



Tecnológico de Monterrey

Challenge 3

Yestli Darinka Santos Sánchez // A01736992

Emiliano Olguín Ortega // A01737561

Erik García Cruz // A01732440

06 de marzo del 2025

TE3001B Fundamentos de Robótica(Gpo 101)

Juan Manuel Ahuactzin Larios

Rigoberto Cerino Jiménez

Alfredo García Suárez

Resumen

El *Mini Challenge 3* tiene como propósito que los estudiantes revisen los conceptos introducidos en sesiones previas mediante la implementación de varios nodos de ROS para regular la velocidad de un motor de corriente continua (*DC Motor*). Para ello, se emplea un computador externo, un microcontrolador (*ESP32*) y un controlador de motor (*Motor Driver*). Durante el proceso, se exploran el uso de señales PWM en ESP32 y la comunicación entre nodos ROS para el control del motor.

Objetivos

Objetivo general

Desarrollar e implementar un sistema de control de velocidad para un motor DC utilizando ROS, un ESP32 y un controlador de motor.

Objetivos específicos

1. Diseñar e implementar nodos de ROS que permitan la comunicación entre el ESP32 y el computador.
2. Configurar y generar señales PWM en el ESP32 para regular la velocidad del motor DC.
3. Integrar el control de velocidad mediante la combinación de hardware y software.
4. Evaluar el desempeño del sistema asegurando su correcto funcionamiento dentro de los parámetros establecidos.

Introducción

Este reto busca que los estudiantes adquieran experiencia en la implementación de sistemas embebidos con ROS, control de motores y procesamiento de señales PWM. Se centra en la regulación de velocidad de un motor DC mediante una conexión entre un microcontrolador ESP32 y un computador externo, estableciendo una arquitectura basada en ROS.

A continuación, se describen brevemente los principales componentes utilizados en la implementación:

Micro-ROS

Micro-ROS es una extensión de ROS diseñada para sistemas embebidos con microcontroladores de bajo consumo, permitiendo la integración de estos en arquitecturas robóticas más amplias. Facilita la comunicación entre microcontroladores y ROS mediante mensajes y tópicos, reduciendo la carga computacional en sistemas de tiempo real.

Módulos ADC y PWM en ESP32

El ESP32 cuenta con conversores analógico-digitales (*ADC*) y generación de señales PWM.

- **ADC (Analog-to-Digital Converter):** Permite convertir señales analógicas en digitales, facilitando la lectura de sensores.
- **PWM (Pulse Width Modulation):** Técnica de modulación que varía el ciclo de trabajo de una señal digital para simular una salida analógica.

En el ESP32, el uso de PWM se maneja con los siguientes parámetros:

1. **Canales PWM:** Dispone de 16 canales (0-15) configurables.
2. **Frecuencia de señal:** Se recomienda una frecuencia estándar de 980 Hz.
3. **Resolución del ciclo de trabajo:** Puede variar entre 1 y 16 bits (8 bits = 0-255).
4. **Salida GPIO:** Se debe especificar el pin de salida de la señal PWM.

Solución del problema

Explicación del código empleado (código en el anexo)

El ESP32 actúa como un nodo ROS 2, permitiendo la comunicación con un agente ROS 2 en un ordenador host. El código configura el hardware necesario, establece la comunicación con ROS 2 y controla el motor en función de comandos PWM recibidos a través de una suscripción.

Primeramente se incluyeron las bibliotecas necesarias para la implementación de **Micro-ROS** en el ESP32, permitiendo la comunicación con el agente ROS 2. También se incluyen las bibliotecas estándar `stdio.h` y `math.h` para la manipulación de datos.

Se definen los pines del ESP32 que controlarán el motor:

- **ENA (26):** Controla la velocidad del motor mediante PWM.
- **In1 (14) e In2 (27):** Determinan la dirección de giro del motor.
- **LED_PIN (22):** Se usa para indicar errores mediante parpadeo.

Se configuran los parámetros del **PWM**:

- **Frecuencia de 980 Hz**
- **Resolución de 8 bits**
- **Canal 0** para el PWM
- **Valores mínimo y máximo de la señal recibida (-1 a 1)**

Si ocurre un error crítico, el código entra en un bucle infinito de error (`error_loop()`).

Se define un **nodo** de ROS 2 llamado "`motor_robotronicos`".

Se inicializa un **ejecutor**, que maneja las tareas de Micro-ROS.

Se configura un **publicador** para enviar el valor de PWM y un **suscriptor** que recibe comandos PWM.

Cuando se recibe un mensaje en el tópic `"cmd_pwm_robotronicos"`, su valor es **limitado** entre -1 y 1 para evitar valores fuera de rango.

El código determina la **dirección de giro del motor**:

- **pwm_set_point > 0**: Gira en una dirección.
- **pwm_set_point < 0**: Gira en la dirección opuesta.
- **pwm_set_point = 0**: Motor apagado.

El valor absoluto del `pwm_set_point` se escala a un rango entre **0 y 255** para controlar la velocidad del motor mediante **PWM**.

El código publica el valor PWM ajustado en el tópic `"pwm_robotronicos"` para ser monitoreado desde ROS 2.

En el bucle principal cada 100 ms, el ejecutor de Micro-ROS revisa si hay nuevos mensajes en el tópic `"cmd_pwm_robotronicos"` y ejecuta la función `subscription_callback()` si es necesario.

Zona muerta

Para la investigación de muestreo para encontrar la zona donde el motor se comporta como no debería, es decir, no enciende a pesar de que se le mande voltaje por medio en el PWM comenzamos probando con los datos más comunes para este tipo de motores, 0.3 según el rango establecido: -1 mínimo (máxima velocidad para un lado), 0 apagado, 1 máximo valor (máxima velocidad para el lado contrario).

De este punto se desprenden dos variantes:

El motor parte del reposo: al intentar encender el motor mediante con un valor relativamente bajo de PWM como 0.3 no inicia el movimiento, lo mismo sucede para un valor de 0.4 al igual que sus contrapartes negativas, para que logre comenzar el movimiento a partir de un estado de reposo necesita forzosamente un valor de PWM de al menos 0.5 o -0.5. Esto da como resultado:

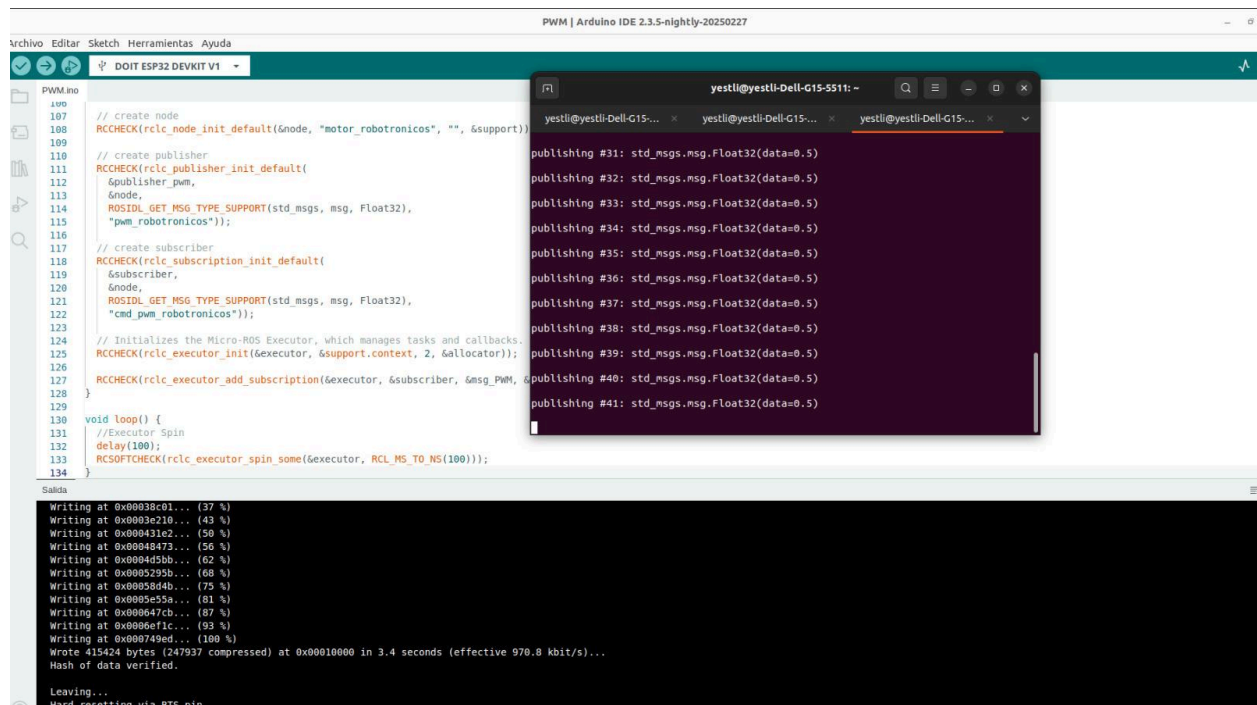
Del reposo : -0.4 a 0.4 zona muerta.

El motor parte del movimiento: al estar el motor girando para algún sentido, en nuestro sistema ya sea negativo o positivo (solo cambia el sentido del giro del motor) no se puede introducir un valor menor a 0.35 de PWM pues éste se detiene al momento y la potencia no es suficiente para que siga en movimiento, es así incluso cuando al

está girando en un sentido, por ejemplo a 0.5 y se cambia a -0.35 el motor todavía logra cambiar el sentido del giro y continuar en movimiento. Si se ingresa un valor de menor a -0.3 o mayor 0.3 el motor se detiene. Esto da como resultado:

Partir del movimiento: -0.35 a 0.35 zona muerta.

Resultados



The screenshot displays the Arduino IDE interface with a sketch named 'PWM.ino' for an ESP32 DevKit V1. The code implements a ROS2 node 'motor_robotronicos' that publishes PWM values. A terminal window in the foreground shows the output of the ROS2 node, displaying a series of 'publishing' messages with 'std_msgs.msg.Float32(data=0.5)' values. The terminal output also shows the progress of the firmware upload to the ESP32, indicating a successful write and verification.

Liga del vídeo del funcionamiento:

https://youtube.com/shorts/8FiR1_2-ZEA?si=6nK1_4Fg3u5wpMFo

En el vídeo se muestra cómo funciona el circuito, cómo mediante el ESP32 se conecta con un script en Arduino con librerías de MicroROS para suscribirse a parámetros mandados por mediante un agente de ROS2 en la terminal donde se le ajusta el PWM al motor para que determine la dirección del giro del mismo.

Conclusión

El desarrollo del Mini Challenge 3 se logró implementar un sistema funcional de control de velocidad para un motor DC mediante la integración de ROS, un ESP32 y un controlador de motor. Se diseñó un programa en Arduino donde se conectó correctamente los nodos de ROS, estableciendo una comunicación eficiente entre el microcontrolador y el computador externo. Asimismo, se configuraron y generaron señales PWM en el ESP32 de manera adecuada, permitiendo regular la velocidad del motor de forma precisa y estable mandando el parámetro del PWM desde la terminal de la computadora con ROS2.

Se implementó las reglas del reto, como la investigación independiente y la optimización de algoritmos, favoreció la comprensión profunda del funcionamiento de ROS y del ESP32 en sistemas de control embebidos.

A pesar del éxito general, se identificaron algunos desafíos menores durante la ejecución, principalmente en la configuración inicial del ESP32 como el hecho de que en dos computadoras del equipo no se pudo establecer la conexión de MicroRos debido a que a algún error que impedía la identificación del puerto; así mismo en la calibración del PWM para lograr una respuesta óptima del motor. Estas dificultades fueron superadas mediante ajustes en la programación y pruebas experimentales. Como posible mejora, se podría optimizar la metodología implementada mediante la incorporación de algoritmos de control más avanzados, como un controlador PID, para mejorar la estabilidad y precisión de la velocidad del motor.

Anexos

Código de Arduino

// Include Libraries to be used

#include <micro_ros_arduino.h> //micro-ros-arduino library

#include <rcl/rcl.h> //Core ROS 2 Client Library (RCL) for node management.

#include <rcl/error_handling.h> //Error handling utilities for Micro-ROS.

#include <rcl/rclc.h> //Micro-ROS Client library for embedded devices.

#include <rcl/executor.h> //Micro-ROS Executor to manage callbacks

#include <std_msgs/msg/float32.h> //Predefined ROS 2 message type for float32

#include <stdio.h> //Standard I/O library for debugging.

#include <math.h>

//Specifies GPIO pin 13 for controlling an LED

#define ENA 26

#define In1 14

#define In2 27

//Declare nodes to be used

rcl_node_t node; //Represents a ROS 2 Node running on the microcontroller.

//Instantiate executor and its support classes

rcl_executor_t executor; //Manages task execution (timers, callbacks, etc.).

rcl_support_t support; //Handles initialization & communication setup.

rcl_allocator_t allocator; //Manages memory allocation.

```
//Declare Publishers to be used
```

```
rcl_publisher_t publisher_pwm; //Declares a ROS 2 publisher for sending messages.
```

```
//Declare Subscribers to be used
```

```
rcl_subscription_t subscriber;
```

```
//Declare Messages to be used
```

```
std_msgs__msg__Float32 msg_PWM; //Defines a message of type Float32.
```

```
//Define Macros to be used
```

```
#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)){}}
```

```
//Specifies GPIO pin 13 for controlling an LED
```

```
#define LED_PIN 22
```

```
// PWM Configuration
```

```
#define PWM_FRQ 980 //Define PWM Frequency
```

```
#define PWM_RES 8 //Define PWM Resolution
```

```
#define PWM_CHNL 0 //Define Channel
```

```
#define MSG_MIN_VAL -1 //Define min input value
```

```
#define MSG_MAX_VAL 1 //Define max input value
```



```

// Variables to be used

float pwm_set_point = 0.0;


//Define Error Functions

void error_loop(){

    while(1){

        digitalWrite(LED_PIN, !digitalRead(LED_PIN)); // Toggle LED state

        delay(100); // Wait 100 milliseconds

    }

}


//Define callbacks

void subscription_callback(const void *motor_robotronicos)

{

    //Get the message received and store it on the message msg

    const std_msgs__msg__Float32 *msg = (const std_msgs__msg__Float32
*)motor_robotronicos;

    pwm_set_point = constrain(msg->data, MSG_MIN_VAL, MSG_MAX_VAL);


//Movimiento del motor

    if (pwm_set_point > 0){

        digitalWrite(In1, HIGH);

        digitalWrite(In2, LOW);

    } else if (pwm_set_point == 0) {

        digitalWrite(In1, LOW);

```

```
        digitalWrite(ln2, LOW);  
    } else {  
        digitalWrite(ln1, LOW);  
        digitalWrite(ln2, HIGH);  
    }
```

```
int val_PWM = (int)(fabs(pwm_set_point) * 255); // PMW entre 0 y 255  
ledcWrite(PWM_CHNL, val_PWM);
```

```
//Publica el valor de PWM  
msg_PWM.data = pwm_set_point;  
RCSOFTCHECK(rcl_publish(&publisher_pwm, &msg_PWM, NULL));  
}
```

```
//Setup  
void setup() {  
    // Initializes communication between ESP32 and the ROS 2 agent (Serial).  
    set_microros_transports();
```

```
//Setup Microcontroller Pins  
pinMode(ENA, OUTPUT);  
pinMode(ln1, OUTPUT);  
pinMode(ln2, OUTPUT);  
pinMode(LED_PIN, OUTPUT);
```

```

ledcSetup(PWM_CHNL, PWM_FRQ, PWM_RES); //Setup the PWM

ledcAttachPin(ENA, PWM_CHNL);          //Setup Attach the Pin to the Channel


//Connection delay

delay(2000);


//Initializes memory allocation for Micro-ROS operations.

allocator = rcl_get_default_allocator();


//Creates a ROS 2 support structure to manage the execution context.

RCCHECK(rclc_support_init(&support, 0, NULL, &allocator));


// create node

RCCHECK(rclc_node_init_default(&node, "motor_robotronicos", "", &support));


// create publisher

RCCHECK(rclc_publisher_init_default(
    &publisher_pwm,
    &node,
    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Float32),
    "pwm_robotronicos"));


// create subscriber

RCCHECK(rclc_subscription_init_default(

```

```

    &subscriber,

    &node,

    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Float32),

    "cmd_pwm_robotronicos"));

// Initializes the Micro-ROS Executor, which manages tasks and callbacks.
RCCHECK(rclc_executor_init(&executor, &support.context, 2, &allocator));

RCCHECK(rclc_executor_add_subscription(&executor, &subscriber, &msg_PWM,
&subscription_callback, ON_NEW_DATA));

}

void loop() {

    //Executor Spin

    delay(100);

    RCSOFTCHECK(rclc_executor_spin_some(&executor, RCL_MS_TO_NS(100)));

}

```