Instituto Tecnológico y de Estudios Superiores de Monterrey

TE3001B Fundamentación de Robótica (Gpo 101)

Reto semanal 3. Manchester Robotics

Autores:

Mariam Landa Bautista // A01736672

Bruno Manuel Zamora García // A01798275

Elías Guerra Pensado // A01737354

Profesor:

Rigoberto Cerino Jimenez

Jueves 06 de Marzo de 2025 Semestre (6) Feb-Jul 2025

Campus Puebla

RESUMEN

El Challenge 3 consiste en la implementación de Micro-ROS en una tarjeta de desarrollo ESP32 para regular la velocidad de un motor mediante la publicación de un valor (que puede variar entre -1 y 1) en un tópico, dicho valor actuará como un PWM. Además, se busca implementar tópicos que transmitan datos relacionados con el valor entregado por un potenciómetro.

OBJETIVOS

El objetivo principal consiste en implementar micro-ROS en una tarjeta de desarrollo ESP32 para controlar un motor con encoder al publicar a través de un tópico en terminal, para ello es necesario:

- ★ Configurar y establecer correctamente la comunicación entre la tarjeta de desarrollo ESP32 y el motor con encoder, utilizando módulos PWM.
- ★ Integrar la librería adecuada de micro-ROS en el entorno de desarrollo de Ubuntu.
- ★ Probar el código de Manchester Robotics necesario para la adquisición de datos que un usuario proporcione y su transmisión a través de tópicos de ROS.
- ★ Implementar nodos de control para recibir los datos y enviar las lecturas a través de tópicos.

INTRODUCCIÓN

El uso de tecnologías como Micro-ROS, junto con módulos ADC y PWM, ha abierto nuevas posibilidades en el desarrollo de sistemas robóticos distribuidos y conectados. Micro-ROS es un framework diseñado para dispositivos con recursos computacionales limitados, como microcontroladores de 32 bits, permitiendo la integración eficiente de componentes en entornos distribuidos para ROS. Los módulos ADC permiten a microcontroladores como el ESP32 convertir señales analógicas en digitales, mientras que los módulos PWM ofrecen un control preciso sobre motores y otros actuadores.

Gracias a su estrecha integración con los microcontroladores, Micro-ROS les permite actuar como nodos dentro de un sistema robótico basado en ROS. Dado que estos dispositivos tienen restricciones en memoria, procesamiento y consumo energético, Micro-ROS ha sido optimizado para funcionar de manera eficiente en este tipo de entornos, minimizando el uso de recursos.

Un convertidor analógico-digital (ADC) es un dispositivo clave en la electrónica, ya que permite transformar señales analógicas en información digital.

La modulación por ancho de pulso (PWM, por sus siglas en inglés) es una técnica utilizada para regular la cantidad de energía suministrada a distintos dispositivos, como motores, luces LED y servomotores. Su funcionamiento se basa en la generación de pulsos eléctricos de voltaje constante cuya duración varía. Estos pulsos alternan entre dos estados: alto (encendido) y bajo (apagado).

La cantidad de tiempo que la señal permanece en estado alto dentro de un ciclo completo determina la energía entregada al dispositivo. A esto se le conoce como ciclo de trabajo, el cual se expresa en porcentaje y representa la proporción entre el tiempo en estado alto y el tiempo total del ciclo.

SOLUCIÓN DEL PROBLEMA

En el Challenge se solicita específicamente que el microcontrolador (ESP32 o Hackerboard) sea el encargado de recibir un comando de velocidad o PWM desde un tópico de ROS 2, y que en función de ese valor regule la velocidad del motor.

Por esta razón, el código del microcontrolador sólo necesita actuar como Subscriber:

1. El microcontrolador no necesita publicar información (no envía mensajes de estado de vuelta a ROS). Solo recibe el valor deseado (setpoint) para la velocidad/potencia del motor.
2. Al suscribirse a un tópico, el microcontrolador está esperando un mensaje (por ejemplo, un Float32) con la consigna de velocidad (0–1, −1–1, etc.) y luego ajusta el PWM y la dirección del motor de acuerdo a ese valor.
3. Como el reto no solicita explícitamente retroalimentar datos (por ejemplo, la velocidad real medida, posición, etc.) al resto del sistema ROS, no hay necesidad de crear un nodo Publisher en el microcontrolador.
4. El nodo en ROS (en la PC) es quien se encarga de “publicar” el comando de velocidad y el microcontrolador (solo Subscriber) ejecuta la acción de controlar el motor.

En pocas palabras, solo se requiere un Subscriber porque la lógica del reto es que el microcontrolador reciba un comando de velocidad y lo aplique; no se requiere que publique nada.

A continuación se describe, la metodología seguida para cumplir con los objetivos del Mini Challenge 3, que consistía en regular (mediante PWM) la velocidad de un motor de corriente directa utilizando Micro-ROS y un puente H. Se describen los elementos, funciones y la estructura del código, resaltando los aspectos esenciales que debe llevar un nodo en Micro-ROS y la forma en que se implementó el PWM para controlar el motor.

1. **Metodología**

➔ Análisis de los requerimientos del reto

- ◆ Objetivo principal: Desarrollar un sistema que regule la velocidad de un motor DC a partir de un valor de consigna (velocidad deseada) recibido desde ROS2.
- ◆ Restricción: No implementar un controlador complejo (PID u otro), sino un regulador que simplemente ajuste el PWM en función del comando.

➔ Selección de Hardware

- ◆ Microcontrolador: ESP32 o Hackerboard (compatible con Micro-ROS y PWM).
- ◆ Driver de motor: Puente H que reciba dos señales de dirección (In1, In2) y una señal PWM para controlar la velocidad.
- ◆ Motor DC: Actuador final cuya velocidad y sentido se regularán.
- ◆ Uso de las mismas conexiones y pines proporcionados por Manchester Robotics.

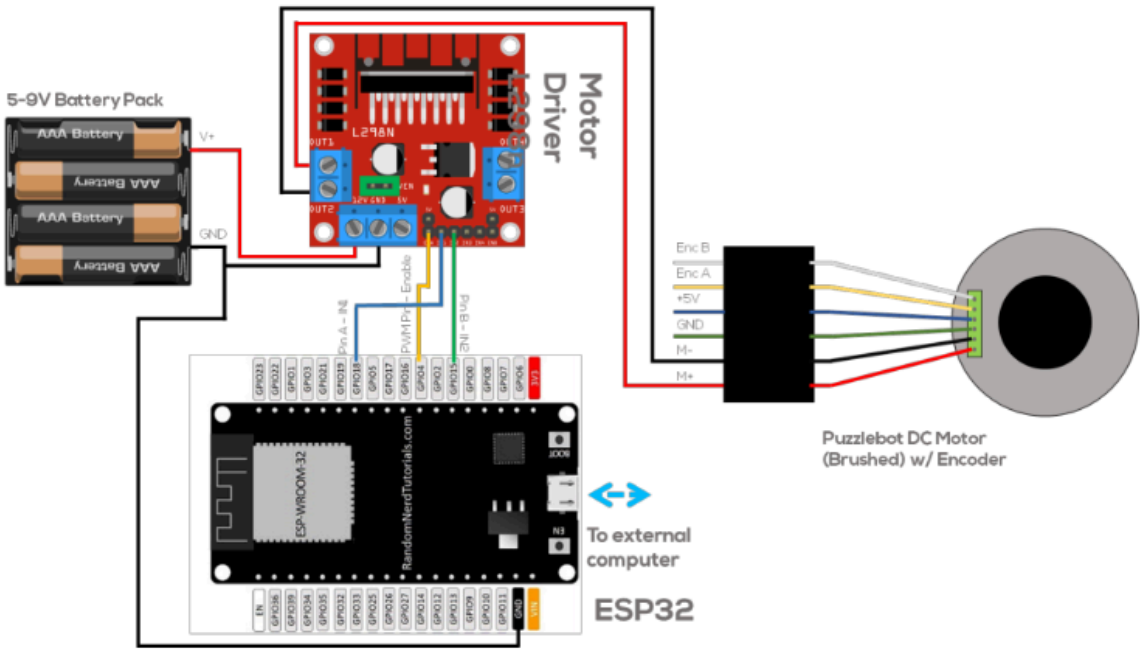


Figura 1. Diagrama de conexión

➔ Diseño de la arquitectura ROS 2 / Micro-ROS

- ◆ Nodo en el microcontrolador: Se crea un nodo llamado, por ejemplo, "motor".

- Suscripción a tópico: El nodo se suscribe a /cmd_pwm, donde se publican valores de tipo std_msgs/msg/Float32.
- Generación de PWM y dirección: A partir del valor recibido, se configura el duty cycle y la dirección (In1, In2) del motor.

➔ Implementación y pruebas

- Programación: Se escribe el código con la estructura típica de Micro-ROS (setup, callback, executor...), que el socio formador nos presentó.
- Flasheo: Se sube el código al ESP32.
- Ejecución: En la PC se inicia el micro_ros_agent y se prueban diversos comandos con ros2 topic pub /cmd_pwm ....
- Verificación: Se observa el cambio de velocidad y sentido del motor para diferentes valores publicados.

2. Descripción de los elementos y funciones en el código

Para que el código cumpla con la especificación de Micro-ROS y controle el PWM del motor, se deben incluir y configurar adecuadamente las siguientes partes:

➔ Librerías y dependencias

- #include <micro_ros_arduino.h>
- #include <rcl/rcl.h>
- #include <rclc/rclc.h>
- #include <rclc/executor.h>
- #include <std_msgs/msg/float32.h>

Estas librerías proporcionan las funciones necesarias para crear y manejar nodos, suscripciones y callbacks dentro del entorno de Micro-ROS.

➔ Definiciones de pines y constantes

- Pines:
  - MOTOR_PWM_PIN (4) → Señal PWM para el driver.
  - MOTOR_IN1_PIN (18) y MOTOR_IN2_PIN (15) → Señales de dirección para el puente H.

- Constantes de PWM::
  - Frecuencia (PWM_FREQ, 980 Hz).
  - Resolución (PWM_RESOLUTION, 8 bits).
  - Canal (PWM_CHANNEL, 0).
  - Rango de valores para el comando recibido (p. ej. -1.0 a 1.0 o 0.0 a 1.0).

➔ Variables y objetos de Micro-ROS

- rcl_node_t node; → Nodo que se ejecuta en el microcontrolador.
- rcl_subscription_t subscriber; → Suscriptor que escuchará el tópico /cmd_pwm.
- rclc_executor_t executor; → Administrador de callbacks (ejecuta las funciones de suscripción).
- std_msgs__msg__Float32 msg; → Estructura para almacenar el valor Float32 recibido.

➔ Estructura principal del código

- setup()
  - Llama a set_microros_transports() para inicializar la comunicación con el agente de ROS 2 (vía serial).
  - Configura los pines como OUTPUT (PWM, In1, In2).
  - Inicializa el canal PWM (frecuencia, resolución y canal) con ledcSetup(...) y lo asocia al pin PWM con ledcAttachPin(...).
  - Crea el soporte (rclc_support_init), el nodo (rclc_node_init_default) y la suscripción (rclc_subscription_init_default).
  - Inicializa el executor (rclc_executor_init) y registra la suscripción (rclc_executor_add_subscription).
- setup()subscription_callback(const void* msg_in)
  - Convierte el mensaje entrante a tipo std_msgs__msg__Float32.
  - Obtiene el valor (ej. cmd_value) y lo limita al rango permitido.
  - Dirección: Si cmd_value es positivo, se pone In1=HIGH, In2=LOW (giro hacia adelante). Si es negativo, In1=LOW, In2=HIGH (giro inverso). Si es cero, ambos en LOW (motor detenido).

- PWM: Se calcula el duty cycle como |cmd_value| * 255 (para 8 bits). Se escribe con ledcWrite(PWM_CHANNEL, duty).

◆ loop()

- Llama periódicamente a rclc_executor_spin_some(...) para procesar las callbacks y manejar los eventos de suscripción sin bloquear el flujo principal.
- Usa un delay() pequeño para no saturar el loop.

➔ Control de errores

◆ Se incluye una función error_loop() que hace parpadear un LED si ocurre un error en las llamadas a funciones Micro-ROS (por ejemplo, si rclc_support_init falla).

3. Seguimiento paso a paso del código de programación

➔ Declaraciones globales

◆ Se importan las librerías de Micro-ROS, se definen pines y constantes.

◆ Se declaran las variables node, executor, subscriber, etc.

➔ Función setup()

◆ Paso 1: set_microros_transports() → habilita la comunicación con el agente de ROS 2.

◆ Paso 2: Configuración de pines: pinMode(MOTOR_IN1_PIN, OUTPUT), pinMode(MOTOR_IN2_PIN, OUTPUT), etc.

◆ Paso 3: Ajuste de PWM con ledcSetup(PWM_CHANNEL, PWM_FREQ, PWM_RESOLUTION) y ledcAttachPin(MOTOR_PWM_PIN, PWM_CHANNEL).

◆ Paso 4: Inicialización de Micro-ROS

- rclc_support_init(...) para configurar el "contexto" de Micro-ROS.
- rclc_node_init_default(...) para crear el nodo.
- rclc_subscription_init_default(...) para suscribirse a /cmd_pwm.

- rclc_executor_init(...) y rclc_executor_add_subscription(...) para manejar la suscripción.

➔ Callback de suscripción

- ◆ Se ejecuta automáticamente cuando un nuevo mensaje llega a /cmd_pwm.
- ◆ Convierte el valor Float32 recibido y lo restringe a un rango ([−1,1]).
- ◆ Ajusta la dirección del motor con In1/In2.
- ◆ Calcula el duty cycle como el valor absoluto multiplicado por 255.
- ◆ Llama a ledcWrite(PWM_CHANNEL, duty) para actualizar la salida PWM.

➔ Función loop()

- ◆ Llama a rclc_executor_spin_some(...) para procesar los eventos ROS 2 sin bloquear el programa.
- ◆ Un pequeño delay(50) o delay(100) evita un loop excesivamente rápido.

4. Partes más importantes para el PWM con el motor

➔ Configuración de la frecuencia y resolución

- ◆ ledcSetup(PWM_CHANNEL, PWM_FREQ, PWM_RESOLUTION);
- ◆ Para el reto, se nos recomienda 980 Hz y 8 bits de resolución.

➔ Asociación del pin al canal

- ◆ ledcAttachPin(MOTOR_PWM_PIN, PWM_CHANNEL);
- ◆ Sin esto, no se genera PWM real en el pin.

➔ Escritura del duty cycle

- ◆ ledcWrite(PWM_CHANNEL, duty);
- ◆ El valor de duty varía entre 0 (0 %) y 255 (100 %).

➔ Control de dirección

- ◆ Para giro adelante: digitalWrite(IN1, HIGH); digitalWrite(IN2, LOW);
- ◆ Para giro atrás: digitalWrite(IN1, LOW); digitalWrite(IN2, HIGH);
- ◆ Para detener: digitalWrite(IN1, LOW); digitalWrite(IN2, LOW);

Todo el texto en verde, son variables o funciones empleadas en arduino.

5. Código a implementar

```
#include <micro_ros_arduino.h>
#include <rcl/rcl.h>
#include <rcl/error_handling.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>
#include <std_msgs/msg/float32.h>
#include <Arduino.h>

/*************************************************************************
* Definición de los pines usados para el puente H y el PWM.              *
* Ajustar estos valores según el hardware (Hackerboard o ESP32).         *
*************************************************************************/
#define MOTOR_PWM_PIN  4    // Pin físico que se conecta a la entrada PWM del driver de
motor (GPIO 04)
#define MOTOR_IN1_PIN  18   // Pin físico que se conecta a la entrada IN1 del driver (GPIO
18)
#define MOTOR_IN2_PIN  15   // Pin físico que se conecta a la entrada IN2 del driver (GPIO
15)

// Pin que usaremos para parpadear en caso de error (LED integrado o externo).
#define ERROR_LED_PIN  2

/*************************************************************************
* Configuración del PWM                                                  *
* - Frecuencia en Hz (aquí 980 Hz).                                      *
* - Resolución en bits (aquí 8 bits -> valores de 0 a 255).              *
* - Canal de PWM que vamos a usar (0, pero puede haber hasta 16).        *
*************************************************************************/
#define PWM_FREQ       980
#define PWM_RESOLUTION 8
#define PWM_CHANNEL    0

/*************************************************************************
* Definimos el rango del mensaje que vamos a recibir en /cmd_pwm.        *
* Aquí, -1.0f indica giro inverso al 100% y 1.0f giro directo al 100%.   *
* El valor 0.0f detiene el motor.                                        *
*************************************************************************/
#define MIN_CMD -1.0f
#define MAX_CMD  1.0f

/*************************************************************************
* Declaración de objetos y variables propias de micro-ROS                *
* - node              : Representa el nodo en ROS 2.                     *
* - executor          : Maneja la ejecución de callbacks (suscripciones). *
* - support           : Estructura de soporte para inicializar micro-ROS. *
* - allocator         : Administrador de memoria.                        *
* - subscriber        : Estructura de suscripción al tópico.             *
* - msg               : Mensaje (Float32) que recibimos.                 *
*************************************************************************/
rcl_node_t node;
rclc_executor_t executor;
rclc_support_t support;
```

```
rcl_allocator_t allocator;
rcl_subscription_t subscriber;
std_msgs__msg__Float32 msg;

/*****************************************************************************
* Macros para simplificar la verificación de errores en llamadas rcl.      *
* - RCCHECK(fn)    : Si falla la función fn, llama a error_loop().          *
* - RCSOFTCHECK(fn): Hace la llamada fn, pero ignora el error si ocurre.    *
*****************************************************************************/
#define RCCHECK(fn)  { rcl_ret_t temp_rc = fn; if ((temp_rc != RCL_RET_OK)) { error_loop();
} }
#define RCSOFTCHECK(fn) { rcl_ret_t temp_rc = fn; if ((temp_rc != RCL_RET_OK)) {} }

/*****************************************************************************
* Función de error: se llama cuando alguna inicialización o llamada falla. *
* Parpadea el LED definido en ERROR_LED_PIN indefinidamente.               *
*****************************************************************************/
void error_loop() {
 pinMode(ERROR_LED_PIN, OUTPUT);
 while (1) {
   digitalWrite(ERROR_LED_PIN, !digitalRead(ERROR_LED_PIN));
   delay(100);
 }
}

/*****************************************************************************
* Callback que se llama al recibir un nuevo mensaje en el tópico /cmd_pwm. *
*                                                                           *
* El mensaje es de tipo Float32 y se asume que viene en el rango [-1,1].   *
* - Si el valor es positivo, el motor gira hacia adelante.                  *
* - Si el valor es negativo, el motor gira hacia atrás.                     *
* - Si el valor es cero, se detiene el motor.                               *
*                                                                           *
* El valor absoluto del mensaje define el nivel de PWM (0% a 100%).        *
*****************************************************************************/
void subscription_callback(const void *msg_in)
{
 // Convierto el puntero genérico msg_in al tipo std_msgs__msg__Float32
 const std_msgs__msg__Float32 *rcv_msg = (const std_msgs__msg__Float32 *)msg_in;
 float cmd_value = rcv_msg->data;  // Guardo el valor recibido

 // Aseguro que cmd_value no exceda los límites [-1,1]
 if (cmd_value > MAX_CMD) cmd_value = MAX_CMD;
 if (cmd_value < MIN_CMD) cmd_value = MIN_CMD;

 // Lógica para decidir la dirección según el signo de cmd_value
 if (cmd_value > 0.0f) {
   // Giro hacia adelante: IN1 = HIGH, IN2 = LOW
   digitalWrite(MOTOR_IN1_PIN, HIGH);
   digitalWrite(MOTOR_IN2_PIN, LOW);
 } else if (cmd_value < 0.0f) {
   // Giro hacia atrás: IN1 = LOW, IN2 = HIGH
   digitalWrite(MOTOR_IN1_PIN, LOW);
   digitalWrite(MOTOR_IN2_PIN, HIGH);
 } else {
```

```
  // cmd_value == 0 -> Detener motor: IN1 = LOW, IN2 = LOW
  digitalWrite(MOTOR_IN1_PIN, LOW);
  digitalWrite(MOTOR_IN2_PIN, LOW);
 }

 // Obtengo el valor absoluto de cmd_value para calcular el duty cycle
 float duty_norm = fabs(cmd_value);          // duty_norm va de 0 a 1
 uint8_t duty = (uint8_t)(duty_norm * 255.0f); // Convierto [0,1] a [0,255]

 // Aplico el duty cycle al canal PWM (PWM_CHANNEL)
 ledcWrite(PWM_CHANNEL, duty);
}

/*****************************************************************************
* setup(): Se ejecuta una vez al inicio.                                     *
*  1. Configura la comunicación de micro-ROS (ej. vía Serial).               *
*  2. Configura pines de motor como salida y PWM.                            *
*  3. Inicializa Micro-ROS (allocator, support, node, etc.).                 *
*  4. Crea y configura la suscripción al tópico "/cmd_pwm".                  *
*  5. Inicia el executor para procesar callbacks.                            *
*****************************************************************************/
void setup() {
 // Inicializa la comunicación con el agente micro-ROS
 set_microros_transports();

 // Configuro los pines del motor como salidas digitales
 pinMode(MOTOR_IN1_PIN, OUTPUT);
 pinMode(MOTOR_IN2_PIN, OUTPUT);
 pinMode(MOTOR_PWM_PIN, OUTPUT);

 // Configuro el canal PWM en la frecuencia y resolución deseadas
 ledcSetup(PWM_CHANNEL, PWM_FREQ, PWM_RESOLUTION);
 // Asigno el pin físico (MOTOR_PWM_PIN) al canal de PWM
 ledcAttachPin(MOTOR_PWM_PIN, PWM_CHANNEL);

 // Espero 2 segundos para asegurar que el agente se conecte correctamente
 delay(2000);

 // Inicializo el allocator por defecto
 allocator = rcl_get_default_allocator();

 // Inicializo la estructura de soporte de micro-ROS
 // (maneja el contexto y la comunicación con el agente)
 RCCHECK(rclc_support_init(&support, 0, NULL, &allocator));

 // Creo el nodo de micro-ROS con nombre "motor"
 RCCHECK(rclc_node_init_default(&node, "motor", "", &support));

 // Creo la suscripción al tópico "/cmd_pwm" de tipo Float32
 RCCHECK(rclc_subscription_init_default(
   &subscriber,
   &node,
   ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Float32),
   "/cmd_pwm"
 ));
```

```
 // Inicializo el executor con capacidad para manejar 1 'handle' (la suscripción)
 RCCHECK(rclc_executor_init(&executor, &support.context, 1, &allocator));

 // Agrego la suscripción al executor y le indico la función callback
 RCCHECK(rclc_executor_add_subscription(
   &executor,
   &subscriber,
   &msg,
   &subscription_callback,
   ON_NEW_DATA
 ));
}

/*******************************************************************************
* loop(): Se ejecuta en bucle infinito.                                        *
*  - Aquí se llama a rclc_executor_spin_some() para procesar callbacks         *
*    pendientes (cuando llegan mensajes nuevos).                               *
*  - Se usa un delay(50) para no saturar la CPU con spin en cada ciclo.        *
*******************************************************************************/
void loop() {
 // Procesa las llamadas pendientes del executor (mensajes, etc.)
 RCSOFTCHECK(rclc_executor_spin_some(&executor, RCL_MS_TO_NS(100)));
 // Pausa breve para evitar un bucle muy rápido
 delay(50);
}
```

*Justificacion de no usar **rcl_timer_t** en el código:*

La razón principal es que, en este challenge, no necesitamos ejecutar ninguna tarea de forma periódica. El programa se basa en reaccionar a los mensajes que llegan por el tópico /cmd_pwm. Cada vez que se recibe un nuevo valor (suscripción), el callback (subscription_callback) ajusta el PWM y la dirección del motor en ese momento.

En Micro-ROS , el uso de un timer (rcl_timer_t) se vuelve útil cuando queremos ejecutar algo de manera periódica, por ejemplo:

- Publicar un mensaje cada cierto intervalo (como un sensor que envía datos cada 100 ms).

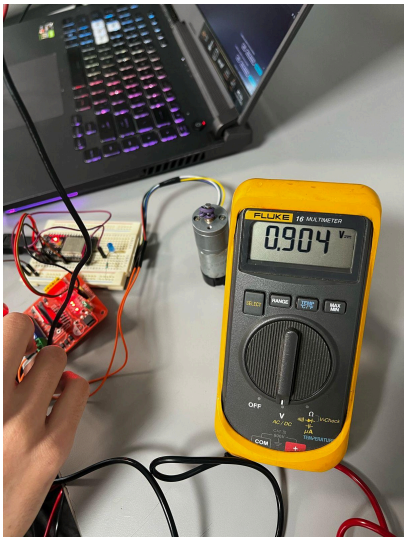
Dado que aquí todo se desencadena por la llegada de datos (suscripción) que el usuario decide cuando hacerlo y no hay necesidad de hacer nada periódico, no es necesario crear ni registrar un timer. Por eso el código no incluye estas líneas:

```
rcl_timer_t timer;
```

```
rclc_timer_init_default(&timer, ...);
rclc_executor_add_timer(...);
```

En nuestra implementación, determinamos que una batería de 9V no sería una opción viable, ya que no proporciona la corriente suficiente para el correcto funcionamiento del motor. Por esta razón, optamos por utilizar una fuente de voltaje a 12V, la cual no solo suministra el voltaje máximo de operación del motor, sino que también garantiza la corriente necesaria para su adecuado desempeño.

Una vez establecida la fuente de alimentación, nos enfocamos en medir dos parámetros fundamentales para la correcta configuración de nuestro código: el voltaje de zona muerta y el voltaje de zona de saturación. Estos valores nos permiten determinar el voltaje mínimo a partir del cual el motor comienza a girar y el voltaje máximo en el que la velocidad deja de aumentar, respectivamente. Con estos datos, podemos optimizar el control de velocidad y mejorar la eficiencia de nuestra implementación.



Zona muerta: Área de valores de entrada donde el sistema no responde porque los valores son demasiado pequeños para reaccionar.
Este valor lo obtuvimos conectando el motor directamente a la fuente de alimentación, y variamos el voltaje disminuyendo poco a poco hasta que encontramos el valor mínimo en el que se empezaba a mover, en nuestro caso fue que el voltaje mínimo de funcionamiento es 0.9V.
Consultar la carpeta de videos, ver el llamado "demostracion_led.MOV"

Zona de saturación: Está zona de saturación es la región donde, al aumentar el voltaje, la velocidad del motor ya no aumenta significativamente. Es decir, el motor ha alcanzado su límite físico. En nuestro caso es de 11.5 V aproximadamente, esto según las datasheets del motor.

RESULTADOS

Con la fuente de voltaje, aplicamos 12V al circuito (este voltaje fue constante para todas las pruebas):

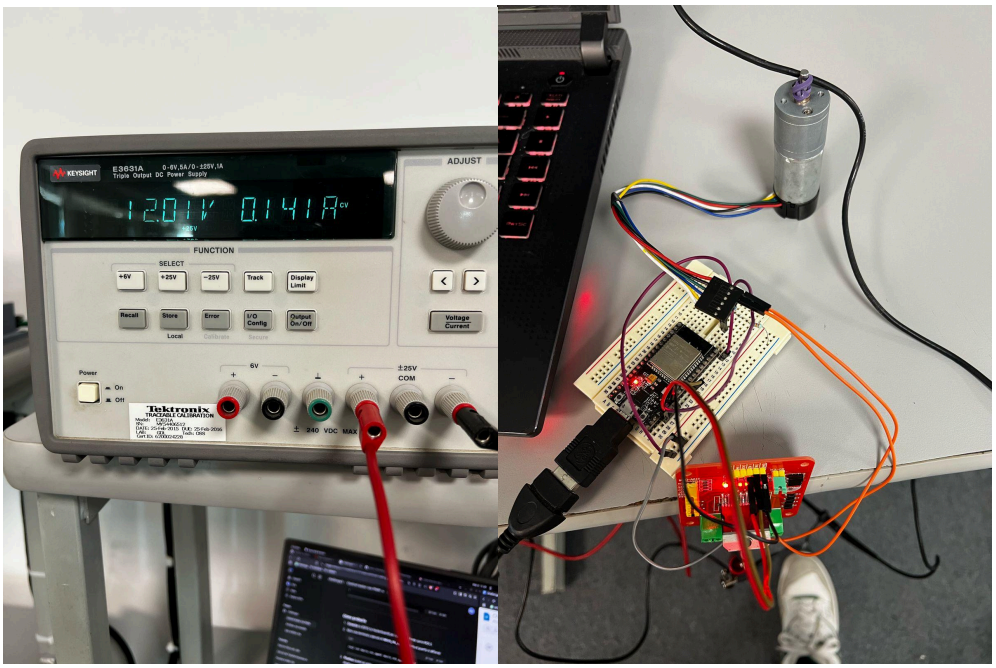


Figura 1: Fuente de voltaje y circuito físico

Para obtener los resultados requeridos, cargamos el archivo a la ESP32, tal como lo hacemos en ARDUINO IDE, seleccionamos la tarjeta y el puerto correspondiente, para esto tomamos en cuenta lo siguiente:

- Instalar las librerías correspondientes y además instalar los paquetes correspondientes para hacer uso de la ESP32.
- Para no presentar problemas, tenemos que darle permiso al puerto asignado para la práctica, para ello se requiere abrir una nueva terminal.

  cd/ dev

  *sudo cdmod 666 /dev/ttyUSB**

Con el primer comando, podemos encontrar el puerto designado de ESP32 y posteriormente, usando el comando 2 nos aseguramos de darle permiso al puerto correspondiente, terminado esto procedemos a cargar el archivo .ino.

Una vez cargado el programa, si por algún motivo existe un error, el LED que hemos conectado se encenderá. Para darle solución ejecutamos lo siguiente:

ros2 run micro_ros_agent micro_ros_agent serial --dev /dev/ttyUSB0

El caso de cuando hay un error se ve de la siguiente manera: Consultar la carpeta de videos, ver el llamado “demostracion_led.MOV”

Ya que se ha ejecutado el comando, se presiona el botón de reset de la ESP32, para eliminar los errores y finalmente podemos trabajar con los nodos creados.

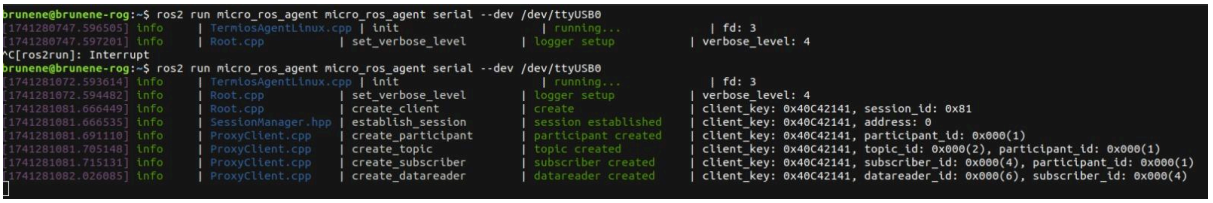


Figura 2. Salida en terminal tras configuración ESP32.

Posteriormente abrimos una nueva terminal, solicitando la lista de topicos con el siguiente comando:

ros2 topic list

Figura 3. Lista de tópicos y nodos.

Como podemos observar en la figura 3, contamos con el nodo cmd_pwm:

- /cmd_pwm: Representa un tópico, donde el nodo pública mensajes y otros nodos pueden suscribirse para recibirlos. Transmite PWM, que son las señales para controlar la velocidad de un motor.
- /motor: Representa un nodo que recibe los mensajes de cmd_pwm, interpreta los comandos PWM y ejecuta el movimiento en el motor.

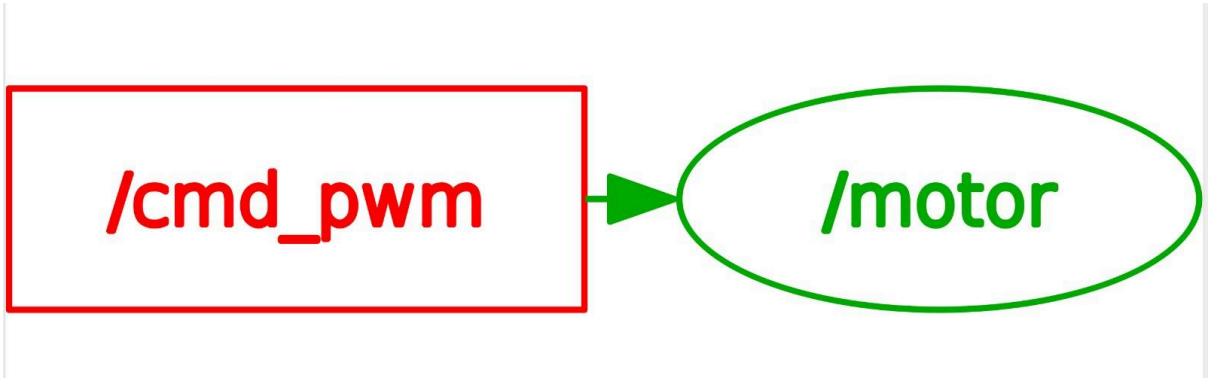


Figura 4: Conexión de nodos.

Ahora bien, una vez que hicimos todo lo anterior tenemos que correr el comando:

ros2 topic pub /cmd_pwm std_msgs/msg/Float32 "data: 0.5"

La terminal nos mostrará lo siguiente (por defecto 'data' corre en 0.5):

```
brunene@brunene-rog:~$ ros2 topic pub /cmd_pwm std_msgs/msg/Float32 "data: 0.5"
publisher: beginning loop
publishing #1: std_msgs.msg.Float32(data=0.5)

publishing #2: std_msgs.msg.Float32(data=0.5)

publishing #3: std_msgs.msg.Float32(data=0.5)

publishing #4: std_msgs.msg.Float32(data=0.5)

publishing #5: std_msgs.msg.Float32(data=0.5)

publishing #6: std_msgs.msg.Float32(data=0.5)
```

Figura 5: Ejecución de PWM

Cuando queramos cambiar el PWM, solo tenemos que cambiar el valor de "data" por un valor entre -1 y 1.

Finalmente se muestra mediante videos como funciona el sistema completo, empezando por la zona de muerta, el video llamado "Demostracion_voltaje_minimo.MOV", mostramos como el funcinamieno del motor en su punto más lento (zona muerta) es cuando el valor de "data" es 0.23 es cuando el motor recibe 0.9V, en teoria tendria que darme 2.76V, pero se pierde voltaje entre los cables y el puente H

```
brunene@brunene-rog:~$ ros2 topic pub /cmd_pwm std_msgs/msg/Float32 "data: 0.23
publisher: beginning loop
publishing #1: std_msgs.msg.Float32(data=0.23)

publishing #2: std_msgs.msg.Float32(data=0.23)

publishing #3: std_msgs.msg.Float32(data=0.23)

publishing #4: std_msgs.msg.Float32(data=0.23)

publishing #5: std_msgs.msg.Float32(data=0.23)

publishing #6: std_msgs.msg.Float32(data=0.23)
```

Figura 6: Zona muerta

Para la demostración general, hicimos 4 pruebas (consultar el video "Demostracion_general.MOV"):

1. Se corre con PWM a 0.0, donde el motor no gira nada, está completamente parado por que el duty es de 0%

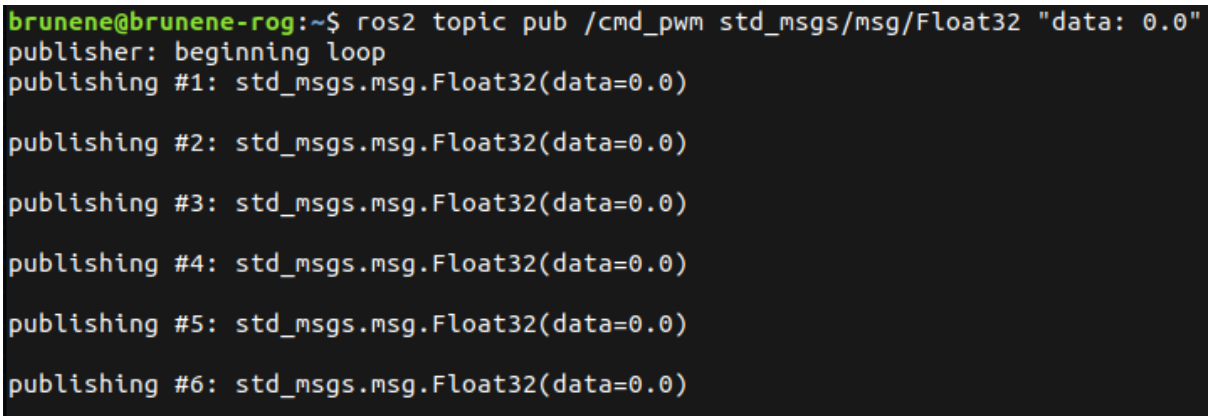


Figura 7: PWM a 0%

2. Se corre con PWM a 0.5, donde el motor gira a su mitad de velocidad, por que el duty esta al 50%

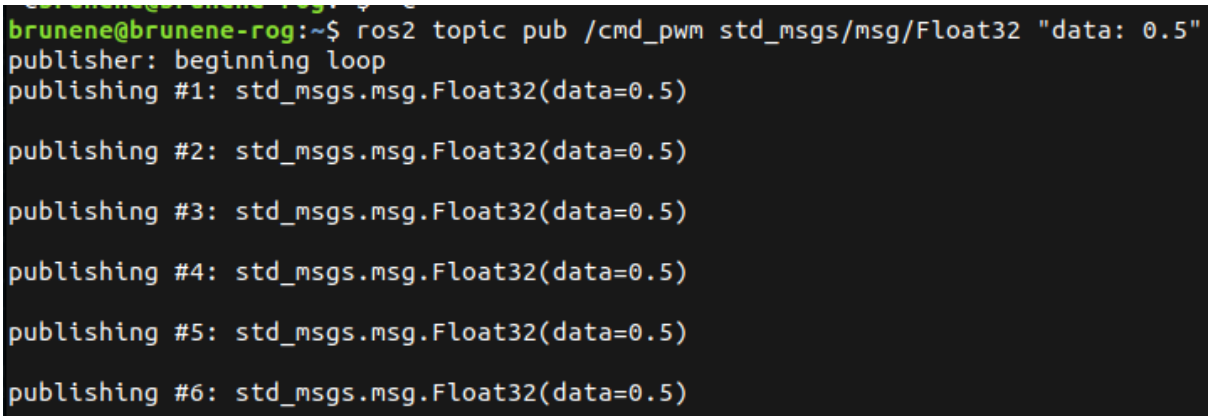


Figura 8: PWM a 50%

3. Se corre con PWM a 1.0, donde el motor gira a su máxima de velocidad, por que el duty esta al 100%

```
brunene@brunene-rog:~$ ros2 topic pub /cmd_pwm std_msgs/msg/Float32 "data: 1.0"
publisher: beginning loop
publishing #1: std_msgs.msg.Float32(data=1.0)

publishing #2: std_msgs.msg.Float32(data=1.0)

publishing #3: std_msgs.msg.Float32(data=1.0)

publishing #4: std_msgs.msg.Float32(data=1.0)
```

Figura 9: PWM a 100%

4. Se corre con PWm a -1.0, donde el motor gira a su máxima velocidad, pero en sentido contrario

```
brunene@brunene-rog:~$ ros2 topic pub /cmd_pwm std_msgs/msg/Float32 "data: -1.0"
publisher: beginning loop
publishing #1: std_msgs.msg.Float32(data=-1.0)

publishing #2: std_msgs.msg.Float32(data=-1.0)

publishing #3: std_msgs.msg.Float32(data=-1.0)

publishing #4: std_msgs.msg.Float32(data=-1.0)
```

Figura 9: PWM a 100% en sentido contrario

Con estos ejemplos se demuestra el funcionamiento que se nos solicitó.

Por último se demuestra como se graficó el pin de PWM en el puente H Y el pin de init1 en el puente H en el video llamado "Graficacion_PWM_INIT1.MOV" , nos muestra como cambia la gráfica de estas dos señales en base a los valores que se le dan a 'data.'

CONCLUSIONES

La implementación de Micro-ROS en una tarjeta de desarrollo ESP32 permitió controlar la velocidad de un motor de corriente continua mediante la recepción de comandos PWM desde un tópico de ROS 2. Se logró una correcta comunicación entre los dispositivos y la integración de los módulos necesarios, cumpliendo con los objetivos planteados. La metodología aplicada demostró la eficiencia de Micro-ROS en sistemas embebidos con recursos limitados, facilitando la regulación del motor sin necesidad de un controlador complejo. Este proyecto representa un paso hacia la automatización y control de motores en sistemas robóticos de bajo costo y alta eficiencia.

Además, la implementación permitió identificar posibles mejoras, como la incorporación de un sistema de retroalimentación para optimizar la precisión del control de velocidad. A futuro, se podría explorar el uso de algoritmos de control más avanzados, como PID, para mejorar la estabilidad del sistema y adaptarlo a aplicaciones más exigentes.

CARPETA VIDEOS

https://drive.google.com/drive/folders/1cQB4PQQieqtPyHZk0tncRezzAG1pMc-F?usp=drive_link

REFERENCIAS

- Solectroshop. (s.f.). *¿Qué es PWM y cómo usarlo?* Solectroshop. https://solectroshop.com/es/blog/que-es-pwm-y-como-usarlo--n38
- r/engineering. (2022, enero 17). *Meaning of deadzone in hydraulic machines?* [Publicación en un foro en línea]. Reddit. https://www.reddit.com/r/engineering/comments/s6zh0m/meaning_of_deadzone_in_hydraulic_machines/