



Facultad de  
**INFORMÁTICA**



UNIVERSIDAD  
NACIONAL  
DE LA PLATA

Taller de Proyecto 2  
Informe Final

# CYCL<sup>o</sup>PS

A2 - Armado de robot mapeador con sensor LIDAR V5310

Guerrico Leonel - 02009/5

Ossola Florencia - 02226/2

Pérez Balcedo Octavio - 02236/4

**7 de Febrero de 2025**

# Índice

<b>Índice</b>	<b>2</b>
<b>1.1 Introducción</b>	<b>3</b>
1.2 Objetivos y Propuesta	4
1.3 Correcciones Realizadas	5
<b>2 Proyecto</b>	<b>8</b>
2.1 Grafico Procesos y Funciones	8
2.2 Esquemático de Conexiones del Hardware	15
2.3 Estado alcanzado en la funcionalidad	17
<b>3 Documentación del Software</b>	<b>18</b>
3.1 Pre-requisitos	18
3.2 Librerías del ESP32	22
3.2.2 Mapping	24
<b>4 Documentación Relacionada</b>	<b>37</b>
4.1 Interfaz Web	37
4.2 Hardware	39
4.3 Manual de Usuario	41
4.4 Enlaces	42
<b>5 Conclusión</b>	<b>43</b>
5.1 Mejoras	43
<b>6 Bibliografía</b>	<b>45</b>
<b>7 Apéndice A: Materiales y Presupuesto</b>	<b>46</b>

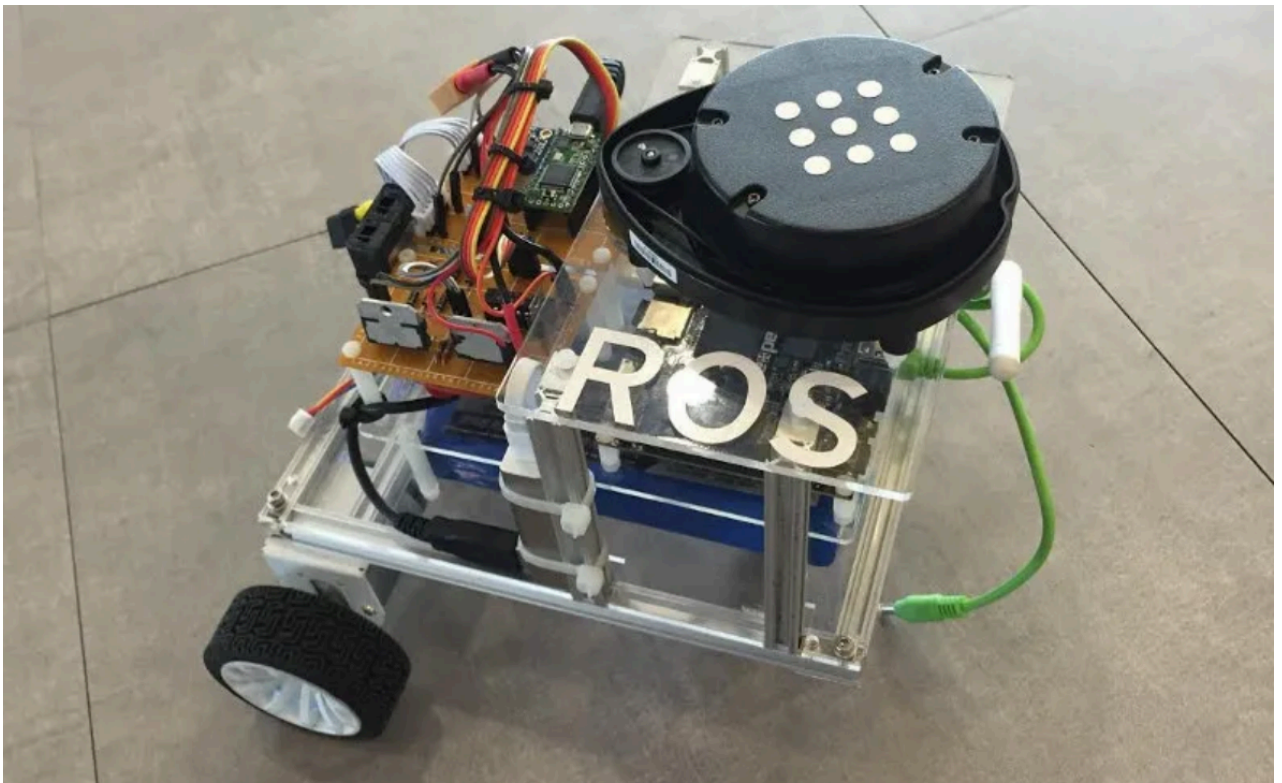
## 1.1 Introducción

Al ingresar a la materia se nos presentaron unas series de posibles proyectos a realizar, y tras analizarlos decidimos elegir el conocido como proyecto - A.2 Armado de sensor LIDAR[1] con V15310-, y al que, para simplificar, apodamos Cyclops, en referencia al héroe de Marvel.

El mismo nos presentaba el desafío de diseñar un vehículo a control remoto el cual contará con la capacidad de detectar los obstáculos de su entorno mediante la utilización de tecnología Lidar; y a partir de esto generar un mapa donde se visualizan dichos obstáculos. Además, se nos planteó la utilización del chip ESP32[2] como el microcontrolador a cargo de todo el sistema.

En tanto que para el control del vehículo y para visualizar dicha información, se nos solicitó implementar una aplicación web.

Este proyecto está inspirado en los robots y drones que se utilizan normalmente para el mapeo de entornos con tecnología LiDAR, y en particular en el proyecto Linorobot del usuario Rud Merriam [3]. El cual es un robot de pequeñas dimensiones equipado con un LiDAR de alta capacidad, el cual realiza un mapa de su entorno utilizando ROS[4].



*Imagen 1: Linorobot de Rud Merriam (fuente: [hackaday.com](https://hackaday.com) )*

## 1.2 Objetivos y Propuesta

Como se comentó anteriormente, el objetivo principal de este proyecto fue desarrollar un robot que se maneja a control remoto, y que mapee su entorno mediante un sensor LiDAR, y una app web como complemento para visualizar los datos provenientes del mismo, así como facilitar su manejo.

Por tanto el proyecto se subdividió de la siguiente manera:

### 1.2.1 Objetivos Primarios

Se propuso como objetivo primario el diseño e implementación de un vehículo a control remoto, el cual utilizara un mcu ESP32 como centro del sistema, y un LiDAR VL53L0X[5] como sensor para el mapeo. Además de una aplicación web para centralizar el manejo del mismo, y visualización de los datos; y por otro lado, el diseño físico del mismo, pensándolo para ser impreso en 3D.

A partir de esto se esquematizan los siguientes funcionalidades principales:

1. Mapeo 2D completo de un área determinada: Implementar el algoritmo y diseño de hardware necesario para que el robot realice un mapeo del área utilizando el sensor LiDAR para detectar obstáculos en el mismo.
2. Medición de distancia: Implementar el algoritmo correspondiente para que el robot realice el cálculo necesario para obtener la distancia entre él y los obstáculos durante el mapeo.
3. Desplazamiento de vehículo: El robot debe poder desplazarse hacia adelante, atrás o doblar según el usuario lo desee en el área en la que se encuentra.
4. Comunicación entre el cliente y el robot: El robot debe ser capaz de responder a las órdenes del usuario, estas pueden ser con respecto al desplazamiento, pausa, entre otras.
5. Visualización de datos mediante una interfaz: Desarrollar una página web donde el usuario podrá visualizar los datos relacionados al mapeo y mensajes importantes del sistema.
6. Montaje de Hardware: Realizar el diseño del robot y montaje de los componentes necesarios para su funcionamiento.

- a. Comunicación entre el microcontrolador y los motores.
- b. Comunicación entre el microcontrolador y el servomotor.
- c. El servomotor debe controlar el movimiento del sensor LiDAR.

### **1.2.2 Objetivos Secundarios**

Además de los objetivos principales, se propusieron una serie de objetivos secundarios para complementar el sistema, los cuales no eran prioritarios. Estos eran:

1. Incorporar baterías para el funcionamiento independiente del vehículo.
2. Incorporar al hardware un INA219[6] para la medición de carga de las baterías, y la lógica para informar de dichos valores al usuario.
3. Diseñar un software para el manejo autónomo del vehículo, es decir, que el vehículo funcione sin necesidad de controlarlo, en base a detectar caminos viables entre los obstáculos.

## **1.3 Correcciones Realizadas**

### **1.3.1 Protocolos de Comunicación ESP32 - Servidor**

Originalmente se implementó HTTP[7] como protocolo de comunicación entre el ESP32 y el Servidor(notebook), no obstante, tras un análisis, se determinó que este protocolo no era conveniente para una comunicación rápida y eficiente, pues no era óptimo para IOT[8], por tanto se lo reemplazó con MQTT[9]. Dicho protocolo nos permite mantener conexiones activas de forma permanente (salvo desconexión), lo que facilita el envío rápido de múltiples datos desde el ESP32 al servidor.

No obstante, posteriormente debido a un problema con MQTT que no se logró solucionar, se volvió a implementar HTTP nuevamente, aunque solo para el envío de instrucciones al ESP32.

### **1.3.2 Servomotor de giro continuo**

Inicialmente se trabajó bajo la idea errónea, de que el servomotor dado, funcionaba de forma similar al servomotor típico sg90[10], lo que implicó el diseño de hardware e implementación de una librería no funcionales.

Posteriormente, tras analizar el servomotor, y entender que él mismo era de giro continuo, se rediseñó todo el sistema, ya que dichos servomotores, no proveen información alguna, solo giran en el sentido indicado.

Esto implicó un rediseño completo de la lógica y hardware, pues ahora se debió incluir un sensor de fin de carrera, de tal modo que al ser presionado por el servomotor, se genere a nivel software un valor de referencia, a partir del cual se implementa todo el comportamiento, como la inversión del sentido, cálculo de ángulo, y demás.

### **1.3.3 Sensor INA219**

Como parte de los objetivos secundarios se propuso la inclusión de un sensor INA219, para el sensado del nivel de batería, lo que se tuvo en cuenta desde un primer momento en el diseño del circuito. No obstante, por una mala lectura de la documentación, se colocó el sensor en paralelo a la carga, cuando se debe conectar en serie, pues mide la corriente y no la tensión.

Debido a esto, se tuvo que corregir el circuito, para poder utilizarlo correctamente.

### **1.3.4 I2C**

Como parte central del proyecto se diseñó un circuito pensándolo para implementarlo como un PCB de una capa, el cual se llevó a cabo, de cara al ensamblado final del vehículo..

Inicialmente el mismo pareció funcionar correctamente, hasta el momento de testear los sensores, momento en el cual ninguno de los dos(INA219 y LiDAR VL53L0X) respondía. Tras múltiples testeos y análisis, se llegó a la conclusión de que el diseño estaba mal, y que no se consideró como el ruido afectará a las pistas I2C[11], así como se debían ubicar las mismas, para que no se vieran afectadas por las demás. Esto implicó el rediseño completo del circuito, y el descarte del PCB[12] hecho.

### 1.3.4 Memory Leak

Durante la etapa de pruebas para dejar funcionando el robot, se detectó memory leak, lo cual provocaba el crasheo del sistema en el ESP32. Para solucionar esto se realizaron pruebas exhaustivas para detectar el origen del leak, y solucionarlo. Dando como resultado::

- Manejo de JSON: como formato a la hora de la comunicación, se utiliza el formato JSON. Y para utilizarlo en el ESP32 se optó por la librería cJSON, una de las más extendidas y prácticas de utilizar. No obstante, al testear se encontró que era extremadamente ineficiente en el uso de memoria, dado que esta utiliza internamente listas enlazadas, una estructura de datos altamente demandante de memoria. Razón por la cual se la reemplazó por [\*frozen\*](#), la cual es mucho más eficiente, por ejemplo: haciendo el envío de datos 10 veces más rápido.
- Manejo de Punteros: como parte de muchas funcionalidades se debió utilizar punteros, algunos de los cuales al no manejarse adecuadamente, en concreto, liberarse; al cabo de un minutos ejecutando, colapsaban la memoria.

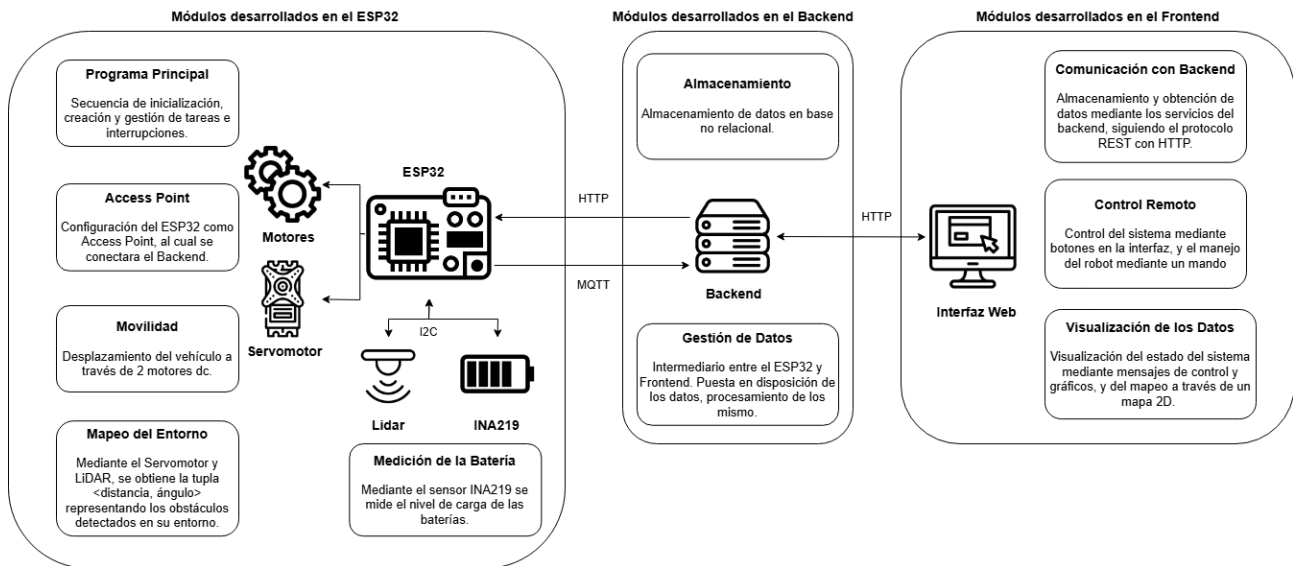
### 1.3.5 Conexión Backend Broker

Uno de los problemas que tuvimos al preparar la presentación, fue la conexión del Backend a MQTT, especialmente en la notebook para la demostración del proyecto, algo que sucedía en otra de las computadoras. Tras sucesivas pruebas, resultó que `http://localhost:8080` no funcionaba, y que al reemplazar localhost por la IP que le asigna el ESP32 al servidor, esto se solucionaba.

Cabe aclarar que dicha IP al ser único dispositivo que se conecta al ESP32, es siempre la misma, por lo que es seguro hacerlo de esta forma.

## 2 Proyecto

### 2.1 Grafico Procesos y Funciones



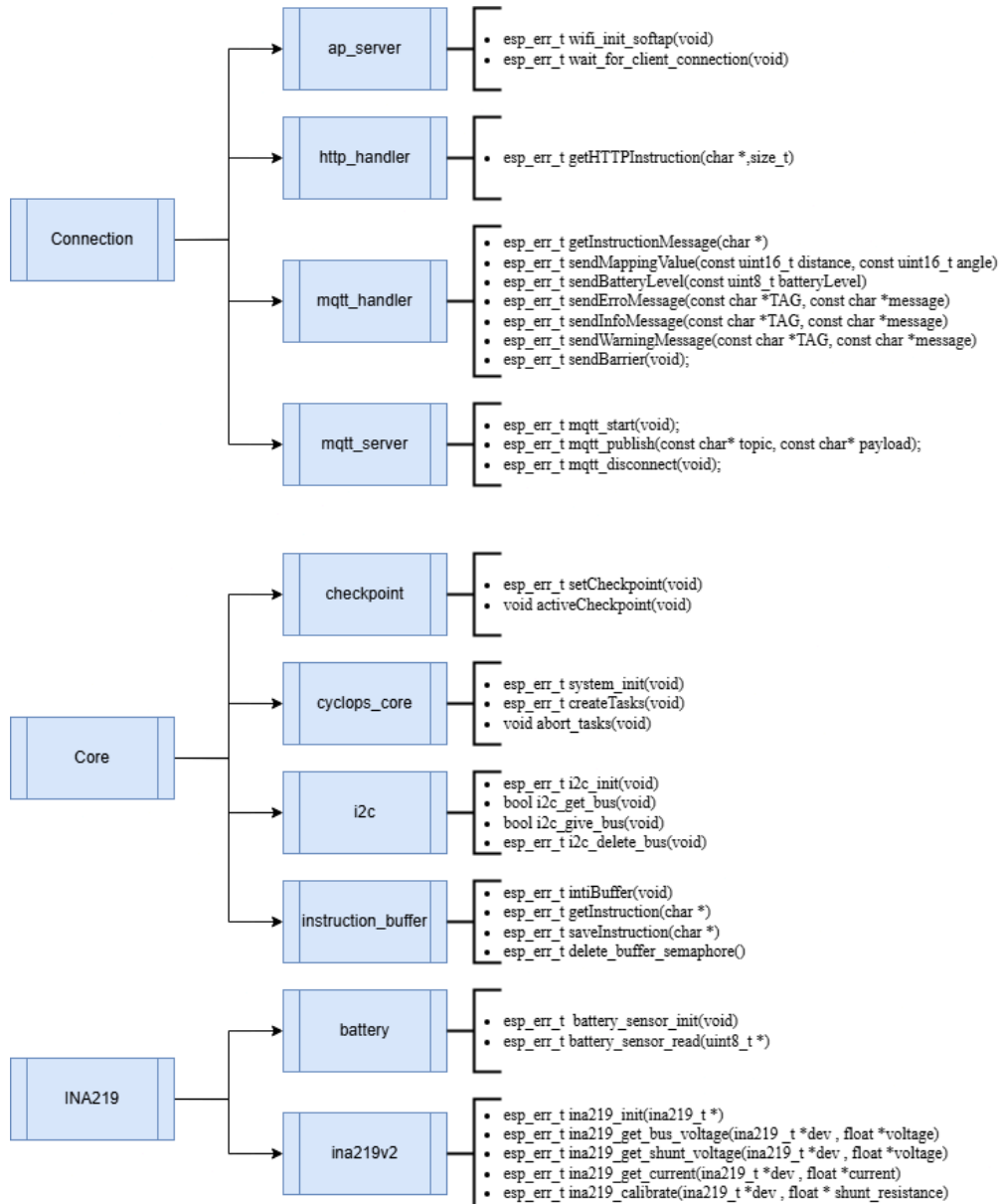
*Figura 1: Esquema Completo del Sistema*

Para visualizar las funciones y procesos que posee el código fuente del robot, se realizó un esquema del sistema que permita rápidamente entender cada una de las funciones en el mismo. (**Figura 1**). Para una mejor vista se adjunta este diagrama en el **Anexo**.

La mayoría de las funciones y procesos fueron desarrolladas por el equipo de trabajo, no obstante dos librerías fueron adaptadas de librerías públicas en github[13]. Dichas librerías son las que permiten comunicarse mediante I2C con los sensores, `i2c_vl53l0x` para el LiDAR, basada en el código de Artful Bytes[29], y de igual manera la `ina219v2` basadas en el código de Ruslan V. Uss. Esto, ya que el manejo de los sensores es altamente complejo (principalmente respecto al LiDAR) debido al manejo de los registros. Razón por la cual, se optó por la utilización de código ya probado y funcional.

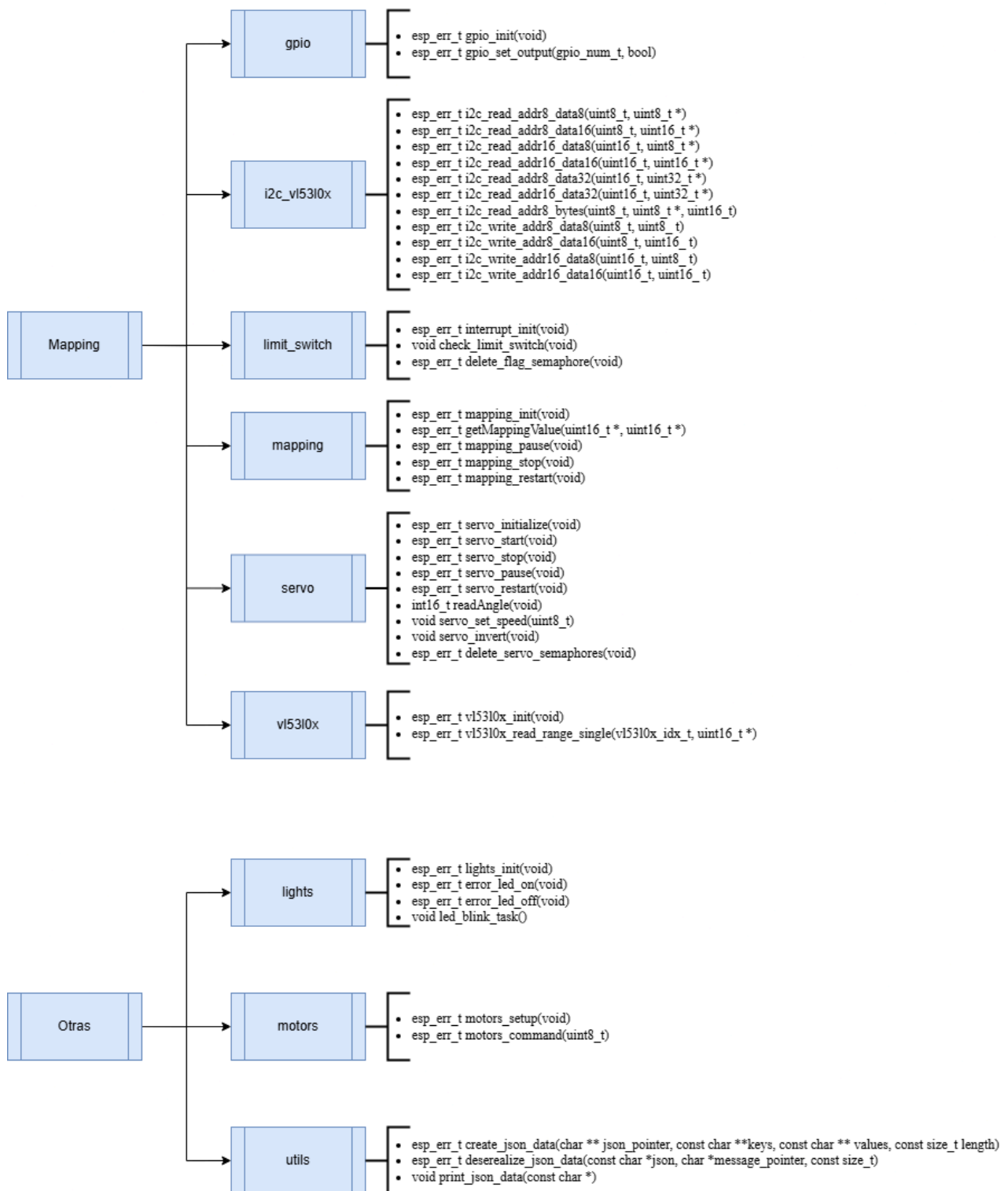


## 2.1.2 Funciones de Cyclops



**Figura 2:** Parte 1 - Funciones del Robot

La Figura 2, representa una parte de las funciones del robot, englobadas en sus librerías y estas en grupos según a la tarea con la que se relacionaban. Por ejemplo, ap\_server, http\_handler, mqtt\_handler y mqtt\_server, comprenden la lógica necesaria para comunicar el ESP32 con el Backend, por tanto se agrupan en la carpeta Connection.

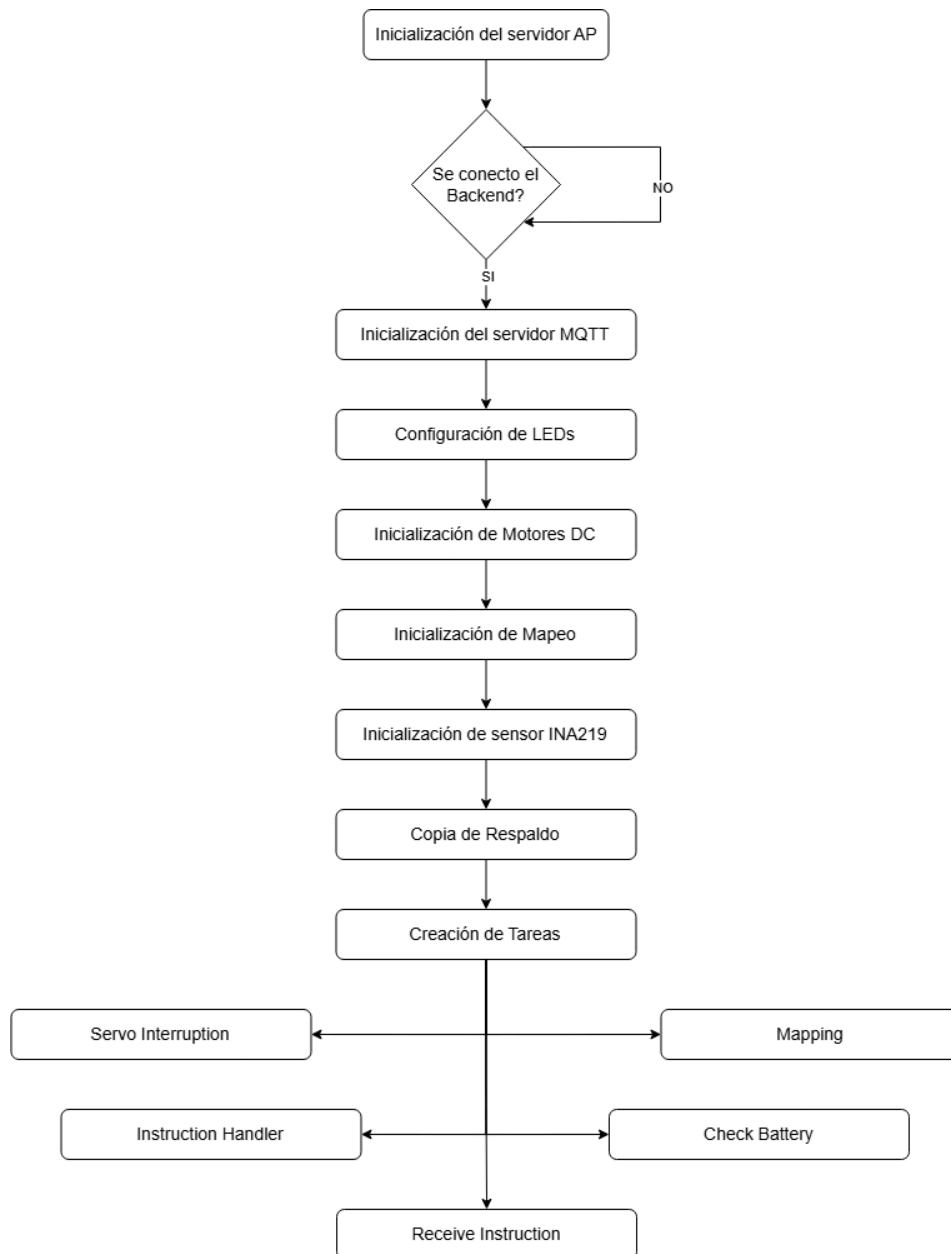


**Figura 3: Parte 2 - Funciones del Robot**

En esta última figura, se ilustran los demás grupos de librerías, en particular Mapping, la cual comprende las librerías necesarias para llevar a cabo el mapeo del entorno. Función que

requirió de una lógica extensiva, que incluye el manejo del servomotor, el manejo del LiDAR junto con su comunicación I2C, y el sensor de carrera antes mencionado.

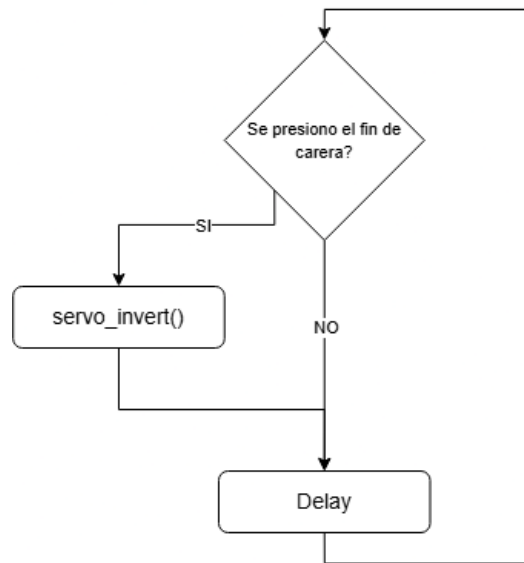
### 2.1.3 Diagrama de Flujo



**Figura 4:** Diagrama de flujo General

Esta figura representa el diagrama de flujo general del programa principal. En específico la cadena de inicialización del sistema, finalizando con la creación de la tareas, que llevarán a cabo las funciones principales del sistema, las cuales se detallan a continuación.

### 2.1.3.1 Servo Interruption

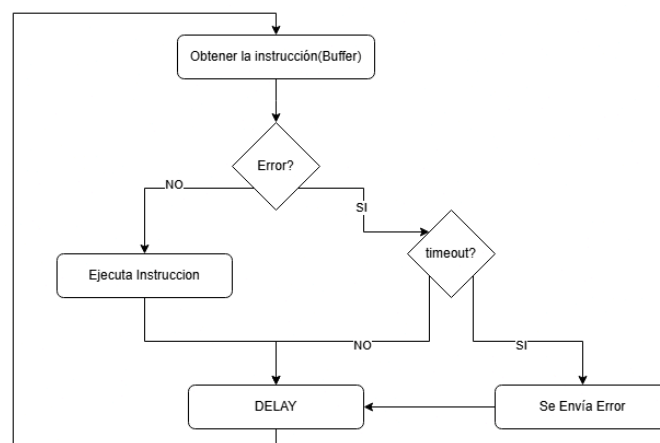


**Figura 5:** Diagrama de flujo de la tarea Servo Interruption

Esta tarea, como se ve en el diagrama se basa en accionar la función `servo_invert()` siempre que se presione el fin de carrera; esto de forma periódica mediante un delay.

La función `servo_invert()`, se encarga de invertir el sentido de giro del servomotor, y establecer el tiempo de referencia, entre otras cosas.

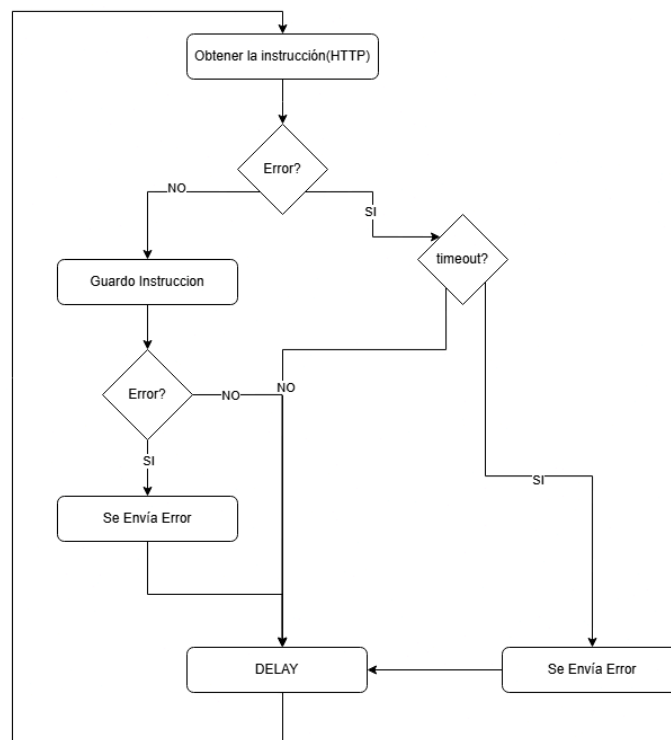
### 2.1.3.2 Instruction Handler



**Figura 6:** Diagrama de flujo de la tarea Instruction Handler

Esta tarea se encarga de obtener la instrucciones a ejecutar del buffer de instrucciones. Si la obtiene, ejecuta la mismas, sino, analiza si el error fue por timeout, lo que indicaría que hubo un problema considerable al intentar obtenerla, por lo que se envía el error al usuario. En cualquier otro caso, se ejecuta un delay, se vuelve a iniciar el bucle.

### 2.1.3.3 Receive Instruction

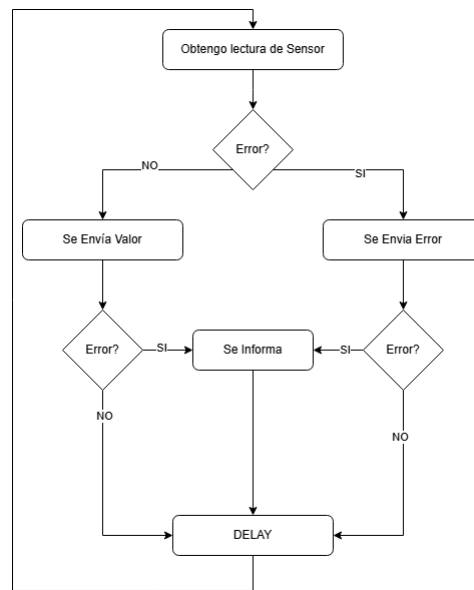


**Figura 7:** Diagrama de flujo de la tarea Receive Instruction

Esta tarea comienza realizando un get HTTP al backend para obtener la siguiente instrucción, y si la consulta resulta exitosa, es decir, se obtuvo una instrucción, se guarda en el buffer de instrucciones. En caso de fallar la función, y devolver un error por timeout, se envía el error al usuario, si no, continúa el bucle ejecutando el delay.

Si al guardar la instrucción ocurre un error, sucede lo mismo, se envía el error al usuario, y se continua el bucle.

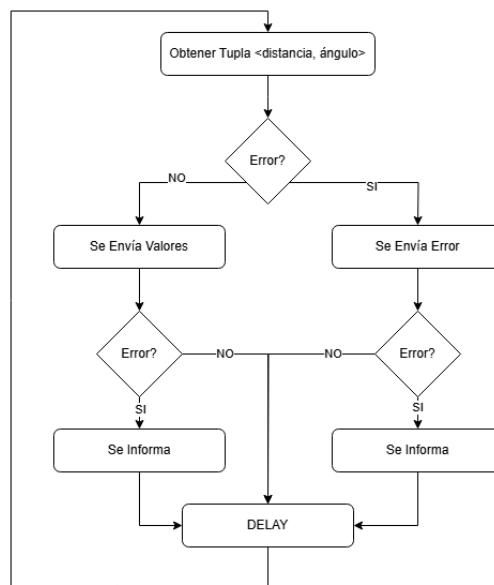
#### 2.1.3.4 Check Battery



**Figura 8:** Diagrama de flujo de la tarea Check Battery

Esta tarea se encarga de obtener periódicamente el valor de carga de la batería y enviarlo al backend. En caso de no poder obtenerlo, se envía el error al usuario. No obstante, tanto el envío del error, como del valor, pueden fallar, en este caso, se informa por conexión serie, pues falla la comunicación.

#### 2.1.3.5 Mapping



**Figura 9:** Diagrama de flujo de la tarea Check Battery

Esta tarea compone la función principal del vehículo, la cual es el mapeo de su entorno. Comienza al tratar de obtener la tupla <distancia, ángulo>, y si se obtiene, se envía el valor; en caso contrario se envía el error. A su vez, si se produce un error en alguno de los envíos, se informa de ello mediante conexión serie. Y posteriormente, indiferentemente del caso, se continúa el bucle.

Como se puede apreciar en los distintos diagramas, no se entró en detalle en cómo se realizan las distintas operaciones, ya que se explicará más detalladamente en secciones posteriores.

Otro punto a destacar es que cuando el envío de errores, o “valores” falla, se informa de esto mediante conexión serie, algo que requiere tener conectado el ESP32 a un PC, lo que no corresponde al funcionamiento normal. Esto se debe a que estos fallos no deberían ocurrir, ya que serían resultado de comportamientos anormales, y evidentes por sí mismos, al no llegar los “valores” al usuario, y en este punto sería necesario conectarlo al pc para ver qué sucede.

## 2.2 Esquemático de Conexiones del Hardware

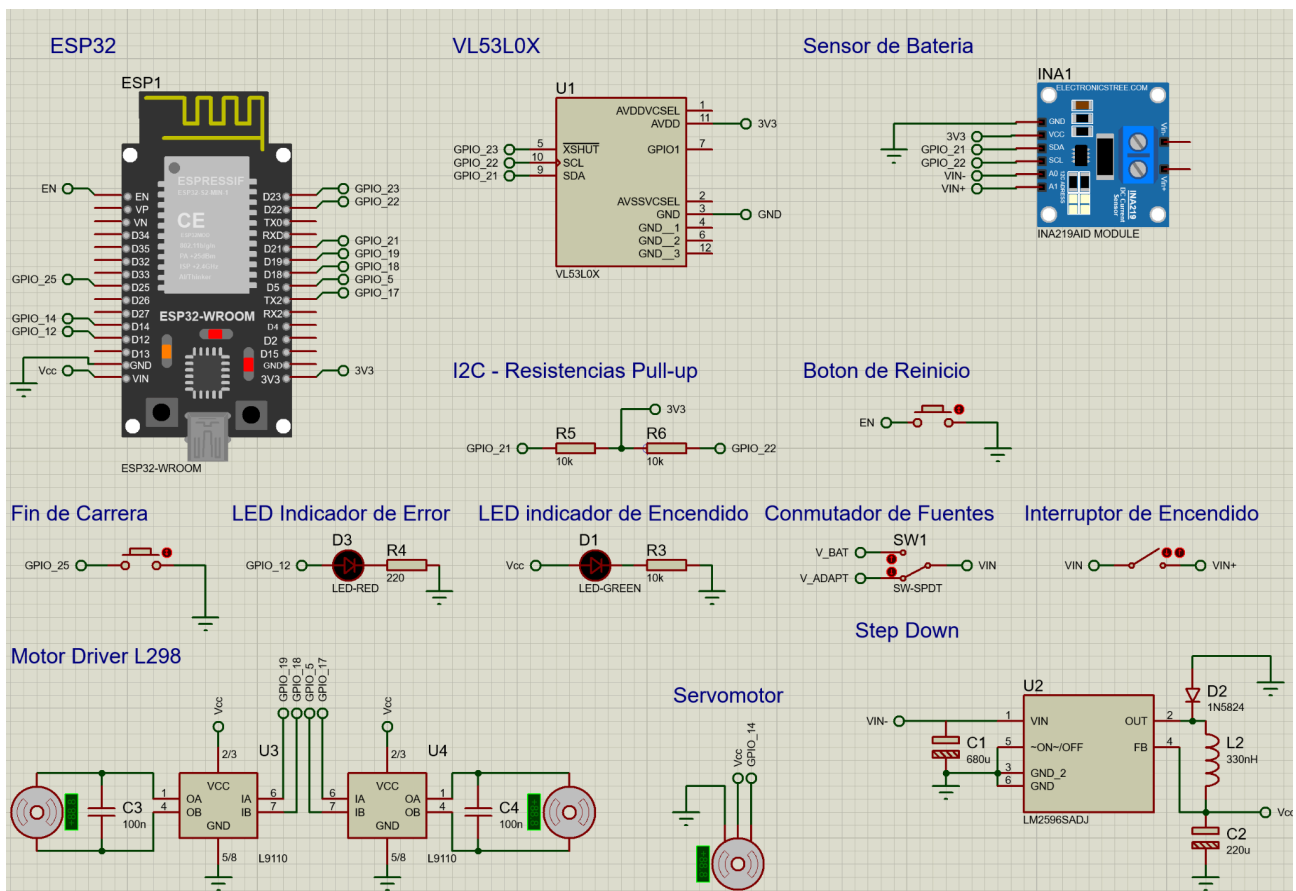


Figura 10: Esquema de Conexiones

La figura anterior es una captura del Esquemático realizado en Proteus[14], en el cual se pueden apreciar las distintas conexiones del circuito del robot. Principalmente el ESP32, el Servomotor, y INA219, siendo los más voluminosos. Aunque también se encuentran el LiDAR VL53L0X del cual no encontró una versión “más visual”, colocando una versión técnica, junto con el Motor Driver (con los 2 motores) y el step down, de los cuales no se encontró integrados. En el caso del driver, se lo representó por los 2 puentes H, mas no es el circuito completo, y en el caso del step down se armó una aproximación del circuito del integrado siguiendo la datasheet.

Por otro lado, también se puede apreciar conexiones adicionales, como las resistencias pull-up para las conexiones I2C de los sensores, los dos leds indicadores, el fin de carrera, y los botones de reinicio, encendido y apagado, junto con el conmutador, el cual nos permite optar entre la fuente de alimentación y las baterías. En este último punto, debemos aclarar que el robot, cuenta con dos fuentes de energía, una fuente de alimentación que convierte la corriente de 220V a 9V provista por la cátedra, y dos pilas 18650[15], que proveen la energía para que el robot opere independientemente a distancia, sin estar conectado a la red eléctrica.

En ambos casos, la tensión protista será bajada a 5V por el step-down, pues es el voltaje máximo utilizado por la mayoría de componentes. En el caso del LiDAR y el INA219 que funcionan con 3.3V, se alimentan del ESP32, el cual funcionará además, como convertidor de 5V a 3.3V.



## 2.3 Estado alcanzado en la funcionalidad

En la siguiente tabla se podrá observar un listado de las funciones propuestas en el comienzo del proyecto indicando el estado alcanzado al momento de finalizar el proyecto.

Funcionalidad	Requerimiento	Estado
<b>Mapeo 2D completo del área</b>	Implementar algoritmo para controlar el sensor LiDAR para realizar un mapeo del área.	<b>TOTAL</b>
	Implementar algoritmo para controlar el servomotor y manejar el movimiento del sensor LiDAR.	
	Comunicación con el backend para almacenar los valores.	
	Implementar algoritmo que calcule la distancia entre el robot y el obstáculo detectado.	
<b>Desplazamiento del vehículo</b>	El robot debe ser capaz de desplazarse hacia adelante, atrás, girar y frenar según el usuario se lo indique.	<b>TOTAL</b>
<b>Comunicación entre el usuario y el robot</b>	El usuario debe ser capaz de enviarle instrucciones al robot.	
	El robot debe ser capaz de ejecutar la instrucción recibida.	
	Diseño e implementación de la página web.	
	Visualización de los datos en la interfaz web.	
	Backend como intermediario.	
<b>Montaje de hardware</b>	Diseño del robot.	<b>TOTAL</b>
	Montaje de los componentes.	
	El MCU debe poder comunicarse correctamente con los otros componentes (motores, servo, sensores).	
	El servomotor debe estar posicionado para poder controlar el movimiento del sensor LiDAR.	
<b>Medición de batería</b>	Incorporar baterías en el diseño.	
	Incorporar el sensor INA219 en el diseño.	

	Implementar algoritmo para la lectura y medición de carga.	
<b>Manejo autónomo del vehículo</b>	Implementar algoritmo para que el vehículo sea capaz de desplazarse sin necesidad de controlarlo.	<b>SIN REALIZAR</b>
	Detectar caminos viables entre los obstáculos.	
<b>Testeo</b>	Realizar una fase de testeos para comprobar el funcionamiento del sistema.	<b>PARCIAL</b>
	Testeo de la interfaz.	
	Testeos de los componentes físicos.	
	Testeo del mapeo	

**Tabla 1:** Estado de las funciones propuestas.

Como se puede observar, hay 3 estados: TOTAL, PARCIAL y SIN REALIZAR. Estos se utilizan para indicar que tareas fueron completadas totalmente, cuales se iniciaron pero no se logró realizar un testeo más exhaustivo o agregar determinadas optimizaciones, y por último qué tareas no se pudieron comenzar por falta de tiempo o que luego de un análisis no resultaba conveniente desarrollarlas.

## 3 Documentación del Software

### 3.1 Pre-requisitos

El proyecto requiere de los siguientes componentes a nivel software para su ejecución:

- VScode (Visual Studio Code) [[16](#)]
- PlatformIO (extension para VScode) [[17](#)]
- Mosquitto MQTT Broker [[18](#)]
- MongoDB [[19](#)]
- Node [[20](#)]

Habiendo instalado los componentes, procedemos a explicar los pasos necesarios para montar el código del ESP32, levantar el server y el frontend.

### 3.1.1 MQTT Broker

El protocolo MQTT exige un broker, es decir, un gestor de la conexión entre los dispositivos. En nuestro caso utilizamos Mosquitto, un broker de Open Source[21] de Fundación Eclipse[22].

0. Instalar Mosquitto MQTT Broker con el instalador descargado.
1. Abrir la carpeta donde se instaló, en nuestro caso “*C:\Program Files\mosquitto*”
2. Configurar el Broker modificando el archivo .conf para admitir conexiones sin credenciales y anónimas, y la salida por consola de eventos, mediante las siguientes líneas:

```
listener 1883
allow_anonymous true
log_type all
connection_messages true
log_timestamp true
```

3. Abrir consola en modo administrador y ubicarse en la carpeta de instalación del Broker
4. Ejecutar el broker mediante el comando `mosquitto -v -c mosquitto.conf`

De este modo el broker quedará configurado y ejecutándose, además se podrá visualizar el flujo de mensajes en la consola.

### 3.1.2 ESP32

Para desarrollar el código del ESP32 se utilizó el framework del fabricante Expressif, IDF [23] en la versión 5.2.2 en lenguaje C, y como plataforma VScode con la extension PlatformIO version 3.16.

0. Abrir VScode, clonar el repositorio, y posicionarse en la carpeta del mismo: 2024-A2-LIDAR-VL53L0X
1. Posicionarse sobre la carpeta Microcontroller donde se encuentra el código. En la carpeta src se halla el main.c y lib, las distintas librerías.
2. En la consola ejecutar ***cd Microcontroller***, para poder ejecutar comandos de PlatformIO.
3. Abrir platformio.ini, y modificar upload\_port con el número de puerto asignado en tu computadora.

4. Ejecutar `pio run -t upload`, para cargar el código al ESP32.

**NOTA:** no se debe estar conectado por monitor serie al ESP32 al momento de ejecutar.

De este modo se logra cargar el código al ESP32.

Debe mencionarse que se puede evitar ejecutar los comandos, utilizando la interfaz provista por PlatformIO.

Comandos extra:

- `pio run`: compila el código.
- `pio run -t clean`: limpia el proyecto.
- `pio run -t erase`: limpia el ESP32.

### 3.1.3 Backend

Para el backend se utilizó lenguaje Java con el JDK [\[24\]](#) versión 23, y los frameworks Maven [\[25\]](#) (versión 4.0) y Spring Boot [\[26\]](#) (versión 3.0).

0. Abrir VScode, clonar el repositorio, y posicionarse en la carpeta del mismo: 2024-A2-LIDAR-VL53L0X
1. Posicionarse sobre la carpeta Backend donde se encuentra el código.
2. En la consola ejecutar `cd backend`, para poder ejecutar comandos en el código.
3. Ejecutar `mvn clean install` para instalar todas las dependencias
4. Ejecutar `mvn spring-boot:run` para levantar el servidor
5. El servidor queda a la espera de conectarse a la red del ESP32.

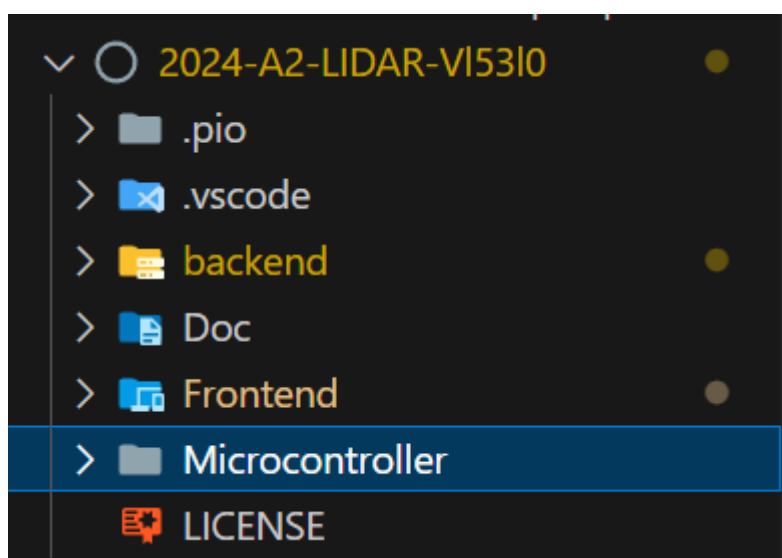
Respecto a la base de datos, esta es una no relacional, en nuestro caso MongoDB. Para utilizarla se debe ejecutar el instalador descargado, sin necesidad de alguna configuración adicional, solo habilitar su ejecución en segundo plano. El backend se conectará de forma automática

### 3.1.4 Frontend

En el caso del Frontend habiendo instalado Node.js, ya no es necesario hacer ninguna otra modificación.

0. Abrir VScode, clonar el repositorio, y posicionarse en la carpeta del mismo: 2024-A2-LIDAR-VL53L0X
1. Posicionarse sobre la carpeta Frontend donde se encuentra el código.
2. En la consola ejecutar `cd Frontend`, para poder ejecutar comandos en el código.
3. Ejecutar `npm install`, para instalar las dependencias.
4. Ejecutar `ng version` para verificar la correcta instalación de Angular [27].
5. Ejecutar `ng server`.

Con esto hecho, la página ya es accesible en el link `localhost:4200`.



*Figura 11: Carpeta del Proyecto*

Como se mencionó anteriormente, se debe clonar el repositorio, al hacerlo, este se ve como en la imagen.

### 3.1.4 Hardware

Respecto al hardware, además de la PC para ejecutar el backend, base de datos y frontend, y del propio robot, solo se debe contar con un mando tipo consola, para poder controlar el vehículo, pues como se explicará más adelante, la interfaz web, cuenta solo con funciones básicas de control.

## 3.2 Librerías del ESP32

En la carpeta “Microcontroller” se encuentran todas las bibliotecas necesarias desarrolladas para interactuar y manejar el microcontrolador. Estas bibliotecas están compuestas por funciones para configurar y controlar diferentes módulos, conexión WiFi, manejo de sensores y motores, además de las herramientas necesarias para la comunicación. Es decir, se compone de aquellas bibliotecas fundamentales para la programación del hardware. Estas son las siguientes:

- **INA219:** contiene aquellas bibliotecas destinadas a la configuración y manejo del sensor INA219, utilizado para la medición de carga de las baterías, además de la biblioteca correspondiente para realizar la lectura de la carga.
- **Mapping:** compuesta por las bibliotecas necesarias para realizar el mapeo, es decir, aquellas para la configuración del sensor, servo y la lógica para realizar el mapeo correspondiente.
- **Connection:** contiene las bibliotecas necesarias para gestionar las conexiones de red y la comunicación entre el microcontrolador y el backend.
- **Core:** incluye las bibliotecas necesarias para la comunicación, manejo de tareas concurrentes, almacenamiento de instrucciones y para realizar una copia del estado del sistema en un momento determinado.
- **Lights:** para el manejo de LED, indicador de errores.
- **Motors:** contiene las bibliotecas que permiten el control de los motores del vehículo.
- **Utils:** contiene una biblioteca que se encarga del manejo de string en formato JSON[[28](#)], los cuales se utilizan para la comunicación entre dispositivos.

Además del programa principal para el funcionamiento completo del sistema.

### 3.2.1 INA219

Contiene las bibliotecas necesarias para gestionar el componente INA219, el cual es utilizado para monitorear la carga de batería del sistema. Está compuesto por dos bibliotecas, las cuales se explicaran a continuación.

#### 3.2.1.1 battery.h

Está compuesta por las funciones necesarias para inicializar y leer el nivel de carga del sensor de batería. Se definieron el mínimo y máximo valor de carga (en V) como constantes:

- **FULL\_CHARGE:** 8.4V
- **MIN\_CHARGE:** 6.4V

Las funciones incluidas son:

- **battery\_sensor\_init():** Realiza la inicialización, configuración y calibración del sensor INA219 para comenzar con la medición de la batería.
- **battery\_sensor\_read(uint8\_t \*):** Lee el estado de la batería y, luego de realizar las verificaciones necesarias y calcular el porcentaje según el voltaje leído, almacena el valor en la dirección indicada por el puntero recibido como parámetro.

### 3.2.1.2 ina219v2.h

Esta biblioteca contiene las funciones necesarias para inicializar, configurar, obtener mediciones de voltaje, además de permitir las calibraciones del sensor INA219.

Se definió una única estructura denominada **ina219\_t** que se utiliza para almacenar la dirección I2C del dispositivo. Además de esto, se definieron las siguientes constantes:

- **INA219\_ADDRESS:** Dirección I2C por defecto del INA219, se puede modificar dependiendo del hardware. En este caso es la dirección 0x40.

Las siguientes se utilizan para almacenar las direcciones de los registros del INA219:

- **INA219\_REG\_CONFIG:** 0x00
- **INA219\_REG\_SHUNT\_VOLTAGE:** 0x01
- **INA219\_REG\_BUS\_VOLTAGE:** 0x02
- **INA219\_REG\_POWER:** 0x03
- **INA219\_REG\_CURRENT:** 0x04
- **INA219\_REG\_CALIBRATION:** 0x05

Por último, para la configuración del INA219:

- **INA219\_CONFIG\_DEFAULT:** Se establece en 0x299F, lo que significa modo normal, rango de 16V, y ADC 12 bits.

Las funciones incluidas en la librería son las siguientes:

- **ina219\_init(ina219\_t \*dev):** Se utiliza para la inicialización del INA219.
- **ina219\_get\_bus\_voltage(ina219\_t \*dev, float \*voltage):** Obtiene el voltaje del bus en V. Esta es la que utilizaremos en nuestro caso.
- **ina219\_get\_shunt\_voltage(ina219\_t \*dev, float \*voltage):** Obtiene el voltaje del shunt en mV.
- **ina219\_get\_current(ina219\_t \*dev, float \*current):** Se utiliza para obtener la corriente en mA.

- **ina219\_calibrate(ina219\_t \*dev, float shunt\_resistance):** Para la configurar la calibración.

Debemos aclarar que esta biblioteca, como se adelantó, es una adaptación de la biblioteca hecha por Ruslan V. Uss, y publicadas en Github.

### 3.2.2 Mapping

Dentro de esta sección, se encuentran todas las bibliotecas que se encargan y facilitan el mapeo. Una parte fundamental es el manejo del sensor, para esto se utilizó una librería desarrollada por Artful Bytes[29] en GitHub. La misma está compuesta de 3 sub-librerías, encargadas del manejo de los elementos necesarios para llevar a cabo la tarea de medición. Estas son:

- Librería de control de los pines GPIO del ESP32
- Librería de control del sensor VL53L0X
- Librería de control de la comunicación I2C requerida por el sensor

Cada una de estas se explicará a continuación.

#### 3.2.2.1 mapping.h

Teniendo en cuenta el esquema físico del robot, se desarrolló esta librería que permite obtener el ángulo en el que el sensor esté apuntando, y la distancia a la que se encuentra del obstáculo.

El motivo por el cual se implementó fue para facilitar la obtención de los valores necesarios para generar el mapa. El sensor VL53L0X puede medir distancias, pero solo el servo sabe en qué dirección está apuntando. Por lo tanto, esta librería se encarga de tanto realizar las mediciones con el sensor, como de consultarle al servo el ángulo de giro en el que se encuentra.

Esta librería introduce las siguientes funciones:

- **mapping\_init():** realiza la inicialización necesaria para poder utilizar el servomotor y el sensor LiDAR. Esta función utiliza las inicializaciones previamente creadas (gpio\_init(), i2c\_init(), vl53l0x\_init(), servo\_initialize() y servo\_start())
- **getMappingValue(\*angle, \*distance):** obtiene el ángulo actual del servo y la distancia medida por el sensor, y los guarda en las variables angle y distance respectivamente. En caso de que el sensor falle en tomar una medición, intentará reiniciarlo.
- **mapping\_pause():** esta función detiene el servomotor, y apaga el LiDAR, de tal manera que no gasten energía.



- **mapping\_restart():** se utiliza para reiniciar el servomotor.

### 3.2.2.2 vl53l0x.h

Esta librería incluye las funciones y definiciones necesarias para la inicialización, configuración y control del sensor VL53L0X. Como fue mencionado previamente, se utilizó una librería ya existente.

Se definió la siguiente constante:

- **VL53L0X\_OUT\_OF\_RANGE (500)**

También se definió la siguiente estructura:

- **vl53l0x\_idx\_t:** para especificar el sensor, en este caso es VL53L0X\_IDX\_FIRST.

En lo que respecta al control del sensor VL53L0X, la librería está compuesta por múltiples funciones, pero las más importantes son funciones:

- **vl53l0x\_init():** realiza la inicialización del sensor VL53L0X, seteando su dirección I2C por default, y su configuración.
- **vl53l0x\_read\_range\_single(idx, \*range):** realiza una medición del sensor idx (al solo usar uno, este parámetro es obsoleto) y guarda el resultado en la variable range.
- **vl53l0x\_reset():** reinicia el sensor en caso de que exista algún fallo.

### 3.2.2.3 i2c\_vl53l0x.h

La librería encargada de las comunicaciones I2C del sensor LiDAR posee múltiples funciones encargadas de, por ejemplo, escribir o leer registros de varios tamaños diferentes. Se definió la siguiente constante para almacenar la dirección por default del slave (VL53L0X):

- **DEFAULT\_SLAVE\_ADDRESS (0x29)**

Se definieron las siguientes estructuras:

- **addr\_size\_t:** para almacenar los diferentes tamaños de direcciones posibles, estas son ADDR\_SIZE\_8BIT o ADDR\_SIZE\_16BIT.
- **reg\_size\_t:** para almacenar los diferentes tamaños de registros posibles, estos son REG\_SIZE\_8BIT, REG\_SIZE\_16BIT y REG\_SIZE\_32BIT.

Además, está compuesta por las siguientes funciones:

- Funciones encargadas de leer de registros para diferentes tamaños de datos y de direcciones:
  - **i2c\_read\_addr8\_data8(uint8\_t, uint8\_t \*)**
  - **i2c\_read\_addr8\_data16(uint8\_t, uint16\_t \*)**

- `i2c_read_addr16_data8(uint16_t, uint8_t *)`
- `i2c_read_addr16_data16(uint16_t, uint16_t *)`
- `i2c_read_addr8_data32(uint16_t, uint32_t *)`
- `i2c_read_addr16_data32(uint16_t, uint32_t *)`
- Función encargada de leer una cantidad determinada de bytes de una dirección:
  - `i2c_read_addr8_bytes(uint8_t, uint8_t *, uint16_t)`
- Funciones encargadas de escribir en registros para diferentes tamaños de datos y de direcciones:
  - `i2c_write_addr8_data8(uint8_t, uint8_t)`
  - `i2c_write_addr8_data16(uint8_t, uint16_t)`
  - `i2c_write_addr16_data8(uint16_t, uint8_t)`
  - `i2c_write_addr16_data16(uint16_t, uint16_t)`
- Función encargada de escribir una cantidad determinada de bytes en una dirección:
  - `i2c_write_addr8_bytes(uint8_t, uint8_t *, uint16_t)`

### 3.2.2.4 gpio.h

La librería que maneja los pines GPIO se encarga de controlar el pin XSHUT del sensor. Esto le permite controlar el apagado y encendido del sensor, de forma tal que este pueda ser reiniciado en caso de suceder algún inconveniente, y además permite la modificación de la dirección I2C de uno o varios sensores al mismo tiempo. De esta manera, es posible tener múltiples sensores VL53L0X en simultáneo, cada uno con su propia dirección I2C.

Se define la siguiente constante:

- **GPIO\_XSHUT\_FIRST**: indica el número de pin para XSHUT, en este caso se utiliza el `GPIO_NUM_23`.

Además, se define también la siguiente estructura:

- **gpio\_t**: almacena los tipos de GPIO posibles, estos son `GPIO_XSHUT_FIRST`, `GPIO_XSHUT_SECOND` y `GPIO_XSHUT_THIRD`.

La librería está compuesta por las siguientes dos funciones:

- **gpio\_init()**: para inicializar el GPIO, configurandolo como salida y estableciendo un nivel bajo en el pin.
- **gpio\_set\_output(gpio\_num\_t gpio, bool enable)**: para establecer el gpio en un nivel alto o bajo dependiendo del parámetro enable.

### 3.2.2.5 servo.h

El sensor LiDAR dispone de una biblioteca particular para su manejo, sin embargo, por encima de esta, se encuentra otra biblioteca la cual abstrae el funcionamiento del sensor, junto con el del Servomotor. La razón de esto, es por cómo funcionará el mapeo o sondeo del entorno. El sensor rota aproximadamente 360° mediante el servomotor, por lo que ambos deben operar en conjunto, debiéndose sincronizar el sondeo con la rotación.

Cabe aclarar que realmente el sensor no rota unos 360 grados pues, se enredara el cable y causando daños tanto al sensor, como al resto del robot. Para evitar esto, la pieza en la que el sensor está montado cuenta con unas aspas que accionan al fin de carrera al llegar a ciertos ángulos de giro, los cuales invierten el sentido de giro.

Siendo más específicos, el funcionamiento del servomotor es el siguiente: una vez configurado, el mismo espera a la orden de start (cuando se haya inicializado el LiDAR). A partir de ese instante, estará siempre rotando, salvo que se lo pause. Su dirección de giro se invertirá cada vez que una de las aspas del soporte del LiDAR accione el fin de carrera, pero manteniendo su velocidad (salvo que se indique lo contrario).

El accionar el fin de carrera no solo indica el cambio de sentido, sino que es el punto de referencia para el cálculo de la posición del servo (y por tanto del LiDAR). Ya que el servo gira continuamente, sin un momento de referencia sería imposible calcular su posición, por lo que cada vez que invierte su sentido de giro, también se re-establece una variable con dicho tiempo, la cual se utiliza para calcular el ángulo desde ese momento hasta que se invierta el sentido. Es decir, cada vez que se quiera obtener el ángulo como parte del cálculo se hará una resta entre dicho instante y el tiempo base, que, sabiendo la velocidad de giro y otros parámetros, nos permite calcular la posición.

$$\text{Angulo} = (\text{BaseSpeed} * (\text{duty} - \text{STOP}) * (\text{TimeNow} - \text{TimeBase})) / (\text{DIFF} * \text{CF})$$

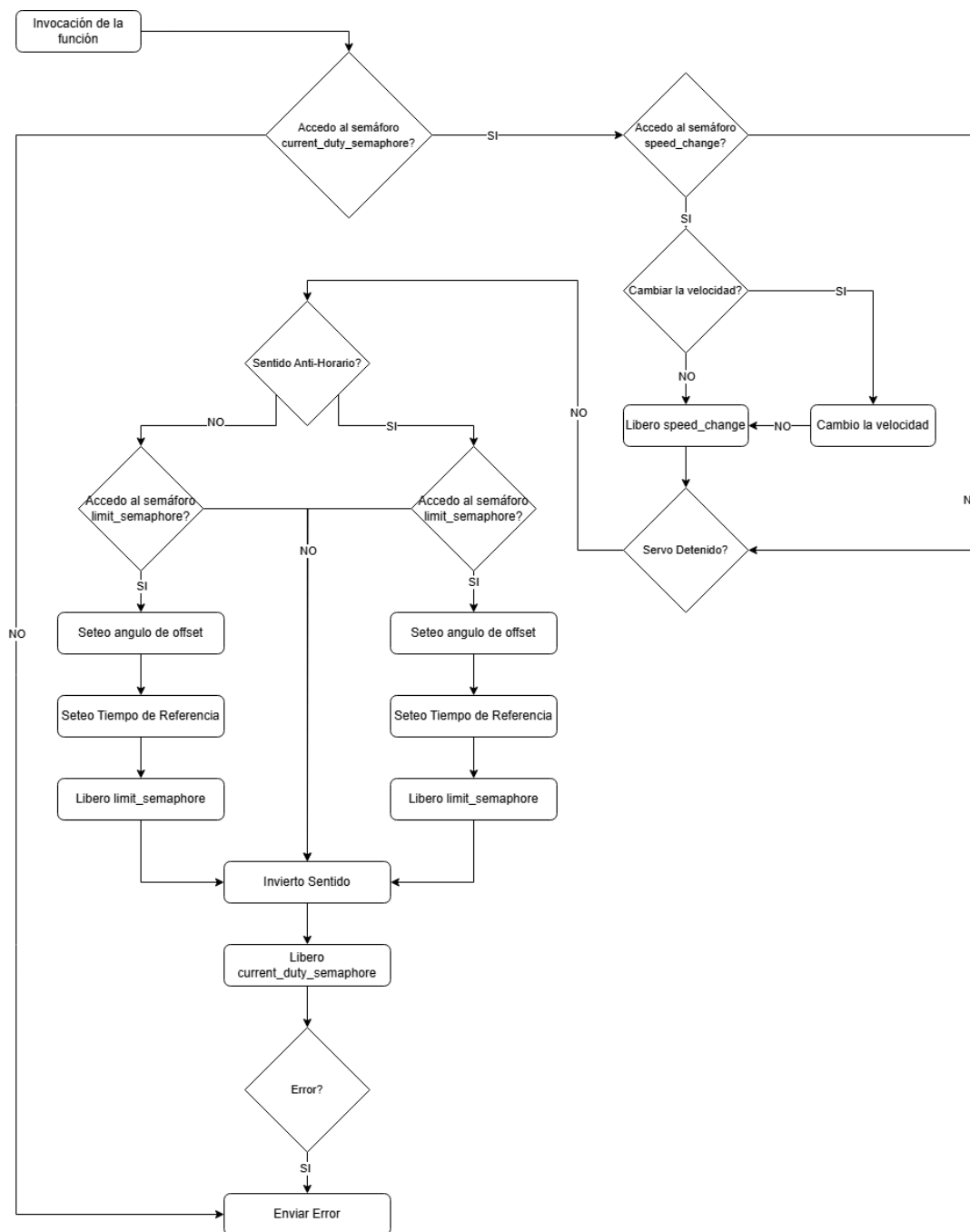
- BaseSpeed[s]: velocidad máxima giro del servo.
- Duty: Ciclo de trabajo de la PWM.
- STOP[us]: Ciclo de trabajo que indica al Servo detenerse
- TimeNow[us]: tiempo actual al llamar a la función (referencia al encendido del MCU)
- TimeBase[us]: tiempo referencia establecido al accionar el fin de carrera
- DIFF(Diferential)[us]: Diferencia entre STOP y la velocidad máxima.
- CF(Conversion Factor): Factor de conversión para pasar de us a s.

A partir de esta fórmula, se puede obtener los ángulos, no obstante como se dijo anteriormente, no se completa el giro, por lo que a esta fórmula se suma un ángulo Offset correspondiente al sentido por el que se presionó el final de carrera.

En cuanto al funcionamiento de todo esto, es considerablemente complejo, para esto la librería está compuesta por:

- Constantes:
  - **SERVO\_MIN\_PULSEWIDTH\_US:** Ancho de pulso mínimo (giro rápido en un sentido), se definió como 900.
  - **SERVO\_MAX\_PULSEWIDTH\_US:** Ancho de pulso máximo (giro rápido en el sentido opuesto), se definió como 2100.
  - **SERVO\_STOP\_PULSEWIDTH\_US:** Ancho de pulso para detener el servo, es 1500.
  - **SERVO\_PULSE\_GPIO:** Se establece con el pin GPIO\_NUM\_14.
  - **SERVO\_TIMEBASE\_RESOLUTION\_HZ 1000000:** 1MHz, 1us per tick.
  - **SERVO\_TIMEBASE\_PERIOD 20000:** 20000 ticks, 20 ms.
  - Además de otras constantes que sirven para el cálculo de los ángulos y definición de las velocidades baja, media y alta (teniendo como referencia el ancho de pulso mínimo y máximo previamente mencionados).
- Estructura:
  - **SERVO\_DIRECTION:** compuesto de los valores UP y DOWN, facilita indicar el aumento o decremento de la velocidad..
- Funciones:
  - **servo\_inialize():** esta función inicializa la PWM para controlar el servomotor, lo que implica la inicializar de un Timer así como, de una serie de estructuras que permiten operar el mismo, entre ellas, la inicialización de la interrupción accionada por el fin de carrera,.
  - **servo\_start():** inicializa el servo a velocidad media.
  - **servo\_stop():** detiene el servo.
  - **servo\_pause():** detiene el servo, sin alterar el cálculo del ángulo.
  - **servo\_restart():** reanuda el giro del servo (la contraparte de pause).
  - **readAngle():** retorna el ángulo al que apunta el servo.

- **servo\_set\_speed(SERVO\_DIRECTION):** permite aumentar o disminuir la velocidad de giro. Los cambios se reflejan al invertir, para evitar alterar el cálculo de ángulo.
- **servo\_invert():** la función principal, permite invertir el giro, establecer el tiempo base o referencia, el ángulo de offset, y de ser indicado cambiar la velocidad.
- **delete\_servo\_semaphores():** elimina todos los semáforos creados, esto se utiliza en caso de ejecutar ABORT.



**Figura 12:** Diagrama de Flujo de la función `servo_invert()`

### 3.2.2.6 limit\_switch.h

Como se mencionó anteriormente, una parte clave del funcionamiento del servomotor es el fin de carrera, el cual permite realizar la inversión del giro, el cálculo del ángulo y demás. Para esto, el mismo se implementó como una interrupción por hardware, de tal manera que al accionarse cierre el circuito conectado a GND el pin correspondiente, activando la interrupción. Y esta cambia el valor de un variable que sirve a modo de flag indicando que fue presionado. Además se considera el efecto rebote, por lo que se ignora cualquier interrupción hasta no haber pasado tiempo determinado desde la anterior (50ms).

Esta librería proporciona las funciones necesarias para controlar y gestionar el interruptor de límite, es decir, el fin de carrera. Para almacenar el número de pin del fin de carrera, se definió la siguiente constante:

- **LIMIT\_SWITCH\_PIN GPIO\_NUM\_25**

También, contiene las siguientes funciones:

- **interrupt\_init():** Esta función inicializa la configuración de la interrupción para el interruptor de límite, en nuestro caso el fin de carrera. Se encarga de configurar el pin para que genere interrupciones cuando pase de un nivel alto a bajo.
- **check\_limit\_switch():** Esta función verifica el estado del interruptor de límite (limit switch). En el caso de haber sido activo, indica que el servo alcanzó la posición límite (fin de carrera).
- **delete\_flag\_semaphore():** Elimina el semáforo utilizado para modificar el flag que indica si el fin de carrera fue presionado o no, se utiliza al ejecutar ABORT.

### 3.2.3 Connection

Esta carpeta se utiliza para implementar las bibliotecas necesarias para el módulo de comunicación entre el microcontrolador ESP32 y el servidor, el cual es fundamental para el intercambio de datos. Utilizando el MCU configurado como un Access Point, se establece una red local donde otros dispositivos pueden conectarse. Esta configuración permite la transmisión de mensajes y comandos, como las instrucciones para controlar el vehículo o alertas de error.

En cuanto al protocolo de envío de mensajes, se optó por utilizar MQTT, y para la recepción de instrucciones HTTP, por las razones antes mencionadas..

### 3.2.3.1 ap\_server.h

En principio, se creó el Access Point adaptando un programa existente brindado por ESP-IDF. En este programa se configuraron los parámetros de la red, como el nombre (SSID), la contraseña, el canal de comunicación y la cantidad máxima de conexiones simultáneas permitidas. Estos son parámetros fundamentales para que el ESP32 actúe como un punto de acceso, permitiendo que otros dispositivos se conecten para intercambiar información, como instrucciones de control del vehículo o mensajes de error, facilitando la comunicación dentro del sistema.

Se definen las siguientes constantes:

- **ESP\_WIFI\_SSID:** Nombre de la red creada, en este caso es: "Cyclops".
- **ESP\_WIFI\_PASS:** Contraseña de la red, la cual es "A2TdP2Lidar".
- **ESP\_WIFI\_CHANNEL:** Canal utilizado, se definió el canal 1.
- **MAX\_STA\_CONN:** Máxima conexión a la red, es 4.
- **MAX\_NUM\_CONNECTIONS:** Máximo número de conexiones, es 1.

Se divide principalmente en tres funciones:

- **wifi\_init\_softap():** Esta función configura el ESP32 para que actúe como un Access Point. Inicializa la red WiFi y establece los parámetros de la red, como el SSID, la contraseña y el canal. Además, registra los manejadores de eventos necesarios para la conexión de clientes y la asignación de IP. Por último, inicia el WiFi, permitiendo que otros dispositivos se conecten a la red del ESP32 y envíen o reciban mensajes.
- **initialize\_server():** Inicializa la memoria flash NVS (almacenamiento no volátil) y configura el ESP32 como un Access Point mediante la función `wifi_init_softap()`, explicada previamente. Si ocurre un error al inicializar la memoria flash, la función la borra y vuelve a intentarlo.
- **wait\_for\_client\_connection():** Esta función se bloquea hasta que un dispositivo se conecte al Access Point. Utiliza un semáforo para esperar la conexión, lo que asegura que no continúe la ejecución hasta que se haya establecido una conexión.

### 3.2.3.2 mqtt\_server.h

La biblioteca `mqtt_server.h`, encargada de llevar a cabo la comunicación entre el ESP32 y el Servidor, consta de 3 funciones principales:

- **mqtt\_start():** función que inicializa el servidor MQTT. Lo que implica configurar el ESP32 y conectarse al Broker MQTT.

- **mqtt\_disconnect():** esta función se encarga de desconectar el servidor MQTT.
- **mqtt\_publish():** esta función permite enviar un mensaje recibido por parámetro, en un tópico indicado por parámetro.

### 3.2.3.3 mqtt\_handler.h

Esta biblioteca se diseñó para facilitar el envío y recepción de información mediante MQTT, y para ello consta de 6 funciones:

- **getInstruccionMessage():** devuelve por referencia una instrucción recibida. Dicha instrucción es obtenida del instruction buffer antes mencionado.
- **sendMappingValue():** esta instrucción recibe la distancia y ángulo, del LiDAR y Servo, y los envía mediante el tópico Mapping.
- **sendBatteryLevel():** esta función recibe el nivel de carga en la batería obtenido mediante el INA219, y lo envía mediante el tópico Battery.
- **sendErrorMessage():** esta función recibe un Tag(etiqueta de origen), y un mensaje, y los envía mediante el tópico Messages, marcado como un mensaje de error.
- **sendWarningMessage():** idem al anterior, pero marca el mensaje como una advertencia.
- **sendInfoMessage():** idem a los dos anteriores, pero marca el mensaje como información.

Un punto clave de estas funciones, y del manejo con MQTT, es que no se envía (ni recibe) texto plano, sino que utilizamos el formato JSON. Por lo que cada función crea un JSON acorde al dato a enviar. Y en caso de getInstruccionMessage() recibe y deserializa el JSON para obtener la instrucción. Cabe aclarar que todas las funciones devuelve si tuvieron éxito o no con su tarea, ya que son tipo esp\_err\_t.

### 3.2.3.4 http\_handler.h

Esta librería contiene una única función necesaria para realizar una petición GET al backend para obtener la siguiente instrucción no leída y luego almacenarlo en un buffer para posteriormente ejecutar. Primero se define un tamaño máximo de mensaje con la siguiente constante:

- **INST\_MAX\_SIZE 20**

Esta función es:

- **getHTTPInstruction(char \*, size\_t):** realiza la petición GET al backend para luego con un manejador de eventos, obtener la instrucción y decodificarla, ya que está en formato JSON.



### 3.2.4 Core

Esta carpeta contiene la librerías centrales del funcionamiento del sistema, es decir, aquellas con funciones claves del sistema, entre ellas las librerías con la funcion de inicializacion del sistema, y las tareas que ejecutan las tareas claves, la librería con buffer de instrucciones, la de gestión del bus I2C, y la que nos permite abortar la ejecución del sistema.

#### 3.2.4.1 checkpoint.h

Esta biblioteca permite ejecutar la acción ABORT, mediante una serie de funciones que permiten crear un “punto de guardado” y ejecutar dicho punto. En concreto, permite crear una copia del estado del sistema, y volver a él, abortando la ejecución actual del sistema. Consta de 2 funciones:

- **setCheckpoint():** para realizar la creación del punto de guardado.
- **activeCheckpoint():** para realizar los pasos necesarios para el ABORT (liberar recursos utilizados y abortar tareas) para luego activar el punto de guardado.

#### 3.2.4.2 cyclops\_core.h

Esta biblioteca contiene funciones que permiten facilitar la inicialización del sistema y el manejo de las tareas concurrentes del ESP32, utilizando FreeRTOS[\[30\]](#) para crearlas. Estas funciones son las siguientes:

- **system\_init():** se utiliza para la inicialización del sistema, iniciando motores, sensores, servidor AP y MQTT. Además de esperar la conexión del backend al mismo.
- **createTasks():** Esta función se encarga de crear las tareas necesarias para el funcionamiento del sistema.
- **abort\_tasks():** Se encarga de finalizar las tareas en ejecución, en este caso la encargada de manejar las interrupciones del servo, liberando los recursos necesarios para el reinicio del sistema.

#### 3.2.4.3 i2c.h

Para la comunicación entre el MCU y los sensores tanto el LiDAR como el INA219, como se explicó en el informe anterior se implementó una biblioteca para manejar el bus I2C. No obstante

tras implementar las bibliotecas para conectar ambos sensores, se terminó modificando esta biblioteca reduciendola a las tareas esenciales de configurar la conexión I2C, y arbitrar el acceso al bus. La razón tras esta decisión fue, la complejidad en la forma de operar y utilizar el bus por parte de ambos sensores, lo que devino en crear librerías aparte para cada uno, y en limitar esta a las funcionalidades anteriormente mencionadas.

Se definieron las siguientes constantes:

- **I2C\_MASTER\_SCL\_IO GPIO\_NUM\_22**
- **I2C\_MASTER\_SDA\_IO GPIO\_NUM\_21**
- **I2C\_MASTER\_NUM I2C\_NUM\_0**
- **I2C\_MASTER\_FREQ\_HZ: 400000**
- **I2C\_MASTER\_TX\_BUF\_DISABLE: 0**
- **I2C\_MASTER\_RX\_BUF\_DISABLE: 0**
- **I2C\_MASTER\_TIMEOUT\_MS:** tiempo de timeout de I2C en ms, en este caso se le asignó 1000 ms.

Con respecto a las funciones, contiene las siguientes:

- **i2c\_init():** esta función inicializa al ESP32 como I2C en modo Master, configurando los pines SDA y SCL en modo pull-up, y la frecuencia en 400KHz correspondiente a Fast I2C. Además inicializa un semáforo para bloquear el acceso simultáneo al bus..
- **i2c\_get\_bus():** esta función chequea si el bus i2c está libre, y de estarlo lo toma, bloqueandolo para cualquier otro dispositivo.
- **i2c\_give\_bus():** esta función es la contraparte de la anterior. Si el dispositivo terminó de operar sobre el bus, lo libera para su uso.
- **i2c\_delete\_bus():** esta función se utiliza para eliminar el semáforo utilizado para el bus.

#### 3.2.4.4 instruction\_buffer.h

Debido a la diferencia de velocidad con la que puede recibir y atender instrucciones el MCU, se creó esta biblioteca, la cual permite almacenar las instrucciones entrantes en un buffer circular, para que la tarea encargada de ejecutarlas las tenga cuando sea necesario. De esta manera, no se pierden instrucciones.

Para determinar el tamaño del buffer, se definieron las siguientes constantes:

- **INSTRUCTIONS\_BUFFER\_SIZE:** Cantidad de instrucciones que se pueden almacenar en el buffer, se determinó 10 instrucciones como máximo.

- **INSTRUCTION\_MAX\_LENGTH:** Máximo largo de cada instrucción, en este caso es 40.

Además de esto, se incluyeron las siguientes funciones:

- **initBuffer():** realiza la creación del semáforo necesario para gestionar el acceso al buffer, devuelve error en caso de fallar.
- **getInstruction(char \*):** devuelve la siguiente instrucción del buffer y actualiza los punteros necesarios. En caso de no haber nuevas instrucciones, devuelve error de timeout.
- **saveInstruction(char \*):** almacena una nueva instrucción en el buffer y actualiza los punteros necesarios. En el caso de no haber espacio en el buffer, retorna error.
- **delete\_buffer\_semaphore():** elimina el semáforo previamente creado, se utiliza en el caso de ejecutar un ABORT.

### 3.2.5 Lights

#### 3.2.5.1 lights.h

La carpeta lights contiene una única librería, la cual se encarga de gestionar el control de las luces del sistema, en este caso un único LED de error.

Esta librería está compuesta por las siguientes funciones para inicializar y controlar el estado del LED:

- **lights\_init():** Realiza la inicialización y configuración de los pines utilizados para los leds.
- **error\_led\_on():** Establece el pin del led en un nivel alto para encenderlo e indicar que un error ocurrió.
- **error\_led\_off():** Establece el pin del led en un nivel bajo para apagarlo.
- **led\_blink\_task(time\_ms):** crea una tarea que hace titilar el led, en base al tiempo de entrada. Si la tarea ya se encuentra creada, la elimina, deteniendo el titileo y apagando el led.

### 3.2.6 Motors

#### 3.2.6.1 motors.h

Para el manejo de los motores, se desarrolló una librería en el framework ESP-IDF que, en conjunto del driver L9110s [31], permite el desplazamiento del robot. Esta realiza diferentes movimientos como avanzar, retroceder, rotar a la derecha, y rotar a la izquierda.

Se definieron las siguientes constantes para almacenar el número de pin de control para los motores:

- **MOTOR1\_PIN1:** pin de control para Motor 1 (dirección 1), se le asignó el GPIO\_NUM\_19.
- **MOTOR1\_PIN2:** pin de control para Motor 1 (dirección 2), se le asignó el GPIO\_NUM\_18.
- **MOTOR2\_PIN1:** pin de control para Motor 2 (dirección 1), se le asignó el GPIO\_NUM\_5.
- **MOTOR2\_PIN2:** pin de control para Motor 2 (dirección 2), se le asignó el GPIO\_NUM\_17.

Además, las siguientes estructuras:

- **MOTOR\_STRUCT:** se utiliza para almacenar el estado del pin 1 y 2 de cada motor.
- **DIRECTIONS:** se almacenan las distintas direcciones posibles, estas son las siguientes FORWARD, BACKWARD, ROTATE\_RIGHT, ROTATE\_LEFT y STOP.

La misma presenta también las siguientes funciones:

- **motors\_setup():** inicializa los pines que controlan los motores y se asegura que los motores estén apagados poniendo los pines en bajo.
- **motors\_command(direction):** controla la dirección de giro de los motores, permitiendo que estos desplacen al robot hacia adelante, hacia atrás, que rote con sentido de giro hacia la derecha y hacia la izquierda.

### 3.2.7 Utils

Esta carpeta incluye la biblioteca json\_helper.h, la cual fue desarrollada para facilitar el manejo de mensajes en formato JSON. Es fundamental para manejar y facilitar la comunicación entre los distintos componentes del sistema.

#### 3.2.7.1 json\_helper.h

Esta biblioteca está compuesta por las funciones necesarias para crear, procesar y visualizar objetos de tipo JSON. Aquellas son:

- **create\_json\_data(char \*\*, const char \*\*, const char \*\*, const size\_t):** Realiza la creación de un string de formato JSON, a partir de pares clave-valor.
- **deserealize\_json\_data(const char \*,char \*, const size\_t):** Realiza el parseo de un string en formato JSON y extrae los valores asociados a determinadas claves o keys.

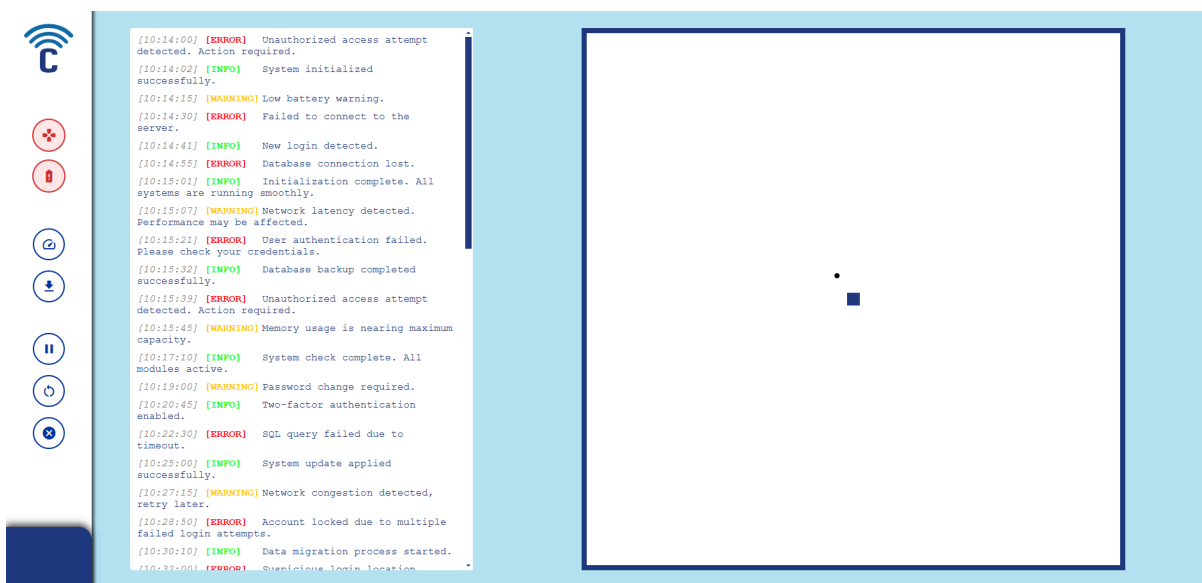
- **print\_json\_data(const char \*):** Imprime el string de formato JSON recibido por argumento, para la depuración del sistema, en el caso de ser NULL, imprime un mensaje de error.

## 4 Documentación Relacionada

### 4.1 Interfaz Web

Para el desarrollo del frontend del proyecto, se diseñó una página web utilizando Angular, con la cual se pueda controlar el vehículo, así cómo también visualizar el mapa generado con los obstáculos detectados por el sensor y los mensajes de control del vehículo. Permite facilitar al usuario la supervisión y control del sistema. Sus funcionalidades principales son:

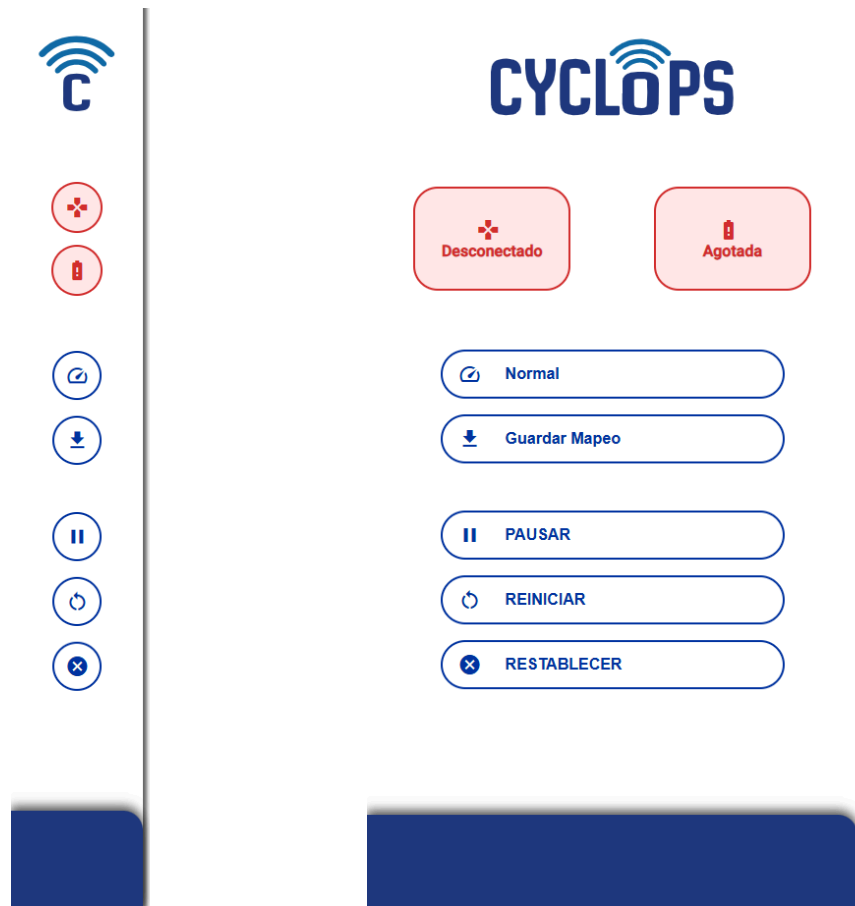
- **Monitoreo de datos:** cuenta con un monitor donde se pueden visualizar cualquier tipo de mensajes, ya sean de error o informativos.
- **Visualización de mapa:** Se presenta un gráfico en 2D, donde se representan los puntos detectados por el sensor LiDAR.
- **Sidebar:** Incluye botones interactivos que permiten enviar instrucciones al vehículo, como ajustar su velocidad o pausar el mapeo.



*Figura 13 : Interfaz web desarrollada.*

En la figura anterior se pueden observar todos sus componentes principales incluyendo: sidebar, monitor y mapa.

Asimismo, el sidebar es un componente interactivo, por lo que puede expandirse o contraerse al hacer click sobre este. Este comportamiento se puede observar en la siguiente figura.



(a) Sidebar contraída

(b) Sidebar expandida

**Figura 14 :** Componente sidebar de la interfaz.

## 4.2 Hardware



*Imagen 2: Cyclops*



*Imagen 3: Cyclops Vista Lateral*



***Imagen 4:** Cyclops Vista Trasera*

En esta última imagen podemos observar una serie de leds y botones importantes para el control del vehículo.

Los Leds como se explicó anteriormente, tienen dos funciones, el led verde(derecha) se enciende siempre que haya energía, indicando que el vehículo está encendido. Y el rojo, indica algún problema u estado especial.

Respecto a los botones, de izquierda a derecha tenemos el conmutador, el cual nos permite seleccionar entre la batería o la fuente de alimentación, como alimentación. Siguiendo, tenemos en el centro el botón de reinicio, que reinicia el ESP32; y por último el de ON/OFF para prender y apagar el robot.



## 4.3 Manual de Usuario

Para manejar el robot se debe utilizar la interfaz web, mediante la cual contaremos con una serie de controles básicos, y un mando que mediante la misma, nos permitirá controlar el robot.

### 4.3.1 Interfaz Web

La interfaz web cuenta con tres componentes principales, los cuales son el monitor donde se visualizan los mensajes que envía el robot, el mapa donde se encuentran los obstáculos detectados por el LiDAR; los cuales se pueden apreciar en la *figura 13*. Y por otro lado el sidenav (*figura 14*), donde se encuentran los botones para controlar el robot, así como el indicador de mando conectado, y el de carga de la batería.

Dichos botones son:





- Velocidad: el primer botón permite incrementar y decrementar la velocidad de giro del servo, y por tanto del proceso de mapeo.
- Guardar Mapeo: genera una captura local en la pc del estado actual del mapa.
- Pausar/Reanudar: pausa/reanuda el proceso de mapeo.
- Reiniciar: reinicia el robot.
- Restablecer: aborta el flujo actual de ejecución, es decir, aborta todas las tareas, y vuelve al estado previo a la creación de las mismas.

Debemos mencionar que los botones de Velocidad y Restablecer se encuentran inhabilitados, pues no se terminaron de implementar dichas funcionalidades.

### 4.3.2 Mando

Para controlar el robot, y en específico, manejarlo se optó por el uso de un mando, pues era mucho más sencillo, e intuitivo.

Botón	Acción
-------	--------

START	Reiniciar el Robot
SELECT	Abortar Ejecución
Triángulo	Aumentar Velocidad de Mapeo
X	Reducir Velocidad de Mapeo
Joystick Derecho 	Avanzar
Joystick Derecho 	Retroceder
Joystick Derecho 	Girar Derecha
Joystick Derecho 	Girar Izquierda
R2	Frenar el Auto

**Tabla 2:** *Controles del Robot*

La tabla anterior ilustra las funciones de los botones utilizados del mando para operar el robot. Como se puede observar no todos los botones se utilizan, por lo que a futuro se pueden asignar más funciones a estos.

Un detalle a mencionar es que el robot frena siempre que se suelte el Joystick Derecho, no obstante también se puede utilizar el R2.

## 4.4 Enlaces

En esta sección adjuntamos los links, para acceder al repositorio en github del código fuente, así como a la bitácora del proyecto y el video de demostración:

- [Enlace a Github](#)
- [Enlace a la Bitácora](#)
- [Video Demostración del Proyecto](#)

## 5 Conclusión

Después del primer informe de avance, se logró integrar todos los componentes planteados de forma exitosa, permitiendo llevar a cabo el funcionamiento general del sistema. Mediante el uso del microcontrolador ESP32, se diseñó un vehículo robot capaz de desplazarse y comunicarse con otros dispositivos. En simultáneo a esto, el mismo es capaz de estimar la presencia de distintos obstáculos a su alrededor. No obstante, considerarlo un mapeo propiamente dicho sería quizás una exageración, puesto que las limitaciones tecnológicas del sistema nos impiden realizarlo con una precisión adecuada para su comprensión.

Fue posible desplegar una página web en donde el usuario puede interactuar con el sistema, y enviar instrucciones al robot, permitiendo tanto el desplazamiento del mismo, como ir hacia adelante o para atrás, y girar tanto a la izquierda como a la derecha, así como el reinicio del microcontrolador, y el pausado y reanudado del sistema de mapeo. Cabe destacar, que para la correcta integración del sistema, fue requerida la implementación de un backend encargado de gestionar las comunicaciones con la página web y el microcontrolador.

En cuanto a los objetivos planteados inicialmente, a pesar de los parciales resultados obtenidos a la hora de realizar un mapeo como tal, fue posible alcanzar el resto de ellos, teniendo un sistema funcional que cumple con lo requerido.

### 5.1 Mejoras

Existen diversos aspectos que pueden ser mejorados en un futuro para la optimización del sistema. Entre estos se encuentran:

- Optimizar los tiempos de carga de los datos provenientes del backend al frontend.
- Optimizar el proceso por el cual se realiza el gráfico.
- Incorporar al mapa generado, el desplazamiento del vehículo, considerando los movimientos que realiza.
- Implementar la funcionalidad del cambio de velocidad de giro del servomotor.
- Mejorar el procesamiento de datos generados por el conjunto servomotor - sensor LiDAR, de forma tal que se pueda obtener un mapeo de mayor precisión y viabilidad.
- Respecto al sensor LiDAR, sería recomendable utilizar la librería oficial de control del mismo, ya que la misma permite realizar diferentes tipos de medición y calibración.

- Realizar las modificaciones necesarias para poder recibir instrucciones entre el ESP32 y el backend mediante el protocolo MQTT, y no por HTTP.
- Materializar el circuito realizado en un PCB.

Si se considera posible mejorar y/o reemplazar partes del sistema en sí mismo, recomendamos combinar el servomotor y lidiar con un giroscopio y/o encoder, para obtener los ángulos con precisión, o si fuera posible un motor con encoder, para mayor precisión y simplicidad del circuito.

## 6 Bibliografía

1. LiDAR. URL: <https://www.ibm.com/mx-es/topics/lidar>
2. ESP32. URL: <https://www.espressif.com/en/products/socs/esp32>
3. Linorobot by Rud Merriam. Hackaday. URL:  
<https://hackaday.com/2016/03/13/petite-package-provides-powerful-robot/>
4. ROS. URL: <https://www.ros.org/>
5. VL53L0X. URL:  
<https://www.st.com/en/imaging-and-photonics-solutions/vl53l0x.html>
6. INA219. URL: <https://www.ti.com/product/es-mx/INA219>
7. HTTP. URL: <https://kinsta.com/es/base-de-conocimiento/que-es-una-peticion-http/>
8. IOT. URL: [https://es.m.wikipedia.org/wiki/Internet\\_de\\_las\\_cosas](https://es.m.wikipedia.org/wiki/Internet_de_las_cosas)
9. MQTT. URL: <https://mqtt.org/>
10. SG90. URL: <https://www.alldatasheet.es/datasheet-pdf/pdf/1572383/ETC/SG90.html>
11. I2C. URL: <https://es.m.wikipedia.org/wiki/I%C2%B2C>
12. PCB. URL: [https://es.m.wikipedia.org/wiki/Circuito\\_impreso](https://es.m.wikipedia.org/wiki/Circuito_impreso)
13. Github. URL: <https://github.com/>
14. Proteus. URL: <https://www.labcenter.com/>
15. Pilas 18650. URL:  
<https://ferretronica.com/products/bateria-recargable-18650-3-7v-400-mah>
16. VScode. URL: <https://code.visualstudio.com/>
17. PlatformIO. URL: <https://platformio.org/>
18. Mosquitto. URL: <https://mosquitto.org/>
19. MongoDB. URL: <https://www.mongodb.com/es>
20. Node. URL: <https://nodejs.org/es>
21. Open Source. URL: [https://es.m.wikipedia.org/wiki/Sistema\\_de\\_c%C3%B3digo\\_abierto](https://es.m.wikipedia.org/wiki/Sistema_de_c%C3%B3digo_abierto)
22. Fundación Eclipse. URL: <https://www.eclipse.org/>
23. ESP-IDF. URL: <https://idf.espressif.com/>
24. JDK. URL: <https://www.oracle.com/ar/java/technologies/downloads/>
25. Maven. URL: <https://maven.apache.org/>
26. Spring-Boot. URL: <https://docs.spring.io/spring-boot/index.html>
27. Angular. URL: <https://angular.dev/overview>
28. JSON. URL: <https://es.wikipedia.org/wiki/JSON>

29. LiDAR Library. URL: [https://github.com/artfulbytes/vl6180x\\_vl53l0x\\_msp430](https://github.com/artfulbytes/vl6180x_vl53l0x_msp430)
30. FreeRTOS. URL: <https://www.freertos.org/>
31. Driver L9110s. URL: <https://naylampmechatronics.com/drivers/66-driver-puente-h-l9110s.html>

## 7 Apéndice A: Materiales y Presupuesto

A continuación se detalla una lista de los materiales utilizados en el robot, incluyendo las cantidades y precio de cada uno.

COMPONENTE	CANTIDAD	PRECIO	NOTA
ESP32	1	\$13.400,00	Dado por La Cátedra
LiDAR VL53L0X	1	\$18.000,00	Dado por La Cátedra
Motor DC	1	\$3.700,00	Dado por La Cátedra
Rueda Motor	2	\$2.200,00	Dado por La Cátedra
Rueda Loca	1	\$4.700,00	Dado por La Cátedra
Fuente (Integrado)	1	\$5.570,00	Dado por La Cátedra
Fuente (Cargador)	1	\$8.000,00	Dado por La Cátedra
Servomotor	1	\$10.000,00	Dado por La Cátedra
Dual Driver Motores	1	\$3.000,00	Dado por La Cátedra
INA219	1	\$10.928,00	-
Pila 18650	2	\$3.860,00	-
Step Down	1	\$3.300,00	-
Bornera	2	\$600,00	-
Placa Perforada	1	\$3.450,00	-
Fin de Carrera	1	\$700,00	-
Resistencias	4	\$300,00	-
Pines Hembra y Macho x 40	3	\$4.000,00	Cantidad estimativa
Cobertor Pines Macho x 40	2	\$2.800,00	-
Tornillos M3	15	\$115,00	-
Tornillos M2	2	\$600,00	-
Impresión 3D	1	\$50.500,00	-
<b>Total</b>		<b>\$170.293,00</b>	

*Tabla 3: Presupuesto del Proyecto*

Los materiales listados, se componen de aquellos dados por la cátedra, y los adquiridos por nuestra cuenta de modo de completar el proyecto. Entre estos últimos, se destacan las piezas impresas que conforman el robot, y la INA219 para medir las pilas, las cuales también fueron puestas por nuestra cuenta.

Cabe destacar que además de los componentes ilustrados, se adquirieron otros, que por distintas razones, no están en el proyecto final, razón por la cual no están presentes en el listado. Un ejemplo de esto, es el PCB que se descartó, por el error cometido al diseñar el circuito.