Universidad de Murcia

Facultad de Informática

GRADO EN INGENIERÍA INFORMÁTICA

4.^{er} CURSO

SISTEMAS EMPOTRADOS Y DE TIEMPO REAL

Implementación de una plataforma robótica

CURSO ACADÉMICO 2025-2026

Oleksandra Ruzhytska Y9512377M

María García Miñarro 23837656L

Profesor: Antonio Flores

Fecha: 11/01/2026

Índice general

Capítulo 1

Conexiones eléctricas de la Plataforma

1.0.1. Hardware de Control

Para el desarrollo de este proyecto, utilizaremos como unidad central de procesamiento la placa de desarrollo **BeagleBone AI Rev A1**.

Toda la nomenclatura de pines utilizada en este documento (referencias a los cabezales **P8** y **P9**) corresponde al mapa de pines oficial de esta placa. Es fundamental verificar la revisión de la placa (Rev A1) para asegurar que la correspondencia de los GPIOs y PWMs sea exacta a la descrita a continuación.

1.0.2. Instalación de los Encoders (Tacómetros)

Tipos de conexión

Para nuestra plataforma utilizada, usaremos los encoders de 3 pines, se trata de la conexión estándar (VCC, GND, Señal).

Tambien es posible usar encoders de 4 pines que incluyen una salida analógica (AO) y una digital (DO). Para este montaje, **sin conectar la salida analógica (AO)**, utilizando únicamente la salida digital.

Mapeo de pines y configuración GPIO

A continuación, se detalla la configuración de los pines asociados a los encoders: El pin configurado en el sistema se calcula de la siguente forma: gpio 5.17 = gpio177 ($32 \times 5 + 17 = 177$)

Pin Físico	Configuración del Sistema (GPIO)
P9.25 ***Encoder izquierdo***	`gpio 5.17 = gpio177`
P9.18 ***Encoder derecho***	`gpio 6.16 = gpio208`

Cuadro 1.1: Configuración de pines para los encoders.

1.1. Instalación y Conexión del Puente H

Control de giro (GPIOs)

Se han configurado 4 pines como GPIOs para controlar el sentido de giro de los dos motores. La asignación es la siguiente:

Pin Físico	Configuración del Sistema (GPIO)
P8.07 ***Pin A (Motor Izquierdo)***	`gpio 5.05 = gpio165`
P8.08 ***Pin B (Motor Izquierdo)***	`gpio 5.06 = gpio166`
P8.09 ***Pin A (Motor Derecho)***	`gpio 5.18 = gpio178`
P8.10 ***Pin B (Motor Derecho)***	`gpio 5.04 = gpio164`

Cuadro 1.2: Pines GPIO para el control de dirección de motores.

Lógica de funcionamiento A/B:

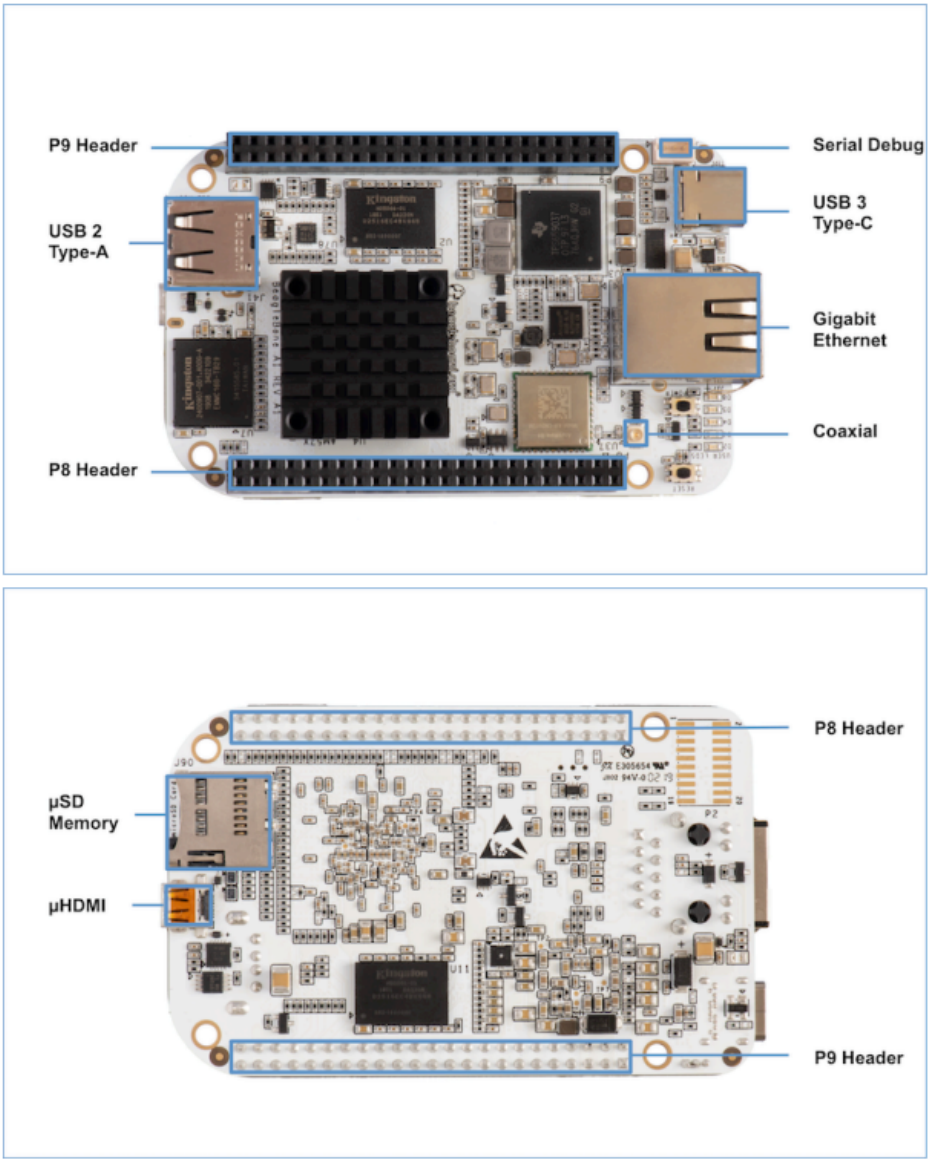

Figura 1.1: Localizaciones de los componentes de la placa

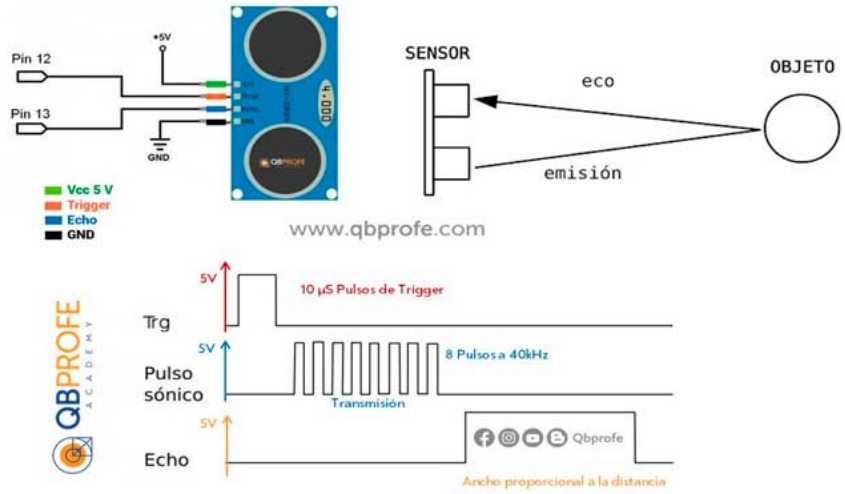

Figura 1.2: Funcionamiento de sensor ultrasonidos

Para determinar el comportamiento del motor, se deben alterar los estados lógicos (Alto/Bajo) de los pines A y B de cada motor de forma simultánea. La siguiente tabla describe la lógica de control:

Estado Pin A	Estado Pin B	Comportamiento del Motor
HIGH (1)	LOW (0)	Giro en Sentido Horario (Adelante)
LOW (0)	HIGH (1)	Giro en Sentido Antihorario (Atrás)
LOW (0)	LOW (0)	Parada libre (Inercia)
HIGH (1)	HIGH (1)	Freno (Parada brusca)

Cuadro 1.3: Tabla de verdad para el control de giro del Puente H.

Nota: La dirección física real (Adelante/Atrás) dependerá de la polaridad del cableado de los motores.

Control de velocidad (PWM)

Para gestionar la velocidad de los motores, se utilizan 2 pines configurados como salidas PWM (Modulación por Ancho de Pulsos):

- **P9.14** ***(Motor Derecho)*****:** Asociado al directorio `pwm-2.0`.
- **P9.16** ***(Motor Izquierdo)***: Asociado al directorio `pwm-2:1`.

1.2. Montaje y Configuración del Sensor de Ultrasonidos

Configuración del sensor HC-SR04

El sensor[1] utiliza dos señales principales, Trigger (disparo) y Echo (recepción), configuradas de la siguiente manera:

Pin Físico	Función	Configuración (GPIO)
P8.36	Trigger (OUT)	`gpio 7.10 = gpio234`
P9.17	Echo (IN)	`gpio 6.17 = gpio209`

Cuadro 1.4: Conexión del sensor HC-SR04.

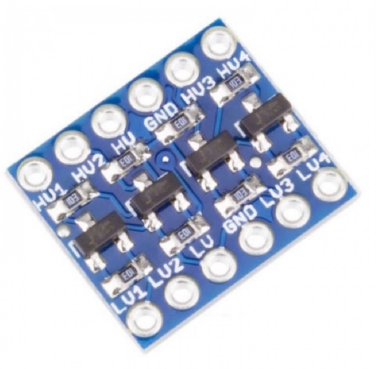

Figura 1.3: Circuito conversor de niveles lógicos usado entre sensor y placa

Adaptación de niveles lógicos

El sensor HC-SR04 opera con una lógica de subsubsection **5V** subsubsection, mientras que la placa controladora trabaja a subsubsection **3.3V** subsubsection. Esto requiere una adaptación de niveles:

1. **Alimentación:** El sensor debe alimentarse obligatoriamente con 5V.
2. **Señal Trigger (Salida de la placa):** Puede conectarse directamente, ya que el voltaje de 3.3V de la placa es suficiente para ser interpretado como un "1 lógico"por el sensor.
3. **Señal Echo (Entrada a la placa):** Tenemos cuidado ya que el sensor envía una señal de retorno de 5V lo que podría dañar el pin de la placa (3.3V).

Solución: Se debe utilizar un circuito conversor de niveles lógicos (Level Shifter) o realizar un divisor de tensión resistivo en la línea de *Echo* para reducir el voltaje de 5V a 3.3V antes de conectarlo a P9.17.

Capítulo 2

Código del programa

El sistema de control del robot se ha diseñado siguiendo una arquitectura modular basada en capas. Este enfoque desacopla el hardware de bajo nivel de la lógica de control y la interfaz de usuario, facilitando el mantenimiento y la escalabilidad. El sistema está implementado en Python 3 y se ejecuta sobre un entorno Linux embebido (BeagleBone), interactuando directamente con el sistema de archivos `SysFS` para el control de periféricos.

La estructura del software se muestra en la figura 2.1

2.0.1. Capa de Orquestación (Raíz `src/`)

`main.py`**:** Es el cerebro del sistema. Inicializa los componentes, lanza los hilos de ejecución paralelos (Lógica del Robot, Servidor Web y Consola) y gestiona el ciclo de vida del programa.

`web_server.py`**:** Gestiona la interfaz de usuario remota mediante WebSockets (Flask + SocketIO), permitiendo el control manual y la visualización de datos del sensor en tiempo real.

2.0.2. Capa de Abstracción de Hardware (HAL `src/hal/`)

Aísla la lógica del programa de los detalles del sistema operativo Linux[2]. La decisión de implementar una capa de abstracción de hardware propia *gpio_sys.py* y *pwm_sys.py* que interactúan directamente con el sistema de archivos SysFS es porque no se ha podido utilizar la librería estándar $Adafruit\_BBIO$ debido a la incompatibilidad con el Kernel y Sistema Operativo Actual, nos daba muchos problemas.

- **Gestión de GPIO (`gpio_sys.py`):** Esta clase maneja los pines de entrada y salida digital. Realiza el mapeo entre los nombres físicos de la placa (ej. `P9_14`) y los números internos del procesador (ej. GPIO 178). Se encarga de exportar los pines en `/sys/class/gpio`, configurar la dirección (`in`/`out`) y gestionar permisos de escritura.

```
import os
import time

class GpioSysfs:
```

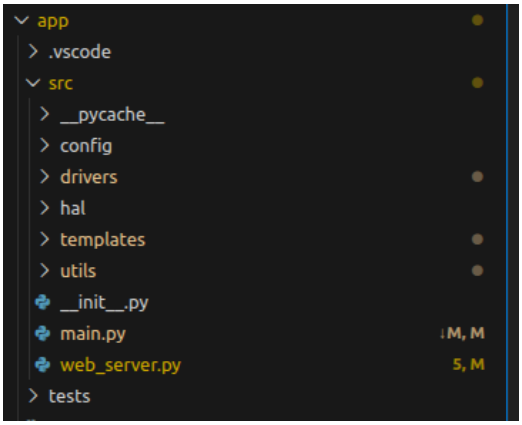

Figura 2.1: Estructura del proyecto

```
    def __init__(self, pin_name, direction="out"):

        self.mapping = {
            "P9_25": "177",
            "P9_18": "208",
            "P8_07": "165",
            "P8_08": "166",
            "P8_09": "178",
            "P8_10": "164",
            "P8_36": "234",
            "P9_17": "209"
        }

        if pin_name not in self.mapping:
            raise ValueError(f"Pin␣{pin_name}␣no␣mapeado␣en␣la␣clase␣GpioSysfs")

        self.gpio_num = self.mapping[pin_name]
        self.path = f"/sys/class/gpio/gpio{self.gpio_num}"

        if not os.path.exists(self.path):
            try:
                with open("/sys/class/gpio/export", "w") as f:
                    f.write(self.gpio_num)
            except OSError:
                print(f"Aviso:␣El␣pin␣{pin_name}␣ya␣estaba␣exportado␣o␣ocupado."
                    )

        time.sleep(0.1)

        try:
            with open(f"{self.path}/direction", "w") as f:
                f.write(direction)
        except PermissionError:
             print("ERROR␣DE␣PERMISOS:␣Recuerda␣usar␣sudo␣o␣--privileged␣en␣
                 Docker")

        print(f"{pin_name}␣configurada␣correctamente")
        print("Path:", f"{self.path}")


    def write(self, value):
        with open(f"{self.path}/value", "w") as f:
            f.write(str(value))

    def read(self):
        with open(f"{self.path}/value", "r") as f:
            return int(f.read().strip())

    def cleanup(self):
        try:
            with open("/sys/class/gpio/unexport", "w") as f:
                f.write(self.gpio_num)
        except:
            pass
```

- **Gestión de PWM (`pwm_sys.py`):** Controla la Modulación por Ancho de Pulso[3] para la velocidad de los motores. Implementa una *secuencia de inicialización segura* para evitar errores del kernel (Errno 22 - Invalid Argument). La secuencia estricta es: deshabilitar → poner ciclo de trabajo a 0 → establecer periodo → habilitar.

```
import os
    import time

    class PwmSysfs:
        def __init__(self, pin_name, period=20000, duty_cycle=0, enable=1):
```

```
            self.mapping = {
                "P9_14": "pwm-2:0", "P9_16": "pwm-2:1"
            }

            if pin_name not in self.mapping:
                raise ValueError(f"Pin␣{pin_name}␣no␣mapeado␣en␣la␣clase␣
                    PwmSysfs")

            self.pwm_num = self.mapping[pin_name]
            self.path = f"/sys/class/pwm/{self.pwm_num}"




            self.set_enable(0)


            self.set_duty_cycle(0)

            self.set_period(period)

            self.set_duty_cycle(duty_cycle)

            self.set_enable("1")
            print(f"PWM␣{pin_name}␣iniciado␣correctamente.")
            print("Path:", f"{self.path}")


        def set_period(self, period_ns):
            try:
                with open(f"{self.path}/period", "w") as f:
                    f.write(str(int(period_ns)))
            except OSError as e:
                print(f"Error␣escribiendo␣Periodo:␣{e}")


        def set_duty_cycle(self, duty_ns):
            try:
                with open(f"{self.path}/duty_cycle", "w") as f:
                    f.write(str(int(duty_ns)))
            except OSError as e:
                print(f"Error␣escribiendo␣Duty:␣{e}␣(Recuerda:␣Duty␣no␣puede␣ser
                    ␣mayor␣que␣Period)")

        def set_enable(self, enable):
            try:
                with open(f"{self.path}/enable", "w") as f:
                    f.write(str(enable))
            except OSError as e:
                print(f"Error␣cambiando␣Enable:␣{e}")

        def cleanup(self):
            self.set_enable("0")

            try:
                with open(f"{self.chip_path}/unexport", "w") as f:
                    f.write(self.channel)
            except:
                pass
```

La figura 2.2 muestra la referencia que hemos seguido a la hora de implementar el rango de frecuencia del PWM

2.0.3. Capa de Controladores (Drivers `src/drivers/`)

Contiene la lógica de alto nivel de los componentes físicos.

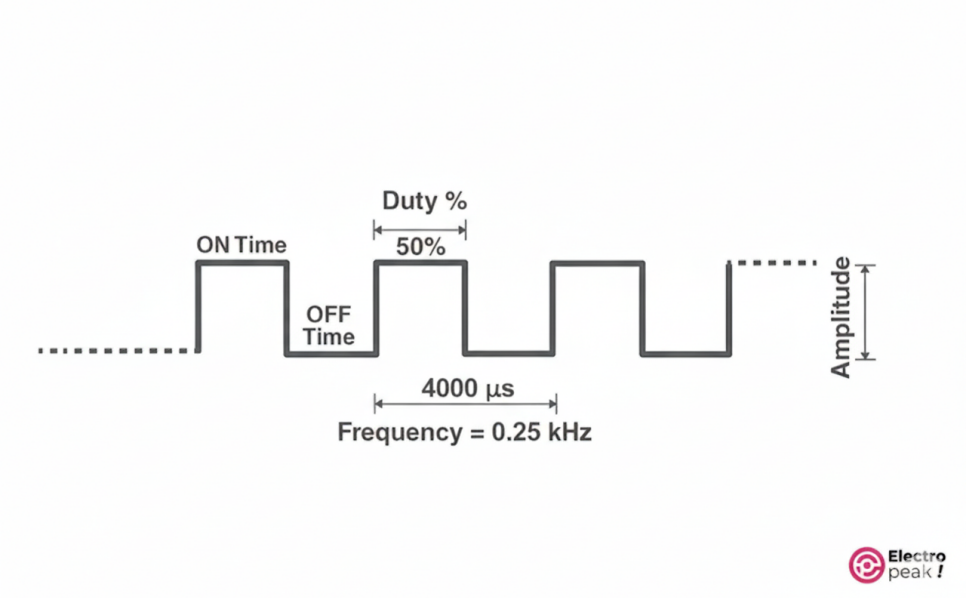

Figura 2.2: Grafica pulsos de onda cuadada PWM

`motor_dc.py` y `sensor_dc.py`: Transforman comandos lógicos (ej. "velocidad 50 %") en señales eléctricas y gestionan la lectura de distancias.

`coche_dc.py`: Actúa como una fachada que unifica el control de ambos motores y el sensor en una sola entidad simplificada.

- **Motores DC (`motor_dc.py`):** Abstrae el control de un motor individual. Recibe comandos de velocidad en porcentaje (-100 a 100) y realiza los cálculos matemáticos para convertir ese porcentaje en nanosegundos para el ciclo de trabajo del PWM.

$$PWM_{value} = \text{abs}(velocidad) \times \left(\frac{Periodo_{ref}}{Escala_{ref}} \right) \tag{2.1}$$

Actualmente, el sistema está calibrado para que un 60 % de velocidad corresponda a un ciclo de trabajo de 2000ns sobre un periodo de 4000ns.

```
from hal.gpio_sys import GpioSysfs as GPIO
from  hal.pwm_sys import PwmSysfs as PWM

class Motor:
    def __init__(self, pin_pwm, pin_dirA, pin_dirB, pin_encoder, frequency=0.25

        self.gpio_dirA=GPIO(pin_dirA, "out")  # Configura el pin
        self.gpio_dirA.write(0)               # Inicializa en bajo
        self.gpio_dirB=GPIO(pin_dirB, "out")
        self.gpio_dirB.write(0)               # Inicializa en bajo
        self.gpio_encoder=GPIO(pin_encoder, "in")  # Configura el pin del
            encoder


        self.periodo = int(1000 / frequency)

        self.gpioPWM = PWM(pin_pwm, self.periodo, 0, 1)
        self.pin_encoder = pin_encoder # Guardamos referencia (aunque no se usa
            aqu  todav a)
        print(f"Motor inicializado: PWM={pin_pwm}, DIR={pin_dirA}/{pin_dirB}")

    def leer_encoder(self):

        return self.gpio_encoder.read()
```

```
    def set_speed(self, speed):

        # 1. Limitar entrada entre -100 y 100
        speed = max(min(speed, 100), -100)

        # 2. Direcci n
        if speed > 0:
            self.gpio_dirA.write(1)
            self.gpio_dirB.write(0)
        elif speed < 0:
            self.gpio_dirA.write(0)
            self.gpio_dirB.write(1)
        else:
            self.stop()
            return

        # 3. Conversi n de Escala (La parte que pediste)

        pwm_value = int(abs(speed) * (2000 / 60))

        if pwm_value >= self.periodo:
            pwm_value = self.periodo - 10

        self.gpioPWM.set_duty_cycle(pwm_value)



    def stop(self):
        self.gpioPWM.set_duty_cycle(0)
        self.gpio_dirA.write(0)
        self.gpio_dirB.write(0)
```

- **Sensor Ultrasónico** (`sensor_dc.py`)**:** Implementa la lógica de medición de distancia mediante pulsos. Envía un pulso `TRIGGER` de $10\mu s$ y mide el tiempo en alto del pin `ECHO`. Incluye mecanismos de *timeout* para evitar bloqueos si el sensor no recibe respuesta o la señal se pierde.

```
import time
from hal.gpio_sys import GpioSysfs as GPIO

class Sensor:
    def __init__(self, pin_echo, pin_trigger):

        self.gpio_echo = GPIO(pin_echo, "in")
        self.gpio_trigger = GPIO(pin_trigger, "out")
        self.gpio_trigger.write(0)

        print(f"Sensor inicializado. TRIG:{pin_trigger}, ECHO:{pin_echo}")

    def medir_distancia(self):
        import time

        self.gpio_trigger.write(1)
        time.sleep(0.000015) # Un pel n  m  s de 10us por seguridad
        self.gpio_trigger.write(0)

        timeout_start = time.time()
        while self.gpio_echo.read() == 0:

            if time.time() - timeout_start > 0.1:

                print("Error: El sensor no responde (Check VCC/GND)")
                return None

        start_time = time.time()

        while self.gpio_echo.read() == 1:
```

```
            if time.time() - start_time > 0.1:
                print("Aviso: Objeto fuera de rango o se al perdida")
                return -1

        stop_time = time.time()

        elapsed_time = stop_time - start_time



        calibracion = 0.0005
        elapsed_time = max(0, elapsed_time - calibracion)

        distance = (elapsed_time * 34300) / 2
        return distance
```

2.0.4. Capa Transversal y Utilidades (`src/utils/` y `src/config/`)

`robot_state.py`: Implementa una memoria compartida segura (*Thread-Safe*) que permite la comunicación entre el servidor web y el hilo de control del robot sin conflictos.

```
class RobotState:

    def __init__(self):
        self.mode = "STOP"      # Modos: STOP, MANUAL, AUTO
        self.command = "libre"  # Comandos: adelante, atras, derecha, izquierda
        self.speed = 2000         # Velocidad actual (0-100)
        self.distance=0      # Distancia del sensor ultras nico
        self.running = True     # Para apagar el programa suavemente
        self.lock = threading.Lock() # Sem foro para evitar conflictos de
            memoria

    def update(self, mode=None, command=None, speed=None, distance=None):
        with self.lock:
            if mode: self.mode = mode
            if command: self.command = command
            if speed is not None: self.speed = int(speed)
            if distance is not None: self.distance = distance

    def get_state(self):
        """Devuelve una copia del estado actual"""
        with self.lock:
            return self.mode, self.command, self.speed, self.distance
```

`logger.py`: Facilita la carga dinámica de la configuración.

```
@staticmethod
    def load_config(file_path):


        configuracion = {}
        try:
            with open(file_path, 'r') as f:
                config = json.load(f)
        except Exception as e:
            print(f"Error al cargar el archivo de configuraci n:{e}")
            newfile_path="src/config/settings.json"
            with open(newfile_path, 'r') as f:
                config = json.load(f)



        for key, value in config.items():
```

```
            if "motor" in key:
                if "derecho" in key:
                    m=Motor(value["pin_pwm"], value["pin_dirA"], value["pin_dirB
                        "], value["pin_encoder"], value.get("frequency", 2000))
                    configuracion["motor_derecho"] = m
                elif "izquierdo" in key:
                    m=Motor(value["pin_pwm"], value["pin_dirA"], value["pin_dirB
                        "], value["pin_encoder"], value.get("frequency", 2000))
                    configuracion["motor_izquierdo"] = m
            elif "sensor" in key:
                sensor=Sensor(value["pin_echo"], value["pin_trigger"])
                configuracion["sensor_ultrasonido"] = sensor

        coche = Coche(configuracion["motor_izquierdo"], configuracion["
            motor_derecho"], configuracion["sensor_ultrasonido"])
        return coche
```

`settings.json`**:** Define el mapeo físico de pines y frecuencias, permitiendo cambios de hardware sin modificar el código.

```
{
"motor_izquierdo": {
   "pin_dirA": "P8_07",
   "pin_dirB": "P8_08",
   "pin_pwm": "P9_16",
   "pin_encoder": "P9_25",
   "frequency": 0.25
 },

   "motor_derecho": {
       "pin_dirA": "P8_09",
       "pin_dirB": "P8_10",
       "pin_pwm": "P9_14",
       "pin_encoder": "P9_18",
       "frequency": 0.25
   },

   "sensor_ultrasonido": {
       "pin_trigger": "P8_36",
       "pin_echo": "P9_17"
   }
}
```

- **Clase Coche (`coche_dc.py`):** Actúa como un fachada (*Facade Pattern*), agrupando los dos motores y el sensor en un solo objeto controlable. Provee métodos de alto nivel como `avanzar()`, `girar_izquierda()` o `medir_distancia()`.

```
{
class Coche:
   def __init__(self, motor_izquierdo, motor_derecho, sensor_ultrasonido):
       self.motor_izquierdo = motor_izquierdo
       self.motor_derecho = motor_derecho
       self.sensor_ultrasonido = sensor_ultrasonido

   def leer_sensores(self):
       distancia = self.sensor_ultrasonido.medir_distancia()
       return distancia

   def avanzar(self, velocidad):
       self.motor_izquierdo.set_speed(velocidad)
       self.motor_derecho.set_speed(velocidad)

   def retroceder(self, velocidad):
       self.motor_izquierdo.set_speed(-velocidad)
       self.motor_derecho.set_speed(-velocidad)
...
```

```
}
```

El sistema se basa en una arquitectura multihilo donde la comunicación entre el servidor web (interfaz de usuario) y el control de hardware (motores y sensores) se realiza mediante un objeto de memoria compartida protegido contra condiciones de carrera.

Clase `RobotState`: Sincronización y Estado

La clase `RobotState` actúa como el nexo de unión entre los diferentes hilos[4] de ejecución. Su función principal es almacenar el estado actual del robot y garantizar la integridad de los datos mediante mecanismos de exclusión mutua.

- **Gestión de Concurrencia:** Se utiliza un objeto `threading.Lock` (semáforo) para gestionar el acceso a las variables. Esto es crítico porque el servidor web escribe comandos asíncronamente mientras el bucle de control los lee. El uso de `with self.lock:` asegura que las operaciones de lectura y escritura sean atómicas.
- **Variables de Estado:**
  - `mode`: Define el comportamiento actual (*STOP*, *MANUAL*, *AUTO*).
  - `command`: Almacena la última instrucción de movimiento recibida (*adelante*, *atras*, etc.).
  - `speed`: Valor entero (0-100) que representa el ciclo de trabajo PWM de los motores.
  - `distance`: Variable de telemetría que almacena la última lectura del sensor ultrasónico para ser consumida por la interfaz web.

```
def robot_logic_thread(shared_state, robot):
    print("Hilo␣del␣robot␣iniciado...")

    last_mode = None
    last_cmd = None

    while shared_state.running:
        # 1. Leer estado actual
        mode, cmd, speed, distance = shared_state.get_state()


        try:

            dist = robot.medir_distancia()
            if dist is not None:

                shared_state.update(distance=dist)
        except Exception as e:
            print(f"Error␣midiendo␣distancia:␣{e}")

        # 2. Solo imprimimos si el modo ha cambiado (Para limpiar la consola)
        if mode != last_mode:
            print(f"\n[ROBOT]␣Cambio␣de␣modo:␣{last_mode}␣->␣{mode}")

            if mode == "STOP":
                robot.detener()

        # 3. L gica continua
        try:
            if mode == "STOP":


                robot.detener()

            elif mode == "MANUAL":

                if cmd != last_cmd or mode != last_mode:
                    if cmd == "adelante":
```

```
                    robot.avanzar(speed)
                elif cmd == "atras":
                    robot.retroceder(speed)
                elif cmd == "derecha":
                    robot.girar_derecha(speed)
                elif cmd == "izquierda":
                    robot.girar_izquierda(speed)
                else:
                    robot.detener()

            elif mode == "AUTO":
                distancia = robot.sensor.medir_distancia()
                if distancia and 0 < distancia < 20:

                    import random
                    if random.choice([True, False]):
                        robot.girar_izquierda(speed)
                    else:
                        robot.girar_derecha(speed)

                else:
                    robot.avanzar(speed)

            last_mode = mode
            last_cmd = cmd

            time.sleep(0.1)

        except Exception as e:
            print(f"Error: {e}")
            robot.detener()
```

Bucle de Control: `robot_logic_thread`

Esta función representa el çerebro"del robot. Se ejecuta en un hilo independiente (*daemon thread*) y procesa la lógica de decisión aproximadamente 10 veces por segundo (`time.sleep(0.1)`).

Fases del Ciclo de Ejecución

1. **Adquisición de Datos:** En cada iteración, el hilo extrae una copia del estado actual (`get_state`) y realiza una lectura del sensor ultrasónico mediante `robot.medir_distancia()`. Si la lectura es válida, actualiza inmediatamente la variable `distance` en la memoria compartida, permitiendo la monitorización en tiempo real.

2. **Máquina de Estados Finitos:** El comportamiento del robot se determina según el modo activo, tambien en la figura:

   - **Modo STOP:** Estado de seguridad. Se fuerza la detención de los motores invocando `robot.detener()`.
   - **Modo MANUAL:** Permite el teleoperado. Para optimizar el uso de la CPU y el bus de comunicaciones, se implementa una lógica de *cambio de estado*: las órdenes de movimiento solo se envían al hardware si el comando o el modo han cambiado respecto a la iteración anterior (`if cmd != last_cmd`).
   - **Modo AUTO:** Implementa un algoritmo básico de evasión de obstáculos:

$$\text{Acción} = \begin{cases} \text{Giro Ante Obstaculo (Aleatorio Derecha o Izquierda)} & \text{si } 0 < d < 20 \text{ cm} \\ \text{Avanzar} & \text{en otro caso} \end{cases} \tag{2.2}$$

     Al detectar un obstáculo a menos de 20 cm, el sistema utiliza la librería `random` para seleccionar una dirección de giro aleatoria, evitando así que el robot quede atrapado en bucles infinitos al enfrentar esquinas.

     Control de Giro mediante Encoders Para garantizar que los giros sean consistentes, se han implementado funciones que utilizan la lectura de pulsos de los motores. A diferencia de un giro basado en tiempo, el uso de encoders permite medir el movimiento real de la rueda.

Algoritmo de Conteo de Pulsos Las funciones `girar_izquierda_endoder` y `girar_derecha_endoder` utilizan un bucle de control basado en el estado del sensor. Para evitar falsos positivos y contar correctamente cada muesca del disco del encoder, se implementa una lógica de espera de flanco:

a) Se activa el motor a la velocidad definida.
b) Se detecta un nivel alto (*flanco de subida*) en el pin del encoder: `if leer_encoder() == 1`.
c) Se incrementa el contador de pulsos.
d) Anti-rebote por software: El programa entra en un bucle de espera (`while encoder == 1: pass`) hasta que el sensor vuelve a nivel bajo, asegurando que un solo pulso físico se cuente una sola vez.
e) El proceso se repite hasta alcanzar el objetivo de **32 pulsos**.

Modelado Matemático del Giro Siendo P el número de pulsos detectados y $P_{target} = 32$, el motor se detiene cuando se cumple la condición:

$$\sum_{t=0}^{T} \Delta Pulsos(t) \geq 32 \tag{2.3}$$

Este método proporciona una mayor precisión frente a las variaciones de voltaje de la batería, ya que el robot no se detiene hasta que la rueda ha girado físicamente la distancia requerida.

```
def girar_izquierda_endoder(self, velocidad):
        """Gira␣a␣la␣izquierda␣girando␣motor␣derecho␣tras␣32␣pulsos␣del␣
            encoder."""
        self.motor_derecho.leer_encoder() # devuelve 0 o 1
        pulsos = 0
        while pulsos < 32:
            self.motor_derecho.set_speed(velocidad)
            if self.motor_derecho.leer_encoder() == 1:
                pulsos += 1
                while self.motor_derecho.leer_encoder() == 1:
                    pass  # Espera a que baje a 0
        self.motor_derecho.set_speed(0)

def girar_derecha_endoder(self, velocidad):
        """Gira␣a␣la␣derecha␣girando␣motor␣derecho␣tras␣32␣pulsos␣del␣encoder
            ."""
        self.motor_izquierdo.leer_encoder() # devuelve 0 o 1
        pulsos = 0
        while pulsos < 32:
            self.motor_derecho.set_speed(velocidad)
            if self.motor_izquierdo.leer_encoder() == 1:
                pulsos += 1
                while self.motor_izquierdo.leer_encoder() == 1:
                    pass  # Espera a que baje a 0
        self.motor_izquierdo.set_speed(0)
```

Interfaz de Usuario (`src/templates/`)

La interacción con el usuario se realiza mediante una aplicación web ligera basada en **Flask** y **SocketIO**[5].
La BeagleBone AI (Rev A1)[6] tiene el modo de punto de acceso (hotspot) crea su propia red inalámbrica para a la que se puede acceder usando la contraseña por defecto *BeagleBone*

- **Servidor Web (`web_server.py`):** Utiliza *WebSockets*[7] para una comunicación bidireccional en tiempo real.
  - **Recepción:** Escucha eventos `'comando'` desde la web para actualizar el estado del robot.
  - **Transmisión:** Ejecuta una tarea en segundo plano (`broadcast_sensor_data`) que lee la distancia del sensor cada 0.5 segundos y la emite a todos los clientes conectados mediante el evento `'sensor_update'`.

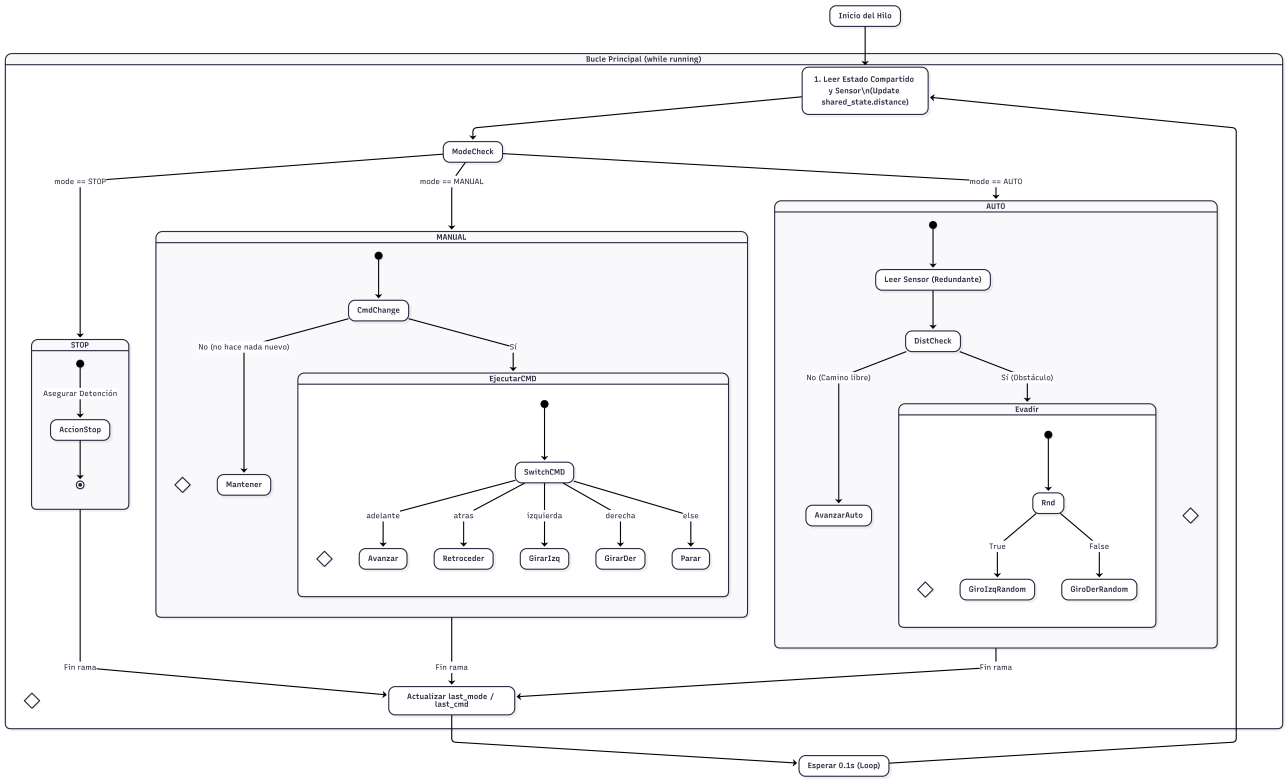

Figura 2.3: Digrama de funcionamiento de los estados de la plataforma

```
robot_shared_state = RobotState()

server_app = Flask(__name__, template_folder='templates')
socketio = SocketIO(server_app, cors_allowed_origins="*", async_mode='eventlet')


@server_app.route('/')
def index():
    return render_template('index.html')

@socketio.on('comando')
def handle_command(json):
    raw_data = json['data'] # Recibe: "adelante 2000" o "stop 0"
    print(f"[WEB] Recibido crudo: {raw_data}")

    # 1. Separar el texto por espacios
    # "adelante 2000".split()  ->  ["adelante", "2000"]
    # "stop".split()           ->  ["stop"]
    parts = raw_data.split()

    cmd = parts[0] # La primera parte es el comando ("adelante")

    # 2. Si hay una segunda parte (la velocidad), la procesamos
    if len(parts) > 1:
        try:
            velocidad = int(parts[1]) # Convertimos "2000" a numero 2000
            if robot_shared_state:
                robot_shared_state.update(speed=velocidad)
        except ValueError:
            print("Error: La velocidad no es un n mero v lido")

    # 3. Ejecutar el comando limpio
    if robot_shared_state:
        if cmd == "stop":
```

```
            robot_shared_state.update(command="stop")
        else:
            robot_shared_state.update(mode="MANUAL", command=cmd)

@socketio.on('velocidad')
def handle_speed(json):
    val = int(json['data'])
    if robot_shared_state:
        robot_shared_state.update(speed=val)

@socketio.on('modo_switch')
def handle_mode():
    if robot_shared_state:
        current_mode = robot_shared_state.get_state()[0]
        new_mode = "AUTO" if current_mode == "MANUAL" else "MANUAL"
        robot_shared_state.update(mode=new_mode)
        print(f"[WEB] Modo cambiado a {new_mode}")

def broadcast_sensor_data():
    """Env a los datos del sensor a todos los clientes conectados cada 0.5s"""
    while True:
        if robot_shared_state:


            state = robot_shared_state.get_state()
            dist = state[3] # El cuarto elemento es la distancia

            socketio.emit('sensor_update', {'distancia': dist})

        socketio.sleep(0.5) # Espera no bloqueante

def start_server(state_instance):
    global robot_shared_state
    robot_shared_state = state_instance
    socketio.start_background_task(broadcast_sensor_data)
    eventlet.wsgi.server(eventlet.listen(('', 5000)), server_app)
```

- **Interfaz de Usuario (`index.html`):** Proporciona controles visuales (botones de dirección, deslizador de velocidad) y visualización de telemetría. Utiliza JavaScript para capturar eventos táctiles (mover hacia delante, atrás, derecha, izquierda) y de ratón, enviando comandos al servidor sin necesidad de recargar la página, la imagen 2.4 muestra la página web.

2.0.5. Orquestación del Sistema (`main.py`)

El punto de entrada principal inicializa la configuración desde un archivo JSON y levanta los hilos de ejecución paralela:

1. **Hilo de Lógica:** Ejecuta el bucle de control que verifica el modo (MANUAL/AUTO/STOP) y actúa sobre los motores. También realiza la lectura continua del sensor.
2. **Hilo de Consola:** Permite el control local mediante teclado para depuración.
3. **Servidor Web:** Se inicia mediante `start_server` utilizando `eventlet` para soportar concurrencia asíncrona, nos conectamos al wifi inalambrico de la placa *BeagleBone-C616* con la constraseña *BeagleBone* , se muestra en la figura 2.5 .

```
def main():
    from web_server import start_server, robot_shared_state

    print("Iniciando Robot...")

    # 2. Cargar Configuraci n y Hardware
    try:
```

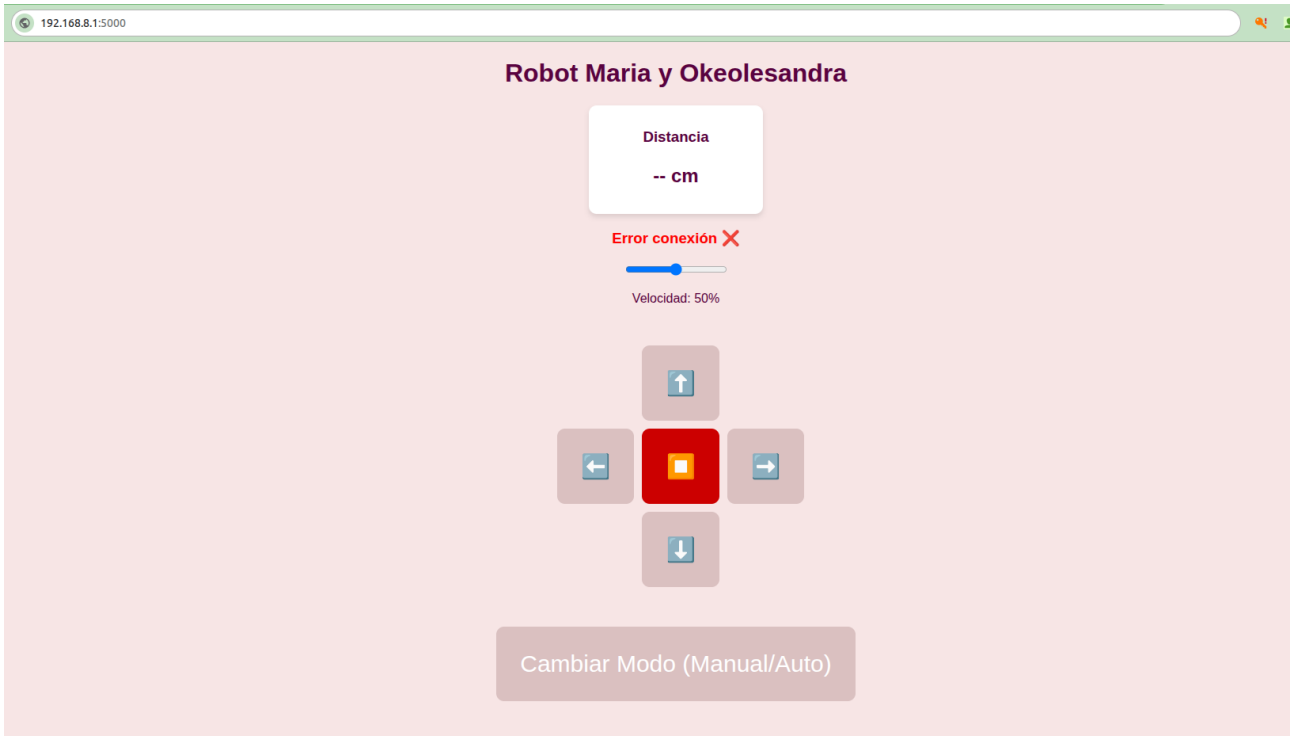

Figura 2.4: Interfaz Web del en https://192.168.8.1:5000

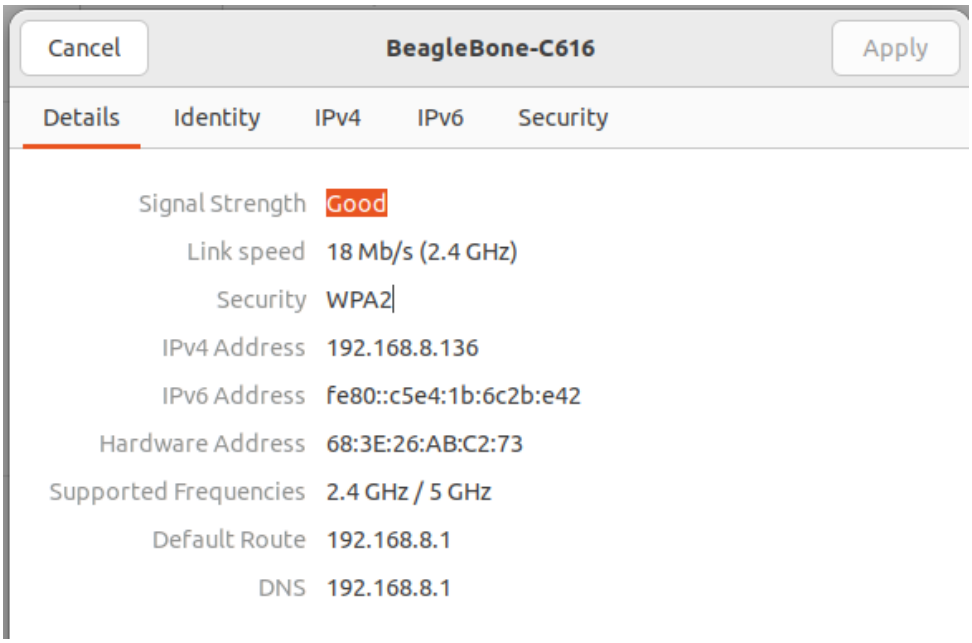

Figura 2.5: Wifi Inalambrico de la placa

```
        robot = Logger.load_config("./config/settings.json")
    except Exception as e:
        print(f"Error␣cargando␣config:␣{e}")
        return

    # 3. Arrancar Hilo de L gica
    t_logic = threading.Thread(target=robot_logic_thread, args=(robot_shared_state,
        robot))
    t_logic.daemon = True # Se cerrar  si el main se cierra
    t_logic.start()

    # 4. Arrancar Servidor Web

    print("Lanzando␣servidor␣web...")
    t_server = threading.Thread(target=start_server, args=(robot_shared_state,))
    t_server.daemon = True
    t_server.start()

    print(f"Web␣disponible␣en:␣http://192.168.8.1:5000")

    # 5. Bucle Principal (Consola)

    console_thread(robot_shared_state)

    print("Main␣cerrado.")
```

2.1. Automatización del Arranque: Systemd Service

Para garantizar que el software del robot se inicie automáticamente al encender la `BeagleBone` y se recupere ante posibles fallos, se ha implementado un servicio de sistema utilizando *systemd*. El archivo de configuración se ubica en `/etc/systemd/system/robot.service`.

2.1.1. Código de Configuración

```
[Unit]
Description=Robot Maria y Okeolesandra
After=network.target

[Service]
Type=simple
User=root
WorkingDirectory=/home/debian/app/src
Environment=PYTHONPATH=/home/debian/.local/lib/python3.7/site-packages
ExecStart=/usr/bin/python3 /home/debian/app/src/main.py
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
```

Capítulo 3

Conclusión

La realización de esta práctica ha representado un desafío técnico de gran envergadura, marcando nuestra primera incursión en la integración de software con sistemas físicos reales. Al inicio del proyecto, conceptos fundamentales como la Modulación por Ancho de Pulsos (PWM) o la gestión de registros en sistemas embebidos eran totalmente desconocidos para nosotras.

El proceso de desarrollo nos ha permitido evolucionar desde el aprendizaje de la arquitectura de la BeagleBone AI hasta la implementación de una infraestructura de control compleja. Durante este camino, nos enfrentamos a retos que trascendieron la programación:

Uno de los mayores aprendizajes fue la gestión de niveles lógicos. Tuvimos que aprender a convivir con sistemas de 3.3V y 5V, entendiendo que una conexión directa podría dañar la placa. La implementación de divisores de voltaje y adaptadores para el sensor ultrasónico fue crítica para proteger la integridad de la BeagleBone. Experimentamos grandes problemas debido a problemas de cableado y falsos contactos. El sensor ultrasónico, que en primera instancia no ofrecía lecturas, nos obligó a realizar un seguimiento exhaustivo de la señal y a entender la importancia de una masa (GND) común y estable. El uso de hilos (threading) para gestionar simultáneamente la lógica de navegación autónoma y el conteo preciso de los encoders.

Cuando Antonio mencionó la posibilidad de crear un un servidor basado en WebSockets (Socket.io), nos desperto la curiosidad, superando la barrera de la programación convencional en PC para crear interfaces de control remoto en tiempo real.

Lo que comenzó como una dificultad para comprender la conexión de un chip, ha culminado en el diseño de una plataforma robótica funcional y autónoma. Esta experiencia no solo ha fortalecido nuestras habilidades de programación, sino que nos ha proporcionado una visión integral sobre cómo el código interactúa con el mundo físico.

Bibliografía

[1] Elecfreaks, "Ultrasonic sensor hc-sr04 datasheet," 2014. Especificaciones para el cálculo de distancia y niveles lógicos.

[2] C. Hallinan, *Embedded Linux Primer: A Practical Real-World Approach.* Prentice Hall, 2nd ed., 2010. Consulta para la gestión de GPIO y drivers en Linux.

[3] T. L. K. Archives, "Pulse width modulation (pwm) interface," 2023. Documentación sobre la interacción con SysFS para control de potencia.

[4] P. S. Foundation, "Threading — manage concurrent threads," 2023. Documentación oficial para la gestión de hilos en el robot.

[5] M. Grinberg, "Flask-socketio documentation," 2023. Versión utilizada para la comunicación bidireccional en tiempo real.

[6] J. Kridner and R. Nelson, *BeagleBone AI System Reference Manual (Rev A1).* BeagleBoard.org, September 2019. Manual de referencia oficial para hardware y arquitectura del procesador Sitara AM5729.

[7] E. Contributors, "Eventlet: Concurrent networking library for python," 2023. Utilizado como servidor WSGI concurrente.