Tecnológico de Costa Rica

Escuela de Ingeniería Electrónica

Programa de Licenciatura en Ingeniería Electrónica

FreeMAES: Desarrollo de una biblioteca bajo el Paradigma Multiagente para Sistemas Embebidos (MAES) compatible con la NanoMind A3200 y el kernel FreeRTOS

Informe de Trabajo Final de Graduación para optar por el título de

Ingeniero en Electrónica con el grado académico de Licenciatura

Daniel Andrés Rojas Marín

Cartago, 28 de abril de 2021

Declaro que el presente documento de tesis ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos y resultados experimentales propios.

En los casos en que he utilizado bibliografía he procedido a indicar las fuentes mediante las respectivas citas bibliográficas. En consecuencia, asumo la responsabilidad total por el trabajo de tesis realizado y por el contenido del presente documento.

Daniel Andrés Rojas Marín

Cartago, 28 de abril de 2021

Céd: 1-1686-0690

Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
Proyecto de Graduación
Acta de Aprobación

Defensa de Proyecto de Graduación
Requisito para optar por el título de Ingeniero en Electrónica
Grado Académico de Licenciatura

El Tribunal Evaluador aprueba la defensa del proyecto de graduación denominado *FreeMAES: Desarrollo de una biblioteca bajo el Paradigma Multiagente para Sistemas Embebidos (MAES) compatible con la NanoMind A3200 y el kernel FreeRTOS*, realizado por el señor Daniel Andrés Rojas Marín 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

LAURA CRISTINA CABRERA QUIROS (FIRMA)
Digitally signed by LAURA CRISTINA CABRERA QUIROS (FIRMA)
Date: 2021.04.29 10:00:42 -06'00'

Dra. Laura Cristina Cabrera Quirós
Profesora Lectora

WILLIAM FAUSTO QUIROS SOLANO (FIRMA)
Digitally signed by WILLIAM FAUSTO QUIROS SOLANO (FIRMA)
Date: 2021.04.29 09:59:28 -06'00'

Dr. William Quirós Solano
Profesor Lector

Firmado digitalmente por JOHAN CARVAJAL GODINEZ (FIRMA)
Fecha: 2021.04.28 19:14:01 -06'00'

Dr. Johan Carvajal Godínez
Profesor Asesor

Cartago, 28 de abril de 2021

Resumen

En este documento se desarrollan las diferentes etapas para la implementación de un paradigma de software nombrado Multiagente para Sistemas Embebidos (MAES por sus siglas en inglés) compatible con el microkernel FreeRTOS y la computadora a bordo NanoMind A3200 desplegada en CubeSats, cuyo microcontrolador (MCU) está basado en la arquitectura AVR32 de la compañia Atmel (adquirida por Microchip Technology en 2016).

Se exponen las clases del modelo MAES y su relación con el modelo propuesto por la Fundación para Agentes Físicos Inteligentes (FIPA), además se describen las estructuras de datos, cambios a funciones equivalentes y adiciones necesarias para transportar una implementación anterior realizada para TI-RTOS al sistema operativo objetivo. Seguidamente se evalúan las clases y métodos de la API mediante pruebas unitarias y casos de estudio, de acuerdo a la metodología Sistemas Multiagentes para Aplicaciones Espaciales (MASSA), mediante una implementación para Microsoft Visual C++. Por último, se examina el consumo de memoria de la implementación para las arquitecturas AVR8 y AVR32, considerando un MCU y una aplicación única para cada arquitectura.

Se determina que el incremento de memoria Flash causado por la implementación de las aplicaciones mediante el paradigma corresponde aproximadamente a 4.6 kB (9.0 % de 48 kB) para AVR8 y 5.2 kB (1.0 % de 512 kB) para AVR32. Adicionalmente, se propone una arquitectura multiagente que administre el protocolo de comunicación CubeSat (CSP).

Palabras clave: AVR32, NanoMind A3200, paradigma de software, sistemas multiagente (SMA), Sistema Operativo de Tiempo Real (RTOS), FreeRTOS

Abstract

This document develops the different stages for porting a software framework named Multi-Agent for Embedded Systems (MAES) to be compatible with the microkernel FreeRTOS and the on-board computer NanoMind A3200 deployed on CubeSats, whose microcontroller (MCU) is based on Atmel's AVR32 architecture (then acquired by Microchip Technology on 2016).

The MAES model's classes and their relationship with the model proposed by the Foundation for Intelligent Physical Agents (FIPA) are displayed. Data structures, function changes and needed additions to translate a previous implementation developed from TI-RTOS to the target OS are further described. Subsequently the classes and API methods are evaluated via unit testing and case studies, according to the Multi-Agent System for Space Applications methodology (MASSA), through a Microsoft Visual C++ port. Lastly, the port's memory usage is examined for AVR8 and AVR32 architectures, considering an MCU and a unique application for each architecture.

It is determined that an increase of Flash memory usage caused by the deployment of the applications using the framework approximately corresponds to 4.6 kB (9.0 % of 48 kB) for AVR8 y 5.2 kB (1.0 % of 512 kB) for AVR32. Additionally, an multi-agent architecture capable of managing the CubeSat Space Protocol (CSP) is proposed.

Keywords: AVR32, CubeSat, NanoMind A3200, software framework, Multi-Agent Systems (MAS), Real Time Operating System (RTOS), FreeRTOS

Para Esmeralda y Luis

Agradecimientos

Agradezco todas las ayudas proporcionadas por mi familia, principalmente a mi madre Esmeralda Marín Marín y mi padre Luis Rojas Murillo por haberme apoyado incondicionalmente durante mi tiempo de estudio que, a pesar de las adversidades y problemas que fueron surgiendo, siempre confiaron en mí y nunca limitaron mis oportunidades de crecimiento. Adicionalmente agradezco a mi tía, tíos, primos y primas; particularmente a Rafael Marín Marín quien siempre asistió mi educación desde muy joven.

Extiendo las gracias a los profesores, tutores e investigadores de los diferentes departamentos del Instituto Tecnológico de Costa Rica que continuamente incrementaron mi interés y aprecio por la ingeniería e investigación. Gracias al tribunal evaluador; al profesor asesor Johan Carvajal Godínez por el acompañamiento en este proyecto y todos los estudiantes con quien compartí el placer de aprender, en especial a mis amigos más íntimos.

Por último un agradecimiento especial a Olman Quiros Jiménez por disponer de su tiempo y experiencia para atender las consultas que tuviese sobre el equipo electrónico, sus requerimientos y herramientas de software.

Daniel Andrés Rojas Marín

Cartago, 28 de abril de 2021

Índice general

Índice de figuras

Índice de tablas

Índice de listados

Lista de símbolos y abreviaciones

Abreviaciones

ACC	Agent Communication Channel
ACL	Agent Communication Language
AID	Agent Identifier
AMS	Agent Management System
AP	Agent Platform
API	Application Programming Interface
AWS	Amazon Web Services
BSP	Board Support Package
CAN	Controller Area Network
CPU	Central Processing Unit
CSP	CubeSat Space Protocol
DF	Directory Facilitator
ELF	Executable Linkable Format
FDIR	Fault Detection and Recovery
FIPA	Foundation for Intelligent Physical Agents
GCC	GNU Compiler Collection
GPIO	General Purpose Input/Output
I2C	Inter-Intergrated Circuit
IDE	Intergrated Development Environment
IMT	Ingegneria Marketing Tecnologia
ISIS	Innovative Solutions In Space

ISP	In-System Programming
LED	Light Emitting Diode
MAES	Multi-Agent for Embedded Systems
MASSA	Multi-Agents Systems for Satellite Applications
MSVC	Microsoft Visual C++
MTS	Message Transport Service
OBC	On board Computer
PDO	Process Data Object
POSIX	Portable Operating System Interface
RISC	Reduced Instruction Set Computer
RS232	Recommended Standard 232
RTOS	Real Time Operating System
SDK	Software Development Kit
SRAM	Static Random Access Memory
UART	Universal Asynchronous Receiver-Transmitter

Capítulo 1

Introducción

El Laboratorio de Sistemas Espaciales del Instituto Tecnológico de Costa Rica (SETEC Lab) constituye una unidad de investigadores, profesores y estudiantes dedicados a realizar proyectos asociados a las tareas pertinentes de las misiones espaciales, tales como comunicación y desarrollo de sistemas de misión crítica [35]. Dichos planes necesitan funcionar con computadores compatibles con sistemas operativos de tiempo real (RTOS). Entre los equipos especializados del laboratorio para diseño de satélites específicamente se incluye la computadora espacial NanoMind A3200 de GomSpace, cuyo kit de desarrollo de software (SDK) permite implementar FreeRTOS (Open Source OS de tiempo real) y otras aplicaciones propietarias de pago asociadas a utilidades de sistemas espaciales como planeación de vuelo y telemetría [24]. Por el otro lado, el flujo de desarrollo económicamente accesible es ofrecido mediante el paquete de soporte de tarjeta (BSP) de la NanoMind. Este BSP únicamente soporta Protocolo Espacial para Cubesat (CSP), FreeRTOS y utilidades de GomSpace Shell.

FreeRTOS fue desarrollado por Real Time Engineers Ltd. hace más de 15 años y es distribuido por Amazon Web Services (AWS). Representa uno de los softwares de uso más extendido, posiblemente el más usado de su familia [5]. FreeRTOS sustancialmente es un kernel para microcontroladores o sistemas empotrados sobre el cual otras aplicaciones pueden correr organizando los hilos de trabajo según la prioridad definida por el diseñador de la aplicación. El kernel no está completamente desarrollado dentro de los descargables, sino que es responsabilidad del ingeniero adaptarlo a su sistema, aun así se pueden encontrar ejemplos abiertos en repositorios públicos desarrollados por otros usuarios.

GomSpace no es la única empresa desarrollando componentes compatibles con este kernel: Soluciones Innovadoras en el Espacio (ISIS), Ingeniería Mercadeo Tecnología (IMT) y Spacemanic tienen sus propios modelos de Computadores a Bordo (OBC) [31] compatibles con diferentes implementaciones y versiones de FreeRTOS, pues la comunidad en foros es muy activa e interesada en apoyar estos sistemas. Entre los intereses de innovar en el diseño de software para OBCs se incluye reducir tiempos de desarrollo dentro del ciclo de vida de un proyecto pues las tareas de verificación y validación de software, junto con la verificación y validación del hardware y simulaciones, corresponden a las tres acti-

vidades más comunes de la comunidad de satélites pequeños[33, p. 64]. Cada uno de estos proyectos espaciales deben programarse con las funciones generales de telemetría más las funciones y tareas específicas de la aplicación del satélite; la suma funciones resulta en una complejidad de software incremental. Este incremento puede ser atacado y minimizado con el metamodelo propuesto por J. Carvajal-Godínez [10, p. 99] y la implementación de C. Chan-Zheng: un paradigma de software basado en sistemas multiagentes llamado MAES que responde a la naturaleza distribuida de la arquitectura del computador[12]. La propuesta anterior también responde a la inquietud ante la falta de propuestas de desarrollo basadas en multiagentes con capacidades de tiempo real y necesidades de diseño para operar en sistemas embebidos con altas limitaciones tales como los equipos basados en el microcontrolador MSP430 [12, p. 1].

1.1. Antecedentes

El sitio web eoPortal [15] reporta detalladamente más de mil misiones realizadas y planificadas desde 1959 hasta 2025. Algunas de las misiones CubeSat ejecutadas entre el 2017 y 2019 incluyen al Dellingr [25], Phoenix [36] y el conocido primer satélite costarricense Proyecto Irazú [1]; todos equipados con la NanoMind A3200 cuyo software fue desarrollado junto con las herramientas del SDK de GomSpace, servicios de telemetría propietarios de GomSpace y el kernel FreeRTOS.

El paradigma de desarrollo de multiagentes para sistemas embebidos (MAES) ya ha sido probado en el sistema operativo de tiempo real TI-RTOS pues incluye herramientas de depuración y controladores preconfigurados de Circuito Inter-Intergrado (I2C), Transmisor-Receptor Asíncrono Universal (UART) y Programación en el Sistema (ISP) [12, p. 15]. Adicionalmente, el calendarizador de tareas preestablecido de TI-RTOS funciona por prioridades fijas de hilos (interrupciones de hardware, interrupciones de software, tareas y “Idle” ordenado de mayor a menor) [22]. Dicho sistema operativo es software de código abierto soportado y desarrollado por Texas Instruments [23].

Los resultados de MAES demuestran una reducción en el tamaño del código en casi un 50 %, un incremento en la asignación de memoria flash y de SRAM de 3 % y 7 % como máximos respectivos. Adicionalmente el uso del CPU y la potencia utilizada también incrementan, más no supera el 1 % [12, p. 42-48] por lo que el impacto en el uso de recursos es mínimo.

No obstante, el paradigma MAES solo se ha probado junto con el kit PocketQube de Alba Orbital [13], el cual por sus limitaciones en factor de forma, potencia y control de actitud se recomienda usarse principalmente para entrenamiento y demostración, no para misiones; contrario a los CubeSats [9].

1.2. Descripción del Problema

FreeRTOS es el software crítico para poner en funcionamiento a la NanoMind A3200 con el paradigma MAES y dar acceso a los demás miembros de la comunidad de desarrolladores de nanosatélites nuevos horizontes para sus proyectos, sin embargo, no existe una implementación funcional del paradigma compatible con el kernel de FreeRTOS.

Dicho paradigma solo se ha implementado en TI-RTOS para verificar sus capacidades y para transportarlo a FreeRTOS, es necesario examinar los aspectos de implementación propios del sistema operativo no considerados con anterioridad, incluyendo las diferencias entre compiladores y las arquitecturas objetivo.

1.2.1. Síntesis del Problema

Debe proveerse una herramienta que implemente el paradigma MAES junto con FreeRTOS para la NanoMind A3200 conciliando así la necesidad de reducir tiempos de desarrollo y volver el incremento de complejidad manejable.

1.3. Objetivo General

Construir una biblioteca que aplique las clases definidas en la implementación original del paradigma MAES mediante una API, que a su vez sean compatibles con las herramientas de desarrollo de software para la tarjeta NanoMind A3200 y FreeRTOS facilitando la implementación de multiagentes en sistemas empotrados con este kernel.

1.4. Objetivos Específicos

1. Adaptar el paradigma MAES para operar con FreeRTOS, construyendo las clases preestablecidas y agregando estructuras no presentes en el kernel necesarias para la empleabilidad de los agentes.

2. Verificar el funcionamiento de la API mediante una serie de programas de prueba desarrollados con la implementación en FreeRTOS para el compilador Microsoft Visual C++ (MSVC) donde se compruebe la comunicación entre agentes y distribución efectiva de información.

3. Evaluar la portabilidad de la API para sistemas de 32 bits de AVR (la familia AVR32 de Atmel) sobre la cual la NanoMind A3200 basa su diseño, específicamente el microcontrolador AT32UC3C, mediante una demostración sencilla que compare el consumo adicional de memoria al modelar el programa con agentes contra la implementación regular.

1.5. Enfoque Metodológico y Organización de los Paquetes de Trabajo

Para cumplir los objetivos expuestos se plantean tres fases de ejecución. Cada fase busca concretar el objetivo correspondiente descomponiéndole en paquetes de trabajo:

1. **Implementación del paradigma** Se contextua el modelo del paradigma MAES para posteriormente trasladarle a FreeRTOS y proponer una nueva API.
   - *Identificación de Dependencias*: Lectura completa de la API anteriormente desarrollada para TI-RTOS. Así se reconoce el comportamiento de los métodos y como los agentes encapsulan tareas y buzones de mensajes.
   - *Ubicación de Equivalencias en FreeRTOS*: Una vez reconocidas las funciones del kernel de TI-RTOS, se trasladan a funciones del Interfaz de programación de Aplicaciones (API) de FreeRTOS.
   - *Incluir Componentes Complementarios*: Se agregan los elementos no incluidos en la API de FreeRTOS como estructuras para la API de la biblioteca a desarrollar.
   - *Replanteamiento del Modelo*: Se presenta la versión del modelos de clases final con FreeRTOS.
2. **Simulación en MSVC** Mediante esta fase se valida el comportamiento adecuado de las funciones de la nueva API.
   - *Pruebas Unitarias*: Validar las estructuras y métodos del paradigma mediante un plan de pruebas unitarias.
   - *Propuesta de Demostraciones*: Tres diferentes arquitecturas con diferente numero de agentes donde el intercambio de datos por mensajes sea el común denominador.
   - *Flujo de Trabajo de MAES*: Programar las demostraciones siguiendo el flujo propuesto.
   - *Depuración de Comportamientos*: Identificar y eliminar errores en la biblioteca mediante las simulaciones.
3. **Síntesis de la Implementación para Arquitecturas AVR (NanoMind)** Concluye el desarrollo del proyecto determinando las capacidades del paradigma dentro de la arquitectura objetivo.
   - *Software Disponible*: Analizar las herramientas disponibles (IDEs y compiladores) compatibles con AVR que incluyan ejemplos de FreeRTOS con los controladores necesarios, pues programar aplicaciones para dispositivos AVR esta fuera del alcance de este proyecto.

- *Tarjeta AVR Objetivo*: Seleccionar los microcontroladores y kits de evaluación para verificar portabilidad según la disponibilidad de implementaciones.
- *Desempeño de Memoria*: Identificar el incremento de memoria al implementar multiagentes.
- *Prueba de Concepto para SDK*: Proponer modelos de aplicaciones que incorporen elementos libres del SDK de la NanoMind A3200.

Durante todas las fases se llevará un control de avances mediante un repositorio Git privado que se hará público con licencia MIT una vez la implementación para MSVC sea funcional. Se depurarán las fallas del código constantemente durante todas las fases.

Finalmente, en la etapa de cierre, se construirá un manual de usuario que ofrezca explicaciones teóricas mínimas para entender la implementación del paradigma, uso adecuado de la biblioteca y documentación completa de las clases y métodos de la API.

Para el desarrollo de software se empleará el IDE Visual Studio para depuración y edición de la biblioteca durante la primera y segunda fase. En la etapa final el ambiente de edición y compilación dependerá de las evaluaciones sobre disponibilidad de toolchains, compiladores e IDEs dedicados a las plataformas AVR como Microchip Studio, WinAVR o AVR-GCC para Linux.

Capítulo 2

Implementación del Paradigma MAES con FreeRTOS

2.1. Multiagentes y FIPA

Los sistemas multiagentes corresponden a redes de componentes de software conocidos como agentes, que interactúan y colaboran para realizar tareas complejas [19]. Cada uno de los agentes solo posee conocimiento parcial del problema y tiene poderes limitados sobre la plataforma correspondientes a su rol dentro del problema. Aún son tecnologías inmaduras, con fuertes promesas, pero carentes de usos generalizados y para extender su uso deben construirse estándares que definan su implementación [7]. Uno de los estándares propuestos corresponde al modelo de FIPA con enfoque en la comunicación entre agentes.

Foundation for Intelligent Physical Agents (FIPA) es una organización suiza sin ánimos de lucro fundada en 1996 con el objetivo de ofrecer estándares para agentes heterogéneos. Durante el 2005 tanto la organización como la IEEE Computer Society aceptaron incorporar las especificaciones de FIPA dentro de los comités [18]. Los componentes del paradigma formativo de FIPA se desarrollan en el FIPA23, Agent Management Specification [16]:

- **Agentes**: Unidades computacionales que contienen la funcionalidad de las aplicaciones. Actor fundamental dentro de las plataformas, debe poseer mínimo un dueño y un identificador (AID).
- **Facilitador de Directorio (DF)**: Componente opcional que provee servicios de paginas amarillas, es decir un repertorio información para los agentes sobre los servicios que proveen los demás agentes. Pueden optar por registrarse o no.
- **Sistema de Administración de Agentes (AMS)**: Es único por plataforma y administra el acceso a servicios. Mantiene un directorio de AIDs y sus direcciones de transporte. Es imposible para un agente obtener su AID sin registrarse en un AMS.

- **Servicio de Transporte de Mensajes (MTS)**: Método predeterminado de comunicación entre agentes. No se especifica ninguna estandarización, pero para conseguir un nivel de estandarización todos los mensajes deberán seguir el ACL (lenguaje de comunicación entre agentes) descrito por FIPA [17] (Tabla 2.1).

Tabla 2.1: Actos comunicativos de FIPA realizados mediante mensajes

Acto Comunicativo	Resumen
Accept Porposal	Aceptar petición para realizar alguna acción
Agree	Estar de acuerdo con realizar alguna acción
Cancel	Agente informa cancelación de solicitud de acción a otro agente
Call for Proposal	Llama propuestas para ejecutar acción
Confirm	Informa que la proposición es cierta cuando es dudosa
Disconfirm	Informa que la proposición es falsa cuando se cree verdadera
Failure	Informar que la acción se intentó y fracasó
Inform	Informa que la proposición es cierta
Inform If	El agente informa si la proposición es cierta o falsa
Inform Ref	Informa un objeto
Not Understood	Informa que la solicitud no fue comprendida
Propagate	Agente solicita a otro agente propagar el mensaje modificado a los agentes marcados
Propose	Realizar una propuesta
Proxy	Agente solicita a otro agente que actúe de intermediario
Query If	Preguntar a otro agente si la proposición es cierta o falsa
Query Ref	Preguntar a otro agente por un objeto
Refuse	Agente Rechaza realizar una acción y explica la razón
Reject Proposal	Agente rechaza propuesta de acción
Request When	Agente solicita a otro agente realizar acción cuando la proposición sea cierta
Request Whenever	Agente solicita a otro agente realizar acción siempre que la proposición sea cierta
Subscribe	Informar intención de ser notificado siempre que un valor de referencia cambie

- **Plataforma de Agentes (AP)**: Infraestructura funciona como contenedor lógico de los agentes. Incluye el sistema operativo, el software de soporte para los agentes y los componentes de FIPA anteriores.

- **Software**: Corresponde al nivel de abstracción más alto, pues describen a nivel funcional las capacidades demandadas por el sistema para su correcta operación.

  El modelo de referencia y sus componentes se ilustra en la figura 2.1.

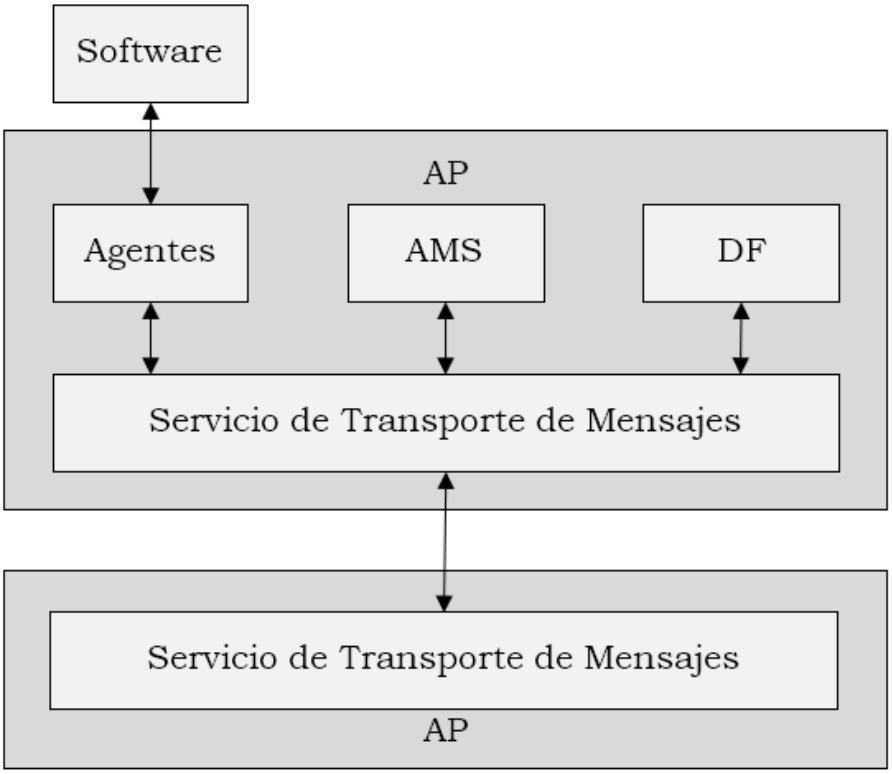

Figura 2.1: Modelo de referencia FIPA [16]

2.2. Microkernel FreeRTOS

El kernel FreeRTOS es descrito por sus desarrolladores como el sistema operativo de tiempo real para microcontroladores y microprocesadores líder en el mercado, con más de 18 años en desarrollo y con licencia MIT abierta a distribución [5]. Completamente documentado, soportado por AWS y con más de 30 implementaciones sobre plataformas de desarrollo.

La API de FreeRTOS [2] cuenta con estructuras como tareas, métodos de comunicación, herramientas de sincronización como temporizadores, interrupciones y eventos grupales; por último, servicios de administración de memoria heap y estático.

Las tareas dentro del kernel operan en dos estados (figura 2.2): Ejecutando (Running) y No Ejecutando (Not Running), donde el segundo es un superestado expansible en los estados Lista (Ready), Suspendida (Suspended) y bloqueada (Blocked) [4]. Las tareas transitan entre los estados dependiendo si esperan eventos, actualmente utilizan el procesador o si están apagadas y las prioridades asignadas a cada tarea. El ente encargado de seleccionar la siguiente tarea a ejecutar es el calendarizador de tareas mediante un algoritmo predefinido. FreeRTOS permite hacer pruebas circulares entre tareas de misma prioridad siempre y cuando estén listas para ejecutar e incluye tres algoritmos (adelanto, prioridades fijas y cortes de tiempo) y la capacidad de incluir un calendarizador de tareas cooperativo [6].

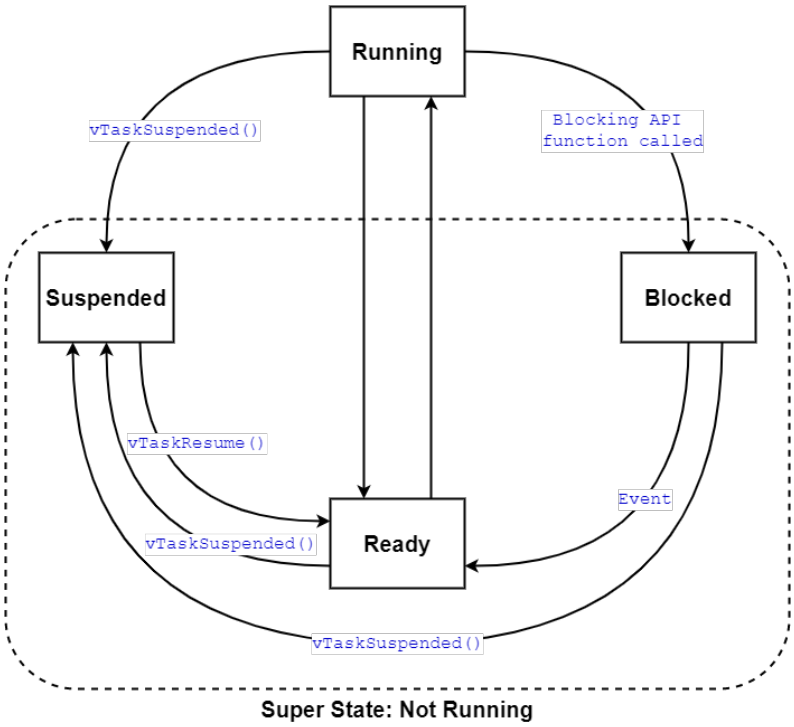

Figura 2.2: Estados y transiciones de las tareas [4]

Respecto a la comunicación entre tareas, se disponen de tres mecanismos: colas, semáforos y candados de exclusión mutua (mutex). Las colas son estructuras estáticas cuyo comportamiento por defecto permite que el primer elemento agregado sea el primer elemento leído (FIFO), sin embargo, el kernel también cuenta con métodos para alterar el orden de escritura y colocar elementos al inicio de la cola. En FreeRTOS, las colas funcionan copiando los datos y no guardado referencias a ellos. Al leer o escribir datos en colas las tareas pueden especificar un tiempo para la operación durante el cual la tarea se moverá al estado de bloqueo y si la cola estuviese llena al finalizar el tiempo la instrucción retornará un error (específicamente errQUEUE_FULL).

La segunda estructura disponible es el semáforo. Su principal objetivo es señalar a otras tareas para su ejecución, logrando sincronizarlas. Existen dos tipos cuya diferencia corresponde al tamaño: El semáforo binario tiene tamaño de uno mientras el semáforo contador tiene tamaño mayor a uno. Los semáforos no almacenan datos como tal sino un tipo de "ficha.º valor binario por espacio. Al igual que las colas darán errores al solicitar un encendido del semáforo si no hay espacios disponibles. Tanto los semáforos y las colas pueden asociarse mediante un conjunto de colas (queue set).

El último de los mecanismos corresponde al mutex, un caso especial de semáforo binario que busca limitar el uso de un recurso a una única tarea, no señalar entre tareas. Su operación es similar a los semáforos al punto que utilizan mismas funciones de la API para escribir y leer, más no para crear el mutex.

2.3. Paradigma MAES

Multi-Agents for Embedded Systems (MAES) es un paradigma propuesto por Carmen Chan-Zheng que desarrolla mediante programación orientada a objetos los componentes definidos por FIPA con el fin de operar en sistemas embebidos en tiempo real [12, p. 3-4]. Pretende conciliar las ventajas de la computación multiagente, las restricciones de las plataformas embebidas y las aplicaciones críticas de los sistemas operativos en tiempo real (RTOS). Específicamente mapea los agentes a través de una estrategia de hilos únicos, es decir que el comportamiento de un agente bloquea el actuar de los otros reduciendo concurrencia, pero aumentando el desempeño general. Además, los agentes poseen estados dentro de su ciclo de vida fácilmente trasladables a los estados de las tareas en los RTOS.

Inicialmente, se desarrolló una API para demostrar su funcionalidad dentro de TI-RTOS que utiliza el SYS/BIOS kernel, gracias a que Texas Instruments incluye un IDE, controladores de hardware y herramientas de análisis, más no es compatible fuera de los productos de la compañía y cuenta con una huella en la memoria mayor que FreeRTOS [12, p. 15]. Para contribuir a la madurez de esta tecnología, se propuso implementar este paradigma con un sistema operativo de uso más extendido.

El paradigma se construye mediante la abstracción orientada a objetos, donde las clases son extensiones de características propias del kernel, sea de tareas a agentes o buzones a mensajes [13]. El modelo propuesto por Chan y Carvajal se presenta en la figura 2.3, donde PQ_OBSW representa al software de la tarjeta utilizada (On-Board Software de Pocket Qube).

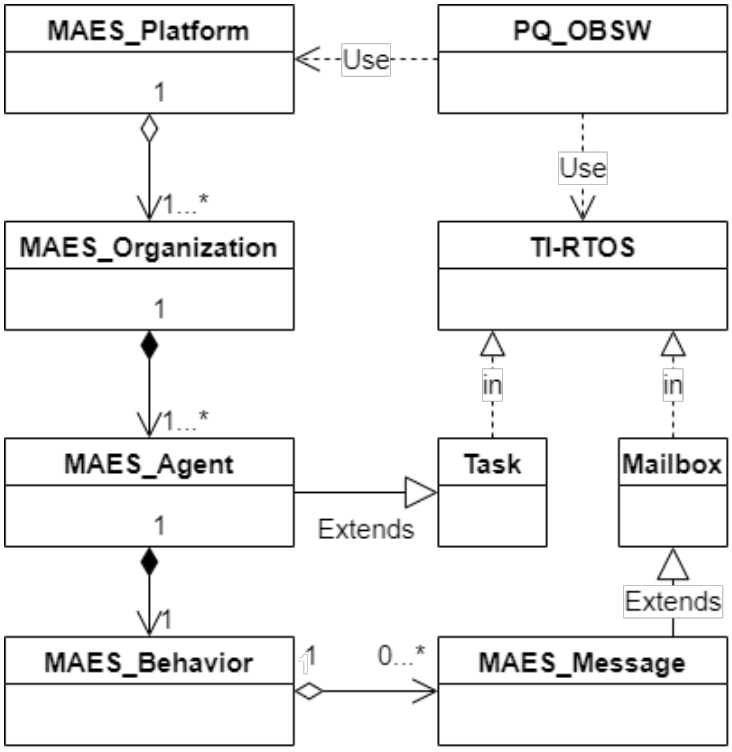

Figura 2.3: Diagrama de clases para MAES[13]

2.4. Implementación del paradigma

La implementación de este paradigma parte desde la documentación incluida con la API de MAES [11] para TI-RTOS, analizando el funcionamiento y propósito de los métodos, las dependencias del kernel SYS/BIOS y remplazándolas por equivalentes funcionales dentro de FreeRTOS. La diferencia conceptual más grande corresponde a los buzones, colas y semáforos. Dentro de FreeRTOS, no existen los buzones y aunque TI-RTOS los define como la unión de una cola que no produce interrupciones y un semáforo que provee la interrupción, el verdadero reemplazo funcional en FreeRTOS es una cola ya que si incluye interrupciones, por lo que no es necesario implementar un conjunto con ambas estructuras.

Para propósitos de simulaciones futuras, la implementación en este capítulo utiliza la implementación de FreeRTOS para MSVC con minGW, sin embargo, hay que considerar limitaciones que pueden existir para otros toolchains y compiladores para sistemas embebidos (IAR, winAVR o GCC por ejemplo); por lo que las estructuras y plantillas de C++ podrían cambiar al adaptar la biblioteca, pero las clases y funciones propuestas permanecen tal como se presentan.

2.4.1. Definiciones

Antes de desarrollar cualquier aplicación con el paradigma hay que considerar algunas redefiniciones que aumentan la legibilidad y separan los servicios específicos del paradigma de los servicios del kernel. Primero que todo, los AID corresponden a los Task Handle de las tareas, pero se redefinen como Agent AID. Segundo, los Queue Handle de las colas se renombran a Mailbox Handle para identificarlos como estructuras que administran los mensajes entre agentes. En adelante, se utilizará UBaseType_t al necesitar un entero no especificado ya que corresponde al tipo recomendado por la implementación de FreeRTOS seleccionado. Además, se definen límites para el máximo número de agentes dentro de la plataforma y máxima cantidad de miembros por organización.

```
#define Agent_AID TaskHandle_t                  // AID
#define Mailbox_Handle QueueHandle_t            // Mailbox Handle
#define AGENT_LIST_SIZE 64                      // Max. Agentes
#define MAX_RECEIVERS AGENT_LIST_SIZE - 1       // Max.
   Receptores
#define ORGANIZATIONS_SIZE 16                   // Max. Miembros
```

Listado 2.1: Definiciones y constantes del paradigma

2.4.2. Clase sysVars

Un elemento importante no presente en FreeRTOS son las variables de entorno de TI-RTOS, por lo que se tuvo que crear una clase y un objeto global que incluye un mapa ordenado como variable miembro privada, donde se introducen pares de AIDs y punteros de la clase Agent, para así identificar a un objeto por su tarea. La clase además cuenta con métodos para obtener, agregar y borrar pares del mapa. La composición de la clase se muestra en el listado 2.2.

```
class sysVars{
public:
    Agent* get_TaskEnv(Agent_AID aid);
    void set_TaskEnv(Agent_AID aid, Agent* agent_ptr);
    void erase_TaskEnv(Agent_AID aid);
    map<TaskHandle_t, Agent*> getEnv();
private:
    map<TaskHandle_t, Agent*> environment;
};
```

Listado 2.2: Clase sysVars (variables de entorno)

2.4.3. Clase Agent

La clase forma parte del componente Agente, junto con la clase Agent Organization (roles y membresías) y Agent Message (mensajes). La información y los recursos del agente se incluyen mediante un registro respectivo (listado 2.3 y listado 2.4), miembros privados de la clase.

```
typedef struct{
    Agent_AID aid;
    Mailbox_Handle mailbox_handle;
    char* agent_name;
    UBaseType_t priority;
    Agent_AID AP;
    org_info* org;
    ORG_AFFILIATION affiliation;
    ORG_ROLE role;
} Agent_info;
```

Listado 2.3: Registro para la información del agente

```
typedef struct{
    uint16_t stackSize;
    TaskFunction_t function;
    void* taskParameters;
} Agent_resources;
```

Listado 2.4: Registro para los recursos del agente

La clase cuenta con un único método público que retorna su AID. El constructor de la clase recibe una cadena de caracteres con el nombre, el número de prioridad y la profundidad del stack (listado 2.5).

```
class Agent{
private:
    Agent_info agent;
    Agent_resources resources;
    Agent();
public:
    Agent(char* name, UBaseType_t pri, uint16_t sizeStack);
    Agent_AID AID();
};
```

Listado 2.5: Clase Agent

Las clases Agent Message, Agent Platform y Agent Organization necesitan acceso a la información privada, por lo que se incluyen como clases amigas (friend class) en C++. Las declaraciones de la clase se incluyen en el listado 2.6.

```
class Agent{
...
public:
    ...
    friend class Agent_Platform;
    friend class Agent_Organization;
    friend class Agent_Msg;
};
```

Listado 2.6: Clases amigas de clase Agent

2.4.4. Clase Agent Organization

Esta clase representa las organizaciones que los agentes forman, los métodos gestionan los miembros y asignan los roles a los diferentes agentes. Cuando se instancia un objeto

de esta clase, se crea una organización vacía con un tipo de dos disponibles: equipo o jerarquía. El objetivo de crear una organización es configurar y limitar la comunicación entre agentes, pues en un equipo los roles de moderador y administrador pueden ser dados a diferentes individuos y la comunicación entre miembros no es limitada mas sí la comunicación fuera de la organización (solo puede ser realizada por el administrador o dueño). Por el otro lado la jerarquía restringe la comunicación entre miembros y solo puede ser realizada hacía el supervisor(moderador y administrador de la organización). Toda la información de la clase se almacena en el registro 2.7.

```
typedef struct{
    ORG_TYPE org_type;
    UBaseType_t members_num;
    UBaseType_t banned_num;
    Agent_AID members[AGENT_LIST_SIZE];
    Agent_AID banned[AGENT_LIST_SIZE];
    Agent_AID owner;
    Agent_AID admin;
    Agent_AID moderator;
} org_info
```

Listado 2.7: Registro con información de la organización

Es importante señalar que las organizaciones no son visibles para la clase plataforma, están desconectadas, pero los servicios del AMS sí identifican roles de administración y moderación, así como membresías.

2.4.5. Clase Agent Platform

La instancia de esta clase incluye al agente AMS y regula los servicios de plataforma, además de mantener un listado de los agentes activos en la plataforma. El registro con la información de la plataforma se muestra en el listado 2.8.

```
typedef struct{
    Agent_AID AMS_AID;
    char* AP_name;
    UBaseType_t subscribers;
} AP_Description;
```

Listado 2.8: Registro con informacion de la plataforma

La mayoría de los métodos de la clase son privados y solo son accesibles si el Agente AMS los invoca, y este recibe sus solicitudes mediante un AMS Task (o tarea de administración): La tarea con mayor prioridad del sistema, siempre expectante de un mensaje y mediante una serie de condiciones filtra el tipo de solicitud como se muestra en la figura

2.4; sea registrar, dar de baja (deregister), terminar, suspender, activar o reiniciar. Las solicitudes son posibles mediante el sistema de mensajería ofrecido por la clase Agent Message tipificando el mensaje como REQUEST según el FIPA ACL.

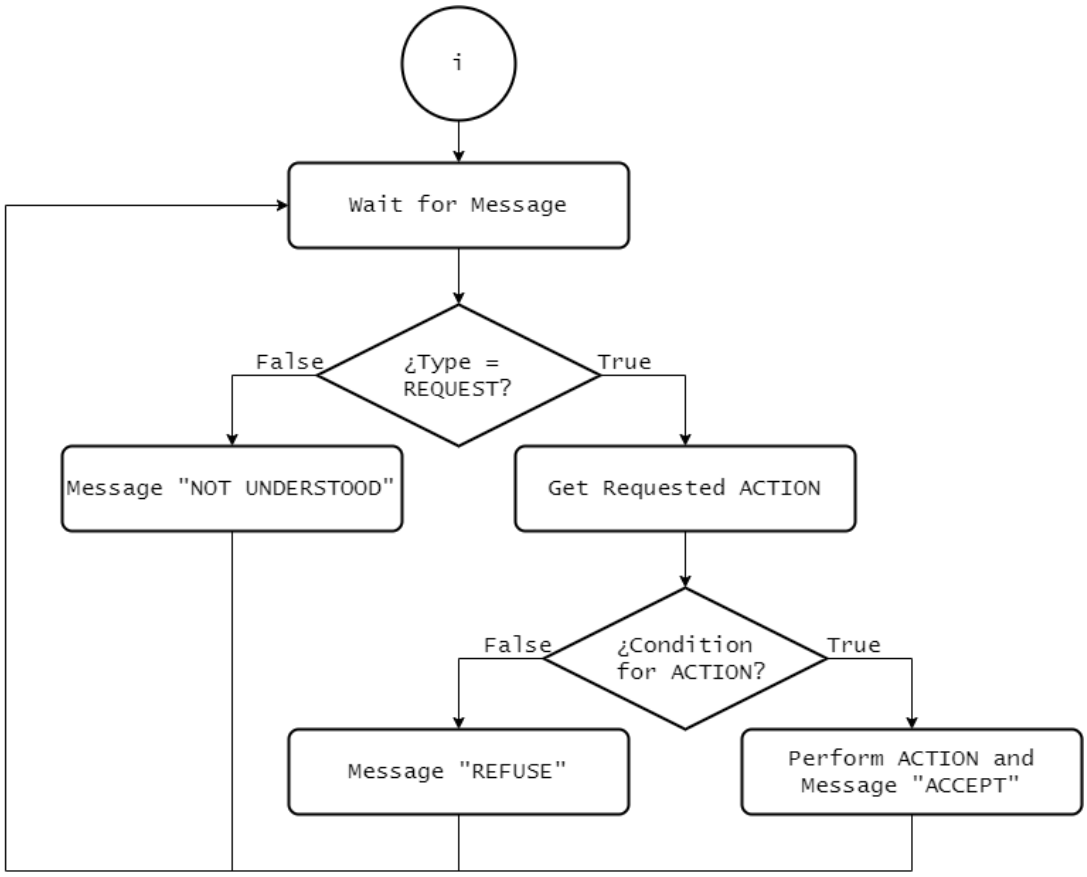

Figura 2.4: Diagrama de flujo del AMS para cualquier acción solicitada

La clase además cuenta con dos constructores posibles, con o sin condiciones de usuario respectivamente que se muestran en el listado 2.9. Esto se incluye para que el usuario pueda tener un control logístico sobre bajo cuales circunstancias se pueda acceder a los servicios de la plataforma y establecer restricciones. De nos ser incluidas la clase apunta a una instancia con valores por defecto.

```
class Agent_Platform{
private:
    ...
public:
    Agent_Platform(char* name);
    Agent_Platform(char* name, USER_DEF_COND* user_cond);
    ...
}
```

Listado 2.9: Constructores de la clase Agent Platform

El desarrollador deberá instanciar esta clase antes de que el calendarizador de tareas de FreeRTOS sea ejecutado. Luego deberá inicializar los agentes, método mediante el cual se crean las colas y las tareas asociadas tomando como parámetros la dirección del objeto agente y la función de tarea, se encarga de iniciarlos con la prioridad mínima posible y suspendidas inmediatamente. Es posible que la función necesite parámetros, por lo que se provee una segunda versión del método que incluye un argumento para los parámetros.

Por último, el inicio de la plataforma debe hacerse justo antes de ejecutar el calendarizador de tareas. Este método toma todos los agentes anteriormente iniciados y los registra en la plataforma, les asigna la prioridad correcta y los reanuda. Es posible registrar más agentes luego del inicio, pero es necesario tener al menos un agente activo (encargado de registrar los demás).

2.4.6. Clase Agent Message

Este segmento corresponde al núcleo de esta propuesta. Mediante la clase se distribuye información entre agentes, se definen tipos según FIPA, contenidos (la carga útil), se realizan solicitudes a la plataforma, se reportan errores y se reciben mensajes. Los mensajes como tales son un registro (listado 2.10) que almacena el remitente, destinatario, tipo y contenido del mensaje. Los métodos también utilizan los roles de la organización y las restricciones de estas.

```
typedef struct{
    Agent_AID sender_agent;
    Agent_AID target_agent;
    MSG_TYPE type;
    char* content;
} MsgObj;
```

Listado 2.10: Registro que encapsula el mensaje

2.4.7. Clase Generic Behaviour

Las funciones de los agentes pueden implementarse mediante wrappers que encapsulan uno o varios comportamientos. El paradigma ofrece una clase de comportamiento genérico que incluye métodos virtuales a ser sustituidos por el desarrollador y una instancia de la clase Agent Message. Los métodos principales son:

- Setup: Método para establecer variables constantes u otras configuraciones relevantes al comportamiento.
- Action: Función principal dentro del comportamiento.

- Done: Regresa un valor booleano verdadero si las condiciones establecidas los determinan; una vez cumplido, la ejecución del comportamiento de detiene.
- Execute: Incluye todas las subrutinas y el orden de ejecución (Figura 2.5).

Opcionalmente se incluyen métodos para detección, identificación y recuperación de fallas (FDIR). La ventaja de esto es agregar un sentido de autonomía a los agentes para administrar sus fallas sin una entidad centralizada (sea otro agente o la plataforma) vigilando y corrigiendo errores.

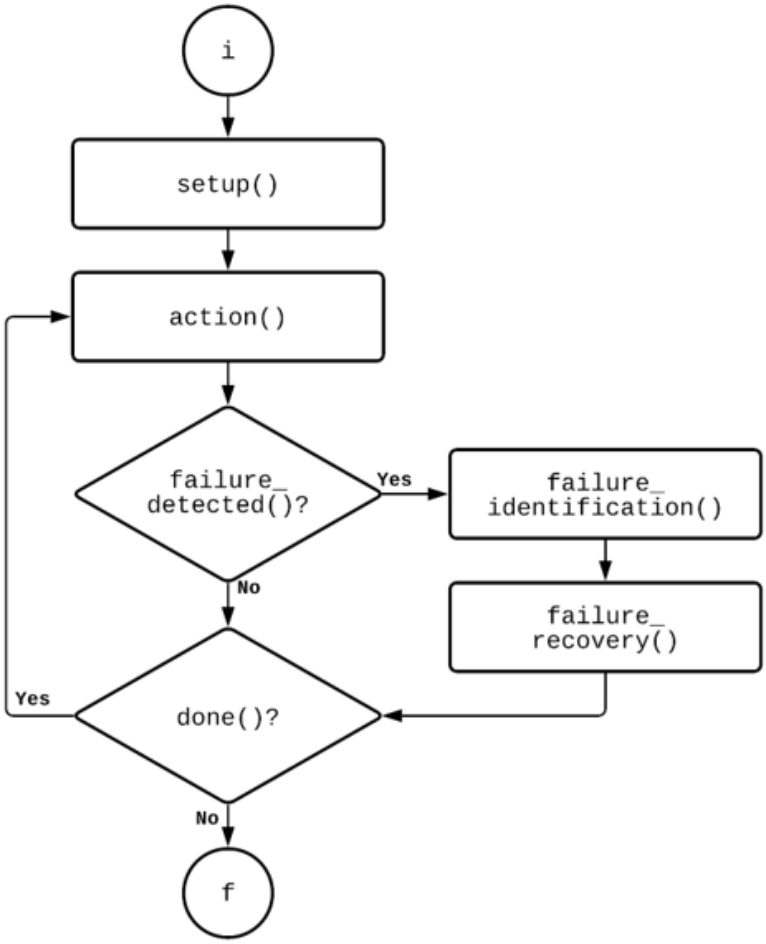

Figura 2.5: Diagrama de flujo del método execute

Se incluyen además dos clases derivadas más simples: Cyclic y One Shot. Heredan los métodos de FDIR, pero redefinen el método done para retornar verdadero o falso respectivamente y así producir comportamientos siempre cíclicos o de una única ejecución.

El listado completo de métodos se muestra en el anexo A.

2.5. API FreeMAES

De acuerdo con los cambios realizados anteriormente se presenta esta implementación con el nombre FreeMAES. Esta adaptación del paradigma es compatible con FreeRTOS e incluye una clase adicional respecto al modelo anterior (SysVars) para incorporar un contenedor que asocie los AID y los punteros de los agentes respectivos, función ya incluida en TI-RTOS. Se propone el modelo de clases presente en la figura 2.6.

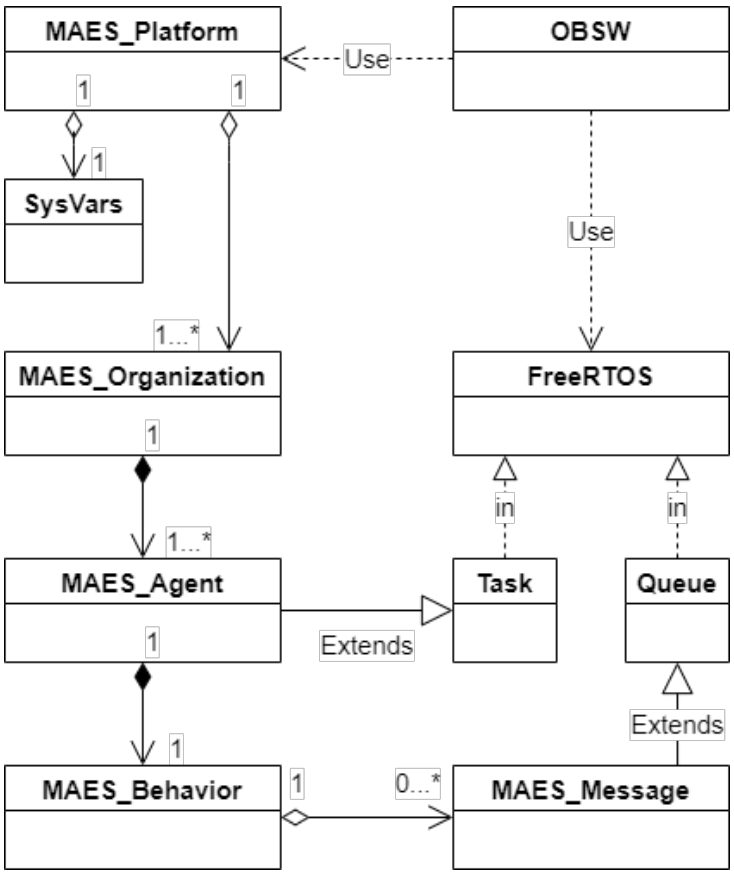

Figura 2.6: Modelo propuesto para MAES en FreeRTOS: FreeMAES

2.6. Observaciones Finales

En este capítulo se expuso brevemente la base teórica sobre la cual se desarrolla el paradigma MAES, así como el modelo de clases propuesto para la implementación del paradigma en el sistema operativo TI-RTOS. Considerando las clases presentadas y las funciones dependientes de TI-RTOS, se enuncian las modificaciones pertinentes a cada clase para la compatibilidad con FreeRTOS y se concreta efectivamente la nueva implementación bajo el nombre FreeMAES.

Capítulo 3

Evaluación de la API FreeMAES

Con el objetivo de verificar la funcionalidad de la API FreeMAES y demostrar como se construyen soluciones bajo un paradigma basado en multiagentes, se plantean tres diferentes pruebas conceptuales que involucran la comunicación entre agentes y la plataforma. Estas demostraciones no solo sirven de referencia para aplicaciones futuras, sino que demuestran cual es la ruta de diseño.

La metodología a seguir parte desde el modelo propuesto por Carvajal-Godínez [10, p. 100-101] llamado MASSA (Multi-Agent Systems For Satellite Applications), cuyo flujo de trabajo se expone en la figura 3.1.

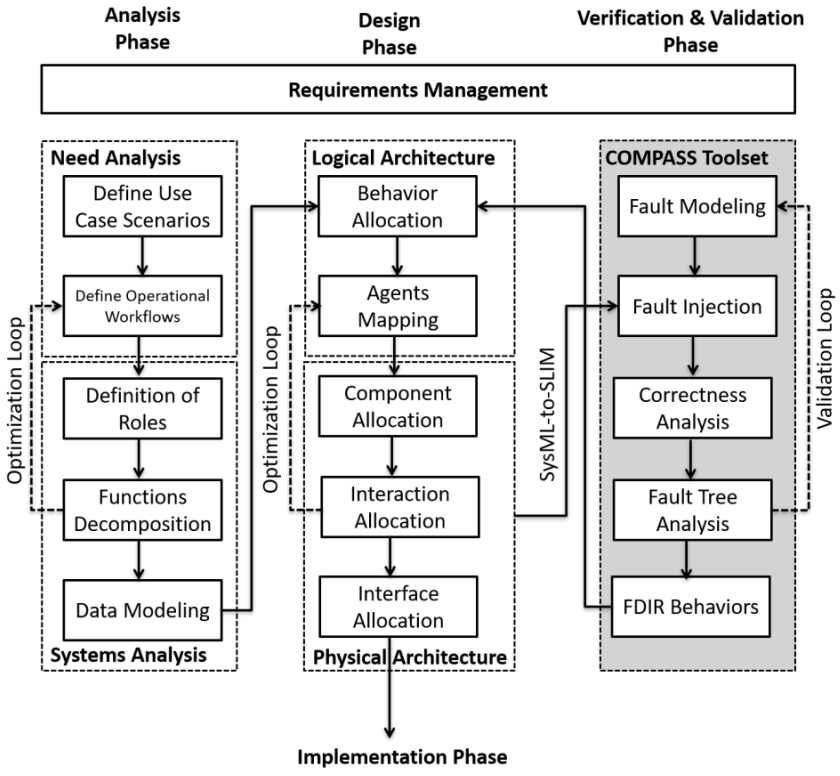

Figura 3.1: Flujo de trabajo para MASSA[10, p. 101]

Dicha metodología expone aspectos que el paradigma ya ha construido, como el modelado de datos y descomposición de funciones. Estas aplicaciones de ejemplo y cualquier solución futura están contempladas dentro de la fase implementación en adelante. Aún así, los ejemplos en este capítulo no incluyen métodos de FDIR, por lo que no se avanza a una fase de verificación y validación de un sistema completo. Sin embargo, de acuerdo a la matriz de validación de la metodología (Tabla 3.1), los diagramas de clases se deben examinar bajo un paradigma de pruebas unitarias y así cumplir con el criterio de probabilidad. La matriz también propone plantear casos de estudio, o aplicaciones de referencia, para validar la factibilidad.

Tabla 3.1: Matriz de Validación propuesta para la metodología MASSA[10, p. 112]

	Criterio de Validación			
Fase MASSA	Completitud	Consistencia	Factibilidad	Probabilidad
Análisis	Escenario de Caso de Uso Diagramas y Funciones Diagramas de Descomposición	Especificaciones del Flujo de Trabajo	Implementación de Casos de Estudio	Evaluación de Métricas Requerida
Diseño	Diagramas de Interacción entre Agentes			Análisis del Presupuesto del Sistema
Verificación	Diagramas de Árboles de Fallas	Modelos de Simulación		Análisis de Exactitud
Implementación	Diagramas de Clases	Escenarios Detallados de Operación		Paradigma de Pruebas Unitarias

De acuerdo a lo anterior, se puede notar que las demostraciones corresponden a casos de estudio del paradigma propuesto, y antes de ser desarrollados se debe asegurar la operación adecuada de las características y funciones de la API mediante una serie de pruebas unitarias ordenadas.

3.1. Pruebas Unitarias

Para el desarrollo de las pruebas se identifican características necesarias y se ejecutan por separado de la aplicación. Para sistemas que implementan orientación a objetos puede ser difícil examinar datos privados sin modificar el contexto de las clases o agregar funciones a la estructura, por ello se debe trazar un plan de pruebas que liste el rasgo a probar, el procedimiento y el resultado esperado para confirmar el funcionamiento adecuado.

La herramienta para realizar las pruebas corresponde al paradigma Microsoft Unit Testing para C++ incluido en el IDE Visual Studio y la documentación adjunta.

3.1.1. Plan de Pruebas Unitarias

Se separan las pruebas en tres fases, ya que algunas funciones dependen del comportamiento de otras y para hacer pruebas unitarias correctamente se debe evitar examinar múltiples atributos a la vez. Este procedimiento se ejecuta iterativamente por cada fase hasta obtener los resultados esperados. Además, las pruebas se concentrarán en funciones de la API FreeMAES que puedan operar fuera del calendarizador, pues una vez activo no se puede detener sin finalizar la aplicación.

La fase inicial examina la creación de los objetos que contienen los diferentes componentes del paradigma. Se prueban primero todos los constructores de objetos y se comparan valores específicos (variables como nombres o números) del objeto contra el valor esperado de un comportamiento correcto o por valores en consola. En la Tabla 3.2 se describen los detalles de esta fase.

Tabla 3.2: Primera Fase del Plan de Pruebas

Característica	Detalles de la Prueba	Resultado Esperado
Constructor de Plataforma	Crear una variable con el nombre de la plataforma, luego instanciar el objeto con dicho parámetro y comparar la variable original con el método de la plataforma que obtiene su nombre	Valores iguales
Constructor de Mensaje	Se examina que el mensaje del comportamiento tenga un valor de Caller igual a 0	Valores iguales
Subclase de Comportamiento	Construir una subclase derivada del comportamiento One Shot. El método Action incluye un lazo for de hasta cinco iteraciones e imprimirá en consola un mensaje por iteración. Luego, instanciar un objeto de la subclase y llamar al método Execute	Cinco veces el mismo mensaje en consola
Constructor de Agente	Instanciar un agente y revisar si su AID es nulo	Valor nulo
Constructor de Organización	Instanciar una organización con tipo TEAM sin agregar ningún agente y revisar si el tipo coincide	Tipos iguales

La siguiente fase se orienta a la inicialización de los agentes y plataforma. Mediante este procedimiento se asignan los comportamientos a cada agente para que el calendarizador

entre en funcionamiento. Se revisará parte por parte la creación de la tarea, correcta asignación al agente y el registro de los agentes a la plataforma como se desarrolla en la Tabla 3.3.

Tabla 3.3: Segunda Fase del Plan de Pruebas

Característica	Detalles de la Prueba	Resultado Esperado
Inicialización del Agente	Con una plataforma, un agente y un comportamiento One Shot encapsulado por una función wrapper ya instanciados se registran a la plataforma y se revisa que el AID del agente sea diferente de nulo	Un valor no nulo
Arranque de la Plataforma	Se inicializa un agente con un comportamiento One Shot y se arranca la plataforma registrando al agente y al administrador. Luego se revisa el número de suscriptores	Dos suscriptores

Por último, una fase para los métodos de mensajes; se debe confirmar tanto el tipo del mensaje y sus contenidos. Para ello se prueban los métodos que ajustan y obtienen el contenido, tipifican los mensajes y controlan el registro de los suscriptores como se indica en la Tabla 3.4.

Tabla 3.4: Tercera Fase del Plan de Pruebas

Característica	Detalles de la Prueba	Resultado Esperado
Agregar Suscriptores	Agregar un agente a los suscriptores y revisar el error reportado por el método	Tipo “NO_ERRORS”
Remover Suscriptores	Agregar un agente a los suscriptores, removerlo con el método correspondiente y revisar el error reportado por el método	Tipo “NO_ERRORS”
Ajuste de Tipo	Establecer el tipo del mensaje como REQUEST y solicitarlo mediante otro método, revisar si coinciden	Tipos iguales
Ajuste de Contenido	Establecer el contenido como la cadena de caracteres “Prueba Unitaria” mediante el método de ajuste y compararlo contra la cadena extraída	Contenidos iguales

Una vez se cumplen los resultados esperados mediante depuraciones de comportamientos e inspecciones de código, se procede a aprobar este segmento de revisiones y avanzar a los casos de estudio.

3.2. Planteamiento de Casos de Estudio

Mediante los siguientes casos de estudio se pretenden probar las características de la API dependientes de FreeRTOS, principalmente las colas y mensajes. Se espera obtener simulaciones que correspondan con los diagramas de secuencias proyectados.

3.2.1. Sender-Receiver

Únicamente dos agentes (figura 3.2); un emisor que envía mensajes periódicamente y un receptor que espera mensajes e imprime en pantalla una confirmación de recibido. El receptor tiene una prioridad mayor que emisor, pero entra en el estado de bloqueo al esperar los mensajes, lo que le permite al emisor usar el procesador y realizar el envío tal como la figura 3.3 ilustra.

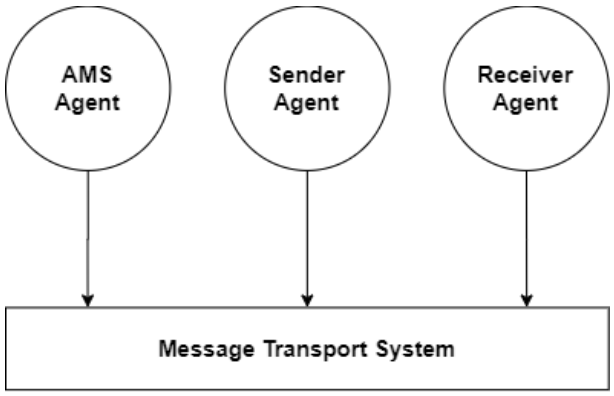

Figura 3.2: Arquitectura de la primera demostración

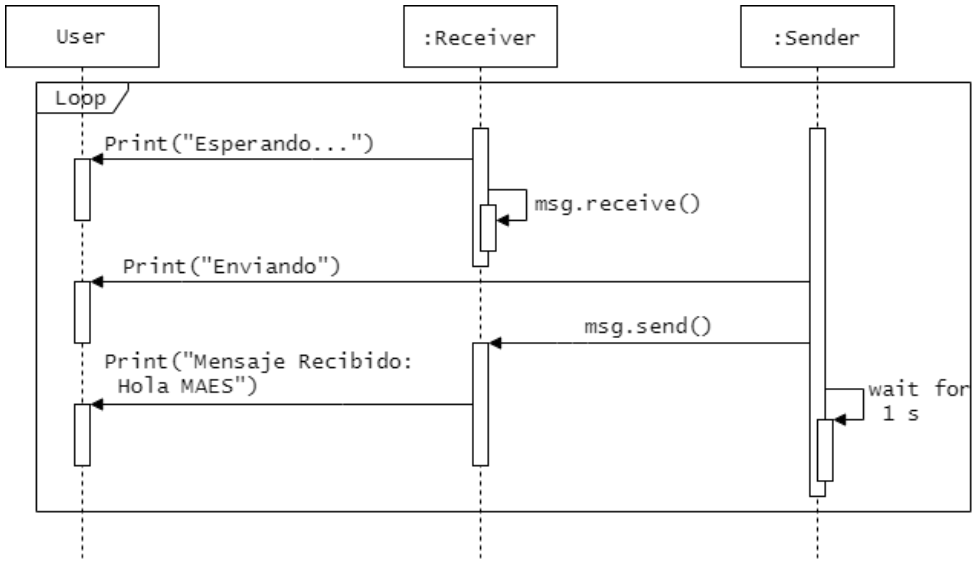

Figura 3.3: Diagrama de secuencia de la primera demostración

3.2.2. Papel, Piedra o Tijeras

Papel, Piedra o Tijeras: Tres agentes (árbitro, jugador A y jugador B ilustrados en la figura 3.4) y dos comportamientos (jugar y arbitrar). Los jugadores se turnan para sortear números y escoger entre papel, piedra o tijera. Cuando el arbitro recibe algún mensaje, solicita a la plataforma suspender al jugador. Una vez ambos jugadores estén suspendidos, el arbitro determina un ganador y reanuda la ejecución de los jugadores. La secuencia que se sigue se incluye en la figura 3.5.

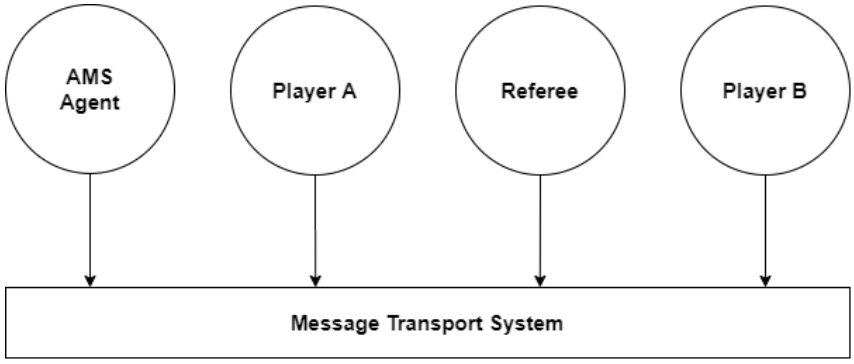

Figura 3.4: Arquitectura de la segunda demostración

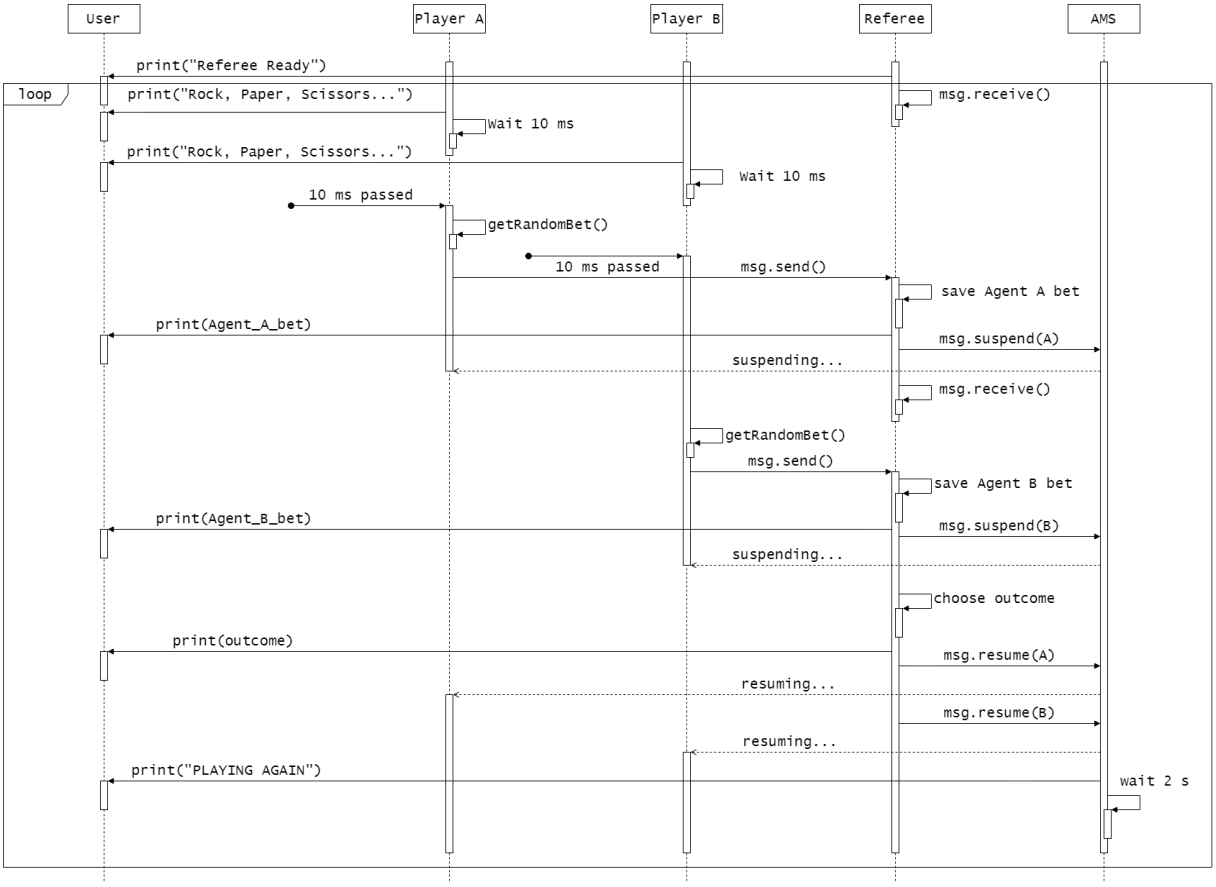

Figura 3.5: Diagrama de secuencia de la segunda demostración

3.2.3. Telemetría

Un ejemplo desarrollado para probarse en una tarjeta física con UARTs y sensores. Tres agentes de registro (corriente, voltaje y temperatura) a diferentes ritmos solicitan a un cuarto agente generador los valores respectivos. Cada agente debería tener una prioridad distinta descendiente de acuerdo al ritmo (entre mayor sea la espera en milisegundos, menor la prioridad), sin embargo, por limitaciones de la plataforma el agente generador y el registrador comparten una prioridad de 3. La arquitectura del ejemplo se muestra en la figura 3.6 y la el diagrama de secuencia en la figura 3.7.

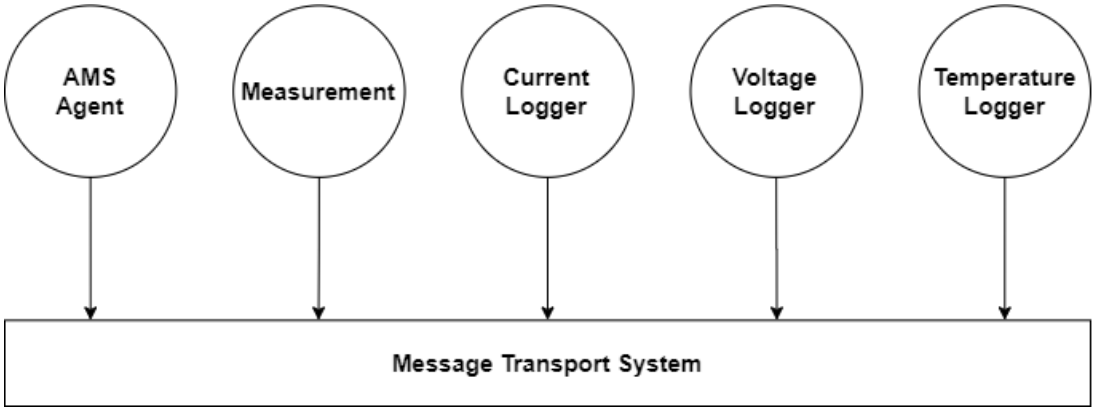

Figura 3.6: Arquitectura de la tercera demostración

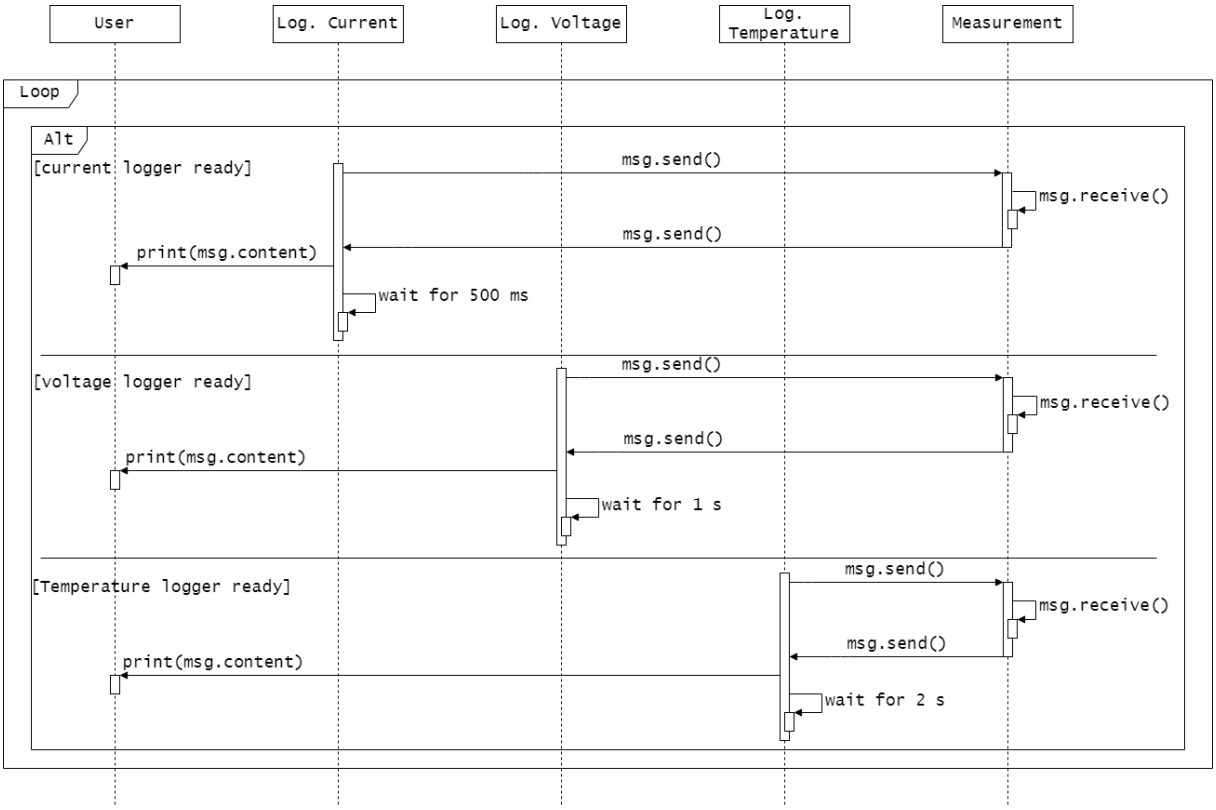

Figura 3.7: Diagrama de secuencia de la tercera demostración

Estos ejemplos se construyeron sobre una solución de Visual Studio que emplea el puerto de MSVC-mingW de FreeRTOS, que en si tiene algunas limitaciones respecto a otras plataformas, principalmente en las prioridades disponibles. Teóricamente es factible implementar la API sobre diferentes plataformas mientras incluyan memoria dinámica pues utiliza las funciones vTaskCreate y xQueueCreate, disponibles solo si el puerto tiene asignación dinámica. Sin embargo, existen versiones estáticas de las funciones por lo que se podría expandir para abarcar la totalidad de plataformas.

3.3. Estructura de la Solución

A continuación se presenta una descomposición de como se preparan los diferentes componentes de una aplicación multiagente junto con de la API. Esta estructura se comparte entre todos los ejemplos simulados que se desarrollaron.

Primero, se debe incluir todos los archivos fuente, FreeRTOS con su respectivo puerto (directorios "include" y "portable" correctos) y el archivo de cabecera de la API (maes-rtos.h). Para acceder a las clases de la API se debe utilizar el namespace MAES como se muestra en el listado 3.1.

```
/* FreeRTOS */
#include "FreeRTOS.h"

/* Incluir MAES API */
#include "maes-rtos.h"

/* Namespace de MAES (incluye std namespace) */
using namespace MAES;
```

Listado 3.1: Llamados a bibliotecas y namespace de MAES

Seguidamente se deben instanciar a los agentes con su nombre, prioridad y profundidad de pila correspondientes. En el listado 3.2 se incluyen instancias de los agentes con misma una prioridad y profundidad de pila.

Es deber del desarrollador elegir un tamaño adecuado y conocer el máximo de prioridades de la plataforma. En FreeRTOS las prioridades pueden tener valores desde 0, la prioridad de la tarea Idle, hasta configMAX_PRIORITIES−1 que es la prioridad reservada a la tarea del AMS dentro del paradigma, por lo tanto los agentes solo pueden operar con prioridades iguales o mayores a 1 y menores a configMAX_PRIORITIES−1. Por ultimo debe instanciar una plataforma y darle nombre.

```
/* Instancias de Agentes */
uint16_t stackDepth = 1024 // Ejemplo

Agent AgentA("A", 1, stackDepth);
Agent AgentB("B", 1, stackDepth);
Agent AgentC("C", 1, stackDepth);
...

Agent_Platform AP("platform");
```

Listado 3.2: Instanciación de los agentes

Luego de instanciar los agentes se construyen los comportamientos (listado 3.3), es decir el código funcional de la aplicación a implementar. Se acceden públicamente como subclases de la clase de comportamiento deseada (cíclica, one shot o genérica) y deben definirse las funciones correspondientes en los métodos como se explican en la subsección 2.4.7. Los métodos de FDIR también se desarrollan aquí como muestra el listado 3.4

```
/* puede ser generico, ciclico o one shot */
class alphaBehaviour :public Generic_Behaviour {
public:
    void setup() {
        /* Ajustes y variables locales */
    }
    void action() {
        /* accion del comportamiento */
    }
    bool done() {
        /* condicion de terminado */
    }
};

class betaBehaviour :public CyclicBehaviour {
    ...
};

class gammaBehaviour :public OneShotBehaviour {
    ...
};
```

Listado 3.3: Subclases de comportamientos

```
/* puede ser generico, ciclico o one shot */
class alphaBehaviour :public Generic_Behaviour {
public:
    ...
    bool failure_detection() {
        /* deteccion de fallas */
    }
    void failure_identification() {
        /* identificacion de fallas */
    }
    void failure_recovery() {
        /* recuperacion de fallas */
    }
};
```

Listado 3.4: Subclases de comportamientos de FDIR

Las funciones anteriores están asociadas mediante el método execute de las clases de comportamientos, sin embargo, para asignar el método como la función de tarea del agente se debe primero encapsular una instancia de la clase y llamar al método dentro de una función wrapper. El listado 3.5 ejemplifica los pasos mencionados.

```
/* definicion del wrapper para comportamiento */
void alphaTask(void* pvParameters) {
    alphaBehaviour b;
    b.execute();
}
void betaTask(void* pvParameters) {
    betaBehaviour b;
    b.execute();
}
void gammaTask(void* pvParameters) {
    gammaBehaviour b;
    b.execute();
}
```

Listado 3.5: Funciones wrapper para los comportamientos

Finalmente, se desarrolla el main de la aplicación. Primero se deben inicializar los agentes con el método agent_init, mediante este método de la la clase plataforma se crean las tareas del programa y se colocan en un estado de suspensión hasta el encendido de la plataforma. El encendido (método boot de la plataforma) permite subscribir a los agentes y reanudar sus tareas. Debe ser el último paso antes de llamar al scheduler y permitirle a

sistema operar. El listado 3.6 concluye el arquetipo de las soluciones mediante la API de agentes.

```
int application_main() {

    /* se inicializan y registran todos los agentes */
    AP.agent_init(&AgentA, alphaTask);
    AP.agent_init(&AgentB, betaTask);
    AP.agent_init(&AgentC, gammaTask);
    ...
    AP.boot();

    /* se inicia el scheduler de FreeRTOS */
    /* a partir de aqui el codigo que se incluya es
       inalcanzable*/
    vTaskStartScheduler();
    for (;;){
    }

    return 0;
}
```

Listado 3.6: Main de la aplicación con agentes

3.4. Resultados de Simulaciones

Como se explicó al inicio de este capítulo, se presentan tres situaciones diferentes para demostrar el comportamiento de la API. Mediante las aplicaciones propuestas se busca verificar el funcionamiento de las distintas formas de comunicación entre agentes: difusión de mensajes vacíos, solicitudes al AMS y transmisión de cadenas de caracteres.

3.4.1. Sender-Receiver

El primer ejemplo involucra un mensajero emisor (Sender) y un agente receptor (Receiver). En la primera marca de la figura 3.8 el receptor adelanta a la tarea de Idle pues debe imprimir en consola que espera recibir un mensaje, luego en la segunda marca es adelantado por el mensajero pues su ejecución ha sido bloqueada hasta recibir un mensaje. En la marca 3, el mensajero muestra en pantalla que está enviando el mensaje y al enviarlo es adelantado por el receptor (marca 4) quien puede seguir operando. Imprime en pantalla una confirmación de recibido y vuelve a esperar un mensaje. Como la espera ocasiona un bloqueo, es nuevamente adelantado por mensajero, quien tiene una única instrucción de

esperar por 500 milisegundos. Al "dormir", el procesador queda relevado a la tarea de Idle hasta que mensajero pueda enviar un mensaje (marca 5). Los eventos desde la segunda y quinta marca se seguirán repitiendo a partir de la marca 6.

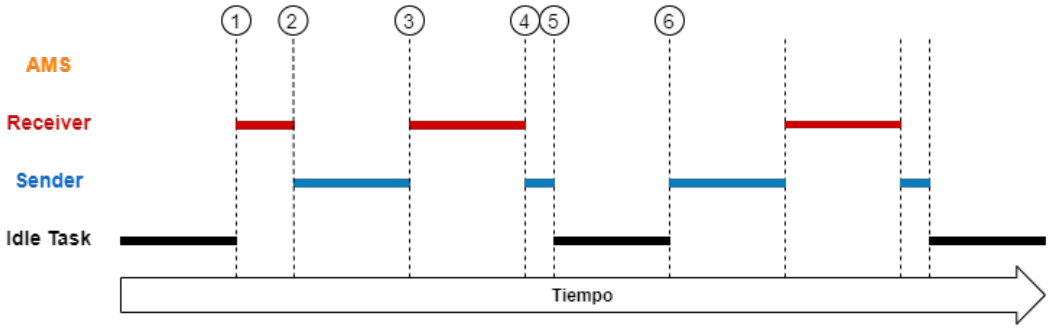

Figura 3.8: Diagrama de tiempo de la primera demostración

La figura 3.9 muestra los diferentes mensajes que los agentes imprimen a lo largo de cinco ejecuciones, además de la secuencia de inicialización.

```
MAES DEMO
Mensajero Listo
Receptor Listo
Boot exitoso
Esperando...
Enviando mensaje...
Mensaje recibido: ¡Hola MAES!
Esperando...
Enviando mensaje...
Mensaje recibido: ¡Hola MAES!
Esperando...
Enviando mensaje...
Mensaje recibido: ¡Hola MAES!
Esperando...
```

Figura 3.9: Mensajes impresos por la primera demostración

3.4.2. Papel, Piedra o Tijeras

El siguiente ejemplo simula múltiples rondas de un juego de papel, piedra o tijera. Esencialmente los agentes se turnan para elegir un valor aleatorio y según ello una mano específica. En la primera marca el agente árbitro (de nombre Referee) adelanta a la tarea Idle, pero cede casi inmediatamente (marca 2) al bloquear la ejecución y el jugador A adelanta al árbitro, este imprime su mensaje al jugar y espera por 10 milisegundos. Al esperar, entrega la ejecución al jugador B (marca 3), este replica al primer jugador y al esperar los 10 milisegundos correspondientes, la ejecución vuelve al primer jugador (marca 4). En la marca 5 el jugador A envía su mensaje y esto desbloquea al árbitro quien registra la elección del jugador y realiza una solicitud al administrador de plataforma para suspender al jugador. Al recibir la solicitud, el AMS adelanta a los demás agentes y suspende al agente objetivo (marca 6). En la marca 7 el AMS vuelve a esperar solicitudes y se bloquea, entregando el procesador al árbitro, como no hay más mensajes, se entrega el procesador al jugador aun activo. Al enviar el mensaje, se vuelve a repetir la secuencia entre las marcas 5 hasta 7 en la marca 9.

Con ambos agentes suspendidos, el árbitro esperará 2 segundos (marca 10). Luego de la espera determina un ganador y seguidamente solicita al AMS reanudar a los agentes jugadores (marcas 11 a 12 para jugador A y la marca 13 para jugador B). En la marca 14 se vuelve a jugar otra ronda.

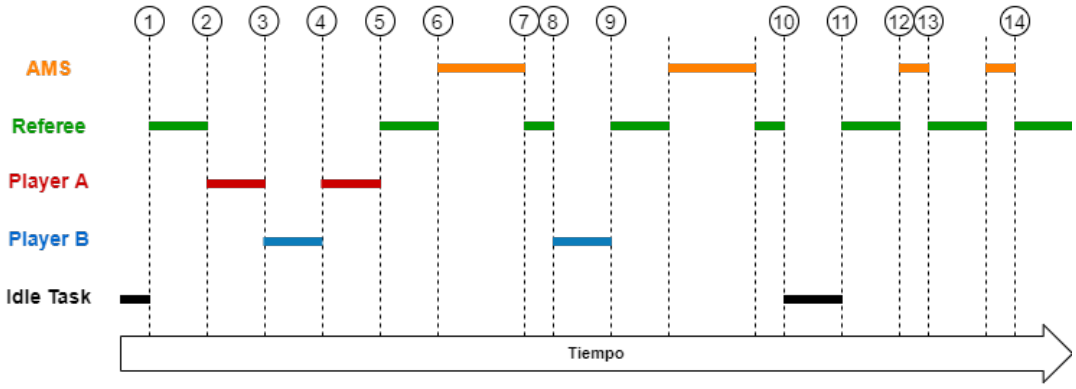

Figura 3.10: Diagrama de tiempo de la segunda demostración

En la figura 3.11 se aprecia como en la primera ronda y segunda ronda el agente jugador B gana, en la tercera ronda hay un empate y en cuarta ronda, la última ejecutada, el jugador A logra su primera victoria. No hay un conteo continuo de los ganes o empates pues se analiza cada ciclo por aparte, pero es completamente implementable. La aplicación principalmente destaca por la comunicación con el AMS y la solicitudes para manipular el estado de otros agentes. Bajo esta configuración sin organizaciones, cualquier agente podría solicitar la suspensión de otro, por lo que un sello extra de control sobre la comunicación puede ser nombrar al árbitro como administrador del equipo o jefe de la jerarquía.

```
MAES DEMO
Boot exitoso

REFEREE READY
Player A: Rock, Paper, Scissors...
Player B: Rock, Paper, Scissors...
Player A: ROCK
Player B: PAPER
Referee: PLAYER B WINS!
------------------PLAYING AGAIN--------------------

REFEREE READY
Player A: Rock, Paper, Scissors...
Player B: Rock, Paper, Scissors...
Player A: SCISSORS
Player B: ROCK
Referee: PLAYER B WINS!
------------------PLAYING AGAIN--------------------

REFEREE READY
Player A: Rock, Paper, Scissors...
Player B: Rock, Paper, Scissors...
Player A: PAPER
Player B: PAPER
Referee: DRAW!
------------------PLAYING AGAIN--------------------

REFEREE READY
Player A: Rock, Paper, Scissors...
Player B: Rock, Paper, Scissors...
Player A: ROCK
Player B: SCISSORS
Referee: PLAYER A WINS!
```

Figura 3.11: Mensajes impresos por la segunda demostración

3.4.3. Telemetría

La última demostración tiene un comportamiento similar al primer ejemplo. No se construye una organización para asociar a los agentes ni se utiliza el AMS, pero el escenario corresponde a una aplicación real para sistemas embebidos y OBCs.

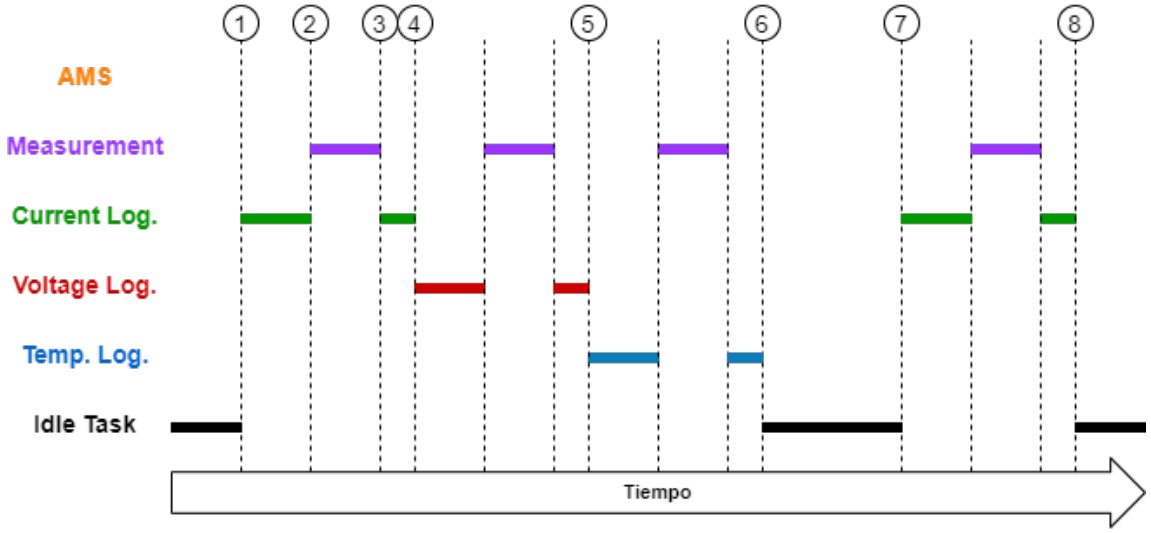

Figura 3.12: Diagrama de tiempo de la tercera demostración

Descomponiendo de la figura 3.12 en sus respectivas marcas:

1. Inmediatamente luego del inicio, la tarea Idle es adelantada por el registrador corriente (el agente con mayor prioridad listo para ejecutar) pues el generador entra inmediatamente luego de su activación al estado de bloqueo hasta recibir un mensaje.

2. El agente registrador de corriente es adelantado por el agente generador, este escribe un número aleatorio y lo envía al solicitante.

3. Al responder vuelve a entregar el procesador al registrador pues espera otro mensaje.

4. El registrador imprime el valor en consola y espera un tiempo asignado dependiendo del tipo de registrador; corriente espera 500 milisegundos. Estando el agente bloqueado, es adelantado por el registrador de voltaje. Se repite la secuencia de eventos anterior para este agente, la diferencia principal corresponde al tiempo de espera de 1000 milisegundos.

5. El último registrador (temperatura) realiza sus solicitudes y espera por 2000 milisegundos.

6. Todos los agentes están bloqueados por lo que la tarea Idle tiene el control hasta la activación de un agente.

7. El agente con menor espera, el registrador de corriente, termina su espera y está listo para volver a imprimir nuevos valores.

8. Nuevamente el agente vuelve a esperar, por lo que las diferentes solicitudes se irán sorteando en adelante tal como los agentes terminen sus tiempos de espera.

Luego de varios segundos se puede apreciar el orden regular de los agentes tal como en la figura 3.13, donde entre cada registro de temperatura se reportan dos registros de voltaje y cuatro de corriente de acuerdo a sus respectivos ritmos.

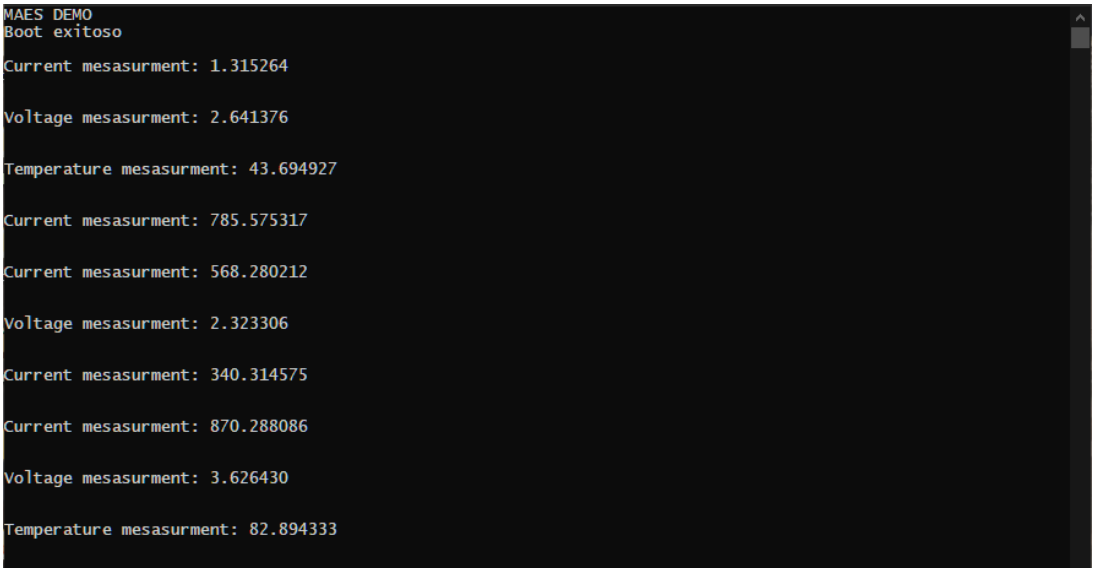

Figura 3.13: Mensajes impresos por la tercera demostración

3.5. Discusión de Resultados

Durante las pruebas unitarias el método de arranque de la plataforma (boot) interfería con otros llamados a si mismo aún afuera de la clase de pruebas, pues el bloque de control de tareas de FreeRTOS no se vacía entre las ejecuciones de cada método de la clase si se ejecutaban simultáneamente; solamente se puede eliminar tareas cuando la rutina de servicio de interrupciones está activa, por lo que cada ejecución se tuvo que examinar por separado. Una vez esto hecho, se pudo depurar los comportamientos no dependientes del calendarizador.

Los casos de estudio justamente progresan la revisión incluyendo el calendarizador y permitiendo verificar si las tareas encapsuladas por los agentes y la comunicación entre ellos se comportan de la manera pensada. De acuerdo a los mensajes de consola obtenidos mediante la implementación de los casos se puede confirmar que el paradigma y sus métodos son funcionales y válidos.

3.6. Observaciones Finales

En base a la metodología MASSA y conforme a su criterios de validación se propuso un plan de pruebas unitarias cuya ejecución fue exitosa, se describieron casos de uso y desarrollaron sus diagramas de secuencia correspondientes con los escenarios detallados de operación. Las simulaciones de dichos casos cumplieron las expectativas de la prueba, ofreciendo finalmente una implementación completamente funcional para MSVC con MinGW.

Capítulo 4

Portabilidad de FreeMAES para AVR8 y AVR32

En los capítulos anteriores se presentó una propuesta de biblioteca que implementa el paradigma MAES en C++ y se demostró su funcionalidad correcta mediante el puerto MSVC con MinGW de FreeRTOS. Aunque esto contribuye al diseño de aplicaciones multiagente, no completa la meta de uso y portabilidad en sistemas embebidos. Por ello se debe demostrar que los microcontroladores disponibles pueden soportar las características de la biblioteca.

La tarjeta para la cual se quiere demostrar la usabilidad de la biblioteca corresponde a la OBC NanoMind A3200 desarrollada por GomSpace, una empresa de sistemas espaciales ubicada en Aalborg, Dinamarca. La tarjeta se basa en el microcontrolador AT32UC3C de la familia AVR32 de microcontroladores Atmel, sin embargo, por la situación de salud pública durante la pandemia de COVID-19 en el periodo 2020-2021, no es posible realizar pruebas presenciales con el acompañamiento del laboratorio en la tarjeta ni acceder a equipos de evaluación para microcontroladores con la misma arquitectura. Por esa razón, para el segmento final de esta propuesta es necesario ofrecer alternativas que demuestren esta capacidad.

Para ello, en este capitulo se presenta el contexto del software relacionado al proyecto, hardware equivalente y sus atributos de memoria; se exponen diferentes herramientas como IDEs compatibles con el compilador GCC para AVR y módulos de software para la arquitectura AVR de distribución abierta,

4.1. Consideraciones de Software

GomSpace incluye dentro de sus productos los diferentes BSPs y SDKs para las tarjetas, El BSP de la A3200 se incluye con la compra del producto y el SDK se adquiere por aparte. Este último está formado por diferentes bibliotecas y módulos de software documentados extensamente para ayudar al desarrollador con el flujo de trabajo, diseño de aplicaciones y

depuración del compilador incluido. Algunos de los módulos adjuntos libres y propietarios corresponden al CubeSat Space Protocol, un protocolo para transferencia de archivos, una consola para pruebas y una biblioteca para housekeeping de telemetría. El protocolo de red (CSP) y la biblioteca de housekeeping utilizan tareas definidas mediante FreeRTOS, sin embargo, solo el protocolo se distribuye libremente.

4.1.1. CubeSat Space Protocol

Inicialmente pensado como un protocolo espacial de red abierto para puerto CAN [14], El CSP (CubeSat Space Protocol) fue desarrollado en la universidad de Aalborg y el equipo principal pronto se integraría a GomSpace, por consiguiente se incluye dentro de los módulos de software distribuidos por GomSpace y desde entonces se han ampliado sus funciones a otras interfaces como I2C y RS232 [26].

El protocolo sigue el modelo TCP/IP (capas de Aplicación, Transporte, Internet y Acceso a la Red tal como se muestra la figura 4.1) con un encabezado de 32 bits con la información de las dos capas intermedias [21]. Se diseñó y probó para dispositivos AVR8 y sistemas de 32 bits tanto de Atmel como de ARM (específicamente para la arquitectura ARM7) [20], aunque no está limitado a esas opciones. Finalmente, está escrito en C y es funcional en sistemas con POSIX, Windows, MacOS y FreeRTOS.

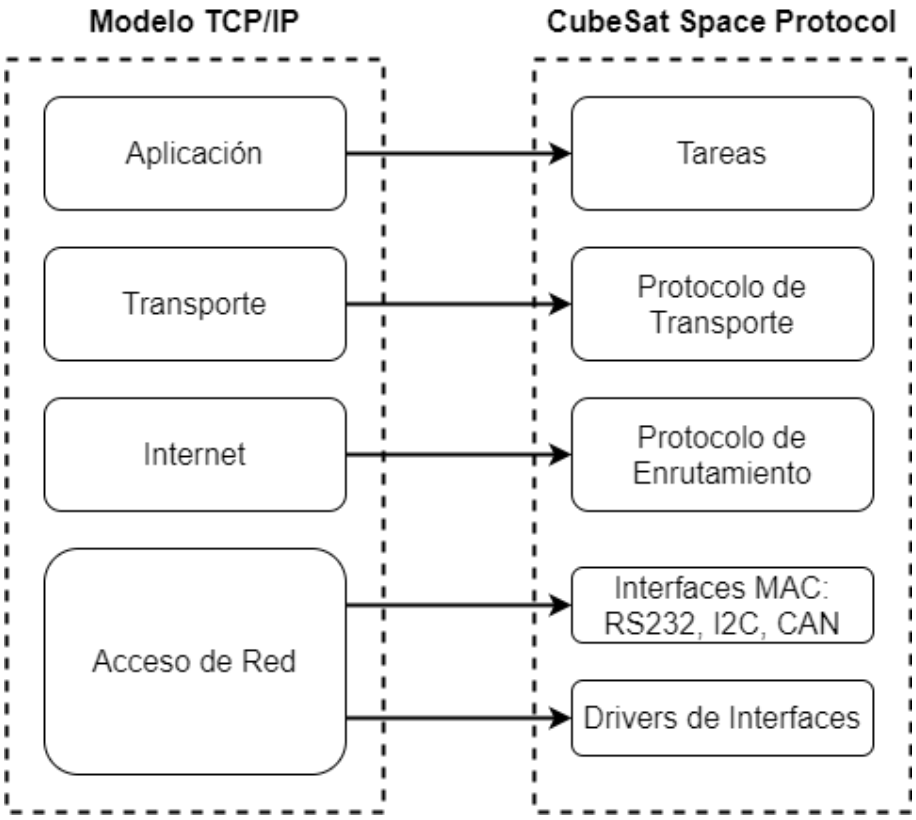

Figura 4.1: CSP respecto al modelo TCP/IP

4.1.2. Compiladores e IDEs

Existen extensas herramientas y entornos donde crear aplicaciones para plataformas AVR: software libre o propietario. Para los productos de GomSpace, se incluye un toolchain específico para el sistema junto a Waf scripts de python para automatizar y una guía de preparación de proyectos en el IDE Eclipse. En una visión más amplia, la suite WinAVR incluye un depurador, simulador y el toolchain abierto de GCC para AVR, sin embargo, solo soporta dispositivos de 8 bits y algunos productos heredados de Atmel.

El toolchain completo se puede adquirir desde el sitio web de Microchip y contiene las herramientas mínimas para desarrollar aplicaciones en todas las familias [29]. También se incluye dentro de los IDEs Microchip Studio (software gratuito antes conocido como Atmel Studio).

El IDE Microchip Studio es una estación completa de desarrollo con capacidades de visualizar memoria, instalar drivers y un catálogo entero de ejemplos para distintas aplicaciones. Aunque tiene un simulador no es confiable pues ciertas familias como la de 32 bits (AVR y SAM) tienen sets de instrucciones incompletos y en general los tiempos de ejecución son muy lentos [27], por lo que no se considera como una característica funcional.

Los demos de FreeRTOS hacen mención del toolchain AVR GCC ya descrito y el IAR workbench [34], el cual si ofrece un simulador completo para múltiples arquitecturas y contiene su propio compilador. No obstante, las ventajas de IAR no se ofrecen de manera gratuita, al contrario, las licencias son altamente costosas y solo están disponibles mediante cotización.

Considerando todo lo anterior, se evalúan los criterios de costo (gratuito o pago), herramientas integradas de depuración, compatible con las arquitecturas de 8 bits y 32 bits AVR y uso de GCC como compilador. La matriz Pugh de la Tabla 4.1 muestra como Microchip Studio es el ambiente mas conveniente para probar la integración de los agentes en las arquitecturas objetivo al cumplir con todos los criterios.

Tabla 4.1: Matriz Pugh sobre ambientes de desarrollo

	Opciones		
Criterios	WinAVR	Microchip Studio	IAR Workbench
Costo	+	+	-
Herramientas incluidas	0	0	+
Arquitecturas AVR8/AVR32	-	+	+
AVR GCC	+	+	-
	1	3	0

4.2. Visión General de las Familias AVR8 y AVR32

Las familias de microcontroladores AVR fueron desarrolladas y distribuidas por Atmel hasta ser adquiridos por Microchip en 2016 [8]. Altamente conocidos por el ATmega328 sobre el cual se construyó el Arduino Uno. Las principales familias (megaAVR, tinyAVR y XMEGA) [30] corresponden a microcontroladores RISC de 8 bits con arquitectura Harvard [32], sin embargo, Atmel también presentó una familia de 32 bits (AVR32) en el 2006. Para el 2016 dicha familia pasaría a ser un sistema heredado y se reemplazó por microcontroladores con procesadores ARM.

Para demostrar la portabilidad de la biblioteca, se construirán dos soluciones para los puertos de FreeRTOS de AVR8 y AVR32 en dos tarjetas distintas, apoyándose en los ejemplos distribuidos por AWS junto a los archivos fuentes de FreeRTOS y los ejemplos de evaluación en Microchip Studio exclusivamente para aplicaciones de RTOS.

Los demos que AWS ofrece en la versión 10.4.1 de FreeRTOS compatibles con GCC pueden o no tener un proyecto de algún IDE asociado. La Tabla 4.2 resume la disponibilidad según microcontroladores (y su familia si existe), equipos de evaluación e IDEs.

Tabla 4.2: Demos para dispositivos AVR incluidos con FreeRTOS

Microcontrolador	Equipos	IDE
ATmega323	-	WinAVR, IAR
ATmega328PB	Xplained Mini	-
ATmega4809 (megaAVR)	Curiosity Nano	Microchip Studio, IAR, MPLAB.X
AVR128DA48 (AVR-Dx)	Curiosity Nano	Microchip Studio, IAR, MPLAB.X
AT32UC3A (AVR32)	EVK1100, EVK1104, EVK1105	-
AT32UC3B (AVR32)	EVK1101	-

Respecto al IDE, los ejemplos se distribuyen mediante la biblioteca por defecto del software (ASF) [28] y la extensión Atmel START. Las aplicaciones de RTOS con AVR32 se presentan para los equipos EVK1100, EVK1101, EVK1104, EVK1105, los microcontroladores UC3C y UC3L. Para 8 bits solo existe un ejemplo en Atmel START para ATmega4809 con el equipo Xplained Pro. Sin embargo, la versión de FreeRTOS que se adjuntan no es la misma entre las arquitecturas. Los ejemplos de AVR32 del IDE tienen FreeRTOS 7.0.0 (distribuido durante 2011) y el ejemplo para AVR8 trae FreeRTOS 10.0.0. La versión es de vital importancia pues justo en los cambios de 7.3.0 se introducen una función necesaria para esta biblioteca: eTaskStateGet, luego renombrada a eTaskGetState en la versión 7.4.0. [3] Además, en la versión 8.0.0 se cambió el tipo de dato utilizado en cadenas de caracteres y la notación de los nombres para las definiciones de datos (typedef).

Con el fin de referenciar de manera cruzada los puertos y lograr una aplicación en la versión más reciente, se eligen una tarjeta por arquitectura que tienen demostraciones en ambos medios de distribución. Las tarjetas que cumplen el requisito son AT32UC3A (equipo EVK1100) para AV32 y ATmega4809 (Xplained Pro).

4.2.1. AVR8: ATmega4809

Es parte de la familia megaAVR Series-0, diseñada para aplicaciones de tiempo real con necesidades deterministas al igual que la familia tinyAVR, pero con un incremento de memoria para programa hasta 48 kB totales. Las parámetros completos de memoria del ATmega4809, el microcontrolador de más tamaño en la familia, se exponen en la Tabla 4.3.

Tabla 4.3: Parámetros de memoria de la ATmega4809

Parámetro	Valor
Tipo de Memoria del Programa	Flash
Tamaño de Memoria del Progama	48 kB
SRAM	6144 B
EEPROM	256 B

4.2.2. AVR32: AT32UC3A

La familia UC3A corresponde a la primera linea de microcontroladores con el procesador AVR32 de Atmel. Pensado para programas con alta densidad de código y alto desempeño con bajo consumo de potencia para aplicaciones embebidas sensibles al costo. Cuenta con una memoria Flash para el programa y SRAM para acceso rápido, además ciertos modelos permiten proveer memoria adicional mediante un bus externo. Los detalles de la familia respecto a la disponibilidad de memoria se muestran en la Tabla 4.4.

Tabla 4.4: Parámetros de memoria de la familia AT32UC3A

Dispositivo	Tamaño de Memoria del Progama	SRAM	Interfaz de Bus Externo
AT32UC3A0512	512	64	Sí
AT32UC3A0256	256	64	Sí
AT32UC3A0128	128	32	Sí
AT32UC3A1512	512	64	No
AT32UC3A1256	256	64	No
AT32UC3A1128	128	32	No

Precisamente el microcontrolador empleado en la NanoMind A3200 corresponde a la familia UC3C que incorpora elementos no existentes en la UC3A como una FPU, interfaces

CAN y USB, así como ampliar el numero de buses periféricos.

4.3. Cambios para la Portabilidad de FreeMAES

Acerca de la biblioteca diseñada en el capítulo 2, existen conflictos entre su diseño y las capacidades de los dispositivos embebidos. La clase sysVars (subsección 2.4.2) funciona como una cápsula para una variable tipo mapa: una plantilla parte de la STL y utiliza asignación dinámica de memoria. El mapa contiene el Task Handle y un puntero al agente que realiza dicha tarea.

Aunque la herramienta Intellisense del IDE reconoce las clases miembro de la STL, el compilador no es capaz de incorporar las plantillas. Para reducir dependencias de bibliotecas externas se opta por no utilizar componentes ajenos y cambiar el mapa por un arreglo estático de un registro apodado sysVar, que contiene las mismas dos variables que las plantillas ya mencionadas (listado 4.1).

```
typedef struct{
    Agent_AID first;
    Agent * second;
    }sysVar;

class sysVars{
public:
    Agent* get_TaskEnv(Agent_AID aid);
    void set_TaskEnv(Agent_AID aid, Agent* agent_ptr);
    void erase_TaskEnv(Agent_AID aid);
    sysVar * getEnv();

private:
    sysVar environment[AGENT_LIST_SIZE];
};
```

Listado 4.1: Clase SysVars para puertos embebidos

4.4. Desempeño de Memoria de FreeMAES

Para evaluar el incremento de memoria principal al incluir la biblioteca FreeMAES y desarrollar el programa bajo se propone desarrollar una aplicación simple a partir de drivers en los ejemplos suministrados para ambas arquitecturas y FreeRTOS versión 10.4.1.

Para AVR8, se desarrolla programa sencillo que enciende cinco LEDs de forma secuencial. Al completar la serie, el contador vuelve al primer LED y repite el procedimiento apagando

los LEDS en el mismo orden.

La aplicación escogida para AVR32 corresponde a dos LEDs intermitentes, controlados cada uno por una tarea distinta con la misma prioridad que se turnan el uso del procesador.

4.4.1. Aplicación de demostración para AVR8: LED Sequence

Sin Agentes

Se ejecutan dos tareas distintas, el flujo de la tarea generador tiene mayor prioridad (figura 4.2a) y modifica un valor iniciado en 0 que incrementa con cada ejecución del lazo, si el valor es igual a 5 se redefine a 0; luego el valor se coloca en una cola. La segunda tarea encargada de controlar los LEDs toma el número en la cola y alterna el estado empleando el valor como índice de un registro que almacena las definiciones de hardware.

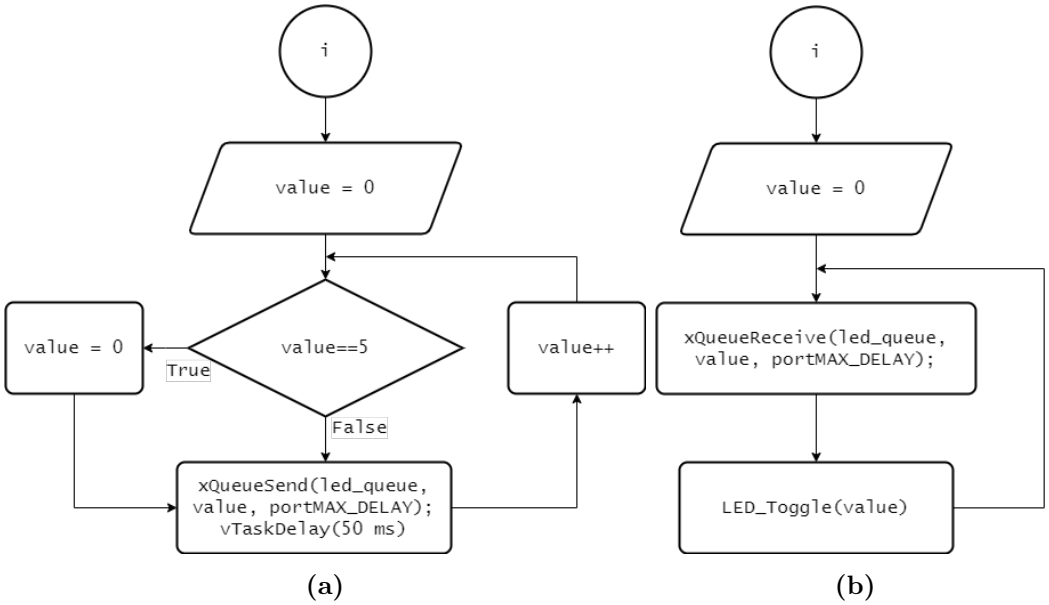

Figura 4.2: Diagrama de flujo para función generadora (a) y función de control del LED (b)

Con Agentes

En el caso de implementación mediante multiagentes, se utilizan siempre un número de agentes según el número de tareas (figura 4.3).

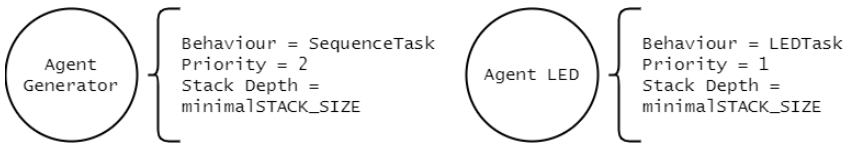

Figura 4.3: Agentes para aplicación LED Sequence

Cada agente se crea con una propia cola necesaria para la mensajería. Los mensajes son de tipo cadena de caracteres, por lo que se debe trasladar el valor a cadena antes de enviarse y retraducirse al ser recibida en la otra tarea. Las instrucciones en los lazos de las funciones ahora se escriben dentro del método acción de la clase de comportamiento. Se agrega una sola instrucción en el método de ajustes del comportamiento del agente Generador donde se agrega al agente LED como receptor.

Las modificaciones del flujo para multiagentes se muestran en los diagramas de la figura 4.4

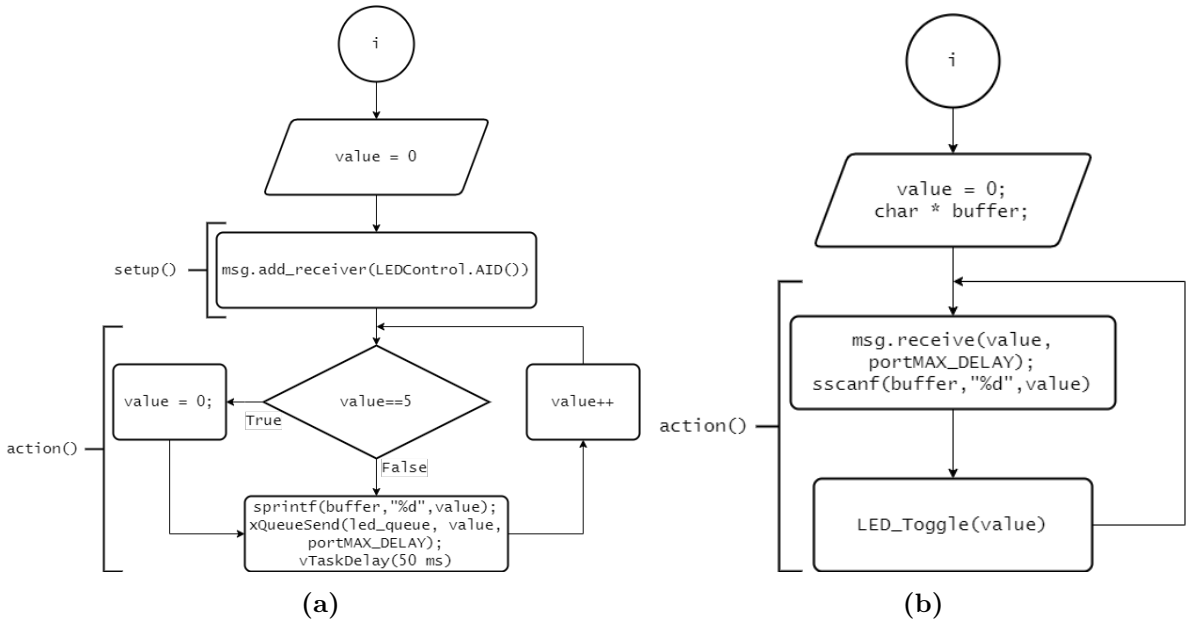

Figura 4.4: Diagrama de flujo para comportamiento del agente generador (a) y comportamiento del agente controlador del LED (b)

4.4.2. Aplicación de demostración para AVR32: LED Flash

Sin Agentes

La implementación sin agentes consiste en un método de creación de tareas (vStartLEDFlashTasks) con asignación de GPIO mediante una variable global (uxFlashTaskNumber), además de los métodos de soporte para iniciar las interfaces de prueba y alternar los estados de los LEDs. Se puede apreciar en la figura 4.5 que la creación de tareas está encapsulada en un lazo for hasta que el iterador sea igual al número de LEDs definido (constante ledNUMBER_OF_LEDS), en este caso 2. Se definen todas las tareas con la misma prioridad y tamaño de stack, un nombre LEDx donde x es el valor iterado (xLEDTask) más uno y la función de tarea vLEDFlashTask. La función no toma ningún argumento extra, es decir que no hace uso de los parámetros de función pvParameters.

La función de tarea se muestra en la figura 4.6 durante la ejecución llama a las definiciones de la API portENTER_CRITICAL y portEXIT_CRITICAL para detener el servicio de

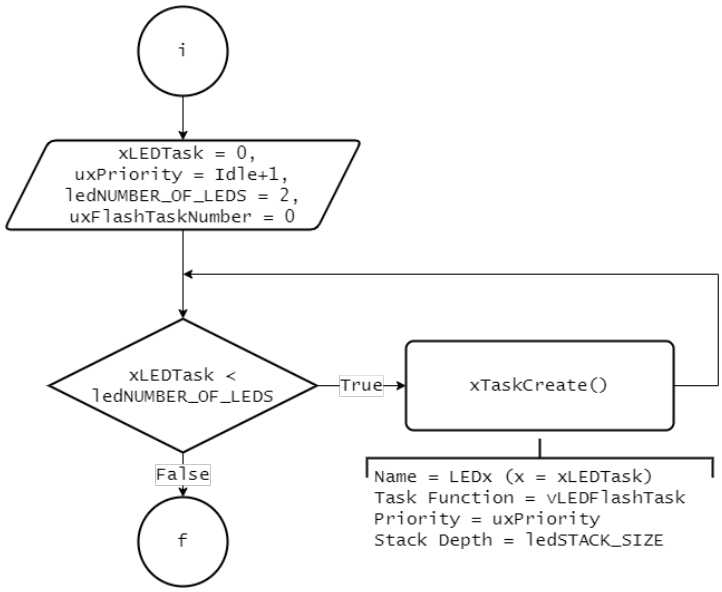

Figura 4.5: Diagrama de flujo para función vStartLEDFlashTasks

interrupciones mientras guarda el valor de uxFlashTaskNumber en uxLED,incrementar en uno y reanudar la ejecución normal. Toma las constantes xFlashRate y xFlashTime y calcula los periodos de retraso entre parpadeos de los LED. Justamente el LED se decide con el valor almacenado en uxLED y el incremento se realiza para que la siguiente tarea revise el valor y utilice un LED disponible. La intermitencia de ambos LEDs se alcanza en un lazo infinito donde el estado de bloqueo, durante los retrasos de las tareas, permite alternar entre ellas.

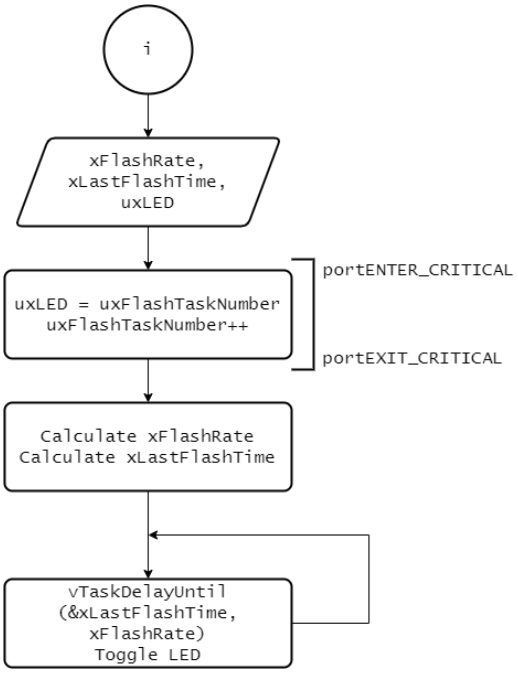

Figura 4.6: Función de tarea vLEDFlashTask

Con Agentes

Por el otro lado, la implementación con agentes requiere seguir la estructura de solución descrita en la sección 3.3. Actualmente solo hay dos tareas, por lo tanto solo dos agentes se deben construir e inicializar, pero si se necesitarán un mayor número de agentes sería necesario implementar una estructura de datos que reduzca la repetición de código. Cada agente encapsula el nombre de la tarea, la función que encierra la clase comportamiento, el parámetro con el LED respectivo, el tamaño del stack y la prioridad (figura 4.7).

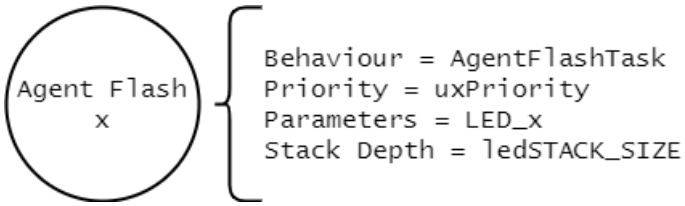

Figura 4.7: Agente genérico de la aplicación de LEDs intermitentes

Al inicializar el agente, se debe pasar el parámetro de LED como parámetro de función a diferencia de utilizar una variable global que se debe modificar deteniendo el servicio de interrupciones.

La función de tarea (figura 4.8) no cambia conceptualmente, pero su flujo debe ser segmentado entre los métodos de ajustes y acción, siendo este último quien encierra el lazo infinito. Las variables necesarias dentro del método se incluyen como variables miembros de la clase comportamiento.

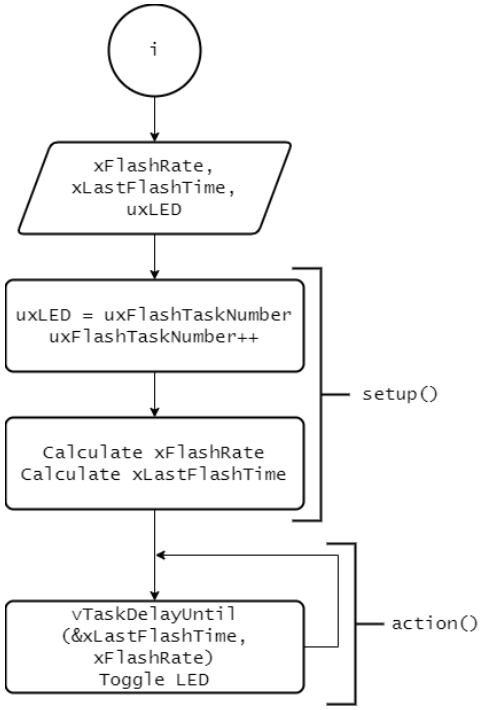

Figura 4.8: Función de tarea de los agentes genéricos

4.4.3. Resultados sobre el consumo de Memoria

El IDE Microchip Studio comparte información sobre el resultado de cada construcción luego de una compilación exitosa. Según los valores impresos en la consola se determinan el total de bytes y total de memoria disponible en el sistema objetivo. el archivo ELF descompone los datos en diferentes secciones; para AVR se consideran text, data y bss.

La memoria del programa corresponde a la memoria Flash del microcontrolador, es decir la parte de la memoria donde se almacenan las instrucciones (segmento text del ELF). La memoria de datos representa los valores constantes y variables del sistema, quienes ocupan el espacio de la SRAM. Ambos programas utilizan heap 3 como administrador de memoria en FreeRTOS.

Memoria de programa en ATmega4809

Como se puede apreciar en la Tabla 4.5, hay un incremento en ambos tipos de memoria. La inclusión de la API (funciones de plataforma, colas de mensajería, tarea de administrador, etc) y funciones de soporte (sscanf, sprintf) aumenta en 4678 B el espacio utilizado por la memoria de programa y 452 B el espacio de datos por sobre la aplicación sin agentes.

Respecto a los valores de la Tabla 4.3 la tarjeta ATmega4809 cuenta con 48 kB totales de memoria Flash y 6144 B de SRAM por lo que el uso de memoria Flash aumenta en un 9.0 % y la RAM en 7.3 %.

Tabla 4.5: Uso de memoria para la aplicación LED Sequence en ATmega4809

Parámetro	Sin Agentes	Con Agentes	Incremento
Memoria de Programa (B)	9918	14596	4678
Memoria Flash Utilizada	20.2 %	29.2 %	9.0 %
Memoria de Datos (B)	115	567	452
Memoria SRAM Utilizada	1.9 %	9.2 %	7.3 %

Memoria de programa en AT32UCA30512

Según la Tabla 4.6 el incremento porcentual de memoria Flash y SRAM es menor para ambos, pues cuenta con más memoria y el espacio requerido por la API FreeMAES varia en 578 B respecto a la aplicación en AVR8, manteniéndose dentro de un margen considerable.

Del total de 512 kB para programa disponibles en el sistema y 64 kB para datos, la aplicación con multiagente incrementa el uso de memoria Flash en un 1 % y la SRAM en un 1.3 %.

El incremento de memoria SRAM en ambos casos se debe a que los agentes se comportan como objetos estáticos con registros de miembros, punteros a funciones, parámetros

Tabla 4.6: Uso de memoria para la aplicación LED Flash en AT32UC3A0512

Parámetro	Sin Agentes	Con Agentes	Incremento
Memoria de Programa (B)	54708	59964	5256
Memoria Flash Utilizada	10.4 %	11.4 %	1.0 %
Memoria de Datos (B)	8244	9172	928
Memoria SRAM Utilizada	12.6 %	13.9 %	1.3 %

condicionales y valores duplicados ya incluidos en las tareas como el nombre y la prioridad. Marca una exigencia mayor sobre el sistema a diferencia de una aplicación pura de FreeRTOS.

4.4.4. Diferencia en Cantidad de Líneas de Código

Como una métrica adicional se puede considerar el cambio en el total de lineas de código entre las implementaciones. En la Tabla 4.7 se presentan la cantidad líneas de código correspondientes a cada aplicación separada por implementación con agentes y sin agentes. La métrica recuenta que los programas con gentes o sin ellos no exceden 100 lineas de código al ser aplicaciones relativamente simples. La diferencia porcentual para la aplicación de AVR8 señala un incremento de un 20 % aproximadamente, en AVR32 el incremento es menor al 5 %.

Tabla 4.7: Líneas de código de cada implementación

	Líneas de Código		
Arquitectura	Sin Agentes	Con Agentes	Diferencia
AVR8	71	85	19.71 %
AVR32	42	44	4.76 %

4.5. Prueba de Concepto: CSP con multiagentes

Para poder incorporar módulos de software externos es necesario primero comprender que funciones de la API dependen directamente del kernel y el contexto del uso. En el caso del CSP, su API desarrolla funciones relevantes para administrar puertos, activar interfaces y enrutar paquetes. Las funciones que incorporan elementos de la arquitectura sea POSIX, FreeRTOS, MacOS o Windows corresponden a creación de hilos (tareas), servicios de comunicación y sincronización entre hilos; es decir colas, multiplexadores y semáforos. El kernel finalmente es relevante únicamente para conseguir que otras tareas accedan a protocolo.

Según la arquitectura propuesta por Carvajal-Godínez [10, p 69-71] para sistemas multiagentes, el agente que encapsula las funciones de tarea relacionadas con la comunicación

a fuera de la plataforma se le llama agente ACC y requiere tres elementos: Una interfaz de comunicación, un diccionario y un proceso.

La aplicación se encierra dentro de un comportamiento cíclico el proceso, así habilita el acceso de la plataforma a las funciones de la interfaz de comunicación. El Diccionario corresponde a la herramienta encargada de traducir el lenguaje de los agentes (ACL) al tipo de mensajes de la interfaz. PDOs en el caso de CAN, por ejemplo. Finalmente, las interfaces del CSP pueden ser CAN, RS232 o I2C. Los elementos se ilustran en la figura 4.9

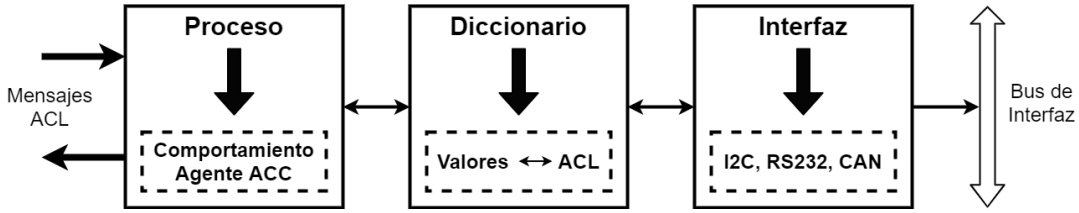

Figura 4.9: Elementos del ACC

Mediante FreeMAES es posible implementar agentes ACC, para CAN o cualquier otra interfaz con sus diccionarios respectivos como se muestra en la figura 4.10, junto a un agente jerarca administrando la comunicación entre los ACC y el resto de la plataforma.

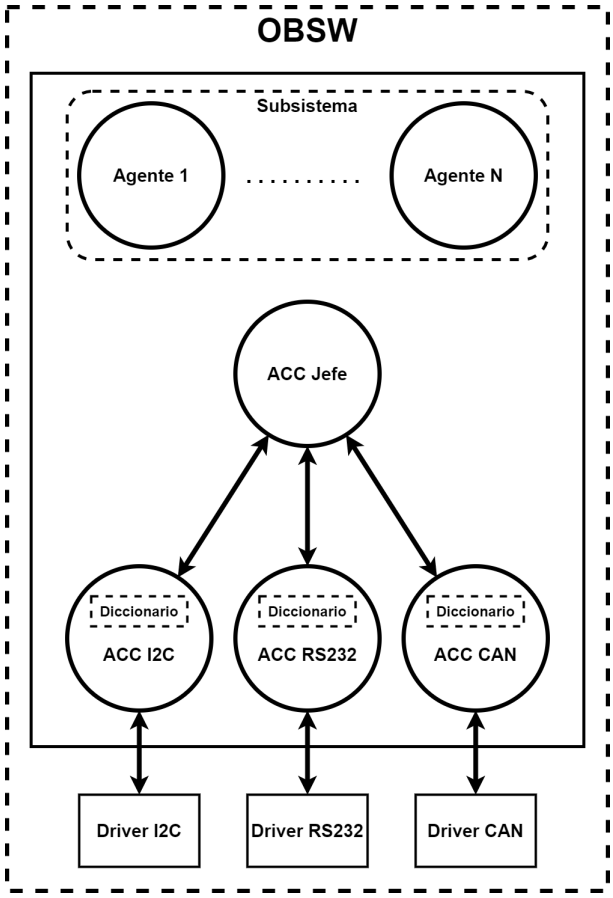

Figura 4.10: Arquitectura propuesta para administrar CSP con multiagentes

4.6. Recomendaciones y Requisitos de Desarrollo

Como se discutió en la sección 4.2, puede suceder que proyectos con más de un año o distribuciones de software con una única versión posean una distribución de FreeRTOS obsoleta, en especial versiones anteriores a 8.0.0 que carecen de funciones fundamentales para control de tareas y cuentan con una nomenclatura de definiciones diferente a la actual. FreeRTOS aun soporta los puertos para dichos dispositivos, por lo que se recomienda al desarrollados emplear esta biblioteca junto a FreeRTOS 10.0.0 en adelante o 8.0.0 como mínimo.

En el archivo de configuración de FreeRTOS se deben activar las funciones opcionales utilizadas en la biblioteca: eGetTaskState, vTaskPrioritySet, uxTaskPriorityGet, vTaskDelete, vTaskDelay, vTaskSuspend y xTaskGetCurrentTaskHandle.

El desarrollador debe considerar que el dispositivo objetivo debe contar con al menos 5 kB de memoria de programa disponible para emplear esta biblioteca.

Por último, la biblioteca contiene funciones definidas virtualmente, por lo que si el compilador no provee las bibliotecas necesarias para el servicio se obtendrá la excepción y no se realizará el enlazado de los objetos. Para corregir este error se debe definir una función que administre el error virtual como se muestra en el listado 4.2.

```
extern "C" {
    void __cxa_pure_virtual() {
        while (1);
    }
}
```

Listado 4.2: Función para manejo del error virtual

El resumen de los requisitos se presenta en la Tabla 4.8.

Tabla 4.8: Requisitos de biblioteca FreeMAES

Versión Kernel	10.0.0 (Recomendada) 8.0.0 (Mínima)
Espacio en Memoria Mínimo	5kB
Funciones Necesarias	eGetTaskState vTaskPrioritySet uxTaskPriorityGet vTaskDelete vTaskDelay vTaskSuspend xTaskGetCurrentTaskHandle

4.7. Observaciones Finales

El incremento de uso de memoria fue exitosamente determinado mediante compilaciones exitosas para las arquitecturas AVR8 y AVR32. Los incrementos correspondían a menos del 10 % de un total de 48 kB y aproximadamente 1 % de 512 kB respectivamente, utilizando un máximo de 5.2 kB en AVR32.

La inclusión de FreeMAES no parece una reducir inmediatamente el código pues agregar las clases para comportamientos de cada agente y la definición de sus métodos establece un mínimo de lineas necesario ineludibles, sin embargo, incluso con esta compensación obligatoria el incremento no alcanza el 50 %. Las aplicaciones cumplen su fin al contribuir a la determinación del incremento de memoria, pero su diseño no aprovechan la totalidad de las capacidades comunicativas de los sistemas multiagentes, dificultando determinar si existe una reducción en la complejidad.

Capítulo 5

Conclusiones y Trabajo Futuro

5.1. Conclusiones

Este trabajo parte de la propuesta MAES y formaliza su distribución abierta multiplataforma a través de la API de la biblioteca FreeRTOS al ser compatible con el microkernel FreeRTOS. Al inicio de esta investigación se desarrolló un puerto para simulaciones en MSVC con MinGW mediante las clases del Framework MAES y basándose en la implementación anterior para TI-RTOS.

El puerto MSVC es completamente funcional, sin embargo, el componente para construir variables de entorno estaba austente del kernel y la clase sysVars que implementa la característica hace uso de plantillas de la biblioteca STL no disponible en sistemas embebidos. Por ello se modificaron las funciones y estructuras de software para operar dentro de sistemas embebidos, manteniéndolo ligero aunque restringido sus funciones a asignaciones estáticas de memoria.

Se determina el uso de memoria de programa y datos a partir dos aplicaciones distintas en dos arquitecturas de AVR respectivamente con controladores distintos. Se observa un incremento desde 4.6 kB a 5.2 kB en el uso de Memoria Flash, el incremento de SRAM no se considera pues es dependiente principalmente de la aplicación y no de la biblioteca.

Ambas arquitecturas son compatibles con el CubeSat Space Protocol y se desarrolla una propuesta conceptual para la inclusión de un Agente ACC que administre la comunicación entre las interfaces y la aplicación mediante este protocolo. De esta compatibilidad se infiere su posible usabilidad en la tarjeta NanoMind A3200 y los modulos de software del SDK de GomSpace.

Finalmente se establecen requisitos de software de la biblioteca FreeRTOS acerca de las versiones del kernel FreeRTOS recomendadas y las funciones opcionales a incluir en el archivo de configuración.

5.2. Trabajo Futuro

Al igual que el Framework MAES, esta biblioteca no implementa totalmente los actos comunicativos descritos por el estándar de FIPA, aun así contiene los mecanismos para permitir al desarrollador construir sus propios métodos dentro de la plataforma multi-agente e integrarlos a la API en versiones posteriores. Adicionalmente, para complacer el modelo FIPA aun es necesario implementar el DF, que a su vez puede ser derivado de la clase sysVars incluida en este proyecto.

La situación sanitaria de 2020-2021 ocasionada por el COVID-19 restringió el acceso al equipo físico y herramientas de software de GomSpace en el Laboratorio de Sistemas Espaciales del ITCR, por lo que se propone realizar pruebas con la NanoMind en el futuro para verificar su funcionamiento y evaluar su desempeño.

Aunque las aplicaciones desarrolladas en este informe no demuestran un incremento o disminución considerable del tamaño del código para aplicaciones con el paradigma MAES, se espera que aplicaciones de mayor complejidad en FreeRTOS con altas tazas de mensajería puedan reducirse considerablemente mediante la implementación FreeMAES.

FreeRTOS ofrece puertos para las arquitecturas de diferentes socios además de Microchip como Texas Instruments, NXP y Cadence. Este trabajo solo utiliza los compiladores de GCC para AVR (8 bits) y AVR32, así pues existe la oportunidad de probar multiagentes con la biblioteca FreeMAES en diferentes microcontroladores y confirmar sus capacidades multiplataforma.

Bibliografía

[1] ACAE, *Proyecto Irazú — ACAE*. dirección: https://irazu.acae-ca.org.

[2] AWS, *FreeRTOS Kernel Developer Docs*. dirección: https://www.freertos.org/features.html.

[3] ——, *FreeRTOS Version History*. dirección: https://www.freertos.org/History.txt.

[4] ——, *Tasks*. dirección: https://www.freertos.org/RTOS-task-states.html.

[5] ——, *The FreeRTOS™ Kernel*. dirección: https://www.freertos.org/RTOS.html.

[6] R. Barry, *Mastering the FreeRTOS™ Real Time Kernel*, 161204.ª ed., Real Time Engineers, dic. de 2016.

[7] F. Bellifemine, A. Poggi y G. Rimassa, «Developing multi-agent systems with JADE,» en *International Workshop on Agent Theories, Architectures, and Languages*, Springer, 2000, págs. 89-103.

[8] J. E. Bjornholt y S. Skaggs, «Microchip to Acquire Atmel,» dirección: https://www.microchip.com/pdf/MCHP_to_Acquire_Atmel.pdf.

[9] J. Bouwmeester, S. Radu, M. S. Uludag, N. Chronas, S. Speretta, A. Menicucci y E. K. A. Gill, «Utility and constraints of PocketQubes,» *CEAS Space Journal*, vol. 12, n.º 4, págs. 573-586, 2020. DOI: 10.1007/s12567-020-00300-0.

[10] J. Carvajal-Godínez, «Agent-Based Architectures Supporting Fault-Tolerance in Small Satellites,» Tesis doct., Delft University of Technology, 2021.

[11] C. Chan-Zheng, *MAES Application Programming Interface*, 1.ª ed., jul. de 2017.

[12] ——, «MAES: A Multi-Agent Systems Framework for Embedded Systems,» Tesis de mtría., Delft University of Technology, Delft, South Holland, 2017.

[13] C. Chan-Zheng y J. Carvajal-Godínez, «A Multi-Agent System Framework for Miniaturized Satellite,» *Revista Tecnología en Marcha*, vol. 32, n.º 1, págs. 54-67, ene. de 2019. DOI: 10.18845/tm.v32i1.4118.

[14] J. De Claville Christiansen, «The Cubesat Space Protocol,» dirección: https://github.com/libcsp/libcsp.

[15] eoPortal, *eoPortal - Earth Observation Directory & News*. dirección: https://directory.eoportal.org/web/eoportal/home.

[16] FIPA, «FIPA Agent Management Specification.»

[17] ——, «FIPA Communicative Act Library Specification,» dirección: https://github.com/libcsp/libcsp.

[18] ——, *Welcome to FIPA!* Dirección: http://www.fipa.org.

[19] I. Global, *What is Multi-Agent System*. dirección: https://www.igi-global.com/dictionary/using-multi-agent-systems-support/19363.

[20] GomSpace, «Product Development Platform,» dirección: https://www.slideshare.net/InfinITnetvaerk/udviklingsplatform-og-programmeringssprog.

[21] D. E. Holmstrøm, «Software and software architecture for a student satellite,» Tesis de mtría., Norwegian University of Science y Technology, Trondheim, Norway, 2012.

[22] T. Instruments, *Getting Started with TI-RTOS: Chapter 7—using tasks*. dirección: https://training.ti.com/getting-started-ti-rtos-chapter-7-using-tasks.

[23] ——, *RTOS: Real-Time Operating System (RTOS) for Microcontrollers (MCU)*. dirección: http://www.ti.com/tool/TI-RTOS-MCU.

[24] KERA, *On-board Computer System for mission critical space applications*, DS 1006901, Rev. 1.16, GomSpace, dic. de 2019.

[25] G. Krebs, *Dellingr (RBLE)*. dirección: https://space.skyrocket.de/doc_sdat/dellingr.htm.

[26] KubOS, *CSP: History*. dirección: https://docs.kubos.com/1.2.0/apis/libcsp/csp_docs/history.html.

[27] Microchip, *AVR Simulator*, 1.ª ed., 2020.

[28] ——, *Advanced Software Framework (ASF)*. dirección: https://www.microchip.com/en-us/development-tools-tools-and-software/libraries-code-examples-and-more/advanced-software-framework-for-sam-devices.

[29] ——, *AVR and Arm Toolchains (C Compilers)*. dirección: https://www.microchip.com/en-us/development-tools-tools-and-software/gcc-compilers-avr-and-arm.

[30] ——, *Microchip AVR MCUs*. dirección: https://www.microchip.com/design-centers/8-bit/avr-mcus.

[31] N. Prasad, *An overview of on-board computer (OBC) systems available on the global space marketplace*. dirección: https://blog.satsearch.co/2020-03-11-overview-of-on-board-computers-available-on-the-global-space-marketplace.

[32] H. Sharabaty, «AVR Microcontroller: History and Features,» dirección: https://www.researchgate.net/publication/320490097_AVR_Microcontrollers_History_And_Features.

[33] B. Shiotani, «Project Life-Cycle and Implementation for a Class of Small Satellites,» Tesis doct., University of Florida, 2018.

[34] I. Systems, *IAR Embedded Workbench*. dirección: https://www.iar.com/iar-embedded-workbench/.

[35] TEC, *Laboratorio de Sistemas Espaciales*, 2018. dirección: https://www.tec.ac.cr/unidades/laboratorio-sistemas-espaciales.

[36] A. S. University, *Phoenix CubeSat*. dirección: https://phxcubesat.asu.edu.

Apéndice A

API FreeMAES

Los archivos fuente de la API se pueden encontrar en el repositorio de FreeMAES.

A.1. Clase Agent

- Constructor de Agente: Crea la instancia de la clase. Requiere los parámetros de nombre, prioridad y tamaño de stack.
- Agent_AID AID(): Entrega el AID del agente.

A.2. Clase Agent Platform

- Constructor de Plataforma: Crea la instancia de la clase. Requiere el nombre de la plataforma.
- bool boot(): El método inicia la plataforma. Solo se puede llamar desde main().
- void agent_init(Agent* agent, void behaviour(void* pvParameters)): El método asigna el comportamiento encapsulado en la función a la tarea y construye la cola del agente. Inicia la tarea con prioridad 0 y suspende su ejecución. Requiere la función y un puntero al agente.
- bool agent_search(Agent_AID aid): El método busca al agente según su AID y regresa verdadero si lo encuentra.
- void agent_wait(TickType_t ticks): llama a la función vTaskDelay de FreeRTOS y bloquea al agente por el tiempo definido.
- void agent_yield(): El agente libera el procesador.
- Agent_AID get_running_agent(): Regresa el AID del agente que actualmente se ejecuta.

- AGENT_STATE get_state(Agent_AID aid): Regresa el estado del agente según su AID.
- Agent_info get_Agent_description(Agent_AID aid): Regresa el registro con información del agente según su AID.
- AP_Description get_AP_description(): Regresa la descripción de la plataforma.
- ERROR_CODE register_agent(Agent_AID aid): Función exclusiva del agente AMS. Asigna la prioridad adecuada a la tarea del agente y la activa. Incluye al agente en la plataforma.
- ERROR_CODE deregister_agent(Agent_AID aid): Función exclusiva del agente AMS. Asigna la prioridad 0 a la tarea del agente y la desactiva. Remueve al agente de la plataforma.
- ERROR_CODE kill_agent(Agent_AID aid): Función exclusiva del agente AMS. Elimina la tarea.
- ERROR_CODE suspend_agent(Agent_AID aid): Función exclusiva del agente AMS. Suspende al agente.
- ERROR_CODE resume_agent(Agent_AID aid): Función exclusiva del agente AMS. Reanuda al agente.
- void restart(Agent_AID aid): Función exclusiva del agente AMS. Elimina la tarea del agente y la cola para luego restaurarla con diferente AID, mismos parámetros.

A.3. Clase Agent Organization

- Constructor de Organización: Crea la instancia organización. Requiere el tipo de organización.
- ERROR_CODE create(): Se puede llamar desde un comportamiento. asigna a la organización su dueño (agente que llama la función) y la información del agente. La variable org apunta a la instancia de organización.
- ERROR_CODE destroy(): Vacía las listas de la organización y las variables de rol, afiliación y org de cada miembro.
- ERROR_CODE isMember(Agent_AID aid): Revisa si el agente es miembro de la organización.
- ERROR_CODE isBanned(Agent_AID aid): Revisa si el agente esta prohibido de la organización.

- ERROR_CODE change_owner(Agent_AID aid): Reasigna la posesión de la organización. Solo puede ser llamado por el dueño de la organización.
- ERROR_CODE set_admin(Agent_AID aid): Asigna la afiliación de administrador al agente. Solo puede ser llamado por el dueño de la organización.
- ERROR_CODE set_moderator(Agent_AID aid): Asigna el rol de moderador al agente. Solo puede ser llamado por el dueño de la organización.
- ERROR_CODE add_agent(Agent_AID aid): Agrega al agente a la organización y cambia su variable org a la organización de quien llama. Solo puede ser llamado por el dueño o el administrador de la organización.
- ERROR_CODE kick_agent(Agent_AID aid): Elimina al agente de la organización y cambia su variable org a NULL. Solo puede ser llamado por el dueño o el administrador de la organización.
- ERROR_CODE ban_agent(Agent_AID aid): Prohíbe al agente de la organización. Asigna el rol de moderador al agente. Solo puede ser llamado por el dueño o el administrador de la organización.
- ERROR_CODE remove_ban(Agent_AID aid): Elimina la prohibición en la organización sobre el agente. Solo puede ser llamado por el dueño o el administrador de la organización.
- void clear_ban_list(): Limpia la lista de prohibiciones de la organización.
- ERROR_CODE set_participant(Agent_AID aid): Agrega al agente a la conversación y cambia su variable org a la organización de quien llama. Solo puede ser llamado por el dueño o el moderador de la organización.
- ERROR_CODE set_visitor(Agent_AID aid): Agrega al agente a la conversación como oyente y cambia su variable org a la organización de quien llama. Solo puede ser llamado por el dueño o el moderador de la organización.
- ORG_TYPE get_org_type(): Regresa el tipo de organización.
- org_info get_info(): Regresa la información de la organización.
- UBaseType_t get_size(): Regresa el número de miembros de la organización.
- MSG_TYPE invite(Agent_Msg msg, UBaseType_t password, Agent_AID target_agent, UBaseType_t timeout): Envía una propuesta a otro agente para unirse a la organización.

A.4. Clase Agent Message

- Constructor de Mensaje: La instancia se crea dentro de la función encapsuladora.
- Mailbox_Handle get_mailbox(Agent_AID aid): Retorna el Mailbox Handle de la cola del agente según su AID.
- ERROR_CODE add_receiver(Agent_AID aid_receiver): Agrega el agente a la lista de receptores del agente que contiene el mensaje.
- ERROR_CODE remove_receiver(Agent_AID aid_receiver): Elimina al agente de la lista de receptores del agente que contiene el mensaje.
- void clear_all_receiver(): Vacía la lista de receptores.
- void refresh_list(): Recorre la lista y remueve receptores que ya no están registrados en la misma organización.
- bool isRegistered(Agent_AID aid): Revisa si el agente está registrado en la misma organización.
- void set_msg_type(MSG_TYPE type): Establece el tipo del mensaje.
- void set_msg_content(char* body): Establece el contenido del mensaje.
- MsgObj* get_msg(): Recupera el registro del mensaje.
- MSG_TYPE get_msg_type(): Recupera el tipo del mensaje.
- char* get_msg_content(): Establece el contenido del mensaje.
- Agent_AID get_sender(): Recupera el AID del emisor del mensaje.
- Agent_AID get_target_agent(): Recupera el AID del receptor del mensaje.
- MSG_TYPE receive(TickType_t timeout): Espera recibir un mensaje y bloquea la tarea por el periodo de timeut definido.
- ERROR_CODE send(Agent_AID aid_receiver, TickType_t timeout): Agrega un mensaje a la cola del agente objetivo en el periodo de timeout definido.
- ERROR_CODE send(): Envía mensajes a todos los receptores de la lista.
- ERROR_CODE registration(Agent_AID target_agent): Envía una solicitud de registro al agente AMS para el agente objetivo.
- ERROR_CODE deregistration(Agent_AID target_agent): Envía una solicitud de registro al agente AMS para el agente objetivo.
- ERROR_CODE suspend(Agent_AID target_agent): Envía una solicitud de suspensión al agente AMS para el agente objetivo.

- ERROR_CODE resume(Agent_AID target_agent): Envía una solicitud de reanudación al agente AMS para el agente objetivo.
- ERROR_CODE kill(Agent_AID target_agent): Envía una solicitud de terminación al agente AMS para el agente objetivo.
- ERROR_CODE restart(): Envía una solicitud de reinicio al agente AMS para el agente emisor.

A.5. Clase sysVars

- void set_TaskEnv(Agent_AID aid, Agent* agent_ptr): Agrega el par de AID y puntero de agente a las variables de entorno.
- Agent* get_TaskEnv(Agent_AID aid): Regresa el puntero al agente que contiene el AID.
- void erase_TaskEnv(Agent_AID aid): Borra el par de AID y puntero de agente de las variables de entorno.
- sysVar * getEnv(): Regresa el arreglo que contiene las variables de entorno.