49', '54', '53', '20', '50', '41', '53', '53', '45',
   '44']
```

Este último mensaje corresponde en ASCII a "TEST transmit_receive 8BITS PASSED", que es lo que la Raspberry Pi decodifica y utiliza como criterio de validación para declarar la prueba como exitosa.

Prueba 4

1. Se configura SPI en modo 2, palabras de 16 bits, MSB first.
2. Se transmite y recibe una secuencia fija.
3. STM32 compara ambos vectores.
4. Si son iguales, se transmite: `0x1A2B, 0x3C4D, 0x5E6F`.

```
MISO:  ['AAAA', 'BBBB', 'CCCC', 'DDDD', 'EEEE', 'FFFF']
MISO:  ['1A2B', '3C4D', '5E6F']
```

Prueba 5

1. Se configura SPI en modo 2.
2. Palabras de 16 bits, orden LSB first.
3. Se transmite: `0x1234, 0x5678, 0x9ABC`.
4. La secuencia se valida con el analizador lógico.

```
MISO:  ['2C48', '1E6A', '3D59']
MOSI: ['2C48', '1E6A', '3D59']
```

Prueba 6

1. SPI en modo 0, 8 bits, MSB first.
2. Se realiza recepción de 6 bytes.
3. Se modifica el valor dummy transmitido a 0xABCD.
4. Se espera observar el byte bajo: 0xCD repetido.

```
MISO:  ['CD', 'CD', 'CD', 'CD', 'CD', 'CD']
```

Prueba 7

1. SPI en modo 0, 16 bits, MSB first.
2. Se realiza la recepción de 6 palabras.
3. Dummy de transmisión = 0xABCD.
4. Se espera recibir 0xABCD repetido.

```
MISO:  ['ABCD', 'ABCD', 'ABCD', 'ABCD', 'ABCD', 'ABCD']
```

Prueba 8

1. SPI en modo 0, 8 bits, MSB first.
2. CRC habilitado con polinomio 0x07.
3. Se genera una secuencia aleatoria de 1 a 64 bytes.
4. STM32 transmite los datos + CRC.
5. Raspberry Pi recalcula y valida el CRC.

```
MOSI: [116, 77, 115, 76, 205, 52, 58, 20, 51, 228, 240, 240, 254, 202,
    97, 165, 135, 88, 54, 239, 173, 190, 229, 17, 34, 29, 115, 138, 56,
    161, 84, 4, 5, 187, 78, 70, 251, 187, 217, 226, 139, 8, 57, 34, 194,
    203, 230, 196, 180, 237, 189, 199, 220, 122, 157, 19, 241, 178, 41,
    17, 1, 192, 54, 101, 154]
>> CRC calculado: 154
```

Prueba 9

1. SPI en modo 0, 16 bits, MSB first.
2. CRC habilitado con polinomio 0x1021.
3. Se genera una secuencia aleatoria de palabras.
4. STM32 transmite datos + CRC.
5. Raspberry Pi compara CRC recibido vs calculado.

```
MOSI: ['0x974', '0x4c4d', '0x7973', '0xdf4c', '0xd2cd', '0xca34', '0xe23a
   ', '0x8914', '0x2a33', '0x9de4', '0x28f0', '0x23f0', '0x2dfe', '0xfbca
   ', '0xd261', '0x60a5', '0x6d87', '0xed58', '0x2c36', '0x50ef', '0xfead
   ', '0xf8be', '0xbee5', '0x8111', '0xba22', '0x721d', '0x1373', '0x1a8a
   ', '0x138', '0x42a1', '0x8954', '0xaf04', '0x8305', '0xeabb', '0xfc4e
   ', '0xdd46', '0x8dfb', '0xe5bb', '0x21d9', '0x20e2', '0x248b', '0x8908
   ', '0x9439', '0x7122', '0x30c2', '0x56cb', '0xf7e6', '0x90c4', '0xd9b4
   ', '0xc5ed', '0x80bd', '0x97c7', '0x7dc', '0xa67a', '0x909d', '0x5f13
   ', '0x10f1', '0xe4b2', '0x6b29', '0x6411', '0x2a01', '0x4c0', '0xfb36
   ', '0xdd65', '0xC0D1']
CRC calculado:    0xC0D1
```

Validación del sistema de archivos en tarjeta microSD

Prueba 1

Esta prueba inicial valida la capacidad del sistema para detectar y montar correctamente la tarjeta microSD. El procedimiento es el siguiente:

1. STM32 invoca la función `microSD_init()`.
2. Se comprueba si la inicialización fue exitosa.
3. Se imprime por UART un mensaje informando el resultado del montaje.

```
[TEST1] SD montada con exito
```

Prueba 2

Se genera un archivo con nombre aleatorio (8 caracteres + extensión '.txt') y contenido aleatorio de 200 caracteres. El procedimiento es:

1. Se genera un nombre aleatorio y se concatena la extensión `.txt`.
2. Se genera una cadena aleatoria de 200 caracteres.
3. STM32 intenta crear el archivo mediante `microSD_put()`.
4. Si la operación fue exitosa, se imprime por UART el nombre y contenido generado.

Además, la Raspberry Pi registra tanto el nombre del archivo como su contenido en un archivo auxiliar local, con el fin de utilizarlos en la siguiente prueba de lectura. Esto permite verificar la integridad del almacenamiento y recuperar los datos esperados con precisión.

```
[TEST2] Archivo creado: hnEZyUg6.txt
[TEST2] Contenido generado: evZnJj6paqzoXDgsc3niW7Fb4AxI ...
   KbTU25EEixw0Dje5ALcr8jMJZepjV
```

Prueba 3

Se intenta leer el contenido del archivo creado previamente. El procedimiento es:

1. STM32 utiliza `microSD_get()` con el nombre almacenado.
2. Si la lectura es exitosa, se imprime por UART el nombre del archivo leído.

3. Luego se imprime el contenido leído.

La Raspberry Pi compara ambos datos con los registrados en el test anterior.

```
[TEST3] Archivo leido: hnEZyUg6.txt
[TEST3] Contenido leido: evZnJj6paqzoXDgsc3niW7Fb4AxI ...
   KbTU25EEixw0Dje5ALcr8jMJZepjV
```

Prueba 4

Se elimina el archivo previamente generado. El procedimiento es:

1. STM32 llama a `microSD_delete()` con el nombre almacenado.
2. Si la eliminación es exitosa, se imprime un mensaje por UART.
3. Luego se solicita un listado actualizado de archivos y se imprime también por UART.

La Raspberry Pi verifica que el archivo ya no figure en la lista.

```
[TEST4] Archivo eliminado correctamente:
[TEST4] Listado actual de archivos en SD:
```

Prueba 5

Se crean tres archivos con nombres y contenidos aleatorios, y se realiza el siguiente procedimiento:

1. STM32 genera tres nombres de archivo aleatorios y sus respectivos contenidos.
2. Se escriben en la microSD utilizando `microSD_put()`.
3. Se imprime un listado de los archivos presentes.
4. Luego se llama a `microSD_deleteAll()` para eliminarlos.
5. Finalmente se imprime nuevamente el listado para verificar que la tarjeta quedó vacía.

```
[TEST5] Archivos antes de borrar:
[UART] - 8IXpMwRE.txt
[UART] - RlN7EQDz.txt
[UART] - FWeNma2C.txt
[UART] [DEBUG] Intentando borrar: 8IXpMwRE.txt
[UART] [DEBUG] Intentando borrar: RlN7EQDz.txt
[UART] [DEBUG] Intentando borrar: FWeNma2C.txt
[TEST5] Archivos despues de borrar:
[TEST5] Verificacion exitosa: No hay archivos en la SD luego del borrado.
```

Validación de memoria FLASH NOR Winbond

A continuación se detallan las pruebas realizadas sobre la memoria NOR Flash Winbond W25Q32. Todas las interacciones se realizaron a través del bus SPI, y los resultados obtenidos se enviaron por UART para su validación en la Raspberry Pi. Cada test transmite un mensaje formateado desde la STM32, que luego es interpretado y verificado automáticamente por el entorno de pruebas.

Prueba 1

El procedimiento de prueba consiste en:

1. STM32 consulta el ID del fabricante y dispositivo a través de SPI.
2. El valor recibido es enviado por UART con el formato "[TEST1] Fabricante/Dispositivo: XXXX".
3. La Raspberry Pi valida si el ID coincide con el valor esperado.

```
[TEST1] Fabricante/Dispositivo: EF15
```

Prueba 2

El procedimiento de prueba consiste en:

1. STM32 solicita el JEDEC ID por SPI.
2. El valor recibido se imprime por UART en el formato "[TEST2] JEDEC ID: XXXXXX".
3. La Raspberry Pi compara el valor recibido con el esperado.

```
[TEST2] JEDEC ID: EF4016
```

Prueba 3

El procedimiento de prueba consiste en:

1. STM32 solicita el UID de la memoria por SPI.
2. El UID de 8 bytes se imprime por UART: "[TEST3] UID: ...".
3. La Raspberry Pi verifica si el UID es válido.

```
[TEST3] UID: C66208B5DB3A3026
```

Prueba 4

El procedimiento de prueba consiste en:

1. STM32 genera una cadena de datos aleatoria de longitud entre 200 y 399 bytes.
2. Se elige una dirección aleatoria y se escribe el contenido por SPI.
3. Se imprime por UART la dirección y la longitud.
4. Luego se imprime el contenido escrito.

```
[TEST4] Escrito en 0x311600 (276 bytes)
[TEST4] Contenido:im1kgL1khHBGqUd ... 1Djmas3UdydBG6Tf9
```

Prueba 5

El procedimiento de prueba consiste en:

1. STM32 lee desde la misma dirección utilizada en la prueba anterior.
2. Se imprime por UART la dirección y longitud.
3. Luego se imprime el contenido leído.
4. La Raspberry Pi compara ambos contenidos.

```
[TEST5] Leido desde 0x311600 (276 bytes)
[TEST5] Contenido leido: im1kgL1khHBGqUd ... 1Djmas3UdydBG6Tf9
```

Prueba 6

El procedimiento de prueba consiste en:

1. STM32 borra el sector donde se había almacenado la información previa.
2. Se imprime la dirección del sector procesado.
3. Luego se informa si el borrado fue exitoso.

```
[TEST6] OK: Sector 0x311000 borrado correctamente
```

Prueba 7

El procedimiento de prueba consiste en:

1. STM32 realiza cinco escrituras consecutivas en direcciones aleatorias, alineadas a páginas de 256 bytes.
2. Cada página escrita contiene datos pseudoaleatorios, y se imprime por UART la dirección utilizada con el formato: "[TEST7] Escrito en 0xXXXXXX".
3. Luego, STM32 solicita un borrado completo del chip.
4. Una vez finalizado el borrado, STM32 accede nuevamente a cada una de las cinco direcciones previamente escritas.
5. Se leen los 256 bytes de cada página, y se compara cada uno con el valor `0xFF`.
6. Si algún byte no coincide, se imprime un mensaje de error con la dirección, el índice del byte y su valor leído.
7. Si todas las direcciones están completamente borradas, se imprime por UART el mensaje final de éxito.

```
[TEST7] Escrito en 0x103800
[TEST7] Escrito en 0x1DE600
[TEST7] Escrito en 0x08C200
[TEST7] Escrito en 0x011600
[TEST7] Escrito en 0x3D1F00
[TEST7] OK: Todas las direcciones borradas correctamente
```

```
[TEST7] Escrito en 0x103800
[TEST7] Escrito en 0x1DE600
[TEST7] Escrito en 0x08C200
[TEST7] Escrito en 0x011600
[TEST7] Escrito en 0x3D1F00
[TEST7] OK: Todas las direcciones borradas correctamente
```

5.2. Validación automatizada e integración continua

La Figura 5.2 muestra una ejecución exitosa del *pipeline* de integración continua implementado sobre *hardware* real. Cada bloque representa una prueba independiente correspondiente a un subsistema del proyecto, y está compuesto por tres etapas: compilación del *firmware*, grabación sobre el microcontrolador, y ejecución de la prueba con recolección de resultados.

El estado verde en cada una de estas etapas indica que el proceso se completó correctamente, lo que garantiza tanto la validez de los binarios como el funcionamiento del sistema bajo prueba. Esta validación se realiza de forma totalmente automática desde la Raspberry Pi, tal como se describió en la sección 3.2.

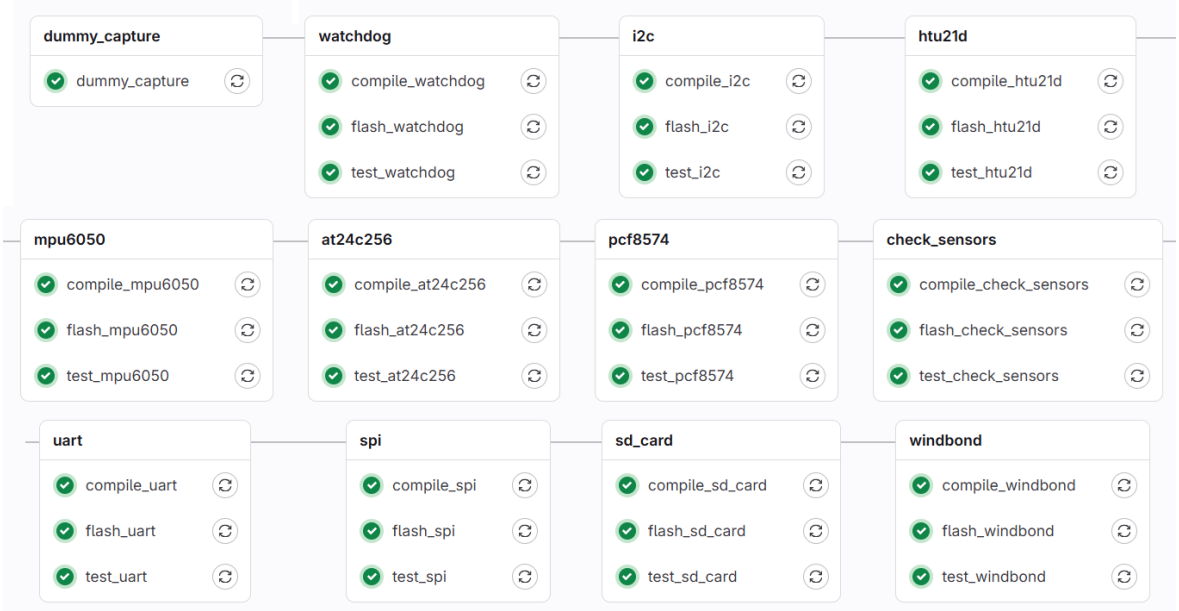

FIGURA 5.2. Ejecución completa del *pipeline*

5.3. Monitor de recursos

Como parte de los resultados obtenidos, se presenta la interfaz gráfica desarrollada para visualizar en tiempo real el estado del sistema embebido. La misma corre sobre la Raspberry Pi y permite monitorear variables clave como temperatura, humedad, aceleración, giroscopio y posición GPS, además de mostrar errores, mensajes UART y métricas internas del sistema.

Estos datos son generados por un conjunto de tareas concurrentes ejecutándose en la STM32, cada una dedicada a la adquisición, procesamiento y transmisión de información

desde distintos sensores. También se incluyen tareas encargadas del almacenamiento en tarjeta SD y del manejo de comandos de descarga de datos. La configuración detallada del *firmware*, incluyendo la asignación de memoria y el esquema de prioridades utilizado, se encuentra documentada en el Anexo J.

En la Figura 5.3 se muestra la parte superior del panel, donde se observan las variables ambientales y los gráficos correspondientes al acelerómetro y giroscopio. En estos últimos, se visualizan tanto los valores crudos como sus versiones corregidas, lo cual permite validar en tiempo real el procesamiento que realiza el *firmware*.

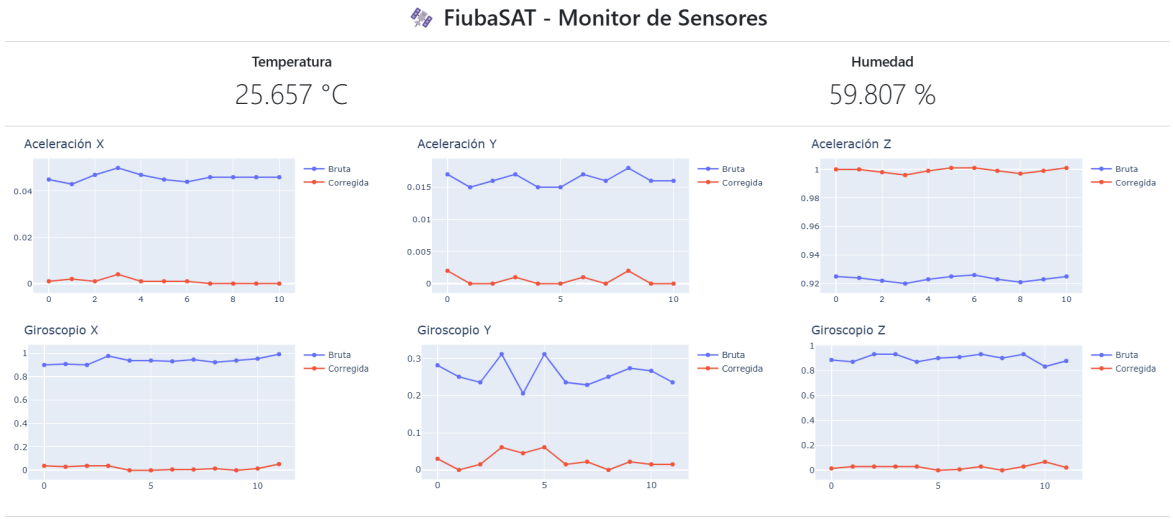

FIGURA 5.3. Encabezado del monitor junto a las mediciones de temperatura, humedad, aceleración y giro.

En el centro del panel se encuentra el mapa GPS (Figura 5.4), que representa la ubicación actual y la trayectoria recorrida. Para facilitar la interacción, se utilizaron datos simulados en tiempo real que permiten evaluar el sistema aun sin señal satelital.

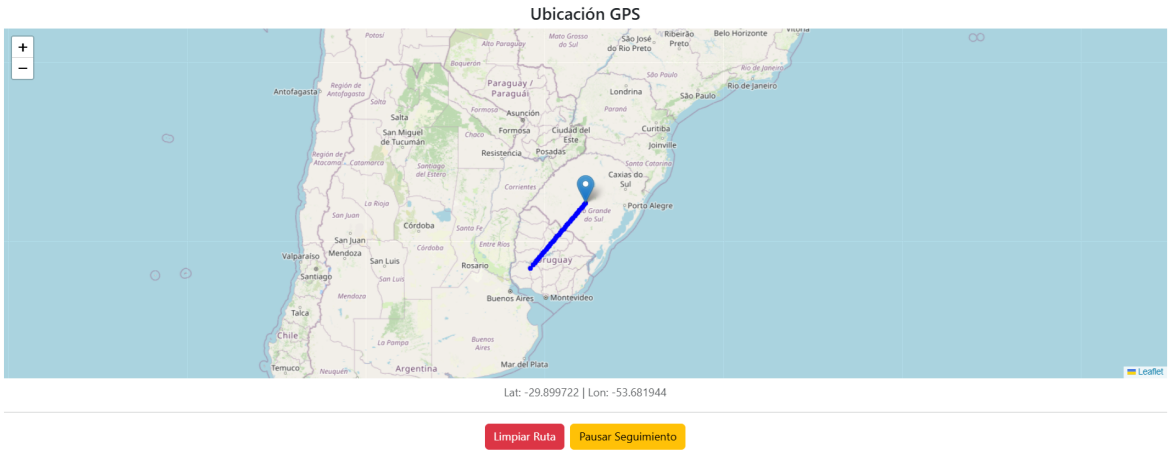

FIGURA 5.4. Mapa con los datos del GPS.

Las secciones inferiores permiten observar en tiempo real tanto los errores registrados por el sistema como la salida UART completa (Figura 5.5). Esto resulta fundamental para depurar el comportamiento de cada tarea y verificar la transmisión de datos entre la STM32 y la Raspberry Pi.

Finalmente, la interfaz cuenta con un módulo para solicitar y descargar archivos desde la tarjeta SD mediante comandos UART (Figura 5.6). Esto simula una interacción con la

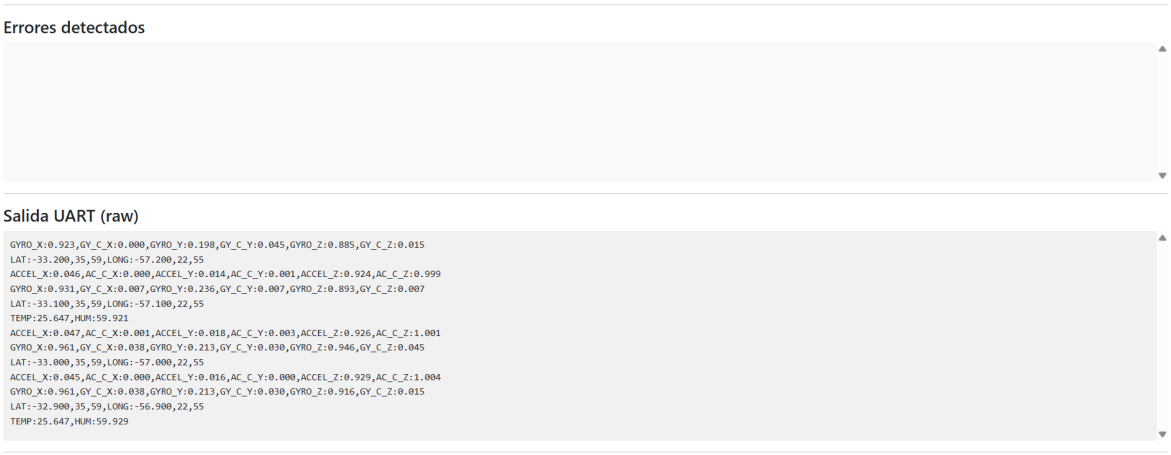

FIGURA 5.5. Errores detectados y salida UART cruda.

estación terrestre, en la que el usuario puede solicitar la transmisión de registros específicos almacenados a bordo del satélite. Los datos se presentan en pantalla y pueden exportarse como archivos CSV estructurados, listos para su análisis posterior.

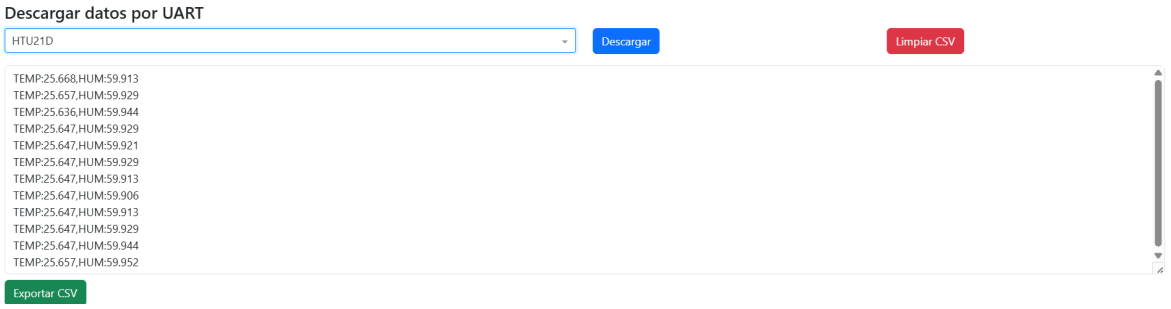

FIGURA 5.6. Descarga y previsualización de archivos desde la tarjeta SD.

5.4. Integración del *hardware*

En las primeras etapas del proyecto, cada integrante del equipo trabajó de forma independiente, desarrollando y probando sus respectivos módulos en entornos locales. Esto implicaba que el *hardware* utilizado para validar cada parte del sistema también estaba separado, montado sobre placas de desarrollo con cableado manual. Si bien esta estrategia fue adecuada durante las fases iniciales, resultó progresivamente ineficiente a medida que el proyecto avanzó y comenzó a requerir mayor integración entre los distintos componentes.

Con la incorporación de nuevos módulos de *hardware* y la necesidad de realizar pruebas combinadas que involucraban múltiples sensores y periféricos, se volvió evidente que las conexiones temporales mediante cables generaban problemas frecuentes. Entre estos se encontraban falsos contactos, interferencias o errores difíciles de reproducir, lo cual complicaba la depuración y hacía difícil distinguir si una falla era causada por el *software* desarrollado o por una conexión defectuosa.

Prototipo inicial

Durante esta etapa se trabajó con un prototipo armado de forma manual, empleando placas de prueba tipo *protoboard* y cables sueltos para conectar los distintos módulos. Esta

configuración permitió validar inicialmente la comunicación con los sensores y periféricos, pero presentó múltiples limitaciones en cuanto a estabilidad, organización y confiabilidad eléctrica. La figura 5.7 muestra una imagen de este primer prototipo, que sirvió como base funcional para las etapas iniciales de integración.

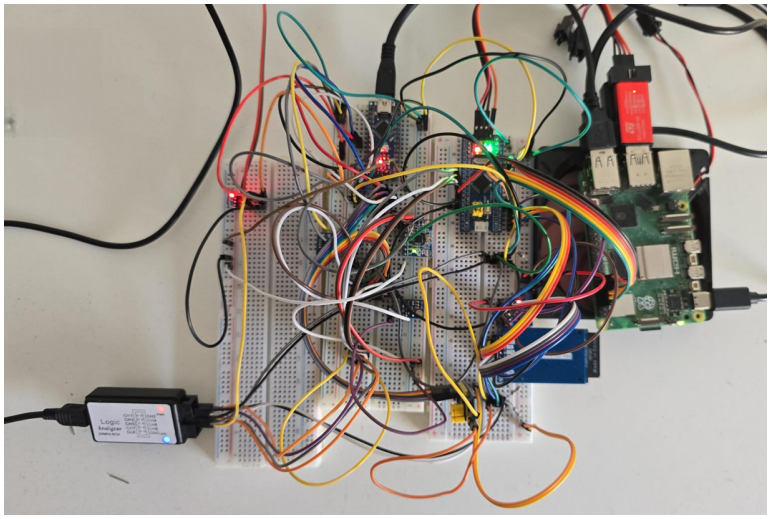

FIGURA 5.7. Prototipo inicial con cableado manual.

Frente a este escenario, y dado que el sistema final debía probar el funcionamiento conjunto de los tres protocolos de comunicación sobre un único microcontrolador, se volvió necesario consolidar todo el *hardware* de pruebas en una única plataforma. Para ello, se diseñó una placa de circuito impreso (PCB) que reúne todos los periféricos requeridos para las pruebas integrales.

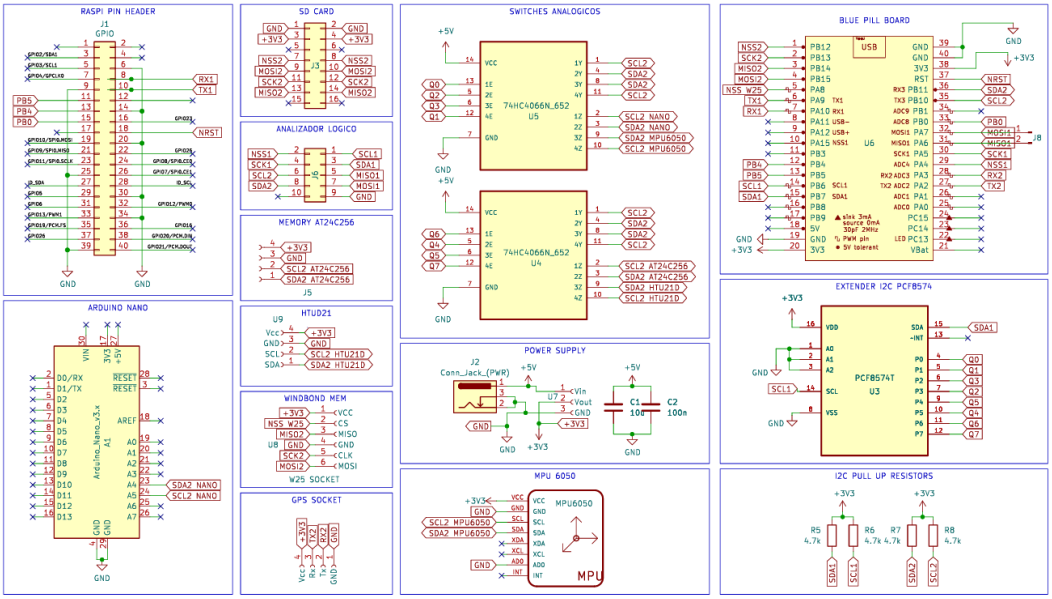

FIGURA 5.8. Esquemático de la placa de desarrollo.

La figura 5.9 muestra el diseño final correspondiente a una placa de doble faz con agujeros metalizados (PTH), que combina componentes de montaje superficial (SMD) y de inserción (DIP). Además, ciertos periféricos fueron montados en zócalos (*sockets*), permitiendo su

fácil reemplazo en caso de fallas o daños durante el proceso de pruebas. Esta solución mejora notablemente la confiabilidad del sistema durante el desarrollo y facilita la organización del entorno de pruebas.

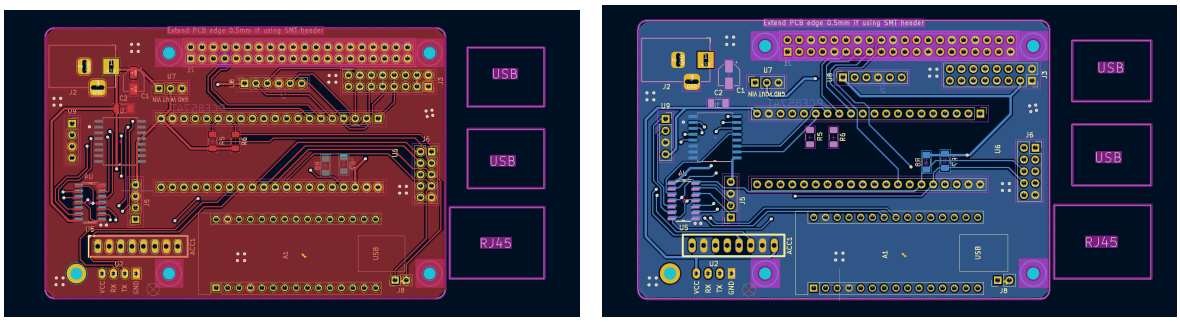

FIGURA 5.9. Vista superior e inferior de la PCB de desarrollo.

Dicho diseño fue pensado para colocarse sobre la Raspberry Pi a modo de HAT (del inglés *Hardware Attached on Top*), es decir, una placa de expansión que se conecta directamente sobre los pines GPIO de la Raspberry, respetando un formato físico y de pines estandarizado. Esta configuración permite prescindir de conexiones cableadas, reduciendo el desorden y mejorando la robustez del conjunto. Como se puede ver en la figura 5.10, la placa desarrollada se monta directamente sobre la Raspberry Pi, formando un único módulo compacto y funcional.

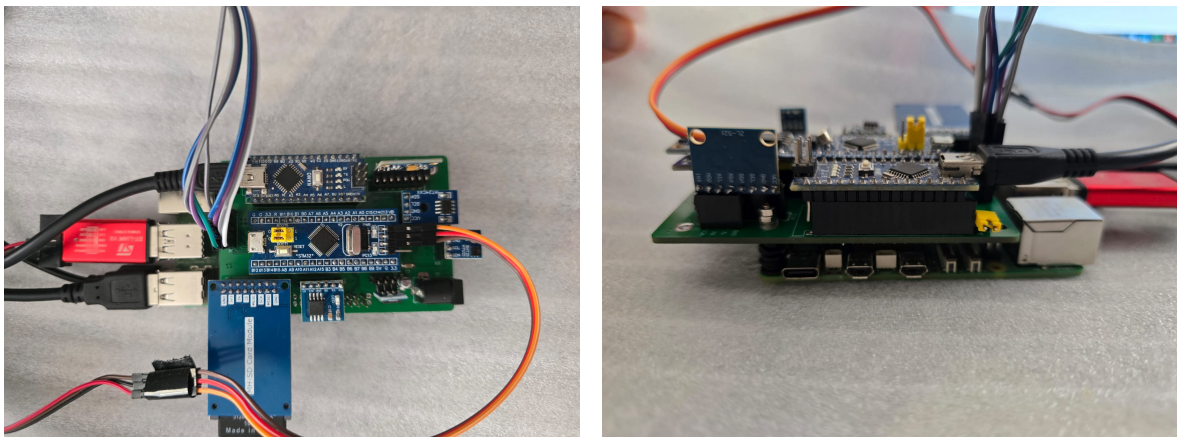

FIGURA 5.10. PCB montado sobre la Raspberry Pi.

Capítulo 6

Conclusiones

El desarrollo del proyecto representó un importante salto cualitativo respecto de nuestras experiencias, las cuales se habían centrado principalmente en el uso de microcontroladores de 8 bits y programación *bare-metal* en *assembly*. A lo largo del proceso, se adquirieron conocimientos sólidos sobre la arquitectura de microcontroladores de 32 bits y su programación en lenguaje C, además del funcionamiento interno de un sistema operativo en tiempo real. La incorporación de FreeRTOS requirió un aprendizaje progresivo para dominar funcionalidades como la planificación de tareas, la gestión del *heap*, el control de prioridades y la utilización de mecanismos de sincronización y exclusión mutua. Un aspecto especialmente desafiante fue la necesidad de diseñar el sistema considerando cuidadosamente los recursos limitados disponibles, tanto en términos de memoria SRAM como de cantidad de pines. Si bien la gestión eficiente de recursos ya había sido abordada en otros proyectos, el nivel de complejidad alcanzado en este caso fue considerablemente mayor.

En el ámbito de la comunicación entre módulos, se avanzó en la validación de los protocolos UART, SPI e I²C. Aunque su implementación fue directa en términos generales, la imposibilidad de utilizar la Raspberry Pi como dispositivo esclavo para estos protocolos, con el fin de validar los controladores implementados, condujo a una etapa de investigación orientada a encontrar soluciones alternativas. Se integró un analizador lógico para capturar las señales de los buses de comunicación, y se desarrollaron scripts en Python capaces de decodificar automáticamente las tramas capturadas. Los datos procesados se utilizaron dentro de un entorno de *testing* automatizado, lo que permitió validar el comportamiento de los protocolos de manera precisa y repetible. Estas prácticas facilitaron la incorporación de herramientas y metodologías previamente no exploradas, como el uso de `pytest` y la ejecución de procesos externos (lectura del puerto serie y del analizador lógico) dentro de los *scripts*.

Asimismo, se logró una estandarización robusta del entorno de desarrollo. Hasta ese momento, nuestra experiencia en trabajo colaborativo se limitaba al uso básico de Git para compartir código de manera informal. Sin embargo, para este proyecto en particular, fue necesario incorporar desde cero prácticas mucho más avanzadas y profesionalizadas. Mediante la creación de un *Dockerfile* y diversos scripts auxiliares, configuramos un entorno de desarrollo homogéneo y replicable, lo que garantizó que todos los integrantes del equipo trabajaran no solo con el mismo repositorio, sino también sobre un entorno de compilación idéntico. Esto redujo errores de configuración local y simplificó la integración del sistema.

Además, abordamos por primera vez el uso de *GitLab CI/CD*, herramienta con la que automatizamos por completo el flujo de trabajo. Configuramos una *Raspberry Pi* como servidor de integración continua, equipada con un *GitLab Runner*, lo que permitió descargar, compilar, grabar y validar el *firmware* directamente sobre el *hardware* físico ante cada cambio en el repositorio. Esta infraestructura no solo mejoró la eficiencia y la trazabilidad del desarrollo embebido, sino que representó un salto cualitativo poco frecuente en entornos académicos e incluso escasamente adoptado en algunos ámbitos industriales.

En resumen, el proyecto no solo alcanzó los objetivos técnicos inicialmente propuestos,

sino que también proporcionó una experiencia formativa integral. Se incorporaron tecnologías innovadoras, nuevas herramientas de desarrollo e integración continua, las cuales sientan las bases para su aplicación en futuros sistemas embebidos y para el crecimiento sostenido del proyecto ASTAR.

Trabajos futuros

Controladores de memoria

Los *drivers* desarrollados para los dispositivos de almacenamiento —como la EEPROM vía I²C, la memoria Flash W25Q32 y la tarjeta microSD mediante SPI— fueron diseñados con un enfoque centrado en la validación funcional de las comunicaciones, tanto para pruebas básicas como para la integración de módulos. No obstante, una mejora necesaria consiste en el desarrollo de controladores más avanzados que incluyan mecanismos internos de manejo de errores y estrategias de almacenamiento redundante, con el fin de preservar la integridad de los datos ante fallos durante las operaciones de escritura o lectura.

En el caso particular de la tarjeta microSD, sería deseable implementar funciones específicas que habiliten y aprovechen la verificación por CRC que ofrece el protocolo SPI, permitiendo así detectar corrupciones silenciosas en la transferencia de datos. Además, dado que se utiliza la biblioteca *FatFs*, la cual ofrece un sistema de archivos completo y altamente configurable, se podrían incorporar mecanismos seguros que restrinjan la apertura o lectura simultánea de archivos desde múltiples tareas. Esto evitaría condiciones de carrera o accesos no controlados que podrían derivar en la corrupción del sistema de archivos o la pérdida de información crítica.

Evaluación de migración a microcontroladores con mayores recursos

El microcontrolador STM32F103C8T6 utilizado en este desarrollo ha demostrado ser apto para la implementación de una OBC funcional, siempre que se realice una gestión cuidadosa de los recursos disponibles. Tal como se analizó en la sección correspondiente, una asignación eficiente del *heap* y del *stack* de cada tarea, junto con un uso controlado del *heap* global y de las estructuras compartidas, permite desplegar un sistema operativo en tiempo real completamente funcional sobre esta arquitectura.

Sin embargo, la viabilidad final del uso de este microcontrolador en misiones aeroespaciales estará fuertemente condicionada por la carga útil (*payload*) y los requisitos específicos asociados a cada misión. La incorporación de sensores adicionales, algoritmos complejos de procesamiento local, rutinas de compresión, esquemas de almacenamiento prolongado o mecanismos de redundancia podría llevar rápidamente a superar el límite de memoria RAM disponible.

En este contexto, una mejora futura razonable consiste en evaluar la migración hacia microcontroladores con mayores prestaciones, como los pertenecientes a las familias STM32F4 o STM32H7. Estos dispositivos ofrecen mayores capacidades de procesamiento, buses de datos más amplios, soporte nativo para operaciones en punto flotante, periféricos avanzados con DMA multicanal y una cantidad significativamente superior de memoria Flash y SRAM, manteniendo compatibilidad con la biblioteca `libopencm3` y facilitando la reutilización del *software* ya desarrollado.

Incorporación de DMA en los controladores de comunicación

Una mejora técnica significativa consiste en la incorporación del uso de DMA (del inglés, *Direct Memory Access*) en los tres controladores de comunicación del sistema. Esta funcionalidad permite realizar transferencias de datos entre la memoria y los periféricos sin intervención continua del procesador, reduciendo la carga sobre la CPU y mejorando la eficiencia general. La transferencia mediante DMA se inicia desde el periférico correspondiente y, una vez finalizada, puede notificarse al sistema a través de interrupciones, semáforos o eventos, integrándose de forma natural con la planificación basada en un sistema operativo en tiempo real.

En la implementación actual, las transferencias de datos por I²C y SPI se realizan utilizando técnicas basadas en *polling*, lo que implica una supervisión activa del estado del periférico por parte del procesador durante la comunicación. En el caso de UART, dada su naturaleza *full-duplex* y el carácter asincrónico de la recepción, se optó por un enfoque basado en interrupciones para capturar los bytes entrantes de forma inmediata.

En escenarios donde se recibe un flujo continuo de datos, la utilización de DMA puede mejorar sustancialmente la eficiencia del sistema. El mecanismo general opera de la siguiente manera:

- El procesador configura inicialmente el periférico y el controlador DMA para una transferencia determinada.
- A partir de ese punto, el controlador DMA se encarga de gestionar automáticamente el flujo de datos entre el periférico y la memoria RAM, sin requerir intervención del CPU durante la transferencia.
- Al finalizar la operación, se genera una interrupción para notificar al procesador, lo que permite iniciar el procesamiento del bloque completo de datos de forma inmediata.

No obstante, el uso de DMA también introduce ciertos compromisos que deben considerarse durante el diseño del sistema. La configuración inicial del controlador DMA conlleva una sobrecarga temporal y, en transferencias de tamaño reducido, esta preparación puede llegar a consumir más tiempo que el necesario con un enfoque tradicional basado en interrupciones o *polling*. Por ello, la decisión de utilizar DMA debe estar respaldada por un análisis de las características y exigencias específicas de cada aplicación.

Integración de herramientas de depuración avanzadas

Si bien durante el desarrollo se utilizó con éxito el método tradicional de depuración mediante el envío de mensajes por la interfaz UART, una posible mejora consiste en incorporar el uso de *debuggers* que permitan ejecutar el código paso a paso y observar el estado interno de registros y periféricos en tiempo real. Esta funcionalidad resulta especialmente útil en sistemas complejos donde es necesario analizar el comportamiento de tareas concurrentes y verificar condiciones internas no observables mediante salidas serie.

Sin embargo, el uso de herramientas de depuración en sistemas operativos en tiempo real como FreeRTOS presenta ciertos desafíos, ya que implica interpretar el comportamiento de múltiples contextos de ejecución y manejar eventos asincrónicos. Existen herramientas propietarias que facilitan esta integración, como STM32CubeIDE, que ofrece soporte nativo para FreeRTOS y visualización de tareas. La adopción de soluciones completamente libres y multiplataforma, por su parte, puede requerir una configuración manual más compleja. Aun así, su incorporación representaría un salto cualitativo en la capacidad de diagnóstico, mantenimiento y escalabilidad del sistema.

ANEXO

A. Dockerfile

Un Dockerfile debe comenzar con una instrucción `FROM`, que indica la imagen base sobre la cual se construirá la nueva imagen. Esta imagen base puede ser, por ejemplo, una distribución de Linux mínima o una imagen preconfigurada con ciertas herramientas. Antes de la instrucción `FROM` sólo pueden incluirse directivas `ARG`, las cuales permiten definir variables que luego pueden usarse dinámicamente en la línea `FROM` o en otras partes del archivo.

También es posible incluir líneas de comentario, que comienzan con el símbolo `#`. Estas líneas son ignoradas durante la ejecución, pero resultan útiles para documentar el propósito de cada sección o instrucción del archivo.

Entre las instrucciones más utilizadas en un Dockerfile se encuentran `RUN`, `CMD` y `ENTRYPOINT`. Estas instrucciones pueden escribirse en dos formatos: forma `exec` y forma `shell`. La forma `exec` utiliza una sintaxis similar a un arreglo JSON, donde cada elemento representa un comando, parámetro o argumento individual. Por ejemplo, `["ls", la"]`. Esta forma es especialmente útil para evitar problemas de interpretación de comandos y para asegurar una ejecución directa, sin pasar por el *shell* del sistema base. Esto reduce errores relacionados con el *parsing* de cadenas y es más predecible en su comportamiento.

En contraste, la forma *shell* se asemeja a la sintaxis tradicional de una terminal, como por ejemplo: `RUN apt update && apt install -y build-essential`. En este caso, el comando se ejecuta utilizando el intérprete de comandos por defecto del sistema base (por ejemplo, `/bin/sh` en distribuciones Linux). Esta forma resulta conveniente para escribir comandos complejos o multilínea, ya que permite el uso de caracteres de escape como la barra invertida (`\`) para dividir instrucciones largas en varias líneas. Sin embargo, en entornos como Windows, donde la barra invertida también se utiliza como separador de rutas, pueden surgir conflictos si no se escapan correctamente, especialmente en la forma `exec`.

La instrucción `RUN` es ampliamente utilizada para ejecutar comandos arbitrarios durante la fase de construcción de la imagen. Cada vez que se invoca una instrucción `RUN`, se crea una nueva capa en el sistema de archivos de la imagen. Estas capas son acumulativas y afectan tanto el tamaño final de la imagen como la eficiencia en su reconstrucción, ya que Docker puede reutilizar capas anteriores si no han cambiado.

Por otro lado, las instrucciones `CMD` y `ENTRYPOINT` están destinadas a definir el comportamiento por defecto del contenedor cuando se ejecuta. Aunque ambas sirven para establecer qué comando se debe ejecutar al iniciar el contenedor, se diferencian en su flexibilidad y propósito.

- `CMD` se utiliza para especificar los parámetros por defecto que se pasarán al contenedor si no se indica ningún comando adicional. Puede ser sobrescrito al momento de ejecutar el contenedor con un comando personalizado.

- `ENTRYPOINT`, en cambio, define el comando principal que siempre se ejecutará al iniciar el contenedor, incluso si se pasan argumentos adicionales. Se utiliza cuando se quiere que el contenedor actúe como un ejecutable específico.

Ambas instrucciones pueden combinarse para lograr comportamientos más flexibles. Por ejemplo, se puede usar `ENTRYPOINT` para definir un comando fijo (como un *script* principal), y `CMD` para proporcionar los argumentos por defecto, que pueden ser reemplazados en tiempo de ejecución.

Un ejemplo sería:

```
CMD ["bash", "-c", "source ~/.bashrc && /bin/bash"]
```

Esta línea establece que, al iniciar el contenedor sin comandos adicionales, se abrirá una terminal interactiva de `bash`. Antes de iniciar el intérprete, se ejecuta `source ~/.bashrc`, lo cual permite cargar variables de entorno, alias y configuraciones personalizadas definidas en ese archivo. Esto resulta útil para que cualquier entorno virtual de Python activado previamente o variables exportadas manualmente (por ejemplo, `PATH` o `PYTHONPATH`) queden disponibles automáticamente en cada nueva sesión del contenedor. De esta forma, el contenedor se comporta como un entorno de desarrollo interactivo ya inicializado.

Además de las instrucciones mencionadas anteriormente, existen otras directivas en el Dockerfile que ofrecen funcionalidades importantes para la construcción de imágenes más completas y flexibles. Una de ellas es la instrucción `ADD`, que permite copiar archivos y directorios desde el sistema del host al sistema de archivos del contenedor durante la construcción. Si bien en la mayoría de los casos se prefiere usar `COPY` por ser más explícita y predecible, `ADD` ofrece funcionalidades adicionales. Entre ellas se destaca la posibilidad de usar una dirección HTTP o SSH de un repositorio Git como origen. En ese caso, Docker clona automáticamente el repositorio al directorio de destino indicado dentro de la imagen. Esto puede ser útil cuando se desea incluir código directamente desde un repositorio remoto en una etapa temprana de la construcción. Por ejemplo:

```
ADD https://github.com/user/repo.git direccion
```

Este comando clonará el contenido del repositorio especificado en el directorio "direccion" del contenedor.

Por otra parte, Docker permite utilizar variables de entorno para parametrizar el comportamiento del Dockerfile y facilitar la configuración del contenedor. Las variables definidas con la instrucción `ENV` pueden ser referenciadas en otras instrucciones, como `RUN`, `WORKDIR`, `USER`, `VOLUME`, entre otras. Esto permite, por ejemplo, definir rutas, versiones de herramientas, puertos expuestos o configuraciones específicas sin tener que codificarlas de forma fija.

Un ejemplo simple sería:

```
ENV APP_HOME=/usr/src/app
WORKDIR $APP_HOME
COPY . $APP_HOME
```

En este caso, se define una variable de entorno llamada `APP_HOME` y se reutiliza en las instrucciones siguientes para establecer el directorio de trabajo (equivalente a hacer un `cd` dentro de un entorno Linux) y copiar archivos.

A.1. Dockerfile del proyecto

A continuación se presenta el análisis del Dockerfile diseñado específicamente para este proyecto.

La instrucción `FROM` define la imagen base a partir de la cual se construye el contenedor. En este caso se utiliza la versión 24.04 de Ubuntu, una distribución estable, ampliamente soportada y con acceso a repositorios actualizados, lo que facilita la instalación de herramientas de desarrollo.

```
FROM ubuntu:24.04
```

Luego, se establece el directorio de trabajo dentro del contenedor. Todas las instrucciones siguientes que operen con rutas relativas lo harán desde este directorio. Esto permite centralizar el código fuente, *scripts* y cualquier otro archivo relacionado con el proceso de compilación y pruebas.

```
WORKDIR /usr/src/
```

En el siguiente bloque se instalan todas las herramientas necesarias para el entorno de pruebas:

- `git`: para clonar repositorios y gestionar versiones de código si fuera necesario.
- `python3`, `pip` y `venv`: base del sistema de pruebas automatizadas escritas en Python.
- `usbutils`: para detección de dispositivos USB, útil en la verificación de conexiones a la placa.
- `openocd`: herramienta que permite la programación de la STM32 mediante interfaces como ST-Link.
- `make`: herramienta de automatización de compilaciones.
- `wget` y `xz-utils`: utilidades auxiliares para descargas y manejo de archivos comprimidos.
- `nano`: editor básico de texto, útil para inspecciones rápidas.
- `gcc-arm-none-eabi`: toolchain específico para compilar código para la arquitectura ARM Cortex-M, como la utilizada en la STM32.

```
RUN apt-get update && apt-get install -y
git
python3
python3-pip
python3-venv
usbutils
openocd
make
wget
xz-utils
nano
gcc-arm-none-eabi
&& apt-get clean && rm -rf /var/lib/apt/lists/*
```

CÓDIGO 1. Preparación del entorno

Al final del comando, se incluye una limpieza del sistema de paquetes para reducir el tamaño final de la imagen, eliminando los archivos temporales y el cache de `apt`.

A continuación, se crea un entorno virtual de Python en `/opt/venv`, aislado del sistema base. Dentro de ese entorno se instala `pytest`, el *framework* de pruebas utilizado en los *scripts* que verifican el comportamiento del firmware. El uso del flag `-no-cache-dir` evita que `pip` almacene archivos temporales innecesarios, lo que también contribuye a mantener la imagen liviana.

```
RUN python3 -m venv /opt/venv &&
/opt/venv/bin/pip install --no-cache-dir pytest
```

Llegando al final, modifica la variable de entorno `PATH` dentro del contenedor, agregando el entorno virtual creado al inicio del PATH. Esto permite invocar directamente herramientas instaladas dentro del entorno virtual (como `pytest`) sin necesidad de activar manualmente el entorno con cada uso.

```
ENV PATH="/opt/venv/bin:$PATH"
```

Por último, se define el comando por defecto que se ejecutará al iniciar el contenedor. En este caso, se abre una sesión interactiva de Bash, lo cual es útil para depurar o ejecutar *scripts* manualmente durante el desarrollo o pruebas.

```
CMD ["/bin/bash"]
```

Se evaluó la posibilidad de clonar el repositorio del proyecto directamente dentro del contenedor durante la construcción de la imagen —de hecho, versiones anteriores del Dockerfile incluían comandos como `ADD`. Sin embargo, esta estrategia fue descartada por varias razones técnicas.

En primer lugar, se buscó mantener una clara separación entre el entorno de desarrollo (la imagen Docker) y el contenido del proyecto (el *firmware*), lo cual permite reutilizar la misma imagen para probar distintas versiones del código sin necesidad de reconstruirla.

Además, el sistema de integración continua utilizado (GitLab CI) ya se encarga de clonar el repositorio antes de ejecutar cualquier instrucción del *pipeline*. Incluir el código también dentro del contenedor resultaría redundante y propenso a inconsistencias.

Por otra parte, clonar el código fuente en tiempo de construcción generaría imágenes que dependen del estado dinámico del repositorio remoto, dificultando la trazabilidad y la reproducibilidad del entorno. Mantener la imagen independiente del código garantiza que cualquier versión del *firmware* pueda evaluarse de forma controlada y consistente.

A.2. Construcción de la imagen

Antes de poder utilizar la imagen construida, es necesario contar con un mecanismo para compartirla de forma eficiente. Para ello, se recurre a **Docker Hub**, el registro de imágenes más utilizado dentro del ecosistema Docker.

Docker Hub es un servicio público de almacenamiento y distribución de imágenes Docker. Funciona como un registro centralizado que permite a los desarrolladores subir, versionar y compartir imágenes, tanto de forma privada como pública. Las imágenes almacenadas en Docker Hub pueden ser accedidas fácilmente desde cualquier máquina que tenga Docker instalado, sin necesidad de transferir archivos manualmente.

Cada imagen alojada en Docker Hub se identifica mediante un nombre en el formato:

`usuario/nombre_imagen:tag`

Donde `usuario` es el nombre del propietario, `nombre_imagen` es el identificador de la imagen, y `tag` corresponde a una etiqueta que suele utilizarse para versionar la imagen (por ejemplo, `latest` o `v1.0`).

Una vez definido el destino de la imagen en Docker Hub, se procede a su construcción local mediante el siguiente comando:

```
$  docker build -platform linux/arm64 -t usuario/nombre_imagen:tag
   ubicacion/del/dockerfile
```

De esta manera, se construye la imagen a partir del Dockerfile ubicado en el directorio especificado por `ubicacion/del/contexto` (comúnmente `.` si se encuentra en el directorio actual), e incluye:

- La opción `-platform linux/arm64`, que indica que la imagen debe construirse para la arquitectura `ARM64`, utilizada por dispositivos como la Raspberry Pi 5. Esto permite compilar la imagen desde una arquitectura diferente (por ejemplo, x86_64) sin pérdida de compatibilidad.
- La opción `-t usuario/nombre_imagen:tag` para nombrar la imagen de acuerdo al esquema requerido por Docker Hub. Esto permite luego subirla fácilmente al repositorio correspondiente.

Una vez construida la imagen localmente con el nombre y la etiqueta correspondientes, el siguiente paso consiste en subirla a Docker Hub para que pueda ser accedida desde otros dispositivos, como la Raspberry Pi o los runners de GitLab CI.

Este proceso se realiza mediante el siguiente comando:

```
$   docker push usuario/nombre_imagen:tag
```

Antes de ejecutar este comando, es necesario haber iniciado sesión en Docker Hub con el cliente de Docker local. Esto se realiza mediante:

```
$   docker login
```

A.3. Uso de imágenes Docker construidas

Una vez publicada la imagen Docker en Docker Hub (en este proyecto bajo el nombre `frans8/fiubasat:rpi_v3.0`), puede ser utilizada tanto localmente desde una computadora personal como en entornos automatizados como GitLab CI/CD.

Desde una computadora local

Para trabajar de forma interactiva desde una computadora personal, se recomienda utilizar **Docker Desktop** junto con **Visual Studio Code**. Este enfoque permite abrir la imagen como si fuera una máquina de desarrollo completa, accediendo directamente a sus herramientas y entorno.

Visual Studio Code puede conectarse directamente a un contenedor Docker mediante la extensión **Remote - Containers**. Los pasos para hacerlo son:

1. Instalar Docker Desktop y la extensión `Remote - Containers` en VS Code.
2. Crear un contenedor persistente con nombre:

   ```
   $   docker run -it -privileged  -v $(pwd):/FiubaSAT -v /dev/bus/usb
      :/dev/bus/usb -name fiubasat frans8/fiubasat:rpi_v3.0
   ```

   Donde:

   - `docker run`: ejecuta un contenedor nuevo a partir de una imagen
   - `-it`: mantiene la entrada estándar abierta y asigna una terminal virtual, habilitando la interacción con el contenedor

- `-privileged`: otorgar permisos extendidos, incluyendo acceso a dispositivos USB, necesarios para utilizar el ST-LINK
- `-v $(pwd):/FiubaSAT`: monta el directorio actual dentro del contenedor en la ruta `FiubaSAT`
- `-v /dev/bus/usb:/dev/bus/usb`: monta los dispositivos USB del *host* dentro del contenedor
- `-name fiubasat`: le asigna un nombre personalizado al contenedo y
- `frans8/fiubasat:rpi_v3.0`: indica el nombre y versión de la imagen utilizada como base del contenedor

Si el contenedor ya fue creado, basta con ejecutar

```
$   docker start -ai fiubasat
```

o el nombre que se le haya asignado durante la construcción. Se puede omitir el parámetro `-ai` (modo interactivo) si se va a trabajar con Visual Studio Code.

3. En Visual Studio Code, abrir el menú de comandos (`Ctrl+Shift+P`) y seleccionar: `Remote-Containers: Attach to Running Container`.
4. Elegir el contenedor `fiubasat` para iniciar una sesión de desarrollo.

Una vez conectada, la experiencia es similar a trabajar sobre una máquina local: se puede editar el código, compilar el *firmware*, ejecutar *scripts* de prueba y acceder a los dispositivos conectados, todo desde el entorno preconfigurado de la imagen Docker.

Desde línea de comandos en integración continua

Hasta este punto, se explicó cómo trabajar con Docker desde un IDE que permite navegar por los archivos contenidos en el contenedor como si se estuviera trabajando directamente en el sistema de archivos local. Sin embargo, en las etapas avanzadas del desarrollo se dejó de utilizar esta modalidad, dando lugar a la ejecución de tareas (como la compilación y la grabación) directamente dentro del contenedor, pero sin interactuar con él.

Esto significa que, en la misma línea de comandos donde se lanza el contenedor, se indica también la acción que debe realizar. Por ejemplo:

```
$   docker run -rm -privileged -rm DIRECTORIO_DEL_PROYECTO:/FiubaSAT -rm
   /dev/bus/usb:/dev/bus/usb frans8/fiubasat:rpi_v3.0 /bin/bash -c "
   ACCION_1 && ACCION_2 ..."
```

Este comando es similar al utilizado anteriormente, pero ya no requiere iniciar el contenedor en modo interactivo. En su lugar, se indica directamente la orden a ejecutar dentro de la terminal `bash`.

Cuando se trabaja localmente (como en el primer caso descrito), es conveniente crear contenedores persistentes, evitando el uso del flag `-rm`, ya que esto permite:

- Mantener entornos virtuales o configuraciones internas.
- Reutilizar el contenedor en sesiones futuras sin necesidad de reinstalar herramientas.
- Conectarse desde Visual Studio Code sin perder el contexto entre sesiones.

En cambio, en entornos de integración continua como GitLab CI, es buena práctica utilizar contenedores efímeros, ya que:

- El repositorio se clona automáticamente al inicio del pipeline.
- No es necesario conservar datos dentro del contenedor una vez finalizado el trabajo.
- Cada ejecución parte desde un entorno limpio, garantizando reproducibilidad.

Por esta razón, en este contexto se utiliza el flag `-rm`.
A modo de ejemplo, la instrucción incluida en el archivo `.gitlab-ci.yml` para compilar un módulo es:

```
$   sudo docker run -rm -privileged -v $CI_PROJECT_DIR:/repo -v /dev/bus/
    usb:/dev/bus/usb frans8/fiubasat:rpi_v3.0 /bin/bash -c "cd /repo/test/
    ${MODULE_PATH} && make debug"
```

Donde:

- `$CI_PROJECT_DIR` es la ruta donde GitLab clona el repositorio.
- `-rm` asegura que el contenedor se elimine automáticamente al finalizar.
- `${MODULE_PATH}` indica dinámicamente el módulo de pruebas a compilar.

Este enfoque garantiza un entorno de compilación limpio, reproducible y consistente con la imagen definida, lo cual es clave para mantener la estabilidad del *pipeline* de pruebas automatizadas.

B. Gitlab runner

B.1. Preparativos

En primer lugar, se deben instalar algunas dependencias necesarias para la instalación segura de paquetes y la comunicación con el servidor:

```
$   sudo apt-get install curl openssh-server ca-certificates apt-
    transport-https perl
```

```
$   curl https://packages.gitlab.com/gpg.key | sudo tee /etc/apt/trusted.
    gpg.d/gitlab.asc
```

Estos paquetes permiten:

- `curl`: realizar solicitudes HTTP. Se utiliza para descargar claves, *scripts* y archivos necesarios desde internet, incluyendo el repositorio de GitLab.
- `openssh-server`: habilita el servicio SSH, necesario para permitir conexiones remotas seguras al sistema.
- `ca-certificates`: proporciona certificados raíz confiables, necesarios para establecer conexiones HTTPS seguras con servidores externos (como los de GitLab).
- `apt-transport-https`: habilita al sistema para descargar paquetes desde repositorios seguros a través del protocolo HTTPS, fundamental para obtener el GitLab Runner desde los repositorios oficiales.
- `perl`: lenguaje de programación utilizado por algunos *scripts* del sistema, requerido por ciertos instaladores o herramientas de automatización.

Una vez instaladas estas dependencias, se agrega la clave pública del repositorio oficial de GitLab, que permite verificar la autenticidad de los paquetes descargados.

B.2. Instalación y registro

El siguiente paso es instalar y registrar el *GitLab Runner* en la Raspberry Pi 5. Para esto, primero es necesario agregar al sistema el repositorio oficial de GitLab compatible con la arquitectura ARM, de forma que el paquete pueda descargarse e instalarse correctamente desde fuentes confiables [19]. Esto se realiza con el siguiente comando:

```
$   sudo curl -sS https://packages.gitlab.com/install/repositories/gitlab
   /raspberry-pi2/script.deb.sh | sudo bash
```

Este comando descarga y ejecuta un *script* de configuración provisto por GitLab. El *script* se encarga de detectar la distribución y arquitectura del sistema, agregar el repositorio oficial de GitLab correspondiente a Raspberry Pi, importar la clave GPG del repositorio, para verificar la autenticidad de los paquetes y actualizar la base de datos de paquetes disponibles (`apt`).

Una vez finalizado este paso, el sistema queda preparado para instalar el *runner* desde el repositorio recién agregado. La instalación se realiza con:

```
$   sudo apt-get install gitlab-runner
```

A continuación, se debe registrar el *runner*. En este proyecto, el registro del *GitLab Runner* se realizó inicialmente mediante el método tradicional, utilizando un ***runner registration token***. Este *token*, generado por GitLab en la interfaz del proyecto, permite vincular el *runner* con el repositorio. Este método fue funcional y permitió registrar el *runner* correctamente. Sin embargo, GitLab ha marcado este procedimiento como obsoleto a partir de la versión 15.6, y ha anunciado su futura eliminación en la versión 20.0 [43]. Por este motivo, a continuación se detalla el procedimiento recomendado actualmente, basado en un ***runner authentication token***[44].

1. En un navegador, ingresar a la página del repositorio de Gitlab
2. En el menú lateral *Project*, seleccionar *Settings* y, luego, **CI/CD** (figura 1)
3. Una vez en la página de *CI/CD Settings*, desplegar el menú *Runners*
4. En *Project runners*, seleccionar *New project runner* (figura 2)
5. Completar el campo *tag* (por ejemplo, con `rpi`; si es más de una, separadas por coma), la descripción (opcional) y el tiempo de espera máximo del *runner* (o si no se ingresa nada se toma el valor por defecto)
6. Seleccionar *create runner*
7. Seleccionar Linux como plataforma y copiar el comando provisto por Gitlab. El mismo tiene la forma:

   ```
   $   gitlab-runner register  -url https://gitlab.com  -token glrt-
      XXXX...XXXX
   ```

8. Ejecutarlo en la terminal de la Raspberry Pi

Es posible que el registro solicite completar algunos campos. Puede presionarse *enter* y dejarlos en sus valores por defecto.

Las etiquetas (*tags*) permiten asociar trabajos específicos del *pipeline* a este dispositivo en particular. Funcionan como un mecanismo de filtrado: cada *job* en el archivo `.gitlab-`

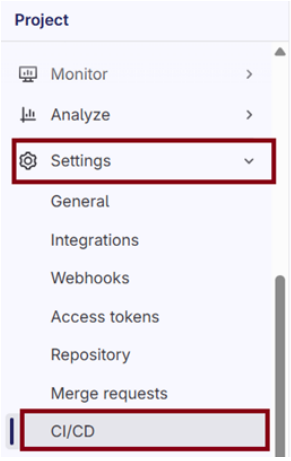

FIGURA 1. Paso 1 del registro del *runner*

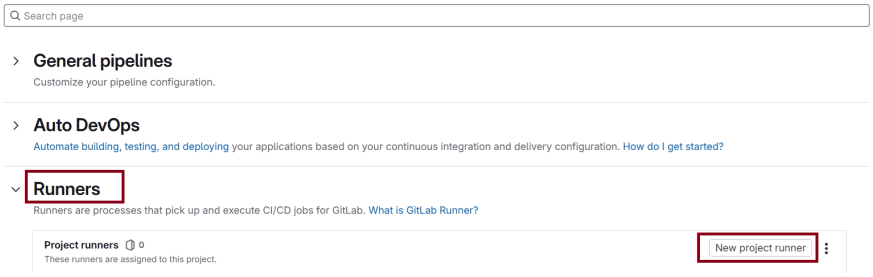

FIGURA 2. Paso 2 del registro del *runner*

`ci.yml` puede especificar qué *runners* están habilitados para ejecutarlo, a través del campo *tags*.

Una vez finalizado, se puede realizar una verificación con el comando

```
$   sudo gitlab-runner status
```

Esto debería mostrar el siguiente mensaje:

```
>>    gitlab-runner: Service is running!
```

C. YAML

El archivo `.gitlab-ci.yml` [45] es el componente central de cualquier *pipeline* de integración continua en GitLab. Define, mediante una sintaxis YAML estructurada, los pasos que deben ejecutarse automáticamente al producirse ciertos eventos en el repositorio, como por ejemplo un `push` o un `merge request`.

Este archivo se ubica en la raíz del proyecto y es interpretado por el *GitLab Runner*, que ejecuta las instrucciones definidas en los distintos bloques del *pipeline*.

El archivo se compone de una o más etapas (llamadas `jobs`), agrupadas opcionalmente en fases (`stages`). Cada *job* puede contener configuraciones específicas como el nombre de la imagen de Docker a usar, variables de entorno, comandos a ejecutar, condiciones de ejecución, entre otros.

Un esquema básico es:

```
    stages:
      - build
```

```
    - test

compilar:
  stage: build
  script:
    - make all

probar:
  stage: test
  script:
    - pytest test1.py
```

Algunos componentes claves del archivo YAML son:

1. `stages`: define el orden general del *pipeline*. Los *jobs* dentro de cada etapa se ejecutan en paralelo, pero una etapa no comienza hasta que la anterior se completa.
2. `script`: lista de comandos a ejecutar dentro de un job. Es el corazón de la tarea.
3. `tags`: permite asociar *jobs* a *runners* específicos. En este proyecto, se utilizan para asegurar que los trabajos se ejecuten únicamente en la Raspberry Pi 5.
4. `only` / `rules`: definen en qué condiciones debe ejecutarse un *job* (por ejemplo, sólo en la rama `main`, ante un `merge request` o ante la definición de cierta variable, como se verá más adelante).

Archivo de pruebas del proyecto

El archivo `.gitlab-ci.yml` está diseñado para ejecutarse en una Raspberry Pi 5 configurada como *runner*, y contempla etapas de compilación, grabación y validación para cada módulo funcional del sistema.

El archivo comienza con la definición de las distintas `stages`, que representan las etapas generales del *pipeline*. Estas incluyen:

```
stages:
  - dummy_capture
  - watchdog
  - i2c
  - htu21d
  - mpu6050
  - at24c256
  - pcf8574
  - check_sensors
  - uart
  - spi
```

Uno de los aspectos más potentes del *pipeline* es la posibilidad de seleccionar qué módulos se desean probar, sin necesidad de editar el archivo ni utilizar *tags* en Git. Esto se logra mediante una variable llamada `TEST_MODULE`, que puede ser definida en el momento de ejecutar el *pipeline* (si no se ingresa nada, mantiene su valor por defecto `all`).

```
variables:
    TEST_MODULE: "all"
```

- Para correr todas las pruebas:

```
$   git push
```

- Para seleccionar una prueba en particular (supongamos I^2C):

```
$   git push -o ci.variables="TEST_MODULE=i2c"
```

- Para seleccionar más de una prueba (por ejemplo, I^2C y SPI):

```
$   git push -o ci.variables="TEST_MODULE=i2c,spi"
```

Para evitar la repetición de lógica, se definieron tres plantillas que son heredadas por todos los módulos:

- `.compile_template`: compila el firmware en un contenedor Docker y genera un archivo binario como *artifact* (archivo generado durante un *job* y que se desea preservar y reutilizar al finalizar ese *job* (en la etapa de grabación).
- `.flash_template`: graba el binario sobre la STM32 utilizando el ST-LINK.
- `.test_template`: ejecuta pruebas en Python usando pytest, capturando resultados por UART o mediante el analizador lógico.

```
.compile_template:
  script:
    - sudo docker run --rm --privileged -v $CI_PROJECT_DIR/:/repo -v
/dev/bus/usb:/dev/bus/usb frans8/fiubasat:rpi_v3.0 /bin/bash -c "cd /
repo/test/${MODULE_PATH} && make debug"
    - sudo chown -R gitlab-runner:gitlab-runner $CI_PROJECT_DIR/test/
${MODULE_PATH}/*.bin
    - ls -l $CI_PROJECT_DIR/test/${MODULE_PATH}
  tags:
    - rpi
  artifacts:
    paths:
      - $CI_PROJECT_DIR/test/${MODULE_PATH}/fiubasat_test.bin
  allow_failure: true

.flash_template:
  script:
    - echo "Verificando artifacts descargados:"
    - ls -l $CI_PROJECT_DIR/test/${MODULE_PATH}
    - sudo docker run --rm --privileged -v $CI_PROJECT_DIR/:/repo -v
/dev/bus/usb:/dev/bus/usb frans8/fiubasat:rpi_v3.0 /bin/bash -c "cd /
repo/test/${MODULE_PATH} && make flash"
  tags:
    - rpi
  needs:
    - compile_${MODULE_PATH}
  artifacts:
    paths:
      - $CI_PROJECT_DIR/test/${MODULE_PATH}/fiubasat_test.bin
  allow_failure: true
```

```
.test_template:
  script:
    - whoami
    - hostname
    - sudo apt-get update
    - sudo apt-get install -y python3 python3-pip python3-venv git
i2c-tools python3-rpi.gpio
    - python3 -m venv venv
    - source venv/bin/activate
    - pip3 install --upgrade pip
    - pip3 install -r requirements.txt
    - ls -l test_ci_cd/${MODULE_PATH}
    - for test_file in test_ci_cd/${MODULE_PATH}/test*.py; do
        pytest -s $test_file;
      done
  tags:
    - rpi
  needs:
    - flash_${MODULE_PATH}
  allow_failure: true
```

CÓDIGO 2. Plantillas utilizadas en el YML

Estas plantillas incluyen `allow_failure: true` para evitar que un fallo en un *job* detenga la ejecución de los siguientes módulos (si la del *stage* actual debido a la dependencia entre *jobs*).

Luego, se detalla cada *stage* en particular, extendiendo las plantillas previamente definidas. Por ejemplo, en el caso del *test_watchdog*:

```
compile_watchdog:
  extends: .compile_template
  stage: watchdog
  variables:
    MODULE_PATH: test_watchdog
  rules:
    - if: '$TEST_MODULE =~ /(^|,)watchdog(,|$)|all/'
      when: always
    - when: never

flash_watchdog:
  extends: .flash_template
  stage: watchdog
  variables:
    MODULE_PATH: test_watchdog
  needs:
    - compile_watchdog
  rules:
    - if: '$TEST_MODULE =~ /(^|,)watchdog(,|$)|all/'
      when: always
    - when: never

test_watchdog:
  extends: .test_template
  stage: watchdog
  variables:
    MODULE_PATH: test_watchdog
  needs:
```

```
    - flash_watchdog
  rules:
    - if: '$TEST_MODULE =~ /(^|,)watchdog(,|$)|all/'
      when: always
    - when: never
```

donde:

- `extends`: permite heredar las instrucciones comunes desde una plantilla definida previamente (por ejemplo, `.compile_template`).
- `stage`: asocia el *job* con una de las etapas definidas en el bloque `stages:`. Los *jobs* dentro de la misma etapa se pueden ejecutar en paralelo, pero una etapa no comienza hasta que la anterior termina completamente.
- `variables`: define variables locales al *job*. En este caso, se usa `MODULE_PATH` para parametrizar el nombre del directorio del módulo (tanto para compilar como para los *tests*).
- `rules`: controla si un *job* debe ejecutarse o no, dependiendo de condiciones definidas. Esto no se puede generalizar en las plantillas (usando, por ejemplo, `MODULE_PATH`) debido a que `rules` se evalua antes de que se expandan las variables del *job*.
- `if: '$TEST_MODULE =~ /(^|,)watchdog(,|$)|all/'`: se cumple si la variable `TEST_MODULE` contiene exactamente `watchdog`, incluso si está dentro de una lista como `watchdog,spi`, o si su valor es `all`.

El patrón dentro del condicional establece:

- `=~`: operador de coincidencia con una expresión regular
- `/ ... /`: delimita la expresión regular
- `(^|,)`: significa que antes del valor a detectar está el inicio de la cadena o hay una coma
- `watchdog`: valor a detectar
- `(,|$)`: significa que luego del valor a detectar hay una coma o el final de la cadena

Por último, la directiva `when: always` le indica al *job* que debe ejecutarse siempre que se cumpla la condición definida en la cláusula `if`. En caso contrario, `when: never` evita su ejecución. Este patrón combinado de `always` y `never` garantiza que el *job* se ejecute únicamente si se cumple la condición especificada, evitando ejecuciones por defecto no deseadas. En nuestro caso particular, si no se define la variable `TEST_MODULE` de forma explícita al lanzar el pipeline, se evalúa utilizando el valor por defecto establecido al comienzo del archivo `.gitlab-ci.yml`; sin embargo, la condición se verifica en todos los casos.

D. Herramientas

D.1. Raspberry Pi 5

Para comenzar a utilizar desde cero una Raspberry Pi, es necesario instalarle un sistema operativo en una tarjeta micro SD, que luego será colocada en el equipo. La forma más sencilla es descargar el *software* **Raspberry Pi Imager** desde el centro de *software* de Raspberry. Los pasos a seguir se detallan a continuación.

1. Descargar e instalar *Raspberry Pi Imager*.
2. Conectar la tarjeta micro SD a la computadora.
3. Abrir el *Raspberry Pi Imager* y elegir el dispositivo, el sistema operativo y la unidad de almacenamiento (tarjeta micro SD) (figura 3) [46]. A los efectos de este trabajo, el dispositivo fue la Raspberry Pi 5 y el sistema operativo el Raspberry Pi OS de 64 bits.

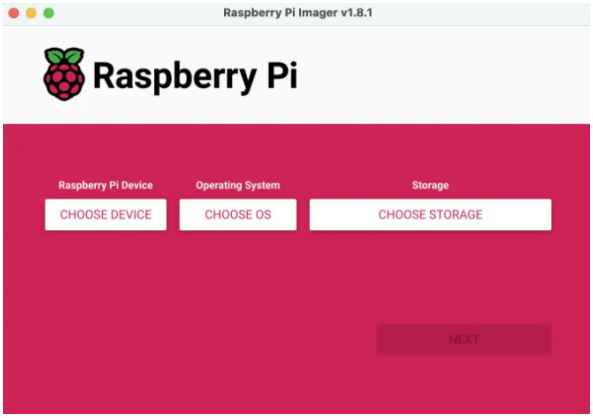

FIGURA 3. Captura de pantalla de *Raspberry Pi Imager*

Una vez finalizado, colocar la tarjeta micro SD en la Raspberry Pi. En este punto, el equipo ya está listo para ser utilizado.

D.2. Visual Studio Code

El siguiente procedimiento describe cómo habilitar el acceso remoto por SSH a la Raspberry Pi y conectarse desde VSCode [47]:

1. En la terminal de la Raspberry Pi, ejecutar el comando `sudo raspi-config`.
2. Navegar a la opción `Interface Options`.
3. Seleccionar `SSH` y habilitar el servicio.
4. En la terminal, ejecutar `whoami` para conocer el nombre de usuario actual, y `hostname -I` para obtener la dirección IP local (usualmente comienza con `192.`).
5. En la máquina local, abrir `Visual Studio Code`.
6. Instalar la extensión **Remote Development** (que incluye *Remote -SSH*).
7. Abrir la paleta de comandos con `Ctrl + Shift + P` y seleccionar la opción `Remote-SSH: Connect current window to host`.
8. Ingresar los datos de conexión en el formato `usuario@ip`.
9. Confirmar y guardar la configuración para facilitar futuras conexiones.

Este enfoque permite trabajar de forma remota y eficiente sobre la Raspberry Pi, con acceso completo a su sistema de archivos, terminal y entorno de desarrollo, directamente desde la interfaz de VSCode. La única limitación es que los usuarios deben encontrarse dentro de una misma red, pero se puede resolver utilizando una VPN (ver sección 3.3.2)

D.3. Tailscale

Para configurar una nueva red (propietario), se deben seguir los siguientes pasos:

1. Ingresar a la página de Tailscale
2. Seleccionar *Get Started*
3. Registrarse e iniciar la descarga del servicio
4. Se redirigirá a la página *Next, add a second device* (figura 4)
5. Seleccionar Linux y copiar el comando provisto
6. En la Raspberry Pi, ejecutar el comando anterior para instalar el programa
7. Iniciar el servicio con `sudo tailscale up`
8. Para ver la IP asignada, ejectuar `tailscale ip -4`

Esta última IP será la utilizada para la conexión SSH con Visual Studio Code (sección 3.3.1).

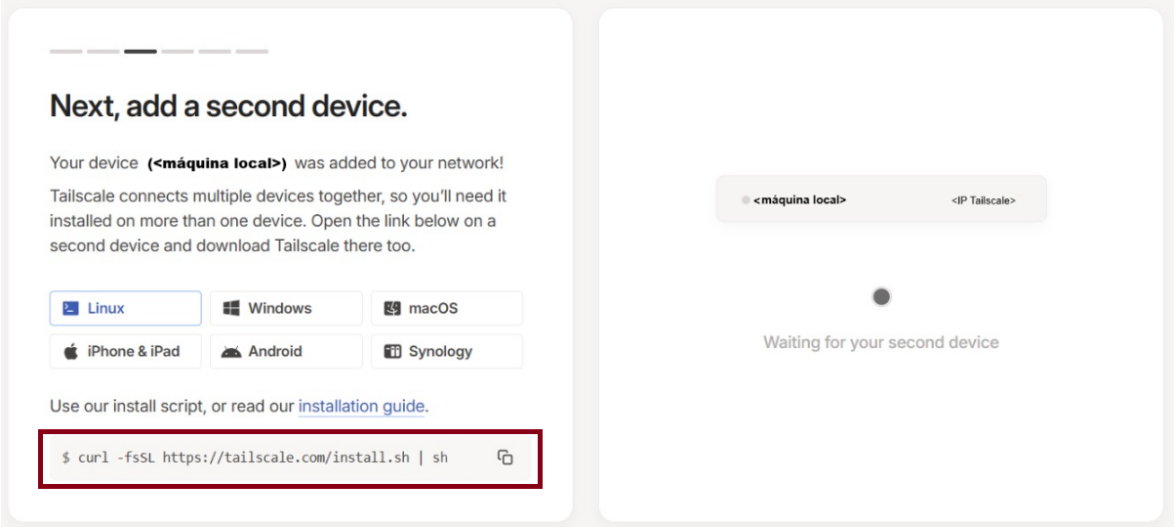

FIGURA 4. Ventana *Add second device* en la página de Tailscale

Para unirse a la nueva red (invitados), se deben seguir los siguientes pasos:

1. Desde la aplicación en la máquina local, desplegar el menú de usuario y seleccionar *Admin console...* (figura 5). También se puede ingresar desde la página de Tailscale.
2. En la ventana que se abrirá en el navegador, ingresar a la pestaña *Users*
3. Seleccionar *Invite external users* y completar la dirección de correo electrónico de la persona a invitar a la red
4. Elegir el rol (*member*, por ejemplo) y seleccionar *invite*
5. El invitado debe abrir el correo electrónico que se le envió y seguir los pasos indicados por el instalador de Tailscale

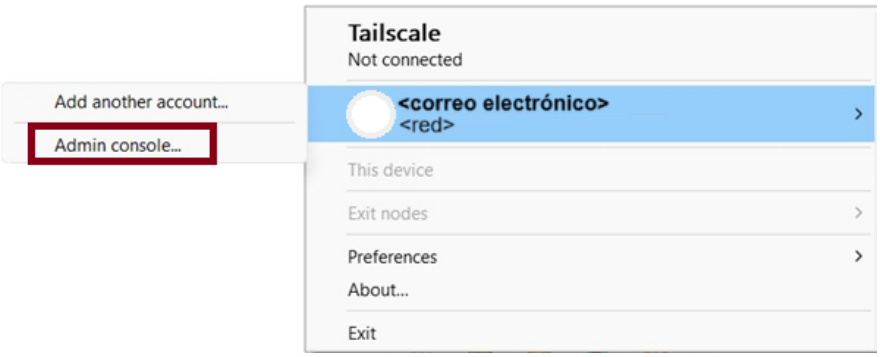

FIGURA 5. Opción *Admin console...* en Tailscale

Una vez completado el proceso de invitación, el nuevo usuario debería integrarse automáticamente a la red de Tailscale. Sin embargo, en algunos casos —especialmente si el usuario ya tenía Tailscale instalado antes de recibir la invitación— es posible que el sistema lo incorpore inicialmente a una red propia y no a la del anfitrión. Ante esta situación, se recomienda cerrar sesión desde la aplicación (accediendo al menú de usuario) o desde la interfaz web de Tailscale, y luego volver a iniciar sesión. En el segundo ingreso, se solicitará al usuario que seleccione a qué red desea unirse, y deberá elegir el correo electrónico correspondiente al anfitrión.

Por defecto, todos los miembros de una red Tailscale tienen acceso a todos los dispositivos conectados. Esta política puede ajustarse desde la *Admin Console*, específicamente en la sección *Access Control*. Dentro del bloque `"acls": [...]`, se puede modificar la línea:

```
  "action": "accept", "src": [""], "dst": [":*"]
```

por una versión más restrictiva, por ejemplo:

```
  "action": "accept", "src": [""], "dst": ["<ip de Tailscale de la RPi
>:"]
```

Esta configuración limita el acceso de todos los usuarios exclusivamente a la Raspberry Pi, permitiendo conexiones a cualquiera de sus puertos, pero impidiendo el acceso a otros dispositivos de la red.

E. Compilación

La construcción del *firmware* embebido para el microcontrolador STM32F103C8T6 se gestiona mediante un `Makefile`. Este archivo permite automatizar la compilación, enlazado, conversión de binarios y flasheo del sistema. Su diseño modular está orientado a facilitar la reutilización de código base y la adaptación rápida entre distintos módulos de prueba, algo fundamental en el entorno de desarrollo iterativo de CubeSATs.

E.1. Variables generales

Las primeras líneas del `Makefile` definen variables globales necesarias para parametrizar rutas, herramientas de compilación y nombres de archivos. Entre ellas se destacan:

- `ROOT_DIR`: define el directorio raíz del proyecto. Es utilizado como prefijo en todas las rutas relativas de archivos fuente y cabecera.
- `PROJECT_NAME`: establece el nombre base de los binarios generados (`.elf` y `.bin`).

- `CC`, `CXX`, `OBJCOPY`, `OBJDUMP`, `SIZE`: herramientas del toolchain `arm-none-eabi`, usadas para compilar, enlazar, generar binarios y mostrar información de tamaño.

Estas definiciones permiten portar fácilmente el sistema a diferentes módulos simplemente ajustando unas pocas variables.

E.2. Archivos incluidos

Para soportar múltiples pruebas independientes, el `Makefile` define dos bloques de archivos que deben ser ajustados manualmente según el test que se desea compilar:

- `INCLUDES`: contiene los directorios donde se encuentran los archivos `.h`. Se deben incluir tanto los componentes comunes (como `uart`, `i2c`, `pcf8574`, etc.), como el directorio de encabezados específico del módulo de prueba actual. El desarrollador debe asegurarse de no incluir headers innecesarios que no serán utilizados en ese test.
- `SOURCES`: contiene los archivos fuente `.c` necesarios para compilar el firmware. Este listado incluye los archivos del sistema base (como `uart.c`, `i2c.c` o los archivos del RTOS), pero también debe modificarse para incorporar los archivos de la prueba correspondiente y aquellos relevantes para el módulo en cuestión. Por ejemplo, si una prueba no requiere comunicación I^2C, no es necesario incluir `i2c.c` ni su cabecera asociada.

Esta selección manual contribuye a mantener los tiempos de compilación bajos, evita errores por dependencias innecesarias y reduce el tamaño del binario generado.

E.3. Parámetros de compilación

Los parámetros de compilación definidos en `CFLAGS_BASE` fueron cuidadosamente seleccionados para lograr un equilibrio entre seguridad, rendimiento y visibilidad en la depuración:

- `-Wall -Wextra -Wshadow -Wmissing-prototypes`, entre otros: activan advertencias estrictas del compilador, ayudando a prevenir errores sutiles en tiempo de compilación.
- `-mthumb -mcpu=cortex-m3 -msoft-float`: esta combinación de flags configura la compilación para que sea compatible con el microcontrolador STM32F103, que utiliza un núcleo ARM Cortex-M3. En detalle:
  - `-mcpu=cortex-m3`: indica al compilador que genere código específicamente para el núcleo Cortex-M3, optimizando el uso de registros, excepciones y periféricos disponibles en ese modelo.
  - `-mthumb`: obliga al compilador a generar instrucciones en el set *Thumb* (de 16 bits), en lugar del set ARM tradicional (de 32 bits). Esto es obligatorio para el Cortex-M3, ya que sólo ejecuta instrucciones Thumb-2. Además, estas instrucciones ocupan menos espacio en memoria Flash, lo cual es especialmente valioso en sistemas embebidos con recursos limitados.
  - `-msoft-float`: desactiva el uso de instrucciones de coma flotante por *hardware* (FPU, por sus siglas en inglés). El STM32F103 no posee coprocesador de punto flotante, por lo que cualquier operación de este tipo debe resolverse por software. Este *flag* garantiza que el compilador utilice funciones auxiliares para manejar `float` y `double`, evitando errores en tiempo de ejecución por el uso de instrucciones no soportadas por el hardware.

- `-ffunction-sections -fdata-sections -Wl,-gc-sections`: permiten eliminar automáticamente funciones o variables no utilizadas, reduciendo el tamaño final del binario.
- `-O0 -g`: desactiva optimizaciones y habilita información de depuración. Esto es útil durante el desarrollo, pero puede modificarse para producción.
- `-Wl,-undefined,_printf_float`: asegura que se incluya el soporte para `printf` con números flotantes.
- `-flto`: habilita Link Time Optimization para mejorar el rendimiento final, incluso sin optimización de compilación.

El `linker script` se especifica mediante `LDFLAGS`, apuntando a una definición de memoria personalizada para el STM32F103C8T6 [48].

E.4. Compilación condicional y flujos de depuración

Se definen targets adicionales que modifican las `CFLAGS`:

- `make debug`: agrega la macro `ERROR_TREATMENT`, que permite activar mecanismos de validación o mensajes especiales en tiempo de ejecución.

Estas configuraciones condicionan el comportamiento del firmware y son esenciales durante la validación de bajo nivel.

E.5. Reglas auxiliares

Además de compilar, el `Makefile` provee comandos útiles para el entorno de prueba:

- `check_libopencm3`: clona y compila la biblioteca `libopencm3` si no está presente. Esto automatiza la preparación del entorno sin necesidad de pasos manuales.
- `flash`: flashea el firmware en el micro usando `openocd`, programándolo en la dirección 0x08000000.
- `reset`: ejecuta un reset remoto, útil tras flashear para reiniciar el sistema sin intervención física.
- `clean` y `delete`: eliminan archivos objeto, `.elf` y binarios. La diferencia es que `delete` también borra el binario final, asegurando recompilación completa.

F. Grabación

El proceso de grabación del *firmware* en el microcontrolador STM32F103 puede realizarse mediante dos interfaces distintas: **ST-LINK** (usando el protocolo SWD) o **USART** (modo bootloader serial). Ambas opciones fueron implementadas con objetivos diferentes y complementarios. La grabación mediante ST-LINK se utilizó principalmente durante el desarrollo y validación funcional en laboratorio. Por otro lado, la grabación mediante USART fue diseñada específicamente para ser utilizado en el entorno espacial, ya que otro Trabajo Profesional en desarrollo, encargado del módulo de comunicaciones con la estación terrestre, estableció que la transferencia de datos a la OBC se realizará mediante un enlace serie. Esto hace imprescindible contar con una interfaz de programación que funcione exclusivamente por USART, permitiendo actualizaciones de *firmware* en órbita de forma segura y remota, sin depender de interfaces externas como ST-LINK.

F.1. ST-LINK

El método principal de grabación utilizado durante el desarrollo fue mediante la herramienta ST-LINK junto con **OpenOCD** (*Open On-Chip Debugger*). Esta herramienta permite programar, depurar y resetear dispositivos STM32 mediante la interfaz SWD (Serial Wire Debug).

Para ejecutar la grabación, se utiliza el siguiente comando:

```
$   openocd -f /usr/share/openocd/scripts/interface/stlink.cfg -f /usr/
    share/openocd/scripts/target/stm32f1x.cfg -c "program fiubasat.bin 0
    x08000000 verify reset exit"
```

Este comando:

- Configura OpenOCD para usar el archivo de interfaz del programador ST-LINK y el archivo de configuración del microcontrolador STM32F1.
- Programa el archivo `fiubasat.bin` en la dirección de memoria de inicio de la Flash interna del STM32F103: `0x08000000`.
- Realiza verificación, reset y cierre de conexión.

Este procedimiento está automatizado en el `Makefile` mediante la regla `make flash`, por lo que el desarrollador no necesita ejecutarlo manualmente.

Compatibilidad con placas STM32 clonadas

En el caso de utilizar placas STM32 clonadas, es común encontrar diferencias en el identificador TAP (Test Access Port). Por defecto, OpenOCD espera el valor `0x1ba01477`, pero algunos clones requieren que se configure manualmente el valor `0x2ba01477`. Para lograr esto, debe modificarse el archivo:

```
/usr/share/openocd/scripts/target/stm32f1x.cfg
```

Sustituyendo la línea:

```
set _CPUTAPID 0x1ba01477
```

por:

```
set _CPUTAPID 0x2ba01477
```

Este cambio permite la detección del dispositivo, aunque puede incrementar el tiempo de grabación. Cabe destacar que, a partir de la versión 3.0 de la imagen Docker utilizada en el proyecto, este cambio ya no está contemplado, con lo cual su uso debe hacerse dentro imágenes del proyecto anteriores a esta.

Desde Windows (WSL)

En sistemas Windows, la ejecución de contenedores Docker basados en Linux requiere de un entorno compatible con dicho sistema operativo, dado que Docker hace uso de características propias del *kernel* de Linux para gestionar los contenedores. Para resolver esta incompatibilidad, **Docker Desktop** utiliza **WSL 2** (*Windows Subsystem for Linux* versión 2) como capa de compatibilidad, permitiendo la ejecución de distribuciones Linux dentro de un entorno virtualizado liviano integrado al sistema.

Como consecuencia de esta arquitectura, al utilizar dispositivos físicos conectados por USB —como el programador ST-Link— desde un *host* Windows, puede ser necesario compartir explícitamente dichos dispositivos con el entorno Linux dentro de WSL 2. Para ello, es necesario instalar la herramienta **usbipd** (*USB/IP Device Manager*), que permite reenviar dispositivos USB desde Windows hacia una instancia WSL[1].

1. Para obtener el `busid` del dispositivo ST-LINK, ejecutar desde una terminal con privilegios de administrador:

```
$   usbipd list
```

2. Si el dispositivo no aparece como compartido (*shared*), habilitar el acceso desde WSL:

```
$   usbipd bind -busid <ID>
```

3. Finalmente, conectar el dispositivo a WSL:

```
$   usbipd attach -wsl -busid <ID>
```

Una vez realizado esto, es posible utilizar `openocd` desde WSL como si se estuviera en un entorno Linux nativo.

F.2. UART (Bootloader serial)

Como alternativa a la grabación por ST-LINK, se implementó un sistema de grabación mediante UART utilizando el *bootloader* nativo del STM32[2]. Este método permite actualizar el firmware sin necesidad de programadores externos, lo cual es fundamental en un entorno como el espacial, donde el acceso físico al dispositivo no es posible.

Funcionamiento del bootloader

El microcontrolador STM32F103 incluye un bootloader pregrabado por ST en memoria ROM. Este bootloader permite recibir instrucciones a través de interfaces estándar como UART, USB o CAN. En el caso del STM32F1, la interfaz USART1 es la predeterminada. La referencia principal para el funcionamiento del protocolo son las notas de aplicación AN3155 (*USART protocol used in the STM32 bootloader*) y la AN2606 (*STM32 microcontroller system memory boot mode*) [49][50].

Para ingresar al modo de bootloader serial, deben cumplirse las siguientes condiciones durante un reinicio:

- El pin `BOOT0` debe estar en nivel lógico alto (`1`) al momento del reset.
- El pin `NRST` (reset) debe activarse (llevarse a nivel bajo) y luego liberarse.

Bajo estas condiciones, el microcontrolador no ejecuta el firmware cargado en Flash, sino que entra en modo bootloader y queda esperando comandos a través de la UART configurada por *hardware* (USART1 en este caso, pines PA9 y PA10).

[1] Se puede realizar la descarga desde https://github.com/dorssel/usbipd-win

[2] Solo incluido en placas STM32 originales

Secuencia de grabación

La secuencia general para grabar por UART es la siguiente:

1. Activar el pin `BOOT0` (GPIO alto).
2. Ejecutar un pulso de reset en `NRST`.
3. Enviar por UART un byte de sincronización `0x7F`, que debe ser reconocido por el bootloader.
4. Usar el protocolo descrito en AN3155 para enviar comandos como:
   - Leer versión del bootloader.
   - Borrar memoria.
   - Escribir el nuevo firmware binario.
   - Verificar datos escritos.
5. Una vez terminado el proceso, bajar el pin `BOOT0` y reiniciar el microcontrolador para que ejecute el nuevo firmware desde Flash.

Implementación con Raspberry Pi

Este procedimiento fue automatizado en el *script* `Flash.py`, que utiliza la biblioteca `gpiozero` para controlar los pines `BOOT0` y `NRST`, y llama a la herramienta `stm32loader` para realizar la comunicación serie.

El *script* sigue esta lógica:

- Eleva el pin `BOOT0`.
- Aplica un pulso de reinicio (`NRST` bajo y luego alto).
- Llama a `stm32loader` con los siguientes parámetros:

```
$   stm32loader -port /dev/ttyAMA0 -family F1 -baud 115200 -e -v
   -address 0x08000000 -w fiubasat.bin
```

- Baja nuevamente `BOOT0` y reinicia el dispositivo.

Este sistema fue diseñado para ejecutarse desde una Raspberry Pi como plataforma de validación local, pero puede integrarse fácilmente a un sistema embebido de vuelo si se requiere una reprogramación en órbita.

El *script* `Flash.py` puede utilizarse desde consola con los siguientes comandos:

```
$   python3 Flash.py -programar fiubasat_test.bin
$   python3 Flash.py -reset
```

El primer comando graba el *firmware* en la dirección `0x08000000`, mientras que el segundo simplemente reinicia el microcontrolador.

G. Detalles de implementación de USART en STM32

A continuación se detallan ejemplos para la configuración de los puertos, las *flags* y fuentes de interrupción disponibles y también las funciones públicas desarrolladas e implementadas, junto con su firma y descripción.

Se introdujo la posibilidad que se realice un *setup* de los USART tanto en modo sincrónico como asincrónico, pero las pruebas y funciones apuntaron a una ejecución sin *clock* ya que no se requirió de altas velocidades de transmisión, y se optó por liberar la línea necesaria para la interfaz de SPI.

G.1. Configuración del puerto USART

Como se mencionó anteriormente, en este proyecto se hace uso de la librería `libopencm3` como capa de abstracción para el uso del microcontrolador. Es por esto que, mediante el uso de funciones ya predefinidas se pueden configurar las interfaces haciendo referencia a etiquetas nativas. A continuación se muestra un ejemplo de configuración de la interfaz `USART1` en modo asincrónico:

1. **Habilitación del reloj del periférico** (USART y GPIO correspondiente) mediante el módulo RCC:

```
rcc_periph_clock_enable(RCC_USART1);
rcc_periph_clock_enable(RCC_GPIOA);
```

CÓDIGO 3. Configuracion parámetros para USART

2. **Configuración de los pines GPIO** en modo función alternativa:

```
gpio_set_mode(GPIOA, GPIO_MODE_OUTPUT_50_MHZ,
              GPIO_CNF_OUTPUT_ALTFN_PUSHPULL, GPIO_USART1_TX);
gpio_set_mode(GPIOA, GPIO_MODE_INPUT,
              GPIO_CNF_INPUT_FLOAT, GPIO_USART1_RX);
```

CÓDIGO 4. Configuracion parámetros para USART

3. **Configuración de parámetros de comunicación**:
   - Velocidad en baudios (*baud rate*)
   - Bits de datos (8 o 9)
   - Paridad (ninguna, par o impar)
   - Bits de parada (1, 1,5 o 2)
   - Control de flujo (si aplica)
   - Modo de operación (solo TX, solo RX o ambos)

```
usart_set_baudrate(USART1, BAUDRATE);
usart_set_databits(USART1, 8);
usart_set_stopbits(USART1, USART_STOPBITS_1);
usart_set_mode(USART1, USART_MODE_TX_RX);
usart_set_parity(USART1, USART_PARITY_NONE);
usart_set_flow_control(USART1, USART_FLOWCONTROL_NONE);
```

CÓDIGO 5. Configuracion parámetros para USART

4. **Habilitación del USART**:

```
usart_enable(USART1);
```

CÓDIGO 6. Configuracion parámetros para USART

5. **(Opcional) Configuración de Clock para USART sincrónico**: se puede habilitar el clock de la interfaz para que la misma funciones como sincrónico maestro:

```
gpio_set_mode(GPIOA,
            GPIO_MODE_OUTPUT_50_MHZ,
            GPIO_CNF_OUTPUT_ALTFN_PUSHPULL,
            GPIO_USART1_CK);

USART_CR2(USART1) &= ~USART_CR2_CPOL;   // Idle low
USART_CR2(USART1) &= ~USART_CR2_CPHA;   // 1st edge capture
USART_CR2(USART1) |= USART_CR2_CLKEN;   // Modo sincrono
```

CÓDIGO 7. Configuracion parámetros para USART

6. **(Opcional) Configuración de interrupciones**: se puede habilitar la recepción por interrupción con:

```
usart_enable_rx_interrupt(USART1);
nvic_enable_irq(NVIC_USART1_IRQ);
```

CÓDIGO 8. Configuracion parámetros para USART

Y además, como se menciona en 4.1.7, en la implementación del manejo de la interfaz en este proyecto se definió una estructura `uart_t` que se inicializa de la siguiente forma, con su respectivo manejo de errores:

```
static uart_status_t uart_init(uart_t *uart, uint32_t usart) {
        uart->usart = usart;  // Asigna el USART correspondiente

        uart->txq = xQueueCreate(SIZE_BUFFER, sizeof(uint16_t));  // Cola
    de transmision
        if (uart->txq == NULL) return UART_FAIL;

        uart->rxq = xQueueCreate(SIZE_BUFFER, sizeof(uint16_t));  // Cola
    de recepcion
        if (uart->rxq == NULL) return UART_FAIL;

        uart->mutex = xSemaphoreCreateMutex();
        if (uart->mutex == NULL) {
            vQueueDelete(uart->txq);
            return UART_FAIL;
        }

        uart->semaphore = xSemaphoreCreateBinary();
        if (uart->semaphore == NULL) {
            vQueueDelete(uart->txq);
            vSemaphoreDelete(uart->mutex);
            return UART_FAIL;
        }
```

```
        xSemaphoreGive(uart->semaphore);
        return UART_PASS;
}
```

CÓDIGO 9. Configuracion parámetros para USART

En este fragmento de código pueden apreciarse la creación de distintos objetos del *Kernel* de FreeRTOS tales como las colas para la transmisión y recepción de datos, un *mutex* para control de acceso y un semáforo para la sincronización de tareas.

A su vez, se realiza un control de errores para que, en caso de que falle alguna instrucción de creación, se retorne de la función de manera correcta e informando mediante las etiquetas predefinidas cual fue la causa del problema.

G.2. *Flags* importantes USART

- Receive Buffer Full (`RXNE` – `USART_SR`)
- Transmit Buffer Empty (`TXE` – `USART_SR`)
- End of Transmission (`TC` – `USART_SR`)

Control de paridad

- Bit de paridad de transmisión (`PCE`, Parity Control Enable + `PS`, Parity Selection – `USART_CR1`)
- Chequeo de paridad del byte recibido (`PE` – `USART_SR`)

Flags de detección de error

- **Overrun Error (`ORE` – `USART_SR`)**: Este error indica que un nuevo dato ha llegado al receptor antes de que el dato anterior haya sido leído del registro de datos (`USART_DR`). Como consecuencia, el nuevo dato pisa al anterior, sobrescribiendolo y provocando la pérdida irrecuperable de ese byte. Es una situación típica cuando la CPU o la tarea encargada de procesar la UART no lee los datos con suficiente rapidez, ya sea por falta de interrupciones o bloqueos prolongados.
- **Noise Error (`NE` – `USART_SR`)**: Se produce cuando se detecta una transición errónea en los bits recibidos, indicando la presencia de ruido eléctrico en la línea de recepción. Aunque el byte recibido puede no ser descartado automáticamente, su fiabilidad es incierta. Este tipo de error suele estar asociado a problemas de interferencia electromagnética o conexiones defectuosas.
- **Frame Error (`FE` – `USART_SR`)**: Un error de trama ocurre cuando no se detecta correctamente el bit de parada al final de un byte recibido. Esto provoca que la trama completa sea considerada inválida. Las causas más comunes incluyen desajustes en la configuración de la tasa de baudios entre emisor y receptor o condiciones inestables en la señal de transmisión.
- **Parity Error (`PE` – `USART_SR`)**: Este error se detecta cuando el bit de paridad recibido no coincide con el valor esperado, según la configuración de paridad (par o impar). Como consecuencia, el byte recibido se marca como inválido. Las causas típicas incluyen diferencias en la configuración de paridad entre dispositivos o errores de transmisión debido a ruido o sincronización deficiente.

Fuentes de interrupción (con sus banderas asociadas)

- CTS Changes (`CTS` – `USART_SR`, interrupción habilitada con `CTSE` y `CTSIE` – `USART_CR3`)
- LIN Break Detection (`LBDF` – `USART_CR4`)
- Transmit Data Register Empty (`TXE` – `USART_SR`)
- Transmission Complete (`TC` – `USART_SR`)
- Receive Data Register Full (`RXNE` – `USART_SR`)
- Idle Line Received (`IDLE` – `USART_SR`)
- Overrun Error (`ORE` – `USART_SR`)
- Framing Error (`FE` – `USART_SR`)
- Noise Error (`NE` – `USART_SR`)
- Parity Error (`PE` – `USART_SR`)

G.3. Funciones desarrolladas para UART

Las funciones desarrolladas en esta implementación de la interfaz UART se encuentran declaradas y documentadas en el archivo `uart.h` junto con los parámetros que reciben, lo que devuelven y los posibles errores dentro de las mismas.

G.4. Funciones utilizadas de `libopencm3` USART

1. Funciones de configuración
    - `rcc_periph_clock_enable(uint32_t periph)`: Habilita el reloj del periférico especificado (por ejemplo, GPIOA, USART1, etc.).
    - `gpio_set_mode(uint32_t gpio_port, enum gpio_mode mode, enum gpio_cnf cnf, uint16_t gpios)`: Configura el modo y función alternativa de los pines GPIO.
    - `nvic_enable_irq(uint8_t irqn)`: Habilita la interrupción especificada en el NVIC.
    - `usart_set_baudrate(uint32_t usart, uint32_t baud)`: Configura la velocidad en baudios del USART.
    - `usart_set_databits(uint32_t usart, int bits)`: Establece la cantidad de bits de datos por trama (usualmente 8 o 9 bits).
    - `usart_set_stopbits(uint32_t usart, uint32_t stopbits)`: Configura la cantidad de bits de parada (1, $1,5$ o 2).
    - `usart_set_mode(uint32_t usart, uint32_t mode)`: Define si el USART funciona como transmisor, receptor o ambos.
    - `usart_set_parity(uint32_t usart, uint32_t parity)`: Configura el tipo de paridad (ninguna, par, impar).
    - `usart_set_flow_control(uint32_t usart, uint32_t flowctrl)`: Configura el control de flujo por *hardware* (RTS/CTS).
    - `usart_enable(uint32_t usart)`: Habilita el periférico USART.

- `usart_enable_rx_interrupt(uint32_t usart)`: Activa la interrupción por recepción de datos (RXNE).

2. Funciones operativas

- `usart_get_flag(uint32_t usart, uint32_t flag)`: Verifica si una bandera del estado USART está activa (por ejemplo, TXE o RXNE).
- `usart_send(uint32_t usart, uint16_t data)`: Envía un byte (o palabra) a través del USART de manera no bloqueante.
- `usart_recv(uint32_t usart)`: Lee un byte recibido desde el registro de datos del USART.

G.5. GPS GY-NEO6MV2

El GY-NEO6MV2 es un receptor GPS de uso terrestre con una sensibilidad de -161 dBm para el rastreo y navegación que posee una frecuencia máxima de actualización de 1Hz.

Como protocolo de comunicación se utiliza UART, con una posible configuración de baudrate como se indica en la Figura 6 utilizando sus pines **CFG_COM0** y **CFG_COM1** (el valor por defecto es 9600 bauds):

CFG_COM1	CFG_COM0	Protocol	Messages	UARTBaud rate	USB power
1	**1**	**NMEA**	**GSV, RMC, GSA, GGA, GLL, VTG, TXT**	**9600**	**BUS Powered**
1	0	NMEA	GSV, RMC, GSA, GGA, GLL, VTG, TXT	38400	Self Powered
0	1	NMEA	GSV[14], RMC, GSA, GGA, VTG, TXT	4800	BUS Powered
0	0	UBX	NAV-SOL, NAV-STATUS, NAV-SVINFO, NAV-CLOCK, INF, MON-EXCEPT, AID-ALPSERV	57600	BUS Powered

FIGURA 6. Configuraciones COM soportadas.

Este módulo resulta útil para el desarrollo inicial de *software* y pruebas funcionales en tierra, donde se pueden validar la recepción y el procesamiento de tramas NMEA sin necesidad de *hardware* especializado.

Configuración GPS Neo6

El módulo GPS NEO-6MV2 permite diversas configuraciones para adaptarse a las necesidades específicas que se tengan. Estas configuraciones incluyen la tasa de baudios, el formato de salida de datos, la frecuencia de actualización, el control de energía y el modo de operación. A continuación, se describen las principales configuraciones y cómo se implementan utilizando comandos UBX (UBX Protocol, NEO-6-Datasheet).

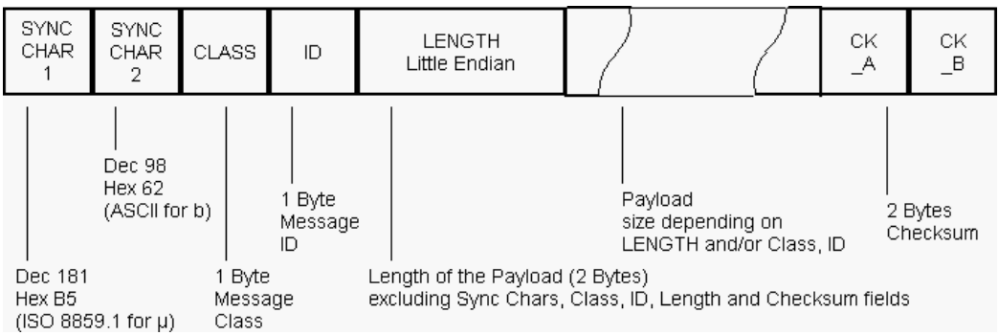

FIGURA 7. Estructura de paquete UBX

- Todo mensaje comienza con 2 bytes de sincronización: `0xB5 0x62`.
- Sigue un campo de **Clase** de 1 byte, que define la categoría del mensaje.
- Luego se encuentra un campo de **ID** de 1 byte, que identifica el tipo específico de mensaje.
- A continuación, hay un campo de **Longitud** de 2 bytes, que indica el tamaño de la carga útil (*payload*). La longitud solo considera la carga útil y no incluye los bytes de sincronización, la clase, el ID ni el CRC. El formato de este campo es un entero sin signo de 16 bits en *Little Endian*.
- La carga útil (*payload*) es un campo de longitud variable que contiene los datos específicos del mensaje.
- Finalmente, el mensaje incluye un **Checksum** (CK_A y CK_B) de 16 bits, calculado según el procedimiento definido en el manual.

Estas configuraciones pueden hacerse o bien formateando una cadena de hexadecimales y enviándola al módulo GPS, o utilizando el *software* **u-center**, con el cual, mediante una interfaz gráfica, pueden seleccionarse los parámetros y valores a configurar en el periférico.

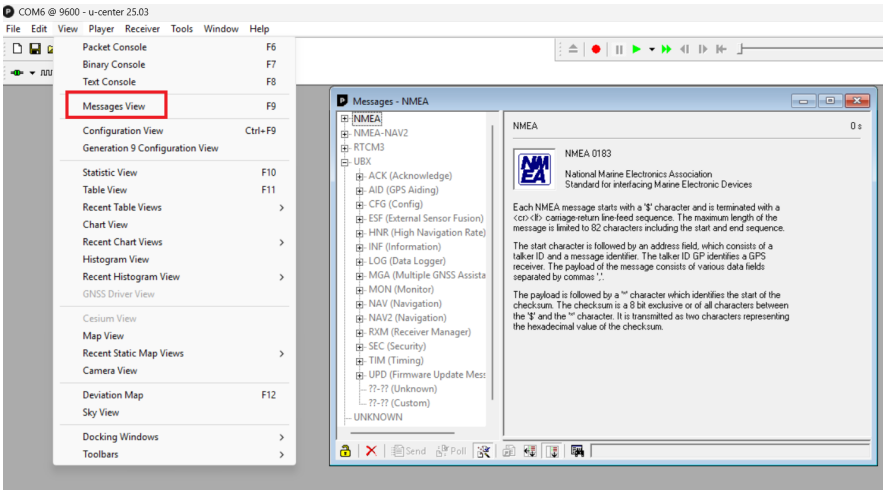

FIGURA 8. Vista de configuración GPS mediante U-center

A su vez, dentro de este programa, en caso de poseer una interfaz serie entre la computadora y la fuente de datos, es posible enviar las tramas NMEA provenientes del GPS para hacer uso de distintos gráficos y mapas que procesan la información recibida.

H. Detalles de implementación de I²C en STM32

La comunicación I²C, si bien definida por un protocolo estándar, impone ciertos requisitos particulares según la implementación de *hardware* del periférico. En el caso de los microcontroladores STM32, el uso del periférico I²C requiere una atención especial al orden en que se configuran los distintos registros, especialmente cuando el dispositivo actúa como maestro receptor. Esto se debe a que el periférico I²C de STM32 no reacciona de forma inmediata a cada evento de manera autónoma, sino que depende de ciertas señales de control (como ACK/NACK o STOP) que deben estar preconfiguradas antes de momentos clave del intercambio de datos. De lo contrario, el comportamiento del bus puede volverse errático, o incluso quedar bloqueado en ciertas condiciones.

Esta exigencia de secuencialidad es más crítica en el modo Master Receiver, donde el maestro no controla directamente el flujo de datos byte a byte, sino que debe anticipar cómo finalizará la comunicación antes de recibir los últimos datos. Esto contrasta con el modo Master Transmitter, en el que la lógica es más directa: cada byte a transmitir se carga explícitamente y se aguarda a que el periférico confirme su envío.

H.1. Configuración del puerto

Para utilizar el periférico I²C en un microcontrolador STM32, es necesario seguir una serie de pasos que configuran tanto el *hardware* de bajo nivel (pines, relojes) como los parámetros del protocolo I²C.

En primer lugar, se deben habilitar el reloj del periférico y de los pines. En los microcontroladores STM32, los periféricos internos están conectados al procesador mediante un conjunto jerárquico de buses internos. Entre ellos, se destacan el *Advanced High-performance Bus* (AHB) y el *Advanced Peripheral Bus* (APB), definidos por la arquitectura AMBA (*Advanced Microcontroller Bus Architecture*) de ARM.

El AHB conecta el núcleo del procesador (Cortex-M3) con memorias (SRAM, Flash) y otros bloques de alto rendimiento, y actúa como el bus principal del sistema. Desde el AHB se derivan dos buses secundarios llamados APB1 y APB2, que se encargan de interconectar al núcleo con los periféricos de menor velocidad. En particular, los periféricos I2C1 e I2C2 están conectados al bus APB1. Este bus opera a una frecuencia derivada del reloj del sistema (`SYSCLK`) mediante divisores configurables dentro del sistema de reloj y distribución de la STM32.

Para que un periférico funcione, es necesario habilitar su señal de reloj interna mediante el RCC (*Reset and Clock Control*), que controla la distribución del reloj a todos los periféricos. Esto se hace activando el bit correspondiente al periférico (por ejemplo, I2C1 o I2C2) en el registro `RCC_APB1ENR`. Si no se habilita el reloj del periférico, este permanecerá inactivo, incluso si se intenta configurar o acceder a sus registros.

También se debe habilitar el reloj del puerto GPIO correspondiente a los pines que se utilizarán como líneas SCL y SDA. Estos pines deben estar configurados en modo salida a 50 MHz con función alternativa (no como un pin E/S normal, es decir, GPIO -*General purpose input/output*) y *open-drain*, tal como lo requiere la especificación del bus I²C. Habilitar el reloj del GPIO también se hace desde el RCC, pero usando el registro `RCC_APB2ENR`.

Antes de comenzar la configuración del periférico I²C, es recomendable asegurarse de que no esté en un estado indeterminado debido a errores previos. Esto se logra desactivando el periférico I²C (limpiando el bit `PE` en el registro `CR1`) y generando un pulso de reset al periférico mediante RCC, lo que lo reinicia completamente (esto se hace escribiendo y limpiando el bit correspondiente en `RCC_APB1RSTR`).

Una vez que el periférico está en un estado limpio, se puede proceder a configurar el protocolo I²C propiamente dicho. Para esto se debe definir:

- Modo a utilizar (en este trabajo, modo estándar, es decir, con una velocidad de 100 kHz)
- Frecuencia del bus APB1 (`PCLK1`): es necesario indicarle al periférico I²C a qué frecuencia está trabajando el bus APB1, ya que de esto depende el cálculo de los tiempos. Esto se hace configurando los primeros bits del registro CR2, indicado en MHz. En la frecuencia típica para este *bus* de $36MHz$, se debe escribir el valor 36.
- Ciclo de trabajo. Este parámetro aplica al modo rápido, donde se puede elegir el ciclo de trabajo entre dos configuraciones: 2:1 (el pulso alto dura 2 veces menos que el bajo) o 16:9 (el pulso alto dura 9 partes y el bajo 16). Esto se configura en el bit `DUTY` del registro `CCR`. En modo estándar (100 kHz), esto no se usa, pero se suele dejar explícito `DUTY` = 0 (modo 2:1) para que el periférico no quede en un estado indefinido.
- Se configura el registro `CCR` (*Clock Control Register*) que define el período del reloj I²C en función de la frecuencia del APB. En modo estándar se calcula como

$$\text{CCR} = \frac{f_{\text{PCLK1}}}{2\ f_{\text{SCL}}} = \frac{36\ MHz}{2 \cdot 100\ kHz} = 180$$

- Tiempo de subida (`TRISE`). Este valor define el tiempo máximo de subida de la señal SCL en cantidad de ciclos de `PCLK1`. En modo estándar, el valor se calcula como

$$\text{TRISE} = \frac{t_{\text{rise}}}{T_{\text{PLCK1}}} + 1 \approx \frac{1000\ ns}{27,78\ ns} + 1 \approx 37$$

  donde el tiempo de crecimiento en segundos corresponde al máximo especificado por el estándar.

Una vez completada toda la configuración, el periférico puede ser activado nuevamente. Esto se hace seteando el bit PE en el registro CR1. Desde este momento, el periférico está listo para iniciar comunicaciones I²C como maestro o esclavo, dependiendo de cómo se use más adelante en el código.

H.2. Comienzo de la comunicación

Para comenzar una comunicación como maestro, el primer paso es generar una condición de *Start* (inicio). Esto se logra activando el bit `START` dentro del registro `CR1` del puerto. Al hacerlo, el controlador toma el control del bus y cambia internamente al modo maestro, siempre que el bus no esté ocupado (es decir, el bit `BUSY` esté en 0). En caso de que ya se esté en una comunicación, volver a escribir el bit `START` genera una condición de *Repeated Start*, permitiendo encadenar transferencias sin liberar el bus.

Una vez generada esta condición de inicio, el periférico espera la confirmación de que el evento fue exitoso. Esto se indica mediante el bit `SB` (*Start Bit*) en el registro `SR1`, que se activa automáticamente cuando la condición de Start ha sido transmitida en la línea SDA. Una vez comprobado esto, se puede continuar con la transferencia, escribiendo en el registro de datos (`DR`) la dirección del esclavo con el cual se desea establecer la comunicación. En caso de que se vaya a realizar la lectura de datos, es un momento oportuno para activar el bit `ACK` del `CR1` para que el maestro responda con un ACK a cada dato que reciba del esclavo.

En modo de direccionamiento de 7 bits, se transmite un único byte de dirección (7 bits de dirección más el bit $\text{R}/\overline{\text{W}}$). Este último bit, correspondiente al bit menos significativo (LSB), determina el modo de operación deseado: si es 0, el maestro indica que desea transmitir datos (modo maestro-transmisor). Si es 1, el maestro solicita recibir datos (modo maestro-receptor).

Una vez que el byte de dirección ha sido colocado en el bus, el periférico monitorea la respuesta del esclavo. Si el esclavo existe y reconoce la dirección enviada, responderá con un ACK, y el controlador I²C activará el bit `ADDR` en el registro `SR1`. Este bit señala que la etapa de direccionamiento ha concluido con éxito.

Limpieza del registro `ADDR`

El *flag* `ADDR` no se borra automáticamente. Para que el periférico pueda continuar con la comunicación (ya sea enviando o recibiendo datos), el *software* debe ejecutar una lectura de `SR1` seguida inmediatamente por una lectura de `SR2`. Esta secuencia específica es obligatoria y permite liberar internamente el bus para que se activen las siguientes etapas del protocolo (por ejemplo, permitir el acceso al registro DR, o la detección de condiciones de parada).

Es muy importante que esta secuencia se realice en el orden preciso especificado en el manual de referencia RM0008 provisto por ST, especialmente en el modo maestro-receptor. Este orden, aunque al principio pueda parecer que entra en contradicción con el protocolo antes descrito, se debe a que en la implementación *hardware* de STM32, la generación de ciertas condiciones (como NACK o STOP) no es instantánea sino que requieren de una cierta anticipación para que se ejecuten justo a tiempo durante el ciclo de reloj correcto. Cuando se reciben datos, el byte entrante primero pasa por un *shift register* interno, donde se arma bit por bit. Al completarse, ese byte se copia al data register (DR) y se activa el *flag* `RxNE`. La STM32 no espera pasivamente. En cuanto el bus está libre para el siguiente byte, se lanza a recibir el próximo automáticamente. Si el *software* no se anticipa, el puerto va a seguir leyendo más bytes incluso si no se requieren más, dando lugar a datos incorrectos.

H.3. Modo maestro-transmisor

Una vez que el maestro ha enviado correctamente la dirección del esclavo, se limpia el bit `ADDR` (con la lectura secuencial de los registros `SR1` y `SR2`). Así comienza la fase propia de la transmisión y el maestro comienza a escribir bytes en el registro de datos (`I2C_DR`), que serán enviados automáticamente al bus SDA por medio del *shift register* interno del periférico I²C.

Cada vez que se escribe un byte en `I2C_DR`, el periférico lo transmite serialmente, bit a bit. Cuando el esclavo confirma la recepción del byte mediante un pulso de ACK, el bit `TxE` (*Transmit Data Register Empty*) se activa automáticamente, indicando que el registro `DR` está libre y se puede escribir el siguiente byte.

En caso de que no se escriba un nuevo byte en `DR` a tiempo, y la transmisión anterior haya finalizado, el periférico activa también el bit `BTF` (*Byte Transfer Finished*), indicando que tanto el *shift register* como el registro de datos están vacíos, y el bus queda en espera de una nueva acción. Mientras tanto, el periférico mantiene la línea SCL en estado bajo (*clock stretching*), lo cual detiene la comunicación hasta que se resuelva esta condición. La escritura de un nuevo dato en `DR` limpia automáticamente el bit `BTF` y le permite al bus reanudar su operación normalmente.

Una vez que se escribió el último dato que se desea enviar en el registro `DR`, se establece el bit `STOP` en el `CR1` en alto para generar la condición de *stop*.

Todos estos pasos pueden observarse en la figura 9

H.4. Modo maestro-receptor

Una vez establecido el bit `ADDR` se da lugar a la fase de recepción de datos. Para implementar la misma, existen dos métodos principales: el basado en interrupciones, recomendado cuando el I²C tiene la más alta prioridad en el sistema, y el basado en *polling*, pensado

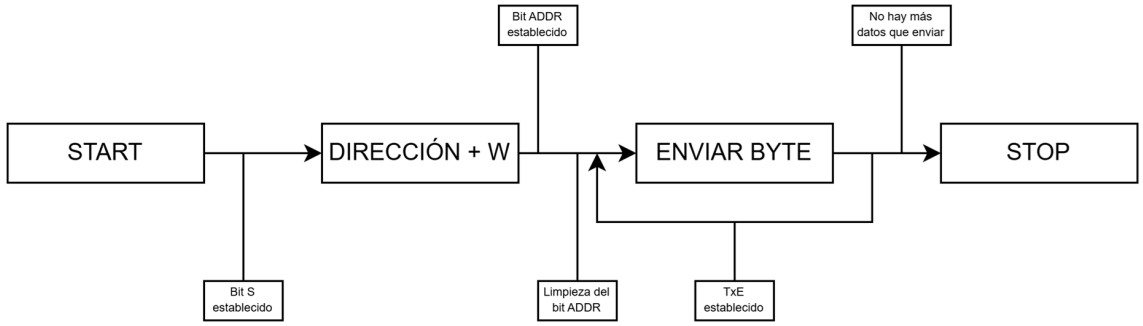

FIGURA 9. Pasos de una transmisión I²C

para situaciones donde no se garantiza esa prioridad o se prefiere una lógica más simple de seguimiento de eventos.

En este trabajo se optó por la segunda opción, aunque con una estrategia optimizada: se utiliza *polling* controlado mediante *FreeRTOS*, lo cual permite monitorear los *flags* `RxNE` y `BTF` sin bloquear completamente la CPU. Durante las esperas por cada evento, se invoca a `taskYIELD()` dentro de un bucle con *timeout*, lo que permite ceder voluntariamente el control al planificador del sistema operativo. De este modo, otras tareas pueden ejecutarse mientras el periférico I²C espera que se complete cada paso del protocolo, logrando así un enfoque híbrido que combina la previsibilidad del *polling* con la eficiencia del *multitasking*.

Como se explicó anteriormente, el orden en el cual se comanda las instrucciones para la lectura de datos puede variar dependiendo la cantidad de datos que se espera recibir, debido a la implementación de doble registro del *hardware* de la STM32.

Recepción de un byte

Los pasos a seguir para la recepción de un único byte luego de enviar la dirección del esclavo y que se haya seteado el bit `ADDR` son los siguientes:

1. Limpiar el bit `ACK` en el `CR1` para deshabilitar la señal de `ACK`
2. Limpiar el bit `ADDR` (leyendo `SR1` y luego `SR2`)
3. Establecer el bit `STOP` en el `CR1` en alto para generar la condición de *stop*
4. Esperar a que el *flag* `RxNE` del `SR1` esté seteada, indicando que hay datos disponibles en el *buffer* de recepción
5. Leer el dato del `DR`

A simple vista podría parecer que se está modificando el orden especificado por el estándar I²C (por ejemplo, puede llamar la atención enviar la señal de parada antes de leer el dato) pero en realidad es seguir esta secuencia lo que permite que las señales se envíen en el orden preciso.

Cuando se deshabilita el bit `ACK`, el periférico se prepara para responder con un *NACK* al siguiente byte recibido. Esto es fundamental, ya que si el `ACK` se desactiva después de leer el dato, el *NACK* se aplicaría al siguiente byte, no al actual, lo cual rompería la secuencia de recepción. A continuación, se limpia el bit `ADDR`, lo que permite que el dato recibido avance desde el *shift register* hacia el `DR`. En ese momento, se genera la condición de *Stop* mediante el bit `STOP`. Aunque pueda parecer que la señal de parada se emite antes de que el dato esté disponible, en realidad este ya fue capturado por el periférico en el *shift register* y se

encuentra listo para ser leído desde el DR. Finalmente, se accede a este registro una vez que el *flag* de RXNE (*Receive buffer not empty*) esté seteada (es decir, que haya datos en el *buffer* de recepción).

Esta secuencia puede observarse en la figura 10

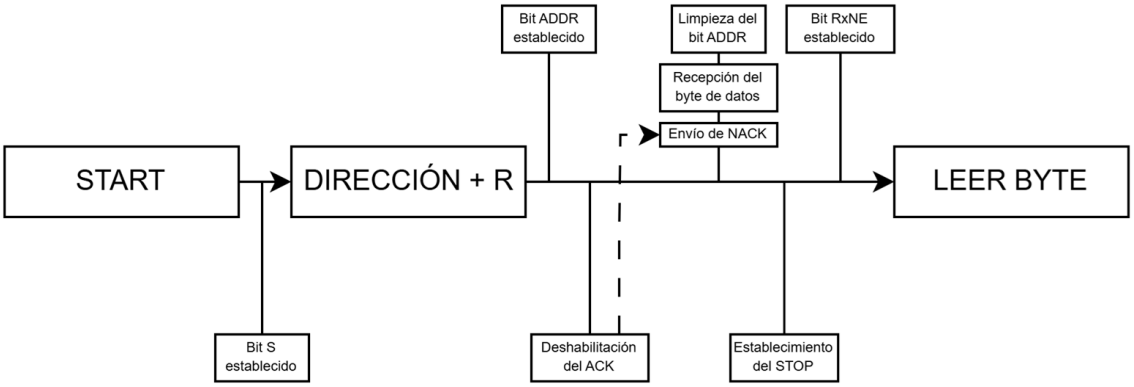

FIGURA 10. Recepción de un byte en STM32

Recepción de dos bytes

Análogamente, luego de enviar la dirección del esclavo y de que se haya seteado el bit ADDR, los pasos a seguir para la recepción de dos bytes son:

1. Establecer el bit POS en el registro CR1 en alto, para indicarle al periférico que postergue un byte la evaluación del bit ACK.
2. Limpiar el bit ADDR (leyendo SR1 y luego SR2).
3. Deshabilitar el bit ACK en el CR1, para que se envíe un NACK tras la recepción del segundo byte.
4. Esperar a que el *flag* BTF en el SR1 se active, lo que indica que hay datos tanto en el registro de desplazamiento como en el registro de datos.
5. Leer el DR dos veces para obtener ambos bytes (verificando que RXNE esté activo antes de cada lectura).

En este caso, al activar el bit POS, se le indica al periférico que evalúe el bit ACK después del segundo byte recibido, es decir, cuando BTF está activado. Esto permite deshabilitar ACK luego de limpiar ADDR, sin riesgo de que el NACK se envíe de forma anticipada.

Una vez que ambos bytes han sido recibidos (es decir, cuando se activa BTF), se puede generar la condición de parada mediante el bit STOP. Al leer el primer byte desde DR, el segundo byte es transferido automáticamente desde el *shift register* al DR, permitiendo su posterior lectura.

Esta secuencia puede observarse en la figura 11

Recepción de tres o más bytes

En este caso, la secuencia de pasos a seguir es más parecida a la lógica mencionada en el estándar, salvo por algunas diferencias en la recepción de los últimos tres bytes.

Mientras la cantidad de bytes esperados sea mayor a tres, y recordando que el bit ACK ya fue activado, la secuencia para recibir cada dato es:

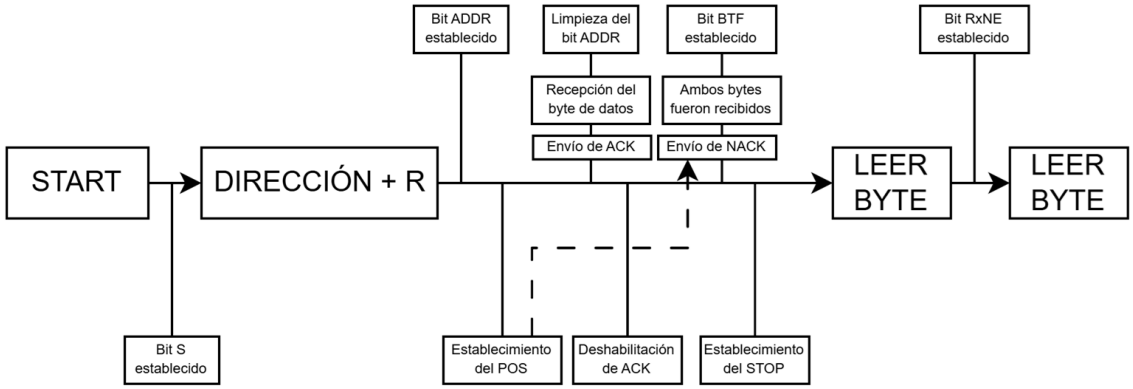

FIGURA 11. Recepción de dos bytes en STM32

1. Esperar a que `RXNE` esté activado, indicando que hay un dato en el *buffer* de recepción
2. Leer el `DR` para poder obtener el dato

En este caso, a cada byte que se recibe del esclavo, el maestro responde con un ACK.

Sean `DataN-2`, `DataN-1` y `DataN` el antepenúltimo, penúltimo y último byte de datos a recibir, respectivamente. Una vez que se leyó el dato anterior a `DataN-2`, la secuencia de pasos a seguir es:

1. Esperar a que el bit `BTF` esté seteado
2. Desactivar el bit de `ACK`
3. Leer el registro `DR` para obtener `DataN-2`
4. Esperar a que el bit `BTF` esté nuevamente seteado
5. Establecer el bit de `STOP` en alto para enviar la señal de parada
6. Leer el `DR` dos veces para obtener ambos bytes (verificando que `RXNE` esté activo antes de cada lectura)

Una vez que quedaban tres bytes por recibir, si el bit `BTF` fue seteado significa que tanto el registro `DR` como el *shift register* están ocupados, con lo cual se recibieron `DataN-2` y `DataN-1`, ambos respondidos con su ACK correspondiente. El puerto, entonces, queda a la espera de la lectura de `DR` para permitir la recepción del último byte. Por eso, se desactiva el bit `ACK` antes de leer dicho registro, para que el microcontrolador sepa que al próximo byte a recibir (`DataN`) debe responder con un NACK.

Luego de esto, se lee el `DR` (`DataN-2`), habilitando así la recepción del siguiente dato. En este punto, y una vez que `BTF` se activa nuevamente, señalando que ambos registros están ocupados (`DataN-1` en el `DR` y `DataN` en el *shift register*), se configura el bit `STOP` para enviar la señal de parada.

Finalmente, se lee el `DR` para obtener `DataN-1`, y una vez que `RXNE` indica que `DataN` ya se copió desde el *shift register* al registro de datos, se vuelve a leer `DR` para obtener el último byte.

Esta secuencia puede observarse en la figura 12.

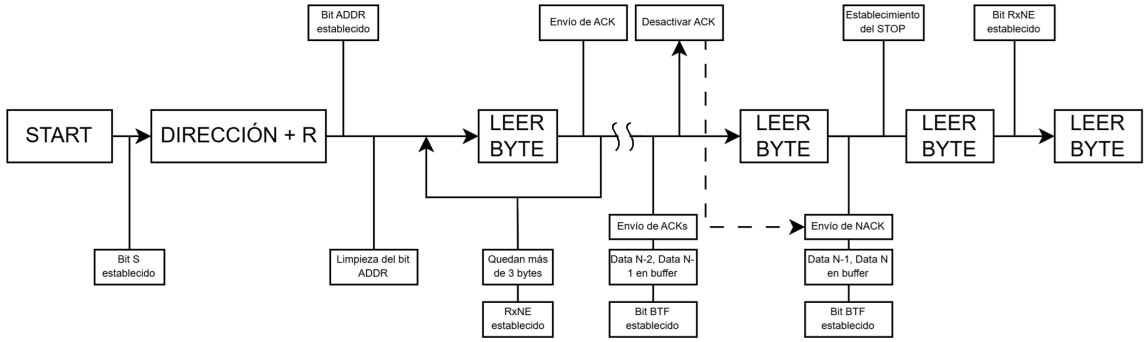

FIGURA 12. Recepción de tres o más bytes en STM32

H.5. *Flags* importantes

A continuación, se muestran los *flags* más relevantes para este trabajo incluidas en los registros de estado `SR1` y `SR2` del periférico I^2C.

1. `SR1`

   - `SB` - *Start Bit* (Bit 0). Indica que la condición de *start* fue generada correctamente y debe verificarse antes de continuar con la comunicación. Para limpiarla correctamente y evitar bloqueos, debe leerse el registro `SR1` antes de cargar la dirección del esclavo en `DR`.
   - `ADDR` - *Address Sent* (Bit 1). Indica que la dirección fue enviada y el esclavo respondió con un ACK. Debe verificarse antes de continuar con la comunicación y su limpieza es mandatoria para desbloquear la comunicación y permitirla continuación de la transacción.
   - `BTF` - *Byte Transfer Finished* (Bit 2). Señala que una transferencia de byte ha terminado: en recepción, indica que hay un byte completo recibido en `DR` y otro en el *shift register*; en transmisión, que se envió un byte y DR ya fue vaciado (TxE $= 1$). Se debe verificar para detectar cuando los *buffers* de recepción están llenos o cuando se puede enviar un nuevo byte o enviar una señal de *stop*, según se esté en recepción o transmisión.
   - `RxNE` - *Receive Data Register Not Empty* (Bit 6). Indica que hay un byte recibido en el registro `DR` listo para ser leído y debe ser evaluado antes de intentar obtener el byte esperado en la recepción.
   - `TxE` - *Transmit Data Register Empty* (Bit 7). Indica que el registro de transmisión (`DR`) está vacío y listo para recibir un nuevo byte a transmitir. Debe ser evaluado antes de intentar cargar un byte en la transmisión.
   - `BERR` - *Bus Error* (Bit 8). Indica que se detectó una condición ilegal en el bus, como un cambio de SDA mientras SCL está alto fuera de una secuencia START/STOP válida. Debería chequearse en casos de error, por ejemplo, si el *bus* quedó colgado.
   - `AF` - *Acknowledge Failure* (Bit 10). Indica que el esclavo no respondió con un ACK luego de recibir una dirección o un byte de datos (es decir, se recibió un NACK). Debe ser evaluado luego de operaciones de transmisión donde se espere una respuesta, y debe limpiarse manualmente escribiendo 0 en el bit correspondiente.

2. `SR2`

- `BUSY` - *Bus Busy* (Bit 1). Indica que hay una comunicación en curso en el bus I²C. Es seteada automáticamente por *hardware* al detectar que las líneas `SDA` o `SCL` están en nivel bajo, y se limpia cuando se detecta una condición de *stop*. Es útil para verificar si el bus está libre antes de iniciar una nueva transmisión, o para detectar condiciones anómalas como un bus colgado.

H.6. Detalles de uso de los dispositivos esclavos

PCF8574 y SN74HC4066

La dirección del multiplexor PCF8574 es de 7 bits y comienza con el prefijo `0100`. Los tres bits restantes están definidos por los pines A_2, A_1, A_0: si están conectados a VCC representan un 1, si están conectados a GND representan un 0. De esta forma, la dirección completa es $0100A_2A_1A_0$. En este trabajo, se conectaron todos los pines a GND con lo cual la dirección es `0x20`.

El orden de la transmisión puede observarse en la figura 13.

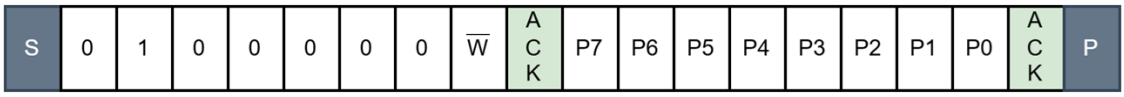

FIGURA 13. Secuencia de escritura para el PCF8574 (bit $\overline{W} = 0$)

El PCF8574 actúa como esclavo en el bus I²C, y para este trabajo se lo utiliza únicamente como salida digital. Para ello, se le debe enviar un byte que define el estado de cada una de sus salidas: un bit en '0' establece el pin en bajo, y un bit en '1' lo coloca en alto. El orden de los bits en el byte de datos sigue el formato: P7 (MSB), P6, ..., P1, P0 (LSB), donde cada bit controla un puerto específico del expansor.

El diagrama de conexiones propuesto en este trabajo puede observarse en la tabla 1.

PIN	SWITCH	C	DISPOSITIVO	LÍNEA I2C
0	1	1C	ATmega328P	SCL
1	1	4C	MPU6050	SCL
2	1	2C	ATmega328P	SDA
3	1	3C	MPU6050	SDA
4	2	1C	AT24C256	SDA
5	2	4C	HTU21D	SCL
6	2	2C	AT24C256	SCL
7	2	3C	HTU21D	SDA

CUADRO 1. Asignación de entrada de control C de los interruptores en el byte de control del multiplexor

HTU21D

El dispositivo HTU21D opera con una tensión de alimentación entre 1,5 V y 3,6 V. Tras la alimentación, el sensor necesita un tiempo de hasta 15 ms —durante el cual la línea SCL debe permanecer en estado alto— para alcanzar el estado inactivo (*sleep mode*) y estar listo para recibir comandos. Por esta razón, se recomienda emitir un reinicio por *software* (*soft reset*) al comenzar una sesión de comunicación. La dirección I²C del HTU21D es `0x40`.

Desde el punto de vista funcional, el HTU21D admite dos modos de operación: *Hold Master* y *No Hold Master*. En el modo *Hold Master*, la línea SCL es controlada por el sensor

durante la medición, forzando al maestro a esperar hasta que los datos estén listos. Una vez finalizada la medición, el sensor libera la línea SCL, permitiendo continuar la transmisión. Por otro lado, en el modo *No Hold Master*, el sensor no bloquea la línea SCL, lo que permite al maestro realizar otras tareas mientras el sensor completa la conversión. En este caso, el maestro debe sondear periódicamente al sensor mediante una condición de inicio y la dirección de lectura. Si la medición aún no está lista, el sensor responde con un NACK; cuando está disponible, responde con un ACK y los datos correspondientes.

Para iniciar una medición o realizar operaciones específicas, el maestro debe enviar uno de los siguientes comandos:

- `0xE3`: Medición de temperatura (modo *Hold Master*).
- `0xE5`: Medición de humedad relativa (modo *Hold Master*).
- `0xF3`: Medición de temperatura (modo *No Hold Master*).
- `0xF5`: Medición de humedad en modo *No Hold Master*.
- `0xFE`: Reinicio por software.

La secuencia para la medición de temperatura (inicializando el sensor con el *reset*) puede observarse en la figura 14. Esto se realiza en tres secuencias: un reinicio, una solicitud de medición y una lectura de 3 bytes (parte alta y baja del dato y CRC). La primera parte es la escritura del comando de reinicio por *software* (0x$FE = 11111110$b). La segunda, la escritura del comando de medición de temperatura en modo *no hold master* (0x$F3 = 11110011$b). La última, por su parte, es una solicitud de lectura donde el NACK es enviado en el tercer dato.

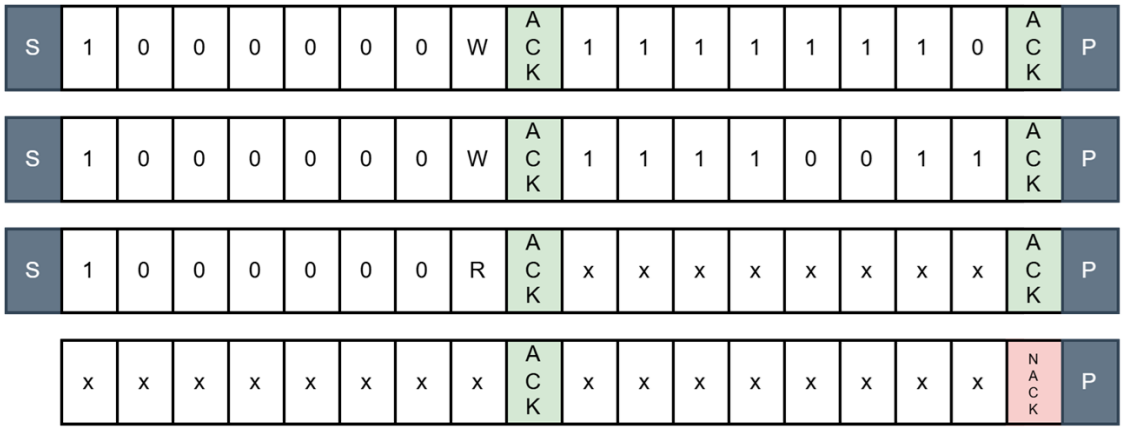

FIGURA 14. Secuencia para la obtención de la temperatura en el HTU21D

Los datos de humedad y temperatura se transmiten como valores de 16 bits. Para obtener el valor físico en unidades reales, se utilizan las siguientes fórmulas:

$$RH\,\% = -6 + 125\,\frac{S_{RH}}{2^{16}}$$

$$T = -46{,}85 + 175{,}72\,\frac{S_T}{2^{16}}$$

donde S_{RH} y S_T son los valores de 16 bits obtenidos en la lectura.

1 U En caso de necesitar una reconfiguración del sensor o ante errores de comunicación, el comando `0xFE` permite ejecutar un reinicio por software. Esta operación restablece todos

los valores del registro de usuario a su configuración predeterminada, con excepción del bit del calefactor. El proceso de reinicio tiene una duración inferior a 15 ms y es recomendable para garantizar una inicialización limpia.

Finalmente, es importante destacar que cada medición enviada por el sensor incluye un byte adicional de CRC, utilizado para verificar la integridad de los datos transmitidos. El polinomio aplicado es:

$$X^8 + X^5 + X^4 + 1 \quad \text{(hex: 0x31)}$$

Este algoritmo CRC-8 permite detectar múltiples tipos de errores, incluyendo errores simples, dobles y ráfagas de hasta 8 bits. En este proyecto, dado que el microcontrolador STM32F103 solo dispone de un módulo de *hardware* compatible con CRC-32, fue necesario implementar el cálculo de CRC-8 por software, conforme al polinomio especificado por el fabricante. Esta verificación resultó esencial para garantizar la confiabilidad de las lecturas, especialmente en condiciones electromagnéticas adversas o ante posibles fallos en la línea I²C.

MPU6050

La dirección I²C del dispositivo MPU6050 es de 7 bits y corresponde a `110100A`$_0$, donde `A`$_0$ es un pin disponible para conectar a VCC (y que el bit sea igual a 1) o GND (y que el bit sea igual a 0). En este trabajo se conectó a GND con lo cual la dirección en hexadecimal corresponde a `0x68` [51] [52].

La MPU6050 posee múltiples registros de control que permiten configurar su funcionamiento y consultar su estado. Para escribir en un registro, se deben realizar dos operaciones de escritura consecutivas: primero se envía la dirección del registro, y luego el valor que se desea almacenar en él. La lectura también requiere dos etapas: una operación de escritura para indicar la dirección deseada, seguida de una operación de lectura. Esta secuencia puede realizarse mediante una condición de *repeated start*, sin liberar el bus entre ambas operaciones, permitiendo así obtener el contenido del registro en cuestión.

En cuanto a la adquisición de datos de aceleración y velocidad angular, el dispositivo dispone de un bloque de 14 registros consecutivos que almacenan las mediciones de todos los ejes. Esta información se organiza en: 6 bytes para el acelerómetro (dos por eje: byte alto y bajo), 2 bytes para la temperatura interna (que no se utilizarán en esta implementación), y 6 bytes para el giroscopio (también distribuidos en pares de bytes por eje). Cada par representa una medición de 16 bits en formato complemento a dos.

Si bien es posible acceder individualmente a un eje o incluso a un solo byte, en esta aplicación se optó por realizar lecturas secuenciales de todos los ejes en una única transacción I²C. Esto se logra solicitando 6 bytes a partir del primer registro del bloque correspondiente (ya sea el del acelerómetro o el del giroscopio). Esta estrategia garantiza que las mediciones de los tres ejes (X, Y y Z) se obtengan de forma simultánea, evitando así inconsistencias que podrían surgir si el sensor actualiza sus valores entre la lectura del byte alto y el byte bajo de una misma variable. De esta manera, se preserva la coherencia temporal de los datos adquiridos, lo cual es fundamental en aplicaciones que requieren precisión en el seguimiento de movimientos, como el control de actitud en un CubeSat.

Entre los registros principales está el denominado `WHO_AM_I`, utilizado específicamente para identificar el dispositivo en el bus I²C. Este registro, ubicado en la dirección `0x75`, contiene los 6 bits más significativos de la dirección I²C del MPU6050. Su valor por defecto es `0x68`, lo cual permite confirmar que el sensor está correctamente conectado y responde a la comunicación. Es importante destacar que el bit menos significativo de la dirección I²C se define por el nivel lógico del pin AD0, pero este bit no se refleja en el contenido

del registro WHO_AM_I. La lectura de este registro suele ser una de las primeras operaciones realizadas durante la inicialización del sensor, ya que garantiza que el dispositivo conectado es efectivamente un MPU6050 y que la interfaz de comunicación funciona correctamente.

Este módulo también cuenta con el registro CONFIG (registro 26, dirección 0x1A), el cual permite configurar, entre otras funciones, el comportamiento del filtro digital pasa bajos (DLPF, por sus siglas en inglés), aplicado tanto al giroscopio como al acelerómetro. Este filtro resulta útil para atenuar el ruido de alta frecuencia y mejorar la estabilidad de las mediciones.

La configuración del filtro se realiza mediante el campo DLPF_CFG, ubicado en los tres bits menos significativos (bits 2:0) del registro. Los valores posibles para este campo, junto con sus efectos sobre el ancho de banda y el retardo temporal, se presentan en la tabla 2. En dicha tabla, BP hace referencia a la banda pasante y Fs a la frecuencia de muestreo asociada.

DLPF_CFG	Acelerómetro			Giroscopio		
	BP (Hz)	Retardo (ms)	Fs (kHz)	BP (Hz)	Retardo (ms)	Fs (kHz)
0	260	0,0	1	256	0,98	8
1	184	2,0	1	188	1,9	1
2	94	3,0	1	98	2,8	1
3	44	4,9	1	42	4,8	1
4	21	8,5	1	20	8,3	1
5	10	13,8	1	10	13,4	1
6	5	19,0	1	5	18,6	1
7	Reservado		1	Reservado		8

CUADRO 2. Parámetros del filtro digital pasa bajos (DLPF) según el valor de DLPF_CFG

La banda pasante define el rango de frecuencias que el sensor permite pasar sin una atenuación significativa, es decir, determina cuán sensibles serán las mediciones a cambios rápidos. El retardo, por su parte, representa el tiempo que demora el sensor en reflejar una variación real de la señal debido al proceso de filtrado.

En el contexto de una misión espacial, la configuración adecuada del filtro digital pasa bajos (DLPF) es especialmente relevante debido a las exigencias de precisión y robustez en la adquisición de datos inerciales. Durante las etapas de calibración en tierra, donde el entorno es estático y se busca minimizar el ruido, es recomendable emplear configuraciones del DLPF con banda pasante baja (por ejemplo, DLPF_CFG = 5 o 6), ya que permiten filtrar eficazmente el ruido de alta frecuencia, mejorando la estabilidad de las mediciones sin comprometer el rendimiento dinámico. En cambio, durante la operación en órbita, particularmente en tareas como el control de actitud o la estimación de orientación mediante sensores inerciales, se requiere un compromiso entre respuesta dinámica y supresión de ruido. En estos casos, es común seleccionar configuraciones intermedias (como DLPF_CFG = 2 o 3), que mantienen una sensibilidad adecuada a movimientos rápidos del satélite sin introducir retardos excesivos en la respuesta.

Por otra parte, el sensor permite configurar el rango completo de medición tanto del acelerómetro como del giroscopio mediante dos registros clave: GYRO_CONFIG (registro 27, dirección 0x1B) y ACCEL_CONFIG (registro 28, dirección 0x1C). El registro GYRO_CONFIG permite establecer el rango de escala completa del giroscopio a través del campo FS_SEL, ubicado en los bits 4 y 3. Este campo de 2 bits determina el rango máximo de velocidad angular (en grados por segundo) que puede medir el sensor en cada eje. Las opciones disponibles se muestran en la tabla 3

FS_SEL	Rango (°/s)
0	± 250
1	± 500
2	± 1000
3	± 2000

CUADRO 3. Definición del giroscopio

De manera similar, el registro `ACCEL_CONFIG` controla el rango de medición del acelerómetro mediante el campo `AFS_SEL`, también ubicado en los bits 4 y 3. Este campo define el rango de aceleración (en unidades de gravedad, g) que puede detectar el sensor. Las opciones disponibles se muestran en la tabla 4

AFS_SEL	Rango
0	$\pm 2\ g$
1	$\pm 4\ g$
2	$\pm 8\ g$
3	$\pm 16\ g$

CUADRO 4. Definición del acelerómetro

Finalmente, el MPU6050 incluye mecanismos para la administración del consumo energético y el control de su estado interno, centralizados principalmente en el registro `PWR_MGMT_1` (registro 107, dirección `0x6B`). Este registro permite seleccionar la fuente de reloj del dispositivo, activar modos de bajo consumo, deshabilitar el sensor de temperatura y realizar un reinicio completo del módulo. Dentro de este registro, se encuentran los bits:

- `DEVICE_RESET` (bit 7): permite reiniciar todos los registros del sensor a sus valores por defecto. Este bit se borra automáticamente una vez completado el proceso de reinicio
- `SLEEP` (bit 6): pone al dispositivo en un modo de bajo consumo, desactivando internamente la mayoría de los circuitos.
- `CYCLE` (bit 5): habilita el modo de ciclo, donde el sensor alterna entre estados de reposo y breves activaciones para tomar muestras del acelerómetro, con una frecuencia determinada por el registro `LP_WAKE_CTRL` (registro 108).
- `TEMP_DIS` (bit 3): permite desactivar el sensor de temperatura interno que tiene la unidad (no es precisa para mediciones ambientales pero se utiliza para ajustar las mediciones del sensor)
- `CLKSEL[2:0]` (bits 2:0): selecciona la fuente de reloj principal del MPU6050. Las opciones disponibles se observan en la tabla 5

CLKSEL	Fuente de reloj
0	Oscilador interno de 8 MHz
1	PLL con referencia en el giroscopio del eje X
2	PLL con referencia en el giroscopio del eje Y
3	PLL con referencia en el giroscopio del eje Z
4	PLL con referencia externa de 32,768 kHz
5	PLL con referencia externa de 19,2 MHz
6	Reservado
7	Detiene el reloj y congela el funcionamiento interno

CUADRO 5. Fuente de reloj del MPU6050

AT24C256

El módulo AT24C256 incorpora dos resistencias *pull-up* de 10 $k\Omega$ conectadas a las líneas SCL y SDA. Presenta cuatro pines accesibles: VCC, GND, SCL y SDA, aunque, internamente, el chip posee ocho pines: `A0` y `A1` (permiten configurar los bits menos significativos de la dirección I^2C; en el módulo están conectados a GND), NC (sin conexión, atado a GND), GND, VCC, WP (protección contra escritura, desactivada al estar conectada a GND), y finalmente las líneas SCL y SDA.

La dirección del dispositivo está dada por el patrón binario `10100A_1A_0`. Como `A1` y `A0` están a GND, la dirección final es `1010000`, es decir, `0x50` en hexadecimal.

Durante las transmisiones I^2C, la EEPROM responde con un bit de reconocimiento (ACK) en el noveno ciclo de reloj luego de recibir correctamente una dirección o un byte de datos. También entra automáticamente en modo de bajo consumo (*stand-by*) al encender o tras finalizar una operación interna seguida de una condición de STOP.

Si el protocolo I^2C se interrumpe (por ejemplo, debido a un corte de energía), la EEPROM puede recuperarse mediante la generación de hasta nueve pulsos de reloj con SDA en alto y luego una condición de inicio.

Operación de escritura

La **escritura de un byte** consiste en enviar primero la dirección de memoria de 2 bytes (MSB primero), luego el byte de datos, y finalizar con una condición de STOP para iniciar el ciclo interno de escritura (t_{WR}), durante el cual la EEPROM no responderá a comandos. Esta operación puede observarse en la figura 15

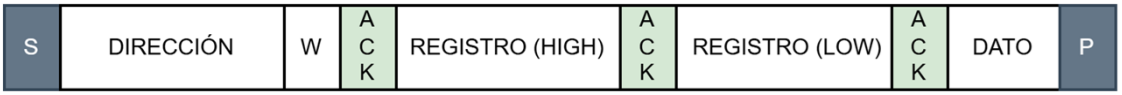

FIGURA 15. Escritura de un byte

Para **escribir una página completa**, se sigue el mismo procedimiento, pero enviando todos los datos en una misma comunicación, recibiendo el ACK correspondiente a cada byte. El maestro debe finalizar la operación con una condición de STOP. Si se envían más de 64 bytes, la dirección interna se desborda y los datos comienzan a sobrescribirse desde el inicio de la página actual. Esta operación puede observarse en la figura 16

FIGURA 16. Escritura secuencial de una página

Para saber si una operación de escritura ha finalizado, se puede utilizar el método de *Acknowledge Polling*: se envía una condición de inicio y la dirección del dispositivo; si responde con un ACK, la escritura interna terminó. Esta operación puede observarse en la figura 17, donde la EEPROM aparece disponible al tercer intento del *polling*.

FIGURA 17. *Acknowledge Polling*

Operación de lectura

Las operaciones de lectura pueden realizarse de tres formas. En la **lectura de dirección actual**, se lee el byte ubicado en la última dirección accedida, incrementada en uno, sin necesidad de enviar dirección de memoria. Este valor se mantiene válido mientras el chip esté alimentado. La lectura se inicia enviando la dirección del dispositivo con el bit de lectura en alto, y la EEPROM devuelve el dato. El maestro debe terminar la lectura con un STOP (sin ACK).

La **lectura aleatoria** permite acceder a una dirección específica de memoria: primero se envía la dirección del dispositivo en modo escritura, seguida de los dos bytes de dirección de palabra deseada. Luego se emite una condición de RESTART, ahora en modo lectura, y la EEPROM devuelve el byte correspondiente. Al igual que antes, el maestro debe finalizar con STOP. Esta operación puede observarse en la figura 18

La **lectura secuencial** permite extraer múltiples bytes contiguos. Puede iniciarse mediante una lectura de dirección actual o una lectura aleatoria. Tras recibir un byte, el maestro responde con un ACK, y la EEPROM continúa transmitiendo datos incrementando la dirección interna. Si se alcanza el límite de memoria, la dirección se reinicia desde cero, permitiendo una lectura circular. La secuencia termina cuando el maestro deja de enviar ACK y emite una condición de STOP. Esta operación puede observarse en la figura 19

I. Detalles de implementación de SPI en STM32

I.1. Análisis de la función `spi_transmit_receive`

La función `spi_transmit_receive()` permite realizar una operación de transmisión y recepción simultánea (*full-duplex*) sobre un periférico SPI, utilizando los registros de bajo nivel provistos por `libopencm3`. A continuación, se describe su funcionamiento detallado.

FIGURA 18. Lectura de un byte

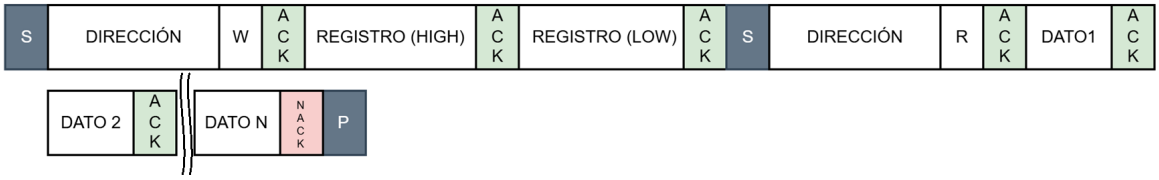

FIGURA 19. Lectura secuencial de una página

```
spi_status_t spi_transmit_receive(uint32_t spi_id,
                                  uint8_t *txdata,
                                  uint16_t *rxdata,
                                  uint16_t size,
                                  TickType_t xTicksToWait);
```

CÓDIGO 10. Prototipo de la función `spi_transmit_receive`

Parámetros de entrada:

- `spi_id`: Identificador del periférico SPI (por ejemplo, `SPI1`).
- `txdata`: Puntero a los datos que serán transmitidos.
- `rxdata`: Puntero donde se almacenarán los datos recibidos.
- `size`: Cantidad de palabras a transmitir/recibir.
- `xTicksToWait`: Tiempo máximo de espera (*timeout*) en *ticks* de FreeRTOS.

La función implementa una transmisión SPI en modo maestro, con soporte para tamaños de palabra configurables y verificación opcional por CRC. A continuación se describe su comportamiento paso a paso, con bloques de código intercalados.

1. Se obtiene un puntero a la estructura `spi_t` asociada al periférico:

```
spi_t *spi = get_spi(spi_id);
if (spi == NULL){
    SPI_ERROR_HANDLER("Error: SPI_ID no valido o estructura SPI no
   definida en spi_transmit_receive.\r\n");
    return SPI_NOT_FOUND;
}
```

   Esta estructura contiene información del estado del SPI, tamaño de datos, mutex y configuración asociada.

2. Si el uso de CRC está habilitado, se espera que no haya transferencia activa:

```
if(crc_enabled == true){
    TickType_t start = xTaskGetTickCount();
    while (SPI_SR(spi_id) & SPI_SR_BSY) {
        if ((xTaskGetTickCount() - start) > WAIT_BSY_TIMEOUT) {
            SPI_ERROR_HANDLER("Error: Timeout esperando SPI_SR_BSY
   en spi_transmit\r\n");
            return SPI_TIMEOUT_ERROR;
        }
    }
    spi_enable_crc(spi_id); //Habilito el CRC
    spi_set_next_tx_from_buffer(spi_id); //Habilito la transmision
   desde el buffer
}
```

Esto garantiza que el motor de CRC arranque desde un estado limpio.

3. Se inicializa el contador para gestionar los *timeouts* de transmisión y recepción:

```
TickType_t start = xTaskGetTickCount();
```

4. Se evalúa el tamaño de datos para determinar si se trata de una transmisión de 8 o 16 bits:

```
if (spi->data_size == DATA_SIZE_8){
    uint8_t *rxdata8 = (uint8_t *)rxdata;
    ...
}
else if (spi->data_size == DATA_SIZE_16){
    uint16_t *txdata16 = (uint16_t *)txdata;
    ...
}
```

5. Dentro de cada caso se recorre el buffer de datos. En cada iteración:

   a) Se espera que el buffer de transmisión esté libre:

```
while (!(SPI_SR(spi_id) & SPI_SR_TXE)) {
    if ((xTaskGetTickCount() - start) > xTicksToWait) {
        SPI_ERROR_HANDLER("Timeout esperando SPI_SR_TXE en
   spi_transmit_receive\r\n");
        return SPI_TIMEOUT_ERROR;
    }
}
```

   b) Se escribe el dato a transmitir:

```
SPI_DR(spi_id) = txdata[i];
```

   c) Si se trata del último dato y CRC está habilitado, se notifica que el próximo dato será el CRC:

```
if(i == (size - 1) && crc_enabled)
    spi_set_next_tx_from_crc(spi_id);
```

   d) Se espera la llegada del dato recibido y se almacena:

```
while (!(SPI_SR(spi_id) & SPI_SR_RXNE)){
    if ((xTaskGetTickCount() - start) > xTicksToWait) {
        SPI_ERROR_HANDLER("Timeout esperando SPI_SR_RXNE en
   spi_transmit_receive\r\n");
        return SPI_TIMEOUT_ERROR;
    }
}
rxdata[i] = SPI_DR(spi_id);
```

6. Una vez terminada la transmisión, si se usó CRC, se valida el resultado:

```
if(crc_enabled){
    while (!(SPI_SR(spi_id) & SPI_SR_TXE)) {
        if ((xTaskGetTickCount() - start) > xTicksToWait) {
            SPI_ERROR_HANDLER("Timeout esperando SPI_SR_TXE en
   spi_transmit\r\n");
```

```
            return SPI_TIMEOUT_ERROR;
        }
    }

    while (SPI_SR(spi_id) & SPI_SR_BSY) {
        if ((xTaskGetTickCount() - start) > xTicksToWait) {
            SPI_ERROR_HANDLER("Error: Timeout esperando SPI_SR_BSY
    en spi_transmit\r\n");
            return SPI_TIMEOUT_ERROR;
        }
    }

    if (SPI_SR(spi_id) & SPI_SR_CRCERR) {
        SPI_SR(spi_id) &= ~SPI_SR_CRCERR;
        SPI_ERROR_HANDLER("Error: Error de CRC en spi_transmit\r\n")
    ;

        spi_disable(spi_id);
        spi_disable_crc(spi_id);
        spi_enable_crc(spi_id);
        spi_enable(spi_id);

        return SPI_CRC_ERROR;
    }
```

7. Independientemente del resultado del CRC, se limpia y reinicializa el motor CRC como se indica en [23], para futuras transmisiones:

```
    spi_disable(spi_id);
    spi_disable_crc(spi_id);
    spi_enable_crc(spi_id);
    spi_enable(spi_id);
}
```

8. Finalmente, se retorna `SPI_PASS` indicando éxito:

```
return SPI_PASS;
```

Puntos destacados de robustez:

- Validación del identificador SPI y estructura interna al inicio.
- Control exhaustivo de timeout tanto en transmisión como en recepción.
- Manejo completo de errores de CRC con reinicialización segura.
- Implementación genérica para transferencias de 8 y 16 bits.

J. Monitor de recursos

Este anexo complementa los resultados presentados en el capítulo principal, y detalla las configuraciones de memoria y prioridades utilizadas en el monitor de recursos desarrollado.

Se documentan comparaciones entre tareas dinámicas y estáticas, variaciones en el uso del *heap*, fragmentación, y comportamiento en condiciones de carga. Asimismo, se explican los criterios adoptados para definir las prioridades de cada tarea y su impacto sobre la ejecución concurrente.

J.1. Uso de memoria

Con el objetivo de optimizar el uso de memoria en el sistema operativo embebido y entender las implicancias prácticas de los distintos métodos de asignación de tareas, se llevaron a cabo una serie de pruebas comparativas entre versiones con tareas completamente dinámicas, parcialmente estáticas y completamente estáticas (salvo una excepción funcional).

El sistema se basa en múltiples tareas concurrentes que acceden a sensores, gestionan comunicaciones y almacenan datos. Como se explicó en la sección 2.3, cada una de estas tareas requiere una reserva de *stack*, la cual se especifica al momento de su creación. La correcta asignación de memoria resulta crítica debido a la limitada cantidad de SRAM disponible (20 KB en el STM32F103C8T6).

La primera etapa consistió en configurar todas las tareas del sistema con asignación dinámica. A cada una se le otorgó una cantidad de memoria considerablemente superior a la estimada, a modo preventivo. Luego, mediante la función `uxTaskGetStackHighWaterMark()`, se evaluó cuánto *stack* quedaba sin utilizar en ejecución normal, y a partir de esa información se calculó un nuevo tamaño de pila con un margen de seguridad del 30 %, redondeado hacia arriba, pudiendo observarse los valores en la tabla 6. Esta nueva configuración con márgenes generosos se volvió a probar, confirmando que el sistema se comportaba correctamente.

TAREA	ASIGNADO	LIBRE	USADO	CON MARGEN
MPU	440	190	210	273
HTU	330	154	176	229
GPS	256	82	174	227
SDWRITER	380	138	242	315
PROCESS	240	142	98	128
BLINK	60	38	22	29
INIT_TASK	90	44	46	60
UART	100	60	40	52
MONITOR	300	164	136	177
FREERTOS HEAP				1144
FREERTOS MINIMUM HEAP				672
SUMA BLOQUES LIBRES				1144
BLOQUE LIBRE MÁS LARGO				672
BLOQUE LIBRE MÁS CORTO				472
N° DE BLOQUES LIBRES				2
MALLOCS EXITOSOS				35
FREES EXITOSOS				2
`configTOTAL_HEAP_SIZE (*1024)`				17

CUADRO 6. Asignación completamente dinámica con márgenes amplios.

Posteriormente, se introdujo una versión intermedia donde todas las tareas seguían siendo dinámicas, salvo el monitor de memoria, que se convirtió a tarea estática. Esta decisión buscó evitar que el propio proceso de monitoreo afectara las métricas que se intentaban observar. Dado que la memoria *heap* de FreeRTOS estaba ocupada por el resto de las tareas dinámicas, fue necesario reducir `configTOTAL_HEAP_SIZE` de 17 KB a 15 KB para que todo pudiera coexistir sin errores de asignación. Los resultados pueden observarse en la tabla 7.

En una tercera fase, se implementó una versión completamente estática, en la cual todas las tareas fueron creadas mediante *buffers* de *stack* y TCB preasignados. En este caso se volvió

TAREA	ASIGNADO	LIBRE	USADO
MPU	273	64	209
HTU	229	54	175
GPS	227	54	173
SDWRITER	315	74	241
PROCESS	128	30	98
BLINK	29	8	21
INIT_TASK	60	14	46
UART	52	12	40
MONITOR	177	40	137
FREERTOS HEAP			3928
FREERTOS MINIMUM HEAP			3576
SUMA BLOQUES LIBRES			3928
BLOQUE LIBRE MÁS LARGO			3576
BLOQUE LIBRE MÁS CORTO			352
N° DE BLOQUES LIBRES			2
MALLOCS EXITOSOS			28
FREES EXITOSOS			2
`configTOTAL_HEAP_SIZE (*1024)`			15

CUADRO 7. Tareas dinámicas con excepción del monitor (estático).

a subir el tamaño del heap a 5 KB, no para almacenar los stacks de las tareas —que ahora son completamente estáticos—, sino como ejemplo de coexistencia entre ambos métodos de asignación. Los resultados se observan en la tabla 8

Durante estas pruebas se observaron varias diferencias relevantes. En la versión completamente dinámica, el uso total de SRAM informado por el binario no varió a pesar de los ajustes en las tareas, ya que FreeRTOS reserva de forma fija el tamaño completo del *heap* desde el arranque. En cambio, al pasar a una versión estática, el uso de memoria global sí disminuyó visiblemente, al eliminar la necesidad de *heap* para la creación de tareas. También se evidenció una mejora en la fragmentación: mientras que en la versión dinámica había dos bloques libres de memoria, en la estática solo uno, lo cual indica menor fragmentación del *heap*. Aunque este valor es bajo en ambos casos, muestra el beneficio estructural del enfoque estático en sistemas que buscan previsibilidad.

Otro punto observado fue que en todos los casos hubo llamadas exitosas a *malloc* (al menos diez), correspondientes al *driver* de la tarjeta SD. Además, en la versión donde solo una tarea era dinámica, se registraron dos *free* adicionales, correspondientes a la liberación de esa tarea tras su autoeliminación (uno por su *stack* y otro por el TCB).

Finalmente, se modificó la creación de `task_initial_enable_connect` a su versión dinámica, ya que, al realizar una autodestrucción (`vTaskDelete`) al terminar su tarea, requiere liberar correctamente la memoria. Además, se simuló el uso intensivo del sistema al realizar múltiples descargas de archivos desde la tarjeta SD. En estas pruebas se detectó que la tarea encargada de procesar los comandos de tierra (`taskProcessMsg`) era la que más incrementaba su uso de *stack*. Aun así, los márgenes definidos permitieron que se mantuviera dentro de los límites seguros. Se remarca aquí que los valores de *stack* libre reportados en las tablas corresponden al estado más exigente alcanzado durante las pruebas, aunque no necesariamente al peor caso absoluto. Esto se aclara para destacar que, si bien los márgenes son generosos, no garantizan cobertura en todos los escenarios posibles, y el dimensionamiento debe contemplar un margen razonable. Los valores obtenidos pueden observarse en

TAREA	ASIGNADO	LIBRE	USADO	CON MARGEN
MPU	400	190	210	273
HTU	330	154	176	229
GPS	256	82	174	227
SDWRITER	380	138	242	315
PROCESS	240	74	166	216
BLINK	60	38	22	29
INIT_TASK	90	44	46	60
UART	100	60	40	52
MONITOR	160	24	136	177
FREERTOS HEAP				2216
FREERTOS MINIMUM HEAP				2216
SUMA BLOQUES LIBRES				2216
BLOQUE LIBRE MÁS LARGO				2216
BLOQUE LIBRE MÁS CORTO				2216
N° BLOQUES LIBRES				1
MALLOCS EXITOSOS				10
FREES EXITOSOS				0
`configTOTAL_HEAP_SIZE (*1024)`				5

CUADRO 8. Asignación completamente estática con reserva mínima de heap.

la tabla 9

Las pruebas realizadas demuestran que el uso de tareas estáticas, con márgenes adecuados de pila y una pequeña reserva dinámica para estructuras auxiliares, permite reducir el uso total de memoria RAM, mejorar la previsibilidad del sistema y evitar problemas de fragmentación. La elección de una única tarea dinámica se justificó por motivos funcionales específicos, y el resto del sistema se mantuvo robusto bajo diferentes condiciones de carga.

Con el objetivo de analizar el impacto real de las distintas estrategias de asignación de memoria, se elaboró una tabla resumen comparando las configuraciones evaluadas (tabla 10. En primer lugar, se observa que el segmento TEXT, correspondiente al código ejecutable, se mantiene prácticamente constante entre todas las configuraciones. Esto confirma que los cambios en la modalidad de asignación (estática vs. dinámica) no afectan el tamaño del binario generado en términos de código.

Por otro lado, el uso de memoria SRAM, reflejado en los segmentos DATA y BSS, sí presenta diferencias significativas. En particular, se destaca que en las configuraciones con asignación totalmente dinámica (con o sin márgenes), el valor de BSS permanece constante, incluso si las tareas utilizan menos memoria en la práctica. Esto se debe a que FreeRTOS, al usar `heap_4.c`, reserva de antemano la cantidad completa de memoria especificada en `configTOTAL_HEAP_SIZE`, lo que se refleja como un aumento artificial en BSS.

Recién en las versiones con asignación estática se observan reducciones en el uso de SRAM, ya que las tareas se implementan sin necesidad de reservar espacio dinámico adicional para sus *stacks* ni TCBs, y el *heap* se puede reducir drásticamente sin afectar el funcionamiento del sistema. En el caso de la versión final, donde todas las tareas son estáticas salvo `INIT_TASK`, la memoria SRAM utilizada es la más baja de todas las configuraciones probadas, sin comprometer funcionalidad ni estabilidad. Esta configuración aprovecha las

[1]Luego de varias descargas queda en 49.

[2]Luego de varias descargas queda en 75.

TAREA	ASIGNADO	LIBRE	USADO
MPU	270	63	207
HTU	228	54	174
GPS	225	54	171
SDWRITER	324	74	250
PROCESS	124	119[1]	5[2]
BLINK	28	7	21
INIT_TASK	59	14	45
UART	51	13	38
MONITOR	176	41	135
FREERTOS HEAP			2208
FREERTOS MINIMUM HEAP			1864
SUMA BLOQUES LIBRES			2208
BLOQUE LIBRE MÁS LARGO			1864
BLOQUE LIBRE MÁS CORTO			1864
N° BLOQUE LIBRES			2
MALLOCS EXITOSOS			10
FREES EXITOSOS			2
`configTOTAL_HEAP_SIZE (*1024)`			5

CUADRO 9. Configuración final: tareas estáticas con INIT_TASK dinámica.

ventajas de la asignación estática, permitiendo dejar disponible memoria que de otro modo quedaría inutilizada por reserva anticipada.

CONFIGURACIÓN	TEXT	DATA	BSS	SRAM (DATA+BSS)
TODO ESTÁTICO	34196	140	20048	20188
TODO ESTÁTICO (CON MARGEN)	34192	140	18296	18436
TODO DINÁMICO	33908	140	19424	19564
TODO DINÁMICO (CON MARGEN)	33900	140	19424	19564
TODO DINÁMICO (menos monitor)	34148	140	19540	19680
FINAL	34332	140	17972	18112

CUADRO 10. Comparación de uso de memoria SRAM y tamaño de código entre configuraciones.

J.2. Prioridades

En la versión final del sistema se definió un esquema de prioridades acorde a los requerimientos de funcionamiento y depuración. La tarea encargada de inicializar la conexión I^2C recibió la prioridad más alta (5), ya que su correcta finalización resulta crítica para el funcionamiento de la mayoría de los sensores utilizados. También se asignó alta prioridad a la tarea encargada de encender y pagar el LED de estado (5), ya que su ejecución periódica permite una inspección visual del estado del sistema —una interrupción prolongada en su parpadeo es indicio de bloqueo o falla grave.

A la tarea encargada de transmitir los datos por UART se le otorgó una prioridad inmediatamente inferior (4), dado que es esencial para la salida de datos por UART, interfaz clave

para la comunicación con tierra y la depuración. Las tareas relacionadas con adquisición de datos (MPU, HTU, GPS) también fueron priorizadas con nivel 4 o 3, según la criticidad relativa de cada sensor y su frecuencia de actualización.

La tarea encargada del almacenamiento persistente en la tarjeta SD fue ubicada en una prioridad baja (2), dado que su operación no es crítica en tiempo real y puede diferirse sin impacto funcional inmediato. Finalmente, la tarea Monitor —destinada exclusivamente a la inspección del uso de memoria durante pruebas— fue ubicada con la menor prioridad (1), evitando interferencias con el funcionamiento principal del firmware. Cabe destacar que en la versión final esta tarea fue desactivada, al haberse cumplido su propósito durante la etapa de ajuste fino del sistema.

Bibliografía

[1] NASA CubeSat Launch Initiative. *CubeSat 101: Basic Concepts and Processes for First-Time CubeSat Developers*. For Public Release – Revision Dated October 2017. National Aeronautics y Space Administration (NASA). 2017. URL: https://www.nasa.gov/content/cubesat-launch-initiative.

[2] European Space Agency. *CubeSats*. https://www.esa.int/Enabling_Support/Preparing_for_the_Future/Discovery_and_Preparation/CubeSats. s.f.

[3] Y. Lu et al. «A Review of the Space Environment Effects on Spacecraft in Different Orbits». En: *IEEE Access* (2019). DOI: 10.1109/ACCESS.2019.2927811.

[4] National Aeronautics and Space Administration. *State-of-the-Art of small spacecraft technology*. https://www.nasa.gov/smallsat-institute/sst-soa/small-spacecraft-avionics/. 2025.

[5] H. Akah et al. «Total Ionizing Dose Effects on Commercial ARM Microcontroller for Low Earth Orbit Satellite Subsystems». En: *Aerospace Sciences & Aviation Technology*. ASAT-17-133-US. 2017.

[6] K. Stewart. *Órbita terrestre baja*. Enciclopedia Británica. 18 de abr. de 2025. URL: https://www.britannica.com/technology/low-Earth-orbit.

[7] W. J. Larson y J. R. Wertz. *Space Mission Analysis and Design*. 3.ª ed. Microcosm, Inc., 2005.

[8] L. T. Lumbwe. «Development of an onboard computer (OBC) for a CubeSat». Thesis. Cape Peninsula University of Technology, Faculty of Engineering, Bellville, 2013.

[9] B. Sheela Rani et al. «A survey to select microcontroller for Sathyabama satellite's On Board Computer subsystem». En: *RSTS & CC-2010*. Chennai, India, 2010, págs. 134-137. DOI: 10.1109/RSTSCC.2010.5712831.

[10] STMicroelectronics. *STM32F103x8/B datasheet – ARM Cortex-M3 32-bit MCU*. CD00161566 Rev 17. 2016.

[11] libopencm3. *Readme*. https://github.com/libopencm3/libopencm3. s.f.

[12] Warren Gay. *Beginning STM32: Developing with FreeRTOS, libopencm3 and GCC*. Apress, 2018.

[13] S. Kelapure. «Design of a Software Architecture for Supervisor System in a Satellite». Master's thesis. Institute for Architecture of Application Systems (IAAS), 2020.

[14] R. Barry. *Mastering the FreeRTOS real-time kernel*. Vol. 1. Real Time Engineers Ltd., 2016.

[15] Christopher Svec. «The Architecture of Open Source Applications, Volumen 2». En: aosabook.org. URL: https://aosabook.org/en/v2/freertos.html.

[16] Oracle. *¿Qué es Docker?* n.d. URL: https://www.oracle.com/ar/cloud/cloud-native/container-registry/what-is-docker/.

[17] Docker. *Dockerfile reference*. n.d. URL: https://docs.docker.com/reference/dockerfile/.

[18] GitLab. *¿Qué es la CI/CD?* n.d. URL: https://about.gitlab.com/es/topics/ci-cd/.

[19] GitLab. *Download and install GitLab*. n.d. URL: https://about.gitlab.com/install/#raspberry-pi-os.

[20] Raspberry Pi Ltd. *Raspberry Pi 5*. 2025. URL: https://datasheets.raspberrypi.com/rpi5/raspberry-pi-5-product-brief.pdf.

[21] pytest. *pytest documentation*. n.d. URL: https://docs.pytest.org/en/stable/.

[22] Analog Devices. «UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter». En: (2020).

[23] STMicroelectronics. *STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced Arm®-based 32-bit MCUs - Reference manual*. Rev 21. RM0008. 2021. URL: https://www.st.com/resource/en/reference_manual/cd00171190.pdf.

[24] Abhishek Prasad et al. «Interfacing Architecture between Telemetry and On-Board Computer for a Nanosatellite». En: *2020 IEEE Aerospace Conference*. Team Anant, BITS Pilani, Rajasthan, India. IEEE. 2020.

[25] B. Rajulu, S. Dasiga y N. R. Iyer. «Open Source RTOS Implementation for On-Board Computer (OBC) in STUDSAT-2». En: *IEEE Aerospace Conference* (2014).

[26] Alif Rachman Harfian et al. «An Investigation of RTOS-Based Sensor Data Management Performance for Tel-USat On Board Data Handling (OBDH) Subsystem». En: *2019 International Conference on Information and Communications Technology (ICOIACT)*. IEEE, 2019, págs. 634-638. DOI: 10.1109/ICOIACT46704.2019.8938499.

[27] Samuel F. Hishmeh, Tyler J. Doering y James E. Lumpp. «Design of flight software for the KySat CubeSat bus». En: *2009 IEEE Aerospace conference*. 2009, págs. 1-15. DOI: 10.1109/AERO.2009.4839646.

[28] M. Unwin et al. «Design and Implementation of Small Satellite Navigation Experiments». En: *Acta Astronautica* 35.11 (1995), págs. 633-641.

[29] P. Kovar. «Experiences with the GPS in Unstabilized CubeSat». En: *International Journal of Aerospace Engineering* (2020). DOI: 10.1155/2020/8894984.

[30] u-blox AG. *u-blox 6 Receiver Description Including Protocol Specification*. Document Number: GPS.G6-SW-10018-F, Revision for FW 7.03. 2013.

[31] NXP Semiconductors. *I2C-bus specification and user manual*. UM10204 Rev. 7.0. 2021.

[32] A. AlBalooshi. «Fault analysis of the I2C bus in CubeSats and proposed mitigation techniques». Khalifa University, 2020.

[33] J. Bouwmeester, M. Langer y E. Gill. «Survey on the implementation and reliability of CubeSat electrical bus interfaces». En: *CEAS Space Journal* (2016). DOI: 10.1007/s12567-016-0138-0.

[34] libopencm3. *I2C peripheral API*. https://libopencm3.org/docs/latest/stm32f2/html/group__i2c__file.html. s.f.

[35] Texas Instruments. *PCF8574 Remote 8-Bit I/O Expander for I2C Bus*. https://www.ti.com/lit/ds/symlink/pcf8574.pdf. SCPS068K. 2024.

[36] Texas Instruments. *SN74HC4066 Quadruple Bilateral Analog Switch*. https://www.ti.com/lit/ds/symlink/sn74hc4066.pdf. SCLS325K. 2024.

[37] Microchip Technology Inc. *MegaAVR® Data Sheet*. DS40002061A. 2018.

[38] TE connectivity sensors. *HTU21D(F) RH/T SENSOR IC*. 2017.

[39] Walt Kester Harrison. *Introduction to SPI Interface*. Analog Dialogue, Analog Devices. 2006. URL: https://www.analog.com/en/resources/analog-dialogue/articles/introduction-to-spi-interface.html.

[40] Winbond Electronics Corporation. *W25Q32JV – 32M-BIT Serial Flash Memory with Dual/Quad SPI*. Rev. F. 2018.

[41] C. Chan. *FatFs - Generic FAT File System Module*. http://elm-chan.org/fsw/ff/. s.f.

[42] CaliBeta. *MicroSD-STM32*. https://github.com/CaliBeta/MicroSD-STM32/tree/master. [Repositorio GitHub]. s.f.

[43] GitLab. *Deprecation - Support for registration tokens and server-side runner configuration parameters in `gitlab-runner register` command (Issue #380872)*. Consultado el 7 de mayo de 2025. n.d. URL: https://gitlab.com/gitlab-org/gitlab/-/issues/380872.

[44] GitLab. *Registering runners*. n.d. URL: https://docs.gitlab.com/runner/register/.

[45] GitLab. *Get started with GitLab CI/CD | GitLab Docs*. n.d. URL: https://docs.gitlab.com/ci/.

[46] Raspberry Pi Foundation. *Raspberry Pi Software*. n.d. URL: https://www.raspberrypi.com/software/.

[47] A. Whittaker. *Coding on Raspberry Pi remotely with Visual Studio Code*. Raspberry Pi. 17 de feb. de 2021. URL: https://www.raspberrypi.com/news/coding-on-raspberry-pi-remotely-with-visual-studio-code/.

[48] ve3wwg. *stm32f103c8t6/stm32f103c8t6.ld at master · ve3wwg/stm32f103c8t6*. https://github.com/ve3wwg/stm32f103c8t6/blob/master/stm32f103c8t6.ld. GitHub. n.d.

[49] STMicroelectronics. *STM32 microcontroller system memory boot mode*. Inf. téc. AN2606 Rev 66. 2025. URL: https://www.st.com/resource/en/application_note/an2606-stm32-microcontroller-system-memory-boot-mode-stmicroelectronics.pdf.

[50] STMicroelectronics. *USART protocol used in the STM32 bootloader*. Inf. téc. AN3155 Rev 19. 2025. URL: https://www.st.com/resource/en/application_note/an3155-usart-protocol-used-in-the-stm32-bootloader-stmicroelectronics.pdf.

[51] InvenSense. *MPU-6000 and MPU-6050 Product Specification*. Rev. 3.4. 2013.

[52] InvenSense. *MPU-6000 and MPU-6050 Register map and descriptions*. Rev. 4.2. 2013.