Universidad de Cantabria

Facultad de Ciencias

ADAPTACIÓN DE LA LIBRERÍA OPENCAT A M2OS PARA EL CONTROL DE UN ROBOT CUADRÚPEDO

Adaptation of the OpenCat library to M2OS for the control of a quadruped robot

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Pablo Odriozola Díaz

Director: Mario Aldea Rivas

Co-Director: Héctor Pérez Tijero

Septiembre – 2023

Índice

Resumen

Este proyecto tiene como objetivo portar la librería OpenCat para control de robots cuadrúpedos desarrollada por los creadores del perro robot Bittle, perteneciente a la compañía Petoi, al sistema operativo de tiempo real M2OS.

Como el proyecto de Petoi, está escrito mayoritariamente en C/C++ y pensado para su uso a través de un programa Arduino, necesitaremos ser capaces de integrar código C en un proyecto de Ada, que es el lenguaje en el que está escrito y con el que funciona M2OS. Para ello haremos uso de algunas herramientas que M2OS nos brinda para exportar código Arduino (escrito en C/C++) a una librería estática. Dicha librería podrá ser enlazada con los programas Ada del usuario que ejecutan sobre M2OS mediante el uso de un código de adaptación.

Además, para sacar partido de la multitarea proporcionada por M2OS, el código original se fragmentará en distintas tareas que el robot ejecutará según los parámetros que se definan, como la prioridad y el periodo. Esta modularización permitirá establecer la frecuencia de ejecución de cada tarea en función de su importancia, a diferencia de la ejecución puramente secuencial, donde todas las funciones se ejecutan un número fijo de veces en cada iteración del bucle final, la nueva implementación brindará mayor versatilidad y flexibilidad.

El conjunto de tareas abarcará diversas acciones del robot, entre las que destacan: comandar a Bittle, reaccionar en función de los comandos preestablecidos y responder ante situaciones inesperadas, como caídas o inclinaciones.

Palabras clave:

Tiempo Real, Microcontroladores, Robot, Arduino, M2OS, Petoi

Abstract

This project aims to port the OpenCat library for quadruped robot control developed by the creators of the robot dog Bittle, belonging to the company Petoi, to the M2OS real-time operating system.

As the Petoi project is mostly written in C/C++ and intended for use through an Arduino program, we will need to be able to integrate C code into an Ada project, which is the language in which it is written and with which M2OS works. To do this we will make use of some tools that M2OS gives us to export Arduino code (written in C/C++) to a static library. Said library can be linked to the user's Ada programs running on M2OS through the use of an adapter code.

In addition, to take advantage of the multitasking provided by M2OS, the original code will be fragmented into different tasks that the robot will execute according to parameters that are defined, such as priority and period. This modularization will allow setting the execution frequency of each task according to its importance, as opposed to purely sequential execution, where all functions are executed a fixed number of times in each iteration of the final loop, the new implementation will provide greater versatility and flexibility.

The set of tasks will cover various actions of the robot, including: commanding Bittle, reacting to preset commands, and responding to unexpected situations, such as falling or tilting.

Key Words:

Real Time, Microcontrollers, Robot, Arduino, M2OS, Petoi

1. Introducción

1.1 Motivación

El uso generalizado de microcontroladores en sistemas embebidos es una realidad cotidiana. Estos dispositivos se encuentran presentes en diversos ámbitos, desde lavadoras modernas hasta automóviles y dispositivos de domótica. Además de aplicaciones profesionales, también existe un amplio interés en proyectos amateur donde entusiastas y empresas, como Petoi, se dedican al desarrollo de robots. Estos proyectos tienen como objetivo brindar experiencias divertidas y didácticas en el campo de la robótica.

Petoi [1], concretamente, desarrolla robots cuadrúpedos como perros y gatos, basados en placas Arduino Uno modificadas [2], los cuales pone en venta al público. Esta empresa ofrece una base establecida, mediante un repositorio de GitHub [3] en forma de una librería llamada OpenCat, donde se alojan y actualizan las funcionalidades básicas para el funcionamiento de sus robots. El código proporcionado en este repositorio, desarrollado en Arduino acompañado de librerías C/C++, es adaptable tanto para robots perros como para gatos. Sin embargo, en el contexto de este proyecto, nos centraremos en el perro Bittle.

La librería OpenCat incluye todo lo necesario para el arranque del robot, la recepción y ejecución de órdenes, así como otras funciones adicionales, como la capacidad de emitir pequeñas melodías utilizando un zumbador incorporado en la placa, medición de la batería restante y respuestas ante caídas, entre otras.

No obstante, este código utiliza un modelo de programación secuencial para su ejecución. Esta elección conlleva algunas desventajas:

- **Eficiencia reducida en el uso de recursos:** En un enfoque secuencial, cada función debe ejecutarse en orden, lo que fuerza la espera a que acabe una función o tarea antes de pasar a la siguiente. Esto puede llevar a una subutilización de los recursos del microcontrolador debido a que algunas tareas pueden quedar inactivas mientras se espera a que otras finalicen.

- **Mayor tiempo de respuesta:** En el caso secuencial, una tarea larga puede llegar a retrasar otras tareas que necesitan ejecutarse rápidamente, produciendo un tiempo de respuesta más lento para operaciones que pueden ser críticas.

- **Dificultad para gestionar eventos en tiempo real:** Si el sistema debe responder a eventos externos en tiempo real, el enfoque secuencial puede no ser la mejor opción. Precisamente por el punto anterior, un enfoque concurrente basado en una jerarquía de tareas puede ofrecer reacciones más ágiles.

Para paliar estos inconvenientes, se propone la utilización del sistema operativo de tiempo real M2OS, el cual ofrece un enfoque multitarea basado en tareas no expulsoras, como base sobre la que funcione esta librería.

1.2 Objetivos

El principal objetivo de este proyecto se centra en la adaptación de la librería de Petoi, llamada OpenCat, para su ejecución sobre M2OS a través de las herramientas de importación de código Arduino escrito en C y C++, obteniendo así un entorno concurrente, basado en la división del código en tareas con una prioridad y periodo establecidos, en el que podamos seguir aplicando las funcionalidades ya brindadas por el equipo de Petoi.

De esta manera, podríamos decir que el objetivo principal se puede descomponer en los siguientes subobjetivos:

- Comprender el funcionamiento y la estructura interna de la librería OpenCat.
- Compilar la librería para M2OS y proporcionar una API básica para su uso.
- Integrar la librería haciendo uso de tareas de M2OS.

2. Herramientas, tecnologías y metodología

2.1 Tecnologías y lenguajes

2.1.1 Arduino

Arduino [4] es una plataforma de software de código abierto y hardware libre cuyo objetivo es la facilitación del desarrollo de proyectos electrónicos. Fue creada en 2005 por un equipo de estudiantes en Ivrea, Italia. Pretende ser una herramienta accesible para cualquiera, tanto entusiastas como profesionales.

Imagen 1. Logo de Arduino

Arduino se compone principalmente de los siguientes tres elementos:

- **IDE de Arduino:** El Entorno de Desarrollo Integrado de Arduino ofrece a los usuarios una plataforma intuitiva y accesible para crear proyectos. Cuenta con todo lo necesario para el desarrollo de código en este lenguaje, incluyendo un editor de código, un compilador, un cargador de programas, y otras funcionalidades adicionales que facilitan las tareas a los usuarios, como un gestor de librerías, inclusión de un monitor de comunicación serie, etc.

  Cabe destacar que este IDE utiliza un proceso de desarrollo cruzado, dado que el código que se diseña y compila en un computador, pero se ejecuta en otro dispositivo (una placa Arduino). Por lo que una vez que el código está listo, se puede cargar en una placa Arduino desde el ordenador de desarrollo a través de una conexión USB.

- **Software de usuario:** El software que el usuario desarrolla en este IDE está basado en el lenguaje C/C++ pero simplificado para facilitar su uso al público más novato y cuenta con diversas librerías y ejemplos ya preparados para su uso inmediato en una placa Arduino. Además, existen muchos casos de usuarios, ya sean individuales o empresas, que crean sus propias librerías de uso general o de manejo de dispositivos y las comparten con los demás usuarios de forma abierta, como es el caso de Petoi con OpenCat.

  La peculiaridad de este lenguaje es que todos los programas desarrollados en este entorno tienen que respetar una estructura específica que consta de dos funciones principales:

- **setup():** Esta función se ejecuta una vez al comienzo del programa y su objetivo es realizar una configuración inicial de variables, comunicación serie, entrada/salida, etc. Para abreviar, en esta sección es donde se deben llevar a cabo todas las inicializaciones necesarias antes de proceder con la ejecución del bucle infinito.

- **loop():** Esta es la función que se ejecutará de forma indefinida en la placa Arduino, al menos, hasta que se cargue otro programa o la placa se quede sin alimentación. Aquí se colocarán las principales acciones que la placa llevará a cabo, como la lectura de sensores, el control de los actuadores, operaciones aritméticas, condicionales, etc.

Arduino cuenta con algunas librerías fundamentales que proporcionan funciones y rutinas que son imprescindibles para el funcionamiento básico de un programa escrito en el entorno Arduino. Este conjunto de librerías es lo que se conoce como “Arduino Core” (núcleo de Arduino) y existen distintas versiones para cada una de las arquitecturas soportadas por Arduino. El acceso al “Arduino Core” desde los programas Arduino se realiza a través de una API estándar [5].

- **Hardware (Placas de Arduino):** Son placas de circuito impreso que cuentan con un microcontrolador y un conjunto de pines entrada/salida que permite conectar distintos dispositivos electrónicos como sensores o actuadores. Con el paso del tiempo, el equipo de Arduino ha ido lanzando nuevas versiones y modelos de estas placas, que cada vez cuentan con más mejoras o características concretas. De entre los modelos más conocidos figuran Arduino Nano, Arduino Mega y, la placa utilizada para este proyecto, Arduino Uno.

  Las placas Arduino son hardware libre, es decir, un hardware en el que todas sus especificaciones están disponibles para todo el público. Esto es importante, porque a partir de ello terceras entidades pueden crear sus propias versiones del microcontrolador añadiendo detalles adicionales sobre una base ya establecida.

2.1.1.1 Arduino Uno

La placa Arduino Uno es de las más populares de entre las desarrollada por la compañía. Fue lanzada en 2010, y se ha convertido en la opción preferida del público amateur dada su versatilidad y sencillez.

Está basada en el ATmega328P, el cual es un microcontrolador AVR de 8 bits con una frecuencia de 16MHz y una memoria Flash de 32KB para el almacenamiento de los programas. También cuenta con 14 pines entrada/salida, una memoria SRAM de 2KB y otra EEPROM de 1KB para el almacenamiento de datos y configuraciones que permanecen incluso cuando la placa no está siendo alimentada.

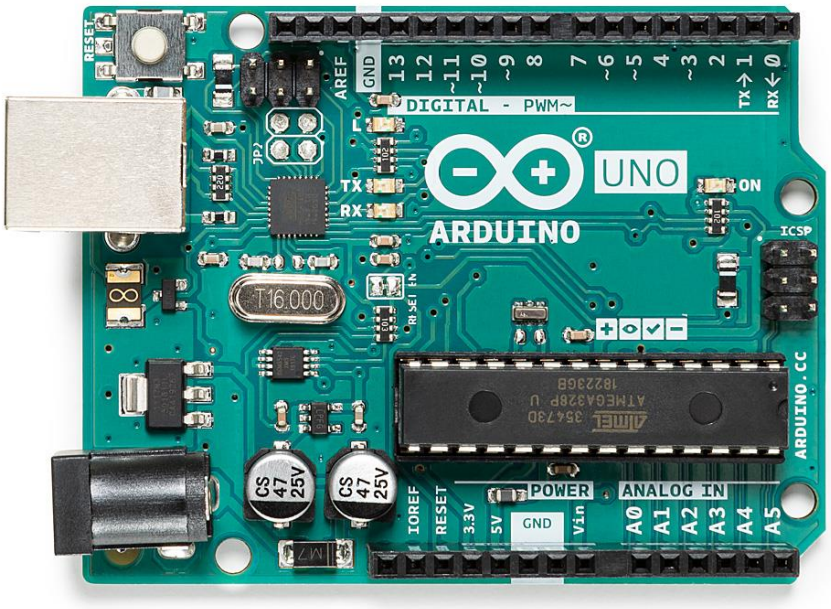

Imagen 2. Placa Arduino Uno

2.1.1.2 NyBoard

Como se acaba de mencionar, crear tu propia versión de una placa Arduino es posible, gracias al hecho de ser un hardware libre. Petoi es uno de los ejemplos de entidades que han lanzado su propia versión de la placa Arduino Uno, llamada NyBoard [2], cuyas caras frontal y trasera se muestran respectivamente en las imágenes 3 y 4.

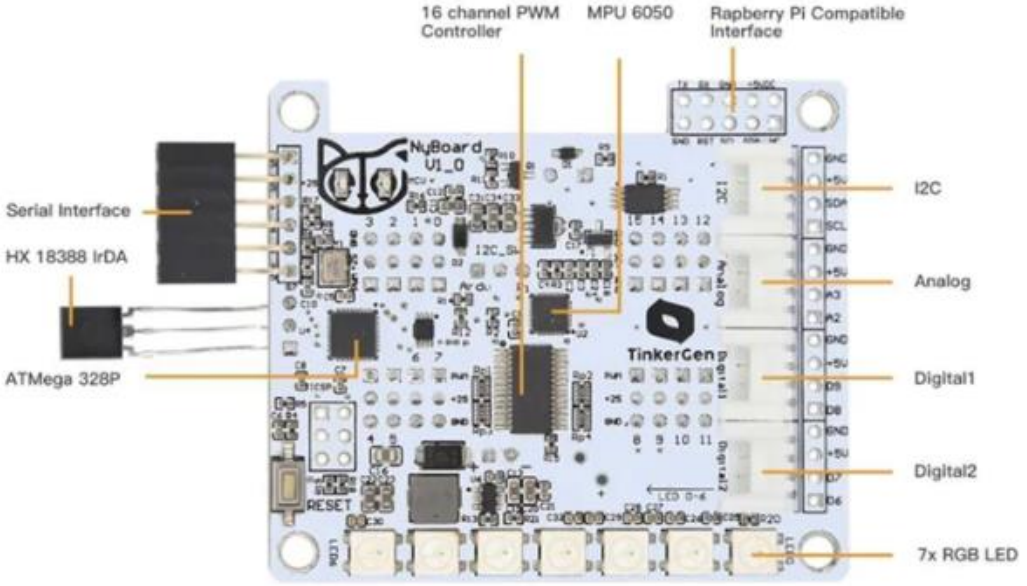

Imagen 3. Placa Nyboard parte frontal

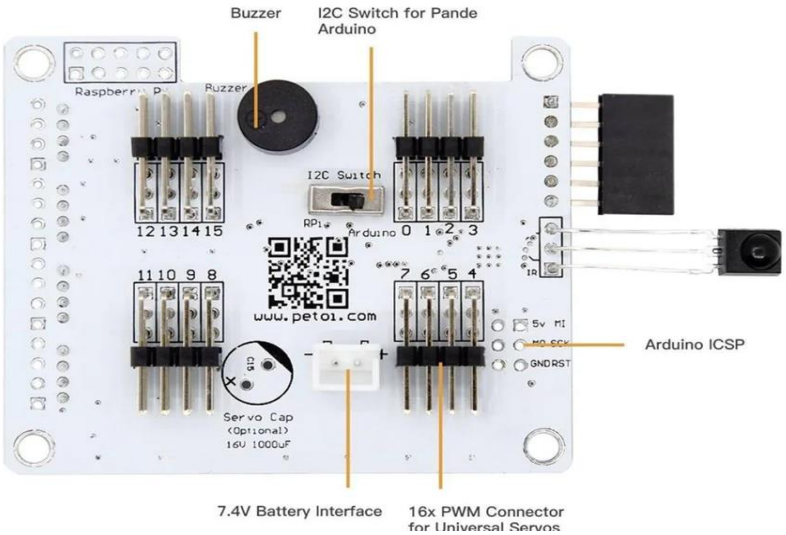

Imagen 4. Placa Nyboard parte trasera

Esta derivación de Arduino Uno cuenta con las siguientes características:

- Manejo de hasta 16 servos universales a través de unos conectores PWM (Pulse-Width Modulation).

- Uso de una IMU (Inertial Measurement Unit) para la detección del estado del cuerpo, midiendo tanto la orientación como la aceleración del mismo para ser capaz de mantener el equilibrio.

- Incluye una EEPROM, conectada a través de I2C, de 8KB donde se guarda la información de las habilidades que Bittle puede ejecutar.

- Infrarrojos, que sirven a modo de comunicación inalámbrica. El robot Bittle viene con un mando a distancia que aprovecha esta tecnología para recibir órdenes predeterminadas como acciones y gestos.

- 4 sockets para módulos Arduino externos.

- Soporte para Raspberry Pi: La placa cuenta también con un socket 2x5 en NyBoard V1 que le permite conectar una Raspberry Pi, aumentando así las posibilidades del sistema, pudiendo analizar más datos, conectarse a Internet, y más.

- Un zumbador con el que poder emitir pequeños sonidos y melodías.

- Una interfaz para conectar una batería 7.4V, necesaria para que el robot pueda actuar sin tener que estar permanentemente conectado a una alimentación.

En cuanto a las especificaciones más básicas como microcontrolador, memoria Flash, memoria SRAM, etc. NyBoard sigue manteniendo las mismas características que la Arduino Uno original previamente detallada, puesto que ambas placas están basadas en el microcontrolador ATmega328P.

2.1.2 Bittle

Bittle es el robot cuadrúpedo que otorga a la placa NyBoard el esqueleto necesario para llevar a cabo todas sus acciones y gestos (ver Imagen 5). Para ello la estructura del perro robot cuenta con una serie de servos y piezas que permiten imitar los movimientos de los perros reales.

Imagen 5. Bittle

Como se puede apreciar en esta imagen, el robot consta de 4 tipos de piezas principales:

- **Torso:** Es la parte que servirá de base a la placa NyBoard, cubierta por un protector negro en la espalda del robot. También alberga la batería que proporciona autonomía al robot, quedando sujeta a través de una ranura en el abdomen del perro.

- **Cabeza:** Tiene la capacidad de rotar ligeramente hacia ambos lados gracias a un servo ubicado en su interior. Además, Bittle puede abrir la boca de forma fija, lo que permite agregar adornos como un hueso de juguete o, de manera más interesante, integrar sensores para detectar obstáculos y actuar en consecuencia mediante la programación adecuada.

- **Mitades superiores de patas:** Estas piezas conectan el torso del robot con las mitades inferiores de las patas. No contienen servos por sí mismas, pero se unen a las partes rotatorias tanto en la parte superior (hombros y cadera del perro) como en la parte inferior (mitades inferiores de las patas), donde se encuentran ubicados los servos.

- **Mitades inferiores de patas:** Representan la porción de las extremidades a partir de las muñecas (en las patas delanteras) y los tobillos (en las patas traseras). Cada mitad inferior cuenta con ranuras donde los servos se encajan para conectarlas con las mitades superiores.

Por otro lado, es conveniente destacar como los desarrolladores de Petoi han indexado los servos de sus robots. Ellos definen un orden en base a la distancia del servo a indexar respecto al torso. Por ejemplo, dado que la articulación del hombro está más cerca del torso de lo que lo está la articulación del codo, el índice de este último será mayor. [6]

Petoi se basa en la estructura física de los animales cuadrúpedos para conocer los puntos que requerirían de un servo si se le quiere otorgar la función de moverse en algún sentido. Definiendo así 16 articulaciones de interés (representado con la constante `DOF`, “degrees of freedom”, en la librería OpenCat), tal y como se aprecia en la Imagen 6.

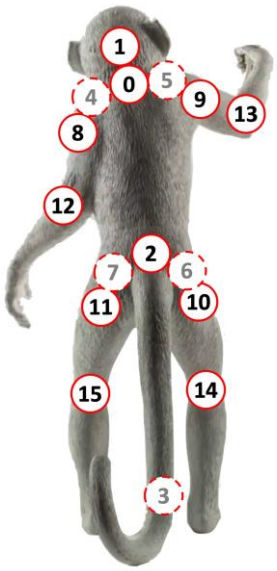

Imagen 6. Indexación de las articulaciones de un mono

Siguiendo esta estructura, se ordenan las articulaciones de la siguiente manera: giro de la cabeza, inclinación de la cabeza, giro de la cola, inclinación de la cola, 4 articulaciones para el balanceo de hombro y cadera, otras 4 para el cabeceo del hombro y 4 articulaciones más para codos y rodillas. Dentro de las articulaciones que se encuentren en el mismo grupo, se indexan en sentido de las agujas del reloj desde la esquina frontal izquierda si el cuerpo se observa desde atrás.

Aunque los robots diseñados por Petoi no tengan tantas articulaciones como las que han esquematizado, respetan el índice de todas sin reducirlos a pesar de que, por ejemplo, Bittle no tenga 16 servomotores. Es por ello, que los servos de Bittle no están enumerados de 0 a 8.

Dicho esto, en el mapeo de estas articulaciones a Bittle obtenemos el siguiente esquema, visible en la Imagen 7:

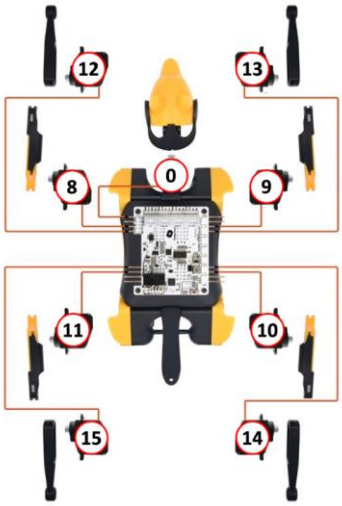

Imagen 7. Indexación de los servomotores de Bittle

De esta forma, con esta estructura y el uso de los 9 servos, el robot es capaz de replicar los movimientos naturales del animal, permitiendo una mayor similitud con un perro real.

2.1.3 Ada

Ada [7] es un lenguaje de programación de alto nivel diseñado principalmente para el desarrollo de aplicaciones críticas, sistemas embebidos y de tiempo real por el Departamento de Defensa de los Estados Unidos en la década de 1970. El nombre de este lenguaje rinde homenaje a Ada Lovelace, considerada una de las primeras programadoras de la historia.

Imagen 8. Logo de Ada

Este lenguaje ha sido diseñado para ofrecer un alto nivel de seguridad, lo que lo convierte en una elección ideal para proyectos que requieren precisión y robustez. Por otro lado, Ada también destaca por un tipado fuerte y estático, un sólido soporte para la ejecución concurrente y una gestión de memoria y errores cuidadosamente diseñada.

Gracias a estas características, Ada es ampliamente utilizada en proyectos donde los errores software son inadmisibles como en los sistemas de control de tráfico aéreo, aviación y proyectos militares, entre otros.

2.1.4 M2OS

M2OS [8][9] es un sistema operativo de tiempo real pequeño que permite la ejecución de programas multitarea en pequeños microcontroladores con escasos recursos de memoria, como lo es Arduino Uno, que es una de las 3 placas soportadas por este RTOS, junto a Epiphany y STM32F4.

M2OS implementa una política de programación multitarea basada en tareas no expulsoras de disparo único (one-shot) que requieren un espacio en memoria considerablemente reducido. Además, gracias a esta política de programación, todas las tareas pueden utilizar una única sección de pila (stack) en memoria, lo que, en consecuencia, supone que solo se necesita dedicar tanta memoria a pila como la que se necesitaría proporcionar a la tarea que más espacio requiriese para su propia pila.

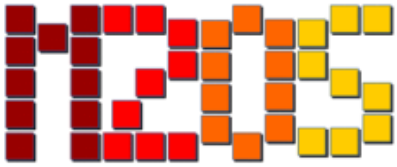

Imagen 9. Logo de M2OS

Este sistema operativo está escrito en Ada, está integrado en el IDE GPS (GNAT Programming Studio) el cual simplifica el proceso de desarrollo, carga y depuración del código. También es la base de un RTS (Run-Time System) sencillo para el compilador GNAT Ada, que proporciona el soporte para la concurrencia basada en tareas Ada.

3. Librería OpenCat y compilación para M2OS

3.1 OpenCat

3.1.1 Objetivos y programa principal original

La librería OpenCat contiene todo el código fuente desarrollado por Petoi, el cual se puede encontrar en su repositorio de GitHub [3]. En esta sección se procura explicar, de la forma más concisa posible, las funcionalidades que la compañía ha desarrollado y que permiten al robot Bittle realizar tantos gestos y acciones.

Esta librería otorga a los robots de Petoi la capacidad de llevar a cabo una correcta inicialización de la máquina (puesta en marcha de los distintos parámetros para que los motores funcionen adecuadamente), para que, después, esta sea capaz de recibir órdenes y actuar adecuadamente en función de las mismas, ya sea mediante una comunicación serie (escribiendo los comandos que se desean realizar) o mediante una comunicación a través de infrarrojos (a través del uso de un mando que viene ya preparado para su uso junto a Bittle).

Originalmente, el programa principal de OpenCat, un fichero ".ino", presentaba la siguiente estructura (simplificada), dividida en las dos funciones indispensables de Arduino (`setup()` y `loop()`):

```
#include "src/OpenCat.h"

void setup() {
  Serial.begin(BAUD_RATE);
  Serial.setTimeout(SERIAL_TIMEOUT_SHORT);
  Wire.begin();
  Wire.setClock(500000L);

  initRobot();
}
```

La función `setup()` que, como se ha dicho con anterioridad, se ejecuta una vez al comienzo del programa, inicializa la comunicación serie y la comunicación I2C, y finalmente llama a `initRobot()` (explicada más adelante) para que Bittle inicie sus servos, IMU, prepare sus habilidades, etc. Con esto preparamos el entorno para que la placa Arduino pueda funcionar correctamente y comunicarse con otros dispositivos.

```
void loop() {
#ifdef MAIN_SKETCH
  lowBattery();  //  block the loop if battery is low
  // can be disabled to save programming space

  readEnvironment();     //reads the IMU (Inertia Measurement
  dealWithExceptions();  //fall over, lifted, etc.

  readSignal();  //commands sent by user interfaces and sensors
```

```
  otherModule();  //you can create your own code here

  reaction();  //handle different commands
#else
  calibratePCA9685();
#endif
}
```

Por otro lado, la función `loop()` tiene 2 modos de operación principales entre los que se cambia mediante la definición (o la ausencia) de una macro llamada `MAIN_SKETCH`.

En el caso de que la macro `MAIN_SKETCH` no se encuentre definida, Bittle lleva a cabo operaciones de configuración, como el guardado de habilidades en la EEPROM o la calibración del robot, para corregir posibles errores a través de `calibratePCA9685()`, que es a lo que se reduce la ejecución del `loop()` en este modo de funcionamiento. Por otro lado, cuando `MAIN_SKETCH` está definido, es cuando tenemos la funcionalidad principal de toda la librería.

En este caso, la función `loop()`, la cual recordemos que se ejecuta en bucle repetidamente, del programa original lleva a cabo las siguientes operaciones:

- Comprobación de batería, se encarga de detener las acciones de robot cuando esta llegue a cierto umbral inferior programable.

- Lectura del entorno a través de los datos medidos por la IMU y actuación en función de ello, por ejemplo, levantándose el robot ante caídas.

- Ejecución de código propio, si así se desea, mediante la función `otherModules()`.

- Actuación del robot a los comandos recibidos, mediante la función `reaction()`, que es la principal función de toda la librería OpenCat.

Como nuestro objetivo es integrar esta librería en M2OS, necesitamos que el programa principal sea escrito en Ada, por lo que este fichero no será utilizado directamente, sino que se añadirán primeramente funciones similares a estas en el archivo principal de la librería “OpenCat.h” y después se explicará cómo se ha hecho el programa principal para su uso en M2OS.

3.1.2 Arquitectura y diseño

Ahora que hemos obtenido una visión general de cómo opera el programa principal original, procederemos a examinar las funciones fundamentales que conforman esta librería y permiten la ejecución del programa. Para lograrlo, comenzaremos explorando los ficheros de cabecera que constituyen la interfaz de la librería OpenCat:

- MemoryFree: Librería para la gestión de memoria en tiempo de ejecución, encontrada en la carpeta del mismo nombre.

- eeprom.h: Proporciona funciones de lectura y escritura con la EEPROM de Arduino, la cual es una memoria no volátil. Gracias a esta librería el robot almacena configuraciones de habilidades, calibración y otros datos importantes que son necesarias incluso cuando el robot se reinicia.

- imu.h: Librería que permite la interacción con el MPU6050, que es el micro sistema electro-mecánico (MEMS), formado por un acelerómetro y giroscopio (ambos de tres ejes), que compone la IMU de Bittle. De tal forma que el robot pueda conocer su orientación y posición para actuar, posteriormente, en base a estas.

- InstinctBittle.h: Contiene las habilidades que Bittle es capaz de realizar. Estas habilidades se almacenan en forma de matrices que contienen conjuntos de rotaciones de cada servo que se ejecutan a intervalos de tiempo regulares. Así, se procesan las matrices para “convertirlas” en acciones, en función de su tamaño y valores. Posteriormente se explican más a fondo (Véase el apartado 3.1.3.2).

- InstinctNybble.h: Contiene las habilidades del otro robot de Petoi con forma de gato, Nybble.

- io.h: Manejo de la entrada de datos desde diferentes fuentes, como comunicación serie e infrarrojos.

- motion.h: Control respecto al movimiento y equilibrio del robot. Sus funciones y parámetros permiten ajustar los servomotores del robot para alcanzar ciertos ángulos y velocidades con el objetivo de llevar a cabo las acciones y gestos de Bittle.

- OpenCat.h: Es el fichero de cabecera principal de todo el conjunto que componen esta librería. Contiene diversos datos y constantes necesarios para el funcionamiento adecuado del resto de la librería, incluye los demás ficheros de cabecera y además es donde he situado el código extraído del programa principal (OpenCat.ino) con sus pertinentes modificaciones, que será después llamado desde Ada.

- PCA9685servo.h: Permite la calibración de los servos del robot, escritura de ángulos y configuración de parámetros, orientada a la preparación de los servomotores más que a la ejecución de gestos y acciones.

- randomMind.h: Permite la ejecución aleatoria de entre un conjunto de gestos y acciones predeterminados.

- reaction.h: Encargada de actuar ante los comandos recibidos, ya sea de forma literal o mediante alguna clase de comunicación (serie, infrarrojos) para llevar a cabo la acción o gesto indicado por dicho comando.

- skill.h: Incluye funciones para copiar datos desde y hacia la memoria EEPROM, cargando movimientos y realizando operaciones físicas según los datos de las habilidades definidas en “InstinctBittle.h”. También cuenta con otros detalles, como la posibilidad de reiniciar las posiciones iniciales de los servos.

- sound.h: Permite la generación de sonidos y melodías a través del zumbador localizado en la placa de Bittle.

- taskQueue.h: Permite la creación y manipulación de listas de elementos cualesquiera, proporcionando operaciones básicas de una lista como agregar, eliminar, obtener el tamaño de la lista, etc. El resto de su código se encuentra en la carpeta “QList”.

De forma adicional, OpenCat incluye una serie de ficheros de cabecera que requieren el uso de hardware adicional no incluido con el robot Bittle para su funcionamiento:

- camera.h: Uso de un sensor de visión para la detección de objetos y control del movimiento del robot en caso de detección a través de funciones que permiten la configuración y lectura de datos del sensor, así como para realizar acciones específicas.

- doubleLight.h: Control de acciones en función de una lectura comparativa entre dos sensores de luz.

- doubleTouch.h: Similar a la anterior con la diferencia de que se usan 2 sensores táctiles.

- gesture.h: Permite la lectura de gestos, colores y proximidad en placas arduino con un sensor conectado mediante I2C. El resto de su código se encuentra en la carpeta “gesture9960”.

- infrared.h: Hace uso de la librería de arduino “IRemote.h” para recibir señales infrarrojas, en este caso, desde el control remoto incluido con Bittle, y traducirlas en acciones específicas para el control del robot a distancia.

- ultrasonic.h: Gestión de LEDs RGB y uso de ultrasonidos para medir distancias. El resto de su código se encuentra en la carpeta “rgbUltrasonic”.

- voiceLD3320.h: Parece ser una librería que permite el reconocimiento de voz, sin embargo, solo es apta para reconocer el habla china, por lo que poco uso le podemos dar en nuestro contexto. El resto de su código se encuentra en la carpeta “ld3320_ch”.

3.1.3 Código clave

Vista la estructura del programa principal y los ficheros de cabecera que componen la totalidad de la librería OpenCat, es necesario destacar que no todas estas cabeceras serán utilizadas en la demostración final, ya sea por optimización de memoria, dependencia de componentes hardware adicionales u otros motivos. Por ello, en esta sección procuraremos explicar las funciones y piezas de código fundamentales para el funcionamiento básico de Bittle.

En el programa principal de Arduino se ha visto como la ejecución se basa en la función `setup()`, que contiene el arranque del robot, y en la función `loop()`, que contiene la ejecución de órdenes dadas y de excepciones. Así que intentaremos responder a las siguientes preguntas para entender el funcionamiento general de este robot:

- ¿Cómo se inicializa el robot?
- ¿Cómo funcionan las matrices de habilidades?
- ¿Cómo se le mandan órdenes?
- ¿Cómo las ejecuta?

3.1.3.1 Inicialización de Bittle

El arranque del robot se basa principalmente en la función `initRobot()`. Esta función se encarga, principalmente, de preparar la IMU y los servomotores, entre otras cosas.

La IMU, como ya se ha comentado previamente, es el dispositivo que cuenta con un acelerómetro y un giroscopio que permite medir el movimiento y la orientación del robot en el espacio tridimensional. La función encargada de la preparación de la unidad de medición inercial se llama `imuSetup()`, y esto lo consigue llevando a cabo ciertas operaciones, como la inicialización de la MPU6050, verificación de la conexión correcta a ella, inicialización y activación del DMP (Digital Motion Processor) que se encarga de realizar cálculos complejos dentro de la propia IMU para obtener información sobre el movimiento y orientación del dispositivo; carga de offsets para prevenir errores en los sensores a petición del usuario, etc.

De forma análoga, la función `servoSetup()` lleva a cabo la puesta en marcha de los servos de Bittle mediante la inicialización de calibraciones en cada uno de los servomotores, preparación de la librería PWM, que es la encargada de manejar los servomotores a través del controlador PWM de la placa; configuración de la frecuencia PWM para todos los servomotores de Bittle (16 servomotores en total, número representado con la constante `DOF` ya mencionada en el apartado 2.1.2), etc.

Por otro lado, `initRobot()`, también se encarga de almacenar las habilidades, en forma de matriz, que Bittle podrá realizar durante la ejecución a través de la función `assignSkillAddressToOnboardEeprom()`. Para ello es necesario eliminar la

constante `MAIN_SKETCH`, de tal manera que Bittle solo ejecute la parte de configuración del código, guardando así las habilidades contenidas en el array de punteros `progmemPointer[]` en la EEPROM de 8KB, y ahorrando así espacio en la memoria Flash y RAM.

Así, `initRobot()` se encarga de establecer las configuraciones iniciales para el funcionamiento de Bittle, incluyendo la inicialización de sensores, servomotores y otras operaciones para poner el sistema en marcha.

3.1.3.2 Habilidades de Bittle

En el apartado previo hemos comentado como Bittle guarda sus habilidades en su memoria EEPROM y como estas vienen en forma de matriz. En este otro, detallaremos cómo funcionan las matrices exactamente, esto es, que representa la posición dentro de cada índice del array, los valores que contienen y el tamaño del mismo. [10]

Lo primero que debemos saber es que existen 3 tipos de habilidades:

- Gaits o Marchas (Acciones de marcha como andar, arrastrarse, ir hacia atrás, etc.).
- Postures o Posturas (Poses estáticas como sentarse, tumbarse, etc.).
- Behaviours o Comportamientos (Acciones complejas, gestos como buscar algo, saludar con la pata, levantarse del suelo, etc.).

Las matrices serán ligeramente distintas dependiendo del tipo de habilidad del que estemos hablando. No obstante, es cierto que poseen características comunes. Veamos un par de ejemplos:

Tenemos 2 habilidades, `rest` (descanso, una postura o “posture”) y `crF` (crawl forward, arrastrarse hacia adelante, que es una marcha o “gait”), que presentan la siguiente estructura:

```
const int8_t rest[] PROGMEM = {
1, 0, 0, 1,
 -30, -80, -45, 0, -3, -3, 3, 3, 75, 75, 75, 75, -55, -55, -55, -55,};

const int8_t crF[] PROGMEM = {
34, 0, 2, 1,
  42,  73,  83,  75, -43, -42, -49, -41,
  37,  75,  78,  77, -41, -41, -50, -41,
  . . .
  51,  70,  89,  71, -46, -42, -48, -42,
  46,  73,  84,  74, -45, -42, -49, -41,
};
```

El significado de estas estructuras de datos se muestra en la Imagen 10:

	Total # of Frames	Expected Body Orientation		Angle Ratio	Indexed Joint Angles															
		Roll	Pitch		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
rest	1	0	0	1	-30	-80	-45	0	-3	-3	3	3	60	60	-60	-60	-45	-45	45	45
crF	36	0	-3	1									61	68	54	61	-26	-39	-13	-26
													...	...	...	...	...	...	...	...
													60	70	53	62	-26	-39	-13	-26

Imagen 10. Estructura de las habilidades

Vemos como las matrices poseen una primera fila con cuatro valores, que representan ciertos parámetros (primeras cuatro columnas en la Imagen 10 → `1, 0, 0, 1`, En el caso de `rest`).

El primero de ellos indica el número de frames que contiene la habilidad. Un frame no es nada más que un conjunto de ángulos en los que los servomotores, que hacen de articulaciones, deben rotar en grados. Por ejemplo, en el caso de `rest`, el servo con índice 8, que representa el hombro izquierdo de Bittle, tiene que rotar hasta alcanzar un ángulo de 60 grados respecto de un punto de referencia de que tienen los servomotores, en el que estos se colocan cuando se ejecuta la habilidad `calib` (calibración). En el caso de la rotación de las patas, cuando el ángulo es positivo, la rotación se hace en sentido contrario a las agujas del reloj, mientras que, si es negativo, se hacen en el mismo sentido que las agujas del reloj teniendo como referencia la parte izquierda del robot (es decir, viéndolo desde la izquierda). Mientras que, si hablamos de la rotación de la cabeza, la referencia se toma mirando la parte superior de la cabeza del robot, permaneciendo la relación entre el signo del ángulo y el sentido de rotación igual.

En el caso de la habilidad `rest`, solo existe un frame dado que, al ser una pose estática, solo es necesario un movimiento de los motores para imitar dicha postura. Por otro lado, `crF` al ser una marcha posee varios frames (en este caso, 34). Cabe destacar que, si este primer valor es negativo, la habilidad que contiene el array es un comportamiento, siendo una marcha o postura en caso de ser positivo.

El segundo y tercer elementos de la matriz representan el valor esperado de la dirección del cuerpo, es decir, el ángulo de inclinación del cuerpo del robot al realizar habilidades, correspondientes al ángulo de balanceo (Roll) y al ángulo de cabeceo (Pitch) respectivamente, en grados. Cuando el robot realiza una habilidad, pero el cuerpo de este no se encuentra en el grado de inclinación esperado, el algoritmo de equilibrio ajustará los valores de los ángulos de los servos correspondientes para mantener la inclinación lo más cerca posible del valor esperado.

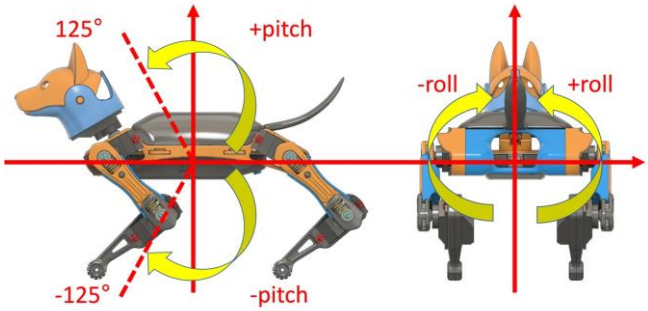

Imagen 11. Sentidos del cabeceo y balanceo

Finalmente, el cuarto elemento de la primera fila de la matriz es un factor de ángulo. Este factor de ángulo puede ser incrementado cuando sea necesario guardar valores fuera del rango (-128, 127) de ángulos. Por ejemplo, si este elemento vale 2 en una habilidad cualquiera, todos los ángulos de cada frame que contenga esa habilidad se multiplicarán por el valor de este elemento, en este caso, por 2; siendo el resultado de esta operación los verdaderos grados de rotación que realizarán los servomotores.

Conociendo las características comunes de los arrays de habilidades, destaquemos ahora las diferencias entre los 3 tipos de habilidades explicados:

- Como se ha mencionado previamente, las posturas solo tienen un frame de acción.

- Las habilidades de tipo “gait” contienen diversos frames coherentes entre sí que se repiten continuamente en un bucle secuencial, imitando así los movimientos que realiza un animal real al moverse. Este bucle no se detendrá hasta que el robot reciba una nueva orden o se apague.

  Los frames de matrices de marchas solo contienen 8 elementos, que se enumeran de menor a mayor, desde el 8 hasta el 15, lo que se traduce en el caso de Bittle, en que las habilidades de marcha no incluyen rotación de la cabeza.

- Los comportamientos también contienen una serie de frames coherentes entre sí que permiten la ejecución de distintos gestos. Todos estos frames se ejecutan, a priori, una sola vez, como suceden en las otras habilidades, sin embargo, se definen algunos subconjuntos de frames que se ejecutarán en bucle un número determinado de veces antes de volver a la ejecución normal. Veamos un ejemplo de una habilidad de tipo comportamiento en la que Bittle hace flexiones:

```
const int8_t pu[] PROGMEM = {
-10, 0, 0, 1,
 7, 8, 3,
  0, 0, 0, 0, 0, 0, 0, 0, 30, 30, 30, 30, 30, 30, 30, 30,     8, 0, 0, 0,
  15, 0, 0, 0, 0, 0, 0, 0, 30, 35, 40, 21, 50, 15, 15, 41,    12, 0, 0, 0,
```

```
  . . .
  0, 0, 0, 0, 0, 0, 0, 0, 30, 30, 110, 110, 60, 60, 60, 60,  12, 1, 0, 0,
  30, 0, 0, 0, 0, 0, 0, 0, 70, 70, 85, 85, -50, -50, 60, 60, 16, 0, 0, 0,
  0, 0, 0, 0, 0, 0, 0, 0, 30, 30, 30, 30, 30, 30, 30, 30,    8, 0, 0, 0,
};
```

Se puede observar cómo esta estructura de datos contiene más información que las posturas y marchas.

Los 4 primeros elementos de la primera fila son los ya explicados (Véase como el primer elemento es negativo, indicando que la habilidad es de tipo comportamiento), sin embargo, aparece una segunda fila con 3 elementos. Estos valores en la fila 2 indican el bucle interno contenido dentro del comportamiento: frame inicial, frame final y número de iteraciones.

En este ejemplo de la habilidad llamada `pu`, 7, 8 y 3 quieren decir que el comportamiento contiene un bucle interno que ejecuta desde el frame 7 hasta el 8, 3 veces seguidas, antes de seguir con los frames restantes.

Por otra parte, los frames contienen 4 valores más que los que contienen las posturas. Los 16 primeros elementos representan lo mismo, pero los 4 nuevos elementos representan lo siguiente:

- El primero representa la velocidad de actuación. Esta por defecto es 4, pero puede cambiarse dentro del rango 1-127 de más lento a más rápido. Las unidades son grados por paso.

- El segundo representa el tiempo de retardo. Por defecto a valor 0 y también puede ajustarse dentro del mismo rango que el elemento anterior. La unidad es 50 milisegundos, lo que quiere decir que, si tenemos un valor de 2 en este elemento, tendremos un tiempo de retardo de 100 milisegundos.

- El tercer elemento representa el eje de disparo, que es la dirección de rotación del cuerpo cuando el robot activa el siguiente frame de acción. Rango de valores entre -2 y 2.

- Finalmente, el cuarto y último elemento representa el ángulo de disparo. Los valores de ángulo tienen los mismos significados positivos y negativos que las expectativas de orientación del cuerpo. Rango entre -125 y 125 grados.

A continuación, incluyo en la Imagen 12, una tabla con todas las habilidades desarrolladas por Petoi:

Skills that can be called by 'k' token (kbdF, kbk, etc.)		
	Abbreviation	**Skill Name**
posture	balance	stand up neutral
	buttUp	butt up
	calib	calibration pose
	dropped	dropped by back legs
	lifted	lifted by neck
	lnd	landing pose
	rest	rest
	sit	sit
	str	stretch
	up	stand up neutral (= balance)
	zero	all joint at 0 degrees
gait	bdF	bound forward
	bk	backward
	bkL	backward Left
	crF	crawl Forward
	crL	crawl Left
	gpF	gap Forward
	gpL	gap Left
	hlw	halloween gait
	jpF	jump Forward
	phF	push Forward
	phL	push Left
	trF	trot Forward
	trL	trot Left
	vtF	step at origin
	vtL	sping left
	wkF	walk Forward
	wkL	walk Left
behavior	ang	angry
	bf	backflip
	bx	boxing
	chr	cheers
	ck	check
	cmh	come here
	dg	dig
	ff	front flip
	fiv	high five
	gdb	good boy
	hds	handstand
	hg	hug
	hi	hi
	hsk	hand shake
	hu	hands up
	jmp	jump
	kc	kick
	lpov	leap over
	mw	moon walk
	nd	nod
	pd	play dead
	pee	pee
	pu	push ups
	pu1	push ups with on hand
	rc	recover
	rl	roll
	scrh	scratch
	snf	sniff
	tbl	be table
	ts	test
	wh	wave head
	zz	all joint at 0 degrees
When callling a skill, you may specify its direction by adding a suffix -L, -R, or -X		
The rightward skill is mirrored from the leftward skill		
	L	left
	R	right
	X	random direction

Imagen 12. Listado de habilidades desarrolladas por Petoi

3.1.3.3 Mandato de acciones a Bittle

Aunque existen diferentes formas de enviar órdenes a Bittle gracias al uso de diferentes dispositivos, permitiendo comunicación infrarroja, por voz o incluso por gestos. En este proyecto nos hemos limitado al mandato de órdenes explícito a través del propio código de OpenCat.

Para ello, la acción a realizar por Bittle se configura dando valor a 3 variables globales definidas en el fichero "OpenCat.h": `newCmd`, `newCmdIdx` y `token`.

Desde el punto de vista del usuario, con conocer mínimamente el propósito de cada una de estas variables, estaríamos preparados para comandar a Bittle:

- `newCmd`: Se trata del comando que queremos llevar a cabo, existe una lista variada de ellos, que puedes consultar en el fichero "InstinctBittle.h", en la Imagen 12 (la tabla de habilidades) o en la página oficial de Petoi. Ejemplos de estas habilidades son `sit`, `rest`, `wkF`, etc. Normalmente las acciones con nombres más largos se abrevian ("`wkF`" → walk forward, o "crL" → crawl left) para que los nombres de todas las habilidades tengan más o menos el mismo número de caracteres.

- `newCmdIdx`: Aunque sea una variable de tipo entero, es principalmente usado como un booleano, donde solamente necesitamos que valga distinto de 0 para contemplar más condiciones respecto a la habilidad que se está procesando, obteniendo así una funcionalidad más completa (Véase la función `reaction()` en el fichero "reaction.h").

- `token`: Se trata de un carácter que delimita ciertos condicionales dentro del código encargado de ejecutar acciones, y existen unos cuantos tipos predefinidos (presentes en el fichero OpenCat.h). Estos tokens desempeñan diversos papeles, como `T_CALIBRATE` que coloca al robot en una pose de calibración para arreglos puntuales como acoplar patas o ajustar los ángulos de las articulaciones. O `T_RANDOM_MIND` que activa la ejecución aleatoria de habilidades dentro de un conjunto predefinido.

  Sin embargo, para el caso de ejecución de habilidades estándar solo necesitamos asignar el valor `'k'` (representado con la constante `T_SKILL`), pudiéndonos olvidar del resto.

Concluyendo este apartado, se puede apreciar que el proceso de dictar acciones para que el robot las lleve a cabo no tiene ningún misterio, basta con indicar la nueva acción a ejecutar en `newCmd`, fijar `newCmdIdx` a 1 (o cualquier valor distinto de 0) y token al valor `T_SKILL`.

3.1.3.4 Ejecución de habilidades

Habiendo visto el resto de apartados, solo nos falta conocer como Bittle lleva a cabo las operaciones dictadas en los arrays de habilidades. La función encargada de ello es

`reaction()` y se encuentra ubicada en el fichero de cabecera del mismo nombre. Dada la densidad del código de la función, su relativa complejidad y, en mi opinión, su falta de comentarios, voy a tratar de explicarla con mis propias palabras, en vez de mostrarla directamente.

Esta función verifica los valores de las 3 variables comentadas en el apartado anterior siguiendo un orden. Primero se comprueba si el valor de `newCmdIdx` es distinto de 0, hecho que en nuestro caso siempre se va a dar, ya que queremos la funcionalidad completa del código. A continuación, entramos en una cláusula switch-case donde se comprueba el valor de la variable `token` para actuar en consecuencia. Una vez más, nosotros siempre vamos a establecer esta variable al valor `T_SKILL ('k')`, que es el que da a entender a Bittle que se quiere ejecutar una habilidad.

En el segmento correspondiente al caso en el que `token` es igual a `T_SKILL`, se carga el frame correspondiente de la habilidad que corresponda según el valor de la última variable mencionada previamente, `newCmd`.

Finalizando la función `reaction()`, se llama a la función `perform()` que es la encargada de profundizar en detalles más técnicos, como el tipo de habilidad que se va a realizar, distinguiendo entre comportamientos, por un lado, y posturas y marchas por el otro. De esta manera, y con diversos matices, se llaman más funciones que ayudan a realizar las habilidades, acabando todas ellas en la función de "más bajo nivel" escrita por los desarrolladores de OpenCat: `calibratedPWM()`.

Esta función, recibe como parámetros el índice del servomotor al que le toca actuar, el ángulo que tiene que rotar y la velocidad a la que lo hace. De esta forma es como se van comandando a los servomotores uno a uno para acabar imitando una acción fiel a la realidad.

Creo que es relevante mencionar, que en el análisis de esta función principal (originalmente compuesta por casi 500 líneas de código) se ha encontrado una cierta complejidad y falta de claridad en la estructura del código. Esta complejidad se ha debido principalmente a la presencia de numerosas macros de compilación condicional y la falta de comentarios descriptivos en ciertas secciones del código. Esta práctica ha dificultado la comprensión y creo que puede suponer un problema a futuro, dada la importancia de la legibilidad y claridad del código para facilitar su entendimiento no solo a terceras personas, sino para el propio desarrollador original.

Ahora que conocemos lo suficiente a la librería OpenCat, el siguiente paso será adaptar esta librería para su uso en M2OS.

3.2 Compilación para M2OS

La integración de la librería OpenCat, desarrollada en C/C++, en el entorno Ada mediante M2OS se logra a través de una herramienta específica proporcionada por M2OS. Esta herramienta, presentada en forma de un Makefile ubicado en la ruta

"arch/arduino_uno/drivers/libcore" dentro de la estructura principal de M2OS, posibilita la incorporación de librerías C/C++ en el contexto de Ada y M2OS.

El propósito de este Makefile es generar una librería denominada "lib_core_arduino.a," que contiene funciones de librerías Arduino (escritas en C/C++), con el objetivo final de ser vinculada con M2OS. Esta herramienta admite varias librerías y contiene condiciones que determinan aspectos específicos en la fase de compilación.

Las características más resaltables de este Makefile incluyen:

- Se definen los nombres de las librerías destino, que son:
    - `TARGET_LIB_ZOWI: lib_core_arduino_zowi.a`
    - `TARGET_LIB_EVSHIELD: lib_core_arduino_evshield.a`
    - `TARGET_LIB: lib_core_arduino.a`

    Para mi proyecto, nos centraremos únicamente en la última librería ya que las otras 2 no tienen relevancia para mi caso, siendo la primera utilizada con el robot bípedo Zowi (un robot educativo para niños), y la segunda para una tarjeta adaptadora que permite el uso de motores Lego desde placas Arduino.

- El Makefile verifica la disponibilidad del comando "arduino" en el sistema. Si dicho comando no se encuentra, se muestra un mensaje informativo, y la compilación de la librería se omite. Esto no representa un error fatal, ya que se utilizará una versión precompilada de "lib_core_arduino.a" que se incluye en la distribución de M2OS.

- Se definen algunas variables importantes para el proceso, como `ARDUINO_LIB_PATH` (ruta de las librerías de Arduino instaladas en el sistema), `LIBS_ALL` (las librerías que se intentarán buscar para su compilación) y `BUILD_PATH` (ruta donde se compilarán los archivos).

- Se especifican algunas librerías compatibles con M2OS, entre las que se encuentran "Zowi", "DTH11" (sensor de temperatura y humedad), "Grove-16x2_LCD_Series" (pantalla LCD de 16x2 para Grove), "VoiceRecognitionV3" (reconocimiento de voz), entre otras.

- El Makefile verifica qué librerías están instaladas en el sistema y cuáles no. Además, maneja algunas incompatibilidades con librerías específicas como "SoftwareSerial.h" y "Servo.h".

- El objetivo `all` depende de `build_library`, que se encarga de generar la librería "lib_core_arduino.a." Primero, se incluyen las librerías que se han encontrado en el sistema, de entre las listadas en `LIBS_ALL`, en un sketch “lib_generator.ino”. Este sketch es compilado con el comando “arduino” para forzar la compilación de todas sus dependencias, para que, a continuación, el

Makefile empaquete todos los ficheros objeto en esta librería “lib_core_arduino.a”.

- Adicionalmente, se ha definido el objetivo `clean` para eliminar los archivos generados durante el proceso de compilación.

En resumen, este Makefile verifica la disponibilidad del comando "arduino" y procede a crear una librería que contiene funciones específicas de Arduino, así como algunas librerías adicionales compatibles con M2OS que pueden ser utilizadas en el desarrollo de proyectos.

Para preparar la librería “lib_core_arduino.a” para nuestro proyecto, hemos tenido que llevar a cabo los siguientes pasos:

- Colocar la librería OpenCat en el directorio por defecto de localización de las librerías Arduino: “$(HOME)/Arduino/libraries/”, definido en la variable `ARDUINO_LIB_PATH`.

- Modificaciones en el Makefile:

  - Inclusión de la librería OpenCat en la variable `LIBS_ALL`. Para ello se han creado unos ficheros adicionales “OpenCatM2.h” y “OpenCatM2.cpp”, de tal manera que OpenCat encaje de manera más adecuada con el modo en que funciona este Makefile. Todas las demás librerías soportadas por este fichero cuentan con una versión “.h” y “.cpp” del mismo nombre.

  - Gestión de errores con la librería “Servo.h”. Este Makefile se encarga de añadir de forma manual, ciertas librerías básicas de Arduino en la librería estática final, siendo una de ellos “Servo.h”. Sin embargo, OpenCat, incluye por sí misma este fichero de cabecera, por lo que, para evitar posibles problemas de doble definición, se suprime la inclusión de esta librería, para el control de servos, en el fichero Makefile.

- Abrir una terminal en el directorio "arch/arduino_uno/drivers/libcore" dentro de la estructura principal de M2OS, y ejecutar el comando `make` para obtener la librería estática que incluye OpenCat lista para su uso.

Ahora que comprendemos cómo se empaqueta la librería OpenCat, podemos proceder a integrar esta nueva librería en el proceso de enlace de nuestro programa Ada. Esto se logra añadiendo explícitamente "lib_core_arduino.a" en la sección de enlace dentro del código fuente de nuestro proyecto Ada. En mi caso específico, en el archivo “bittle_m2os_arduino_uno.gpr” ubicado en “examples/api_m2os/arduino_uno/bittle”.

Con esta configuración la próxima vez que compilemos un proyecto que contenga alguna función almacenada en esta librería, esta se podrá llamar correctamente, extrayéndose las funciones mínimas necesarias para ello.

4. Adaptación de OpenCat a M2OS

4.1 Capa de adaptación

Hasta el momento hemos sido capaces de incorporar la funcionalidad de OpenCat a nuestra librería “lib_core_arduino.a”, no obstante, necesitamos conseguir que desde la parte de Ada se tomen estas funciones de alguna manera para su uso.

Como se ha comentado en la sección anterior, esta librería se incluye en la sección de linkado del proyecto Ada para que este pueda servirse de las funciones pertinentes. Pero necesitamos un paso adicional, una capa de interfaz que haga posible llamar a las funciones C/C++ desde el código Ada escrito por el usuario. Para ello, entra en juego el `pragma Import` de Ada, que es una característica de este lenguaje que permite la interoperabilidad con otros lenguajes, principalmente con C/C++. Gracias a esta propiedad de Ada, se pueden utilizar funciones escritas en este lenguaje como si fueran nativas de Ada, lo que facilita el proceso de integración entre piezas de código escritas en distintos lenguajes.

A continuación, se muestra un ejemplo de utilización de este pragma, recogido del fichero “arch/arduino_uno/drivers/libcore/arduino-bittle.ads”:

```
procedure Setup
     with Import, Convention => CPP,
     External_Name => "_Z8bitsetupv";
```

Podemos descomponer esta pieza de código en las siguientes partes:

- `procedure Setup`: Indica el procedimiento que se está definiendo el cuál será usado en nuestro código Ada. Si la función importada recibe parámetros, se añadirán en esta línea parámetros de entrada compatibles. (Por ejemplo, `procedure Action (command : String)` para una función C++ que recibe un array de caracteres como argumento)

- `with Import`: La palabra clave `with`, en este contexto, se utiliza para proporcionar más información sobre cómo se debe tratar este procedimiento. En este caso, al usar `Import` se especifica que el cuerpo de este procedimiento está siendo importado desde una fuente externa.

- `Convention => CPP`: Especifica que el lenguaje en el que está escrito la función importada es C++, en este caso. Esto es necesario, dado que las convenciones determinan como se pasan los parámetros y como se gestionan las llamadas a estas funciones importadas.

- `External_Name => "_Z8bitsetupv"`: Esta es la parte fundamental del `Import`. Aquí se especifica el nombre de la función escrita en C++. En este ejemplo, la función se llama `bitsetup()` en C++, sin embargo, se le añaden caracteres extra que son parte de una convención específica utilizada en C++, relacionada con la firma de la función y otros detalles internos del compilador C++, llamada

"name mangling" en inglés, que se podría traducir por "decoración de nombres" [11].

Gracias a este bloque de código, ahora podremos llamar indirectamente a la función `bitsetup()` de C++, a través de este procedimiento `Setup` de Ada.

4.2 Funciones importadas desde C++

En el subapartado anterior, mencionamos como ejemplo a la función `bitsetup()`, de la cual no habíamos hablando en ningún momento, y es que, como se comentó previamente, no podemos usar las funciones del archivo original Arduino para su ejecución desde Ada, por lo que en "OpenCat.h" se han definido una serie de funciones que serán importadas desde Ada en su lugar:

- `bitsetup()`: Esta función es prácticamente una copia exacta de la función `setup()` original de Arduino y solo se llamará una vez en la ejecución final de Ada. Contiene lo necesario para "arrancar" el robot, incluyendo la puesta en marcha de la IMU y servomotores (Véase el apartado 3.1.3.1, para la inicialización de Bittle).

- `bitaction()`: Recibe un parámetro en forma de `const char[]` el cual sirve para asignar un valor a la variable global `newCmd`, indicando así la habilidad que deseamos que Bittle realice. Para ello también tiene en cuenta el resto de variables globales ya explicadas para mandar órdenes (Véase el apartado 3.1.3.3, para mandato de acciones).

  Es importante mencionar que, para pasar los parámetros desde Ada, se han definido una serie de constantes de tipo `String`, compatibles con el parámetro que esta función espera recibir, para todas las habilidades que Bittle puede realizar. Esto se considera la forma más conveniente y menos propensa a problemas, en lugar de usar una variable de longitud variable.

- `checkPhysicalPosition()`: Llama a `readEnvironment()` y `dealWithExceptions()`, que son las funciones localizadas originalmente en el `loop()` de Arduino, y que se encargan de leer la IMU del robot, para verificar su orientación en el espacio y actuar en consecuencia.

- `reaction()`: La función que se encarga de ejecutar las habilidades comandadas a Bittle (Véase el apartado 3.1.3.4).

De manera adicional, se ha incluido 2 funciones que permiten la modificación de la velocidad con la que los servos rotan para llevar a cabo las distintas habilidades:

- `change_action_speed()`: Recibe como parámetro un `float` y establece la velocidad con la que se ejecutan marchas y posturas. Esto lo hace igualando una variable global del mismo tipo (`actionSpeed`) al parámetro de esta función. Después, en la función `calibratedPWM`, encargada del movimiento de los

servos, se iguala el parámetro de velocidad `speedRatio` a esta variable, para establecer un valor por defecto:

```
void calibratedPWM(byte i, float angle, float speedRatio =
actionSpeed)
```

- `change_transform_speed()`: Exactamente el mismo funcionamiento, solo que modifica la velocidad para comportamientos. Para ello, se define otra variable global `transformationSpeed`, la cual será usará en la función `transform`, utilizada para la ejecución de los parámetros, igualando su parámetro de velocidad, también llamado `speedRatio`, a esta variable global, para darle un valor por defecto:

```
void transform(T *target, byte angleDataRatio = 1, float
speedRatio = transformationSpeed, byte offset = 0)
```

Estas dos últimas funciones no son estrictamente necesarias para el funcionamiento de Bittle, ya que las variables con las que operan ya tienen valores iniciales predefinidos. Sin embargo, se han incluido para brindar flexibilidad y mejorar la calidad del código.

Todas estas funciones serán las que importaremos desde Ada, gracias al uso del `pragma Import`. Tanto estas, como las constantes de tipo `String` para las habilidades, se encuentran en el paquete Arduino.Bittle (fichero "arduino-bittle.ads").

4.3 Organización en tareas

La organización en tareas de OpenCat tendrá una base común para cualquier programa que haga uso de esta librería en M2OS. Para la implementación de la funcionalidad de esta librería en M2OS, y sacar así partido de la concurrencia que este sistema operativo proporciona, se han definido dos tareas que recogen el uso básico de OpenCat (los paquetes de estas tareas se ubican en “arch/arduino_uno/drivers/libcore” dentro de la carpeta de M2OS):

- La primera tarea (`Bittle_Check`) se encarga de verificar la orientación física del robot, y actuar ante posibles caídas. Para ello, incluye en su función de ejecución en bucle una llamada al procedimiento `Check_Physical_Position`, que importa la función `checkPhysicalPosition()` definida en OpenCat y explicada en el apartado 4.2:

```
procedure Check_Task_Body is
begin
            Bittle.Check_Physical_Position;
end Check_Task_Body;
```

  Esta tarea cuenta con un periodo de 2 segundos y prioridad 2.

- Por otro lado, la segunda tarea (`Bittle_Reaction`) se encarga de reaccionar ante los comandos que Bittle reciba:

```
procedure Reaction_Task_Body is
begin
   Bittle.Reaction;
end Reaction_Task_Body;
```

El procedimiento `Reaction` llama a la función `reaction()`. Esta tarea tiene un periodo de 0 segundos y prioridad 1, lo que se traduce en que la tarea se ejecutará de fondo de forma indefinida, siempre y cuando ninguna tarea de mayor prioridad esté lista para ejecutarse (es decir, mientras no haya completado su periodo).

Para mejorar la modularidad del código, se ha optado por definir cada una de estas tareas en un paquete diferente. La especificiación del paquete (fichero ".ads") contiene únicamente la definición del paquete y un pragma que indica que debe elaborarse el cuerpo del paquete (fichero ".adb" del mismo nombre). Por ejemplo, la especificación del paquete Bittle_Reaction es la mostrada a continuación:

```
package Arduino.Bittle_Reaction is
   pragma Elaborate_Body;
end Arduino.Bittle_Reaction;
```

Por otro lado, el fichero “.adb” contiene la implementación de la funcionalidad que la tarea va a desempeñar. Este fichero incluye una función de inicialización de una sola ejecución, y otra función que se repite constantemente en bucle, análogo a la estructura de los programas Arduino:

```
with M2.Direct_IO;
with AdaX_Dispatching_Stack_Sharing.Periodic_Task;
with Ada.Real_Time;
with Arduino.Bittle;

package body Arduino.Bittle_Reaction is
   package DIO renames M2.Direct_IO;
   package Bittle renames Arduino.Bittle;


   Reaction_Task_Prio : constant := 1;
   Reaction_Task_T_In_Ms : constant := 0;

   ----------------------
   -- Reaction_Task_Init --
   ----------------------

   procedure Reaction_Task_Init is -- Una sola ejecución
   begin
```

```
      DIO.Put_Line("Reaction"); -- Imprime por el monitor serie
   end Reaction_Task_Init;

   ---------------------
   -- Reaction_Task_Body --
   ---------------------

   procedure Reaction_Task_Body is -- Ejecución en bucle
   begin
      Bittle.Reaction; -- Llama a la función reaction() de OpenCat
   end Reaction_Task_Body;

   ----------------
   -- Reaction_Task --
   ----------------

   package Reaction_Task is new
   AdaX_Dispatching_Stack_Sharing.Periodic_Task
     (Init_Ac  => Reaction_Task_Init'Access,
      Body_Ac  => Reaction_Task_Body'Access,
      Priority => Reaction_Task_Prio,
      Period   => Ada.Real_Time.Milliseconds (Reaction_Task_T_In_Ms));
end Arduino.Bittle_Reaction;
```

En este ejemplo, se han incluido los ficheros .ads y .adb pertenecientes a la tarea encargada de ejecutar la función de C/C++ `reaction()` a través del procedimiento del mismo nombre `Reaction`, en cuyo cuerpo se importa esta función. Es al final del fichero cuerpo donde se crea la tarea que se ejecutará en el programa final, indicando cuáles son sus funciones de inicialización y de ejecución en bucle, así como su prioridad y periodo de ejecución.

4.4 Ejemplo sencillo de uso

Para probar el correcto funcionamiento de OpenCat a través de M2OS, hemos preparado un programa de prueba, que haga uso de las funcionalidades más básicas de esta librería.

Para ello, se ha añadido una tercera tarea (`Demo_Commands`, ubicada en "examples/api_m2os/arduino_uno/bittle") cuyo objetivo es el mandato de acciones a Bittle, alternando, en este caso entre una habilidad de cada tipo (un total de 3 habilidades), con el objetivo de verificar que Bittle ejecuta todos los tipos de habilidad correctamente y poder observar el desempeño de cada tipo de habilidad:

```
procedure Commands_Task_Body is
begin
      if Index = 0 then
         Bittle.Action(Bittle.sit); -- Postura (Posture)
         Index := 1;
```

```
      elsif Index = 1 then
         Bittle.Action(Bittle.crF); -- Marcha (Gait)
         Index := 2;
      elsif Index = 2 then
         Bittle.Action(Bittle.hi); -- Comportamiento (Behaviour)
         Index := 0;
      end if;
end Commands_Task_Body;
```

En el cuerpo de esta tarea se ha definido una variable de tipo entero con la que se irán alternando las habilidades a ejecutar. En este caso, rotando entre sentarse (postura), arrastrarse hacia adelante (marcha) y saludando (comportamiento). El procedimiento `Action` llama a la función `bitaction()`, que hace saber a Bittle que habilidad ejecutar. Esta tarea cuenta con un periodo de 10,5 segundos, con el motivo de poder apreciar cada tipo de habilidad, y prioridad 3 (Por encima de las dos tareas básicas. La operación de esta tarea es muy corta, por lo que no es un problema que se ejecute antes que ellas).

De esta manera, ahora solo necesitamos incluir las dos tareas principales y la recién explicada en el programa principal de prueba para este ejemplo, llamado “actions_example.adb” (ubicado en “examples/api_m2os/arduino_uno/bittle”):

```
pragma Ravenscar;
pragma Task_Dispatching_Policy(FIFO_Within_Priorities);
pragma Locking_Policy(Ceiling_Locking);
pragma Queuing_Policy(Priority_Queuing);

with Arduino.Bittle;
with Arduino.Bittle_Reaction;
with Arduino.Bittle_Check;
with Actions_Example_Task;

with M2.Direct_IO;
with AdaX_Dispatching_Stack_Sharing;

procedure Actions_Example is
   package DIO renames M2.Direct_IO;
   package Bittle renames Arduino.Bittle;

begin
   Bittle.Setup;
   DIO.Put_Line("Bittle preparado");
end Actions_Example;
```

Este programa principal, el cual siempre se ejecuta antes que las tareas incluidas, llama al procedimiento `Setup` que importa la función `bitsetup()` para preparar al robot antes de que se empiecen a llamar al resto de funciones incluidas en las tareas, evitando así

errores. Por otro lado, incluye una serie de `pragmas` que establecen las políticas de planificación, orden de las tareas según prioridad, etc.

El programa principal no incluye más código relevante, ya que su propósito es incluir todas las tareas que se vayan a utilizar en la ejecución final para arrancarlas en el momento en el que termina el código de su cuerpo, dando pie a la ejecución concurrente de dichas tareas.

Finalmente, solo resta compilar y linkar el proyecto con todas sus dependencias y cargar el ejecutable final sobre la placa NyBoard, permitiendo así al robot Bittle llevar a cabo las funcionalidades descritas en estas tres tareas.

4.5 Medida del tiempo de ejecución

De manera adicional, hemos querido medir el tiempo que tarda en ejecutarse la función reaction(), dado que es la principal función de toda la librería OpenCat y creemos que puede ser relevante como dato a futuro.

Para ello, hemos utilizado las herramientas de medición temporal incluidas dentro del propio M2OS, localizadas en el paquete “measurements” (ubicado en “tests/reports”, dentro de la carpeta principal de M2OS). Este paquete incluye funciones y procedimientos de los cuales nos hemos servido para medir esta función, como `Measurement_Avg_Time_Us`, que como su propio nombre indica, devuelve la media de tiempos medidos en microsegundos, y `Measurement_Time` que devuelve la última medición en milisegundos.

Gracias a este paquete se ha determinado que el tiempo medio de ejecución de la función `reaction()` se aproxima a los 2800 microsegundos o 2,8 milisegundos. El código utilizado para llevar a cabo estas medidas se encuentra comentado tanto en el fichero “actions_example.adb” mencionado en el apartado anterior, como en el cuerpo de la tarea `Bittle_Reaction`.

5. Optimización de memoria

5.1 Por qué optimizar

En el ámbito de la robótica, bien es sabido que los recursos de los que disponen los pequeños robots diseñados para tareas específicas son altamente limitados, siendo su tamaño de memoria uno los recursos más críticos. En este contexto, la optimización de memoria se convierte en una práctica necesaria para lograr un rendimiento óptimo y ser capaces de aumentar las funcionalidades que el robot pueda llevar a cabo.

La librería estándar proporcionada por Petoi encaja perfectamente en la memoria disponible de Bittle, que consta de 32KB de Flash y 2KB de RAM. Esta librería ocupa aproximadamente 29KB en Flash y 1134B en RAM (sin contar los 8KB de EEPROM que están reservados para el guardado de habilidades y algún otro dato menor), lo que deja un margen bastante estrecho para futuras expansiones. Para usuarios que simplemente deseen darle funcionalidad básica al robot, este espacio es suficiente. Sin embargo, en nuestro caso, al incorporar M2OS, el espacio ocupado aumenta hasta un valor muy cercano a los 32KB en Flash, lo que prácticamente agota el espacio disponible para añadir más funcionalidades al proyecto.

Esto no solo es poco eficiente, sino que, además, ocupar tanta memoria puede resultar incluso peligroso, ya que ante un descuido en el que excedamos la cantidad de memoria disponible en el sistema, podemos llegar a dañar el bootloader de Arduino, como ha sido nuestro caso. Este bootloader es un pequeño programa que se encuentra precisamente en la memoria no volátil del dispositivo, cuya función principal es permitir la programación del microcontrolador. Cuando este problema sucede, la placa se ve incapaz de ser programada a través del puerto USB de manera convencional y, por lo tanto, esta se vuelve inservible.

Por suerte, existe una solución efectiva para arreglar este problema: quemar el bootloader. Existen muchas guías que enseñan cómo llevar a cabo este proceso, y por suerte, en Petoi es un escenario que han contemplado y cuentan con una sección para asistir a sus usuarios ante la aparición de este problema. [12]

5.2 Optimizaciones en OpenCat

Para evitarnos estos problemas a futuro y para poder añadir más funcionalidades a este proyecto en general, se han tomado las siguientes medidas para ahorrar espacio en la librería de OpenCat.

Primeramente, hemos eliminado de la carpeta OpenCat todos los ficheros de cabecera que no usamos, principalmente las pertenecientes a periféricos adicionales como sensores, cámaras, etc. Ejemplos de estos ficheros son “gesture.h”, “doubleLight.h”, “camera.h” o “InstinctNybble”, que se elimina por no servir para el robot Bittle. Esto no nos supone un ahorro real en memoria porque no estábamos usando ninguno de estos ficheros desde una primera instancia, pero sí que ayuda a descartar aquello que no sea

necesario, aportando mayor claridad en la librería OpenCat a nivel general y dejando exclusivamente lo único y necesario para el funcionamiento del robot en nuestro caso.

Ahora sí, los cambios que nos suponen un ahorro de memoria son los siguientes:

- Se elimina la librería “sound.h”, encargada de emitir sonidos a través del zumbador. El robot originalmente emite algunos sonidos durante su inicialización principalmente, pero dado el ahorro que nos aporta y que no juega un papel fundamental en la ejecución del programa, queda descartada.

    - Nos brinda un ahorro de 2628 Bytes en Flash y 17 Bytes en RAM.

- Eliminamos la librería “infrared.h”, que se encarga originalmente del procesamiento de los infrarrojos enviados por el mando a distancia incluido con Bittle ya que no será necesaria para nuestra demostración.

    - Ahorramos 776 Bytes en Flash y 217 Bytes en RAM.

- El fichero de cabecera “reaction.h” es uno de los que más espacio ocupan en memoria, concretamente la función, del mismo nombre, `reaction()` que ocupa originalmente ~10KB (casi 1/3 de toda la memoria disponible en el sistema). Obviamente, no podemos eliminar esta librería porque contiene la función principal de todo OpenCat, así que necesitamos eliminar bloques que no sean necesarios, de tal forma que mantengamos la funcionalidad principal, que es la ejecución de habilidades, mientras ahorramos en memoria.

  En el apartado 3.1.3.3, sobre el mandato de las habilidades a Bittle, se ha explicado como la variable `token` debía tener un valor concreto `'k'`, definido por la constante `T_SKILL`, y como existía una serie de valores para token predefinidos los cuales se pueden ver en la Imagen 13:

Token (refer to OpenCat.h and reaction.h)	ASCII	Function and format
T_QUERY	'?'	get the model and software version
T_ABORT	'a'	abort the calibration values
T_BEEP	'b'	beep. b note1 duration1 note2 duration2 ... e.g. b12 8 14 8 16 8 17 8 19 4
T_BEEP_BIN		beep. B note1 duration1 note2 duration2 ... e.g. B12 8 14 8 16 8 17 8 19 4. A single B will toggle the melody on/off
T_CALIBRATE	'c'	send the robot to calibration posture for attaching legs and fine-tuning the joint offsets. E.g. c jointIndex1 offset1 jointIndex2 offset2 ... e.g. c0 7 1 -4
T_REST	'd'	turn the robot to rest posture and shut down the servos
T_INDEXED_SIMULTANEOUS_ASC	'i'	i jointIndex1 jointAngle1 jointIndex2 jointAngle2 ... e.g. i0 70 8 -20 9 -20. A single 'i' will free the head joints if it were previously manually controlled.
T_INDEXED_SIMULTANEOUS_BIN		I jointIndex1 jointAngle1 jointIndex2 jointAngle2 ... e.g. I0 70 8 -20 9 -20
T_JOINTS	'j'	A single "j" returns all angles. "j index" prints the joint's angle. e.g. "j 8" or "j11".
T_SKILL	'k'	call skills
T_SKILL_DATA		send the complete skill data as a binary char array
T_LISTED_BIN		a list of the DOFx joint angles: angle0 angle1 angle2 ... angle15
T_INDEXED_SEQUENTIAL_ASC	'm'	m jointIndex1 jointAngle1 jointIndex2 jointAngle2 ... e.g. m0 70 0 -70 8 -20 9 -20
T_INDEXED_SEQUENTIAL_BIN		M jointIndex1 jointAngle1 jointIndex2 jointAngle2 ... e.g. M0 70 0 -70 8 -20 9 -20
T_PAUSE	'p'	pause the movement and shut down the servos
T_SAVE	's'	save the calibration values
T_TEMP	'T'	call the last skill data received from the serial port
T_XLEG	'x'	
T_RANDOM_MIND	'z'	toggle random behaviors
T_READ		read pin R
T_WRITE		write pin W
TYPE_ANALOG	'a'	Ra(analog read) Wa(analog write)
TYPE_DIGITAL	'd'	Rd(digital read) Wd(digital write)
T_COLOR		change the eye colors of the RGB ultrasonic sensor \ a single 'C' will cancel the manual eye colors
T_TASK_QUEUE	'q'	
T_PRINT_GYRO	'v'	print Gyro data once
T_VERBOSELY_PRINT_GYRO	'V'	toggle verbosely print Gyro data
T_GYRO_BALANCE	'G'	toggle on off the gyro adjustment
T_GYRO_FINENESS	'g'	adjust the finess of gyroscope adjustment to accelerate motion
T_SLOPE	'l'	inverse the slope of the adjustment function
T_MELODY	'o'	play a pre-defined melody
T_TILT	't'	
T_MEOW	'u'	
T_SERVO_MICROSECOND	'w'	PWM width modulation
T_TUNER	'}'	use the serial communication to tune the parameters, so there's no need to upload the sketch everytime
T_ACCELERATE	'.'	make the gait faster by reducing the delay between loops
T_DECELERATE	','	make the gait slower by increasing the delay between loops
T_RESET	'!'	Reset the board and parameters

Imagen 13. Posibles valores constantes para la variable “token”

Cada una de estas constantes (o tokens, en este contexto) tiene un propósito asignado, tal y como se aprecia en la Imagen 13, y es la función `reaction()` la principal encargada de actuar en base a la constante almacenada en la variable `token`. Como en nuestro caso solo nos interesa ejecutar habilidades, podemos prescindir de prácticamente todos los tokens, exceptuando `T_SKILL`. De forma más concreta, se han eliminado (comentado) los siguientes tokens, ubicados en “OpenCat.h”:

- `T_PRINT_GYRO` y `T_VERBOSELY_PRINT_GYRO`
- `RANDOM_MIND`
- `T_QUERY`
- `T_JOINTS`
- `T_SAVE`
- `T_ABORT`
- `T_BEEP`
- `T_PAUSE`

- T_GYRO_FINENESS y T_GYRO_BALANCE
- T_INDEXED_SEQUENTIAL_BIN y T_INDEXED_SIMULTANEOUS_BIN
- T_INDEXED_SEQUENTIAL_ASC y T_INDEXED_SIMULTANEOUS_ASC

Al habernos desecho de estas constantes, el código ubicado dentro de las macros condicionales que observan estos valores, dentro de la función `reaction()`, no se incluirá en la compilación final, consiguiendo un ahorro de memoria:

```
#ifdef T_PRINT_GYRO
          else if (token == T_PRINT_GYRO) {
            print6Axis();
          }
#endif

#ifdef T_VERBOSELY_PRINT_GYRO
          else if (token == T_VERBOSELY_PRINT_GYRO) {
            printGyro = !printGyro;
            token = printGyro ? 'V' : 'v';  //V verbosely print
data
          }
#endif

#ifdef RANDOM_MIND
            if (token == T_RANDOM_MIND) {
            autoSwitch = !autoSwitch;
            token = autoSwitch ? 'Z' : 'z';  //Z active random
mind
          } else
#endif
```

Sin embargo, también existen, dentro de esta función, ciertas clausulas `case` dentro de un `switch`, encargado de verificar el valor de `token` mediante las constantes de la Imagen 13, que no están encapsulados en ninguna macro condicional, lo que provoca que, en tiempo de compilación, no se encuentren estas constantes, como es lógico por haberlas borrado. A continuación, un extracto de este switch que comprueba el valor de `token`:

```
case T_SAVE:
  {
    PTLF("saved");
    saveCalib(servoCalib);
    break;
  }
case T_ABORT:
```

```
  {
    PTLF("aborted");
    for (byte i = 0; i < DOF; i++) {
      servoCalib[i] = eeprom(CALIB, i);
    }
    break;
}
```

Para solucionar esta inconveniencia, hemos encapsulado todos estos cases en macros condicionales que comprueban la existencia de estas mismas constantes. Por ejemplo, en el caso `T_SAVE`:

```
#ifdef T_SAVE
      case T_SAVE:
        {
          PTLF("saved");
          saveCalib(servoCalib);
          break;
        }
#endif
```

- La eliminación de todos estos tokens y sus correspondientes bloques de código nos supone un ahorro de 4286 Bytes en Flash y 32 Bytes en RAM

Con todos estos cambios hemos conseguido reducir la memoria ocupada en 7690 Bytes, lo cual es una cantidad bastante generosa teniendo en cuenta el, relativamente limitado, espacio que tiene la placa NyBoard, quedándonos de forma aproximada sobre los 25KB en Flash y 868B en RAM. Gracias a estos cambios ahora contamos con más espacio para posibles añadidos futuros.

6. Demostrador

Para la demostración de uso de Bittle con M2OS, hemos implementado una aplicación con la que el robot se desplaza esquivando los obstáculos que encuentre en su camino. Para ello hemos incorporado al robot el sensor de distancia por infrarrojos Sharp GP2D120, el cual permite la medición de distancias con procesamiento de señal integrado y voltaje de salida analógico.

Imagen 14. Sensor Sharp GP2D120

El sensor funciona midiendo la distancia entre él y un obstáculo, y acto seguido devuelve un valor que es proporcional a esa distancia. Su rango de detección cubre distancias que van desde los 4 centímetros hasta los 30 centímetros.

La aplicación contará con las tareas comentadas en el apartado 4.3 (Bittle_Reaction y Bittle_Check) que aportan la funcionalidad básica de la librería. Junto a estas tareas se crearán dos tareas específicas de la aplicación, una de ellas encargada de realizar la medida de la distancia a los objetos y la otra encargada de comandar las acciones a Bittle en función de la distancia detectada. Concretamente contamos con las siguientes cuatro tareas, expuestas de mayor a menor prioridad:

- La labor de la primera tarea (`Demo_Distance`) es la medición de distancias mediante el uso del sensor de infrarrojos. Para ello, se ha definido una función en el cuerpo de la tarea Ada, `Read_GP2D120`, encargada de recibir los valores proporcionales a la distancia devueltos por el sensor, hacer una media de un conjunto de ellos y retornar la distancia final en centímetros.

  Así, el procedimiento en bucle de esta primera tarea llama a esta función que recoge la distancia medida por el sensor y la almacena en una variable compartida localizada en su fichero de especificación “demo_distance.ads” llamada `Distance`:

  ```
  procedure Distance_Task_Body is
  begin
        Distance := Read_GP2D120 (Sensor_Pin);
  end Distance_Task_Body;
  ```

  `Sensor_Pin` es una constante que sirve para indicar a esta función a través de que pin dentro la placa se desean comunicar los datos obtenidos por el sensor, eligiendo para mi caso el pin 3 (A3). Esta tarea cuenta con un periodo de 1,5 segundos.

- La segunda tarea (`Demo_Barrier_React`) es la encargada de comandar las acciones pertinentes al robot para evitar los obstáculos que encuentre en su camino:

```
procedure Barrier_React_Task_Body is
begin
      if Demo_Distance.Distance < Min_Distance or
         Periods_Of_Rotation /= 0 then

         Bittle.Action(Bittle.vtL); -- Rota hacia la izquierda

         if Periods_Of_Rotation = Total_Num_Of_Periods then
            Periods_Of_Rotation := 0;
         else
            Periods_Of_Rotation := Periods_Of_Rotation + 1;
         end if;
      else
         Bittle.Action(Bittle.wkF); -- Avanza
      end if;
end Barrier_React_Task_Body;
```

Principalmente, esta tarea verifica el valor de la distancia almacenada en la variable compartida sobre la que escribe la tarea anterior, `Distance`. Este valor se compara con la constante `Min_Distance`, que contiene la distancia a partir de la cual Bittle evita el obstáculo (fijada a 10 centímetros).

Entendiendo esto, se puede simplificar la tarea, de tal manera que, si Bittle detecta un obstáculo a menos de 10 centímetros, este, rota para procurar esquivarlo, mientras que, si el sensor devuelve una distancia mayor, el robot avanza en línea recta.

Sin embargo, con esta forma de funcionar surge un inconveniente: Imaginémonos que el robot avanza directamente hacia un obstáculo. Cuando Bittle detecta este obstáculo rota, únicamente, mientras que el sensor devuelva un valor menor que 10 cm. A medida que el perro rota, la distancia al objeto va a incrementar dado que ya no va a encararlo perpendicularmente. Esto va a causar que, eventualmente, la distancia detectada sea mayor que 10, provocando que Bittle vuelva a avanzar hacia adelante, sin embargo, a pesar de volver a ponerse en marcha, aún no se habrá esquivado este obstáculo por completo, volviendo a provocar una nueva detección.

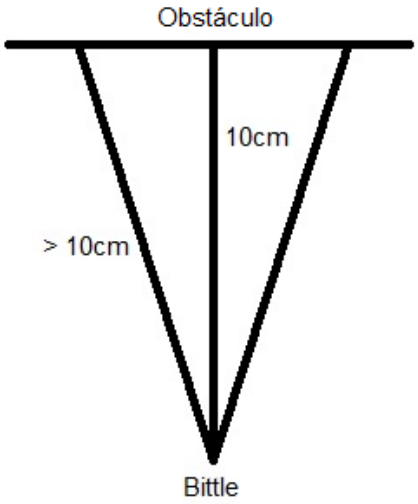

Imagen 15. Boceto del problema del obstáculo

Dado que lo más eficiente sería que Bittle evitase por completo la barrera encontrada hemos decidido añadir una ligera capa de complejidad adicional, cuyo objetivo es que Bittle rote durante una cantidad de tiempo más razonable para lograr así la evasión completa del objeto que se interpone en el camino.

Para lograr esto se ha añadido una variable `Periods_Of_Rotation`, y se ha sumado a la condición principal de esta tarea la comprobación del valor de la misma. Ahora, cuando Bittle detecta un obstáculo a menos de 10 centímetros, este no solo rotará mientras la distancia sea menor que 10, sino que también rotará, como poco, durante tantos periodos como los que indique la constante `Total_Num_Of_Periods`, en la condición encargada de comprobar esta primera variable (en nuestro caso, 5 periodos). Como el periodo de esta tarea es de 1,5 segundos (al igual que la primera tarea), todo esto se traduce en que, como mínimo, Bittle rotará durante 7,5 segundos cada vez que el sensor detecte un impedimento a menos de la distancia marcada por `Min_Distance`.

- Finalmente, la tercera y cuarta tareas, son, las ya mencionadas anteriormente, Bittle_Check y Bittle_Reaction respectivamente, manteniéndose exactamente tal y como se explicaron en el apartado 4.3 (mismo periodo y prioridad).

Así, con estas 4 tareas operando en conjunto, obtenemos el comportamiento deseado para este demostrador, aprovechando el uso del sensor infrarrojo conectado a Bittle.

Las 2 primeras tareas de este demostrador se encuentran ubicadas en el subdirectorio “examples/api_m2os/arduino_uno/bittle”, dentro de la carpeta principal de M2OS, junto al programa de prueba para este demostrador “demo_bittle.adb”.

7. Conclusiones y Trabajos futuros

7.1 Conclusiones

En este proyecto se ha logrado la adaptación de la librería OpenCat para su ejecución sobre el sistema operativo de tiempo real M2OS, gracias al empleo de las herramientas de importación de código Arduino escrito en C y C++.

Para ello, se ha estudiado el funcionamiento original de la librería OpenCat, probando la ejecución de la misma a través de Arduino, observando como Bittle recibe órdenes y como actúa ante ellas, verificando las variables, constantes y funciones que trabajan en conjunto para permitir el movimiento del robot, además de otras funcionalidades adicionales.

También se ha conseguido la compilación de esta librería para su uso en M2OS, realizando los cambios necesarios dentro del proyecto original de Petoi para permitir su uso en un programa Ada, incluyendo una API básica que permite el llamamiento de las funciones C/C++ de OpenCat a través de procedimientos y funciones Ada. Todo esto acompañado de un proyecto M2OS que hace uso de esta API para la integración de la librería en un conjunto de tareas.

Personalmente, me he decantado por este proyecto porque me parecía interesante todo lo que rodea la programación de un robot, con todos los detalles que se han de tener en cuenta para ello. Además, es una temática que se relaciona fuertemente con la rama de computadores la cual he elegido en mi carrera, donde se han aplicado conocimientos previamente adquiridos en asignaturas como “Sistemas de Tiempo Real” o “Programación Paralela, Concurrente y de Tiempo Real”.

Finalmente, me alegra poder contribuir al proyecto de M2OS, y que mi código pueda ser utilizado para posibles trabajos a futuro.

7.2 Trabajos futuros

A continuación, propongo algunas ideas que podrían resultar interesantes en relación a mi proyecto:

- Comparación de las prestaciones entre OpenCat original (Arduino) y OpenCat sobre M2OS.

- Realización de aplicaciones más complejas con mayor número de tareas M2OS sobre OpenCat, para el posterior análisis de planificabilidad y optimización de tareas haciendo uso de MAST.

Bibliografía

[1] Petoi. *About Us*: https://www.petoi.com/pages/about

[2] Petoi. *Nyboard customized arduino board*: https://www.petoi.com/products/nyboard-customized-arduino-board

[3] Petoi. *OpenCat Library*: https://github.com/PetoiCamp/OpenCat

[4] Wikipedia. *Arduino*: https://es.wikipedia.org/wiki/Arduino

[5] Arduino. *Language reference for Arduino core*: https://www.arduino.cc/reference/en/

[6] Petoi. *Petoi robot joint index*: https://docs.petoi.com/petoi-robot-joint-index

[7] Wikipedia. *Ada programming language*: https://en.wikipedia.org/wiki/Ada_(programming_language)

[8] Universidad de Cantabria. *M2OS, small RTOS intended for multitasking applications in small microcrontrollers*: https://m2os.unican.es

[9] Mario Aldea Rivas y Héctor Pérez Tijero. “Leveraging real-time and multitasking Ada capabilities to small microcontrollers”. *Journal of Systems Architecture* 94 (2019), págs. 32-41. ISSN: 1387-7621. DOI: https://doi.org/10.1016/j.sysarc.2019.02.015. https://www.sciencedirect.com/science/article/abs/pii/S1383762118302212

[10] Petoi. *Skill creation and explanation*: https://docs.petoi.com/applications/skill-creation

[11] IBM. *Name Mangling C++*: https://www.ibm.com/docs/en/i/7.5?topic=linkage-name-mangling-c-only

[12] Petoi. *Burn Bootloader for NyBoard*: https://docs.petoi.com/technical-support/burn-bootloader-for-nyboard