Instituto Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Laboratorio de Sistemas Espaciales (SETEC Lab)

Implementación de un sistema multiagente embebido para la validación de algoritmos de misión crítica en la arquitectura ATmega32.

Informe de Trabajo Final de Graduación para optar por el titulo de Ingeniero en Electrónica con el grado académico de Licenciatura

Oscar Fernández Zúñiga

Cartago, 26 de noviembre del 2025

Declaratoria de Autenticidad

Yo, Oscar Fernández Zúñiga, estudiante de Ingeniería en Electrónica en el Instituto Tecnológico de Costa Rica, declaro que el presente proyecto es un trabajo original de mi autoría exclusiva. Todo el contenido técnico, teórico y metodológico aquí presentado ha sido producto de mi investigación personal o, en los casos donde se han utilizado fuentes externas, estas han sido debidamente citadas y referenciadas conforme a las normas académicas vigentes.

Me comprometo formalmente a mantener los estándares de integridad académica, absteniéndome de cualquier forma de plagio o falsificación de información, en cumplimiento de los reglamentos institucionales.

Oscar Fernández Zúñiga
Cédula: 1-1780-0988
Cartago, Costa Rica
26 de noviembre del 2025

INSTITUTO TECNOLÓGICO DE COSTA RICA

ESCUELA DE INGENIERÍA ELECTRÓNICA

TRABAJO FINAL DE GRADUACIÓN

ACTA DE APROBACIÓN

Defensa del Trabajo Final de Graduación
Requisito para optar por el título de Ingeniero en Electrónica
Grado Académico de Licenciatura
Instituto Tecnológico de Costa Rica

El Tribunal Evaluador aprueba la defensa del Trabajo Final de Graduación denominado Implementación de un sistema multi-agente embebido para la validación de algoritmos de misión crítica en la arquitectura ATmega32, realizado por el señor Oscar Fernández Zúñiga y, hace constar que cumple con las normas establecidas por la Escuela de Ingeniería Electrónica del Instituto Tecnológico de Costa Rica.

Miembros del Tribunal Evaluador

JUAN JOSE MONTERO RODRIGUEZ (FIRMA)
Digitally signed by JUAN JOSE MONTERO RODRIGUEZ (FIRMA)
Date: 2025.11.28 08:31:24 -06'00'

Ing. Juan José Montero Rodríguez

Profesor lector

Juan Scott Chaves
JUAN SCOTT CHAVES NOGUERA (FIRMA)
2025.11.29 00:01:44 Z
2025.001.20937

Ing. Juan Scott Chaves Noguera

Profesor lector

JOHAN CARVAJAL GODINEZ (FIRMA)
Firmado digitalmente por JOHAN CARVAJAL GODINEZ (FIRMA)
Fecha: 2025.11.27 07:21:25 -06'00'

Ing. Johan Carvajal Godínez

Profesor asesor

Cartago, 26 de noviembre de 2025

Resumen

El presente proyecto desarrolla una plataforma embebida de validación para algoritmos de navegación autónoma mediante el uso del paradigma de sistemas multiagentes aplicados a un rover experimental. Esta plataforma surge como parte del proyecto ELANav del Laboratorio de Sistemas Espaciales (SETEC) del Instituto Tecnológico de Costa Rica, con el objetivo de establecer metodologías robustas, adaptables y redundantes para aplicaciones de misión crítica en exploración espacial.

El trabajo incluye la adaptación de la biblioteca FreeMAES para microcontroladores ATmega utilizando FreeRTOS, el diseño e implementación de múltiples agentes encargados de comunicación, percepción y control, y la creación de una aplicación de escritorio para el monitoreo y operación del rover. Los resultados demuestran la viabilidad del enfoque multiagente en sistemas embebidos con recursos limitados y su potencial para futuras aplicaciones en robótica y navegación espacial.

Palabras clave: sistemas multiagentes, sistemas embebidos, FreeRTOS, FreeMAES, navegación autónoma, rover, misión crítica.

Abstract

This project presents an embedded validation platform for autonomous navigation algorithms using the multi-agent systems paradigm applied to an experimental rover. The platform is developed as part of the ELANav project at the Space Systems Laboratory (SETEC) of the Costa Rica Institute of Technology, aiming to establish robust, adaptable, and redundant methodologies suitable for mission-critical space exploration systems.

The work includes adapting the FreeMAES library for ATmega microcontrollers using FreeRTOS, designing and implementing multiple agents responsible for communication, perception, and control, and developing a desktop application for monitoring and operating the rover. The results demonstrate the feasibility of the multi-agent approach in resource-constrained embedded systems and highlight its potential for future applications in robotics and space navigation.

Keywords: multi-agent systems, embedded systems, FreeRTOS, FreeMAES, autonomous navigation, rover, mission-critical applications.

Agradecimientos

Agradezco principalmente a mi familia: mi madre Niry, mi padre Oscar, mis hermanas María y Keilyn, mi cuñado Jimmy y mis sobrinos Gustavo y Karla, por todo el apoyo que me han dado a lo largo de los años. Especialmente durante mi tiempo de estudio, ya que a pesar de los desafíos y dificultades que aparecieron en el camino, siempre conté con su ayuda, paciencia y confianza.

Agradezco a mi grupo de amigos "Los Papos", con quienes he permanecido unido desde el colegio. Hemos compartido, crecido y nos hemos apoyado mutuamente, incluso cuando cada uno tomó rumbos distintos en la vida. También quiero agradecer a mis amigos de carrera, Jean Carlos y Kendall, con quienes enfrenté los retos académicos de la universidad y compartí grandes experiencias.

Extiendo mi agradecimiento a los profesores y profesoras del Instituto Tecnológico de Costa Rica, quienes han sido parte fundamental de mi formación profesional y han alimentado de forma constante mi interés por la ingeniería. Agradezco de manera especial al tribunal evaluador, en particular al profesor Johan Carvajal Godínez, por su acompañamiento, guía y apoyo durante el desarrollo de este proyecto.

Índice general

1. Introducción **10**

1.1. Antecedentes . . . 10
1.2. Definición del problema . . . 11
1.3. Síntesis del problema . . . 13
1.4. Meta . . . 13
1.5. Objetivo General . . . 13
1.6. Objetivos Específicos . . . 13
1.7. Enfoque Metodológico . . . 14
1.8. Estructura de la Tesis . . . 15

2. Marco Teórico **16**

2.1. Conceptos Fundamentales . . . 16
2.2. Revisión Bibliográfica . . . 18
2.3. Estado del Arte . . . 19

3. Diseño de una aplicación para el control del GalaxyRVR **21**

3.1. ControlRVR App . . . 21
3.2. Diseño de la aplicación . . . 23
3.2.1. Clase RoverClient . . . 23
3.2.2. Clase VideoThread . . . 24
3.2.3. Clase ManualPage . . . 25
3.2.4. Clase SystematicPage . . . 26
3.2.5. Clase MonitorPage . . . 26
3.3. Resultados Obtenidos . . . 27
3.4. Reflexiones Finales . . . 29

4. Diseño del sistema multi-agentes **31**

4.1. Fase de Análisis . . . 32
4.2. Fase de Diseño . . . 38
4.3. Arquitectura física . . . 39
4.4. Adaptación de la librería FreeMAES a la familia AVR de Arduino . . . 42
4.5. Rediseño del sistema . . . 47
4.6. Resultados Obtenidos . . . 50
4.7. Reflexión Final . . . 51

5. Implementación Definitiva **53**
5.1. Repositorio *GalaxyRVR Control App* . . . 53
5.2. Repositorio *FreeMAES_Sim* . . . 54
5.3. Repositorio *FreeMAES for ATmega Devices* . . . 55

6. Evaluación de FreeRTOS + FreeMAES **57**
6.1. Aplicaciones de prueba . . . 57
6.2. Tamaño del programa (Flash Memory y SRAM) . . . 58
6.3. Ciclos de CPU en estado Idle . . . 58
6.4. Consumo máximo de Stack . . . 59
6.5. GalaxyRVR original vs GalaxyRVR con agentes . . . 60
6.6. Comparaciones Adicionales . . . 62
6.7. Reflexiones finales . . . 65

7. Conclusiones y Trabajo Futuro **67**
7.1. Conclusiones . . . 67
7.2. Trabajo Futuro . . . 68

A. Repositorios Desarrollados Durante el Proyecto **72**

Índice de figuras

3.1. Pantalla principal de la aplicación GalaxyRVR Control App. . . . 28
3.2. Modo Manual con control mediante teclas y sliders. . . . 28
3.3. Modo Sistematico con control mediante archivo .csv. . . . 29
3.4. Modo Monitor mostrando telemetría, sensores, información del sistema y streaming del video. . . . 29

4.1. Flujo metodológico de desarrollo de agentes, tomado de [1]. . . . 32
4.2. Diagrama de flujo de las operaciones de los agentes. . . . 34
4.3. Mapeo de agentes . . . 39
4.4. Salida del monitor serial del demo Sender Receiver en Arduino UNO R3. . . . 44
4.5. Salida del monitor serial del demo Rock Paper Scissors en Arduino UNO R3. . . . 44
4.6. Salida del monitor serial del demo Telemetry en Arduino UNO R3. . . . 45
4.7. Salida del monitor serial del demo Sender Receiver en Arduino MEGA. . . . 46
4.8. Salida del monitor serial del demo Rock Paper Scissors en Arduino MEGA. . . . 46
4.9. Salida del monitor serial del demo Telemetry en Arduino MEGA. . . . 47
4.10. Mapeo de agentes . . . 49
4.11. Salida del monitor serial al inicio del programa. . . . 51
4.12. Salida del monitor serial durante la comunicación con la aplicación. . . . 51

6.1. Estructura general del código en FreeMAES. obtenido de [2] 63

Índice de cuadros

3.1. Descripción de los métodos de la clase `RoverClient`. 24
3.2. Descripción de los métodos de la clase `VideoThread`. 25
3.3. Descripción de los métodos de la clase `ManualPage`. 25
3.4. Descripción de los métodos de la clase `SystematicPage`. 26
3.5. Descripción de los métodos de la clase `MonitorPage`. 27

4.1. Definición de roles de los agentes del sistema multiagente. 35
4.2. Descomposición de funciones de los agentes del sistema multiagente. 36
4.3. Modelado de datos de los agentes del sistema multiagente con tipos de variables. 37
4.4. Asignación de comportamientos por agente, basada en entradas, procesamiento y salidas. 38
4.5. Asignación de componentes físicos a los agentes del sistema multiagente. 40
4.6. Asignación de comportamientos por agente, basada en sus entradas, procesamiento y salidas. 48
4.7. Asignación de componentes físicos a los agentes del sistema multiagente. 49

6.1. Uso de memoria en Rock Paper Scissors 58
6.2. Uso de memoria en Sender Receiver 58
6.3. Uso de memoria en Telemetry 58
6.4. Ciclos de CPU en estado Idle – Rock Paper Scissors 59
6.5. Ciclos de CPU en estado Idle – Sender Receiver 59
6.6. Ciclos de CPU en estado Idle – Telemetry 59
6.7. Consumo máximo de stack – Rock Paper Scissors 59
6.8. Consumo máximo de stack – Sender Receiver 59
6.9. Consumo máximo de stack – Telemetry 60
6.10. Uso de memoria – GalaxyRVR 60
6.11. Uso total de RAM – GalaxyRVR 61

Listings

4.1. Redirección de salida estándar al puerto serie de Arduino. 42
4.2. Modificacion de `Agent_Platform::get_state()` 43
4.3. Configuraciones clave en FreeRTOSConfig.h. 43
4.4. Inicialización del sistema multi-agentes en main.ino 50
5.1. Instalación de dependencias para GalaxyRVR-Control-App 54
6.1. Archivo `infrared_agent.h`. Declaraciones e inclusiones del agente de sensores infrarrojos. 64
6.2. Archivo `infrared_agent.cpp`. Implementación del comportamiento del agente de sensores infrarrojos. 64

Capítulo 1

Introducción

La exploración espacial moderna ha impulsado avances significativos en áreas como la cooperación internacional, el desarrollo tecnológico y la investigación científica. Entre estos avances, la navegación autónoma de vehículos tipo rover ha emergido como un componente esencial para la exploración de superficies planetarias, donde las condiciones remotas y hostiles requieren sistemas robustos, confiables y altamente adaptativos. Misiones como Perseverance de la NASA y Rosalind Franklin de la Agencia Espacial Europea han demostrado la necesidad de integrar algoritmos avanzados de percepción, odometría visual y planificación autónoma que permitan operar con altos niveles de autonomía y resiliencia en terrenos no estructurados [3, 4].

En Costa Rica, el Laboratorio de Sistemas Espaciales (SETEC) del Instituto Tecnológico de Costa Rica ha destacado por su participación en el diseño y ejecución de misiones satelitales orientadas al monitoreo ambiental y los sistemas de navegación. Entre sus proyectos se encuentran Irazú, el primer CubeSat costarricense puesto en órbita en 2018, que superó ampliamente su tiempo de vida estimado [5], y la misión GWSat, desarrollada en conjunto con la Universidad George Washington, la cual incorporó tecnología de propulsión y sistemas de análisis ambiental con resultados altamente satisfactorios [6]. Además, iniciativas como el proyecto ITCP han permitido explorar nuevas metodologías para la validación de subsistemas críticos en CubeSats.

Actualmente, el SETEC Lab desarrolla el proyecto ELANav, cuyo objetivo es crear una biblioteca de software embebido con capacidades de redundancia y adaptabilidad para sistemas de navegación en misiones espaciales críticas. Este proyecto contempla el desarrollo de algoritmos, un rover de exploración y una plataforma de validación embebida. Dado que diseñar y validar componentes críticos es un proceso complejo, una alternativa prometedora consiste en emplear el paradigma de sistemas multiagentes en entornos embebidos, permitiendo distribuir tareas, aislar fallos, mejorar la modularidad y facilitar la replicación de comportamientos autónomos en hardware de bajo consumo.

El presente trabajo se enmarca dentro de esta iniciativa, contribuyendo al desarrollo de una plataforma experimental que permita validar algoritmos de navegación mediante la implementación de un sistema multiagente embebido en un rover de prueba. Con ello, se busca fortalecer las capacidades nacionales en tecnologías de navegación espacial y sentar las bases para futuros proyectos de investigación, transferencia tecnológica y colaboración con la industria aeroespacial.

1.1 Antecedentes

El presente proyecto forma parte de una línea de investigación que el Laboratorio de Sistemas Espaciales (SETEC Lab) ha desarrollado durante varios años en torno a las arquitecturas de software distribuidas, la ingeniería de sistemas

multiagente y su aplicación en misiones espaciales con recursos limitados. Esta línea de trabajo ha evolucionado a través de varias contribuciones cada una aportando componentes esenciales que dan fundamento al desarrollo de este proyecto.

La primera contribución en esta línea proviene de la tesis doctoral de Carvajal-Godínez titulada *Agent-Based Architectures Supporting Fault-Tolerance in Small Satellites* [1]. En esta investigación se plantea el uso de arquitecturas basadas en sistemas multiagente (MAS) como un mecanismo para enfrentar el incremento en la complejidad del software a bordo de satélites pequeños. A partir de un análisis profundo de algoritmos de detección, aislamiento y recuperación de fallas (FDIR), de los modelos de comunicación en topologías de buses lineales y de los desafíos del subsistema de determinación y control de actitud (ADCS), el autor establece fundamentos teóricos y prácticos para la aplicación de MAS en sistemas críticos.

Uno de los aportes más relevantes para este proyecto es la propuesta de la metodología *Multi-Agent Systems for Satellite Applications* (MASSA), la cual integra principios de desarrollo basado en modelos con las necesidades particulares del software espacial. Esta metodología establece un flujo de trabajo formal para diseñar, modelar, verificar e implementar aplicaciones multiagente orientadas a subsistemas críticos, y sirve como base conceptual para investigaciones posteriores en el SETEC Lab.

La segunda contribución clave es la tesis de maestría de Chan-Zheng titulada *MAES: A Multi-Agent Systems Framework for Embedded Systems* [7]. En esta investigación se desarrolla un marco de trabajo multiagente diseñado específicamente para sistemas embebidos con restricciones de tiempo real y recursos limitados. El framework, denominado MAES, se construyó sobre el sistema operativo TI-RTOS con el propósito de dotar a las aplicaciones multiagente de propiedades de predictibilidad temporal, estandarización en el intercambio de mensajes y una API uniforme para la construcción de agentes.

Los resultados experimentales demuestran que MAES ofrece tiempos de ejecución consistentes y predecibles, factor clave para aplicaciones críticas como las empleadas en misiones espaciales. No obstante, la autora identifica también un incremento moderado en el uso de memoria Flash y SRAM asociado al paradigma multiagente, lo cual evidencia la necesidad de estudiar y optimizar estos costos en futuras implementaciones. Este análisis resulta particularmente relevante para este proyecto, dado que se trabaja con arquitecturas aún más restringidas (familia ATmega).

La tercera contribución corresponde a la tesis de licenciatura de Rojas-Marín titulada *FreeMAES: Desarrollo de una biblioteca bajo el paradigma Multiagente para Sistemas Embebidos (MAES) compatible con la NanoMind A3200 y el kernel FreeRTOS* [8]. En esta investigación se realiza la migración del paradigma MAES desde TI-RTOS hacia el sistema operativo de tiempo real FreeRTOS, con el objetivo de habilitar su uso en plataformas como la computadora a bordo NanoMind A3200 (arquitectura AVR32), ampliamente utilizada en CubeSats.

El autor desarrolla una librería multiagente completa (FreeMAES), describe sus clases, sus estructuras de datos, las adaptaciones necesarias para el modelo de agentes y los mecanismos de mensajería, y valida la implementación mediante pruebas unitarias y casos de estudio siguiendo la metodología MASSA. Un resultado de alto valor para este trabajo es la caracterización del costo en memoria asociado a FreeMAES, tanto para AVR8 como para AVR32, lo cual establece un punto de referencia cuantitativo para evaluar nuevas implementaciones o adaptaciones del framework.

1.2 Definición del problema

Los sistemas de misión crítica son fundamentales para el éxito de cualquier exploración espacial, ya que su correcto funcionamiento determina la viabilidad de la misión. Un fallo en estos sistemas puede comprometer todas las operaciones dependientes, poniendo en riesgo no solo los objetivos científicos, sino también la integridad del rover. En el caso de los vehículos de exploración, como los rovers lunares o marcianos, los desafíos son aún más complejos

debido a las condiciones extremas en las que operan. A diferencia de la Tierra, donde el GPS proporciona referencias de navegación precisas, estos vehículos deben moverse en entornos sin infraestructura de apoyo, lo que exige soluciones autónomas y altamente robustas. Entre los principales desafíos que enfrentan se encuentran los terrenos hostiles e impredecibles, donde un error en los subsistemas de detección de obstáculos puede llevar a colisiones o atascamientos. Además, las comunicaciones con retraso, demora de minutos (o incluso horas) en las señales entre la Tierra y Marte o la Luna, obligan a los rovers a tomar decisiones críticas sin intervención humana. Un sistema de navegación impreciso podría resultar en decisiones erróneas, como seleccionar rutas inestables o malinterpretar la ubicación de objetivos científicos valiosos, lo que afectaría directamente el éxito de la misión. Las consecuencias de no resolver estos desafíos pueden ser graves. Por ejemplo, durante la recolección de muestras, un posicionamiento incorrecto podría dañar o contaminar materiales científicos de alto valor, como ocurrió en los primeros intentos del rover Perseverance [9]. Asimismo, errores en la navegación pueden exponer al rover a terrenos abrasivos, acelerando el desgaste del hardware, tal como sucedió con el deterioro prematuro de las ruedas del Curiosity debido al contacto prolongado con superficies rocosas [10]. Además, cada fallo requiere un análisis exhaustivo por parte de los equipos en Tierra, consumiendo horas valiosas en misiones que operan bajo ventanas de tiempo críticas, lo que reduce la eficiencia operativa y retrasa el avance de los objetivos científicos.

Para dimensionar el impacto de los fallos en sistemas de misión crítica, resulta ilustrativo el caso de los rovers gemelos Spirit [11] y Opportunity [12]. Ambos enfrentaron problemas relacionados con la generación de energía: la acumulación de polvo en los paneles solares redujo de forma progresiva su producción eléctrica, lo que obligó a limitar las operaciones científicas. En el caso del Spirit, dos de sus ruedas dejaron de funcionar, lo que provocó que quedara atascado en una zona de arena blanda y su misión concluyera tras 6 años de servicio. Por otra parte, el Opportunity, a pesar de operar durante 15 años, vio finalizada su misión debido a una tormenta de polvo global que impidió la generación de energía suficiente para mantener sus sistemas activos. El mantenimiento de ambas misiones durante ese periodo tuvo un costo de 335.8 millones de dólares [13]. Aunque los rovers cumplieron ampliamente sus objetivos iniciales, los problemas de energía y movilidad limitaron su potencial, reduciendo la cantidad de descubrimientos científicos que pudieron haberse logrado si no hubieran enfrentado dichas fallas.

Los sistemas embebidos de misión crítica están diseñados para satisfacer requisitos muy estrictos de recursos, como memoria limitada, procesamiento en tiempo real y máxima confiabilidad. Como resultado, el software suele estar altamente acoplado a una plataforma específica, lo que complica su uso en arquitecturas diferentes. Según un informe técnico de Carnegie Mellon University, software embebido de tiempo real *"embedded real-time software is particularly difficult to rehost because of ... its tailoring and optimization to fit the limited resource footprint of the hardware and the need to support specialized device interfaces"* [14]. Esta situación refleja la falta de bibliotecas o kits de desarrollo estandarizados, lo que obstaculiza la reutilización tecnológica entre plataformas y genera costos adicionales de desarrollo cada vez que se quiere adaptar el software a un nuevo hardware o sistema operativo.

Actualmente, el GalaxyRVR del SETEC Lab presenta limitaciones técnicas que impiden su uso como plataforma de validación de algoritmos de misión crítica. En cuanto al manejo del rover, solo permite control manual desde la aplicación móvil, lo que imposibilita la ejecución de pruebas sistemáticas y la automatización de secuencias de movimiento. A nivel del sistema operativo embebido, no es posible obtener información en tiempo real sobre el estado interno del rover, como la cantidad de tareas que se ejecutan, el estado de cada una o el tiempo de CPU que consumen, lo que genera tiempos de ejecución no predecibles y reduce la confiabilidad en la validación de algoritmos. Finalmente, en el área de desarrollo de software, no existe un estándar para el diseño de algoritmos, lo que dificulta su reutilización

entre proyectos y prolonga el tiempo de implementación.

En contraste, el estado deseado para el proyecto ELANav requiere que el rover sea capaz de ejecutar pruebas automatizadas mediante un control sistemático y la programación de secuencias, además de contar con un sistema operativo embebido que proporcione métricas en tiempo real sobre tareas concurrentes, consumo de CPU y estados de ejecución, asegurando así tiempos predecibles y repetibles. Asimismo, es indispensable establecer un estándar de desarrollo de algoritmos que facilite su portabilidad y reutilización entre distintas plataformas. Esta brecha entre el estado actual y el deseado limita directamente la capacidad del laboratorio para validar de manera eficiente y confiable los sistemas de misión crítica.

1.3 Síntesis del problema

EL SETEC Lab necesita de una plataforma que le permita diseñar agentes de software que realicen las funciones críticas del rover en menos tiempo, además, debido a la naturaleza de los sistemas de misión crítica todos realizan funciones complejas lo que aumenta el tiempo de desarrollo. También se necesita de una interfaz de trabajo que permita realizar pruebas sistematizadas con el fin de depurar los agentes diseñados y el dispositivo en el que se ejecuten.

El problema se puede sintetizar en la siguiente frase:

"El proyecto ELANav carece de una interfaz personalizable para el GalaxyRVR y una plataforma integrada de gestión de hardware y software, lo que impide realizar pruebas controladas, dificulta la depuración de algoritmos y aumenta el tiempo de implementación de agentes"

1.4 Meta

Validar que la integración de FreeRTOS y la biblioteca FreeMAES en la arquitectura ATmega32 sea eficiente en el uso de recursos computacionales, con un diseño modular que facilite la implementación en proyectos de misión crítica como ELANav y al menos un proyecto adicional.

1.5 Objetivo General

Desarrollar un sistema multiagente embebido basado en FreeRTOS y FreeMAES sobre la arquitectura ATmega32 del GalaxyRVR, que permita validar algoritmos de misión crítica mediante una interfaz configurable y medible.

1.6 Objetivos Específicos

1. Diseñar una interfaz gráfica de usuario (GUI) configurable que permita la operación remota y monitoreo del GalaxyRVR.

2. Adaptar la biblioteca FreeMAES (desarrollada para la plataforma NanoMind A3200) a la arquitectura ATme-

ga32 de Arduino, asegurando su compatibilidad con el hardware del GalaxyRVR.

3. Evaluar el rendimiento de la implementación de software embebido propuesta frente a la versión original del GalaxyRVR, identificando mejoras en eficiencia, funcionalidad y usabilidad.

4. Documentar la migración de la biblioteca FreeMAES a la arquitectura ATmega32 y el uso de la aplicación desarrollada, con el fin de facilitar su adopción y reutilización por parte de futuros usuarios e investigadores.

1.7 Enfoque Metodológico

1. Ensamblaje del SunFounder GalaxyRVR: Esta etapa consiste en montar el rover siguiendo la guía oficial proporcionada por el fabricante. El propósito es contar con una plataforma física en condiciones óptimas para el desarrollo del proyecto, garantizando que los motores, sensores y demás componentes se encuentren instalados correctamente y listos para su uso en pruebas posteriores.

2. Estudio del código del rover: Una vez ensamblado, se realiza un análisis del software original del GalaxyRVR. Esta revisión permite comprender cómo están programadas las funciones de control de motores, lectura de sensores y comunicación, lo cual servirá como línea base para desarrollar la aplicación y los agentes.

3. Investigación de sistemas multi agentes: En esta fase se estudia el paradigma multiagente aplicado a sistemas embebidos. La investigación se enfoca en cómo los agentes pueden distribuir responsabilidades críticas, facilitar la modularidad del software y mejorar la robustez en entornos de misión crítica. Esto servirá como fundamento teórico para el diseño de los agentes en el rover.

4. Investigación de la biblioteca FreeMAES: Aquí se profundiza en la biblioteca FreeMAES a partir de la tesis y guía técnica que documentan su diseño. El objetivo es entender cómo esta biblioteca implementa el paradigma multiagente en sistemas embebidos y qué modificaciones serán necesarias para llevarla de la plataforma NanoMind A3200 al Arduino Uno R3.

5. Simulación FreeMAES: Antes de migrar la biblioteca al hardware, se lleva a cabo una simulación en un ambiente con FreeRTOS. Esto permite familiarizarse con el funcionamiento de FreeMAES, comprender su API y validar, de manera preliminar, la interacción entre tareas y agentes en un entorno controlado.

6. Diseño de la Aplicación: En este punto se desarrolla una aplicación de escritorio con interfaz gráfica que permita enviar instrucciones al rover, configurar pruebas y recibir datos de telemetría en tiempo real. La aplicación también gestionará la visualización del estado de los sensores y del sistema operativo, y permitirá generar reportes con los resultados obtenidos durante los experimentos.

7. Migración de la biblioteca FreeMAES: La siguiente actividad consiste en adaptar la biblioteca FreeMAES para que pueda ejecutarse en el Arduino Uno R3, que utiliza la arquitectura ATmega32. Esto implica ajustar aspectos relacionados con el manejo de memoria, temporizadores y periféricos para garantizar la compatibilidad con el hardware del rover.

8. Verificación de la biblioteca FreeMAES: Una vez migrada la biblioteca, se realizan pruebas funcionales ejecutando los demos incluidos en FreeMAES. Estas pruebas permiten comprobar que la biblioteca opera correctamente en el nuevo hardware, validando la comunicación entre agentes y la estabilidad del sistema.

9. Diseño de los agentes: En esta fase se definen e implementan los agentes que llevarán a cabo las funciones críticas del rover, como percepción, navegación, arbitraje de decisiones y monitoreo de la salud del sistema. Cada agente será diseñado bajo los principios del paradigma multiagente, buscando modularidad y tolerancia a fallos.

10. Comparación experimental: Finalmente, se lleva a cabo una serie de pruebas comparativas entre dos configuraciones del rover: la primera utilizando el código original y la segunda implementando FreeRTOS con FreeMAES y los agentes diseñados. Durante estas pruebas se recolectarán métricas como el número de tareas activas, el consumo de CPU y el desempeño general del sistema. Estas comparaciones permitirán evidenciar las mejoras alcanzadas y validar los objetivos del proyecto.

11. Documentación y entrega final: En la última etapa se elabora la documentación completa del proceso. Esta incluirá la migración de la biblioteca FreeMAES, la descripción de los agentes implementados y la guía de uso de la aplicación desarrollada, de manera que futuros usuarios o investigadores del SETEC puedan reutilizar y ampliar el trabajo realizado.

1.8 Estructura de la Tesis

La tesis está organizada de la siguiente manera: el Capítulo 2 presenta el marco teórico desarrollado para esta investigación. Incluye la definición de los conceptos fundamentales, así como la revisión bibliográfica y el estado del arte relacionado con el uso de sistemas multiagentes y sistemas operativos de tiempo real en vehículos de exploración espacial y otros sistemas de misión crítica. En particular, los Capítulos 3, 4, 5 y 6 corresponden al desarrollo de los cuatro objetivos específicos planteados en este trabajo.

El Capítulo 3 aborda el diseño de la aplicación para el control del GalaxyRVR. Se describen la arquitectura general, las herramientas de software utilizadas (principalmente PyQt5) y la implementación del protocolo de comunicación basado en WebSockets.

A continuación, el Capítulo 4 presenta la metodología empleada para el diseño del sistema multiagentes, el proceso de adaptación de la biblioteca FreeMAES y el posterior rediseño de la arquitectura debido a las restricciones del hardware seleccionado.

Posteriormente, el Capítulo 5 presenta la implementación definitiva donde se detalla los repositorios de documentación desarrollados para facilitar la comprensión, el uso y la replicabilidad del proyecto. Se describe la estructura de cada repositorio y su relevancia dentro del flujo de trabajo general.

El Capítulo 6 presenta los resultados de la evaluación de FreeRTOS + FreeMAES. Este capítulo se divide en dos secciones: en la primera se comparan FreeRTOS y FreeRTOS+FreeMAES mediante programas de prueba, y en la segunda se analiza el rendimiento del programa original del GalaxyRVR frente al sistema multiagente desarrollado durante este proyecto.

Finalmente, en el Capítulo 7 se discuten las conclusiones generales, las recomendaciones derivadas del proceso de investigación y las posibles líneas de trabajo futuro.

Capítulo 2

Marco Teórico

2.1 Conceptos Fundamentales

Arduino

Arduino es una plataforma de hardware y software de código abierto diseñada para facilitar el desarrollo de sistemas electrónicos interactivos. Se basa en microcontroladores que pueden programarse a través del entorno de desarrollo Arduino IDE y cuenta con una amplia variedad de placas y módulos. Su popularidad radica en la sencillez de uso, la gran comunidad de usuarios y la posibilidad de conectar sensores, actuadores y dispositivos de comunicación, lo que la convierte en una herramienta ampliamente utilizada en prototipado, investigación y enseñanza en ingeniería electrónica y sistemas embebidos. [15]

SunFounder GalaxyRVR

El SunFounder GalaxyRVR es un rover educativo de código abierto desarrollado como plataforma de aprendizaje en robótica, control y sistemas embebidos. Está equipado con un Arduino Uno R3 como unidad de procesamiento principal, sensores infrarrojos y ultrasónicos para detección de obstáculos, así como una cámara para visión artificial. Su diseño modular lo convierte en un vehículo ideal para la experimentación en algoritmos de navegación, control autónomo y validación de software embebido en entornos académicos. [16]

Agente

Según la especificación FIPA, un agente es un proceso computacional autónomo capaz de comunicarse con otros mediante un Agent Communication Language. Representa la unidad fundamental de ejecución dentro de una plataforma de agentes y combina una o varias capacidades de servicio bajo un modelo integrado. Cada agente debe poseer un mecanismo de identidad único denominado *Agent Identifier* (AID), el cual lo distingue inequívocamente dentro del sistema y puede asociarse a múltiples direcciones de transporte donde puede ser contactado. [17]

Sistema Multiagente

Un sistema multiagente (MAS, por sus siglas en inglés) es un conjunto de agentes que interactúan entre sí para resolver problemas de manera cooperativa o competitiva. Cada agente tiene un grado de autonomía, pero el comportamiento colectivo emerge de su comunicación y coordinación. Este paradigma es útil en sistemas distribuidos y embebidos, ya que permite dividir tareas críticas en componentes independientes, mejorando la escalabilidad, la modularidad y la tolerancia a fallos de la aplicación. [17]

Agent Management System (AMS)

El Agent Management System (AMS) es el componente obligatorio y supervisor de una plataforma de agentes FIPA. Su función es controlar el acceso y uso de la plataforma, mantener un directorio centralizado de Agent Identifiers (AIDs) con sus direcciones de transporte, y ofrecer servicios de paginas blancas para identificar y localizar agentes. Solo puede existir un AMS por plataforma, y todo agente debe registrarse en él para obtener un AID válido y operar dentro del sistema. [17]

Sistema Operativo en Tiempo Real (RTOS)

Un Sistema Operativo en Tiempo Real (RTOS) es un sistema operativo diseñado para ser pequeño, eficiente y determinista, capaz de responder a eventos externos dentro de límites de tiempo estrictamente definidos. Este tipo de sistema se utiliza en aplicaciones embebidas como dispositivos médicos, controladores automotrices y sistemas robóticos, donde la precisión temporal es esencial para garantizar un funcionamiento seguro y confiable. A diferencia de los sistemas operativos de propósito general, un RTOS optimiza la ejecución de tareas mediante planificadores basados en prioridades y mecanismos predecibles de control, permitiendo cumplir requisitos de tiempo real incluso en dispositivos con recursos limitados de memoria, procesamiento y energía. [18]

FreeRTOS

FreeRTOS es un sistema operativo de tiempo real (RTOS) de código abierto diseñado para sistemas embebidos. Ofrece planificación determinista de tareas, sincronización mediante semáforos y colas, y gestión eficiente de memoria en microcontroladores con recursos limitados. Su ligereza, portabilidad y amplia documentación lo han convertido en uno de los RTOS más populares en aplicaciones críticas, incluyendo automoción, dispositivos médicos, IoT y plataformas aeroespaciales educativas como el GalaxyRVR. [19]

FreeMAES

FreeMAES es una biblioteca que permite implementar el paradigma multiagente sobre un sistema operativo de tiempo real de manera estandarizada, lo que reduce la complejidad y el tiempo necesario para diseñar sistemas de misión crítica. Su diseño original fue desarrollado para la plataforma NanoMind A3200, pero su arquitectura modular facilita su migración a otros entornos embebidos. Esta integración combina la robustez de un RTOS con las ventajas de los sistemas multiagente, como modularidad, escalabilidad y tolerancia a fallos. [8]

Qt

Qt es un framework multiplataforma orientado al desarrollo de interfaces gráficas de usuario (GUIs) y aplicaciones interactivas. Soporta múltiples lenguajes, siendo C++ y Python (a través de PyQt o PySide) los más usados. Qt destaca por su robusta colección de widgets, su motor gráfico avanzado y la capacidad de integrarse con sistemas embebidos, lo que lo hace una opción popular para desarrollar aplicaciones de control, monitoreo y visualización de datos en tiempo real. [20]

WebSockets

WebSockets es un protocolo de comunicación que permite establecer un canal bidireccional y persistente entre un cliente y un servidor sobre una única conexión TCP. A diferencia del modelo tradicional basado en peticiones HTTP, WebSockets facilita la transmisión en tiempo real de datos con baja latencia, lo cual es ideal para aplicaciones interactivas como el control remoto de robots, sistemas de telemetría y GUIs de sistemas embebidos. [21]

2.2 Revisión Bibliográfica

La investigación en control de rovers y sistemas multiagente embebidos ha mostrado avances significativos en distintas áreas de aplicación, desde la exploración espacial hasta el diseño de arquitecturas de software reconfigurables. A continuación, se presenta una revisión de trabajos relevantes que ilustran el progreso en estos campos.

En primer lugar, se han desarrollado múltiples técnicas de control aplicadas a rovers planetarios, las cuales buscan garantizar movilidad robusta en entornos extremos como Marte o la Luna. El estudio de [22] analiza distintos enfoques de control, subrayando la importancia del Fault-Tolerant Control (FTC) como estrategia para mantener la estabilidad y confiabilidad del vehículo incluso ante fallas de sensores, actuadores o subsistemas. Se analizan anomalías recurrentes en misiones pasadas, como fallos de motores, hundimiento de ruedas o daños estructurales, y se presentan métodos de diagnóstico y control adaptativo para reducir la propagación de errores. El FTC es planteado como esencial en entornos tan extremos como la superficie lunar, ya que permite mitigar degradaciones de rendimiento, asegurar continuidad de operación y aumentar la autonomía de los rovers frente a situaciones imprevistas.

Por otro lado, [23] introduce un diseño innovador de rovers para cuerpos de baja gravedad que emplean tres volantes de inercia internos como sistema de actuación, eliminando la necesidad de ruedas o sistemas de propulsión convencionales. A través del control de la velocidad de giro de los volantes, el rover logra saltos controlados, movimientos por tumbos y reorientación precisa, mientras que los picos externos protegen la estructura y facilitan el contacto con la superficie. El diseño es compacto, económico y robusto al encapsular todos los subsistemas en una carcasa sellada, lo cual reduce su complejidad operativa. Los modelos teóricos fueron validados en un banco de pruebas de microgravedad de seis grados de libertad, demostrando su viabilidad para exploraciones seguras y eficientes en asteroides, cometas o lunas pequeñas.

En el ámbito de los sistemas multiagente embebidos, [24] analiza los desafíos de prueba en sistemas multiagente embebidos (EMAS), destacando que, a diferencia del software clásico, estos requieren verificar no solo las funcionalidades individuales de los agentes, sino también sus interacciones y el comportamiento colectivo del sistema. Los autores proponen una estrategia de pruebas estructurada en tres ejes: pruebas a nivel de agente, pruebas de características colectivas (sociedades u organizaciones de agentes) y pruebas de aceptación, considerando aspectos como autonomía, concurrencia, heterogeneidad y seguridad. Se concluye que la complejidad de los EMAS, derivada de su naturaleza distribuida y descentralizada, exige metodologías de prueba específicas que integren tanto las dimensiones de hardware como de software, especialmente para evaluar protocolos de interacción y autoorganización.

Finalmente, [25] aborda la problemática de la reconfiguración dinámica en sistemas embebidos multiagente. El artículo introduce un protocolo de reconfiguración basado en la norma IEC61499, donde los agentes pueden modificar arquitecturas y políticas de control en tiempo de ejecución sin comprometer la coherencia global del sistema. El aporte central consiste en la definición de agentes de coordinación y mecanismos formales de verificación, lo cual permite dotar a los sistemas de mayor flexibilidad y capacidad de adaptación frente a fallos o cambios en el entorno.

En conjunto, estos trabajos muestran la evolución de la investigación desde el desarrollo de técnicas de control robustas para rovers individuales, pasando por propuestas de cooperación distribuida, hasta la incorporación de paradigmas multiagente en sistemas embebidos con capacidades de prueba y reconfiguración. Estos avances reflejan un movimiento hacia arquitecturas más complejas, adaptativas y confiables, que permiten abordar con mayor eficacia los retos de la exploración espacial y el control distribuido en entornos críticos.

2.3 Estado del Arte

Sistemas Multiagentes en sistemas embebidos

Los sistemas multiagentes embebidos (EMAS) existen, pero no son tan comunes en la industria como otros enfoques tradicionales, como los sistemas operativos de tiempo real (RTOS). Esto se debe a varias causas:

- **Complejidad:** diseñar arquitecturas multiagente en plataformas con recursos computacionales limitados resulta difícil, dado que la mayoría de los sistemas embebidos cuentan con poca memoria y capacidad de procesamiento.
- **Determinismo:** en entornos críticos se privilegia la ejecución con tiempos predecibles, lo cual hace que los RTOS clásicos sean más atractivos que agentes autónomos que negocian decisiones.
- **Madurez de herramientas:** existen menos librerías y estándares consolidados para EMAS en comparación con los RTOS tradicionales, lo que limita su adopción a gran escala.

Aun así, se han desarrollado aplicaciones que demuestran la utilidad de los sistemas multiagentes en entornos embebidos. Un ejemplo claro es el informe "GPMAPPO: Collaborative SAR Optimization of Human-UAV in Post-Disaster Scenarios" [26], el cual aborda la necesidad de optimizar misiones de búsqueda y rescate (SAR) en escenarios post-desastre, combinando equipos humanos con vehículos aéreos no tripulados (UAVs). Para lograrlo, se introduce una Estrategia de Colaboración Regional (RCS) y un nuevo algoritmo llamado GPMAPPO (Path-Greedy Multi-Agent Proximal Policy Optimization), que combina un enfoque voraz de selección de rutas con técnicas de reinforcement learning multiagente. En este esquema, los UAVs son tratados como agentes autónomos que cooperan con los equipos humanos, lo que les permite tomar decisiones rápidas sin depender completamente de órdenes externas.

En relación con este escrito, aunque el paper mencionado no utiliza explícitamente un RTOS combinado con un sistema multiagente embebido, demuestra con claridad las fortalezas del paradigma multiagente aplicado en entornos de misión crítica. Los UAVs del estudio funcionan como sistemas embebidos que, gracias a la inteligencia distribuida, logran mayor autonomía, resiliencia y capacidad de reacción en tiempo real. Esta misma lógica respalda la propuesta de implementar RTOS + sistemas multiagente en plataformas embebidas, como en el caso de un rover de exploración, donde la cooperación entre agentes permite gestionar tareas críticas de manera más robusta y confiable frente a condiciones adversas.

Sistemas operativos de tiempo real

Los sistemas operativos en tiempo real (RTOS) constituyen la base para el control de aplicaciones críticas en áreas como aeroespacial, automotriz, telecomunicaciones, defensa y sistemas embebidos en general. Uno de los más destacados es VxWorks. Desarrollado por Wind River [27], es uno de los RTOS más avanzados y ampliamente utilizados en aplicaciones críticas. Se caracteriza por su determinismo, modularidad y escalabilidad, y soporta arquitecturas modernas de hardware multicore. Además, ofrece compatibilidad con certificaciones de seguridad (DO-178C, ISO 26262, IEC 61508, entre otras), lo que lo convierte en una opción preferida en sectores donde la confiabilidad es innegociable. Su ecosistema incluye soporte para virtualización, seguridad avanzada, bibliotecas POSIX y herramientas de desarrollo integradas, lo que lo posiciona como el RTOS líder en la industria.

En el paper "Design of Scalable Software Architecture for Spotlight SAR Imaging on UAV Platform" [28], se utilizó VxWorks 7.0 como RTOS principal para implementar un sistema de procesamiento en tiempo real de radar de apertura sintética (SAR) montado en un UAV. El desafío principal era procesar en tiempo real un volumen masivo de

datos (13.37 GB de IQ samples) para generar imágenes de alta resolución en modo spotlight. Gracias a VxWorks, junto con el uso de una arquitectura multicore (Intel Xeon-D DSP board), fue posible coordinar múltiples tareas paralelas, gestionar memoria compartida no contigua y garantizar ejecución determinista, logrando que todo el procesamiento de la cadena algorítmica se completará cumpliendo con los requisitos de tiempo real del sistema.

En el contexto de este proyecto se utiliza FreeRTOS debido a que es de uso libre, cuenta con una gran comunidad activa, amplia documentación y una implementación relativamente simple. Estas características lo convierten en una opción muy atractiva para entornos de investigación y desarrollo, especialmente considerando que en este trabajo se está comenzando a aplicar el paradigma multiagente en sistemas embebidos, lo cual demanda un sistema operativo fácil de integrar y flexible para experimentar. Por otro lado, VxWorks representa una alternativa más robusta y madura, aunque también más compleja y con requerimiento de licenciamiento. Su uso sería ideal en un futuro, una vez que se demuestren de forma clara las capacidades y ventajas del paradigma multiagente en sistemas embebidos con RTOS.

Capítulo 3

Diseño de una aplicación para el control del GalaxyRVR

3.1 ControlRVR App

Descripción general

La **ControlRVR App** es una interfaz gráfica desarrollada en *Python* utilizando la biblioteca *PyQt5*, protocolo de comunicacion *Web Sockets*, streaming de video con *OpenCV* y soporte asincrónico con *qasync*. Su propósito principal es brindar un medio de interacción entre el **rover GalaxyRVR** (basado en un ESP32-CAM y un Arduino UNO R3) y el usuario, facilitando tanto el control manual como la ejecución de planes de movimiento predefinidos y la supervisión en tiempo real de los sensores.

La aplicación funciona como un **puente de comunicación** entre la computadora y el rover. Se conecta mediante *WebSocket* al firmware del vehículo, permitiendo:

- Enviar órdenes de control (motores, ángulo del servo, encendido de lámpara, modos de navegación).
- Recibir telemetría (voltaje de batería, sensores infrarrojos, distancia ultrasónica).
- Visualizar en tiempo real la señal de video proveniente de la cámara del GalaxyRVR.

Menú Principal

El primer nivel de interacción con la aplicación es el **Menú Principal**, una ventana sencilla que muestra tres botones:

1. Modo Manual
2. Modo Sistemático
3. Modo Monitor

Al seleccionar cualquiera de estas opciones, la aplicación redirige al usuario al submenú correspondiente. En la parte superior del menú se incluye un **indicador de estado global** de la conexión con el rover (por ejemplo, “Desconectado” o “Conectado”), lo que permite verificar rápidamente si el sistema se encuentra enlazado y listo para operar, sin importar en qué modo se esté trabajando.

Modo Manual

El Modo Manual permite al usuario tener un control inmediato y sencillo del GalaxyRVR. Su utilidad radica en que sirve para realizar pruebas rápidas de funcionamiento, verificando que los motores, el servo y la comunicación estén operando correctamente. Además, este modo funciona como respaldo en situaciones donde el comportamiento autónomo del rover no es el adecuado. Si los algoritmos de decisión no responden como se espera, el usuario puede tomar el control directo y asegurar que el vehículo se mueva de forma segura. Sus funciones principales son:

- **Sliders de motores:** controlan de forma independiente la velocidad del motor izquierdo y derecho en un rango de -100 a 100, permitiendo avanzar, retroceder y girar.
- **Slider de servo:** regula el ángulo de dirección del servo, con valores entre 0° y 140°.
- **Botón de lámpara:** activa o desactiva la luz frontal del rover.
- **Selección de modos de navegación:** incluye control Manual, *Obstacle Avoidance* (evitación de obstáculos) y *Obstacle Following* (seguimiento de objetos o paredes).
- **Botón de conexión/reconexión:** asegura que la comunicación con el vehículo pueda restablecerse en cualquier momento.
- **Control de movimiento con el teclado** permite enviar comandos de movimiento de manera sencilla por medio de las teclas WASD (adelante, izquierda, atras y derecha respectivamente) pero no permiten el control especifico de la velocidad.

Además, un **temporizador (QTimer)** se encarga de enviar cada 200 ms al rover el estado actual de los controles, garantizando fluidez y continuidad en la comunicación.

Modo Sistemático

El Modo Sistemático está diseñado para la ejecución repetida de pruebas bajo condiciones controladas. Al cargar o escribir una secuencia de instrucciones, el rover siempre va a ejecutar los mismos movimientos en el mismo orden, lo que garantiza consistencia entre pruebas. Esta característica es clave en entornos de experimentación, ya que permite comparar resultados, identificar diferencias tras cambios de programación y evaluar de manera objetiva la respuesta del sistema. En este sentido, el modo sistemático facilita la validación de algoritmos y el registro de datos de forma confiable y reproducible. Sus funciones principales son:

- **Editor de secuencias:** cuadro de texto donde se escriben o cargan instrucciones en el formato:

```
motor_izq, motor_der, servo, luz, modo_e, modo_f, tiempo_ms
```

Ejemplo:

```
50,50,90,0,0,0,2000   # Avanzar recto 2 s
-50,50,90,1,0,0,1000  # Girar hacia la izquierda con luz encendida por 1s
0,0,90,0,0,1,1500     # Activar el modo de seguimiento por 1.5s
```

- **Carga desde archivo CSV:** permite seleccionar un archivo externo con la lista de instrucciones.
- **Ejecución automática:** la aplicación envía periódicamente los valores al rover, manteniendo cada instrucción durante el tiempo especificado.
- **Control de ejecución:** incluye botones *Run Sequence* (inicia) y *Stop Sequence* (detiene y pone motores en cero).
- **Estado en pantalla:** indica en qué paso de la secuencia se encuentra el rover o si ya finalizó la ejecución.

Modo Monitor

El Modo Monitor ofrece al usuario una visión clara del estado interno y externo del rover en tiempo real. A través de los datos de la batería, sensores infrarrojos y ultrasónicos, además de la cámara, el operador puede verificar si el vehículo está respondiendo correctamente y detectar posibles fallos o limitaciones en su desempeño. Este modo no solo proporciona seguridad al permitir conocer el estado del sistema durante las pruebas, sino que también ayuda a evaluar el rendimiento general del rover en distintos entornos, combinando telemetría y video para una supervisión completa. Sus funciones principales son:

- Muestra en pantalla el estado de los sensores infrarrojos y la distancia medida por el sensor ultrasónico.
- Permite iniciar la cámara en tiempo real con un botón, mostrando la visión del rover.
- Incluye un botón de conexión/reconexión para mantener la comunicación activa.
- Incluye un botón para encender y apagar la luz.
- Incluye un slider para controlar el angulo de la cámara.
- Incluye una sección de visualización de la cantidad de stack (SRAM) consumido por cada agente (En el caso que el rover este programado con un sistema multiagente)

Este modo es especialmente útil para pruebas de campo, pues combina telemetría y video, facilitando la validación del comportamiento del rover en entornos reales.

3.2 Diseño de la aplicación

A continuación, se presenta una explicación detallada del funcionamiento de las clases y métodos principales que conforman la aplicación. Estas estructuras son las responsables de gestionar la comunicación con el rover, controlar los distintos modos de operación, procesar el flujo de video y mantener la interfaz actualizada con la información de telemetría. Comprender cómo interactúan estas partes permite entender de manera integral el diseño de la aplicación y cómo se logra que el sistema funcione de forma estable y coordinada.

3.2.1 Clase RoverClient

La clase `RoverClient` desempeña un papel fundamental en la arquitectura de la aplicación, ya que centraliza la comunicación con el rover en un único punto. Gracias a esta centralización, las diferentes clases asociadas a los modos de operación (Manual, Sistemático y Monitor) pueden acceder al mismo cliente de comunicación sin necesidad de implementar lógicas de conexión independientes. Su propósito principal es gestionar la comunicación mediante el

protocolo WebSocket, garantizando la apertura, mantenimiento y supervisión de la conexión de red, así como el envío de comandos en formato JSON y la recepción de datos de telemetría de manera eficiente.

Método	**Descripción**
`connect()`	Establece la conexión WebSocket con el rover. Si existía una conexión previa, la cierra antes de abrir una nueva con la dirección IP y puerto definidos. Una vez conectada, actualiza el estado, envía una notificación de "connected" y lanza la tarea asincrónica que recibe los mensajes mediante `receive_data`. En caso de error, marca al cliente como desconectado y ejecuta el callback correspondiente. No recibe parámetros ni devuelve valores.
`disconnect()`	Cierra de forma segura la conexión WebSocket y marca al cliente como desconectado. Asegura la liberación de recursos incluso en caso de error. No recibe parámetros y no devuelve valores.
`send()`	Envía un diccionario en formato JSON al rover. Valida primero que la conexión esté activa; en caso de error (como pérdida de enlace), cambia el estado a desconectado y notifica al manejador de desconexión. Recibe como entrada un diccionario con valores de control y no devuelve nada; únicamente transmite los datos al rover.
`receive_data()`	Escucha de manera continua los mensajes entrantes desde el rover. Convierte los mensajes JSON en diccionarios y los pasa al callback `on_message`. Si la conexión se cierra o ocurre un error, detiene el bucle, marca al cliente como desconectado e inicializa el protocolo de reconexion. No recibe parámetros ni devuelve valores; se limita a procesar los mensajes recibidos.
`reconnect_loop()`	Cuando la aplicación se pierde la conexión con el rover esta función inicializa un ciclo que intenta restablecer la conexión.

Cuadro 3.1: Descripción de los métodos de la clase `RoverClient`.

3.2.2 Clase VideoThread

La clase `VideoThread` resulta esencial en la aplicación, ya que permite gestionar el flujo de video en un hilo independiente. La captura y procesamiento de video son tareas de alto consumo de recursos que, de ejecutarse en el hilo principal, afectarían la capacidad de respuesta de la interfaz gráfica y el envío de comandos al rover. Al utilizar un hilo separado, se garantiza que las operaciones críticas como el control del rover y la recepción de telemetría no sufran retrasos. Asimismo, esta arquitectura ofrece la posibilidad de iniciar o detener la transmisión de video dinámicamente, lo que mejora la estabilidad general de la aplicación al evitar sobrecargas en la comunicación.

Método	Descripción
`run()`	Abre la cámara del rover mediante `OpenCV`, captura cuadros en un bucle, los convierte a formato RGB y emite cada frame como `QImage` utilizando una señal de PyQt. De esta forma, la interfaz gráfica puede actualizar la vista en tiempo real sin bloquear el hilo principal. No recibe parámetros ni devuelve valores; se encarga de abrir la cámara, capturar imágenes y emitir cada cuadro convertido.
`stop()`	Detiene el bucle de captura de video, libera la cámara y garantiza que el hilo finalice correctamente. No recibe parámetros ni devuelve valores; su propósito es detener la captura y liberar los recursos de la cámara.

Cuadro 3.2: Descripción de los métodos de la clase `VideoThread`.

3.2.3 Clase ManualPage

La clase `ManualPage` constituye la interfaz gráfica de control manual del rover dentro de la aplicación. Su función principal es proporcionar al usuario controles intuitivos como *sliders* para motores y servomotor, botones para la selección de modos de operación y control de la lámpara, así como el envío periódico del estado actual del sistema al rover. Esta clase es fundamental para mantener una interacción en tiempo real con el vehículo, ya que asegura la transmisión continua de comandos, evitando pérdidas de conexión o comportamientos inesperados.

Método	Descripción
`_slider_box()`	Crea dinámicamente un control compuesto por una etiqueta, un slider y un valor numérico. Los sliders permiten manejar los motores y el servomotor, actualizando automáticamente las variables internas de la clase. Recibe como parámetros: un título (`str`), dos enteros (`int`) que definen el rango mínimo y máximo, y el nombre de la variable a modificar (`str`). Devuelve un objeto de tipo `QWidget` que contiene el slider y sus etiquetas.
`_toggle_lamp()`	Alterna el estado de la lámpara del rover (encendido/apagado) y actualiza el texto del botón correspondiente. No recibe parámetros ni devuelve valores; se limita a modificar el estado interno de la lámpara y reflejarlo en la interfaz.
`_set_mode()`	Cambia el modo de operación del rover entre *manual*, *avoidance* (evitación de obstáculos) o *follow* (seguimiento). Modifica internamente las banderas `mode_e` y `mode_f`. Recibe como parámetro una cadena de texto (`str`) con el modo deseado y no devuelve valores.
`_send_controls()`	Construye un diccionario con el estado actual de los controles (motores, servomotor, lámpara y modos) y lo envía al rover mediante el cliente WebSocket (`RoverClient`). Este método se ejecuta periódicamente a través de un `QTimer`. No recibe parámetros ni devuelve valores. Su importancia radica en mantener la conexión activa, ya que el rover requiere recibir comandos de manera constante para evitar la desconexión o quedar en un estado no deseado.

Cuadro 3.3: Descripción de los métodos de la clase `ManualPage`.

3.2.4 Clase SystematicPage

La clase `SystematicPage` implementa el modo sistemático de la aplicación, cuyo propósito es permitir la ejecución de secuencias de instrucciones predefinidas para el rover, normalmente cargadas desde un archivo en formato CSV. Este enfoque facilita la realización de pruebas repetibles y automatizadas, garantizando que las instrucciones se ejecuten en el orden y con los tiempos definidos por el usuario. Además, la clase asegura la transmisión continua de comandos hacia el rover mediante un temporizador, evitando pérdidas de conexión y asegurando la estabilidad del sistema.

Método	**Descripción**
`_load_csv()`	Abre un archivo en formato CSV y extrae las instrucciones de control. Filtra comentarios o líneas inválidas y muestra las instrucciones válidas en el editor de texto de la interfaz. Actualiza el estado en pantalla indicando éxito o error en la carga. No recibe parámetros de entrada salvo la ruta obtenida mediante el diálogo de selección de archivo y no devuelve valores.
`_run_sequence()`	Ejecuta la secuencia de instrucciones cargadas recorriendo línea por línea. Actualiza los valores de motores, servomotor y modos, esperando el tiempo indicado en cada instrucción. El envío real de comandos lo realiza un `QTimer` que corre en paralelo. El proceso puede detenerse en cualquier momento. No recibe parámetros ni devuelve valores.
`_stop_sequence()`	Interrumpe la secuencia en ejecución y pone los motores en cero. Actualiza la interfaz indicando que la secuencia fue detenida. No recibe parámetros ni devuelve valores.
`_send_control_values()`	Se ejecuta cada 100 ms mediante un `QTimer`. Envía al rover un diccionario con los valores actuales de los actuadores (motores, servomotor, lámpara y modos), asegurando que este reciba comandos de manera constante incluso durante secuencias largas. No recibe parámetros ni devuelve valores. Su importancia radica en mantener la conexión activa, al igual que el método `_send_controls()` de la clase `ManualPage`.

Cuadro 3.4: Descripción de los métodos de la clase `SystematicPage`.

3.2.5 Clase MonitorPage

La clase `MonitorPage` constituye el módulo de monitoreo de la aplicación, encargado de mostrar la telemetría y el video en tiempo real provenientes del rover. Su objetivo principal es brindar al usuario información crítica durante las pruebas, incluyendo el nivel de batería, el estado de los sensores y la visualización directa de la cámara. De esta manera, permite verificar en todo momento el correcto funcionamiento del sistema y detectar posibles fallos de hardware o conexión.

Método	Descripción
`_toggle_camera()`	Activa o desactiva el hilo de video. Si se inicia, crea un objeto `VideoThread`, lo conecta a la interfaz y comienza a mostrar los frames. Si se detiene, libera la cámara y actualiza el texto del botón correspondiente. No recibe parámetros ni devuelve valores; alterna entre iniciar y detener el hilo de captura de video.
`_update_video()`	Actualiza el `QLabel` de la interfaz con el frame recibido desde el hilo de video. Convierte la señal en un objeto de tipo `QImage` y lo muestra en pantalla. Recibe como parámetro un objeto `QImage` correspondiente a un frame de la cámara y no devuelve valores.
`handle_message()`	Procesa la telemetría recibida desde el rover. Actualiza en la interfaz el estado de conexión, el voltaje de la batería, los valores de los sensores infrarrojos (izquierdo y derecho), la medición del sensor ultrasónico y la cantidad de stack consumida por cada agente. Recibe como parámetro un diccionario con datos de telemetría y estado de conexión y no devuelve valores.

Cuadro 3.5: Descripción de los métodos de la clase `MonitorPage`.

3.3 Resultados Obtenidos

Durante el desarrollo de la aplicación *GalaxyRVR-Control-App* se logró implementar un sistema de control unificado capaz de interactuar con el rover en tiempo real mediante una interfaz gráfica intuitiva y estable. La estructura modular de la aplicación permitió integrar tres modos de operación principales:

- **Modo Manual:** ofrece el control directo de los motores, el servomotor y las luces del rover mediante controles deslizantes y botones, con una comunicación WebSocket de baja latencia.
- **Modo Sistemático:** posibilita la ejecución automatizada de secuencias de movimiento definidas en archivos CSV, reproduciendo trayectorias de prueba de forma precisa y repetible.
- **Modo Monitor:** integra el flujo de video proveniente de la cámara del rover y la lectura en tiempo real de las variables de telemetría enviadas por los agentes embebidos, tales como voltaje de batería y estados de sensores infrarrojos y ultrasónicos.

El uso de bibliotecas como *PyQt5*, *qasync*, *OpenCV* y *websockets* permitió mantener un diseño asincrónico y responsivo, evitando bloqueos en la interfaz y garantizando una comunicación continua con el rover. Además, la aplicación demostró una estructura fácilmente extensible para futuras mejoras, consolidándose como una herramienta fundamental en la validación del sistema multiagente *FreeMAES + FreeRTOS* implementado en el GalaxyRVR.

La Figura 3.1 presenta la pantalla principal de la aplicación, desde la cual el usuario puede seleccionar cualquiera de los tres modos de operación. En la Figura 3.2 se muestra el *Modo Manual*, el cual permitió realizar movimientos y pruebas rápidas del rover mediante controles directos. Posteriormente, la Figura 3.3 ilustra el *Modo Sistemático*, donde las trayectorias cargadas desde archivos CSV se ejecutan de forma automatizada y repetible. Finalmente, la Figura 3.4 exhibe el *Modo Monitor*, destacando la visualización simultánea del video, los sensores, la telemetría y el estado interno del sistema embebido, lo cual fue clave para la depuración y validación del sistema multiagente.

Figura 3.1: Pantalla principal de la aplicación GalaxyRVR Control App.

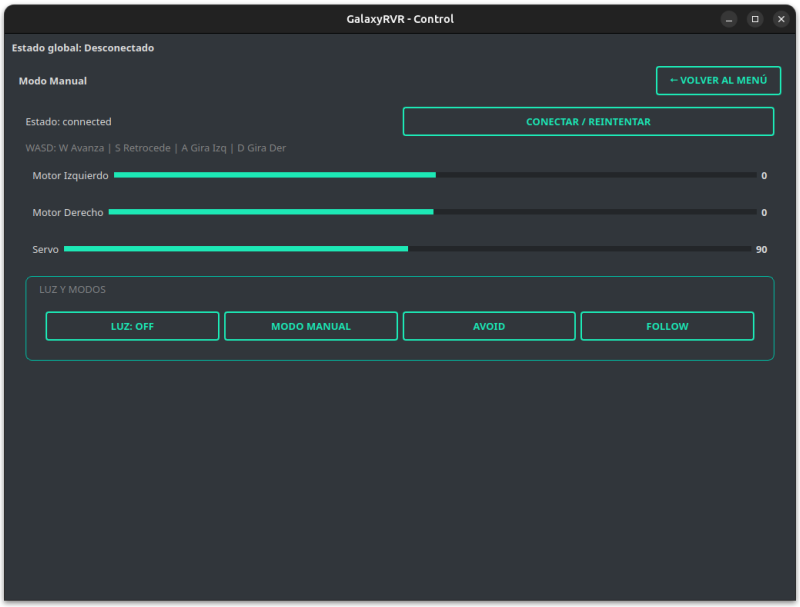

Figura 3.2: Modo Manual con control mediante teclas y sliders.

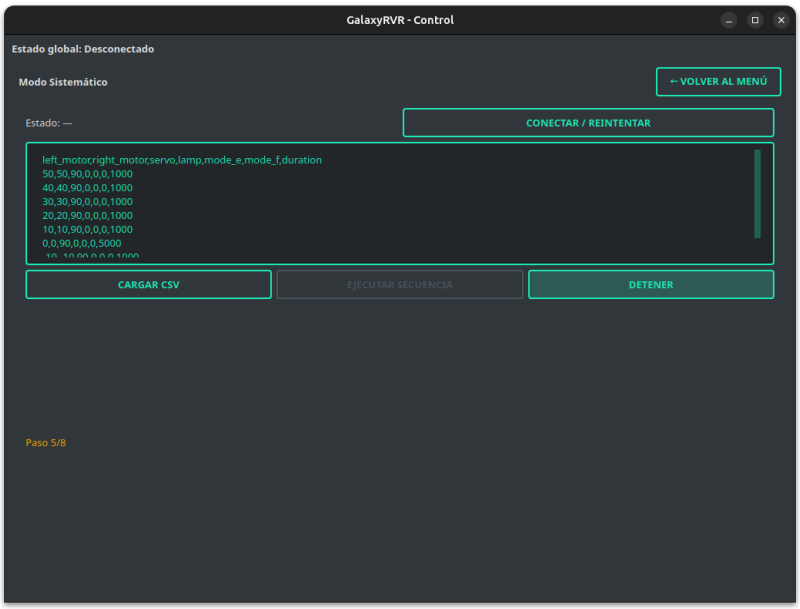

Figura 3.3: Modo Sistematico con control mediante archivo .csv.

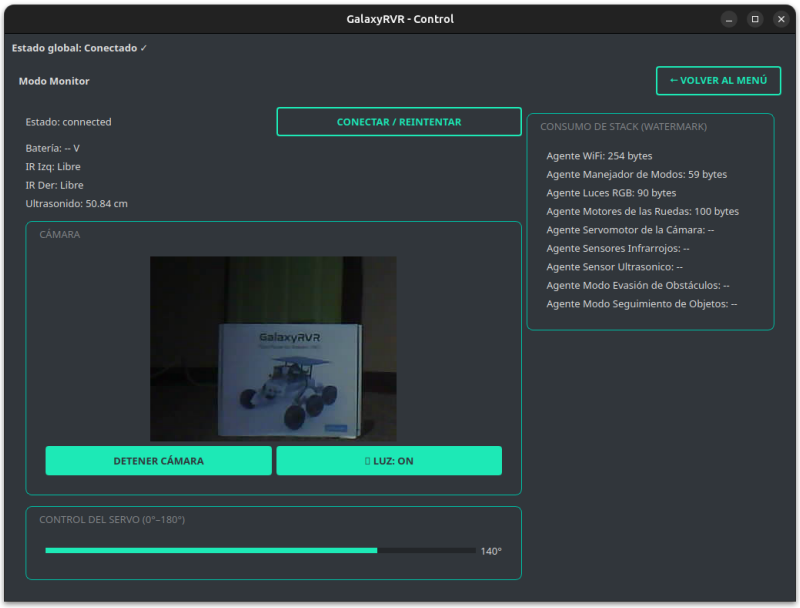

Figura 3.4: Modo Monitor mostrando telemetría, sensores, información del sistema y streaming del video.

3.4 Reflexiones Finales

El desarrollo de la aplicación representó un elemento clave en la integración entre el sistema embebido y las herramientas de supervisión externas. A lo largo del proceso se evidenció la importancia de diseñar arquitecturas de software flexibles y desacopladas, capaces de adaptarse a entornos de comunicación no deterministas como los enlaces inalámbricos utilizados por el rover.

El enfoque basado en eventos y tareas asincrónicas permitió mantener la estabilidad del sistema incluso ante pérdidas temporales de conexión, resaltando la importancia del manejo de excepciones y estrategias de reconexión en sistemas distribuidos. Asimismo, el proyecto permitió reflexionar sobre la necesidad de contar con una interfaz que no solo sirva para operar el rover, sino también como una *plataforma de experimentación* para evaluar el desempeño de los agentes embebidos, los algoritmos de navegación y la eficiencia en la gestión de recursos del sistema operativo.

Capítulo 4

Diseño del sistema multi-agentes

En esta sección se presenta el proceso de diseño de los agentes que conforman el sistema multiagente propuesto para el rover GalaxyRVR. Para su implementación se utilizará la biblioteca FreeMAES (Free Multi-Agent Embedded Systems), la cual permite desarrollar y coordinar agentes en sistemas embebidos de manera modular y escalable. El diseño seguirá el flujo metodológico de desarrollo de agentes mostrado en la figura 4.1, tomada de la tesis doctoral de CARVAJAL-GODÍNEZ [1], que abarca desde la fase de análisis hasta la fase de implementación. Este enfoque asegura que cada agente sea definido en función de sus responsabilidades, interacciones y recursos, permitiendo estructurar el sistema de forma organizada y orientada a los objetivos de control y validación de algoritmos de misión crítica.

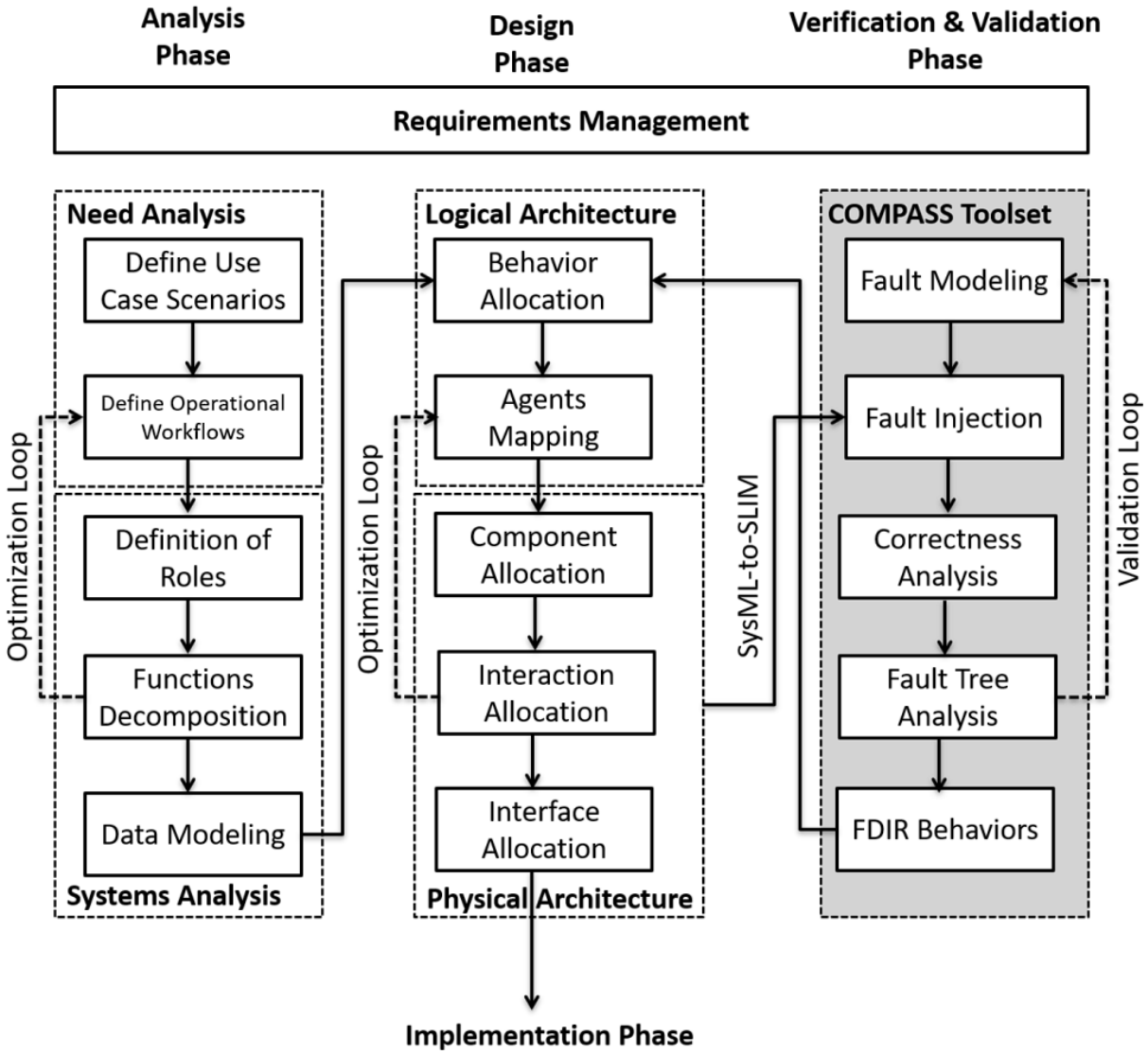

Figura 4.1: Flujo metodológico de desarrollo de agentes, tomado de [1].

4.1 Fase de Análisis

Definición de casos de uso

En el marco del sistema multiagente propuesto, se definen los siguientes casos de uso del rover, cada uno asociado a un modo de operación específico:

Caso de Control Manual: El usuario controla directamente al rover mediante la aplicación, utilizando controles gráficos como *sliders* y botones que funcionan como un control remoto. De esta forma, puede ajustar en tiempo real la potencia de los motores izquierdo y derecho, el ángulo del servomotor y la lámpara de la cámara.

Caso de Control Sistemático: La aplicación permite al usuario cargar un archivo en formato `.csv` que contiene

una secuencia de instrucciones predefinidas. El rover ejecuta dichas instrucciones en el orden establecido, permitiendo la realización de pruebas repetibles y la automatización de secuencias de movimientos o comportamientos. Además, este caso puede activar los modos autónomos como parte de la secuencia planificada.

Caso de Evitación de Obstáculos: Inicia el control autónomo; este caso se activa desde la aplicación o como parte de un plan sistemático. El rover avanza en línea recta hasta identificar un obstáculo mediante los sensores infrarrojos o el sensor ultrasónico; al detectarlo, ejecuta una maniobra de giro para esquivarlo y continuar su recorrido.

Caso de Seguimiento de Objetos: También corresponde al control autónomo activado desde la aplicación o desde un plan sistemático. En este caso, el rover se mantiene detenido hasta detectar un objeto mediante los sensores infrarrojos o el sensor ultrasónico; una vez detectado, procede a seguirlo, ajustando su trayectoria en función del desplazamiento del objeto.

Definición de flujo operacional

El flujo de operaciones describe la secuencia de funciones que ejecutaran los distintos agentes del sistema, desde la recepción de comandos de la aplicación hasta la ejecución de acciones en el rover y la retroalimentación mediante la telemetría de los sensores. La representación gráfica de este proceso puede observarse en la Figura 4.2.

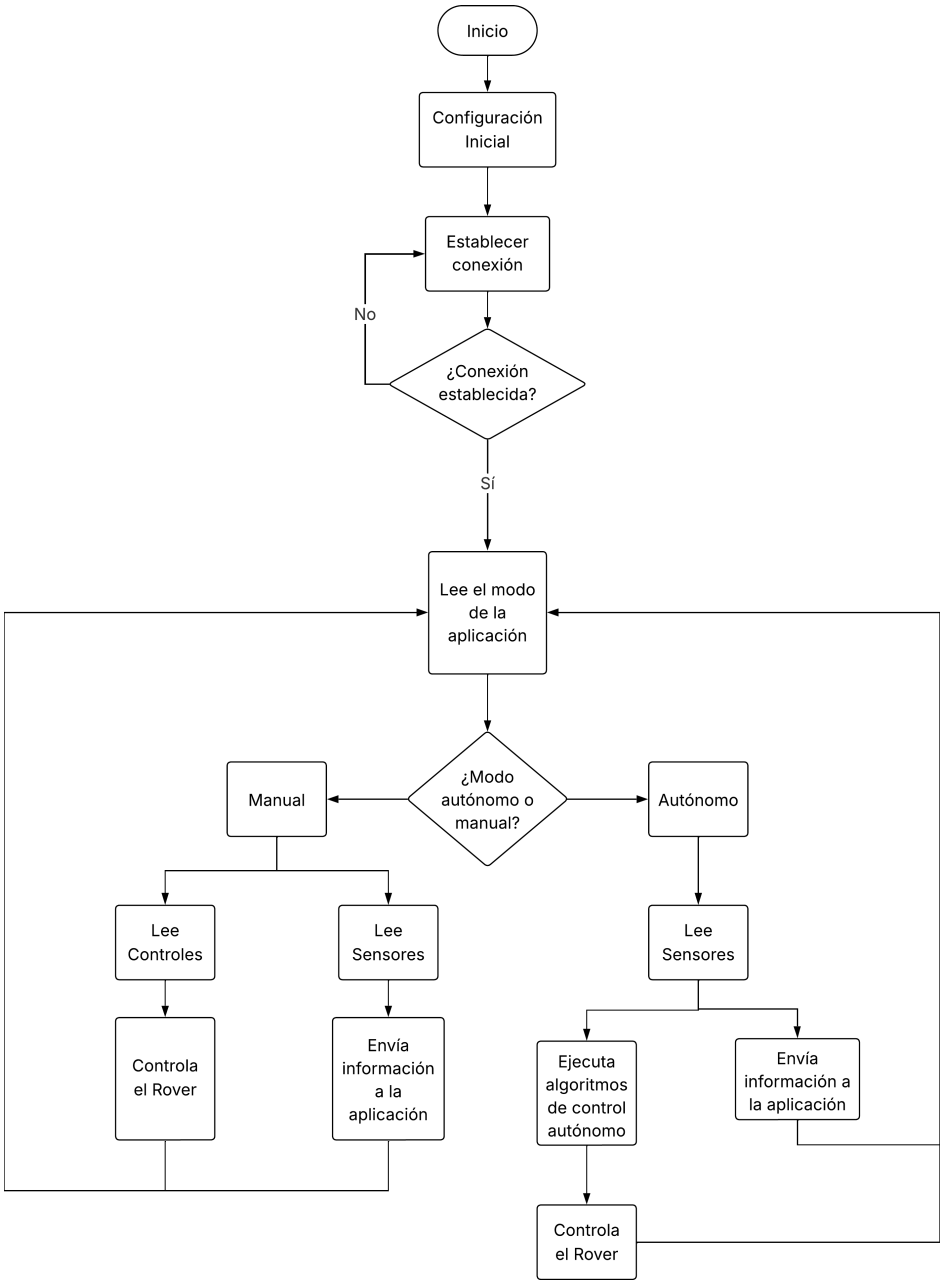

Figura 4.2: Diagrama de flujo de las operaciones de los agentes.

Definición de roles

En esta sección se definen los roles de cada agente del sistema multiagente, especificando sus responsabilidades principales dentro de la arquitectura del rover GalaxyRVR.

Agente	**Descripción del rol**
Conexión con la Aplicación	Establece y mantiene la conexión del GalaxyRVR con la aplicación mediante la red WiFi de la cámara y la librería *SunFounder AI Camera*. Recibe datos de la aplicación (modo, potencia de motores, ángulo del servomotor, estado de la luz de la cámara, color de las luces RGB) y los almacena para el Manejador de Modos. Además, solicita valores a los agentes de sensores y los envía a la aplicación.
Manejador de Modos	Determina el modo de operación actual del rover y ejecuta las tareas correspondientes. Activa los agentes de Seguimiento de Objetos y Evasión de Obstáculos, y envía instrucciones a los agentes de motores, servomotor y luces RGB.
Evasión de Obstáculos	Activado por el Manejador de Modos, ejecuta el algoritmo de evasión solicitando datos a los agentes de sensores infrarrojos y ultrasónico. Con esta información determina la navegación y envía instrucciones al agente de motores.
Seguimiento de Objetos	Activado por el Manejador de Modos, ejecuta el algoritmo de seguimiento solicitando datos a los agentes de sensores infrarrojos y ultrasónico. Determina la trayectoria del rover y envía instrucciones al agente de motores.
Control del Sensor Ultrasónico	Lee y comunica la distancia medida por el sensor ultrasónico. La información se envía al agente Conexión con la Aplicación y a los agentes de Evasión de Obstáculos y Seguimiento de Objetos.
Control de los Sensores Infrarrojos	Lee y comunica el estado de los sensores infrarrojos izquierdo y derecho. La información se envía al agente Conexión con la Aplicación, así como a los agentes de Evasión de Obstáculos y Seguimiento de Objetos.
Control de los Motores de las Ruedas	Inicializa y controla los motores del rover. Recibe instrucciones sobre dirección y potencia desde el Manejador de Modos o los agentes de Evasión de Obstáculos y Seguimiento de Objetos, según el modo activo.
Control del Servo de la Cámara	Inicializa y controla el servomotor de la cámara. Recibe únicamente instrucciones del Manejador de Modos, específicamente el ángulo al que debe posicionarse.
Control de Luces RGB	Administra el encendido, apagado y color de las luces RGB. Su función es brindar una indicación visual del modo de operación actual del rover, comunicándose únicamente con el Manejador de Modos.

Cuadro 4.1: Definición de roles de los agentes del sistema multiagente.

Descomposición de funciones

En esta sección se presentan las funciones asignadas a cada agente del sistema multiagente, con el fin de detallar las responsabilidades específicas de cada uno.

Agente	**Funciones**
Conexión con la Aplicación	- Recibir valores: Guarda los parámetros recibidos por WiFi de la cámara en variables accesibles al Manejador de Modos. - Enviar valores: Solicita lecturas a los agentes de sensores y las envía a la aplicación, junto con el estado de la batería. - Establecer modo: Define el modo de operación del rover según los casos de uso. - Detener el rover: Cancela todas las órdenes activas y coloca al rover en condiciones iniciales. - Encender la lámpara: Comunica a la cámara la orden de encender la lámpara.
Manejador de Modos	- Inicializar modos: Lee el modo actual e inicializa el comportamiento correspondiente, enviando instrucciones a motores, servomotor, luces RGB, y a los agentes de Evasión y Seguimiento.
Evasión de Obstáculos	- Leer sensores infrarrojos: Solicita el estado al agente de sensores IR para el algoritmo de evasión. - Leer sensor ultrasónico: Solicita la distancia al agente ultrasónico. - Enviar instrucciones al motor: Ordena al agente de motores según el resultado del algoritmo.
Seguimiento de Objetos	- Leer sensores infrarrojos: Solicita el estado al agente de sensores IR para el algoritmo de seguimiento. - Leer sensor ultrasónico: Solicita la distancia al agente ultrasónico. - Enviar instrucciones al motor: Ordena al agente de motores según el resultado del algoritmo.
Control del Sensor Ultrasónico	- Leer sensor ultrasónico: Obtiene la distancia medida. - Indicar obstáculo: Determina si existe un obstáculo con base en la distancia. - Indicar camino libre: Determina si el recorrido está despejado.
Control de los Sensores Infrarrojos	- Inicializar sensores: Configura pines como entradas en el Arduino. - Leer sensores: Obtiene el estado de los sensores (obstruido o libre).
Control de los Motores de las Ruedas	- Inicializar motores: Configura pines de salida en el Arduino. - Avanzar: Aplica potencia positiva a ambos motores. - Retroceder: Aplica potencia negativa a ambos motores. - Girar a la izquierda: Motor izquierdo negativo y derecho positivo. - Girar a la derecha: Motor izquierdo positivo y derecho negativo. - Detener motores: Potencia cero en ambos motores. - Definir potencia específica: Ajusta una potencia determinada para cada motor.
Control del Servo de la Cámara	- Inicializar servo: Configura el pin correspondiente como salida. - Definir ángulo: Convierte un valor de ángulo en el pulso de control para posicionar el servomotor.
Control de Luces RGB	- Inicializar luces: Configura pines como salidas. - Cambiar color: Recibe un valor hexadecimal y aplica intensidades RGB. - Apagar luces: Desactiva las salidas de los pines de las luces.

Cuadro 4.2: Descomposición de funciones de los agentes del sistema multiagente.

Modelado de datos

En esta sección se describe el modelado de datos utilizado por los agentes del sistema multiagente. Cada agente que interactúa directamente con hardware (sensores o actuadores) requiere estructuras de datos específicas para el envío y recepción de información. Dichas estructuras se definen mediante enumeraciones de instrucciones y paquetes de control implementados en C++. Los demás agentes (como el Manejador de Modos, Evasión de Obstáculos, Seguimiento de Objetos y Conexión con la Aplicación) utilizan estas mismas estructuras sin necesidad de definir paquetes propios.

Agente	**Modelado de Datos**
Control de Luces RGB	**Enumeración:** `rgbInstructions` (`uint8_t`) con valores `Begin`, `Write`, `Off`. **Paquete:** `rgbControlPackage` con campos: - `instruction` (`rgbInstructions`): instrucción a ejecutar. - `color` (`uint32_t`): color en formato `0xRRGGBB`. - `red` (`uint8_t`): intensidad del canal rojo. - `green` (`uint8_t`): intensidad del canal verde. - `blue` (`uint8_t`): intensidad del canal azul.
Control de Motores	**Enumeración:** `carInstructions` (`uint8_t`) con valores `Begin`, `Forward`, `Backward`, `TurnLeft`, `TurnRight`, `Stop`, `SetMotors`. **Paquete:** `carControlPackage` con campos: - `instruction` (`carInstructions`): instrucción de movimiento. - `power1` (`int8_t`): potencia del motor izquierdo. - `power2` (`int8_t`): potencia del motor derecho.
Control de Sensores Infrarrojos	**Enumeración:** `irInstructions` (`uint8_t`) con valores `Begin`, `Read`. **Paquete:** `irControlPackage` con campos: - `instruction` (`irInstructions`): instrucción solicitada. - `result` (`uint8_t`): estado de los sensores IR (bit 7 = izquierdo, bit 6 = derecho).
Control del Servo de la Cámara	**Enumeración:** `servoInstructions` (`uint8_t`) con valores `Begin`, `Attach`, `Write`. **Paquete:** `servoControlPackage` con campos: - `instruction` (`servoInstructions`): instrucción a ejecutar. - `pin` (`uint8_t`): pin de conexión del servo. - `angle` (`uint8_t`): ángulo de posicionamiento del servo.
Control del Sensor Ultrasónico	**Enumeración:** `ultrasonicInstructions` (`uint8_t`) con valores `Read`, `IsObstacle`, `IsClear`. **Paquete:** `ultrasonicPackage` con campos: - `instruction` (`ultrasonicInstructions`): instrucción a ejecutar. - `distance` (`float`): distancia medida por el sensor. - `obstacle` (`bool`): indica si hay un obstáculo cercano. - `clear` (`bool`): indica si el camino está libre.

Cuadro 4.3: Modelado de datos de los agentes del sistema multiagente con tipos de variables.

4.2 Fase de Diseño

Asignación de comportamientos

En esta sección se presentan los comportamientos asignados a cada función del sistema multiagente, mostrando sus entradas, la lógica de procesamiento, las salidas generadas y los casos de uso en los que participan.

Agente	**Entrada**	**Comportamiento**	**Salida**
Control de motores	- Instrucción - Potencia de los motores	Traduce la instrucción y parámetros recibidos en órdenes de movimiento (avanzar, retroceder, girar, detener, ajustar potencia) aplicadas a cada rueda.	Potencia aplicada a ambos motores.
Control de servo	- Instrucción - Ángulo	Interpreta la instrucción y el ángulo objetivo; genera la señal de control para posicionar el servomotor de la cámara.	Ángulo aplicado al servo.
Control de luces RGB	- Instrucción - Color (Hexadecimal)	Interpreta la instrucción y los valores de color/intensidad; actualiza el estado visual para indicar el modo u otras señales.	Color e intensidad aplicado a las luces.
Control de sensor de IR	Solicitud de datos	Lee el estado de los sensores infrarrojos (izquierdo/derecho), consolida la lectura y prepara la respuesta.	- Instrucción - Estado de los sensores
Control de sensor ultrasónico	Solicitud de datos	Realiza medición de distancia y evalúa condiciones (obstáculo / camino libre); prepara la respuesta de telemetría.	- Instrucción - Distancia - Indica si hay un obstáculo o no
Evasión de obstáculos	Inicialización del agente	Ejecuta el algoritmo de evasión: consulta lecturas de IR/ultrasónico, decide giros/avance y genera órdenes de movimiento.	Instrucciones para *Control de motores*.
Seguimiento de objetos	Inicialización del agente	Ejecuta el algoritmo de seguimiento: consulta lecturas de IR/ultrasónico, calcula la trayectoria y genera órdenes de movimiento.	Instrucciones para *Control de motores*.
Manejador de modos	Lectura del modo actual	Orquesta el comportamiento del sistema: configura servo, luces y motores según el modo; inicializa (cuando corresponda) los agentes de evasión y seguimiento.	Inicializa agentes de control autónomo Envía el paquete correspondiente a: *-Control de servo* *-Control de luces RGB* *-Control de motores*
Conexión a la aplicación	- Comandos desde la aplicación - Telemetría de sensores	Mantiene la sesión, normaliza y distribuye comandos; integra telemetría para la interfaz y actualiza el modo leído.	Envía telemetría a la aplicación Define/actualiza el modo actual.

Cuadro 4.4: Asignación de comportamientos por agente, basada en entradas, procesamiento y salidas.

Mapeo de agentes

El mapeo de agentes describe cómo se organizan y comunican entre sí los distintos componentes del sistema multiagente. Este mapeo es fundamental porque permite identificar los flujos de información, las dependencias y las interacciones entre agentes de sensores, actuadores, control autónomo, el manejador de modos y la conexión con la aplicación. De esta forma se logra una visión integral de la arquitectura del sistema y se asegura que cada agente cumpla su función en coordinación con los demás. El mapeo de las interacciones entre agentes puede observarse en la imagen siguiente.

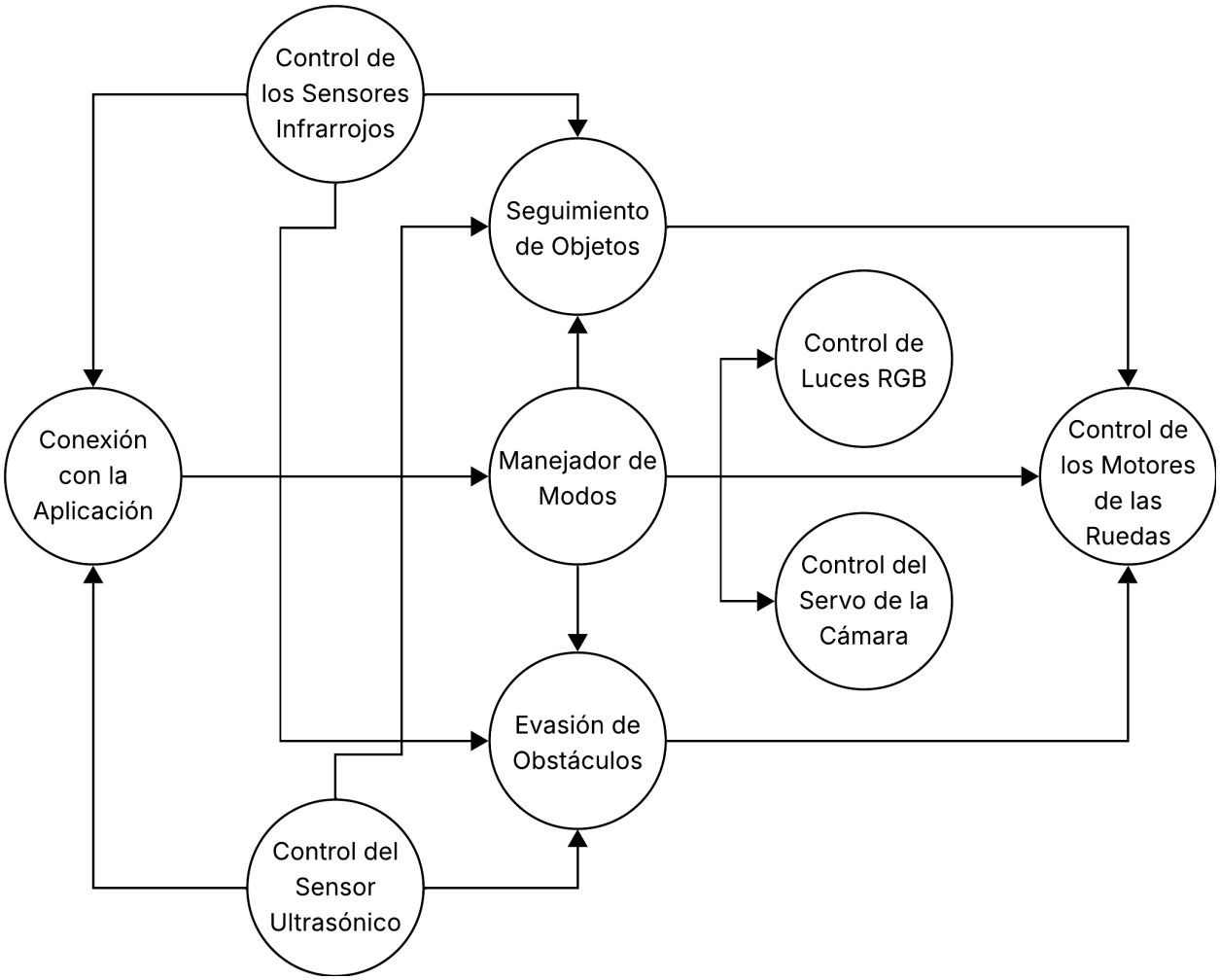

Figura 4.3: Mapeo de agentes

4.3 Arquitectura física

La arquitectura física del sistema multiagente implementado en el rover GalaxyRVR establece la relación entre los agentes definidos en el software y los recursos físicos disponibles en el entorno embebido. Esta sección describe la asignación de componentes y de interfaces, tanto internas como externas, que permiten la interacción entre el sistema operativo, los agentes y los periféricos del rover.

Asignación de componentes

La asignación de componentes corresponde a la vinculación entre los agentes del sistema multiagente y los recursos físicos del hardware embebido. Cada agente se ejecuta dentro de su propio hilo de **FreeRTOS**, con una asignación estándar de **192 bytes de SRAM** para su pila de ejecución. La Tabla 4.7 muestra la relación entre cada agente y los elementos físicos que controla o supervisa dentro del sistema del rover GalaxyRVR.

Agente	**Recursos asignados**	**Descripción**
Conexión con la aplicación	ESP32-CAM 192 bytes de SRAM	Gestiona la comunicación entre el rover y la aplicación de control mediante el protocolo WebSockets. Se encarga de recibir comandos, enviar telemetría y mantener la conexión WiFi estable.
Control de sensores infrarrojos	2 sensores IR 192 bytes de SRAM	Lee el estado de los sensores infrarrojos (izquierdo y derecho) para detectar obstáculos o líneas, enviando los datos al manejador de modos y a los agentes autónomos.
Control del sensor ultrasónico	Sensor ultrasónico 192 bytes de SRAM	Mide la distancia a objetos frontales y determina la presencia de obstáculos. Provee esta información al manejador de modos y a los algoritmos de evasión o seguimiento.
Control de luces RGB	4 tiras de LEDs RGB 192 bytes de SRAM	Actualiza el color e intensidad de las luces RGB para representar el estado del sistema o el modo activo. Controlado mediante la librería *SoftPWM*.
Control del servo de la cámara	1 servomotor 192 bytes de SRAM	Controla la posición del servomotor de la cámara del rover, ajustando su ángulo según las órdenes del manejador de modos o la aplicación.
Control de motores de las ruedas	6 motores TT 192 bytes de SRAM	Gestiona el movimiento del rover, controlando la dirección y potencia de cada rueda en función de los comandos de navegación recibidos.
Manejador de modos	192 bytes de SRAM	Coordina el comportamiento general del sistema, activando los modos manual, evasión de obstáculos o seguimiento de objetos según la orden recibida.
Evasión de obstáculos	192 bytes de SRAM	Ejecuta el algoritmo de evasión utilizando los datos de sensores IR y ultrasónico. Genera comandos de giro o retroceso para evitar colisiones.
Seguimiento de objetos	192 bytes de SRAM	Implementa el algoritmo de seguimiento, utilizando las lecturas de sensores para mantener al rover orientado hacia un objeto objetivo.

Cuadro 4.5: Asignación de componentes físicos a los agentes del sistema multiagente.

Cada agente opera de forma independiente dentro de su hilo de ejecución, compartiendo únicamente los recursos del sistema operativo como el planificador de tareas, colas de comunicación y temporizadores. Esta asignación asegura una distribución uniforme de los recursos y mejora la capacidad de respuesta del sistema bajo condiciones concurrentes.

Asignación de interfaces

La asignación de interfaces describe los medios de comunicación entre el hardware del rover, los agentes del sistema y la aplicación externa. Se distinguen tres niveles principales: interfaces de hardware, interfaces entre agentes e interfaces externas.

Interfaces del hardware

- **Galaxy RVR Shield:** Placa desarrollada por SunFounder que actúa como interfaz entre el Arduino y los periféricos del rover. Está conectada a todos los pines del Arduino y simplifica la conexión de sensores y actuadores.
- **ESP32-CAM:** Módulo encargado de la transmisión de video y datos mediante la librería *SunFounder AI Camera*. Se comunica con el Arduino a través de los pines `Rx` y `Tx`.
- **Placa adaptadora de cámara:** Interconecta la ESP32-CAM con el Galaxy RVR Shield, facilitando el montaje físico y el cableado de la cámara.
- **Módulo ultrasónico:** Conectado directamente al Galaxy RVR Shield. Utiliza el pin digital externo `GPIO 10` del Arduino para la lectura de distancia.
- **Módulo infrarrojo:** Conectado al Galaxy RVR Shield, utiliza los pines digitales externos `GPIO 7` y `GPIO 8` para la detección de obstáculos.
- **Cintas de LEDs RGB:** Conectadas al Galaxy RVR Shield. Usan la librería *SoftPWM* y los pines `GPIO 11`, `GPIO 12` y `GPIO 13` para el control de color e intensidad.
- **Servomotor:** Conectado al pin `GPIO 6`, controlado mediante *SoftPWM*. Permite la orientación de la cámara del rover.
- **Motores TT:** Conectados directamente al Galaxy RVR Shield. Utilizan los pines digitales `GPIO 0--5` para el control de velocidad y dirección mediante señales PWM generadas por *SoftPWM*.

Interfaces entre agentes

Los agentes del sistema se comunican a través del **API de FreeMAES**, el cual implementa un sistema de colas y estructuras de mensajes estandarizadas. Cada agente puede enviar y recibir información del *Agent Management System (AMS)* o directamente de otros agentes, lo que permite mantener la coherencia de datos y la sincronización entre tareas concurrentes.

Interfaces externas

El módulo **ESP32-CAM**, junto con la librería *SunFounder AI Camera*, utiliza el protocolo de red **WebSockets** para establecer comunicación bidireccional con la aplicación de control del rover. A través de esta interfaz, se transmiten

los comandos de usuario, los datos de telemetría de sensores y las imágenes en tiempo real, permitiendo un control remoto completo del sistema.

4.4 Adaptación de la librería FreeMAES a la familia AVR de Arduino

La librería **FreeMAES** fue originalmente desarrollada para entornos de escritorio o simuladores, haciendo uso de funciones estándar de C (`printf`, `conio.h`, `kbhit`, etc.) y dependencias de **FreeRTOS** en plataformas con amplia disponibilidad de memoria. El objetivo de esta adaptación fue lograr que la librería funcionara correctamente en el entorno embebido del **Arduino UNO R3**, basado en el microcontrolador **ATmega328P**, con los siguientes recursos limitados:

- **SRAM:** 2 KB
- **Flash:** 32 KB
- **Arquitectura:** 8 bits, single-core
- **Sistema operativo:** FreeRTOS (Arduino_FreeRTOS)

Eliminación de dependencias no compatibles con Arduino

En los archivos originales de la librería (`supporting_functions.c` y `maes-rtos.h`) se eliminaron las referencias a librerías de consola como `<conio.h>` y funciones como `kbhit()` y `printf()`, sustituyéndolas por el uso del puerto serie de Arduino.

Ejemplo de modificación:

Listing 4.1: Redirección de salida estándar al puerto serie de Arduino.

```
// Original:
printf("Agent␣initialized\n");

// Arduino:
Serial.println(F("Agent␣initialized"));
```

Esto permitió redirigir toda la salida estándar al *Serial Monitor* del Arduino IDE, eliminando dependencias del entorno de PC.

Compatibilización de `supporting_functions.c`

El archivo `supporting_functions.c` fue reescrito para integrarse al entorno de Arduino:

- Inclusión de `<Arduino.h>` en lugar de `<stdio.h>` o `<conio.h>`.
- Reemplazo de funciones de impresión por `Serial.print()` y `Serial.println()`.
- Eliminación de rutinas basadas en interrupciones de teclado (`kbhit()`).

De esta manera, el módulo de soporte se transformó en un componente universal, compatible tanto con FreeRTOS en PC como en microcontroladores AVR.

Ajustes en el núcleo de FreeMAES

En los archivos principales de FreeMAES (`Agent.cpp`, `Agent_Platform.cpp`, `Agent_Msg.cpp`, etc.) se realizaron los siguientes ajustes:

- Sustitución de funciones dependientes del sistema operativo (`printf`, `Sleep`) por equivalentes en FreeRTOS y Arduino (`Serial.print`, `vTaskDelay`).
- Modificación de la función `Agent_Platform::get_state()` para soportar el port de FreeRTOS en Arduino:
  - `eTaskGetState()` no estaba habilitada por defecto.
  - Se agregó una verificación condicional de compilación:

Listing 4.2: Modificacion de `Agent_Platform::get_state()`

```
#ifdef INCLUDE_eTaskGetState
    eTaskGetState();
#else
    return NO_MODE;
#endif
```

Integración con FreeRTOS para Arduino

Se empleó la librería **Arduino_FreeRTOS** (v11.x), instalada mediante el *Library Manager* del Arduino IDE. Se revisó y ajustó el archivo `FreeRTOSConfig.h` para incluir las siguientes definiciones:

Listing 4.3: Configuraciones clave en FreeRTOSConfig.h.

```
#define INCLUDE_uxTaskGetStackHighWaterMark 1
#define INCLUDE_eTaskGetState 0
#define configUSE_MUTEXES 0
#define configSUPPORT_DYNAMIC_ALLOCATION 1
```

Con estas modificaciones, FreeMAES pudo crear tareas, colas y mensajes dentro de los límites de memoria del ATmega328P.

Corrección del flujo de inicialización

Se verificó que el flujo de arranque de FreeMAES, a través de la función `AP.boot()`, funcionara correctamente bajo FreeRTOS en Arduino.

- `vTaskStartScheduler()` se ejecuta desde la función `setup()`.
- La función `loop()` se elimina, ya que FreeRTOS gestiona las tareas concurrentes.

Optimización de memoria (SRAM)

Dadas las limitaciones de 2 KB de SRAM, se realizaron optimizaciones para evitar desbordamientos:

- Reducción del tamaño de pila (`stackSize`) de los agentes a 192 palabras.
- Uso del macro `F()` para almacenar cadenas constantes en memoria Flash:
- Monitoreo del stack mediante `uxTaskGetStackHighWaterMark()` para determinar el tamaño mínimo seguro.

Validación final

La validación de la adaptación de la biblioteca **FreeMAES** al entorno embebido del **Arduino UNO R3** se realizó ejecutando los tres programas de demostración incluidos en la librería: *Sender Receiver*, *Rock Paper Scissors* y *Telemetry*. Estos programas permiten comprobar el funcionamiento del sistema multiagente bajo diferentes condiciones de comunicación y carga de procesamiento, confirmando la correcta interacción entre agentes, colas de mensajes y el *Agent Management System (AMS)*.

Resultados de la simulación

Durante la validación experimental se obtuvieron los siguientes resultados:

- El programa **Sender Receiver** funcionó correctamente, mostrando la comunicación estable entre los agentes *Sender* y *Receiver*. 4.4
- Los programas **Rock Paper Scissors** y **Telemetry** no lograron iniciar su ejecución; ambos quedaron bloqueados al inicio del proceso de arranque. 4.5 4.6

El análisis posterior indicó que el **Arduino UNO R3** solo dispone de suficiente memoria SRAM para ejecutar tres agentes (con 192 bytes de memoria SRAM asignadas) de manera simultánea, lo cual explica el comportamiento observado. El demo *Sender Receiver* requiere exactamente tres agentes (Sender, Receiver y AMS), mientras que *Rock Paper Scissors* y *Telemetry* utilizan cuatro y cinco agentes respectivamente, superando así los 2 kB de SRAM disponibles.

```
22:51:06.449 -> FreeMAES Sender/Receiver Demo
22:51:06.449 -> Sender Agent Inicializado
22:51:06.449 -> Receiver Agent Inicializado
22:51:06.449 -> Boot exitoso
22:51:06.493 -> Esperando mensaje...
22:51:06.493 -> Enviando mensaje...
22:51:06.493 -> Mensaje recibido: Hola MAES!
22:51:06.493 -> Esperando mensaje...
22:51:07.443 -> Enviando mensaje...
22:51:07.443 -> Mensaje recibido: Hola MAES!
```

Figura 4.4: Salida del monitor serial del demo Sender Receiver en Arduino UNO R3.

```
22:56:27.225 -> FreeMAES RPS Demo
22:56:27.225 -> Player A Agent Inicializado
22:56:27.225 -> Player B
```

Figura 4.5: Salida del monitor serial del demo Rock Paper Scissors en Arduino UNO R3.

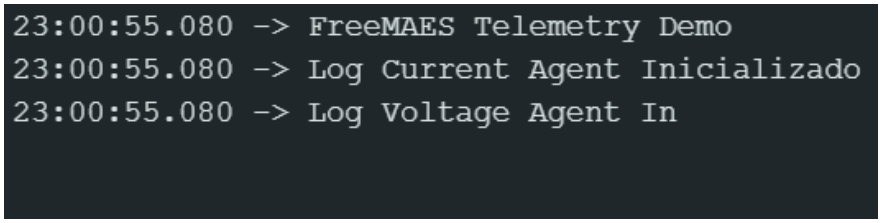

Figura 4.6: Salida del monitor serial del demo Telemetry en Arduino UNO R3.

El codigo de los programas de demostracion *Sender Receiver*, *Rock Paper Scissors* y *Telemetry* se encuentran en el repositorio de Github FreeMAES-for-ATmega-Devices [29].

Identificación del problema

Este hallazgo representa una limitación significativa para el proyecto, ya que el sistema multiagente diseñado a lo largo de este capítulo contempla un total de **nueve agentes**, lo cual excede ampliamente las capacidades de memoria del microcontrolador ATmega328P. Por tanto, se hizo necesario evaluar estrategias para continuar el desarrollo sin comprometer los objetivos del sistema ni su modularidad.

Opciones de solución

A partir del análisis anterior, se consideraron dos alternativas principales:

Opción A: Reducción del sistema a 2 agentes + AMS Ventajas:

- Permite conservar el hardware actual (Arduino UNO R3) más la compatibilidad con el GalaxyRVR.
- Reduce significativamente el consumo de SRAM.

Desventajas:

- Se pierden muchas de las ventajas del enfoque multiagente (autonomía y modularidad).
- Aumenta la complejidad del código de cada agente al concentrar más funciones.

Opción B: Migrar a una placa con mayor capacidad de memoria SRAM Ventajas:

- Permite implementar sistemas multiagentes más completos y realistas.
- Facilita el diseño modular, manteniendo responsabilidades bien separadas entre agentes.

Desventajas:

- Riesgo de incompatibilidad con el *GalaxyRVR Shield* y los periféricos actuales.
- Posible necesidad de adaptar nuevamente la biblioteca FreeMAES al nuevo hardware.

Decisión adoptada

Inicialmente se optó por la **Opción A**, intentando implementar un sistema reducido de dos agentes más el AMS. Sin embargo, esta alternativa también fracasó: aunque el número de agentes disminuyó, cada uno debía manejar más responsabilidades, aumentando su consumo de memoria hasta sobrepasar nuevamente los 2 kB disponibles.

Por este motivo se adoptó la **Opción B**. Se seleccionó el **Arduino MEGA 2560** como reemplazo del UNO R3, dado que pertenece a la misma familia AVR y ofrece **8 kB de SRAM** además de mantener el mismo *pinout*, lo que asegura la compatibilidad total con el **GalaxyRVR Shield** y sus periféricos.

Tras la migración, se realizaron pruebas adicionales que confirmaron la correcta ejecución de los programas de demostración y la estabilidad del sistema, tal como se muestra en las figuras 4.7, 4.8 y 4.9. No obstante, se determinó que incluso con 8 kB de SRAM, un sistema con **9 agentes + AMS** supera el límite práctico de memoria disponible. Se determino que con 8 kB se pueden implementar no más de 7 agentes (con 192 bytes de memoria SRAM asignadas).

```
10:32:51.326 -> FreeMAES Sender/Receiver Demo
10:32:51.326 -> Sender Agent Inicializado
10:32:51.326 -> Receiver Agent Inicializado
10:32:51.326 -> Boot exitoso
10:32:51.326 -> Esperando mensaje...
10:32:51.326 -> Enviando mensaje...
10:32:51.326 -> Mensaje recibido: Hola MAES!
10:32:51.326 -> Esperando mensaje...
10:32:52.327 -> Enviando mensaje...
10:32:52.327 -> Mensaje recibido: Hola MAES!
```

Figura 4.7: Salida del monitor serial del demo Sender Receiver en Arduino MEGA.

```
10:29:55.349 -> FreeMAES RPS Demo
10:29:55.350 -> Player A Agent Inicializado
10:29:55.350 -> Player B Agent Inicializado
10:29:55.350 -> Referee Agent Inicializado
10:29:55.350 -> Boot exitoso
10:29:55.350 ->
10:29:55.350 -> REFEREE READY
10:29:55.350 -> Player A: Rock, Paper, Scissors...
10:29:55.350 -> Player B: Rock, Paper, Scissors...
10:29:55.430 -> Playing now: Player A -> PAPER
10:29:55.430 -> Playing now: Player B -> PAPER
10:29:55.430 -> DRAW!
```

Figura 4.8: Salida del monitor serial del demo Rock Paper Scissors en Arduino MEGA.

```
10:37:03.333 -> FreeMAES Telemetry Demo
10:37:03.333 -> Log Current Agent Inicializado
10:37:03.333 -> Log Voltage Agent Inicializado
10:37:03.333 -> Log Temperature Agent Inicializado
10:37:03.333 -> Measurement Agent Inicializado
10:37:03.333 -> Boot exitoso
10:37:03.333 ->
10:37:03.333 -> Current measurement: 807.02 mA
10:37:03.333 -> [Current Logger] OK
10:37:03.333 -> --------------------------------
10:37:03.333 -> Voltage measurement: 1.20 V
10:37:03.333 -> [Voltage Logger] OK
10:37:03.333 -> --------------------------------
10:37:03.333 -> Temperature measurement: 35.11 °C
10:37:03.375 -> [Temperature Logger] OK
10:37:03.375 -> --------------------------------
```

Figura 4.9: Salida del monitor serial del demo Telemetry en Arduino MEGA.

4.5 Rediseño del sistema

Finalmente, se realizó un **rediseño de la arquitectura multiagente**, reduciendo la cantidad total de agentes a **cuatro más el AMS**. Esta configuración permitió disminuir considerablemente el uso de SRAM sin perder los beneficios principales del enfoque multiagente, como la modularidad, la concurrencia y la tolerancia a fallos.

Este resultado marcó la consolidación de la adaptación de FreeMAES en microcontroladores AVR de recursos limitados, validando su funcionamiento estable y demostrando la necesidad de balancear cuidadosamente la arquitectura de software con las restricciones físicas del hardware.

Reestructuración de agentes

Los cambios principales fueron los siguientes:

- Los agentes de **Control de motores**, **Control de servo** y **Control de luces RGB** se combinaron en un único agente denominado **Control de periféricos**.
- Los agentes **Control de sensor infrarrojo (IR)** y **Control de sensor ultrasónico** se fusionaron en un solo agente llamado **Control de sensores**.
- Los agentes **Manejador de modos**, **Evasión de obstáculos** y **Seguimiento de objetos** se integraron en un único agente denominado **Manejador de modos**.
- El agente de **Conexión con la aplicación** se mantuvo sin modificaciones, ya que gestiona las comunicaciones externas mediante WiFi y constituye un proceso fundamental del sistema.

Esta reorganización permitió reducir significativamente el número total de tareas activas y, por ende, el espacio reservado en el *stack*, manteniendo la modularidad y las capacidades de ejecución concurrente de FreeMAES.

Asignación de comportamientos

En la Tabla 4.6 se presentan los comportamientos asignados a cada agente, especificando sus entradas, la lógica de procesamiento principal, las salidas generadas y los casos de uso en los que participan.

Agente	**Entradas**	**Comportamiento**	**Salidas**
Control de periféricos	- Instrucciones de movimiento o acción. - Parámetros (potencia, ángulo, color).	Interpreta las instrucciones recibidas desde el manejador de modos o la aplicación. Traduce los comandos en señales físicas aplicadas a motores, servomotor de cámara y luces RGB.	Potencia aplicada a los motores. Ángulo del servo. Color e intensidad de las luces.
Control de sensores	Solicitud de datos desde otros agentes.	Lee los sensores infrarrojos (izquierdo y derecho) y el sensor ultrasónico. Combina las lecturas para determinar la presencia de obstáculos o caminos libres.	Valores combinados de sensores IR y distancia medida por ultrasonido.
Manejador de modos	Lectura del modo actual y estados del sistema.	Orquesta el comportamiento global del rover. Dependiendo del modo activo (manual, evasión o seguimiento), configura el control de periféricos y analiza las lecturas de sensores para definir las acciones automáticas.	Comandos dirigidos a los agentes de *Control de periféricos* y *Control de sensores*. Paquetes de estado del modo actual.
Conexión a la aplicación	- Comandos desde la interfaz WiFi. - Telemetría desde los agentes internos.	Mantiene la sesión de comunicación, normaliza comandos recibidos y transmite los datos de telemetría y estado del sistema a la aplicación de control.	Actualización de telemetría. Confirmaciones de comando. Definición del modo activo.

Cuadro 4.6: Asignación de comportamientos por agente, basada en sus entradas, procesamiento y salidas.

La nueva estructura resultante mantiene el principio de independencia funcional de cada agente, mientras optimiza el uso de memoria y mejora la claridad en el flujo de ejecución del sistema multiagente.

Mapeo de Agentes

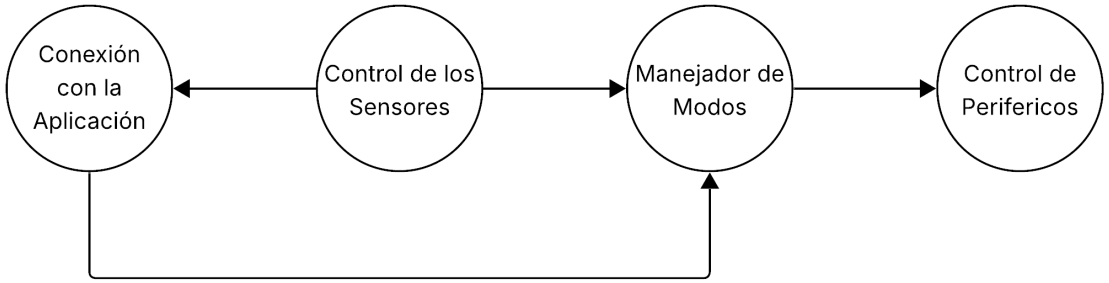

Figura 4.10: Mapeo de agentes

Asignación de componentes

La asignación de componentes corresponde a la vinculación entre los agentes del sistema y los recursos físicos del hardware embebido. Cada agente se ejecuta dentro de su propio hilo de **FreeRTOS**, con una asignación estándar de **192 bytes de SRAM** para su pila de ejecución. La Tabla 4.7 resume la relación entre los agentes y los elementos físicos que controlan o supervisan.

Agente	**Recursos asignados**	**Descripción**
Conexión con la aplicación	ESP32-CAM 500 bytes de SRAM	Gestiona la comunicación con la aplicación de control del rover mediante el módulo ESP32-CAM. Transmite y recibe datos usando la librería *SunFounder AI Camera*.
Control de sensores	2 sensores infrarrojos 1 sensor ultrasónico 192 bytes de SRAM	Obtiene y consolida lecturas de los sensores IR y ultrasónico para la detección de obstáculos y seguimiento de trayectorias.
Manejador de modos	192 bytes de SRAM	Coordina el comportamiento general del sistema, seleccionando entre modos manual, evasión de obstáculos o seguimiento de objetos.
Control de periféricos	6 motores TT 1 servomotor de cámara 4 tiras RGB LED 192 bytes de SRAM	Traduce las órdenes del manejador de modos o la aplicación en señales físicas para los motores, servo y luces RGB.

Cuadro 4.7: Asignación de componentes físicos a los agentes del sistema multiagente.

Cada agente mantiene independencia funcional, pero comparte recursos del sistema operativo, como el planificador de tareas, colas de comunicación y temporizadores proporcionados por FreeRTOS.

4.6 Resultados Obtenidos

Se logró implementar satisfactoriamente la biblioteca *FreeMAES* junto con el sistema operativo en tiempo real *FreeRTOS* en la familia de microcontroladores *Arduino AVR*, específicamente en los dispositivos *ATmega*. A pesar de que la implementación fue exitosa, se identificó que la memoria *SRAM* constituye una limitante significativa para la expansión del sistema.

Durante el proceso se desarrollaron y verificaron cuatro agentes funcionales, cada uno encargado de un conjunto de tareas esenciales para el correcto funcionamiento del *GalaxyRVR*. El **agente de conexión con la aplicación** fue responsable de mantener la comunicación WiFi y de gestionar el envío y recepción de datos con la interfaz gráfica. El **agente de control de periféricos** se encargó de controlar el servomotor de la cámara, las luces *RGB* y los motores de tracción. Por su parte, el **agente de control de sensores** administró la lectura de los sensores infrarrojos y del sensor ultrasónico, mientras que el **agente manejador de modos** controló la lógica de operación del sistema según el modo de funcionamiento, además de inicializar los modos de evasión de obstáculos y seguimiento de objetos.

El sistema multiagente implementado fue capaz de operar correctamente en los distintos casos de uso definidos: *control manual*, *control sistemático*, *control autónomo por evasión de obstáculos* y *control autónomo por seguimiento de objetos*. Estos resultados validan la funcionalidad y estabilidad de la arquitectura multiagente propuesta sobre un entorno embebido con recursos limitados.

Finalmente, para complementar los resultados obtenidos, en el Listing 4.4 se muestra el proceso de inicialización del sistema multiagentes durante el arranque del programa, donde cada agente es registrado y configurado antes de iniciar el planificador de *FreeRTOS*. Asimismo, las Figuras 4.11 y 4.12 ilustran la salida del monitor serial tanto en la etapa de arranque como durante la operación normal del sistema una vez que la aplicación de control establece conexión.

Listing 4.4: Inicialización del sistema multi-agentes en main.ino

```
#include <Arduino_FreeRTOS.h>
#include <task.h>
#include <queue.h>
#include <supporting_functions.h>
#include <maes-rtos.h>
#include <SoftPWM.h>

using namespace MAES;
Agent_Platform AP_GALAXY_RVR("Arduino_Mega");

#include "mode_handler_agent.h"
#include "wifi_agent.h"
#include "peripherals_control_agent.h"
#include "sensor_control_agent.h"

void setup() {
  Serial.begin(115200);
  SoftPWMBegin();
  while (!Serial) {;}

  Serial.println(F("FreeMAES \n"));

  AP_GALAXY_RVR.agent_init(&Mode_Handler_Agent, mode);
  Serial.println(F("Mode Handler Agent Inicializado"));
  AP_GALAXY_RVR.agent_init(&Peripherals_Control, peripherals);
```

```
26    Serial.println(F("Peripherals␣Control␣Agent␣Inicializado"));
27    AP_GALAXY_RVR.agent_init(&Sensor_Control, sensors);
28    Serial.println(F("Sensor␣Control␣Agent␣Inicializado"));
29    AP_GALAXY_RVR.agent_init(&WiFi_Agent, wifi);
30    Serial.println(F("WiFi␣Agent␣Inicializado"));
31
32    AP_GALAXY_RVR.boot();
33    Serial.println(F("boot␣exitoso␣\n"));
34
35    vTaskStartScheduler();
36    for (;;);
37  }
```

```
21:56:12.724 -> FreeMAES
21:56:12.724 ->
21:56:12.724 -> Mode Handler Agent Inicializado
21:56:12.724 -> Peripherals Control Agent Inicializado
21:56:12.724 -> Sensor Control Agent Inicializado
21:56:12.724 -> WiFi Agent Inicializado
21:56:12.724 -> boot exitoso
```

Figura 4.11: Salida del monitor serial al inicio del programa.

```
21:58:53.121 -> [CONNECTED] 192.168.4.2
21:58:53.156 -> WS+{"A1":258,"A2":59,"A3":93,"A4":112,"N":0,"P":0,"O":87.97}
21:58:53.294 -> WS+;;;90;0;0;;;;;0;;0;;;;0;;;;;;;;;
21:58:53.294 -> RX:
21:58:53.294 -> WS+;;;90;0;0;;;;;0;;0;;;;0;;;;;;;;;
21:58:53.294 -> WS+{"A1":258,"A2":59,"A3":93,"A4":112,"N":0,"P":0,"O":87.72}
21:58:53.387 -> WS+;;;90;0;0;;;;;0;;0;;;;0;;;;;;;;;
21:58:53.387 -> RX:
21:58:53.387 -> WS+;;;90;0;0;;;;;0;;0;;;;0;;;;;;;;;
21:58:53.387 -> WS+{"A1":258,"A2":59,"A3":93,"A4":112,"N":0,"P":0,"O":88.04}
21:58:53.453 -> WS+;;;90;0;0;;;;;0;;0;;;;0;;;;;;;;;
21:58:53.453 -> RX:
```

Figura 4.12: Salida del monitor serial durante la comunicación con la aplicación.

4.7 Reflexión Final

La metodología propuesta por CARVAJAL-GODÍNEZ[1] demostró ser una herramienta fundamental en el diseño del sistema multiagente, ya que permitió estructurar el proceso de desarrollo de manera ordenada e iterativa. Las etapas definidas en la metodología facilitaron la detección temprana de problemas y la generación de soluciones efectivas durante el ciclo de implementación.

La adaptación de la biblioteca *FreeMAES* al entorno *Arduino* representó un reto considerable, debido a la necesidad de sustituir las funciones y librerías específicas de Windows e integrarlas en una versión más simplificada del sistema operativo *FreeRTOS*. Aun así, el proceso fue exitoso y permitió validar la portabilidad de la arquitectura multiagente hacia plataformas de menor capacidad.

Resultó ventajoso que la familia *AVR* de *Arduino* incluyera dispositivos con mayores cantidades de memoria *SRAM*, lo que permitió reemplazar el *Arduino Uno R3* por versiones superiores sin modificar sustancialmente el diseño del

sistema. Sin embargo, la principal limitante de este proyecto fue precisamente la disponibilidad de memoria *SRAM*, factor que motivó múltiples rediseños en la arquitectura del sistema.

Gracias al enfoque modular de la metodología multiagente, fue posible ajustar el número de agentes de manera flexible. Finalmente, se adoptó una arquitectura de **cuatro agentes más el AMS (Agent Management System)**, en lugar de la versión original de nueve agentes, con el fin de optimizar el uso de los recursos disponibles y mantener un funcionamiento estable del sistema embebido.

Capítulo 5

Implementación Definitiva

El presente capítulo documenta los repositorios desarrollados durante el proyecto, los cuales contienen tanto el código fuente como las instrucciones necesarias para la instalación, configuración y uso del sistema completo implementado para el rover **GalaxyRVR**. El propósito de esta documentación es facilitar la **replicación, mantenimiento y reutilización** de los resultados obtenidos, permitiendo que otros investigadores o estudiantes del laboratorio SETEC puedan continuar el desarrollo futuro del sistema.

Los repositorios fueron publicados en la plataforma **GitHub** y abarcan las tres etapas principales del trabajo: la aplicación de control, el entorno de simulación y la implementación embebida del sistema multiagente.

1. **GalaxyRVR-Control-App:** aplicación de escritorio en Python para el control y monitoreo del rover.
2. **FreeMAES_Sim:** entorno de simulación de FreeRTOS y FreeMAES en Windows.
3. **FreeMAES for ATmega Devices:** adaptación de la biblioteca FreeMAES para microcontroladores AVR.

Cada repositorio está documentado con sus respectivas guías de instalación, uso y ejemplos funcionales.

5.1 Repositorio *GalaxyRVR Control App*

Descripción general

El repositorio **GalaxyRVR-Control-App** contiene la aplicación de escritorio desarrollada en **Python** utilizando **PyQt5**, **qasync** y **OpenCV**. Esta aplicación establece comunicación en tiempo real con el sistema multiagente embebido del rover (**FreeRTOS + FreeMAES**) mediante el protocolo **WebSockets**, permitiendo al usuario controlar, automatizar y monitorear el comportamiento del vehículo.

La aplicación ofrece tres modos de operación principales:

- **Modo Manual:** control directo de motores, servomotor y luces RGB mediante sliders, botones o teclado (WASD).
- **Modo Sistemático:** ejecución automática de secuencias de comandos almacenadas en archivos `.csv`.
- **Modo Monitor:** visualización de telemetría en tiempo real, incluyendo batería, sensores infrarrojos, distancia ultrasónica, video en vivo y consumo de memoria de los agentes.

Instalación del entorno

Para ejecutar la aplicación se requiere tener instalado **Python 3.10** o superior. El entorno puede configurarse siguiendo los pasos:

Listing 5.1: Instalación de dependencias para GalaxyRVR-Control-App

```
# Clonar el repositorio
git clone https://github.com/Oscar-FZ/GalaxyRVR-Control-App.git
cd GalaxyRVR-Control-App

# Crear y activar entorno virtual
python -m venv venv
source venv/bin/activate

# Instalar las dependencias necesarias
pip install -r requirements.txt

# Ejecutar la aplicaci n
python GalaxyRVR-Control-App.py
```

La conexión con el rover se realiza al conectarse a su red Wi-Fi (por defecto IP 192.168.4.1 y puerto 8765). El botón *"Connect / Retry"* permite restablecer la comunicación en cualquiera de los modos de operación.

Características principales

El sistema de la aplicación integra:

- Control asíncrono de eventos mediante **qasync**.
- Transmisión y recepción de datos JSON por **WebSockets**.
- Interfaz visual adaptada con **qt-material**.
- Envío periódico de comandos (100 ms) para mantener la conexión activa.
- Compatibilidad con control por teclado y ejecución de scripts automatizados.

El repositorio incluye imágenes ilustrativas de cada modo de operación y recomendaciones para garantizar una conexión Wi-Fi estable y un uso seguro del sistema.

5.2 Repositorio *FreeMAES_Sim*

Descripción general

El repositorio **FreeMAES_Sim** proporciona un **entorno de simulación en Windows** que permite ejecutar y depurar aplicaciones basadas en **FreeRTOS** y **FreeMAES** sin necesidad de hardware embebido. El entorno se desarrolló en **Microsoft Visual Studio 2022**, utilizando el **FreeRTOS Simulator para MSVC con MinGW**.

Este simulador facilita la validación de la biblioteca FreeMAES y el estudio del comportamiento de agentes en tiempo real, ejecutándose como programas de consola en el entorno Windows.

Instalación y configuración

Para crear el entorno de simulación se requiere:

- Microsoft Visual Studio 2015 o superior.
- Workload "Desktop development with C++".
- Archivos del **FreeRTOS Simulator**, disponibles en la documentación oficial de FreeRTOS.

Los pasos para configurar el proyecto son los siguientes:

1. Crear un nuevo proyecto C++ vacío en Visual Studio.
2. Copiar las carpetas `FreeRTOS_Source`, `Supporting_Functions` y el archivo `FreeRTOSConfig.h`.
3. Incluir estas carpetas en el proyecto desde el explorador de soluciones.
4. Agregar las rutas de inclusión indicadas en las propiedades del compilador.

Una vez configurado, los demos incluidos pueden compilarse y ejecutarse directamente para observar la comunicación y comportamiento entre tareas o agentes.

Estructura y demos incluidos

El repositorio mantiene una estructura de carpetas estandarizada:

- **FreeMAES_Examples:** programas de demostración de la biblioteca.
- **FreeMAES_Source:** código fuente principal de la librería.
- **FreeRTOS_Source:** sistema operativo base.
- **GalaxyRVR_Agents:** ejemplos derivados del proyecto de FreeMAES para Arduino.
- **Supporting_Functions:** funciones auxiliares para FreeRTOS.

Los demos disponibles —*Sender Receiver*, *Rock Paper Scissors* y *Telemetry*— permiten verificar el intercambio de mensajes, la planificación de tareas y el manejo del AMS en un entorno de desarrollo controlado.

5.3 Repositorio *FreeMAES for ATmega Devices*

Descripción general

El repositorio **FreeMAES for ATmega Devices** contiene la adaptación completa de la biblioteca **FreeMAES** para microcontroladores basados en **ATmega**, permitiendo el desarrollo de sistemas multiagente distribuidos en plataformas embebidas de bajos recursos. La implementación fue validada específicamente en el **rover GalaxyRVR**, utilizando la placa **Arduino MEGA 2560** con el sistema operativo **FreeRTOS**.

Instalación de la biblioteca

Para usar la librería en el entorno **Arduino IDE**, se deben seguir los pasos:

1. Instalar la biblioteca **FreeRTOS** versión 11 desde el gestor de librerías.
2. Copiar las carpetas `Supporting_Functions` y `maes-rtos` al directorio de librerías de Arduino.
3. Asegurar el uso del archivo `FreeRTOSConfig.h` incluido en el repositorio.
4. Reiniciar el IDE y agregar las librerías en el código:

```
#include <supporting_functions.h>
#include <maes-rtos.h>
```

Proyecto MAS del GalaxyRVR

El repositorio incluye una implementación completa del **Multi-Agent System (MAS)** para el GalaxyRVR. Este sistema está compuesto por cuatro agentes cooperativos, ejecutándose sobre FreeRTOS:

- **WiFi Agent:** administra la comunicación con la aplicación GalaxyRVR-Control-App.
- **Mode Handler Agent:** gestiona los modos de operación manual, sistemático y autónomo.
- **Sensor Control Agent:** lee los sensores infrarrojos y ultrasónico.
- **Peripherals Control Agent:** controla los motores, servo y luces RGB.

Cada agente es ejecutado como una tarea de FreeMAES, garantizando modularidad, paralelismo y comunicación entre procesos.

Comparación FreeMAES vs FreeRTOS

El repositorio también contiene el conjunto de programas utilizados para el **benchmark comparativo** entre FreeRTOS y FreeMAES. Las pruebas incluyen los tres demos (*Rock Paper Scissors*, *Sender Receiver*, *Telemetry*) en versiones equivalentes, tanto con tareas como con agentes, y fueron utilizados para medir tamaño de programa, uso de stack y ciclos de CPU.

Recomendaciones de hardware

Debido al consumo de memoria del sistema multiagente, se recomienda utilizar microcontroladores con al menos **6–8 kB de SRAM**. Los modelos validados incluyen el **Arduino UNO WiFi Rev2** y el **Arduino MEGA 2560**, manteniendo compatibilidad con el *GalaxyRVR Shield*. En placas con menor memoria, se aconseja reducir el número de agentes o simplificar su lógica.

Capítulo 6

Evaluación de FreeRTOS + FreeMAES

El objetivo de este capítulo es comparar el rendimiento del sistema **FreeRTOS + FreeMAES** frente a **FreeRTOS** y **Arduino OS**, evaluando el impacto del modelo multiagente en el uso de recursos, la eficiencia del CPU y la estabilidad del sistema. Para ello, se realizaron pruebas en tres aplicaciones de referencia (*Rock Paper Scissors*, *Sender Receiver* y *Telemetry*), ejecutadas en versiones equivalentes bajo FreeRTOS y FreeRTOS + FreeMAES. Estas pruebas permitieron medir el consumo de memoria Flash y SRAM, los ciclos de CPU en estado *Idle* y el uso máximo de stack. Posteriormente, se comparó el código original del **GalaxyRVR** (Arduino OS) con su versión basada en **FreeRTOS + FreeMAES**, analizando diferencias en el modelo de ejecución, estabilidad de la conexión WiFi y comportamiento de la memoria. Finalmente, se incluyen comparaciones adicionales que destacan beneficios no cuantitativos del uso de FreeMAES, como la estructura estandarizada del código, su modularidad, la comunicación predefinida entre agentes y la tolerancia a fallos, aspectos que fortalecen la arquitectura general del sistema embebido.

6.1 Aplicaciones de prueba

Rock Paper Scissors

Simula un juego de "Piedra, Papel o Tijera" entre tres tareas o agentes: *Player A*, *Player B* y *Referee*. Cada jugador elige un valor aleatorio y envía su elección al árbitro, quien compara los resultados y determina el ganador de la ronda. El flujo de ejecución permite evaluar el intercambio de mensajes y la sincronización entre múltiples tareas. En la versión con FreeMAES, la comunicación se realiza mediante el *Agent Message System*, mientras que la versión pura de FreeRTOS utiliza semáforos y colas para sincronizarse.

Sender Receiver

Corresponde a un escenario clásico de comunicación punto a punto: un agente *Sender* envía mensajes a un agente *Receiver*, que los procesa y responde con una confirmación. Esta aplicación permite medir la sobrecarga asociada al envío y recepción de mensajes. En FreeMAES, el intercambio se realiza por medio de mensajes estructurados y administrados por la plataforma de agentes; en FreeRTOS se emulan las mismas operaciones con colas de mensajes (*queues*).

Telemetry

Simula un registrador de telemetría compuesto por cuatro tareas o agentes: *Logger Current*, *Logger Voltage*, *Logger Temperature* y *Measurement*. Cada registrador solicita datos a intervalos distintos (corriente cada 500 ms, voltaje cada 1 s y temperatura cada 2 s). El agente *Measurement* atiende las solicitudes, genera los valores y los devuelve a cada registrador. Esta aplicación permite analizar el comportamiento del framework bajo condiciones periódicas de ejecución, representativas de una misión real de adquisición de datos.

6.2 Tamaño del programa (Flash Memory y SRAM)

En esta prueba se determinó el tamaño del código compilado y el uso de memoria SRAM por variables globales. Los valores se obtuvieron a partir del mapa de memoria generado por el compilador en ambos casos.

Las Tablas 6.1 a 6.3 muestran los resultados obtenidos.

Cuadro 6.1: Uso de memoria en Rock Paper Scissors

Métrica	**FreeRTOS**	**FreeRTOS + FreeMAES**	**Diferencia (%)**
SRAM (bytes)	1181	1408	+19.2 %
Flash (bytes)	12500	16948	+35.6 %

Cuadro 6.2: Uso de memoria en Sender Receiver

Métrica	**FreeRTOS**	**FreeRTOS + FreeMAES**	**Diferencia (%)**
SRAM (bytes)	1157	1363	+17.8 %
Flash (bytes)	11766	15784	+34.1 %

Cuadro 6.3: Uso de memoria en Telemetry

Métrica	**FreeRTOS**	**FreeRTOS + FreeMAES**	**Diferencia (%)**
SRAM (bytes)	1193	1448	+21.4 %
Flash (bytes)	16230	18634	+14.8 %

El incremento de memoria observado concuerda con los resultados históricos de MAES: el framework agrega estructuras de datos y funciones encapsuladas que facilitan la gestión de tareas, pero aumentan el tamaño total del binario y el espacio reservado en SRAM.

6.3 Ciclos de CPU en estado Idle

Para medir la utilización del procesador se empleó el *Idle Hook* de FreeRTOS, que se ejecuta cada vez que no hay tareas listas para correr. La cantidad de ciclos acumulados en esta función indica el porcentaje de tiempo que el procesador permaneció ocioso: un mayor número de ciclos en Idle implica menor carga CPU y, por tanto, un posible ahorro energético.

Cuadro 6.4: Ciclos de CPU en estado Idle – Rock Paper Scissors

Métrica	**FreeRTOS**	**FreeRTOS + FreeMAES**	**Diferencia (%)**
Ciclos Idle	1,053,356	1,112,109	+5.6 %

Cuadro 6.5: Ciclos de CPU en estado Idle – Sender Receiver

Métrica	**FreeRTOS**	**FreeRTOS + FreeMAES**	**Diferencia (%)**
Ciclos Idle	524,190	524,207	+0.003 %

Cuadro 6.6: Ciclos de CPU en estado Idle – Telemetry

Métrica	**FreeRTOS**	**FreeRTOS + FreeMAES**	**Diferencia (%)**
Ciclos Idle	76,572	98,053	+28.0 %

6.4 Consumo máximo de Stack

El consumo máximo de pila se midió utilizando la función `uxTaskGetStackHighWaterMark()` de FreeRTOS. Restando el valor retornado del total asignado (256 bytes), se obtiene el máximo de stack utilizado por cada tarea o agente.

Cuadro 6.7: Consumo máximo de stack – Rock Paper Scissors

Agente / Tarea	**FreeRTOS**	**FreeRTOS + FreeMAES**	**Diferencia (%)**
Player A	92	128	+39.1 %
Player B	92	128	+39.1 %
Referee	106	159	+50.0 %

Cuadro 6.8: Consumo máximo de stack – Sender Receiver

Agente / Tarea	**FreeRTOS**	**FreeRTOS + FreeMAES**	**Diferencia (%)**
Sender	127	128	+0.8 %
Receiver	127	133	+4.7 %

Cuadro 6.9: Consumo máximo de stack – Telemetry

Agente / Tarea	FreeRTOS	FreeRTOS + FreeMAES	Diferencia (%)
Logger Current	197	165	-16.2 %
Logger Voltage	197	165	-16.2 %
Logger Temperature	197	165	-16.2 %
Measurement	192	165	-14.1 %

6.5 GalaxyRVR original vs GalaxyRVR con agentes

A diferencia de los ejemplos anteriores, la comparación entre el código original del GalaxyRVR y su versión con FreeRTOS + FreeMAES no puede realizarse de manera estrictamente uno a uno. En los tres programas previos (Rock Paper Scissors, Sender Receiver y Telemetry), ambas versiones se ejecutaban bajo un mismo sistema operativo —FreeRTOS—, lo que permitía mantener estructuras de código equivalentes y utilizar las mismas herramientas de medición. En cambio, el código original del GalaxyRVR fue desarrollado sobre Arduino OS, mientras que la versión mejorada se implementó sobre FreeRTOS + FreeMAES, lo que introduce diferencias fundamentales en la arquitectura, el comportamiento del procesador y la forma en que se gestionan las tareas.

Tamaño del programa

Se aprecia un incremento en el tamaño del binario y en el uso de memoria global al emplear FreeMAES. Esto se debe a las estructuras adicionales requeridas por el Agent Management System (AMS), las colas de comunicación, y las funciones propias del kernel de FreeRTOS. Aun así, el crecimiento se mantiene dentro de márgenes aceptables considerando los beneficios funcionales que introduce la arquitectura multiagente.

Cuadro 6.10: Uso de memoria – GalaxyRVR

Métrica	Arduino OS	FreeRTOS + FreeMAES	Diferencia (%)
SRAM (bytes)	1005	1561	+55.3 %
Flash (bytes)	15264	27416	+79.6 %

Ciclos de CPU en estado Idle

En este caso no se puede realizar una comparación cuantitativa directa. El modelo de ejecución de **Arduino OS** no contempla estados de inactividad (*Idle*), ya que el microcontrolador ejecuta continuamente la función `loop()`, sin pausas ni suspensión del CPU. Incluso al usar funciones como `delay()`, el procesador continúa activo, manteniendo una ocupación del 100 %.

Por el contrario, **FreeRTOS + FreeMAES** sí dispone de un *Idle Hook*, ejecutado cuando no existen tareas listas para correr. Esto permite que el CPU entre en modos de bajo consumo entre periodos de actividad, reduciendo la carga total y mejorando la eficiencia energética del sistema.

Aunque no existan valores comparables en esta categoría, se puede afirmar que **FreeRTOS + FreeMAES** aprovecha de forma más efectiva los tiempos ociosos del procesador.

Consumo máximo de Stack

En **Arduino OS**, el consumo de pila puede medirse de forma global, ya que el sistema se ejecuta dentro de un único contexto. En **FreeRTOS + FreeMAES**, en cambio, cada tarea o agente posee su propia pila asignada de manera independiente, lo que impide obtener un único valor total de *stack* utilizado durante la ejecución.

Aunque FreeRTOS permite medir el *HighWaterMark* por tarea, existen factores no cuantificables —como la reserva del *Agent Management System (AMS)* o la pila usada por la librería *SunFounder AI Camera* durante las comunicaciones WiFi— que dificultan una comparación directa.

La Tabla 6.11 muestra los valores globales observados al inicio del programa antes del arranque del *scheduler*.

Cuadro 6.11: Uso total de RAM – GalaxyRVR

Métrica	**Arduino OS**	**FreeRTOS + FreeMAES**	**Diferencia (%)**
RAM total (bytes)	1622	4079	+151.5 %

El aumento en el uso de memoria es consecuencia del modelo multitarea, que asigna espacios de pila separados para cada agente y para las funciones internas del sistema operativo. Esto incrementa la memoria total usada, pero a la vez mejora la seguridad y estabilidad al aislar los procesos entre sí.

Single Thread vs Multi-threading

Una diferencia esencial entre ambas versiones del sistema radica en su modelo de ejecución. Las placas basadas en microcontroladores **AVR** cuentan con un CPU de un solo núcleo, y bajo **Arduino OS** el programa se ejecuta de forma completamente secuencial y monohilo (*single-thread*). Esto significa que todas las tareas del rover —conexión WiFi, control de motores, manejo de sensores, servo, luces y algoritmos de control autónomo— se ejecutan una tras otra dentro del bucle `loop()`.

El problema de este enfoque es que el tiempo entre iteraciones puede llegar a ser demasiado largo. Por ejemplo, en el GalaxyRVR original, la conexión WiFi con la aplicación era altamente inestable: en entornos con canales WiFi libres, la conexión podía durar apenas un minuto antes de caer, mientras que en entornos saturados la conexión rara vez se mantenía estable por más de unos pocos segundos. Esto ocurre porque la función encargada de mantener la conexión WiFi no se vuelve a ejecutar hasta que el bucle principal termina de procesar todas las demás tareas del rover, lo que genera intervalos prolongados entre reconexiones.

En cambio, con **FreeRTOS + FreeMAES**, cada agente corre en su propio hilo de ejecución (*thread*) gestionado por el *scheduler* del sistema operativo. Esto permite que las tareas críticas, como el mantenimiento de la conexión WiFi, se ejecuten con una periodicidad mucho mayor y sin interrupciones prolongadas, mejorando drásticamente la estabilidad de la comunicación. En pruebas realizadas en entornos con canales WiFi libres, la conexión permaneció totalmente estable y en entornos saturados, la conexión se mantuvo estable durante la mayor parte del tiempo, presentando únicamente desconexiones ocasionales y breves.

Este comportamiento evidencia una de las ventajas más importantes del modelo multiagente sobre el entorno monohilo: la capacidad de respuesta y la estabilidad del sistema mejoran significativamente.

6.6 Comparaciones Adicionales

Además de los resultados cuantitativos obtenidos en las pruebas de rendimiento, la implementación del sistema con FreeMAES presenta una serie de ventajas cualitativas que refuerzan su idoneidad para el desarrollo de sistemas embebidos multi-agente. Estas características no se reflejan directamente en métricas de memoria o CPU, pero tienen un impacto significativo en la mantenibilidad, modularidad y robustez del código.

Código estructurado

Una de las principales ventajas de **FreeMAES** es que establece una estructura de código uniforme para todos los proyectos desarrollados con la biblioteca. El flujo general de un programa con FreeMAES se compone de cinco secciones principales: *Inclusiones y Definiciones*, *Instanciación de Condiciones, Agentes y Plataforma*, *Derivación y Encapsulado de Comportamientos*, *Inicialización de Tareas y Arranque de Plataforma*, y *Arranque del Calendarizador*.

Esta organización se ilustra en la Figura 6.1, la cual representa la estructura estándar que debe seguir todo sistema implementado con FreeMAES.

En particular, la sección de *Derivación y Encapsulado de Comportamientos* constituye el núcleo funcional de cada agente. Aquí se definen los agentes, se les asigna un tipo de comportamiento —cíclico o *One-Shot*—, y se implementan dos bloques fundamentales:

- `setup()`, que contiene el código de inicialización que se ejecuta una sola vez cuando el agente arranca.
- `action()`, que contiene el código que se ejecuta de manera repetitiva (en agentes cíclicos) o una única vez (en agentes *One-Shot*).

Esta estructura clara y estandarizada facilita la comprensión del sistema y permite extenderlo de manera ordenada, reduciendo errores en el mantenimiento y la depuración.

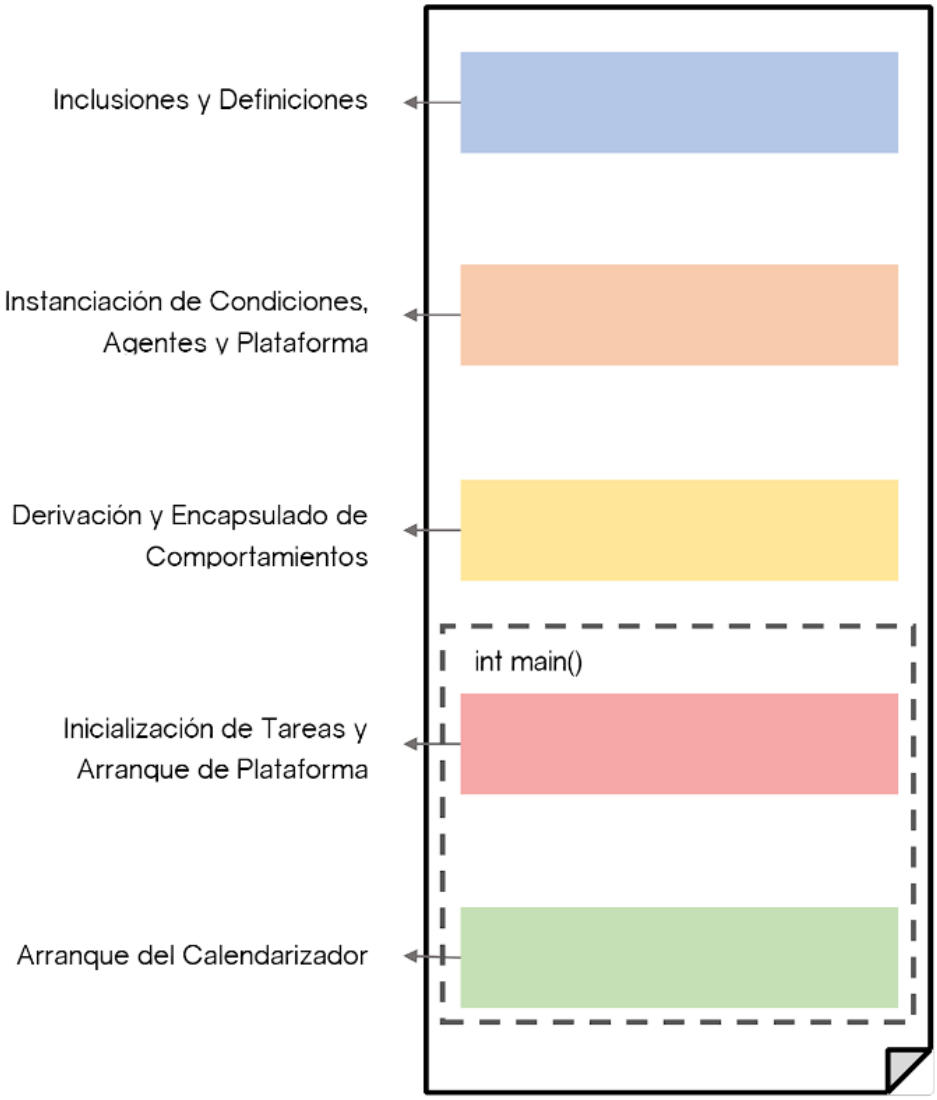

Figura 6.1: Estructura general del código en FreeMAES. obtenido de [2]

Modularidad

La organización de **FreeMAES** promueve una arquitectura altamente modular, donde cada agente puede implementarse en archivos separados, con independencia funcional del resto. En el contexto de este proyecto, cada agente cuenta con dos archivos principales:

- Un archivo **header** (`.h`), que incluye las declaraciones, definiciones y condiciones de inicialización del agente.
- Un archivo **fuente** (`.cpp`), que contiene la sección de *Derivación y Encapsulado de Comportamientos* correspondiente.

De esta manera, el programador puede agregar nuevos agentes sin necesidad de modificar el código ya existente. Basta con incluir los nuevos archivos y realizar su inicialización en la función `main()`.

Este enfoque reduce la complejidad del sistema y evita dependencias innecesarias entre módulos, lo cual representa una mejora significativa respecto al modelo monolítico de **Arduino OS**.

Listing 6.1: Archivo `infrared_agent.h`. Declaraciones e inclusiones del agente de sensores infrarrojos.

```
#ifndef INFRARED_AGENT_H
#define INFRARED_AGENT_H

#include <Arduino.h>
#include <Arduino_FreeRTOS.h>
#include <task.h>
#include <queue.h>
#include <supporting_functions.h>
#include <maes-rtos.h>

using namespace MAES;

#define IR_LEFT_PIN 8
#define IR_RIGHT_PIN 7

extern byte ir_result;
extern Agent Ir_Control;

void ir(void* pvParameters);
#endif
```

Listing 6.2: Archivo `infrared_agent.cpp`. Implementación del comportamiento del agente de sensores infrarrojos.

```
#include "infrared_agent.h"

Agent Ir_Control("Infrared Control Agent", 1, 192);

byte ir_result = 0b00000000;

class irBehaviour : public CyclicBehaviour {
public:
  void setup() override {
    pinMode(IR_LEFT_PIN, INPUT);
    pinMode(IR_RIGHT_PIN, INPUT);
  }

  void action() override {
    byte left = digitalRead(IR_LEFT_PIN);
    byte right = digitalRead(IR_RIGHT_PIN);
    ir_result = (left << 1) | right;
  }
};

void ir(void* pvParameters) {
  irBehaviour b;
  b.execute();
}
```

Canales de comunicación predefinidos

Otra ventaja sustancial de **FreeMAES** es la simplificación de la comunicación entre tareas. En **FreeRTOS**, el programador debe definir manualmente las colas de mensajes, su tamaño, el tipo de datos que transportan y las tareas que pueden acceder a cada canal. Esto genera una sobrecarga de configuración y una mayor propensión a errores en proyectos grandes.

FreeMAES elimina esta necesidad al contar con canales de comunicación predefinidos gestionados por la propia plataforma. El programador únicamente debe indicar qué agentes se comunicarán entre sí; el sistema se encarga automáticamente de establecer los enlaces y formatear los mensajes mediante su propio modelo de datos.

Este mecanismo reduce drásticamente la complejidad del diseño y garantiza coherencia en los intercambios de información entre agentes.

Autonomía y tolerancia a fallos

Cada agente en **FreeMAES** se ejecuta dentro de su propio hilo, gestionado por el kernel de **FreeRTOS**. Esto significa que si un agente falla en tiempo de ejecución —por ejemplo, el agente encargado de los sensores infrarrojos—, el error no afecta al funcionamiento del resto del sistema. Los demás agentes continúan ejecutándose normalmente, manteniendo la estabilidad global del sistema.

En contraste, bajo el entorno de **Arduino OS**, todo el código se ejecuta de forma secuencial dentro del bucle principal `loop()`. Un fallo en una función puede detener la ejecución completa del programa, afectando a todos los componentes del sistema.

La separación de agentes en hilos independientes dentro de FreeMAES ofrece, por tanto, una mayor robustez, aislamiento de fallos y capacidad de recuperación, características fundamentales para sistemas embebidos de misión crítica.

6.7 Reflexiones finales

Los resultados obtenidos evidencian que la incorporación del enfoque multiagente introduce una sobrecarga controlada de recursos, pero a cambio ofrece beneficios sustanciales en organización, robustez y eficiencia del sistema, especialmente en aplicaciones embebidas complejas como el rover **GalaxyRVR**.

Aplicaciones de prueba

Los experimentos con los tres programas de referencia —*Rock Paper Scissors*, *Sender Receiver* y *Telemetry*— muestran una tendencia clara: la versión con **FreeRTOS + FreeMAES** presenta un aumento moderado en el uso de memoria, pero mantiene un rendimiento estable e incluso superior en las métricas de eficiencia del procesador.

En términos de memoria Flash y SRAM, se observa un incremento al incorporar FreeMAES. Este aumento responde principalmente a la creación de estructuras adicionales necesarias para la gestión de agentes, las colas de comunicación y el *Agent Management System (AMS)*. Sin embargo, dicho incremento se considera aceptable frente a las ventajas obtenidas en modularidad y escalabilidad del sistema.

El análisis de los ciclos de CPU en estado *Idle* evidencia que FreeMAES mejora la eficiencia del tiempo ocioso, especialmente en aplicaciones con múltiples tareas concurrentes. El *scheduler* de FreeRTOS, combinado con la planificación distribuida entre agentes, permite un mejor aprovechamiento del procesador al evitar bloqueos prolongados. En el caso del programa *Telemetry*, por ejemplo, el sistema con FreeMAES alcanzó un 28 % más de tiempo en *Idle*

que su equivalente con FreeRTOS puro, indicando una reducción efectiva del uso activo del CPU y, por consiguiente, del consumo energético.

Por último, la medición del consumo máximo de *stack* confirmó que FreeMAES distribuye la carga de memoria de manera más equilibrada. Si bien el consumo total tiende a aumentar ligeramente, la asignación independiente de pila por agente previene desbordamientos y mejora la previsibilidad del sistema, algo esencial en entornos de misión crítica.

Programas del GalaxyRVR

En el caso del **GalaxyRVR**, las diferencias entre las implementaciones son aún más notables. El código original basado en **Arduino OS** opera bajo un esquema monohilo (*single-thread*) y secuencial, mientras que la versión desarrollada con **FreeRTOS + FreeMAES** implementa una arquitectura multitarea y multiagente, donde cada componente del sistema se ejecuta de manera concurrente.

En el uso de memoria, los resultados muestran un incremento del orden del 55 % al 80 % respecto a la versión original. Esta diferencia proviene de las estructuras adicionales requeridas por FreeRTOS y FreeMAES, así como de la creación de pilas individuales para cada agente. No obstante, la mayor parte de este consumo adicional se traduce en beneficios tangibles, como la capacidad de aislar procesos, prevenir fallos en cascada y mantener la estabilidad global del sistema.

La métrica de estado *Idle* no puede compararse directamente, dado que Arduino OS no contempla modos de suspensión o inactividad. Mientras el CPU de la versión original se mantiene al 100 % de ocupación durante todo el ciclo de ejecución, el sistema con FreeRTOS + FreeMAES aprovecha los periodos de inactividad para ejecutar la *Idle Task*, reduciendo el consumo de energía y permitiendo una operación más eficiente.

En cuanto al consumo de *stack*, el sistema multiagente presenta un uso total de RAM más alto (4079 bytes frente a 1622 bytes), pero este valor refleja el costo de ejecutar múltiples hilos de manera independiente. A diferencia del modelo monolítico de Arduino, cada agente cuenta con su propio espacio de pila, lo que mejora la seguridad y la gestión de recursos en ejecución.

Una diferencia particularmente relevante es la relacionada con el modelo de ejecución *Single Thread vs Multithreading*. El uso de un *scheduler* en FreeRTOS permite ejecutar tareas en "paralelo" dentro de un CPU de un solo núcleo, garantizando tiempos de respuesta más cortos y un comportamiento más estable. Este efecto se evidenció de forma clara en la estabilidad de la conexión WiFi del GalaxyRVR: mientras la versión en Arduino OS perdía la conexión en cuestión de segundos en entornos saturados, la versión con FreeMAES mantuvo una comunicación estable y continua, incluso bajo condiciones de congestión.

La ejecución concurrente del agente de conexión WiFi junto con el resto de agentes eliminó los largos intervalos entre llamadas que, en el sistema monohilo, provocaban la desconexión constante. Este resultado confirma que el modelo multitarea no solo mejora la estructura interna del código, sino también el rendimiento operativo y la confiabilidad en sistemas con múltiples procesos dependientes.

Capítulo 7

Conclusiones y Trabajo Futuro

7.1 Conclusiones

El presente trabajo integró de forma exitosa la biblioteca *FreeMAES* con el sistema operativo en tiempo real *FreeRTOS*, validando su funcionamiento en la plataforma *GalaxyRVR* mediante la creación de un sistema multiagente embebido completamente funcional. Esta implementación permitió comprobar las ventajas del paradigma multiagente aplicado a sistemas de control distribuidos y de recursos limitados, consolidando un entorno experimental útil para el desarrollo de futuras misiones y proyectos dentro del laboratorio SETEC.

En primer lugar, el diseño de la aplicación *GalaxyRVR-Control-App* permitió desarrollar una interfaz moderna, multiplataforma y estable que facilita la interacción entre el usuario y el sistema embebido. Su arquitectura modular, basada en PyQt5, WebSockets, qasync y OpenCV, integró de forma eficiente el control manual, la ejecución sistemática de secuencias y la supervisión en tiempo real mediante video y telemetría. Gracias a esta herramienta fue posible validar experimentalmente las funciones de cada agente, realizar pruebas controladas del rover y monitorear el comportamiento del sistema de manera visual y estructurada.

En cuanto al diseño del sistema multiagente, se logró adaptar exitosamente la biblioteca *FreeMAES* a la familia de microcontroladores ATmega, ampliando su aplicabilidad a entornos de bajo costo y recursos limitados. Sin embargo, durante el proceso se evidenció que la memoria SRAM es el recurso más crítico para la implementación de agentes. En el Arduino UNO R3, con solo 2 kB de SRAM, no fue posible implementar más de tres agentes, asignando a cada uno un tamaño de pila de 192 bytes. En el Arduino MEGA 2560, a pesar de contar con 8 kB de SRAM, el sistema no pudo sostener más de siete agentes bajo las mismas condiciones de memoria por tarea. Estas restricciones obligaron a combinar múltiples responsabilidades dentro de algunos agentes, de modo que el sistema multiagente resultante no puede considerarse un sistema completamente puro, pues varios agentes debieron abarcar más de una función para ajustarse a las limitaciones del hardware. Aun así, siguiendo la metodología propuesta por CARVAJAL-GODÍNEZ [1], se obtuvo un sistema modular, estable y funcional, demostrando la viabilidad de emplear arquitecturas multiagente en plataformas embebidas con recursos severamente limitados.

En lo referente a la documentación, se elaboraron tres repositorios de GitHub que complementan el proyecto: uno para la aplicación GalaxyRVR-Control-App, otro para la creación de un entorno de simulación en Visual Studio 2022, y un tercero que contiene la adaptación de la biblioteca FreeMAES junto con el código del sistema multiagente. Esta estructura permite la replicación completa del entorno de trabajo y estandariza las prácticas para futuros desarrollos dentro del laboratorio.

Finalmente, la evaluación comparativa entre FreeRTOS y FreeRTOS + FreeMAES evidenció un incremento en

el consumo de memoria FLASH y SRAM debido a las estructuras adicionales de agentes y al AMS. A pesar de ello, se observó una mejora significativa en la estabilidad, la modularidad y la respuesta del sistema, además de una reducción en el uso promedio del CPU. FreeMAES gestiona eficientemente la activación y suspensión de tareas, lo que se traduce en un sistema más ordenado, predecible y escalable. En conjunto, los resultados confirman que FreeMAES cumple su propósito fundamental: facilitar el diseño, organización y mantenimiento de sistemas multiagentes embebidos mediante un mayor nivel de abstracción.

7.2 Trabajo Futuro

La aplicación *GalaxyRVR-Control-App* puede ampliarse mediante la inclusión de nuevas funciones de diagnóstico y depuración. Entre ellas, destaca la integración de comunicación serial bidireccional, que permitiría realizar pruebas en ambientes aún más controlados y sin depender exclusivamente de la interfaz WiFi. Esta funcionalidad abriría paso a entornos de validación más precisos y a modos de operación alternativos para el rover.

Un siguiente paso importante es la **optimización de la biblioteca FreeMAES**. Aunque FreeRTOS cuenta con versiones altamente optimizadas para microcontroladores con recursos limitados, FreeMAES no está adaptada específicamente para dispositivos con poca memoria FLASH y SRAM, como los ATmega. Una versión optimizada de FreeMAES permitiría mejorar el rendimiento, reducir el uso de memoria y habilitar la ejecución de agentes más complejos en dispositivos de bajo costo.

También es necesario **extender la compatibilidad de FreeMAES a nuevos dispositivos**. Investigaciones previas [8] validan su funcionamiento en arquitecturas AVR y AVR32, pero estas pruebas se realizaron únicamente en entornos de compilación. Sería altamente provechoso ejecutar la biblioteca en hardware AVR32 real, ya que estos microcontroladores cuentan con mayor memoria y permitirían implementar sistemas multiagentes más avanzados.

Una línea importante de trabajo futuro consiste en probar la biblioteca FreeMAES en dispositivos con mayor cantidad de SRAM. Esto permitiría implementar sistemas multiagente más complejos y estudiar con mayor profundidad cómo escala el desempeño del framework respecto a la carga computacional, consumo energético y estabilidad del sistema en aplicaciones de mayor complejidad.

Para aumentar la replicabilidad del proyecto, se propone desarrollar una guía detallada sobre cómo montar el entorno de simulación en sistemas Linux. Esto permitiría que más usuarios, especialmente en entornos académicos, repliquen el simulador sin depender exclusivamente de Windows.

Bibliografía

[1] J. Carvajal-Godínez, "Agent-based architectures supporting fault-tolerance in small satellites," Ph.D. dissertation, 02 2021.

[2] D. A. R. Marín. (2021) Freemaes. [Online]. Available: https://github.com/DRoMarin/FreeMAES.git

[3] "Mars 2020: Perseverance Rover," https://science.nasa.gov/mission/mars-2020-perseverance/, accessed: 25-08-2025.

[4] European Space Agency, "Exomars rover: Rosalind franklin mission overview," in *ESA Publications*, 2020. [Online]. Available: https://www.esa.int/Science_Exploration/Human_and_Robotic_Exploration/Exploration/ExoMars

[5] M. GOMEZ-JENKINS, J. CALVO-ALVARADO, A. CHAVES-JIMÉNEZ, J. CARVAJAL-GODÍNEZ, E. MARTINEZ, Y. ARIAS, A. J. CALVO-OBANDO, V. JIMENEZ, J. J. ROJAS, A. VALVERDE-SALAZAR, and J. RAMIREZ-MOLINA, "Project irazú: Space and ground systems engineering of a 1u cubesat store and forward mission for environmental monitoring," *TRANSACTIONS OF THE JAPAN SOCIETY FOR AERONAUTICAL AND SPACE SCIENCES*, vol. 66, no. 6, pp. 217–225, 2023, accessed: 15-04-2025.

[6] A. CHAVES-JIMÉNEZ, J. CARVAJAL-GODÍNEZ, A. J. CALVO-OBANDO, V. JIMENEZ, and A. COTO-CORTEZ, "Gwsat: Prototipo de monitoreo de humedales a través de un sistema espacial tipo "store forward"," https://repositoriotec.tec.ac.cr/bitstream/handle/2238/14980/PI72_BIB312181_GWSat_Prototipo_de_monitoreo....pdf?sequence=1, 2023, accessed: 15-04-2025.

[7] C. Chan-Zheng, "Maes: A multi-agent systems framework for embedded systems," Ph.D. dissertation, 09 2017, accessed: 17-11-2025.

[8] D.-A. Rojas-Marín, "Freemaes: Desarrollo de una biblioteca bajo el paradigma multiagente para sistemas embebidos (maes) compatible con la nanomind a3200 y el kernel freertos," https://repositoriotec.tec.ac.cr/handle/2238/13300?show=full, 2021, accessed: 14-08-2025.

[9] "Perseverance extrae por primera vez una muestra del núcleo de una roca marciana," https://ciencia.nasa.gov/sistema-solar/perseverance-extrae-por-primera-vez-el-nucleo-de-una-roca-marciana/, accessed: 15-04-2025.

[10] "El rover Perseverance de la NASA "al volante"," https://ciencia.nasa.gov/sistema-solar/el-mars-rover-perseverance-de-la-nasa-al-volante/, accessed: 15-04-2025.

[11] "Mars Exploration Rover – Spirit," https://science.nasa.gov/mission/mer-spirit/, accessed: 25-08-2025.

[12] "Mars Exploration Rover Opportunity (MER-B)," https://science.nasa.gov/mission/mer-opportunity/, accessed: 25-08-2025.

[13] "The Planetary Society: Cost of the Mars Exploration Rovers," https://www.planetary.org/space-policy/cost-of-the-mars-exploration-rovers?utm_source=chatgpt.com, accessed: 25-08-2025.

[14] P. H. Feiler, B. Lewis, and S. Vestal, "Improving predictability in embedded real-time systems," Software Engineering Institute, Carnegie Mellon University, Technical Report CMU/SEI-2000-SR-011, Dec. 2000. [Online]. Available: https://www.sei.cmu.edu/library/improving-predictability-in-embedded-real-time-systems/

[15] "ArduinoDocs what is arduino?" https://docs.arduino.cc/learn/starting-guide/whats-arduino/, accessed: 17-11-2025.

[16] "SunFounder GalaxyRVR Mars Rover Kit for Arduino Uno R3," https://www.sunfounder.com/products/sunfounder-galaxyrvr-mars-rover-kit?_pos=1&_sid=9e0998d7d&_ss=r, accessed: 14-08-2025.

[17] Foundation for Intelligent Physical Agents (FIPA), "Fipa agent management specification," FIPA TC Agent Management, Standard SC00023K, Aug. 2004, supersedes FIPA00002, FIPA00017, FIPA00019.

[18] FreeRTOS. (2023) Rtos fundamentals. Accessed: 17-11-2025. [Online]. Available: https://rcc.freertos.org/Documentation/01-FreeRTOS-quick-start/01-Beginners-guide/01-RTOS-fundamentals

[19] "FreeRTOS what is freertos?" https://www.freertos.org/Why-FreeRTOS/What-is-FreeRTOS, accessed: 14-08-2025.

[20] "Qt about qt," https://wiki.qt.io/About_Qt, accessed: 17-11-2025.

[21] "WebSocket.org the websocket protocol," https://websocket.org/guides/websocket-protocol/, accessed: 17-11-2025.

[22] N. T. Chan and X. He, "A review of control techniques for lunar rovers," in *Proceedings of the 2024 2nd International Conference on Frontiers of Intelligent Manufacturing and Automation*, ser. CFIMA '24. New York, NY, USA: Association for Computing Machinery, 2025, p. 643–649. [Online]. Available: https://doi.org/10.1145/3704558.3704563

[23] B. J. Hockman, A. Frick, R. G. Reid, I. A. Nesnas, and M. Pavone, "Design, control, and experimentation of internally-actuated rovers for the exploration of low-gravity planetary bodies," *Journal of Field Robotics*, vol. 34, no. 1, pp. 5–24, 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21656

[24] C. Barnier, O.-E.-K. Aktouf, A. Mercier, and J.-P. Jamont, "Toward an embedded multi-agent system methodology and positioning on testing," 10 2017, pp. 239–244.

[25] M. Khalgui, O. Mosbahi, and H.-M. Hanisch, "Reconfiguration protocol for multi-agent embedded control systems," *IFAC Proceedings Volumes*, vol. 42, no. 4, pp. 960–965, 2009, 13th IFAC Symposium on Information Control Problems in Manufacturing. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1474667016339180

[26] Y. Jin, N. Lin, and W. Zhang, "Gpmappo: Collaborative sar optimization of human-uav in post-disaster scenarios," in *2024 IEEE International Conference on Embedded Software and Systems (ICESS)*, 2024, pp. 17–24.

[27] "VXWORKS Product Overview," https://www.windriver.com/resource/vxworks-product-overview, accessed: 14-09-2025.

[28] R. Kandari, K. L. Kiran, and B. Vaidya DhavalKumar, “Design of scalable software architecture for spotlight sar imaging on uav platform,” in *2024 IEEE India Geoscience and Remote Sensing Symposium (InGARSS)*, 2024, pp. 1–4.

[29] Óscar Fernández Zúñiga. (2025) Freemaes. [Online]. Available: https://github.com/Oscar-FZ/FreeMAES-for-ATmega-Devices.git

[30] ——. (2025) Galaxyrvr-control-app. [Online]. Available: https://github.com/Oscar-FZ/GalaxyRVR-Control-App.git

[31] ——. (2025) Freemaes. [Online]. Available: https://github.com/Oscar-FZ/FreeMAES.git

Apéndice A

Repositorios Desarrollados Durante el Proyecto

A continuación se listan todos los repositorios desarrollados y utilizados durante el Trabajo Final de Graduación. Cada repositorio contiene módulos fundamentales del ecosistema GalaxyRVR + FreeMAES y procesos de simulación, control y documentación.

Aplicación de Control

- **GalaxyRVR-Control-App** Repositorio de la aplicación desarrollada en Python (PyQt5), utilizada para controlar el GalaxyRVR en sus diferentes modos de operación. `https://github.com/Oscar-FZ/GalaxyRVR-Control-App` [30]

Simulación y Pruebas

- **FreeMAES_Sim** Entorno de simulación basado en Visual Studio 2022 para pruebas de FreeMAES en Windows, utilizando ejemplos y demos diseñados por Daniel Andrés Rojas Marín. `https://github.com/Oscar-FZ/FreeMAES.git` [31]

Implementación en Microcontroladores ATmega y Proyecto General

- **FreeMAES-for-ATmega-Devices** Adaptación completa de la biblioteca FreeMAES para microcontroladores ATmega con FreeRTOS, incluyendo agentes para el GalaxyRVR real ademas de la arquitectura multiagente diseñada durante el proyecto y documentación asociada.. `https://github.com/Oscar-FZ/FreeMAES-for-ATmega-Devices` [29]