

Manual de uso de la biblioteca CMAES

Desarrollado por:

Andrés de Jesús Conejo Boza

Enero, 2022

Resumen

En este manual, se analiza la biblioteca CMAES, la cual implementa el paradigma MAES o Multi-Agent Systems Framework for Embedded Systems en la computadora espacial NanoMind A3200. Sin embargo, esta biblioteca no se limita exclusivamente a ser implementada en dicha computadora espacial. Para poder analizar la biblioteca, se toma en cuenta la estructura de la misma, las clases existentes y la estructura que debe de tener una aplicación para utilizar la biblioteca correctamente. Además, para demostrar su uso, se muestra los resultados obtenidos en la terminal serial utilizando Microsoft Visual Studio. Por esta razón, también se detalla una guía de instalación de la biblioteca en un nuevo proyecto de Visual Studio.

Por otro lado, es importante tomar en cuenta que existen dos versiones de la biblioteca CMAES. La primera de ella hace referencia a una versión que utiliza directamente comandos del sistema operativo en tiempo real FreeRTOS. Por otro lado, existe una segunda versión que utiliza comandos de la biblioteca CSP o Cubesat Space Protocol, la cual toma el control del calendarizador del sistema operativo FreeRTOS.

Índice general

1. Implementación de la biblioteca CMAES	1
1.1. Diagrama de clases	2
1.2. Clase SysVars	3
1.3. Clase Agent	3
1.4. Clase User Defined Conditions	4
1.5. Clase Agent Message	4
1.6. Clase Agent Platform	6
1.7. Behaviours	8
1.7.1. Clase Cyclic Behaviour	9
1.7.2. Clase Oneshot Behaviour	10
1.8. Clase Agent Organization	10
1.9. Constructores	12
1.10. CMAES.h	12
2. Uso de CMAES	13
2.1. Estructura de aplicaciones que utilizan CMAES	13
2.1.1. Includes	13
2.1.2. Enums y structs	14
2.1.3. Definiciones de variables	14
2.1.4. Comportamientos	15
2.1.5. Main	18
2.2. Simulación utilizando Visual Studio	19
3. Instalación de CMAES en Visual Studio	20
Bibliografía	26

Capítulo 1

Implementación de la biblioteca CMAES

La meta principal de la biblioteca CMAES es poder generar una adaptación de la biblioteca FreeMAES hacia el lenguaje procedimental C. Esto, ya que la biblioteca FreeMAES fue desarrollada en el lenguaje orientado a objetos C++. Al no tener la definición de clases y objetos en C, fue indispensable encontrar un método fuera de lo convencional para poder crear una equivalencia de las clases en el lenguaje C.

Si bien es cierto y no se tiene la definición de clases en C, sí existe la definición de una estructura o struct. Por esta razón, no existe problema alguno con implementar los atributos de las clases. Sin embargo, el problema radica en implementar los métodos de las clases, ya que no es permitido por C el establecer una función dentro de un struct. Por ello, la forma encontrada para poder introducir una función dentro de un struct es el generar un puntero que, posteriormente, sea redirigido para que apunte hacia una función. Teniendo esta solución en cuenta, se implementaron las clases de dicha forma y se realizaron algunos cambios dentro de las funciones para que la biblioteca CMAES funcione adecuadamente.

1.1. Diagrama de clases

Con el propósito de poder analizar los atributos y métodos de cada clase implementada en CMAES, se puede analizar el diagrama de clases presentado en la figura 1.1 donde OBSW representa al software de la tarjeta utilizada. Además, para cada una de las clases implementadas, se analiza brevemente cada uno de sus atributos y métodos.

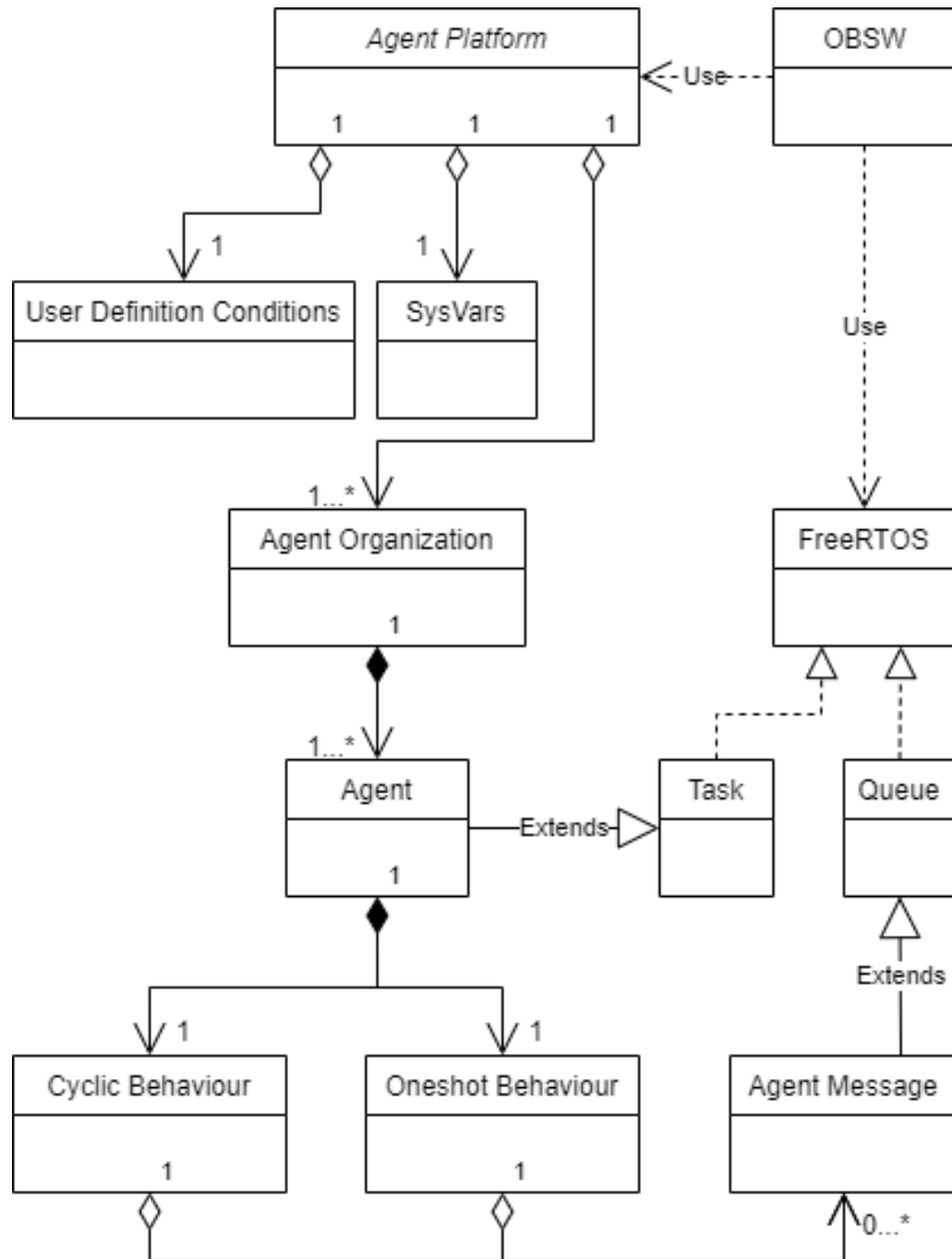


Figura 1.1: Diagrama de clases implementadas en CMAES.

1.2. Clase SysVars

La clase SysVars proviene de la adaptación del paradigma MAES utilizando FreeRTOS. Esto se debe a la falta de variables de entorno en dicho sistema operativo. Tomando esto en consideración, se implementó la clase SysVars con un único atributo llamando `environment`. Este atributo es de tipo struct y almacena los AID de cada agente que se encuentre registrado en dicho ambiente. Por otro lado, esta clase cuenta con los siguientes cuatro métodos:

- **Get Task Environment.** Este método se encarga de buscar un agente dentro del ambiente en base a su AID y regresa un puntero hacia dicho agente.
- **Set Task Environment.** Este método se encarga de registrar un agente dentro del ambiente.
- **Erase Task Environment.** Este método se encarga de eliminar a un agente que se encuentra registrado dentro del ambiente.
- **Get Environment.** Este método se encarga de devolver un puntero hacia el ambiente.

1.3. Clase Agent

La clase Agent hace referencia a la unidad fundamental de un sistema multiagentes. Esta clase engloba toda la información que debe de tener un agente dentro de un sistema multiagentes. Dentro de sus atributos se encuentran:

- **Agent.** Este atributo es un struct que contiene información esencial para un agente, como: AID, dirección de su buzón, nombre del agente, prioridad, plataforma a la que pertenece, organización a la que pertenece, tipo de afiliación en la organización y rol dentro de la organización.
- **Resources.** Este atributo es un struct que se encarga de almacenar información esencial para la tarea llevada a cabo por el agente, como: parámetros de entrada para su tarea, tamaño en memoria o heap destinado para el agente y un puntero hacia la función a ejecutar.

Por otro lado, esta clase también cuenta con un par de métodos, los cuales son:

- **Initializer.** Este método se encarga de asignar valores por defecto a los atributos de un agente recién creado.
- **AID.** Este método se encarga de devolver el AID de un agente.

1.4. Clase User Defined Conditions

La clase User Defined Conditions hace referencia a, como lo dice su nombre, condiciones que el usuario puede definir para habilitar o deshabilitar funciones que actúan dentro del AMS. En particular, esta clase no cuenta con atributos. Sin embargo, cuenta con los siguientes métodos:

- **Register Condition.** Este método habilita la posibilidad del AMS para registrar un agente en la plataforma.
- **Deregister Condition.** Este método habilita la posibilidad del AMS para eliminar un agente dentro de la lista de agentes registrados en la plataforma.
- **Suspend Condition.** Este método habilita la posibilidad del AMS para suspender un agente en la plataforma.
- **Kill Condition.** Este método habilita la posibilidad del AMS para eliminar un agente dentro de la lista de agentes registrados en la plataforma y también elimina su tarea registrada en el calendarizador de FreeRTOS.
- **Resume Condition.** Este método habilita la posibilidad del AMS para resumir la tarea realizada por un agente en la plataforma.
- **Restart Condition.** Este método habilita la posibilidad del AMS para eliminar y volver a crear la tarea de un agente en la plataforma.

1.5. Clase Agent Message

La clase Agent Message tiene un rol fundamental dentro del paradigma MAES, ya que es la clase que se encarga de comunicar un agente con otro. Esta clase no sólo se encarga de almacenar el contenido del mensaje, sino que también se encarga de enviarle al sistema operativo distintos comandos para poder administrar a los agentes. Los atributos contenidos en esta clase son:

- **Message.** Este atributo hace referencia a un struct que almacena información esencial del mensaje, como: AID del agente que envió el mensaje, AID del agente al cual se desea enviar el mensaje, tipo de mensaje y el contenido del mensaje.
- **Receivers.** Este atributo hace referencia a lista de agentes que recibirán el mensaje.
- **Subscribers.** Este atributo hace referencia a un contador que indica la cantidad de receptores del mensaje.
- **Caller.** Este atributo hace referencia al AID del agente que se estaba ejecutando cuando se creó el mensaje.

Por otro lado, los métodos utilizados por esta clase son:

- **Is Registered.** Este método se encarga de verificar si un agente se encuentra dentro del ambiente.
- **Get Mailbox.** Este método se encarga de obtener el un puntero hacia el buzón del agente.
- **Agent Msg.** Este método asigna el valor del atributo caller y ejecuta el método Clear All Receivers.
- **Add Receiver.** Este método se encarga de registrar un agente dentro de la lista de receptores del mensaje.
- **Remove Receiver.** Este método se encarga de eliminar a un agente dentro de la lista de receptores del mensaje.
- **Clear All Receivers.** Este método elimina a todos los receptores del mensaje.
- **Refresh List.** Este método se encarga de mantener actualizada la lista de receptores.
- **Receive.** Este método se encarga de obtener el tipo de mensaje enviado.
- **Send.** Este método se encarga de enviar el mensaje hacia un receptor en particular.
- **Send To Receivers.** Este método se encarga de instanciar el método Send para enviar el mensaje hacia cada uno de los receptores registrados para dicho mensaje.
- **Set Message Type.** Este método asigna el tipo de mensaje.
- **Set Message Content.** Este método asigna el contenido del mensaje.
- **Get Message.** Este método devuelve un puntero hacia el atributo Message de un mensaje.
- **Get Message Type.** Este método obtiene el tipo de mensaje.
- **Get Message Content.** Este método obtiene el contenido del mensaje.
- **Get Sender.** Este método obtiene el AID del agente que envió el mensaje.
- **Get Target Agent.** Este método obtiene el AID del agente al cual fue destinado el mensaje.
- **Registration.** Este método se encarga de solicitar al AMS la incorporación de un agente en la plataforma.
- **Deregistration.** Este método se encarga de solicitar al AMS la eliminación de un agente en la plataforma.

- **Suspend.** Este método se encarga de solicitar al AMS la suspensión de un agente en la plataforma.
- **Kill.** Este método se encarga de solicitar al AMS la eliminación de un agente en la plataforma y el borrar su tarea del calendarizador de FreeRTOS.
- **Resume.** Este método se encarga de solicitar al AMS el resumir la tarea de un agente en la plataforma.
- **Restart.** Este método se encarga de solicitar al AMS la eliminación e incorporación de un agente en la plataforma.

1.6. Clase Agent Platform

La clase Agent Platform resulta fundamental, ya que, mediante el uso de ella, se administra los agentes incorporados a la misma. De hecho, esta plataforma es una arquitectura tecnológica que sirve como una infraestructura física para dichos agentes[1]. Además, para un dispositivo de hardware embebido sólo se podrá crear una plataforma de acuerdo al paradigma MAES. Los atributos que conforman la clase Agent Platform son:

- **AMS Agent.** Este atributo hace referencia al agente AMS, el cual se encarga de administrar el uso de los demás agentes registrados en la plataforma.
- **Agent Handle.** Este atributo hace referencia a una lista que almacena el AID de cada agente registrado en la plataforma.
- **Description.** Este atributo hace referencia a un struct que contiene información esencial de la plataforma, como: AID del AMS, nombre de la plataforma y la cantidad de subscriptores.
- **Conditions.** Este atributo es una instancia de la clase User Defined Conditions que se encarga de administrar los diferentes estados de los agentes.
- **Conditions Pointer.** Este atributo hace referencia a un puntero de la clase User Defined Conditions.
- **Parameter.** Este atributo hace referencia a un struct que se utiliza como entrada de la función XTaskCreate de FreeRTOS para poder inicializar la tarea del agente AMS. Este struct incluye una instancia de la plataforma y una instancia de la clase User Defined Conditions.

Por otro lado, los métodos utilizados por la clase Agent Platform son:

- **Agent Platform.** Este método se encarga de asignar valores por defecto de los siguientes atributos de la plataforma: Parameter, AMS Agent, prioridad del agente AMS, nombre de la plataforma y cantidad de subscriptores de la plataforma.

- **Agent Platform with Conditions.** Este método se encarga de asignar valores por defecto de los siguientes atributos de la plataforma: puntero hacia los parámetros, cantidad de subscriptores y el nombre de la plataforma. A diferencia del método anterior, este únicamente redirige el puntero hacia las condiciones introducidas.
- **Boot.** Este método se encarga de iniciar la plataforma. Esto se realiza al iniciar la tarea del AMS Agent y al registrar, mediante el método Register Agent, cada uno de los agentes que se encuentran listados en el ambiente.
- **Agent Init.** Este método se encarga de crear un buzón para un agente, crear su respectiva tarea en FreeRTOS y registrar el agente dentro del ambiente.
- **Agent Init with Parameters.** Este método se encarga de crear un buzón para un agente, crear su respectiva tarea en FreeRTOS y registrar el agente dentro del ambiente tomando en cuenta parámetros de entrada que serán utilizados por su respectivo comportamiento.
- **Agent Search.** Este método se encarga de verificar si un agente se encuentra registrado dentro de la plataforma.
- **Agent Wait.** Este método se encarga de generar una espera utilizando el comando de FreeRTOS VTaskDelay.
- **Agent Yield.** Este método se encarga de detener la tarea que se encuentra ejecutando FreeRTOS en el momento de su instanciación.
- **Get Running Agent.** Este método se encarga de obtener el AID del agente que actualmente está ejecutando su comportamiento.
- **Get State.** Este método se encarga de obtener el estado del comportamiento de un agente en base a su AID.
- **Get Agent Description.** Este método se encarga de retornar la descripción del agente en base a su AID.
- **Get Agent Platform Description.** Este método se encarga de retornar la descripción de la plataforma.
- **Register Agent.** Este método se encarga de registrar a un agente dentro de la plataforma, donde se le asigna su respectiva prioridad.
- **Deregister Agent.** Este método se encarga de suspender a un agente en base a su AID y, posteriormente, se elimina dentro de la lista de agentes registrados en la plataforma.
- **Kill Agent.** Este método se encarga de eliminar el comportamiento de un agente en base a su AID y, posteriormente, se elimina dentro de la lista de agentes registrados en la plataforma.

- **Suspend Agent.** Este método se encarga de suspender a un agente en base a su AID.
- **Resume Agent.** Este método se encarga de resumir la ejecución del comportamiento de un agente basado en su AID.
- **Restart Agent.** Este método se encarga de eliminar el comportamiento de un agente y, posteriormente, se vuelve a crear el mismo comportamiento en base al AID de un agente.

1.7. Behaviours

Una sección importante sobre la implementación del paradigma MAES es el uso de los comportamientos de los agentes. Para esto, se desarrollan dos clases llamadas Cyclic Behavior y Oneshot Behaviour. Dentro de ellas, se establecen una serie de métodos que buscan encapsular el comportamiento del agente. Para poder analizar el flujo de trabajo de los comportamientos, se introduce un diagrama de flujo del método `execute` obtenido de la tesis desarrollada por Daniel Rojas sobre FreeMAES[2].

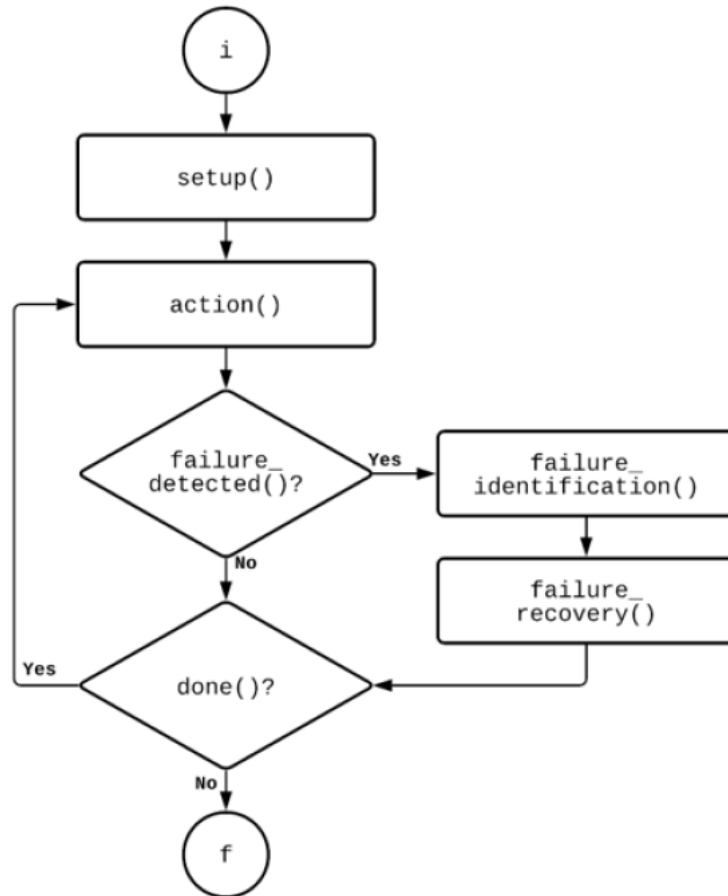


Figura 1.2: Diagrama de flujo de los comportamientos propuesto por Daniel Rojas[2].

Como se puede observar en la figura 1.2, un primer paso dentro del comportamiento hace referencia al setup. Este setup se encarga de establecer una configuración necesaria para el comportamiento. Luego, se ejecuta la acción del comportamiento y, si el usuario desea, se pueden ejecutar métodos FDIR para manejar errores dentro de la ejecución. Por último, mediante done, se realiza un ciclo infinito en caso de ser Cyclic Behaviour.

1.7.1. Clase Cyclic Behaviour

La clase Cyclic Behaviour tiene los siguientes atributos:

- **Task Parameters.** Este atributo hace referencia a parámetros de entrada para el comportamiento.
- **Message.** Este atributo hace referencia al mensaje utilizado por el comportamiento para comunicarse con otros agentes mediante el uso del AMS.

Por otro lado, los métodos utilizados por la clase Cyclic Behaviour son:

- **Action.** Este método hace referencia a la acción principal del comportamiento. En este método es donde se define la tarea del agente.
- **Setup.** Este método hace referencia a definiciones iniciales que se deben dar en el comportamiento antes de llegar a su acción.
- **Done.** Este método hace referencia si la tarea puede terminar o debe seguir en un ciclo. En este caso, al ser la clase Cyclic Behaviour, este método regresa un False. Por ende, se repite el comportamiento infinitamente.
- **Failure Detection.** Este método es un espacio para poder desarrollar un código de detección de errores si se requiere.
- **Failure Identification.** Este método es un espacio para poder desarrollar un código de identificación de errores si se requiere.
- **Failure Recovery.** Este método es un espacio para poder desarrollar un código de recuperación de errores si se requiere.
- **Execute.** Este método se encarga de ejecutar los métodos pasados de acuerdo al diagrama de la figura 1.2.

1.7.2. Clase Oneshot Behaviour

Con respecto a la clase Oneshot Behaviour, los atributos de esta clase son los siguientes:

- **Task Parameters.** Este atributo hace referencia a parámetros de entrada para el comportamiento.
- **Message.** Este atributo hace referencia al mensaje utilizado por el comportamiento para comunicarse con otros agentes mediante el uso del AMS.

Por otro lado, los métodos utilizados por la clase Cyclic Behaviour son:

- **Action.** Este método hace referencia a la acción principal del comportamiento. En este método es donde se define la tarea del agente.
- **Setup.** Este método hace referencia a definiciones iniciales que se deben dar en el comportamiento antes de llegar a su acción.
- **Done.** Este método hace referencia si la tarea puede terminar o debe seguir en un ciclo. En este caso, al ser la clase Oneshot Behaviour, este método regresa un True. Por ende, el comportamiento sólo se ejecuta una única vez.
- **Failure Detection.** Este método es un espacio para poder desarrollar un código de detección de errores si se requiere.
- **Failure Identification.** Este método es un espacio para poder desarrollar un código de identificación de errores si se requiere.
- **Failure Recovery.** Este método es un espacio para poder desarrollar un código de recuperación de errores si se requiere.
- **Execute.** Este método se encarga de ejecutar los métodos pasados de acuerdo al diagrama de la figura 1.2.

1.8. Clase Agent Organization

La clase Agent Organization tiene el propósito de poder organizar a los agentes en dos tipos de topologías: jerárquicas y modulares. En el caso de la topología jerárquica, está conformada por subordinados y un supervisor. Por ende, cada agente subordinado sólo puede comunicarse con su supervisor. Además, si se requiere realizar una comunicación entre organizaciones, esta sólo podrá llevarse a cabo entre los supervisores de ambas organizaciones. En cambio, si la topología es modular, cualquier agente puede comunicarse con otro agente dentro de la organización. Sin embargo, si se desea comunicar con un agente de otra organización, esta comunicación debe llevarse a cabo mediante el administrador

de las organizaciones. Por otro lado, es importante tomar en cuenta que, si no se crean organizaciones, se implementa una estructura donde cualquier agente puede establecer comunicación con cualquier otro agente.

Para la clase `Agent Organization`, existe un único atributo llamado `Description`. Este atributo hace referencia a un struct que contiene información básica de la organización, como: tipo de organización, cantidad de miembros registrados, AID de los miembros registrados, cantidad de miembros prohibidos, AID de los miembros prohibidos, AID del dueño, AID del administrador y AID del moderador.

Por otro lado, existen una serie de métodos en la clase `Agent Organization`, los cuales son:

- **Agent Organization.** Este método se encarga de asignar valores por defecto a toda la información contenida dentro del atributo de la clase.
- **Create.** Este método se encarga de crear una nueva organización.
- **Destroy.** Este método se encarga de eliminar una organización.
- **Is Member.** Este método se encarga de verificar si un agente es miembro a la organización.
- **Is Banned.** Este método se encarga de verificar si un agente se encuentra en la lista de agentes prohibidos de la organización.
- **Change Owner.** Este método se encarga de cambiar el dueño de una organización por otro agente basado en su AID.
- **Set Admin.** Este método se encarga de darle un encargo de administrador a un agente.
- **Set Moderator.** Este método se encarga de darle el rol de moderador a un agente.
- **Add Agent.** Este método se encarga de agregar un agente a la organización.
- **Kick Agent.** Este método se encarga de eliminar a un agente dentro de la lista de miembros de la organización.
- **Ban Agent.** Este método se encarga de agregar un agente a la lista de agentes prohibidos y se realiza el método `Kick Agent`.
- **Remove Ban.** Este método elimina un agente que se encuentre en la lista de agentes prohibidos.
- **Clear Ban List.** Este método elimina a todos los agentes que se encuentren en la lista prohibida.
- **Set Participant.** Este método asigna el rol de participante en un agente.
- **Set Visitor.** Este método asigna el rol de visitante de un agente.

- **Get Organization Type.** Este método retorna el tipo de organización.
- **Get Info.** Este método retorna la descripción de la organización.
- **Get Size.** Este método retorna la cantidad de miembros en la organización.
- **Invite.** Este método se encarga de enviar una invitación hacia un agente para unirse a la organización. Si este acepta, el metodo lo agrega a la lista de miembros de la organización.

1.9. Constructores

Para poder implementar clases en CMAES se crearon punteros dentro del struct de cada clase. Esto con el propósito de poder redirigir el puntero hacia una función que representa el método de una clase. Sin embargo, se requiere de una función que logre dirigir los punteros hacia dichas funciones. Por esta razón, se desarrolló una función llamada Constructor para cada una de las clases implementadas en CMAES. Por ende, estas funciones se encargan de ligar el puntero definido en el struct con todos los métodos descritos en las clases pasadas. A continuación, se preseta la figura 1.3, en la cual se puede observar la definición de los constructores utilizados en la implementación de CMAES.

```
void ConstructorAgente(MAESAgent* agente);  
void ConstructorSysVars(sysVars* Vars);  
void ConstructorAgent_Msg(Agent_Msg* Message, sysVars* env);  
void ConstructorAgent_Platform(Agent_Platform* platform, sysVars* env);  
void ConstructorCyclicBehaviour(CyclicBehaviour* Behaviour);  
void ConstructorOneShotBehaviour(OneShotBehaviour* Behaviour);  
void ConstructorAgent_Organization(Agent_Organization* Organization, sysVars* env);  
void ConstructorUSER_DEF_COND(USER_DEF_COND* cond);
```

Figura 1.3: Constructores implementados en CMAES.

1.10. CMAES.h

Al ser CMAES una biblioteca desarrollada en el lenguaje procedimental C, es necesario crear un archivo header que contenga las definiciones de las estructuras y clases utilizadas en el sistema multiagente. Para ello, se desarrolló un archivo llamado CMAES.h, el cual contiene dichas definiciones.

Capítulo 2

Uso de CMAES

Para poder demostrar la validez de la biblioteca CMAES, se desarrollaron una serie de aplicaciones que fueron ejecutadas en Visual Studio. Las aplicaciones mencionadas son: Sender-Receiver, Rock-Paper-Scissors y Telemetry. Con el propósito de poder evidenciar cómo fueron desarrolladas dichas aplicaciones, se muestra en el presente capítulo la estructura que debe de tener cualquier aplicación que utilice la biblioteca CMAES y se mostrarán los resultados de la simulación de la aplicación Telemetry. Esta aplicación fue escogida entre las tres debido al hecho de que simula el comportamiento de telemetría utilizado por un CubeSat al recolectar mediciones de corriente, tensión y temperatura.

2.1. Estructura de aplicaciones que utilizan CMAES

Para poder observar la estructura que debe tener una aplicación que utiliza CMAES se utiliza como referencia la aplicación Telemetry. Esta aplicación fue dividida en cinco partes que son abordadas en las siguientes secciones.

2.1.1. Includes

La primera sección de la aplicación contiene los includes necesarios para poder utilizar la biblioteca CMAES, entre ellos: headers de FreeRTOS y el header de CMAES. Es importante destacar que, si se piensa utilizar la versión que incluye la biblioteca CSP, se deben de agregar los headers de dicha biblioteca como se muestra en la figura 2.1.

```
#include "FreeRTOS.h"  
#include "task.h"  
#include "queue.h"  
#include "CMAES.h"
```

Figura 2.1: Includes utilizados en Telemetry.

2.1.2. Enums y structs

La segunda sección de la aplicación hace referencia a los Enums y Structs que se utilizan por la misma aplicación. Esta sección es completamente opcional, ya que depende completamente de la aplicación que se desea desarrollar. En la figura 2.2 se pueden observar los Enums y Structs utilizados en la aplicación Telemetry.

```

/*Enums and structs*/
typedef enum meas_type
{
    CURRENT,
    VOLTAGE,
    TEMPERATURE
}meas_type;

typedef struct logger_info
{
    meas_type type;
    UBaseType_t rate;
}logger_info;

```

Figura 2.2: Enums y Structs utilizados en Telemetry.

2.1.3. Definiciones de variables

La tercera sección de la aplicación hace referencia a las definiciones de las variables de la aplicación. Es importante generar variables distintas para cada agente, ya que al usar la misma variable pueden generarse errores donde se sobrescriba valores de otro agente. En la figura 2.3 se pueden observar las definiciones de las variables utilizadas en Telemetry.

```

//Defining the app's variables.

MAESAgent logger_current,logger_voltage,logger_temperature,measurement;
Agent_Platform Platform;
sysVars env;
CyclicBehaviour CurrentBehaviour,VoltageBehaviour,TemperatureBehaviour, genBehaviour;
Agent_Msg msg_current,msg_voltage,msg_temperature, msg_gen;
logger_info log_current, log_voltage, log_temperature, *info, *infox;
portFLOAT min, max, value;
char response[50];

```

Figura 2.3: Definiciones de las variables utilizadas en Telemetry

2.1.4. Comportamientos

La cuarta sección de la aplicación hace referencia a los comportamientos de cada agente. Para cada comportamiento, se deben de tomar en consideración tres secciones en particular.

- **Setup:** En esta sección se ejecutan comandos que serán ejecutados una única vez. Esto con el propósito de ejecutar cualquier comando que sirva de preparación para la acción del comportamiento.
- **Action:** En esta sección se ejecutan las acciones principales del comportamiento. Esta sección del comportamiento es la que se repite indefinidamente en caso de ser un comportamiento cíclico. En caso de ser un comportamiento OneShot, sólo se ejecuta una única vez.
- **Wrapper:** En esta sección se asigna una variable de clase Agent Message al comportamiento y se redirigen los punteros de las funciones setup y action del mismo comportamiento. Por último, se ejecuta la función execute que se encarga de iniciar la ejecución del comportamiento.

Para la aplicación Telemetry se realizaron únicamente dos comportamientos, uno de ellos siendo Logger y el otro siendo Generator. El comportamiento Logger es utilizado por tres distintos agentes llamados Current, Voltage y Temperature Logger. Es importante tomar en cuenta que, aunque los tres agentes utilicen el comportamiento Logger, se debe crear un wrapper para cada agente. Por ello, en la figura 2.4, se puede apreciar la función loggerAction haciendo referencia al comportamiento Logger y wrappers de cada agente.

```

//Functions related to the Logger Action Behaviour//

void loggerAction(CyclicBehaviour* Behaviour,void* pvParameters) {
    info= (logger_info*)pvParameters;
    Behaviour->msg->set_msg_content(Behaviour->msg, (char*)info->type);
    Behaviour->msg->send(Behaviour->msg, measurement.AID(&measurement), portMAX_DELAY);
    Behaviour->msg->receive(Behaviour->msg,portMAX_DELAY);

    /*Logging*/
    printf("%s\n", Behaviour->msg->get_msg_content(Behaviour->msg));
    Platform.agent_wait(&Platform,info->rate);
};

//Current Logger Wrapper:
void Currentlogger(void* pvParameters) {
    CurrentBehaviour.msg = &msg_current;
    CurrentBehaviour.msg->Agent_Msg(CurrentBehaviour.msg);
    CurrentBehaviour.action = &loggerAction;
    CurrentBehaviour.execute(&CurrentBehaviour, pvParameters);
};

//Voltage Logger Wrapper:
void Voltagelogger(void* pvParameters) {
    VoltageBehaviour.msg = &msg_voltage;
    VoltageBehaviour.msg->Agent_Msg(VoltageBehaviour.msg);
    VoltageBehaviour.action = &loggerAction;
    VoltageBehaviour.execute(&VoltageBehaviour, pvParameters);
};

//Temperature Logger Wrapper:
void Temperaturelogger(void* pvParameters) {
    TemperatureBehaviour.msg = &msg_temperature;
    TemperatureBehaviour.msg->Agent_Msg(TemperatureBehaviour.msg);
    TemperatureBehaviour.action = &loggerAction;
    TemperatureBehaviour.execute(&TemperatureBehaviour, pvParameters);
};

```

Figura 2.4: Comportamiento Logger utilizado en Telemetry.

Para el comportamiento Generator sólo se necesita un único wrapper, ya que sólo hay un agente llamado measurement que lo utiliza. En la figura 2.5 se puede observar la implementación del comportamiento Generator.

```
//Functions related to the Generator Behaviour//

void genAction(CyclicBehaviour* Behaviour, void* pvParameters) {
    Behaviour->msg->receive(Behaviour->msg,portMAX_DELAY);
    int i = (int)Behaviour->msg->get_msg_content(Behaviour->msg);
    switch ((int)Behaviour->msg->get_msg_content(Behaviour->msg))
    {
        case CURRENT:
            min = 0.1; //mA
            max = 1000; //mA
            value = min + rand() / (RAND_MAX / (max - min + 1) + 1);
            snprintf(response, 50, "\r\nCurrent mesasurment: %f\r", value);
            break;

        case VOLTAGE:
            min = 0.5; //V
            max = 3.3; //V
            value = min + rand() / (RAND_MAX / (max - min + 1) + 1);
            snprintf(response, 50, "\r\nVoltage mesasurment: %f\r", value);
            break;

        case TEMPERATURE:
            min = 30; //C
            max = 100; //C
            value = min + rand() / (RAND_MAX / (max - min + 1) + 1);
            snprintf(response, 50, "\r\nTemperature mesasurment: %f\r", value);
            break;

        default:
            snprintf(response, 50, "\r\nNot understood");
            printf(response);
            break;
    }

    Behaviour->msg->set_msg_content(Behaviour->msg,response);
    Behaviour->msg->send(Behaviour->msg, Behaviour->msg->get_sender(Behaviour->msg), portMAX_DELAY);
};

//Generator Wrapper
void gen(void* pvParameters) {
    genBehaviour.msg = &msg_gen;
    genBehaviour.msg->Agent_Msg(genBehaviour.msg);
    genBehaviour.action = &genAction;
    genBehaviour.execute(&genBehaviour, &pvParameters);
};
```

Figura 2.5: Comportamiento Generator utilizado en Telemetry.

2.1.5. Main

La quinta sección y última sección de la aplicación hace referencia al código principal o al main. En esta sección se ejecutan todos los comandos que se encargan de: introducir valores iniciales, construir cada una de las clases definidas, iniciar los agentes, inician la plataforma, registrar los agentes en la plataforma, inicializar la plataforma e iniciar el calendarizador de FreeRTOS. Esta serie de comandos se pueden observar en la figura 2.6 donde se puede observar el main de la aplicación Telemetry.

```
//Main
int telemetry() {
    wdt_disable(); //Disables watchdog timer
    wdt_clear();
    printf("-----Telemetry----- \n");

    //Rates and types for each logger.
    log_current.rate = 500;
    log_voltage.rate = 1000;
    log_temperature.rate = 2000;
    log_current.type = CURRENT;
    log_voltage.type = VOLTAGE;
    log_temperature.type = TEMPERATURE;

    //Constructors for each initialized class
    ConstructorAgente(&logger_current);
    ConstructorAgente(&logger_voltage);
    ConstructorAgente(&logger_temperature);
    ConstructorAgente(&measurement);
    ConstructorSysVars(&env);
    ConstructorAgent_Platform(&Platform, &env);
    ConstructorAgent_Msg(&msg_current, &env);
    ConstructorAgent_Msg(&msg_voltage, &env);
    ConstructorAgent_Msg(&msg_temperature, &env);
    ConstructorAgent_Msg(&msg_gen, &env);
    ConstructorCyclicBehaviour(&CurrentBehaviour);
    ConstructorCyclicBehaviour(&VoltageBehaviour);
    ConstructorCyclicBehaviour(&TemperatureBehaviour);
    ConstructorCyclicBehaviour(&genBehaviour);

    //Initializing the Agents and the Platform.
    logger_current.Iniciador(&logger_current, "Current Logger", 3, 10);
    logger_voltage.Iniciador(&logger_voltage, "Voltage Logger", 2, 10);
    logger_temperature.Iniciador(&logger_temperature, "Temperature Logger", 1, 10);
    measurement.Iniciador(&measurement, "Measure", 3, 10);
    Platform.Agent_Platform(&Platform, "telemetry_platform");

    //Registering the Agents and their respective behaviour into the Platform
    Platform.agent_initConParam(&Platform, &logger_current, &Currentlogger, (void*)&log_current);
    Platform.agent_initConParam(&Platform, &logger_voltage, &Voltagelogger, (void*)&log_voltage);
    Platform.agent_initConParam(&Platform, &logger_temperature, &Temperaturelogger, (void*)&log_temperature);
    Platform.agent_init(&Platform, &measurement, &gen);
    Platform.boot(&Platform);
    printf("CMAES booted successfully \n");
    printf("Initiating APP\n\n");

    // Start the scheduler so the created tasks start executing.
    vTaskStartScheduler(); //Might have to be changed because CSP Library takes control of the FreeRTOS scheduler

    for (;;)
    return 0;
}
```

Figura 2.6: Main utilizado en Telemetry.

2.2. Simulación utilizando Visual Studio

Para comprobar la validez de la biblioteca, se realizaron simulaciones de las tres aplicaciones de referencia utilizando Microsoft Visual Studio. Debido al hecho de que este capítulo se dedicó al ejemplo de Telemetry, se incluye en la figura 2.7 una captura de pantalla de la terminal al ejecutar dicha aplicación. A continuación, se presenta la figura 2.7.

```
-----Telemetry-----  
CMAES booted successfully  
Initiating APP  
  
Current mesasurment: 1.315264  
  
Voltage mesasurment: 2.641376  
  
Temperature mesasurment: 43.694927  
  
Current mesasurment: 785.575317  
  
Current mesasurment: 568.280212  
  
Voltage mesasurment: 2.323306  
  
Current mesasurment: 340.314575  
  
Current mesasurment: 870.288086  
  
Voltage mesasurment: 3.626430  
  
Temperature mesasurment: 82.894333  
  
Current mesasurment: 169.199509
```

Figura 2.7: Salida en la terminal serial al ejecutar Telemetry.

Como se puede observar en la figura 2.7, la aplicación Telemetry cumple con la función de indicar el valor medido por parte de cada uno de los agentes Logger. Además, es importante tomar en cuenta que cada agente Logger tiene un tiempo de espera distinto y, por ende, es que se imprimen las mediciones en el orden observado. Además, esto concuerda con el hecho de que la prioridad de ejecución de los agentes es de Current Logger, Voltage Logger y Temperature Logger respectivamente.

Capítulo 3

Instalación de CMAES en Visual Studio

Por último, para poder utilizar CMAES en Visual Studio, se debe de realizar una serie de pasos que se explican a continuación. Es importante destacar que, para esta explicación, se utiliza la versión de CMAES que no incorpora la biblioteca CSP.

En primera instancia, se crea un nuevo documento como se muestra en la figura 3.1.

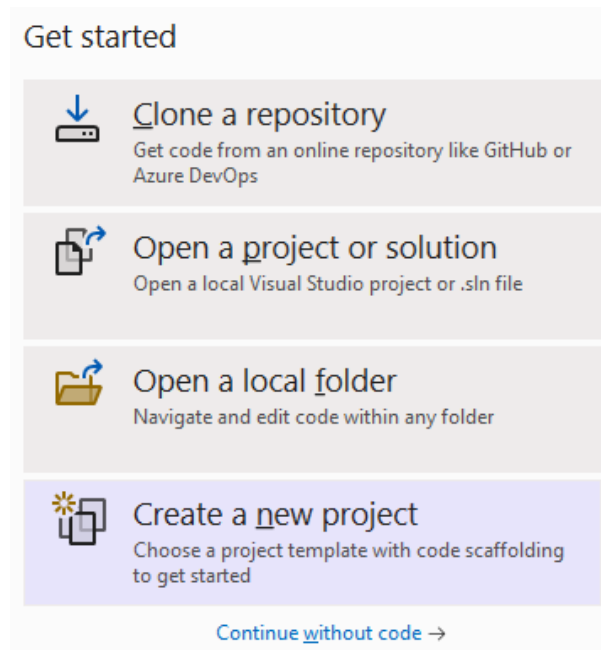


Figura 3.1: 1

Seguidamente, se debe seleccionar un proyecto en blanco con C++ como se muestra en la figura 3.2.

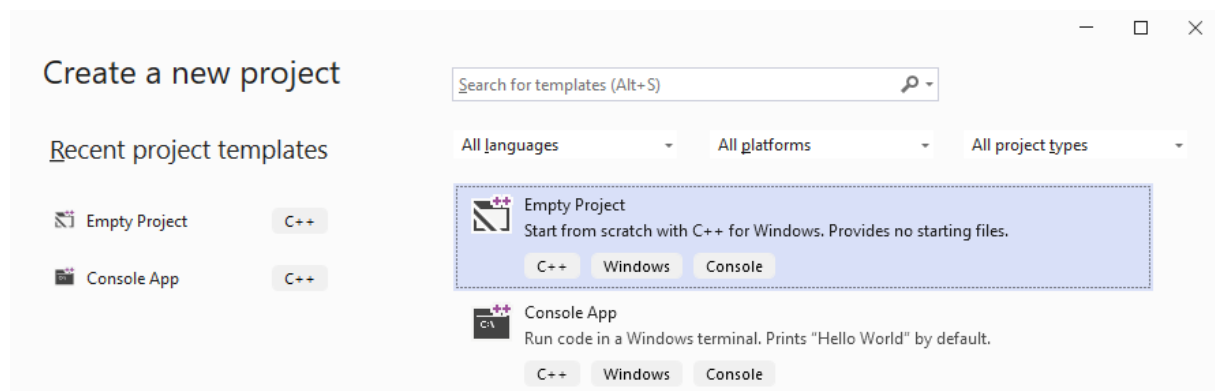


Figura 3.2: Diagrama de clases implementadas en CMAES.

Una vez seleccionado, se debe introducir el nombre del proyecto. Para este caso, se decidió nombrar el proyecto como CMAES. Esto se puede observar en la figura 3.3.

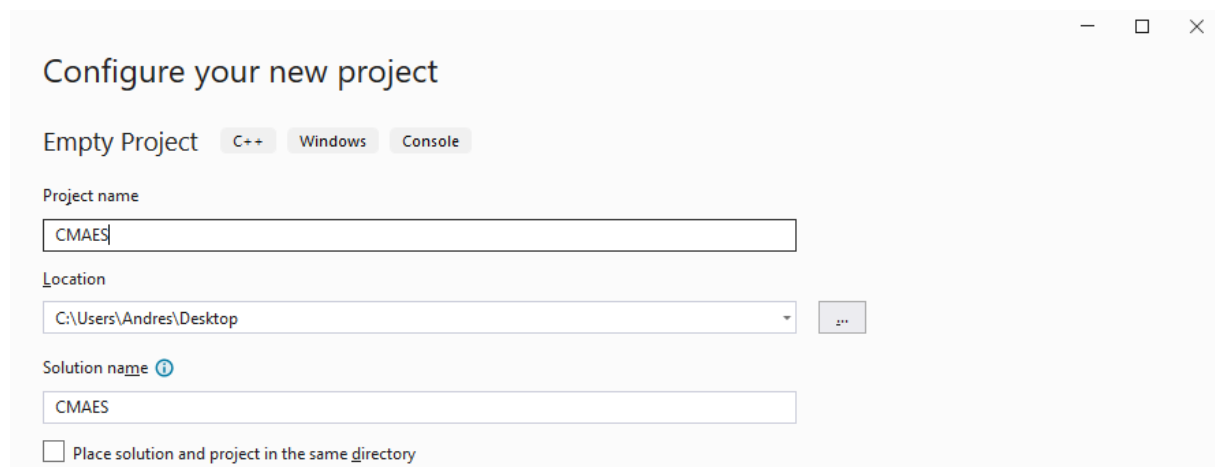


Figura 3.3: Diagrama de clases implementadas en CMAES.

Cuando el proyecto se crea, en la sección derecha de la aplicación se aprecia el explorador de solución que se puede apreciar en la figura 3.4.

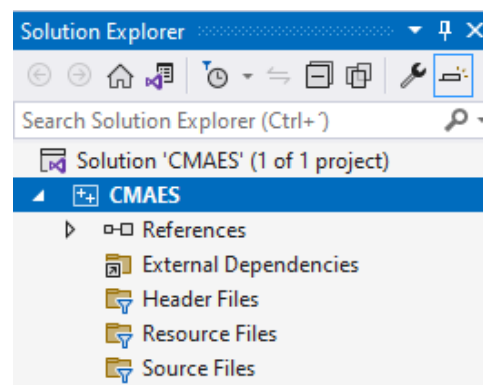


Figura 3.4: Diagrama de clases implementadas en CMAES.

En dicho explorador, se debe dar click derecho sobre el proyecto 'CMAES' y seleccionar la

opción 'Open Folder in File Explorer'. Seguidamente, se abre una pestaña del explorador de archivos como se muestra en la figura 3.5.




Nombre	Fecha de modificación	Tipo	Tamaño
 CMAES.vcxproj	28/1/2023 17:38	VC++ Project	7 KB
 CMAES.vcxproj.filters	28/1/2023 17:38	VC++ Project Filte...	1 KB
 CMAES.vcxproj.user	28/1/2023 17:38	Per-User Project O...	1 KB

Figura 3.5: Diagrama de clases implementadas en CMAES.

Luego, se deben copiar todos los archivos de la carpeta 'Visual Studio' en la carpeta del proyecto, la cual fue abierta en el punto anterior. Además, se debe de acceder a la carpeta 'CMAES' y se debe de copiar la carpeta 'libCMAES-FreeRTOS' en la misma carpeta del proyecto. Al copiar todos los archivos, la carpeta del proyecto debe ser idéntica a la mostrada en la figura 3.6.









 FreeRTOS_Source	28/1/2023 18:37	Carpeta de archivos	
 libCMAES-FreeRTOS	28/1/2023 18:37	Carpeta de archivos	
 Supporting_Functions	28/1/2023 18:37	Carpeta de archivos	
 CMAES.vcxproj	28/1/2023 17:38	VC++ Project	7 KB
 CMAES.vcxproj.filters	28/1/2023 17:38	VC++ Project Filte...	1 KB
 CMAES.vcxproj.user	28/1/2023 17:38	Per-User Project O...	1 KB
 FreeRTOSConfig.h	18/10/2022 12:57	C/C++ Header	7 KB
 Main.c	21/11/2022 15:36	C Source	1 KB

Figura 3.6: Diagrama de clases implementadas en CMAES.

Seguidamente, se debe de volver a la aplicación de Visual Studio y seleccionar el botón de 'Show All Files' que está ubicado abajo del título de 'Solution Explorer' como se observa en la figura 3.7.

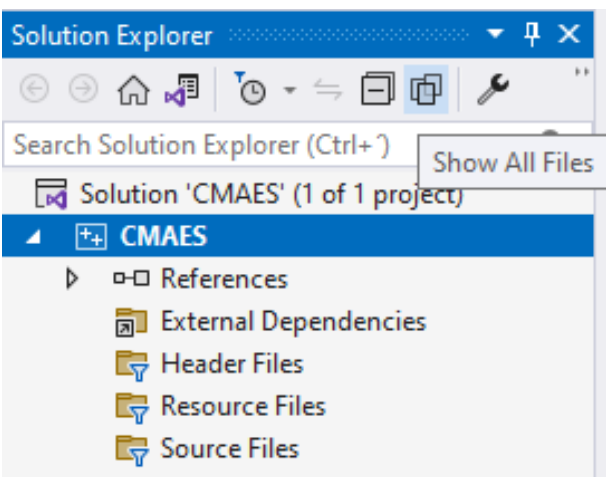


Figura 3.7: Diagrama de clases implementadas en CMAES.

Una vez apretado dicho botón, el explorador de solución debe verse de la forma que se aprecia en la figura 3.8. Luego, se deben de seleccionar las mismas carpetas y archivos que se observa en la misma figura.

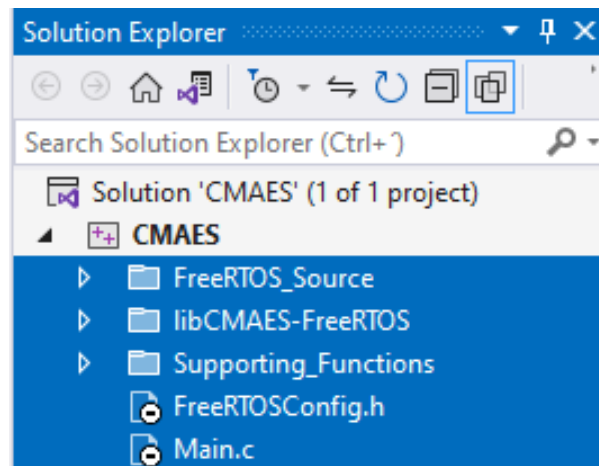


Figura 3.8: Diagrama de clases implementadas en CMAES.

Una vez seleccionados los archivos, se da click derecho sobre alguno de ellos y se selecciona la opción 'Include In Project'. Esto provoca que todos los archivos sean incorporados al proyecto y el aspecto de los archivos debe de ser igual al de la figura 3.9.

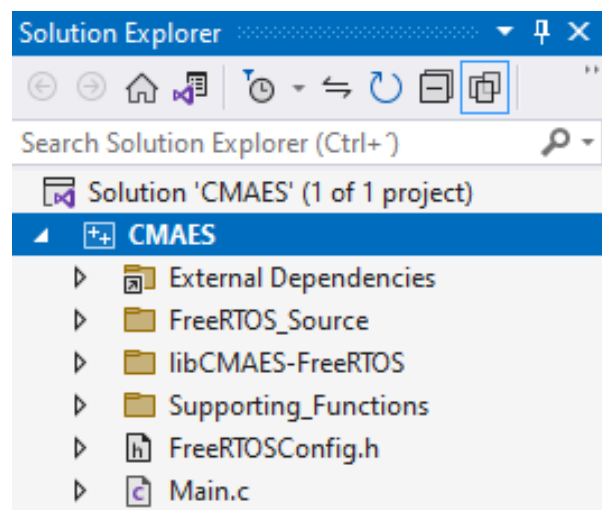


Figura 3.9: Diagrama de clases implementadas en CMAES.

Para terminar de incluir los archivos dentro del proyecto, se debe dar click derecho sobre el proyecto 'CMAES' y seleccionar 'Properties'. Una vez estando en el menú, se selecciona 'C/C++', 'General' y 'Additional Include Directories' como se muestra en la figura 3.10. Luego, se selecciona la opción 'Edit...'.

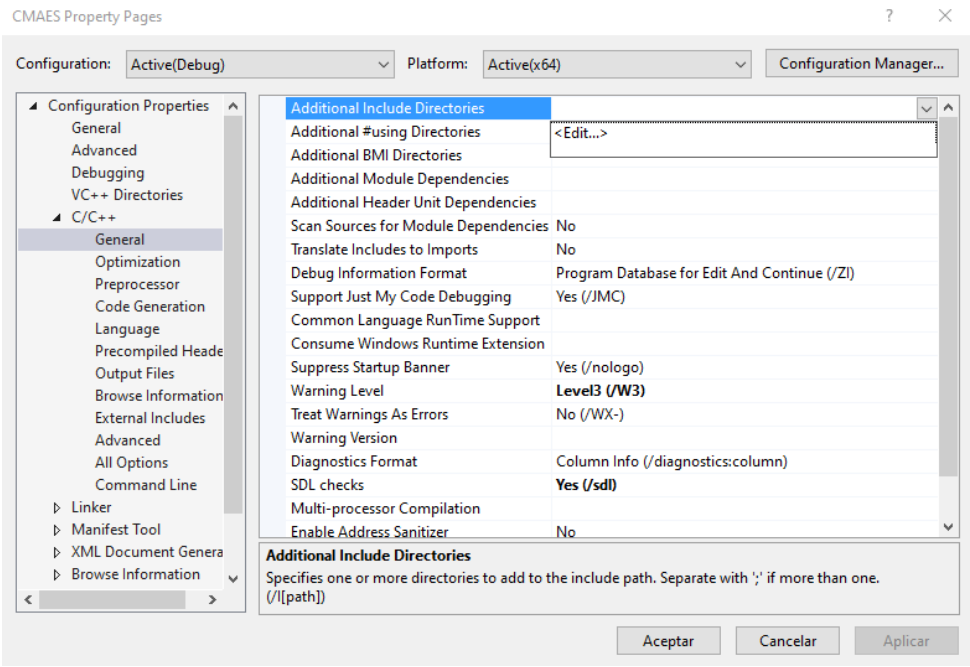


Figura 3.10: Diagrama de clases implementadas en CMAES.

Luego, se abre una pestaña como se muestra en la figura 3.11. En ella, se deben de insertar una serie de comandos al apretar el botón amarillo mostrado en la sección superior derecha de la pestaña.

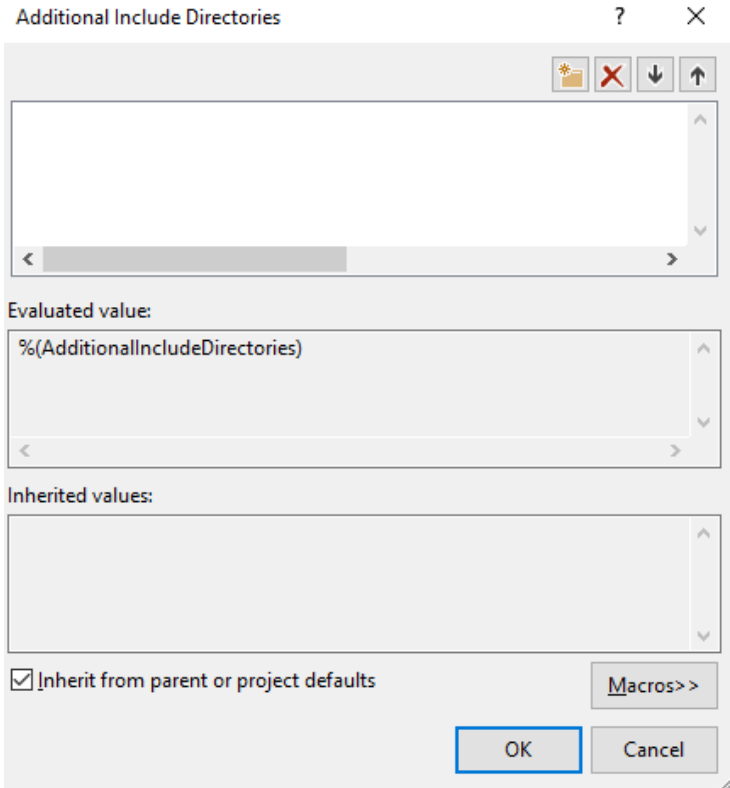


Figura 3.11: Diagrama de clases implementadas en CMAES.

Los comandos a introducir son:

- `$(ProjectDir)FreeRTOS_Source\portable\MSVC-MingW;`
- `$(ProjectDir)FreeRTOS_Source\portable\MemMang;`
- `$(ProjectDir)FreeRTOS_Source\include;`
- `$(ProjectDir)FreeRTOS_Source;`
- `$(ProjectDir)Supporting_Functions;`
- `$(ProjectDir);`
- `$(ProjectDir)libCMAES_FreeRTOS\src;`
- `$(ProjectDir)libCMAES_FreeRTOS\include;`
- `% (AdditionalIncludeDirectories)`

Una vez introducidos los comandos, estos deben de ser aplicados y se debe de obtener una pestaña similar a la observada en la figura 3.12.

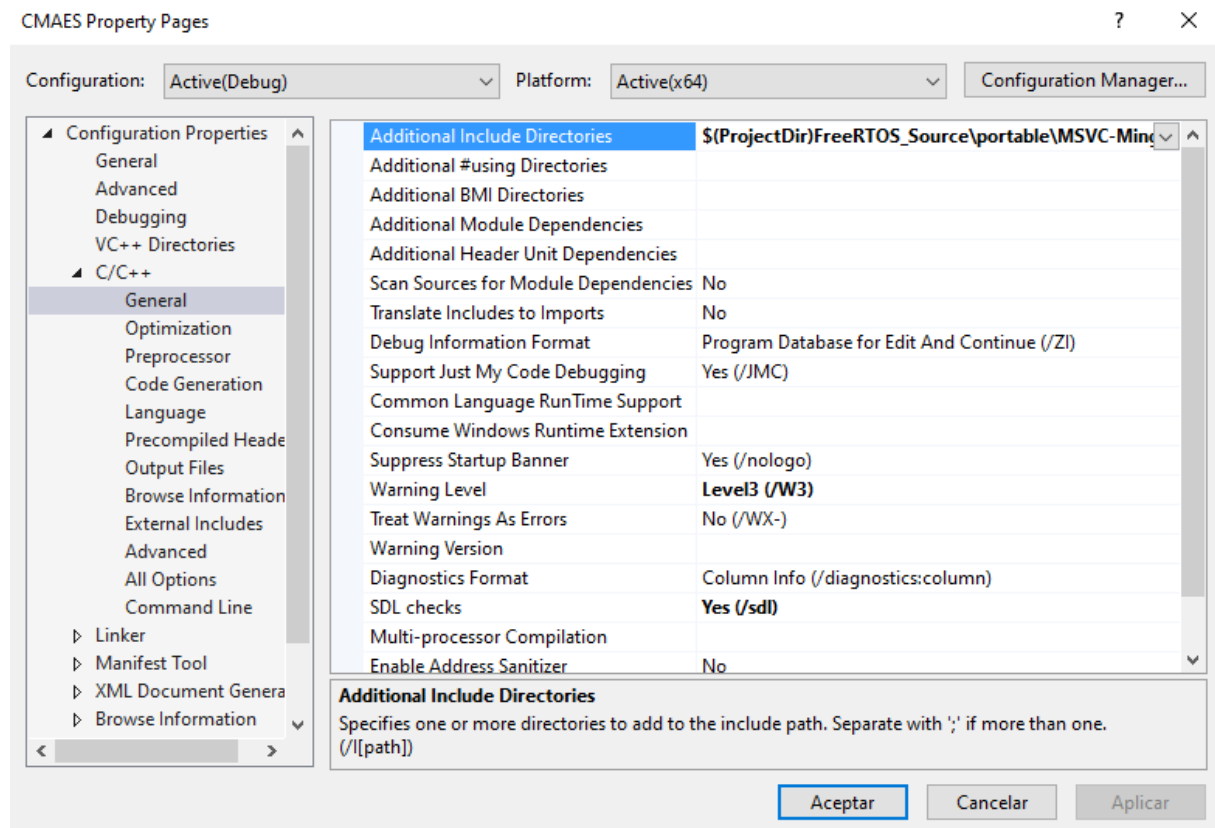


Figura 3.12: Diagrama de clases implementadas en CMAES.

Por último, se acepta la pestaña obtenida. Si se realizaron correctamente todos los pasos anteriores, la biblioteca CMAES se encuentra disponible para ser utilizada por las aplicaciones que se deseen desarrollar.

Bibliografía

- [1] C.Chan, «MAES: A Multi-Agent Systems Framework for Embedded Systems,» *M.S Thesis, Delft University of Technology, Netherlands*, 2017.
- [2] D. Rojas, «FreeMAES: Desarrollo de una biblioteca bajo el Paradigma Multiagente para Sistemas Embebidos (MAES) compatible con la NanoMind A3200 y el kernel FreeRTOS,» *B.S. Thesis, Ingeniería en Electrónica, Instituto Tecnológico de Costa Rica, Costa Rica*, 2021.