Robot controlado por WiFi

Agostini, Emiliano (autor)
Departamento de Ingeniería Electrónica,
Facultad Regional San Francisco, Córdoba, Argentina.
emilianoagostini14@gmail.com

Cortesse, Agustín (autor)
Departamento de Ingeniería Electrónica,
Facultad Regional San Francisco, Córdoba, Argentina.
agucortesse19@gmail.com

Índice

1. Resumen

El presente proyecto consiste en el diseño e implementación de un **robot controlado por WiFi** utilizando el microcontrolador **ESP32** como unidad principal de procesamiento y comunicación. El sistema permite controlar en tiempo real el desplazamiento del robot a través de una interfaz web alojada en el propio microcontrolador, sin requerir conexión a Internet ni dispositivos intermedios.

La arquitectura del proyecto se compone de diferentes bloques funcionales que trabajan de manera coordinada: el módulo de conectividad WiFi, el servidor web embebido, el sistema de control de motores basado en un puente H y la gestión de tareas mediante el sistema operativo en tiempo real **FreeRTOS**. La comunicación entre el usuario y el robot se realiza mediante el protocolo **HTTP**, lo que garantiza una operación universal desde cualquier navegador web.

El sistema demostró un funcionamiento estable, con tiempos de respuesta reducidos y un control fluido de los motores.

2. Introducción

En la actualidad, los sistemas de control inalámbrico constituyen una herramienta esencial dentro del campo de la ingeniería electrónica y de control, permitiendo desarrollar dispositivos cada vez más flexibles, accesibles y con capacidad de interacción remota. En este contexto, el presente proyecto propone el desarrollo de un **robot controlado por WiFi**, cuyo objetivo principal es demostrar la integración entre control embebido, comunicación inalámbrica y arquitectura multitarea.

El proyecto se basa en el microcontrolador **ESP32**, un dispositivo de bajo costo y alta versatilidad que integra conectividad WiFi y Bluetooth junto con una potente capacidad de procesamiento. Gracias a estas características, el ESP32 permite ejecutar tareas simultáneas como la gestión de comunicaciones, el control de motores y la atención de eventos del usuario.

El sistema implementa una estructura modular en la que cada bloque cumple una función específica: la interfaz web facilita la interacción del usuario; el servidor HTTP integrado procesa las solicitudes recibidas; el control de motores, a través de un puente H, traduce los comandos en movimientos físicos; y el sistema operativo en tiempo real **FreeRTOS** garantiza una ejecución concurrente y eficiente de las tareas. Este enfoque no solo permite un control fluido y confiable del robot, sino que también sienta las bases para proyectos de mayor complejidad dentro del ámbito de la robótica móvil y el IoT.

3. Objetivos

3.1. Objetivo general

Diseñar e implementar un **robot móvil controlado por WiFi** utilizando el microcontrolador **ESP32** como unidad central de procesamiento, capaz de recibir y ejecutar órdenes de movimiento enviadas desde una interfaz web. El sistema debe traducir los comandos recibidos en señales de control **PWM** aplicadas a un **puente H**, permitiendo el control direccional y de velocidad de los motores de manera eficiente y estable.

3.2. Objetivos específicos

- Desarrollar un **servidor web embebido** en el ESP32 que funcione como interfaz de comunicación entre el usuario y el robot, permitiendo el envío y recepción de comandos en tiempo real mediante el protocolo HTTP.
- Estructurar el software mediante el uso del sistema operativo **FreeRTOS**, asignando las tareas principales a núcleos diferentes del ESP32 con el fin de optimizar la respuesta y evitar bloqueos en la ejecución.

4. Desarrollo

Con el objetivo de lograr una comprensión clara y ordenada del funcionamiento del sistema, a continuación se presenta la descripción individual de cada uno de los bloques que lo componen. En cada caso se detalla su función específica, el principio de operación que lo rige y su relación con los demás bloques del conjunto, de manera que se evidencie el aporte de cada uno al comportamiento global del sistema.

1. Microcontrolador ESP32

El **ESP32** es un microcontrolador de bajo costo y alto rendimiento desarrollado por la empresa **Espressif Systems**. Está basado en un procesador **dual-core Tensilica Xtensa LX6**, con frecuencia de operación de hasta 240 MHz, e integra una amplia gama de periféricos que lo convierten en una plataforma ideal para aplicaciones de *Internet de las Cosas* (IoT) y control embebido.

Entre sus características principales se destacan:

- Conectividad inalámbrica **WiFi 802.11 b/g/n** y **Bluetooth/BLE** integradas.
- Memoria flash y SRAM internas para almacenamiento y ejecución de programas.

- Amplio número de pines **GPIO (General Purpose Input/Output)** configurables.
- Soporte para interfaces de comunicación como **UART**, **SPI**, **I2C**, **ADC**, **PWM**, entre otras.
- Bajo consumo energético gracias a sus modos de suspensión y gestión avanzada de energía.

Figura 1: Placa que incorpora el microcontrolador - ESP32 Devkit V1

4.1. Control de motores

El control de movimiento del robot se realiza mediante un puente H **L298N**, un circuito ampliamente utilizado para invertir el sentido de giro de motores de corriente continua (DC). Este tipo de configuración permite controlar tanto la dirección como la velocidad de los motores mediante señales digitales y moduladas en ancho de pulso (PWM) generadas por el microcontrolador **ESP32**.

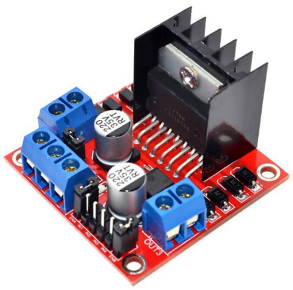

Figura 2: Modulo controlardor de motores

1. Control por modulación PWM

La velocidad de los motores se regula utilizando la técnica de **modulación por ancho de pulso (PWM, Pulse Width Modulation)**. El ESP32 genera una señal cuadrada de frecuencia constante, variando el ciclo de trabajo (*duty cycle*) entre 0 % y 100 %. Cuanto mayor es el tiempo en nivel alto de la señal, mayor es el voltaje promedio aplicado al motor, y por tanto, mayor su velocidad.

En el código del proyecto, se emplean los periféricos de hardware del ESP32 encargados de generar señales PWM precisas, configurando la frecuencia, resolución y pines de salida mediante las funciones del framework **ESP-IDF**. Estas señales se envían a los pines de entrada del puente H, el cual conmuta los transistores de potencia según el ciclo de trabajo establecido.

2. Funciones de dirección y control

El control de dirección se implementa mediante funciones específicas que modifican el estado lógico de los pines asociados al puente H. A continuación, se describen las principales:

- `motores_ava()`: activa las señales de los pines correspondientes a ambos motores en el sentido directo.
- `motores_ret()`: invierte la polaridad de los pines de control, haciendo girar ambos motores en sentido contrario.
- `motores_izq()`: detiene o invierte el motor derecho mientras el izquierdo avanza, permitiendo el giro del robot.
- `motores_der()`: realiza la operación inversa a la anterior, girando el robot hacia el lado opuesto.

3. Conectividad WiFi

El ESP32 tiene la capacidad de establecer enlaces de datos de alta velocidad y baja latencia dentro de una red local, sin requerir módulos externos. El chip incluye además un conjunto de antenas internas y amplificadores de potencia (*PA/LNA*) que garantizan un alcance estable y una comunicación confiable incluso en entornos con interferencias electromagnéticas moderadas.

El sistema WiFi del ESP32 puede configurarse en tres modos principales de operación:

- **Modo estación (STA):** el dispositivo se conecta a una red WiFi existente, como la de un enrutador o punto de acceso externo.

- **Modo punto de acceso (AP):** el ESP32 crea su propia red WiFi, permitiendo que otros dispositivos se conecten directamente a él.
- **Modo mixto (AP+STA):** combina ambos modos, permitiendo al ESP32 conectarse a una red mientras actúa simultáneamente como punto de acceso para otros dispositivos.

En el presente proyecto, el microcontrolador opera en **modo punto de acceso**, asignando una dirección IP fija (`192.168.4.1`) y gestionando directamente las conexiones de los dispositivos clientes. De esta forma, el sistema no depende de un enrutador externo ni de conexión a Internet, lo que simplifica la instalación y garantiza un control local inmediato.

La pila de red del ESP32 implementa el protocolo **TCP/IP** mediante un conjunto de tareas internas controladas por el sistema operativo en tiempo real **FreeRTOS**. Cada conexión WiFi activa se maneja a través de un conjunto de *sockets*, que permiten la transmisión y recepción de datos estructurados. Gracias a esta arquitectura, el microcontrolador puede atender múltiples solicitudes simultáneamente, mantener sesiones activas y responder a comandos del usuario con muy baja latencia.

Para la gestión del módulo inalámbrico, el entorno **ESP-IDF** proporciona una API de alto nivel que permite configurar los parámetros de red (SSID, contraseña, canal, modo, IP estática o dinámica) y registrar *event handlers* que detectan eventos como la conexión o desconexión de clientes. Esta flexibilidad permite adaptar el comportamiento del sistema a distintos escenarios de uso, desde redes cerradas locales hasta aplicaciones distribuidas de IoT.

4.2. Servidor web

El sistema de control WiFi se basa en un microcontrolador **ESP32**, el cual integra un módulo de comunicación inalámbrica y un procesador de doble núcleo, permitiendo ejecutar simultáneamente tareas de conexión y control de hardware. En este proyecto, el ESP32 cumple la función de *servidor web*, posibilitando el control remoto del robot mediante una red inalámbrica.

El microcontrolador se configura en modo *punto de acceso* (*Access Point*), lo que le permite crear su propia red WiFi. De esta forma, cualquier dispositivo, como un teléfono móvil o una computadora, puede conectarse directamente al ESP32 sin necesidad de un enrutador externo. Una vez establecida la conexión, el usuario accede a una página web alojada en la memoria del propio microcontrolador con una direccion IP fija.

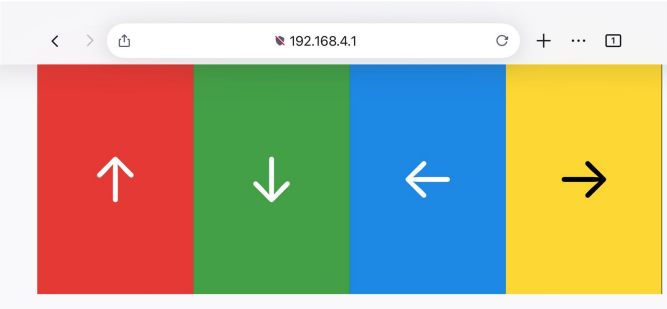

Figura 3: Interfaz Web

La interfaz web está desarrollada utilizando los lenguajes **HTML**, **CSS** y **JavaScript**, y se almacena en la memoria flash del ESP32. Esta compuesta por cuatro botones que permiten enviar comandos de control. Cuando el usuario presiona alguno de los botones, el navegador genera una *petición HTTP* hacia el servidor web integrado en el microcontrolador. Dichas peticiones utilizan el protocolo **HTTP (HyperText Transfer Protocol)**, el mismo que se emplea en la navegación web convencional.

El protocolo **HTTP (HyperText Transfer Protocol)** es el mecanismo de comunicación que permite el intercambio de información entre un *cliente* y un *servidor* dentro de una red. En este proyecto, el cliente es el navegador web del usuario, mientras que el servidor es el microcontrolador ESP32. HTTP se basa en el modelo *cliente-servidor*, donde el cliente envía una *petición* y el servidor responde con la información solicitada o ejecuta una acción en consecuencia.

Las peticiones HTTP se estructuran en mensajes de texto que siguen un formato estándar. Una solicitud típica incluye:

- **Método:** indica la acción que el cliente desea realizar (por ejemplo, `GET` para solicitar datos o `POST` para enviar información).
- **Ruta o recurso:** especifica el destino de la solicitud dentro del servidor (por ejemplo, `/ava_on` o `/ret_on`).
- **Cabeceras (headers):** contienen información adicional sobre la solicitud, como el tipo de contenido, la dirección de origen o la longitud del mensaje.
- **Cuerpo (body):** opcionalmente, puede incluir datos que el cliente envía al servidor.

En el caso del robot controlado por WiFi, el navegador envía peticiones HTTP del tipo `GET` al ESP32 cuando el usuario presiona un botón en la interfaz web. Cada ruta o URL está asociada a una función dentro del firmware que determina la acción a ejecutar. Por ejemplo:

- `/ava_on`: activa las señales PWM para avanzar.
- `/ret_on`: invierte la polaridad del puente H para retroceder.
- `/izq_on` y `/der_on`: modifican los pines de control para girar el robot.

Una vez procesada la solicitud, el ESP32 envía una *respuesta HTTP*, la cual incluye un *código de estado* (como `200 OK` si la acción se ejecutó correctamente) y, en algunos casos, información adicional en formato HTML o texto plano. Esta estructura permite una comunicación simple, confiable y universal, ya que HTTP es un protocolo ampliamente soportado por cualquier navegador moderno.

El funcionamiento del protocolo HTTP se apoya en la capa de transporte **TCP (Transmission Control Protocol)**, que garantiza la entrega ordenada y sin pérdidas de los datos. Gracias a esto, el control del robot es inmediato y seguro, incluso en redes locales sin acceso a Internet.

El firmware del ESP32 fue desarrollado en lenguaje **C/C++**, empleando el entorno de desarrollo **ESP-IDF (Espressif IoT Development Framework)**. Este entorno proporciona las bibliotecas necesarias para la gestión de la conexión WiFi, la implementación del servidor HTTP y el control de los pines **GPIO (General Purpose Input/Output)**, encargados de enviar las señales PWM al **puente H**, el cual regula la dirección y velocidad de los motores.

De manera resumida, el proceso de control se desarrolla en las siguientes etapas:

1. El ESP32 crea una red WiFi a la que el usuario se conecta.
2. El usuario accede, mediante un navegador web, a la interfaz alojada en el microcontrolador.
3. Al presionar un botón, el navegador envía una solicitud HTTP al servidor web del ESP32.
4. El ESP32 interpreta la solicitud y ejecuta la acción correspondiente sobre los motores.
5. El robot responde de forma inmediata al comando recibido.

5. Sistema operativo FreeRTOS y manejo de colas

Para la gestión de múltiples procesos simultáneos dentro del sistema, se utilizó el sistema operativo en tiempo real **FreeRTOS**, integrado de manera nativa en el framework **ESP-IDF** que emplea la placa **ESP32**. FreeRTOS (Free Real-Time Operating System) es un kernel de sistema operativo ligero, diseñado específicamente para sistemas embebidos. Su principal función es permitir la ejecución

de varias tareas en paralelo, organizando su ejecución en función de prioridades, interrupciones y disponibilidad de recursos del microcontrolador.

A diferencia de un programa secuencial, donde las instrucciones se ejecutan de manera lineal, FreeRTOS permite dividir el programa en **tareas independientes** que pueden ejecutarse de forma concurrente. Cada tarea tiene su propio espacio de pila, estado y prioridad. El planificador (scheduler) del sistema operativo se encarga de administrar el tiempo de CPU para que cada tarea se ejecute de manera eficiente según su prioridad. En el caso de la ESP32, al poseer un microcontrolador de doble núcleo, FreeRTOS permite incluso **asignar tareas a núcleos específicos**, aprovechando así el paralelismo físico del procesador.

En este proyecto se implementaron dos tareas principales:

- **Tarea del servidor web**: su función es inicializar y mantener activo el servidor HTTP que gestiona las peticiones provenientes de los clientes conectados a la red Wi-Fi creada por la ESP32. Esta tarea se ejecuta en el **núcleo 0** y se inicia mediante la función `xTaskCreatePinnedToCore()`, la cual permite especificar en qué núcleo correrá la tarea. Si por alguna razón la función `start_webserver()` retorna, la tarea se elimina mediante `vTaskDelete(NULL)` para liberar los recursos asignados.

- **Tarea de motores**: se ejecuta en el **núcleo 1** y es la encargada de controlar el movimiento del robot. Dentro de un bucle infinito, esta tarea espera recibir comandos a través de una cola (Queue) denominada `cola_motores`. Cada comando recibido corresponde a una acción específica: avanzar, retroceder, girar a la izquierda, girar a la derecha o detenerse. Según el comando recibido, se invoca la función correspondiente (`motores_ava()`, `motores_ret()`, `motores_izq()`, `motores_der()` o `motores_fre()`).

El uso de **colas (Queues)** permite la comunicación segura entre tareas dentro del entorno multitarea de FreeRTOS. En este caso, la cola `cola_motores` actúa como un canal de comunicación entre el **servidor web** y la **tarea de motores**. Cuando un usuario presiona un botón en la interfaz web, se envía un mensaje con el comando correspondiente a la cola. La tarea de motores, al recibir dicho mensaje mediante la función `xQueueReceive()`, ejecuta la acción indicada sin interferir con el funcionamiento del servidor.

De esta manera, el uso de FreeRTOS permite una ejecución concurrente y ordenada de los distintos procesos del sistema, asegurando una respuesta rápida a las acciones del usuario y una gestión eficiente de los recursos de la ESP32.

6. Desarrollo del código

En esta sección se presenta el desarrollo del código fuente que permite la interacción y coordinación entre los diferentes módulos del sistema. El programa fue implementado en lenguaje C utilizando el entorno de desarrollo **ESP-IDF**, propio del microcontrolador **ESP32**.

El código se estructura en distintos archivos y módulos que cumplen funciones específicas, lo que permite una organización modular, una mayor claridad en la lectura y facilita futuras ampliaciones o modificaciones. A lo largo de esta sección se describen las tareas principales, el flujo de ejecución y los mecanismos de comunicación interna implementados mediante **FreeRTOS** y colas (**Queues**).

6.1. Estructura general del programa

El programa se organiza en distintos módulos, cada uno con una función claramente definida:

- `main.c`: inicializa los periféricos, crea las colas y tareas principales, y gestiona la ejecución global del sistema.
- `wifi_ap.c`: configura el modo punto de acceso (Access Point) del ESP32, permitiendo la conexión WiFi del usuario.
- `servidor_web.c`: implementa el servidor HTTP, encargado de recibir las solicitudes del cliente y generar las respuestas correspondientes.
- `motores.c`: contiene las funciones de control de los motores mediante señales PWM y las rutinas de movimiento.

Esta división modular permite un desarrollo escalable y mejora la legibilidad del código, separando las responsabilidades de cada componente del sistema.

6.2. Flujo de ejecución

Al encender el sistema, el ESP32 ejecuta la función principal `app_main()` (Codigo 1), donde se inicializan los servicios básicos de la plataforma, como la memoria no volátil (`nvs_flash`), la gestión de eventos y la interfaz de red.

Posteriormente, se crea una cola de mensajes destinada a la comunicación entre el servidor web y el controlador de motores. A continuación, se inicializa el punto de acceso WiFi mediante la función `start_wifi_ap()` y se crean las dos tareas principales del sistema:

- **Tarea de motores**: interpreta y ejecuta las órdenes recibidas, controlando el sentido y la velocidad de los motores mediante las funciones correspondientes (Codigo 2).
- **Tarea del servidor web**: se encarga de recibir las solicitudes HTTP provenientes del usuario, procesarlas e interpretar los comandos de control (Codigo 3).

Cada una de estas tareas se ejecuta en un núcleo distinto del ESP32, lo que garantiza una operación en paralelo y una respuesta inmediata ante las órdenes de movimiento.

```
1           void app_main(void) {
2               // Inicializar NVS (Non-Volatile Storage) para
    configuracion WiFi
3               esp_err_t ret = nvs_flash_init();
4               if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret ==
    ESP_ERR_NVS_NEW_VERSION_FOUND) {
5                   nvs_flash_erase();
6                   nvs_flash_init();
7               }
8
9               // Inicializar stack de red y sistema de eventos
10              esp_netif_init();
11              esp_event_loop_create_default();
12
13              // Crear cola para comunicacion entre tareas (
    servidor web y motores)
14              cola_motores = xQueueCreate(COLA_MOTORES_LENGTH,
    sizeof(motor_command_t));
15              if (cola_motores == NULL) {
16                  ESP_LOGE("MAIN", "Error al crear la cola de
    motores");
17                  return;
18              }
19
20              \// Iniciar punto de acceso WiFi
21              start_wifi_ap();
22
23              // Tarea del servidor web en core 0 - maneja las
    peticiones HTTP
24              xTaskCreatePinnedToCore(
25                  tarea_start_webserver,
26                  "TareaWeb",
27                  4096,
28                  NULL,
29                  1,
```

```
30                      NULL,
31                      0
32                  );
33
34                  // Tarea de motores en core 1 - controla los
    motores mediante la cola
35                  xTaskCreatePinnedToCore(
36                      tarea_motores,
37                      "TareaMotores",
38                      4096,
39                      NULL,
40                      2,
41                      NULL,
42                      1
43                  );
44              }
45
```

Listing 1: Funcion principal app_main()

```
1               void tarea_motores(void *pvParameters) {
2                   motores_init();
3                   ESP_LOGI(TAG, "Tarea motores iniciada en core %d",
     xPortGetCoreID());
4
5                   motor_command_t cmd = MOTOR_CMD_FRE;
6
7                   // Bucle infinito de la tarea - espera comandos de
     la cola
8                   for (;;) {
9                       if (xQueueReceive(cola_motores, &cmd,
    portMAX_DELAY) == pdTRUE) {
10                          switch (cmd) {
11                              case MOTOR_CMD_AVA:
12                                  motores_ava();
13                                  ESP_LOGI(TAG, "Motor: AVANZAR");
14                                  break;
15                              case MOTOR_CMD_RET:
16                                  motores_ret();
17                                  ESP_LOGI(TAG, "Motor: RETROCEDER")
    ;
18                                  break;
19                              case MOTOR_CMD_IZQ:
20                                  motores_izq();
21                                  ESP_LOGI(TAG, "Motor: IZQUIERDA");
22                                  break;
23                              case MOTOR_CMD_DER:
24                                  motores_der();
25                                  ESP_LOGI(TAG, "Motor: DERECHA");
```

```
                                break;
                            default:
                                motores_fre();
                                ESP_LOGI(TAG, "Motor: DETENER");
                                break;
                        }
                    }
                }
            }

```

Listing 2: Tarea de motores

```
            void tarea_start_webserver(void *pvParameters) {
                start_webserver();
                vTaskDelete(NULL); // Eliminar tarea despues de
    iniciar servidor
            }

```

Listing 3: Tarea del servidor web

6.3. Comunicación entre tareas

La comunicación entre la tarea del servidor web y la tarea de control de motores se realiza a través de una **cola** (`xQueue`), creada al inicio del programa con una capacidad para cinco elementos (Codigo 4).

Cuando el servidor web recibe un comando del usuario (por ejemplo, avanzar, retroceder, girar), coloca en la cola un mensaje de tipo `motor_command_t`. La tarea de motores permanece a la espera mediante la función `xQueueReceive()`, la cual bloquea su ejecución hasta que se reciba un nuevo comando. Una vez recibido, el sistema ejecuta la función correspondiente, como `motores_ava()`, `motores_ret()` o `motores_fre()`.

Este mecanismo desacopla la lógica de comunicación del control físico, permitiendo un sistema más estable, predecible y sencillo de depurar.

```
            #define COLA_MOTORES_LENGTH 5

```

Listing 4: Cola creada con capacidad para cinco elementos

6.4. Control de motores y señal PWM

Las funciones definidas en el módulo `motores.c` (Codigo 5) son las encargadas de generar las señales PWM (Pulse Width Modulation) que controlan los transis-

tores del puente H. Cada función de movimiento (`motores_ava()`, `motores_ret()`, `motores_izq()`, `motores_der()`) modifica el sentido de giro de los motores y el ciclo útil de las señales PWM.

Este método de control permite ajustar la velocidad y dirección del robot de forma precisa, aprovechando las capacidades del ESP32 para generar múltiples canales PWM simultáneamente.

```
1            // Funcion para detener ambos motores
2            void motores_fre(void) {
3                gpio_set_level(IN1, 0);
4                gpio_set_level(IN2, 0);
5                gpio_set_level(IN3, 0);
6                gpio_set_level(IN4, 0);
7                ledc_set_duty(LEDC_LOW_SPEED_MODE, PWM_CHANNEL_A,
    0);
8                ledc_update_duty(LEDC_LOW_SPEED_MODE,
    PWM_CHANNEL_A);
9                ledc_set_duty(LEDC_LOW_SPEED_MODE, PWM_CHANNEL_B,
    0);
10               ledc_update_duty(LEDC_LOW_SPEED_MODE,
    PWM_CHANNEL_B);
11               motor_estado = MOTOR_FRE;
12           }
13
14           // Funcion para avanzar (ambos motores hacia adelante)
15           void motores_ava(void) {
16               gpio_set_level(IN1, 1); gpio_set_level(IN2, 0);
17               gpio_set_level(IN3, 1); gpio_set_level(IN4, 0);
18               ledc_set_duty(LEDC_LOW_SPEED_MODE, PWM_CHANNEL_A,
    PWM_DUTY);
19               ledc_update_duty(LEDC_LOW_SPEED_MODE,
    PWM_CHANNEL_A);
20               ledc_set_duty(LEDC_LOW_SPEED_MODE, PWM_CHANNEL_B,
    PWM_DUTY);
21               ledc_update_duty(LEDC_LOW_SPEED_MODE,
    PWM_CHANNEL_B);
22               motor_estado = MOTOR_AVA;
23           }
24
25           // Funcion para retroceder (ambos motores hacia atras)
26           void motores_ret(void) {
27               gpio_set_level(IN1, 0); gpio_set_level(IN2, 1);
28               gpio_set_level(IN3, 0); gpio_set_level(IN4, 1);
29               ledc_set_duty(LEDC_LOW_SPEED_MODE, PWM_CHANNEL_A,
    PWM_DUTY);
30               ledc_update_duty(LEDC_LOW_SPEED_MODE,
    PWM_CHANNEL_A);
```

```
31                  ledc_set_duty(LEDC_LOW_SPEED_MODE, PWM_CHANNEL_B,
    PWM_DUTY);
32                  ledc_update_duty(LEDC_LOW_SPEED_MODE,
    PWM_CHANNEL_B);
33                  motor_estado = MOTOR_RET;
34              }
35
36              // Funcion para girar a la izquierda
37              void motores_izq(void) {
38                  gpio_set_level(IN1, 0); gpio_set_level(IN2, 1);
39                  gpio_set_level(IN3, 1); gpio_set_level(IN4, 0);
40                  ledc_set_duty(LEDC_LOW_SPEED_MODE, PWM_CHANNEL_A,
    PWM_DUTY);
41                  ledc_update_duty(LEDC_LOW_SPEED_MODE,
    PWM_CHANNEL_A);
42                  ledc_set_duty(LEDC_LOW_SPEED_MODE, PWM_CHANNEL_B,
    PWM_DUTY);
43                  ledc_update_duty(LEDC_LOW_SPEED_MODE,
    PWM_CHANNEL_B);
44                  motor_estado = MOTOR_IZQ;
45              }
46
47              // Funcion para girar a la derecha
48              void motores_der(void) {
49                  gpio_set_level(IN1, 1); gpio_set_level(IN2, 0);
50                  gpio_set_level(IN3, 0); gpio_set_level(IN4, 1);
51                  ledc_set_duty(LEDC_LOW_SPEED_MODE, PWM_CHANNEL_A,
    PWM_DUTY);
52                  ledc_update_duty(LEDC_LOW_SPEED_MODE,
    PWM_CHANNEL_A);
53                  ledc_set_duty(LEDC_LOW_SPEED_MODE, PWM_CHANNEL_B,
    PWM_DUTY);
54                  ledc_update_duty(LEDC_LOW_SPEED_MODE,
    PWM_CHANNEL_B);
55                  motor_estado = MOTOR_DER;
56              }
57
```

Listing 5: Funciones para los motores

6.5. Integración con el servidor web

El módulo `servidor_web.c` traduce las solicitudes HTTP en comandos interpretables por el sistema de control. Cada vez que el usuario presiona un botón en la interfaz web, el ESP32 recibe una petición HTTP (por ejemplo, `/avanzar` o `/detener`), la cual es procesada y enviada a través de la cola de comunicación hacia la tarea de motores.

De este modo, las interacciones del usuario en la interfaz gráfica se convierten en acciones físicas sobre el robot, integrando eficazmente la capa de red con el control embebido.

7. Conclusión

El desarrollo del robot controlado por WiFi permitió integrar conceptos fundamentales de sistemas embebidos, redes inalámbricas, electrónica de potencia y programación concurrente. Los resultados obtenidos evidencian el correcto funcionamiento del sistema, tanto en la comunicación inalámbrica como en el control de los motores y la respuesta general del robot frente a las órdenes del usuario.

La utilización del microcontrolador **ESP32** resultó adecuada debido a su doble núcleo de procesamiento, conectividad WiFi integrada y soporte nativo de **FreeRTOS**, lo cual facilitó la implementación de un entorno multitarea robusto. La separación de responsabilidades entre tareas —una dedicada a la gestión del servidor web y otra al control de motores— permitió un funcionamiento estable sin bloqueos ni retardos perceptibles.

Asimismo, la implementación de una interfaz web intuitiva y multiplataforma simplificó la interacción con el sistema, eliminando la necesidad de aplicaciones externas o configuraciones complejas. El uso del protocolo **HTTP** proporcionó una comunicación estándar, confiable y fácilmente extensible.

En conclusión, el proyecto cumplió con los objetivos planteados, demostrando la viabilidad de controlar un sistema físico mediante una red WiFi local. A futuro, se propone incorporar funciones adicionales como sensores de proximidad, control autónomo, y monitoreo en tiempo real mediante protocolos IoT, ampliando las capacidades del sistema y su aplicabilidad en entornos académicos e industriales.

8. Bibliografía

[1] *Proyecto*. Repositorio (2025). Disponible en: `https://github.com/Cortesse/proyectoFinalTD3`

[2] Esp. Ing. Fernando E. Daniele (2025). *Sistemas Operativos - Técnicas Digitales III*. Archivo PDF. Disponible en: `https://gitlab.com/fernandodaniele/tecnicasdigitalesiii/-/blob/main/Apuntes%20y%20diapositivas/Gu%C3%ADa%20de%20teor%C3%ADa%20TD3%202025%201-%20Sistemas%20operativos.pdf?ref_type=heads`

[3] Esp. Ing. Fernando E. Daniele (2025). *Redes y Protocolos - Técnicas Digitales III*. Archivo PDF. Disponible en: `https://gitlab.com/fernandodaniele/tecnicasdigitalesiii/-/blob/main/Apuntes%20y%20diapositivas/Gu%C3%ADa%20de%20teor%C3%ADa%20TD3%202025%202-%20RYP.pdf?ref_type=heads`

[4] *Pinout y detalles del hardware del ESP32*. Publicación. Disponible en: `https://www.luisllamas.es/esp32-detalles-hardware-pinout/`

[5] *ESP32 con motor de CC y controlador de motor L289N usando ESP-IDF*. Publicación (Septiembre 2022). Disponible en: `https://esp32tutorials-com.translate.goog/esp32-dc-motor-l289n-esp-idf/?_x_tr_sl=en&_x_tr_tl=es&_x_tr_hl=es&_x_tr_pto=tc`