Facultad de Ciencias

LIBRERÍA DE COMUNICACIONES PARA CONTROL REMOTO DE ROBOTS

(COMMUNICATION LIBRARY FOR ROBOTS REMOTE CONTROL)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Ignacio J. Gómez Ortiz

Director: Mario Aldea Rivas

Co-Director: Héctor Pérez Tijero

Febrero - 2017

Resumen

La programación de funciones de red ocupa un gran papel en las aplicaciones para sistemas empotrados. Con frecuencia aparecen nuevas tecnologías y protocolos de comunicación diferentes, para conectar los dispositivos a distintos medios físicos e interactuar con dispositivos remotos. Esto aumenta la complejidad de implementación de las aplicaciones y el tiempo de desarrollo del producto final.

En el presente proyecto se pretende desarrollar una librería que simplifique la tarea del programador, añadiendo una capa de abstracción y sencillez en el manejo de las comunicaciones remotas en dispositivos EV3.

Los sistemas EV3, son la respuesta educativa que proporciona Lego para enseñar automatización y sistemas de control a niños de temprana edad. Dispone de un amplio catálogo de kits de sensores y actuadores que pueden ser conectados y manejados a través de un bus serie. Con el uso de su entorno de desarrollo y su lenguaje gráfico, son muy sencillos de programar. Y cuentan con Bluetooth y USB como tecnologías de comunicación integradas de serie.

Gracias a proyectos libres como el sistema ev3dev, un sistema operativo basado en Debian y librerías como ev3c, creada para proporcionar abstracción en el manejo de los periféricos del EV3, podemos convertir la plataforma en un sistema más parecido a la realidad industrial que los alumnos de ingeniería se encontrarán al incorporarse en el mercado laboral y a un precio asequible. Todo esto convierte lo que inicialmente se pensó como un juguete, en una herramienta perfecta para ilustrar problemas de ingeniería, que junto a esta librería permitirá a los alumnos centrarse en aprender y probar los algoritmos de control típicamente utilizados en la industria.

Palabras clave: Comunicación, Bluetooth, USB, Sistema empotrado, EV3

Abstract

Programming network takes a important role in applications for embedded systems. New technologies and different communication protocols often appear to connect systems to different physical layers and interact with remote devices. This fact increases the implementation complexity of the applications and the development time of the final product.

The present project aims to develop a library that simplifies the task of the programmer, adding a layer of abstraction and simplicity in the handling of remote communications in EV3 devices.

EV3 systems are the educational response provided by Lego to teach automation and control systems to young children. It has a wide catalog of sensor kits and actuators that can be connected and handled through a serial bus. With the use of their development environment and their graphic language, they are very simple to program. And they have Bluetooth and USB as standard integrated communication technologies.

Thanks to free projects like the ev3dev system, a Debian-based operating system and libraries like ev3c, created to provide abstraction in the management of the EV3 peripherals, we can turn the platform into a system more similar to the industrial reality that the students of Engineering will find themselves entering the labor market and at an affordable price. All this makes what was initially thought of as a toy, a perfect tool to illustrate engineering problems, which together with this library will allow students to focus on learning and testing the control algorithms typically used in the industry.

Keywords: Communication, Bluetooth, USB, Embedded system, EV3

Tabla de contenidos

1. Introducción

1.1 Sistemas empotrados

Los sistemas empotrados son sistemas electrónicos programables dedicados principalmente a aplicaciones de monitorización y control. Para conseguir este objetivo, estos sistemas se caracterizan una fuerte interacción con el entorno por medio de sensores y actuadores.

Podemos tomar como ejemplo de sistema empotrado el controlador de un ascensor. A grandes rasgos un ascensor dispone de una interfaz sencilla con el usuario, compuesta por una serie de botones para indicar el piso al que nos queremos desplazar y un actuador mecánico o neumático que permite elevar o descender la caja del ascensor para alcanzar el piso deseado.

Inicialmente el diseño de un sistema empotrado dependía totalmente de la aplicación concreta a la que se iba a destinar. Pero con la aparición masiva de microprocesadores, los sistemas empotrados han desembocado en los modelos actuales, formados por una plataforma hardware, compuesta por una cpu de propósito general y periféricos conectados a un bus y un software específico para dar solución a la aplicación deseada.

Con el paso del tiempo, el software fue cobrando cada vez más importancia. Esto originó el desarrollo de librerías estandarizadas de software reutilizable y desarrollo de herramientas de apoyo a los desarrolladores. Hasta que llegó una amplia oferta de sistemas operativos para sistemas empotrados como VxWorks, Windows CE, Linux, etc. Sistemas operativos que tienen que dar solución a los principales requisitos de un sistema empotrado:

- Ejecución en tiempo real. Los tiempos de respuesta deben estar acotados mediante una planificación apropiada del software del sistema.
- Tolerancia a fallos.
- Escasez de recursos. Por motivos de ahorro de energía, espacio, coste y peso, los sistemas empotrados tienen una arquitectura reducida con unos recursos de computación y memoria limitados.

Sin embargo, incluso hoy en día muchos sistemas empotrados no disponen de un sistema operativo, tan solo implementan un bucle de control mediante un ejecutivo cíclico, que ejecuta las mismas tareas de forma periódica. En aplicaciones simples esto es suficiente, pero conforme aumenta la complejidad del sistema se hace necesaria la capa de abstracción proporcionada por el sistema operativo. Unido a la creciente complejidad de los sistemas, tenemos cada vez mayor demanda de incorporar conexión del sistema empotrado a alguna clase de red y por tanto aparece la necesidad de implementar pilas de protocolos y librerías para establecer comunicación con el exterior.

1.2 Protocolos de comunicación

Existen multitud de especificaciones y protocolos de comunicación para su uso en sistemas empotrados. Dependiendo del medio físico por el que queramos transmitir nuestros datos, la velocidad de transmisión y la tolerancia a perturbaciones, elegiremos uno u otro protocolo. Vamos a hablar en concreto de ZigBee y Bluetooth, dos protocolos inalámbricos que se suelen emplear con frecuencia en sistemas empotrados. Hablaremos un poco de cada uno de ellos y sus principales

características para finalmente hacer una comparativa y una justificación de nuestra elección.

ZigBee es una especificación de un conjunto de protocolos de alto nivel, para permitir la comunicación inalámbrica de bajo consumo entre dispositivos. Está basada en el estándar IEEE 802.15.4. ZigBee integra las tareas básicas de comunicación como el enlazado de dispositivos, asignación de direcciones dinámicas, enrutamiento, etc. Su objetivo principal son las aplicaciones que requieren comunicaciones seguras con baja tasa de transferencia de datos y alto ahorro energético. ZigBee se centra en la sencillez y el bajo coste. En comparación con Bluetooth, un nodo ZigBee requiere en teoría un 10% del hardware necesario para un nodo Bluetooth.

Bluetooth es una especificación para redes inalámbricas de área personal, que posibilita la transmisión de voz y datos entre diferentes dispositivos mediante un enlace por radiofrecuencia. Cada dispositivo debe llevar equipado un transceptor que transmite y recibe en la frecuencia de 2.4 GHz. Ese chip lleva asociada una dirección única de 48 bits basado en el estándar IEEE 802.15.1. Los dispositivos se clasifican en clase 1, clase 2 o clase 3 en referencia a su potencia de transmisión, siendo totalmente compatible la comunicación entre dispositivos de distintas clases.

Normalmente Bluetooth dispone de 79 canales de comunicación. Dos o más dispositivos que usen el mismo canal forman un conjunto llamado piconet. El maestro de la piconet es el responsable de la sincronización entre los dispositivos del conjunto, su reloj y sus saltos de frecuencia controlan al resto de dispositivos. Es el maestro el que por defecto, lleva a cabo el procedimiento de búsqueda y establecimiento de la conexión. Los esclavos simplemente se sincronizan y siguen la secuencia de saltos determinada por el maestro.

Para terminar, se hace una comparación de las principales características de ambos protocolos:

- Una red ZigBee puede constar de un máximo de 65535 nodos distribuidos en subredes de 255 nodos, frente a los 8 máximos de una piconet Bluetooth.
- ZigBee tiene un menor consumo eléctrico que Bluetooth.
- Zigbee consigue velocidades de hasta 250 kbit/s, frente a los 32 Mbit/s de las últimas versiones de Bluetooth.
- Mientras que Bluetooth se utiliza principalmente para aplicaciones como los teléfonos móviles y la informática general, la velocidad de ZigBee lo desvía hacia usos tales como la domótica y aplicaciones de bajo consumo.

En este trabajo se ha elegido el protocolo Bluetooth en lugar de ZigBee, principalmente por el mayor ancho de banda que proporciona, pero en otras aplicaciones podría ser preferible el uso de ZigBee.

1.3 Objetivos y motivación del proyecto

El objetivo principal de este trabajo es realizar un marco de abstracción para el control de robots, que facilite las comunicaciones entre los dispositivos EV3 de Lego y un computador personal (ver Figura 1.1). Los kits de robótica de Lego Mindstorms proporcionan una plataforma barata y de fácil configuración. Son un modelo a escala de los elementos empleados en la industria de la automatización, lo que les convierte en una plataforma educativa perfecta como introducción a la robótica para alumnos de ingeniería.

Habilitando la comunicación entre el dispositivo EV3 y un PC conseguimos que ambas plataformas

se complementen: por un lado resolvemos los problemas de carencia de potencia de cómputo y capacidad de memoria que tiene el dispositivo EV3 y por otro lado, proporcionamos al PC una interfaz de interacción con una amplia gama de sensores y actuadores.

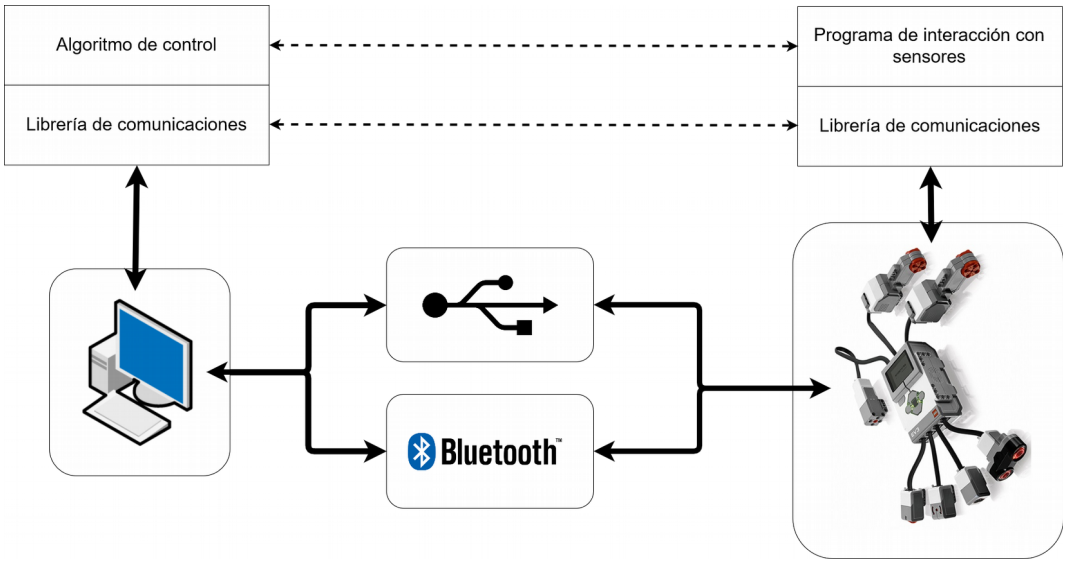

Figura 1.1: esquema general del sistema

La meta de este proyecto es proporcionar una interfaz sencilla y portable a distintos sistemas operativos. Además, deberá facilitar el desarrollo de sistemas de comunicación en las aplicaciones entre un PC y un dispositivo EV3, independientemente del robot a controlar, esto es el número y tipo de sensores/actuadores que utilice. La introducción de estas comunicaciones, permite dar a cada dispositivo el mejor uso para el que fue diseñado.

En particular, este proyecto aborda los siguientes objetivos:

- Estudio de las opciones de comunicación que nos proporciona el dispositivo EV3.
- Desarrollo de una librería Linux para facilitar las operaciones de comunicación entre los algoritmos e interfaces de control y el dispositivo EV3.
- Diseñar e implementar una librería software capaz de gestionar el uso de esas interfaces de comunicación presentes en el dispositivo EV3.
- Implementar un desarrollo sobre EV3 para ejemplificar el uso de la librería implementada.

1.4 Organización de la memoria

La memoria está organizada de la siguiente manera:

- En el capítulo 2 se introducen las herramientas y tecnologías sobre las que se va a apoyar nuestra librería de comunicaciones. Se hace una descripción del dispositivo EV3, explicando sus características y explicamos los protocolos y tecnologías de comunicaciones sobre los que se apoya la librería.
- En el capítulo 3 damos una visión sobre el entorno de trabajo y su configuración y el uso del sistema ev3dev y la librería ev3c para el desarrollo de aplicaciones sobre el EV3.
- En el capítulo 4 explicamos capa a capa el diseño y desarrollo de la librería de

comunicaciones y como cada capa interactúa con las demás.

- En el capítulo 5, realizamos una implementación de un algoritmo de control para un robot segway con kits de lego y lo adaptamos para el uso y demostración de la librería de comunicaciones.
- Para terminar, en el capítulo 6 repasamos los objetivos y los resultados obtenidos.

2. Herramientas y tecnologías utilizadas

2.1 Plataforma EV3

El sistema programable EV3 (ver Figura 2.1) es el centro de control de la línea de robótica de lego. Desarrollado para manejar los distintos sensores y actuadores de los sets de piezas Lego Mindstorms. Guarda compatibilidad con los dispositivos de la versión anterior, el NXT. Es decir, todos los sensores y actuadores NXT son reconocidos por el sistema EV3.

Figura 2.1: sistema programable EV3

Las principales características del dispositivo son:

- Dispone de una cpu TI Sitara AM1808 (core ARM926EJ-S) 300 MHz. ARM es una arquitectura RISC muy utilizada en sistemas empotrados, orientada al bajo consumo.
- 64 MB de memoria RAM.
- 16 MB de memoria Flash.
- Un puerto de conexión para tarjetas de memoria microSDHC. Gracias al firmware integrado en el dispositivo EV3, que inicializa el sistema y carga automáticamente la imagen del sistema operativo guardado en la tarjeta SD, podemos realizar una rápida configuración inicial y tener el dispositivo en funcionamiento en cuestión de minutos, como ocurre con un ordenador personal.
- Un puerto de conexión mini USB 2.0/1.1.
- Comunicación Bluetooth v2.1 integrada.
- Cuatro puertos para sensores. Analógico/digital. Controlador por una UART (460.9 Kbit/s).
- Cuatro puertos con encoders para el uso de motores. Nos permiten obtener lecturas tanto de posición angular como de velocidad angular de los motores.
- Seis botones con luz de fondo.
- Display 178x128 pixel monochrome LCD.

Lego proporciona en sus kits una gran variedad de sensores, desde un sensor giroscópico, a sensores de proximidad, de detección de color, sensor táctil… y dos tipos de motores. Pero también proporciona retrocompatibilidad con los sensores de la versión anterior de Lego Mindstorms. Así que los sensores de NXT también son compatibles con EV3, lo que aumenta todavía más la gama de dispositivos disponibles.

Además, Lego proporciona un ide basado en LabView, para programar el control de los robots con un lenguajes de programación gráfico (ver Figura 2.2). Esto es perfecto para iniciar al público infantil en la programación y el control de robots, aunque no proporciona un nivel de interacción con el sistema suficiente para la enseñanza universitaria. Pero gracias a desarrollos de la comunidad como lejOS (basado en la máquina virtual de java) y ev3dev (basado en Debian), podemos usar el dispositivo para amplias aplicaciones en ingeniería.

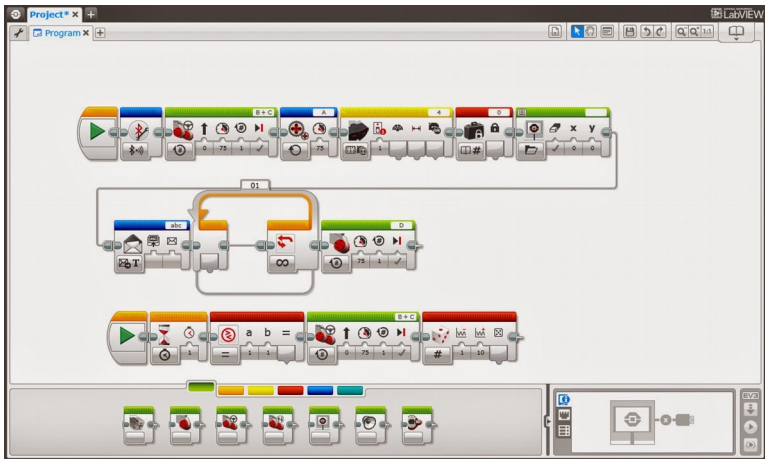

Figura 2.2: IDE de Lego para programación gráfica

2.2 Bluetooth

Una parte significativa de la especificación Bluetooth está dedicada a las tareas de bajo nivel, como la especificación de diferentes radios de frecuencias en los que transmitir, el timing y los protocolos de señalización y las necesidades para establecer comunicación. En este apartado, de toda la especificación de la tecnología Bluetooth, vamos a centrarnos en un pequeño marco concerniente a la visión del programador de aplicaciones Bluetooth.

Aunque Bluetooth fue diseñado desde cero, con independencia de Ethernet y el protocolo TCP/IP, es razonable pensar en la programación Bluetooth de la misma forma que con la programación Ethernet. Las diferentes partes de la programación de red pueden ser separadas en diferentes componentes:

- Elegir un dispositivo con el que comunicarse
- Especificar como nos vamos a comunicar con él
- Hacer una conexión con el dispositivo
- Aceptar una conexión entrante del dispositivo
- Enviar datos
- Recibir datos.

Todos los chips Bluetooth contienen una dirección global única de 48 bits, la cual identifica el dispositivo. Es idéntica a la dirección MAC de Ethernet y ambos espacios de direcciones está gestionados por el IEEE Registration Authority.

Para que un dispositivo Bluetooth comunique con otro, debe tener alguna forma de averiguar su dirección Bluetooth. Esta dirección es usada por todas las capas del proceso de comunicación

Bluetooth, en contraste con TCP/IP que usan la dirección MAC únicamente en la capa de enlace. En Ethernet el usuario no suministra la dirección MAC del dispositivo objetivo, sino que proporciona una dirección amigable como un dominio o una ip. En Bluetooth ocurre lo mismo, tenemos nombres de dispositivo amigables como *My Phone*, que el sistema cliente se encarga de traducir en dirección numérica.

Una vez que se ha determinado la dirección del dispositivo al que se quiere conectar, es necesario determinar que protocolo de transporte usar. Tenemos dos protocolos de transporte en Bluetooth:

- RFCOMM provee aproximadamente los mismos servicios y las mismas garantías de comunicación que TCP. La mayor diferencia con TCP, es que RFCOMM solo tiene 30 puertos distintos, frente a los 65535 disponibles en TCP.
- L2CAP permite la adaptación del grado de fiabilidad en la transmisión, para adaptar la comunicación al uso que el usuario requiere. Por defecto es un protocolo orientado a la conexión, que envía los paquetes de forma fiable.

Después de elegir el protocolo de transporte, debemos elegir el número de puerto. Casi todos los protocolos de transporte en Internet tienen un uso común de la noción de número de puerto, de modo que múltiples aplicaciones en el mismo equipo puedan utilizar simultáneamente un mismo protocolo de transporte. Bluetooth también los utiliza pero con una terminología diferente. En L2CAP los puertos se llaman Protocol Service Multiplexers y puede tomar valores entre 1 y 32767, con el rango de 1 a 4095 como puertos reservados. Mientras que en RFCOMM, solo tenemos canales o puertos del 1 al 30.

En la programación de aplicaciones para internet sobre los protocolos de transporte de la familia TCP/IP, se hace uso de números de puerto bien conocidos, que son elegidos en el momento del diseño. No se pueden ejecutar dos aplicaciones de servidor que utilizan el mismo número de puerto, pero gracias al gran número de puertos disponibles, esta restricción todavía no ha supuesto un problema. Bluetooth sin embargo, fue diseñado con muchos menos números de puertos disponibles en el caso del protocolo de transporte RFCOMM. En este caso el diseñador de aplicaciones debe elegir bien los números de puerto a ocupar por las distintas aplicaciones.

Pero Bluetooth propone una solución al uso de los números de puerto: el SDP (Service Discovery Protocol). En lugar de acordar los números de puerto de cada aplicación en el momento del diseño, Bluetooth los asigna en tiempo de ejecución. La máquina opera una aplicación de servidor llamada SDP, que utiliza uno de los puertos reservados L2CAP. Las demás aplicaciones de servidor se asignan dinámicamente números de puerto en tiempo de ejecución, realizando un registro en el servidor SDP. Las aplicaciones clientes consultan el servidor SDP de la máquina que aloja los servidores, para obtener la información del uso de puertos.

Una vez elegido el equipo al que nos queremos conectar, el protocolo de transporte y el número de puerto por el que vamos a escuchar, escribir una aplicación Bluetooth es esencialmente el mismo tipo de programación que el que se utiliza en cualquier aplicación de red. Una aplicación de servidor que espera una conexión entrante y una aplicación cliente que intenta establecer una conexión de salida.

2.3 Protocolos de internet

A continuación daremos una visión de las capas de internet y transporte, en concreto de los protocolos TCP, UDP e IP que son de interés para el desarrollo de este proyecto.

El protocolo IP es el protocolo de interconexión de redes más utilizado. Es un protocolo de nivel de internet no orientado a la conexión y no fiable. Es decir los paquetes enviados a la red son tratados de forma independiente a la hora de encaminarlos, y no garantiza la recepción del paquete ni el orden de llegada. Tampoco detecta ni corrige errores, esa funcionalidad se desplaza a las capas superiores.

IP proporciona sus servicios añadiendo una cabecera de 20 bytes de longitud al PDU (Protocol Data Unit) suministrado por el nivel superior. Dentro de esos servicios, están la posibilidad de identificar el dispositivo remoto utilizando la dirección IP como identificador de la red y del equipo y la posibilidad de fragmentar los paquetes.

El protocolo TCP (Protocolo de Control de Transmisión) proporciona una conexión fiable para transferir los datos entre las aplicaciones. Cada PDU de TCP, contiene una cabecera de longitud 20 bytes, donde se incluyen la identificación de los puertos de origen y destino además de otros campos. Los valores de los puertos identifican a las respectivas aplicaciones de usuario de las dos entidades TCP. Durante la conexión, cada entidad seguirá la pista de los segmentos TCP que vengan y vayan hacia la otra entidad, para así regular el flujo de segmentos y recuperar aquellos que se pierdan o dañen.

UDP o protocolo de datagramas de usuario (user datagram protocol), no garantiza la entrega, la conservación del orden secuencial, ni la protección frente duplicados. UDP posibilita el envío de mensajes entre aplicaciones con la complejidad mínima. Su cometido es básicamente añadir a la capa IP, la capacidad de identificar los puertos de aplicación y para ello emplea una cabecera de solo 8 bytes

Para el desarrollo de la librería de comunicaciones se han elegido el protocolo UDP sobre IP como protocolos de base sobre los que apoyar toda la funcionalidad. Proporcionan una capa de abstracción suficiente para hacer uso de todos los medios físicos disponibles en el dispositivo EV3 y con el uso de UDP reducimos los bits de cabecera, delegando parte del control de flujo a la capa ev3comm como veremos en el apartado 4.

La interfaz de sockets proporcionada por Linux para interactuar con la capa de transporte del protocolo de red, se fundamenta sobre la llamada socket(). Esta función retorna un puntero a un descriptor de socket. Con ese puntero podemos configurar la conexión, indicando la dirección ip de destino, los puertos de emisión/recepción, el protocolo a utilizar, etc. Así como hacer llamadas a las operaciones de emisión y recepción de mensajes. Todas estas operaciones quedarán ocultas al usuario dentro de la librería de comunicaciones.

2.4 Tethering

A raíz del auge tecnológico promovido por la aparición de los smartphones, surgió el uso y estandarización de nuevas tecnologías para dar solución a nuevos casos de uso de los usuarios de dispositivos móviles. Uno de estos ejemplos es el proceso Tethering o anclaje a red.

En el proceso Tethering, intervienen dos tipos de dispositivos: el dispositivo con acceso a internet que llamaremos *nodo pasarela* y el resto de dispositivos que se conectan al nodo pasarela para recibir conexión a internet y que llamaremos simplemente *nodos*.

En esta topología, el nodo pasarela asume el papel de “modem” para los nodos. Los nodos se pueden conectar al nodo pasarela mediante una conexión inalámbrica como Bluetooth o Wi-Fi o con un bus físico, como por ejemplo USB. Cuando conectamos un nodo a un nodo pasarela, el sistema operativo crea una interfaz de red en el propio nodo y una interfaz de red externa en el nodo pasarela. Esta interfaz de red externa está a su vez conectada a la red LAN del nodo pasarela, permitiendo al nodo tener comunicación con la red local del nodo pasarela y por tanto con internet y con el propio nodo pasarela.

Todo esto se consigue a través de los drivers del sistema operativo y es totalmente transparente. El usuario hace el mismo uso de la red que haría si estuviese directamente conectado al gateway, viéndose reducida su velocidad de transferencia y ancho de banda por el uso de un enlace extra. Las aplicaciones no son modificadas, ya que han sido construidas sobre la familia de protocolos de internet y son los drivers los que se encargan de modificar la capa física y la capa de enlace para hacer uso de la tecnología Tethering. Ni el usuario ni la librería de comunicaciones tienen que realizar ninguna configuración sobre Tethering, tan sólo establecer el enlace deseado (en nuestro caso conectando el cable USB o sincronizando los controladores Bluetooth como veremos en el apartado 3.1).

Esto nos permite como desarrolladores, realizar una aplicación escrita en un lenguaje de alto nivel, utilizando la interfaz de sockets proporcionada por el sistema operativo para realizar sus operaciones de comunicación en red y abstraernos totalmente del medio físico utilizado.

Dado que el dispositivo EV3 utiliza un sistema operativo basado en Linux, dispone de los drivers que nos permiten utilizar Tethering entre el computador y el EV3. En este caso, la idea no es proporcionar acceso a internet al dispositivo EV3, sino al revés, proporcionar al computador acceso a la red local del dispositivo EV3 y por tanto conexión al dispositivo EV3. Esto nos facilita enormemente el desarrollo de la librería de comunicaciones, y nos proporciona compatibilidad directa con Bluetooth, Wi-Fi y USB, simplemente programando nuestra librería de comunicaciones sobre la pila de protocolos UDP/IP.

3. Entorno de desarrollo

3.1 Entorno de trabajo

Antes de empezar a trabajar con el brick EV3, tenemos que cargar una imagen del sistema ev3dev[4] en la tarjeta microSDHC. Esta operación no borrará el firmware original del dispositivo. Es el firmware original el que se encarga de iniciar el proceso de arranque del sistema ev3dev y por tanto, solo tendremos que sacar la tarjeta SD para dejar el dispositivo en sus condiciones iniciales.

Una vez configurado el sistema EV3, nos enfrentamos al desarrollo software en dos plataformas diferentes. Por un lado un computador personal, en el que se ejecuta Ubuntu (una distribución del sistema operativo Linux) sobre un procesador Intel. Y por otro lado tenemos el dispositivo EV3, que ejecuta una distribución Debian, pero en este caso sobre un procesador ARM9. Esta diferencia de arquitecturas nos obliga a utilizar dos compiladores distintos.

En nuestro caso, hemos decidido utilizar el compilador GCC para la compilación de código ejecutable para el computador personal. Y un compilador GCC cruzado para arquitectura ARM. Este nos permite la compilación del código del dispositivo EV3 en el computador personal.

Una vez compilado el código del dispositivo EV3 en el computador personal, tenemos dos posibilidades para cargar la aplicación en el EV3:

- Copiando el programa en el sistema de ficheros de la tarjeta flash.
- Enviando el programa vía ssh con una conexión Tethering.

La configuración del Tethering es automática y transparente para el usuario. USB es plug&play, así que en el momento en que conectemos el cable USB al computador y al dispositivo EV3, el enlace y la comunicación Tethering quedará automáticamente configurados. Sin embargo, para realizar la comunicación Tethering sobre Bluetooth, antes debemos emparejar ambos controladores Bluetooth para configurar el enlace.

En Ubuntu es recomendable descargarse una herramienta que mejore la aplicación de gestión de conexiones Bluetooth que el sistema proporciona por defecto. Nosotros utilizaremos la aplicación *blueman bluetooth manager*, incluida en el repositorio *blueman*. Una vez configurada la aplicación blueman y encendido el dispositivo EV3, realizar un emparejamiento Bluetooth es tan sencillo como activar el Bluetooth en el dispositivo EV3 y hacerle visible, para que desde el PC podamos encontrarle en la lista de dispositivos enlazables. En el PC elegimos la opción de usar el dispositivo EV3 como punto de acceso y confirmamos la passkey de emparejamiento tanto en el PC como en el dispositivo (ver Figura 3.1).

Figura 3.1: confirmación de emparejamiento Bluetooth EV3

Para facilitar y automatizar en cierto modo la conexión ssh, se recomienda configurar la autentificación de clave privada – clave pública. Esto nos permite validar nuestros credenciales mediante el uso de la clave privada, en lugar de estar introduciendo constantemente el password de usuario para acceder al dispositivo.

3.2 Ev3dev

Ev3dev es un sistema operativo Debian basado en Linux, compatible con los dispositivos EV3 y Raspberry Pi. Lo importante de este producto es que la comunidad de desarrolladores han incluido un framework de drivers de bajo nivel [2] que nos permiten controlar tanto los sensores como los actuadores que conectamos al dispositivo, simplemente leyendo y escribiendo de un fichero. Y dado que es un sistema operativo Debian, disponemos de todos los beneficios de usar el kernel de Linux, como son los drivers para dispositivos USB, Bluetooth, Wi-Fi, cámaras, teclados, etc. Y lo más importante para este proyecto, proporciona la tecnología Tethering sobre la que se apoya la librería de comunicaciones y que nos permite comunicarnos con equipos remotos haciendo uso de los protocolos de internet, como ya hemos visto en el capítulo anterior.

Además de este sistema operativo, contamos con la librería ev3c, escrita completamente en lenguaje C, que agiliza el desarrollo de software sobre la plataforma ev3dev, facilitando el acceso a los periféricos de Lego. Esta librería añade una capa de abstracción sobre el acceso a los ficheros de los dispositivos. Esta interfaz con las funciones de bajo nivel, nos evita lidiar constantemente con descriptores de ficheros y llamadas write y read de bajo nivel para operar con los sensores y actuadores.

3.3 Arquitectura de una aplicación en ev3dev

Si quisiéramos desarrollar desde cero una aplicación para el dispositivo EV3 tendríamos dos opciones. La primera es utilizar el software y el lenguaje gráfico que nos proporciona Lego. Esta opción nos limitaría las vías de desarrollo al uso del código implementado por las librerías propietarias de Lego Mindstorms. La otra opción más libre, que es la que utilizaremos en este proyecto, es apoyarse en el sistema ev3dev.

Los pasos a seguir para trabajar con un sensor usando la librería ev3c son los siguientes:

- Cargar los sensores disponibles. Para ello, la función `ev3_load_sensors()` crea una lista enlazada de estructuras `ev3_sensor`, en donde la librería almacena toda la información disponible de cada sensor, mediante la lectura de sus correspondientes ficheros.
- Elegir el sensor concreto que queremos manejar. Bien con su identificador o mediante el puerto al que está conectado. La función `ev3_search_sensor_by_port()` busca por nosotros la estructura `ev3_sensor` asociada a un puerto concreto, de entre todas las estructuras de la lista devuelta por `ev3_load_sensors()`.
- Ejecutar la operación `ev3_open_sensor()` sobre la estructura del sensor que queremos operar.

Después de estas operaciones, ya tenemos el sensor listo para operar en nuestra aplicación. Con la estructura `ev3_sensor` debidamente inicializada, podemos realizar operaciones variadas sobre el sensor, como el cambio de modo de operación, modificar parámetros de su configuración u obtener lecturas.

El funcionamiento de la librería es muy similar para manejar los actuadores de lego, dado que gracias al framework de ev3dev, también controlamos su funcionalidad mediante el sistema de ficheros.

Como desarrolladores de una aplicación para el dispositivo EV3, ya tenemos resuelto el problema de manejar los sensores y actuadores. Solo nos queda resolver el problema de las comunicaciones. ¿Qué podemos hacer para comunicarnos con otros dispositivos?.

Dado que desarrollamos nuestra aplicación sobre una plataforma que cuenta con puerto USB y Bluetooth incorporado, la respuesta a esta pregunta tiene dos soluciones directas:

- Comunicación por puerto USB.
- Comunicación por Bluetooth.

Normalmente para desarrollar comunicaciones sobre un bus USB se suele recurrir a librerías desarrolladas por terceros para facilitar la programación, como por ejemplo la librería libusb. Para ello el sistema tiene que ser compatible con el USB Device Filesystem. Si el dispositivo no es compatible, habría que recurrir al desarrollo de un driver USB para comunicarnos con el dispositivo concreto. En nuestro caso solo queremos conectar el EV3 a un ordenador personal, así que no tendríamos ningún problema y podríamos apoyarnos en la librería libusb para realizar la comunicación, con la complejidad que ello conlleva.

Por otro lado, si queremos que nuestra comunicación se realice a través de Bluetooth, tenemos la pila de protocolos BlueZ. Es la pila de comunicaciones Bluetooth oficial para Linux. Nos proporciona una API similar a las funciones socket proporcionadas para el uso de Ethernet, pero además de éstas, se añaden otras llamadas para gestionar y comunicar con el controlador Bluetooth.

Como podemos observar, proporcionar comunicación a nuestra aplicación aumenta el grado de complejidad de desarrollo. Por suerte disponemos de una tercera opción, que además nos proporciona soporte para comunicarnos a través de ambos medios físicos, tanto USB como Bluetooth y sin añadir complejidad adicional a la aplicación. La solución para nuestra librería es el uso del Tethering. Esta tecnología nos permite desarrollar las comunicaciones sobre la interfaz de sockets proporcionada por los protocolos TCP/IP y UDP/IP, olvidándonos de las capas inferiores con las que estamos operando.

4. Librería de comunicaciones

La propuesta de realizar una librería de comunicaciones para dispositivos EV3, surgió con la idea de dotar a un sistema empotrado como el dispositivo de Lego, una mayor potencia de cómputo. Lo primero en que tenemos centrarnos es en qué medio físico vamos a utilizar para transmitir los mensajes. EV3 dispone de un controlador Bluetooth integrado y un puerto USB, así que las posibilidades pasaban por una comunicación inalámbrica por Bluetooth directa, una comunicación por cable USB o una comunicación por Wi-Fi mediante el uso de un *dongle* Wi-Fi.

Realizar una librería que soporte estos tres protocolos de forma nativa, aumenta demasiado la complejidad de implementación, como ya hemos explicado en el apartado anterior. Pero con el uso del protocolo IP sobre el proceso Tethering, obtenemos una capa de abstracción proporcionada por el sistema operativo, para operar sobre los distintos protocolos sin necesidad de interactuar directamente con cada tecnología. Esto agiliza el desarrollo, ya que la librería sólo interactúa con la interfaz de sockets del sistema operativo. Y el usuario de la librería solo tiene que preocuparse de conectar los dispositivos por cable USB (plug&play) o enlazarlos por Bluetooth (como vimos en el apartado 3.1).

En los siguientes apartados veremos las capas que implementa la librería y los protocolos sobre los que se apoya.

4.1 Diseño de la librería

Una vez que nos hemos documentado sobre el dispositivo EV3, sus características y posibilidades, es el momento de crear la librería. Dado que disponemos de la librería ev3c desarrollada por terceros, que funciona correctamente a la hora de interactuar con los sensores y actuadores disponibles para el dispositivo EV3, no es necesario implementar esa funcionalidad en nuestro trabajo. Nos centramos en el desarrollo de la funcionalidad de comunicación entre el dispositivo EV3 y un equipo funcionando bajo Linux.

La librería ev3c está implementada en lenguaje C. Por mantener la compatibilidad sin demasiadas complicaciones, la versión de la librería de comunicaciones también estará escrita en el mismo lenguaje.

El diseño de la arquitectura de la librería se descompone en dos capas, con el fin de encapsular por un lado los componentes destinados a dar la funcionalidad de comunicación de red y por el otro los componentes de gestión de los datos. La capa *controller* y la capa *ev3comm* son las encargadas de encapsular estas funcionalidades (ver Figura 4.1). Para ello, están comunicadas entre sí a través de una interfaz y necesitan apoyarse en un paquete de herramientas independientes del sistema operativo, compuesto por las dos herramientas implementadas: cola circular (librería *cola*) y máquina de estados (librería *sm*).

Para el uso de la librería, es necesario compilarla dos veces. Una para el sistema EV3 usando el compilador cruzado e indicando el flag __EV3__. Y otra con el compilador general, indicando el flag __LINUX__. Es necesario indicar el flag en cada caso, porque existen diferencias en el comportamiento de la capa ev3comm entre la versión de PC y la versión de EV3.

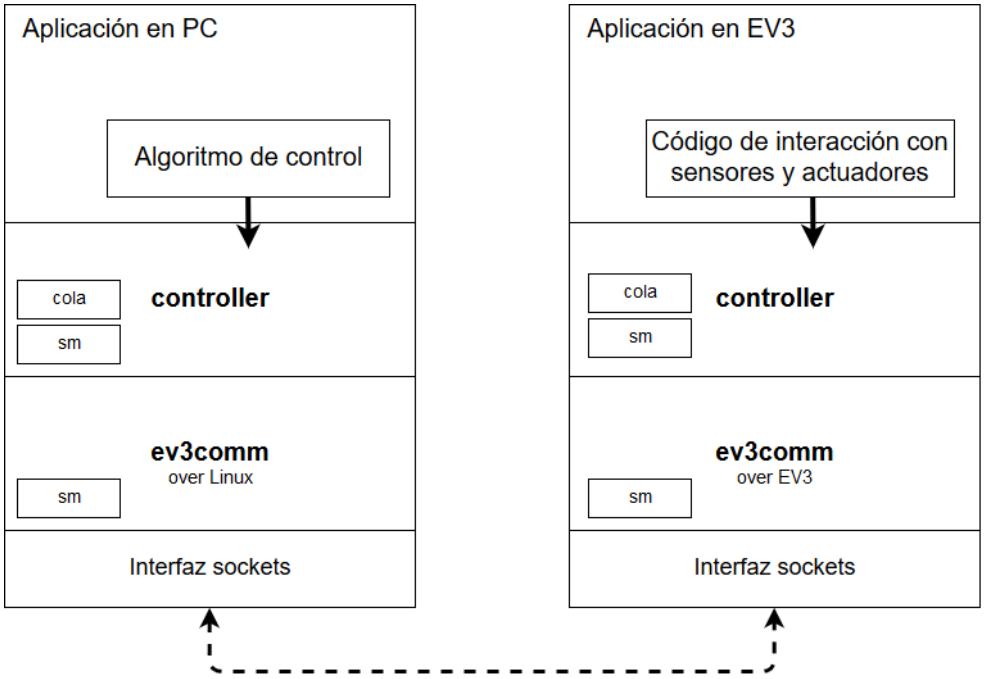

Figura 4.1: Organización en capas de la librería

Se deja en manos del usuario el desarrollo de la aplicación para PC, que invoque a la librería *controller* y que deberá encapsular el algoritmo de control, en un prototipo de función proporcionado por la librería. En el lado del EV3, la librería proporciona una función con las interacciones básicas del EV3 con los sensores y actuadores utilizando la API de la librería *ev3c*. El usuario puede modificar esta función para ajustar su funcionamiento a los requisitos del robot. Pero como en el caso del desarrollo en PC, el usuario deberá realizar un programa básico que invoque a la librería.

4.2 Herramientas

Como ya hemos comentado, la implementación de la librería se sostiene sobre dos herramientas fundamentales, la cola y la máquina de estados. Las cuales se describen a continuación.

Una máquina de estados, es en esencia un conjunto de nodos unidos por un conjunto de transiciones. Cada nodo contiene una funcionalidad que se ejecuta cuando la máquina de estados visita el nodo. Las transiciones nos dicen qué nodos podemos visitar desde el nodo actual, dibujando así la topología de la máquina de estados. Además, para que la máquina funcione correctamente necesitamos especificar un nodo de inicio (donde comienza el ciclo de ejecución) y un nodo final o aceptador.

En nuestra especificación de máquina de estados contamos con cuatro llamadas que proporcionan la funcionalidad necesaria al usuario, todo ello sustentado sobre la estructura *state_machine* (ver Tabla 4.1), que almacena todos los datos de configuración necesarios para hacer funcionar la máquina de estados.

`struct state_machine {`	
`state_function *function;`	Array de punteros a funciones de estado

`byte`	`**transition_table;`	Tabla de transiciones
`byte`	`num_states;`	Número total de estados
`byte`	`num_transitions;`	Número total de transiciones
`byte`	`entry_state;`	Estado inicial
`byte`	`exit_state; };`	Estado final

Tabla 4.1: formato estructura state_machine

La interfaz de la herramienta máquina de estados está contenida en el fichero *sm.h* e implementada en el archivo *sm.c*. Se detallan a continuación las funciones proporcionadas por dicha interfaz:

- `config_state_machine(struct state_machine *machine,...)` : inicializa la máquina de estados. Para ello inicializa la tabla interna de transiciones, con el número de estados y el número de transiciones solicitado y fija el estado final y el inicial de la máquina.
- `add_state_function(struct state_machine *machine,...)` : añade la función con el código que será ejecutado cuando la máquina de estados visite el nodo del estado especificado.
- `add_transition(struct state_machine *machine,...)` : añade una transición o camino desde un estado origen a un estado destino.
- `init_state_machine(struct state_machine *machine)` : inicia la ejecución de la máquina de estados, con la configuración de estados y transiciones realizada hasta el momento de la llamada.

La herramienta cola, formada por los archivos *cola.h* y *cola.c*, implementa la funcionalidad del tipo abstracto de datos cola circular sobre una estructura array. Es una estructura de datos FIFO, es decir almacena y extrae los datos en el orden estricto de llegada (*first in, first out*).

Desde el punto de vista de la implementación, el pilar sobre el que se sustenta la funcionalidad de la cola, es la estructura *cola_circular* (ver Tabla 4.2), que almacena toda la información necesaria para la gestión de la cola.

`struct cola_circular {`		
`void`	`*__encola;`	Puntero a la posición donde se almacenará el próximo nuevo elemento
`void`	`*__desencola;`	Puntero a la posición del primer elemento de la cola
`void`	`*__inicial;`	Puntero a la primera posición del array
`void`	`*__final;`	Puntero a la última posición del array
`int`	`__size;`	Tamaño en bytes del tipo de dato de los elementos de la cola
`int`	`__len;`	Número de elementos que contiene la cola
`int`	`__max; };`	Longitud máxima de la cola en número de elementos

Tabla 4.2: formato estructura cola_circular

Las siguientes funciones forman la interfaz de gestión de la cola:

- `init_queue(cola_circular *cc, int size, int length)` : inicializa la estructura *cola_circular*. Al ser una cola con tipo genérico, es necesario que el usuario especifique el tamaño del tipo de dato que se va a almacenar en la cola, a través del parámetro *size*. Y la longitud de la cola en número de elementos de ese tamaño a través del parámetro *length*.

- `push(cola_circular *cc, const void *data)` : esta función nos permite almacenar un dato en la cola. Conforme se vayan añadiendo nuevos elementos se irán agregando al final de la cola, hasta que el buffer se llene y la función no permita añadir más elementos.
- `pop(cola_circular *cc, void *data)` : es la operación contraria a *push*. Nos permite extraer elementos de la cabeza de la cola, hasta que no queden más elementos que extraer.
- `length(cola_circular *cc)` : esta operación nos devuelve el número de elementos que contiene la cola actualmente.

4.3 Capa ev3comm

La capa *ev3comm*, es la encargada de establecer una interfaz de abstracción entre la API de programación de red y el usuario. Genera una única instancia de una máquina de estados (Figura 4.3) que se encarga de gobernar las comunicaciones del proceso. Dado que la configuración de la comunicación es punto a punto, solo vamos a tener en un extremo un PC y en el otro extremo un dispositivo EV3. Esta sencillez en el esquema de comunicación, unido a que la mayoría de aplicaciones de control deben cumplir unos requisitos temporales, es otra razón a añadir a las ya mencionadas en el apartado 2.3, que nos hace desechar el protocolo TCP y elegir UDP/IP (ver Figura 4.2) como pila de protocolos sobre la que se apoya nuestra librería de comunicaciones. No necesitamos la mayoría de servicios que ofrece el protocolo TCP, pero aún así ofreceremos al usuario la opción de utilizar un control de flujo ligero, implementado en el protocolo de la capa *ev3comm*. No es tan sofisticado como el control de flujo de TCP, pero al mismo tiempo libera a la comunicación de algunos bytes no necesarios en la cabecera de los paquetes.

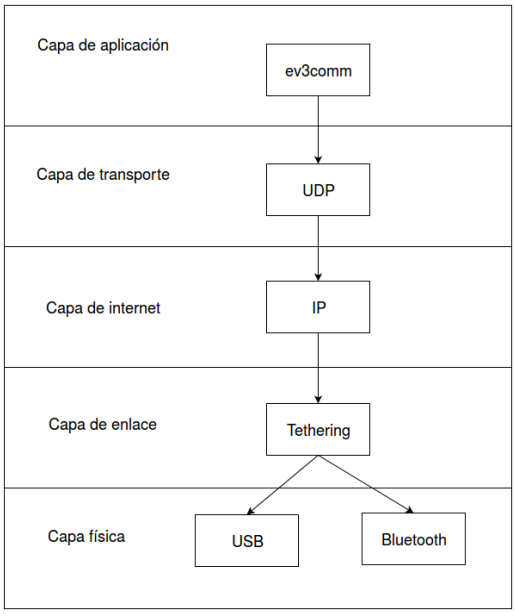

Figura 4.2: Pila de protocolos simplificada

El protocolo de comunicaciones implementado en la capa *ev3comm*, cuenta con los tipos de trama especificados en la Tabla 4.3. Su misión es asegurar la comunicación extremo a extremo, y proporcionar un servicio de control de flujo ligero utilizando las tramas de tipo `MSG_C` (mensaje con confirmación) y las tramas `ACK` para su confirmación.

Tabla 4.3: Tipos de tramas disponibles

Tipo de trama	Descripción
`NUL`	Trama vacía
`ACK`	Trama de confirmación
`HELLO`	Trama de inicialización de la comunicación
`MSG`	Trama de mensaje
`MSG_C`	Trama de mensaje con confirmación
`CLOSE`	Trama de finalización

A continuación explicamos la interfaz con la capa superior:

- `enviar(char *buffer, int len)` : esta función deberá ser llamada cada vez que se desee enviar un paquete de datos de tipo MSG a través de la librería de comunicación.
- `recibir(char **buffer)` : esta llamada debe ser invocada cada vez que nuestro programa demande la recepción de mensajes de tipo MSG del dispositivo remoto.
- `enviar_c(char *buffer, int len)` : el concepto es el mismo que el de la llamada enviar, con la diferencia de que una vez enviado el paquete de tipo MSG_C, la llamada espera una confirmación de mensaje recibido.
- `recibir_c(char **buffer)` : función análoga a la llamada `enviar_c`, pero en el caso de recepción de datos. Después de recibir los datos, se enviará una confirmación de paquete recibido al emisor.
- `cerrar()`: esta llamada termina la comunicación enviando un paquete `CLOSE` al otro extremo y libera los recursos asociados a la librería.

Y funciones adicionales para modificar la configuración por defecto de la librería:

- `config_timeout(time_t segundos, suseconds_t microsegundos)` : modifica el tiempo por defecto que la llamada `recibir` se quedará bloqueada a la espera de interceptar un nuevo paquete entrante.
- `config_broadcast_address(char *addr)` : modifica la dirección broadcast asignada por defecto. Es útil a la hora de buscar el dispositivo EV3 en otras subredes.
- `config_ports(int rx_port, int tx_port)` : permite modificar los puertos de envío y recepción por los que escucha la librería de comunicaciones.

Para proporcionar estas funcionalidades al usuario, la librería implementa una máquina de estados (Figura 4.3) subyacente que se encarga de realizar todo el trabajo. Y un conjunto de primitivas, que encapsulan las llamadas a las funciones del sistema operativo proporcionando a la librería una interfaz que facilite el portado a otros sistemas operativos. Vamos a explicar a continuación su funcionamiento interno.

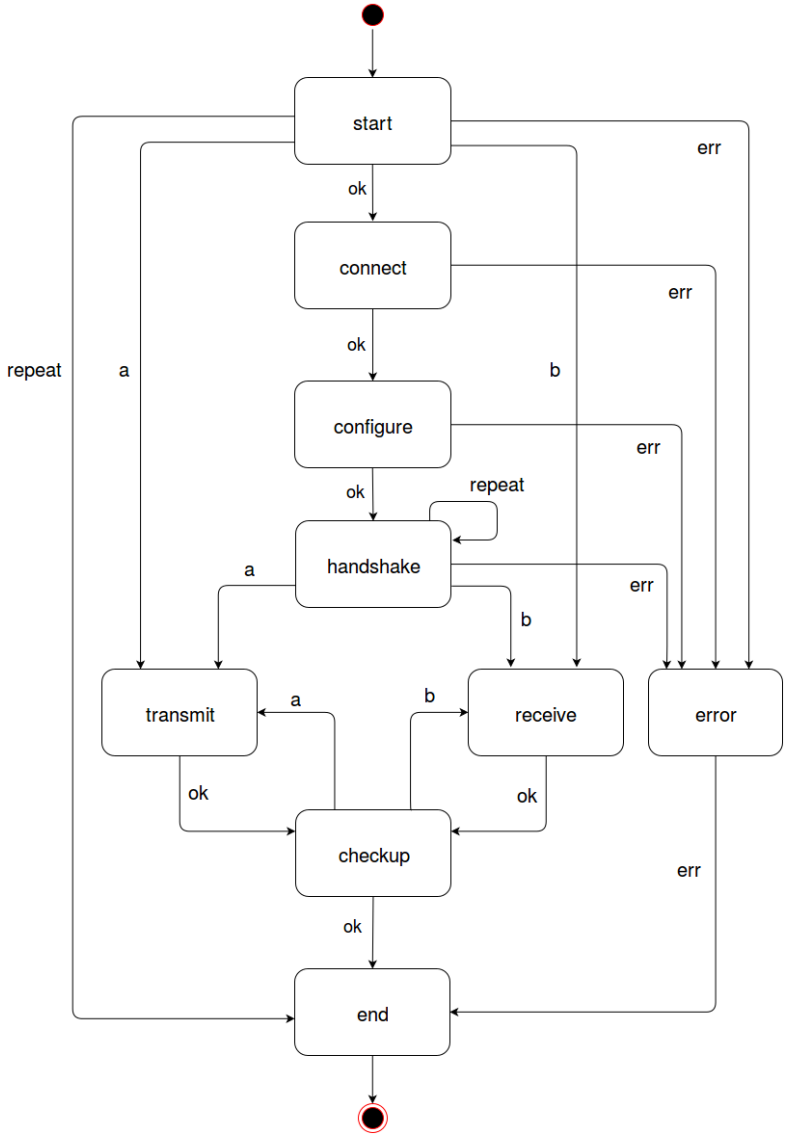

Figura 4.3: Máquina de estados simplificada (ev3comm)

Gran parte de la funcionalidad proporcionada por la máquina de estados se consigue gracias al uso del registro `status`. Una variable global que junto a las transiciones, ayuda a la máquina de estados a decidir el flujo de ejecución. En la Tabla 4.4 se resumen el uso de los bits del registro `status`.

Tabla 4.4: Organización del registro *status*

Bit	Nombre	Descripción
0x01	`FSM`	Indica si la máquina de estados ha sido configurada
0x02	`FSS`	Indica si el socket ha sido configurado e inicializado
0x04	`FRX`	Indica que la operación elegida es de recepción de datos
0x08	`FTX`	Indica que la operación elegida es de emisión de datos
0x10	`FAC`	Indica que la operación requiere confirmación
0x20	`ADR`	Indica que se ha configurado un dirección de broadcast

0x40	TIM	Indica que se ha configurado un timeout diferente al por defecto
0x80	ERR	Indica que ha surgido un error
0x100	FAR	Indica que la recepción ha sido aceptada
0x200	FAT	Indica que la transmisión ha sido aceptada
0x400	FRE	Ante un fallo, indica si debe repetirse la operación o no
0x800	FHR	Ante un fallo en el handshake, indica si debe repetirse el intento
0x1000	BND	Indica un fallo al realizar la operación de enlace
0x2000	SET	Indica un fallo en la configuración del timer
0x4000	NDR	Indica que no se han recibido datos
0x8000	FCL	Marca la orden de finalización de la comunicación

La ejecución en cualquier ciclo de la máquina, comienza en el estado `start`. Los estados `connect`, `configure` y `handshake` están dedicados a la configuración y establecimiento de la comunicación y por tanto solo se ejecutan cada vez que sea necesario iniciar la comunicación, bien porque es la primera ejecución o porque se ha perdido la conexión. Una vez que la comunicación está establecida, el bit `FSS` del registro `status` se pone a uno y se pasa al estado `transmit` o al estado `receive`, en función de la operación a realizar. Una vez realizada la operación de transmisión o recepción de datos, se visita el nodo `checkup`, que chequea que la comunicación se haya realizado correctamente. Si ha ocurrido algún error, el estado chequea el bit `FRE` y si está puesto a uno entonces se repite la operación una vez más.

Este es el funcionamiento general de la máquina de estados. Para terminar de exponer la implementación, a continuación se explica la funcionalidad de cada estado por medio de su pseudocódigo.

El estado inicial `start` actúa como encaminador de la ejecución de la máquina de estados (ver Tabla 4.5). Consulta bits del registro `status` y en función de su valor, retorna una transición haciendo variar la ruta de la máquina de estados. La máquina de estados contempla un conjunto de cinco transiciones posibles, {`ok, err, b, a, repeat`} como podemos ver en la Figura 4.3.

```
si el bit FSS está a cero
        retornamos la transición ok
fin si
si el bit FTX está a 1
        retornamos la transición a
fin si
si el bit FRX está a 1
        retornamos la transición b
fin si
Si el bit FCL está a 1
        retornamos la transición repeat
fin si
retornamos la transición err
```

Tabla 4.5: pseudocódigo estado start

El estado `connect` inicializa el socket UDP. El estado `configure` configura el timeout de la operación recibir y las estructuras addr (Tabla 4.6) necesarias para indicar al socket los puertos de emisión y recepción y la dirección IP a la que enviar el mensaje. En el caso del dispositivo master (el PC), se configuran las estructuras addr para enviar tramas broadcast y recibir tramas de cualquier dirección. Mientras que en el caso del dispositivo slave (EV3), no se configura la dirección de envío y no se restringe la dirección de recepción.

`struct sockaddr_in {`		
`u_short`	`sin_len;`	No utilizado por el usuario
`short`	`sin_family;`	Protocolo que vamos a utilizar
`u_short`	`sin_port;`	Puerto al que se asocia el socket
`struct in_addr`	`sin_addr;`	Dirección IP a la que se asocia el socket
`char`	`sin_zero[8]; };`	No utilizado por el usuario

Tabla 4.6: formato estructura addr

El estado `handshake` establece la comunicación extremo a extremo. En el caso del dispositivo master (PC), envía una trama broadcast de inicialización (*`HELLO`*) a través de la subred asignada, buscando algún dispositivo EV3 con el que conectar. Si el dispositivo EV3 existe y recibe el paquete, lo que hará a continuación es enviar al equipo master un mensaje tipo *`ACK`*, confirmando su presencia y dando lugar al establecimiento de la comunicación. En la Tabla 4.7 se expone el pseudocódigo de este estado.

```
programamos el timeout de la librería

activamos el flag de operación broadcast

transmitimos la trama HELLO a la dirección broadcast de la subred

realizamos una llamada a la función de recepción de datos

si el paquete recibido contiene una trama ACK
        reconfiguramos la estructura addr de transmisión incluyendo la dirección fuente del ACK

        suprimimos el flag broadcast del socket

        ponemos a 0 el bit FHR

        si el bit FTX está a 1
                retornamos la transición a
        sino si el bit FRX está a 1
                retornamos b
        fin si
fin si

si el bit FHR está a 0
        ponemos el bit FHR a 1
```

```
        retornamos la transición repeat
fin si

retornamos la transición err
```

Tabla 4.7: Pseudocódigo de estado handshake para el dispositivo master

En el caso del dispositivo slave (EV3), al ejecutar el estado `handshake` la máquina entra en un estado de bloqueo temporal en la primitiva de recepción hasta que se reciba un paquete de tipo *`HELLO`* o salte el timeout programado. En caso de que se reciba una trama *`HELLO`*, se retorna una trama *`ACK`* para confirmar (ver Tabla 4.8).

```
programamos el timeout de la librería

realizamos una llamada a la función de recepción de datos

si el paquete recibido contiene una trama HELLO
        reconfiguramos la estructura addr incluyendo la dirección fuente del HELLO

        transmitimos una trama ACK al emisor

        ponemos a 0 el bit FHR

        si el bit FTX está a 1
                retornamos la transición a
        sino si el bit FRX está a 1
                retornamos b
        fin si
fin si

si el bit FHR está a 0
        ponemos el bit FHR a 1
        retornamos la transición repeat
fin si

retornamos la transición err
```

Tabla 4.8: Pseudocódigo de estado handshake para el dispositivo slave

El estado `transmit` añade la cabecera del tipo de mensaje al mensaje y lo envía al dispositivo remoto. Si el tipo de mensaje enviado es *`MSG_C`* entonces invocará la primitiva de recepción a la espera de llegada de una confirmación. Si salta el timeout antes de recibir el mensaje *`ACK`*, la operación puede repetirse o no en función de la configuración de la máquina. Se detalla el pseudocódigo en la Tabla 4.9.

```
si el bit FAC está a 1
        añadimos la cabecera MSG_C al mensajes
sino
        añadimos la cabecera MSG
```

```
fin si

transmitimos la trama al otro extremo

si el bit FAC está a 1
        configuramos el timeout privado con la configuración del usuario

        llamamos a la función de recepción de tramas

        si el mensaje recibido es un ACK
                ponemos a 1 el bit FAT
        sino si el mensaje recibido es un CLOSE
                ponemos a 1 el bit FCL
        fin si
fin si

retornamos la transición ok
```

Tabla 4.9: Pseudocódigo del estado transmit

El estado `receive` como indica su nombre, se bloquea en espera de recibir un mensaje del dispositivo remoto. En función de si recibe mensaje o no recibe y el tipo de mensaje, toma una decisión u otra. Se muestra en más detalle en la Tabla 4.10.

```
configuramos el timeout

vaciamos el buffer de recepción

llamamos a la función de recepción de tramas

si el bit FAC está a 1 y la trama recibida es de tipo MSG_C
        enviamos una trama ACK al emisor
        ponemos a 1 el bit FAR
sino si la trama es de tipo MSG
        retornamos la transición ok
sino si la trama es de tipo CLOSE
        ponemos a 1 el bit FCL
sino
        ponemos a 1 el bit NDR
fin si

retornamos la transición ok
```

Tabla 4.10: Pseudocódigo del estado receive

El estado `checkup` es otro estado encaminador como el estado `start`. En función de la transición del estado anterior y una serie de bits del registro `status`, repite la operación anterior o nos envía a un estado siguiente como vemos en la Figura 4.3. A continuación se muestra su pseudocódigo:

```
si el bit FTX tiene valor 1 y el bit FAC tiene valor 1
        si el bit FAT tiene valor 0 y el bit FRE tiene valor 1
                ponemos el bit FRE a valor 0
                retornamos la transición a
        fin si
sino si el bit FRX tiene valor 1 y el bit FAC tiene valor 1
        si el bit FAR tiene valor 0 y el bit FRE tiene valor 1
                ponemos el bit FRE a valor 0
                retornamos la transición b
        fin si
fin si

ponemos a 0 el bit FRE
ponemos a 0 el bit FAR
ponemos a 0 el bit FAT

retornamos la transición ok
```

Tabla 4.11: Pseudocódigo del estado checkup

El estado aceptador `end`, termina la ejecución de un ciclo de la máquina de estados (ver Tabla 4.12). Si el ciclo ejecutado es un ciclo de cierre, entonces el estado `end` se encarga de enviar al dispositivo remoto un mensaje de tipo *CLOSE* indicando el fin de la comunicación y libera los recursos empleados por la capa ev3comm.

```
si el bit FCL tiene valor 1
        enviamos la trama CLOSE al otro extremo
        vaciamos el registro status
        cerramos el socket de comunicación
fin si
si el bit ERR tiene valor 1
        vaciamos el registro status
        cerramos el socket de comunicación
        retornamos -1
sino si el bit FRX tiene valor 1 y el bit NDR tiene valor 1
        ponemos a 0 el bit NDR
        retornamos 1
sino
        retonamos la transición ok
fin si
```

Tabla 4.12: Pseudocódigo del estado end

Para finalizar, se muestran dos diagramas que ejemplifican dos usos de la capa *ev3comm* y la interacción entre los objetos que participan. En el diagrama de secuencia de la Figura 4.4 se muestra el funcionamiento de la máquina de estados en una primera ejecución de la llamada `enviar()` de la interfaz *ev3comm*.

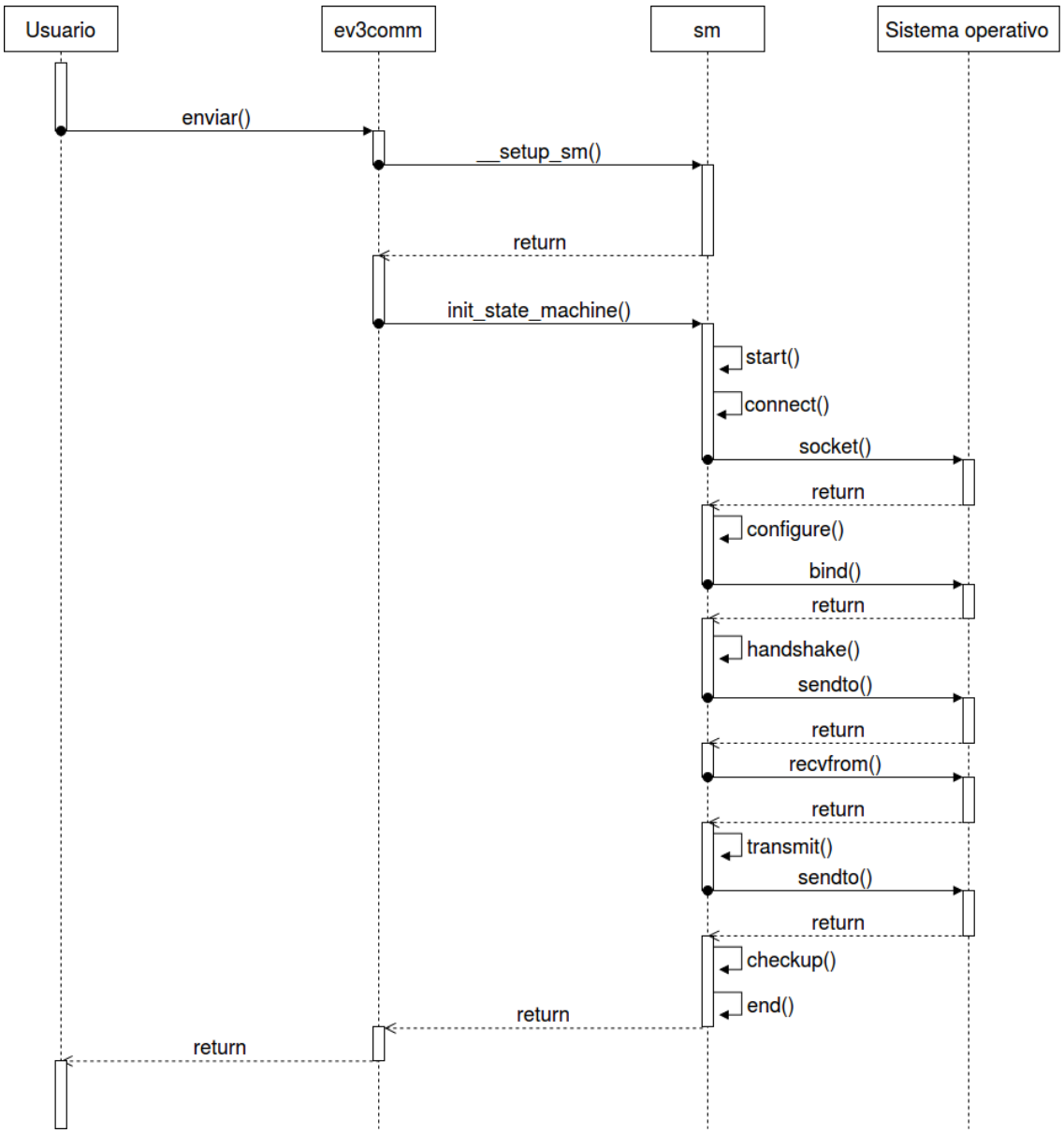

Figura 4.4: diagrama de flujo de una operación enviar() sin inicializar la sm

En la Figura 4.5 se dibuja un diagrama de secuencia de la operación `recibir()` de la interfaz *ev3comm*. Pero en esta ocasión la máquina de estados y la comunicación ya habían sido inicializadas en llamadas anteriores, con lo que se simplifica el flujo de ejecución comparándolo con el de la Figura 4.4.

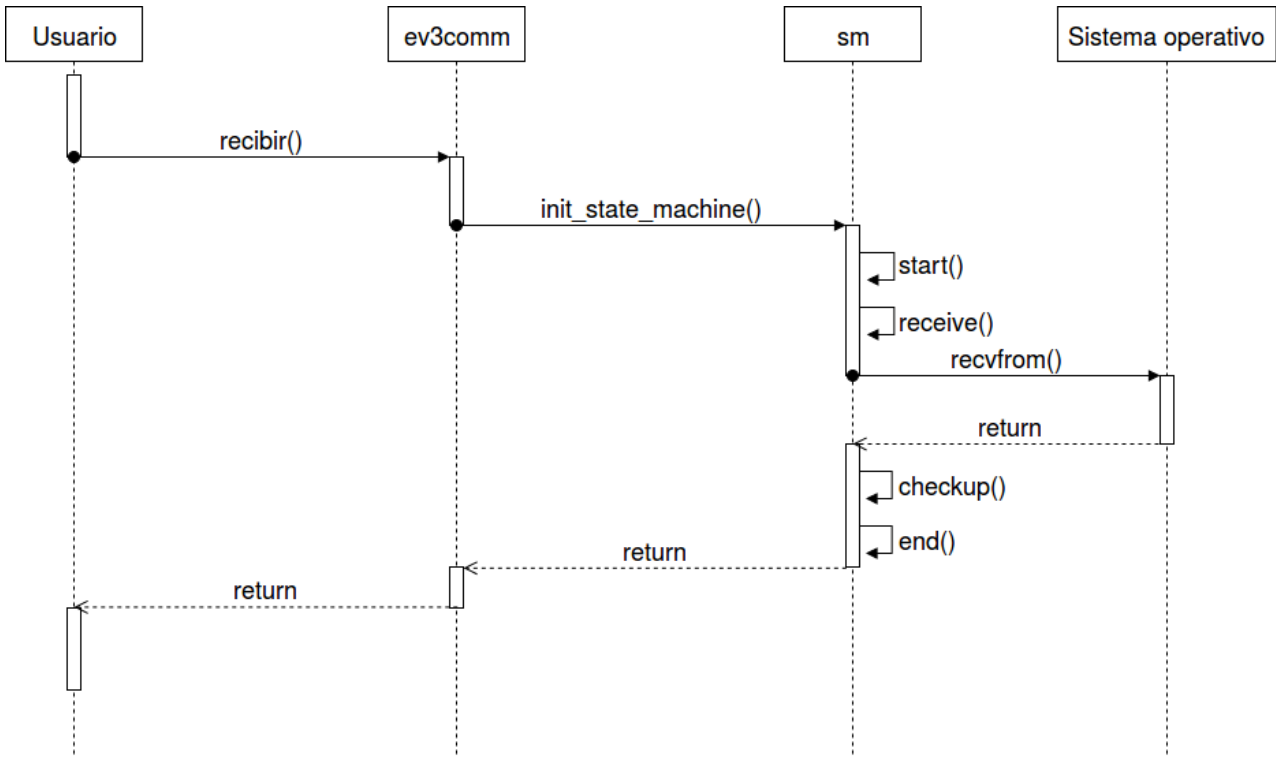

Figura 4.5: diagrama de secuencia de una operación recibir() inicializada la sm

4.4 Capa controller

Ya hemos visto el diseño y el funcionamiento de la capa *ev3comm*. Sobre esta capa se acopla la capa *controller*. La idea de *controller* es la de proporcionar al desarrollador un marco de abstracción en el que programar sus algoritmos de control. Para ello implementa un sistema de ocho colas de recepción de parámetros y ocho colas de envío de parámetros controladas por una máquina de estados (ver Figuras 4.6 y 4.7). A estas colas se las representa en la librería con el nombre de puertos.

Cada puerto o cola, está asociado a un puerto físico del dispositivo EV3. El puerto o cola `port_a` tiene el mismo nombre que el puerto *a* del dispositivo EV3. Con esto si se quiere enviar un parámetro a un actuador conectado al puerto *b* del dispositivo, se escribe ese parámetro en el `port_b` usando la interfaz de *controller* y la librería se encarga de enviarlo y colocarlo en la cola del `port_b` del dispositivo remoto. De esta forma el código desarrollado por el usuario solo tiene que encargarse de leer de la cola correspondiente con las funciones que proporciona la librería y escribir el parámetro recibido en el actuador mediante el uso de la librería ev3c o usarlo en el cómputo de su algoritmo. En este sentido la capa actúa como un multiplexor/demultiplexor (Figura 4.6), multiplexando los datos de envío para varios puertos, en un mismo paquete antes de enviarlo al equipo remoto donde se demultiplexarán para ser usados.

En la Figura 4.6 podemos ver un esquema conceptual del diseño de la capa *controller*. A la izquierda se representa el diseño implementado en el PC, con la ejecución del algoritmo de control y que interacciona con distintos puertos. A la derecha tenemos el diseño para el dispositivo EV3, donde se ejecuta un código orientado a la interacción con los sensores y actuadores.

En la sección 5, en el desarrollo del modelo de ejemplo veremos con mayor profundidad una demostración en la que se clarifica el código ejecutable que debe contener cada una de las máquinas.

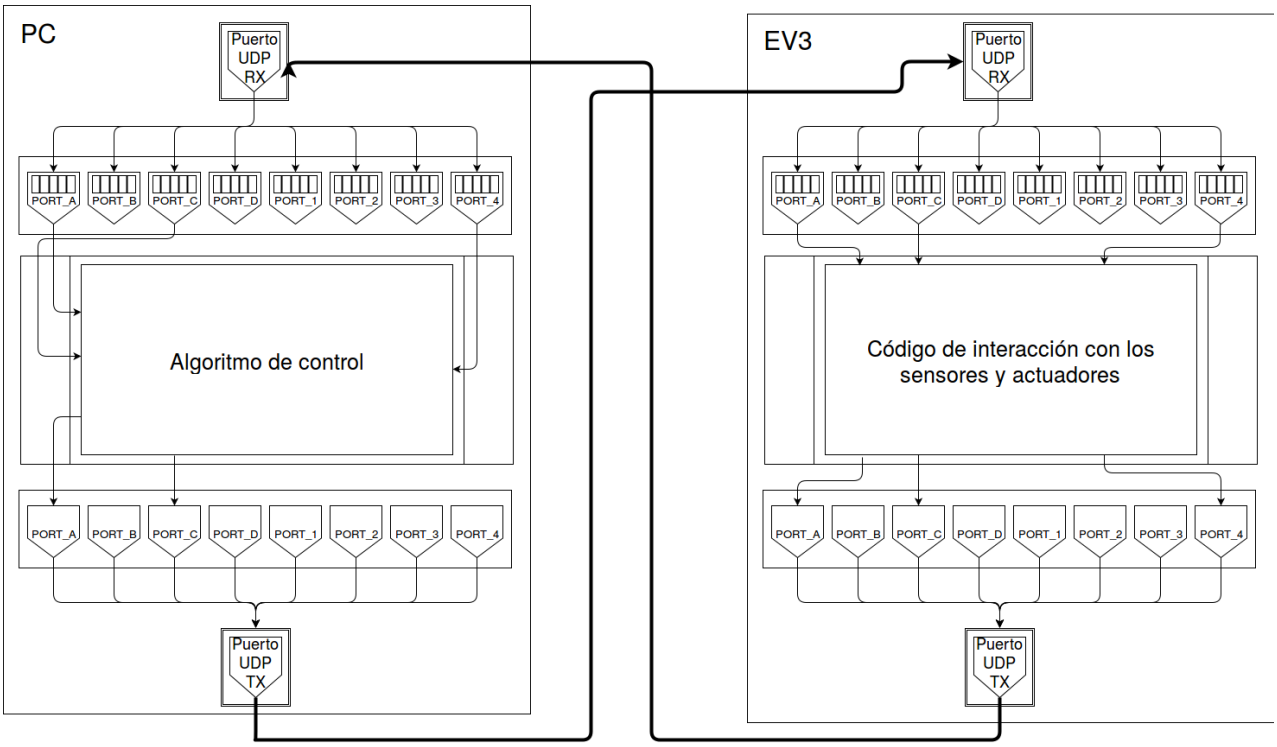

Figura 4.6: Modelo de abstracción (controller)

La librería proporciona un ciclo periódico de ejecución para el código interno de cada máquina. El desarrollador sólo tiene que escribir su algoritmo de control dentro de una función y pasárselo a la librería mediante el uso de la interfaz. Internamente, la librería se encarga de ejecutar periódicamente el algoritmo y proporciona al usuario la posibilidad de modificar el modo de ejecución tanto del *controller* local como del remoto, ejecutado en el dispositivo EV3. En la Tabla 4.13 podemos ver los diferentes modos disponibles en la capa *controller*. Los modos 1 y 2 están reservados para el uso de la aplicación de usuario, no tienen funcionalidad dentro de la librería.

Tabla 4.13: Modos de ejecución

Tipo de trama	Descripción
`MODE_RUN`	Modo de ejecución normal
`MODE_STOP`	Parada de ejecución
`MODE_CONFIGURE`	Modo de configuración de la máquina de estados. Permite inicializar los puertos y modificar el código de ejecución.
`MODE_1`	Modo de usuario 1
`MODE_2`	Modo de usuario 2

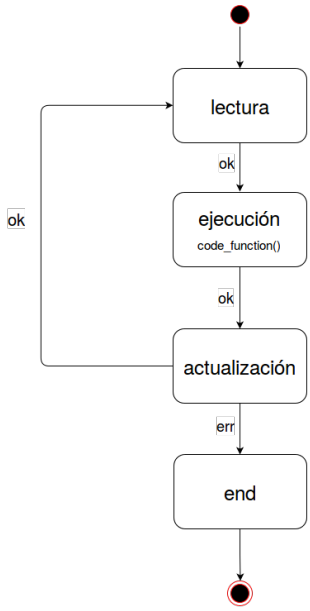

Figura 4.7 Máquina de estados simplificada (controller)

A continuación se recopila el conjunto de funciones que proporciona la interfaz para clarificar la funcionalidad de esta capa:

- `init_ports(int queue_size, int ports)`: esta función inicializa los puertos de entrada/salida indicados en el parámetro `ports`. Es necesario indicarle el tamaño de la cola de recepción deseado. Sólo funciona cuando la máquina está en `MODE_CONFIGURE`.
- `read_port(name_port port, int *data)` : retorna en el parámetro `data`, el primer parámetro de la cola de recepción asociada al puerto `port`.
- `write_port(name_port port, int32_t data)` : escribe el dato dado por el parámetro `data` en la cola de transmisión asociada al puerto `port`.
- `update_code(code_function function)` : permite actualizar la función que encapsula el código de ejecución de usuario, siempre que la máquina esté en `MODE_CONFIGURE`. Este código se ejecuta una vez por ciclo, cuando la máquina de estados llega al estado ejecución.
- `read_mode()` : retorna el modo de ejecución actual de la máquina de estados.
- `change_mode(mode md)` : permite al usuario modificar el modo de ejecución de la máquina de estados local.
- `int send_cmd(mode md)` : permite al usuario modificar el modo de ejecución de la máquina de estados remota.
- `int init_controller()`: permite iniciar la ejecución en, siempre que esté en `MODE_RUN`.

Una vez más, esta funcionalidad se consigue con la implementación de una máquina de estados (ver figura 4.7), apoyada en un registro `__status`, una variable global a las funciones de la capa *controller*, que almacena información sobre el estado de las colas y el modo de ejecución de la máquina (ver Tabla 4.14). En este caso también cuenta con un registro `__init_queues` donde se identifican los puertos inicializados (ver Tabla 4.15). Y dos transiciones distintas: {ok, err}.

Tabla 4.14: Organización del registro __status

Bit	Nombre	Descripción
0x01	port_a_rx	Informa de nuevo mensaje recibido en port_a
0x02	port_b_rx	Informa de nuevo mensaje recibido en port_b
0x04	port_c_rx	Informa de nuevo mensaje recibido en port_c
0x08	port_d_rx	Informa de nuevo mensaje recibido en port_d
0x10	port_1_rx	Informa de nuevo mensaje recibido en port_1
0x20	port_2_rx	Informa de nuevo mensaje recibido en port_2
0x40	port_3_rx	Informa de nuevo mensaje recibido en port_3
0x80	port_4_rx	Informa de nuevo mensaje recibido en port_4
0x100	port_a_tx	Informa de nuevo mensaje disponible para su envío en port_a
0x200	port_b_tx	Informa de nuevo mensaje disponible para su envío en port_b
0x400	port_c_tx	Informa de nuevo mensaje disponible para su envío en port_c
0x800	port_d_tx	Informa de nuevo mensaje disponible para su envío en port_d
0x1000	port_1_tx	Informa de nuevo mensaje disponible para su envío en port_1
0x2000	port_2_tx	Informa de nuevo mensaje disponible para su envío en port_2
0x4000	port_3_tx	Informa de nuevo mensaje disponible para su envío en port_3
0x8000	port_4_tx	Informa de nuevo mensaje disponible para su envío en port_4
0x10000	cmd_tx	Informa a la máquina de la solicitud de envío de un comando de cambio de modo
0x80000	mode_2	Informa de que el modo actual de ejecución es mode_2
0x100000	mode_1	Informa de que el modo actual de ejecución es mode_1
0x200000	mode_configure	Informa de que el modo actual de ejecución es mode_configure
0x400000	mode_stop	Informa de que el modo actual de ejecución es mode_stop
0x800000	mode_run	Informa de que el modo actual de ejecución es mode_run

Tabla 4.15: Organización del registro __init_queues

Bit	Nombre	Descripción
0x01	init_queue_a_rx	Informa de inicialización de cola del port_a de recepción
0x02	init_queue_b_rx	Informa de inicialización de cola del port_b de recepción
0x04	init_queue_c_rx	Informa de inicialización de cola del port_c de recepción
0x08	init_queue_d_rx	Informa de inicialización de cola del port_d de recepción
0x10	init_queue_1_rx	Informa de inicialización de cola del port_1 de recepción
0x20	init_queue_2_rx	Informa de inicialización de cola del port_2 de recepción
0x40	init_queue_3_rx	Informa de inicialización de cola del port_3 de recepción
0x80	init_queue_4_rx	Informa de inicialización de cola del port_4 de recepción
0x100	init_queue_a_tx	Informa de inicialización de cola del port_a de transmisión
0x200	init_queue_b_tx	Informa de inicialización de cola del port_b de transmisión
0x400	init_queue_c_tx	Informa de inicialización de cola del port_c de transmisión
0x800	init_queue_d_tx	Informa de inicialización de cola del port_d de transmisión

0x1000	`init_queue_1_tx`	Informa de inicialización de cola del port_1 de transmisión
0x2000	`init_queue_2_tx`	Informa de inicialización de cola del port_2 de transmisión
0x4000	`init_queue_3_tx`	Informa de inicialización de cola del port_3 de transmisión
0x8000	`init_queue_4_tx`	Informa de inicialización de cola del port_4 de transmisión

Como en el caso de la capa *ev3comm*, para clarificar el funcionamiento haremos un desglose de la implementación de los estados de la librería *controller*, por medio de pseudocódigo.

El estado inicial de la máquina de estados implementada en esta capa es el estado `lectura`. Su misión es inicializar las colas de los puertos de envío y recepción, en el primer ciclo de ejecución de la máquina en modo configuración. Además, invoca la llamada `recibir()` de la capa *ev3comm* para capturar los parámetros enviados por el dispositivo remoto y deserializarlos (demultiplexarlos) en las colas de los diferentes puertos de recepción inicializados. Ver pseudocódigo en Tabla 4.16.

```
si el modo de ejecución es modo configure
        inicializamos las colas solicitadas en el registro __init_queues
fin si

realizamos una llamada al método recibir()

deserializamos los datos recibidos

ponemos a cero los bits de mensajes disponibles para su envío

retornamos la transición ok
```

Tabla 4.16: pseudocódigo del estado lectura

La función principal que realiza el estado `lectura` es la deserialización o demultiplexación del paquete a los puertos disponibles. Es decir, descompone el paquete recibido en mensajes. Cada mensaje contiene una cabecera y un cuerpo. La cabecera indica el puerto al que se envía el mensaje, y el cuerpo contiene el parámetro para el puerto. La función distribuye a cada puerto los parámetros que ha recibido en ese ciclo. Se muestra el código de la deserialización en la Tabla 4.17.

```
extraemos el número de comandos de la cabecera del paquete
para todos los comandos
        si el comando transmite un parámetro a un puerto
                si el puerto está inicializado
                        ponemos a 1 el bit de mensaje recibido a puerto
                        copiamos el parámetro a la cola de puerto
                sino
                        si el comando corresponde con un cambio de modo
                                actualizamos los bits de modo en __status
                        fin si
                fin si
        fin si
fin para
```

Tabla 4.17: pseudocódigo del método de deserialización

El estado ejecución se centra en ejecutar el código periódico especificado por el usuario de la librería.

```
ejecutamos la función suministrada por el usuario
retornamos la transición ok
```

Tabla 4.18: pseudocódigo del estado ejecución

El estado `actualización` produce la operación inversa a la del estado lectura. Examina si hay mensajes nuevos esperando a ser enviados en los puertos de salida en ese ciclo, los empaqueta serializándolos y los envía al dispositivo remoto. Además es el único estado que puede alcanzar al estado final `end`.

En el estado `end` se finaliza la ejecución de la máquina de estados, se liberan recursos y se finaliza la comunicación haciendo una llamada a la función `cerrar()` de la interfaz de *ev3comm*. En la Tabla 4.19 se profundiza en su funcionamiento con la descripción en pseudocódigo.

```
serializamos los parámetros contenidos en las colas de los puertos de envío

realizamos una llamada al método enviar() pasándole como parámetro el paquete serializado

ponemos a cero los bits de mensajes disponibles para ser leídos

si el bit de modo de ejecución STOP está a 1
        retornamos la transición err
fin si

retornamos la transición ok
```

Tabla 4.19: pseudocódigo del estado actualización

A continuación mostramos el pseudocódigo de la operación `serialización`. Analiza los puertos de salida que tienen parámetros disponibles para su envío y los empaqueta en un único mensaje (una secuencia de pares clave-valor, donde la clave es el puerto asignado y el valor es el parámetro escrito) listo para ser enviado (ver Tabla 4.20).

```
para todos los puertos
        si el puerto dispone de un nuevo mensaje
                sumamos 1 al contador u
fin para

reservamos memoria en un buffer temporal para encolar los u mensajes mas la cabecera
inicializamos buffer temporal a cero

para todos los puertos
        si el puerto dispone de un nuevo mensaje
                extraemos el mensaje de la cola
                almacenamos el mensaje en una posición de buffer temporal
        fin if
fin para
```

Tabla 4.20: pseudocódigo del método de serialización de datos

En la Figura 4.8 se dibuja un diagrama de secuencia para clarificar la relación entre los distintos objetos que participan en una ejecución de un ciclo de la capa *controller*.

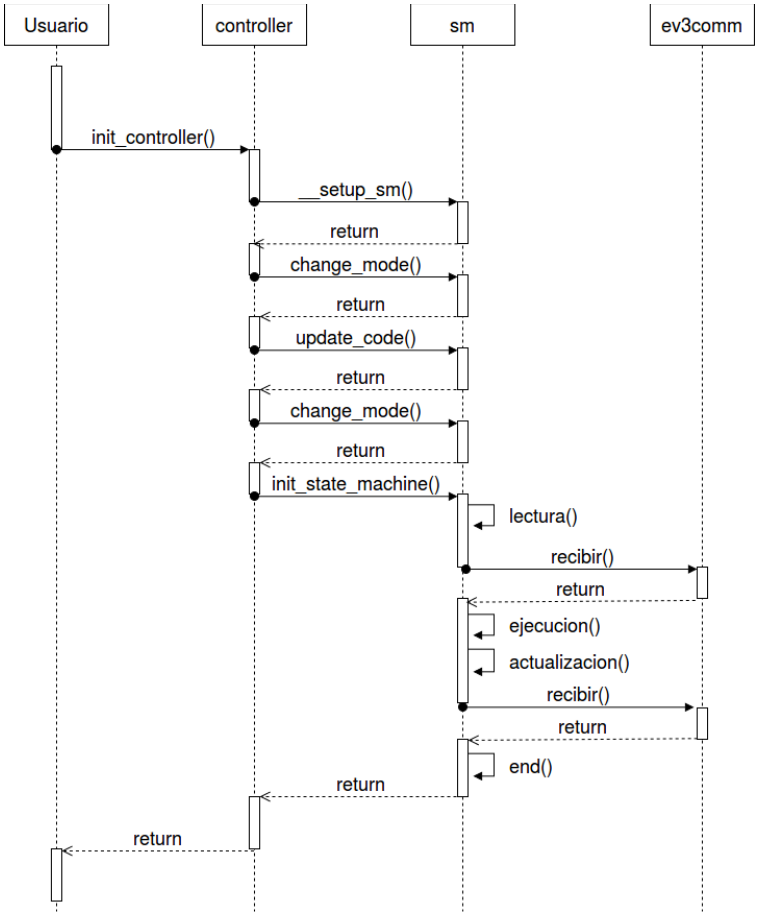

Figura 4.8: diagrama de secuencia un ciclo de ejecución de controller

4.5 Pruebas

La librería cuenta con un modo *debug* individual a cada una de sus capas. El modo *debug* para una capa, se activa especificando el flag de debug concreto de esa capa a la hora de invocar el compilador. El modo *debug* imprime información por consola de la ejecución de la capa.

Para demostrar el correcto funcionamiento de la librería, se han realizado aplicaciones de test por cada capa en modo *debug*. Primero se probaron las herramientas *cola* y *máquina de estados* de manera individual. Se comprobó que la máquina de estados enlaza correctamente los estados mediante las transiciones y que la tabla de estados se rellena correctamente. Se construyeron distintas máquinas de estados y se comprobó su correcta ejecución. En el caso de la cola, se comprobó el funcionamiento de toda su interfaz.

Después se realizó una aplicación de prueba para ilustrar la funcionalidad de la capa *ev3comm*, enviando mensajes en formato char y en formato estructura con varios campos. Se analizó el protocolo de `handshake` con la herramienta de monitorización de red: Wireshark, escuchando la interfaz creada por el mecanismo Tethering.

Por último, en el caso de la capa *controller*, se realizó una aplicación básica para comprobar el correcto funcionamiento de las funciones de la interfaz y del correcto tratamiento de los datos.

5. Desarrollo de modelo de ejemplo

5.1 Introducción

La teoría de control es el estudio que se encarga del análisis y diseño de un sistema de control. Los principales componentes de un sistema de control básico son la planta y uno o más sensores, que miden las variables que se quieren controlar. La planta de un sistema de control es la parte del sistema a ser controlada (los actuadores).

Los primeros estudios de sistemas de control se basaban en la solución de ecuaciones diferenciales por los medios clásicos. Salvo en los casos simples, el análisis es pesado y no indica fácilmente que cambios deben hacerse para mejorar el comportamiento del sistema. El empleo de la transformada de Laplace simplifica este análisis. Una vez descrito el sistema físico por un juego de ecuaciones matemáticas, estas se transforman para lograr un determinado modelo matemático, que servirá para controlar la planta.

Características principales de todo sistema de control:

- Estabilidad: La respuesta a una señal, debe alcanzar y mantener un valor útil durante un período de tiempo razonable. Un sistema inestable producirá oscilaciones en la señal o bien puede hacer que la señal tome valores de límites extremos.
- Exactitud: Debe ser exacto dentro de los límites de tolerancia especificados. Esto significa que el sistema debe ser capaz de reducir cualquier error a un límite aceptable. No existen sistemas de control que puedan mantener un error cero durante todo el periodo de vida, porque siempre es necesario que exista un error para que el sistema inicie la acción correctora. En muchos sistemas no se requiere una exactitud absoluta, ya que la exactitud crece junto con el coste de desarrollo.
- Rapidez de respuesta: Es necesario que el sistema de control actúe a tiempo.

5.2 Diseño del modelo

Como modelo de ejemplo vamos a construir un robot segway (figura 5.1). Para ello hemos adaptado un algoritmo de control realizado por Laurens Valk[10], a nuestra plataforma. Desde un punto de vista físico, el segway está basado en el problema del péndulo invertido. El algoritmo de control simplifica el sistema a un modelo matemático aproximado como el que vemos en la figura 5.2.

Figura 5.1: Montaje del robot con kit de Lego Mindstorms

Tenemos un modelo formado por dos estructuras. Por un lado el tren inferior, compuesto por las ruedas sobre las que se apoya la segunda estructura, que es el cuerpo del robot. El cuerpo del robot gira libremente alrededor del eje central de las ruedas.

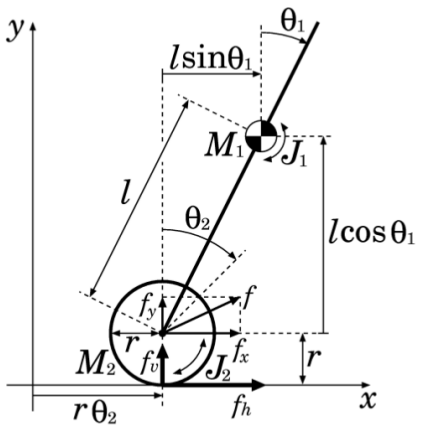

Figura 5.2: Modelo matemático para el robot segway

Para modelar el sistema necesitamos conocer el desplazamiento del robot y el ángulo de giro del cuerpo del robot. No podemos medir directamente el desplazamiento del robot, pero podemos obtener de la lectura de los encoders de los motores, que nos proporciona el número de grados que han rotado las ruedas. Esto nos da una medida del desplazamiento del robot hacia delante o hacia atrás. Para modelar el sistema también necesitamos la velocidad angular a la que giran las ruedas, que también lo obtenemos del encoder de los motores. Y por último, necesitamos controlar la inclinación del cuerpo del robot, para compensar la caída y hacer que el sistema retorne a su posición de equilibrio. Para ello utilizamos la velocidad angular y la posición angular que nos proporciona el giroscopio moviéndose solidariamente al cuerpo del robot.

Se muestra a continuación una descripción de estos dos componentes del kit Lego necesarios para la construcción del robot:

- Giroscopio. Es un sensor digital que detecta el movimiento rotativo entorno a un eje. Si movemos el sensor en la dirección que señalizan las flechas (ver Figura 5.3), el giroscopio detecta tanto la velocidad de giro en esa dirección, como el ángulo total de rotación. Tiene una precisión en modo ángulo de ± 3 grados por 90 grados de giroscopio. Mientras que en modo giro puede medir hasta un máximo de 440grados por segundo. Ratio de muestreo: 1kHz.

Figura 5.3: giroscopio

- Motor grande. 160 - 170 rpm con un torque de 20Ncm y un torque de bloqueo de 40Ncm. Incluye un encoder para medir la rotación con 1 grado de resolución.

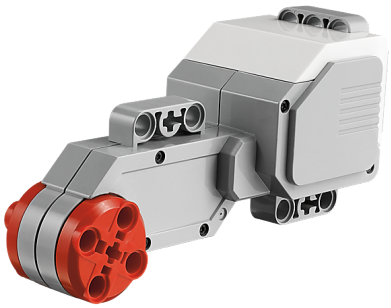

Figura 5.4: motor grande

El algoritmo de control de balanceo, está basado en el algoritmo utilizado por el MIT para la construcción del robot Balancer[10]. El núcleo del algoritmo de balanceo está formado por las ecuaciones de la tabla 5.1.

```
// Cómputo de las ecuaciones lineales
motorDutyCycle = (gainGyroAngle  * gyroEstimatedAngle
                + gainGyroRate   * gyroRate
                + gainMotorAngle * motorAngleError
                + gainMotorAngularSpeed * motorAngularSpeedError
                + gainMotorAngleErrorAccumulated * motorAngleErrorAccumulated);
```

Tabla 5.1: ecuaciones para el cómputo de la potencia de los motores

Para hacernos una idea de la función de cada una de estas ecuaciones lineales a la hora de mantener el robot en equilibrio, vamos a verlas termino a término. En primer lugar imaginemos que el robot está completamente vertical y en su posición deseada de cero grados con respecto al eje vertical. En ese caso, las cuatro variables del cómputo de ecuaciones lineales serán cero y el resultado del cómputo también será cero. El robot no está cayendo hacia adelante ni hacia atrás, por lo tanto *gyroEstimatedAngle* será cero y *gyroRate* también será cero.

Ahora imaginemos que todos los términos son cero salvo la velocidad angular que es positiva. El robot no se desplaza de la posición actual, pero cae hacia delante porque la velocidad angular del cuerpo del robot es positiva. Éste término permite al robot responder. Además de permitir responder a esa caída, también evita que el término *gyroEstimatedAngle* haga que el robot responda cuando en realidad no está cayendo.

Y en cuanto a los dos términos referentes a los motores, supongamos que el robot está en la posición de referencia pero se ha movido 10 radianes adelante de la posición deseada. En ese caso *motorAngleError* será 10 radianes. Si intentamos mover el robot hacia atrás con el fin de volver a la posición original, el robot caerá hacia delante. Con el fin de volver hacia atrás debemos moverle hacia delante para que caiga hacia atrás, y después moverle hacia atrás para compensar la caída y dirigir al robot a su posición original.

El término *motorAngularSpeedError* funciona de la misma manera que *motorAngleError*.

5.3 Adaptación a ev3c

En esta sección se muestran las partes más interesantes del algoritmo de control para mantener estable el balanceo del robot segway. El código mostrado no ha sido adaptado al uso de la librería de comunicaciones, eso lo veremos en el siguiente apartado. Se ha adaptado el algoritmo mediante el uso de la librería ev3c y la ejecución del programa se hará por completo en el dispositivo EV3, sin comunicación con el exterior.

Antes de comenzar a ejecutar el lazo de control, tenemos que encargarnos de inicializar el giroscopio y los dos motores que vamos a utilizar. Para ello utilizamos la interfaz de la librería ev3c. Se expone a continuación un resumen con los pasos para inicializar los sensores y actuadores.

Antes de interactuar con ningún sensor ni actuador, la librería ev3c requiere que se busquen los dispositivos conectados tal y como vimos en el apartado 3.3. Una vez localizados el giroscopio y los motores, debemos configurarlos:

- El giroscopio se calibra con un cambio de modo a GYRO_CAL, mediante la llamada a *ev3_mode_sensor()*. Esto establece la posición angular actual como posición de referencia. Una vez calibrado, se debe cambiar al modo GYRO_RATE, para que en cada lectura se obtenga la velocidad angular.
- Los motores deben ser inicialmente reinicializados con *ev3_reset_motor()*. Después se asigna un modo de parada del motor, en nuestro caso “hold”, con una llamada a *ev3_stop_command_motor_by_name()*. Este modo, frena la rotación del eje del motor cuando se da la orden de parar, y le dice al motor que intente mantener la posición aplicando un par al eje de giro, de sentido contrario a la fuerza externa aplicada al eje. Y por último, hay que indicarle a los motores el modo de funcionamiento, en nuestro caso *"run-direct"*. Este modo le indica al motor que mantenga la última potencia de giro indicada, hasta que se no se indique otra potencia distinta.

Una vez que los dispositivos ya están inicializados, podemos comenzar a usar los periféricos. Durante la ejecución del algoritmo de control, se necesitan realizar lecturas y escrituras a los motores y al giroscopio. Esto se consigue con las llamadas contenidas en la Tabla 5.2.

```
// Lectura de la velocidad angular dada por el giroscopio
ev3_update_sensor_val(gyro_sensor);
gyroRateRaw = gyro_sensor->val_data[0].s32;

// Actualización de la potencia suministrada a los motores
ev3_set_duty_cycle_sp(left_motor, 0);
ev3_set_duty_cycle_sp(right_motor, 0);

// Lectura de la posición angular de los motores
motorAngleRaw = (ev3_get_position(left_motor) + ev3_get_position(right_motor))/2;
```

Tabla 5.2: código de inicialización de los motores

5.4 Adaptación al uso de la librería de comunicaciones

En esta sección, se actualiza el código de la aplicación del apartado anterior para incluir el uso de la funcionalidad implementada por nuestra librería de comunicaciones.

El primer paso es separar el código que debe ejecutarse en el PC del código que debe ejecutarse en el dispositivo EV3.

En el PC deben ejecutarse los cálculos del algoritmo de control. Para ello el usuario debe encapsular el código en la función `ob()`, que sigue un prototipo fijado por la capa *controller*. En la misma función, antes del código del algoritmo, el usuario debe incluir una llamada a la función `init_ports()` de la capa *controller*, indicando los puertos que va a usar la aplicación y dos llamadas a la función `send_cmd()` como se indica en la Tabla 5.3. `send_cmd` nos permite enviarle al dispositivo EV3 la señal para que se configure e inicialice sus sensores y actuadores. Las llamadas ev3c para lecturas y escrituras del giroscopio y los motores en el algoritmo de control, deben ser sustituidas por las funciones `read_port()` y `write_port()` de la capa *controller.* Se le debe pasar a estas funciones como argumento, los puertos del dispositivo EV3 a los que están conectados los sensores y actuadores. La aplicación de usuario debe hacer una llamada a `init_controller()` para iniciar la ejecución.

```
// Se repite en cada ciclo en la máquina de estados
void ob()
{
        // Inicializa los puertos usados por el robot, sólo funciona en MODE_CONFIGURE
        init_ports(...);

        // Envía comandos de cambios de modo para configurar el EV3
        si el modo de configuración es MODE_CONFIGURE
                send_cmd(mode_configure);
                send_cmd(mode_run);
        finsi

        // El código del algoritmo adaptado a la librería de comunicaciones
        Código del algoritmo de control, con llamadas a read_port() y write_port()

        // Fin de la ejecución (o pulsando ctrl+c desde el PC)
        si se cumple la condición fijada por el usuario
                send_cmd(mode_stop);
                change_mode(mode_stop);
        finsi
}
```

Tabla 5.3: código de ejecución para el controller del PC

En el dispositivo EV3 debe ejecutarse el código de interacción con el giroscopio y los motores. Para ello se encapsula en la función `ob()` el mismo código de inicialización de motores y sensor que hemos visto en el apartado anterior, acompañado por las llamadas de escritura y lectura de la librería ev3c y de la correcta inicialización de los puertos, tal como se recoge en la Tabla 5.4. Como en el caso anterior, el usuario debe crear una aplicación donde llame a `init_controller()`.

```
// Se repite en cada ciclo en la máquina de estados
void ob()
{
	// Inicializa los puertos usados por el robot, sólo funciona en MODE_CONFIGURE
	init_ports(...);

	// Inicialización de motores y giroscopio
	if(read_mode() == mode_configure) {
		Código de inicialización del giroscopio y de los motores
	}

	// Lectura de consignas para motores enviadas por el algoritmo de control del PC
	motorA = read_port(portA);
	motorB = read_port(portB);

	// Actualizamos los valores de la potencia asignada a cada motor
	ev3_set_duty_cycle_sp(left_motor, motorA);
	ev3_set_duty_cycle_sp(right_motor, motorB);

	// Lectura del giroscopio
	ev3_update_sensor_val(gyro_sensor);
	gyro = gyro_sensor->val_data[0].s32;

	// Lectura de posición angular de los motores
	posA = ev3_get_position(left_motor);
	posB = ev3_get_position(right_motor);

	// Envío de las lecturas al PC
	write_port(port1, gyro);
	write_port(portA, posA);
	write_port(portB, posB);
}
```

Tabla 5.4: código de ejecución para el controller del EV3

6. Conclusiones

En el apartado 1.3 se expusieron los objetivos del proyecto. A continuación se resume el trabajo realizado:

- Se ha realizado un estudio sobre la plataforma EV3. Se han examinado las características que ofrece el hardware y cómo manejar el hardware desde el sistema ev3dev, apoyándonos en el uso de la librería ev3c para el manejo de actuadores y sensores.
- Se ha realizado un estudio sobre distintos protocolos de comunicaciones que se pueden aplicar sobre el dispositivo EV3. Se han comparado los protocolos y tecnologías y finalmente se ha escogido la tecnología Tethering y la pila de protocolos UDP/IP para dar soporte a la librería.
- Se ha diseñado e implementado la librería de comunicaciones, encapsulando las distintas funcionalidades en capas y teniendo en cuenta la portabilidad a distintos sistemas. Y se ha comprobado su correcto funcionamiento con la codificación de aplicaciones de test.
- Se ha estudiado una introducción a la teoría de control y se ha buscado un algoritmo de control para resolver el problema del péndulo invertido (robot segway). Se ha estudiado el funcionamiento del algoritmo y se ha adaptado al uso de la librería ev3c y de la librería de comunicaciones.

6.1 Trabajo futuro

Actualmente la librería solo funciona en el sistema operativo Linux. Sería interesante portar la librería a otros sistemas operativos, como Windows por su gran uso entre los usuarios de PC y MarteOS por ser un sistema operativo de tiempo real. Para portarlo a Windows, sólo sería necesario modificar las primitivas que encapsulan las funciones socket de la capa *ev3comm*, para adaptarlas a los sockets de Windows (ligeramente diferentes). El portado a MarteOS requeriría sustituir esas llamadas a la interfaz de socket, porque el sistema no implementa la tecnología Tethering. Una posible solución sería sustituir las llamadas a la interfaz de sockets, por llamadas al driver del puerto serie y adquirir un dispositivo conversor de RS232 a Bluetooth, para poder comunicar de forma inalámbrica con el dispositivo EV3.

Bibliografía

[1] Shahin Farahni. “Zigbee. Wireless Networks and Transceivers”.

[2] Repositorio github “LEGO MINDSTORMS and LEGO WeDo drivers for Linux from the ev3dev project”. https://github.com/ev3dev/lego-linux-drivers

[3] Albert Huang - “An Introduction to Bluetooth Programming” 2005-2008.

[4] Documentación oficial proyecto ev3dev. http://www.ev3dev.org/

[5] Documentación oficial Lego: Mindstorms. https://www.lego.com/es-es/mindstorms

[6] William Stallings - “Comunicaciones y Redes de Computadores” 7ª Edición.

[7] Ogata K. - “Ingeniería de Control Moderna”. Ed. Prentice Hall.

[8] Márquez García - “UNIX, programación avanzada”. Ed. Ra-Ma.

[9] Ryo Watanabe - “Motion Control of NXTway”. Waseda University.

[10] Laurens Valk - “Self-Balancing EV3 Robot”. http://robotsquare.com/2014/07/01/tutorial-ev3-self-balancing-robot/

[11] Apuntes asignatura - “Desarrollo de Software para Sistemas Empotrados”. Michael González Harbour 2016.

[12] Documentación librería ev3c. https://github.com/theZiz/ev3c

[13] Xander Soldaat - “Bot Bench Blog”. http://botbench.com/blog/

[14] Rubén Miguélez García - “Estudio diseño y desarrollo de una aplicación de tiempo real y de un simulador para su comprobación: péndulo invertido”.

[15] Luis Joyanes, Ignacio Zahonero - “Programación en C”. Ed. McGraw Hill.

[16] The Open Group - “The Single Unix Specification, Version 2”. http://pubs.opengroup.org/

[17] Dr. Dobb’s - “Designing a Network Protocol”

[18] Brian “Beej Jorgensen” Hall - “Beej's Guide to Network Programming”.