



Instituto Tecnológico de Estudios Superiores de Monterrey

Campus Puebla

Fundamentación de Robótica (Gpo 101)

Reporte: Reto 3 Manchester Robotics

Alumno

José Diego Tomé Guardado A01733345
Pamela Hernández Montero A01736368
Victor Manuel Vázquez Morales A01736352
Fernando Estrada Silva A01736094

Fecha de entrega

Jueves 7 de Marzo de 2024

RESUMEN

El reto se centra en la implementación de micro-ROS en la tarjeta de desarrollo ESP32 para poder controlar la velocidad de un motor mediante la publicación de mensajes a través de tópicos. De igual forma, se busca implementar tópicos que puedan publicar datos relacionados con el valor entregado por el potenciómetro. También se detallaron los pasos para poder instalar la librería adecuada en Arduino Uno y usar nodos y tópicos.

OBJETIVOS

El objetivo principal radica en implementar micro-ROS en una tarjeta de desarrollo ESP32 para controlar un motor con encoder al publicar el duty cycle 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 ADC y 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 del potenciómetro y su transmisión a través de tópicos ROS.
- Implementar nodos de control para recibir los datos del potenciómetro y enviar las lecturas a través de tópicos.

INTRODUCCIÓN

La inclusión de tecnologías como Micro-ROS, módulos ADC y PWM en dispositivos como ESP32 ha generado nuevas perspectivas en el diseño e implementación de sistemas robóticos distribuidos y conectados. Micro-ROS, un marco robótico diseñado específicamente para dispositivos con recursos computacionales limitados (como microcontroladores de 32 bits), simplifica la integración de componentes en sistemas distribuidos para el desarrollo de aplicaciones de robots ROS 2.0. Por otro lado, los módulos ADC posibilitan que microcontroladores como ESP32 conviertan señales analógicas en digitales, mientras que los módulos PWM brindan un control preciso sobre dispositivos como motores y actuadores.

Micro-ROS se integra estrechamente con los microcontroladores, permitiéndoles funcionar como nodos de ROS en un sistema robótico. Dado que estos dispositivos están sujetos a restricciones de memoria, capacidad de procesamiento y energía, Micro-ROS está optimizado para operar en este entorno, utilizando la menor cantidad posible de recursos. Emplea DDS-XRCE (Data Distribution Service - eXtremely Resource Constrained Environments) como middleware. Esto habilita a los microcontroladores para intercambiar mensajes y datos con otros nodos de ROS 2.0 en la red, facilitando la comunicación en tiempo real entre los diversos componentes de un sistema robótico distribuido.

Por otro lado, un módulo convertidor analógico-digital (ADC) es un componente electrónico fundamental que convierte señales analógicas en señales digitales. Este proceso de conversión se lleva a cabo en múltiples etapas, como se señala en el diagrama de la figura 1. A través de cada paso, la señal de entrada analógica se transforma progresivamente en un

código digital comprensible para el sistema. Cada etapa de este proceso de conversión contribuye a la precisión y calidad del resultado final. A continuación se explica más detalladamente cada paso.

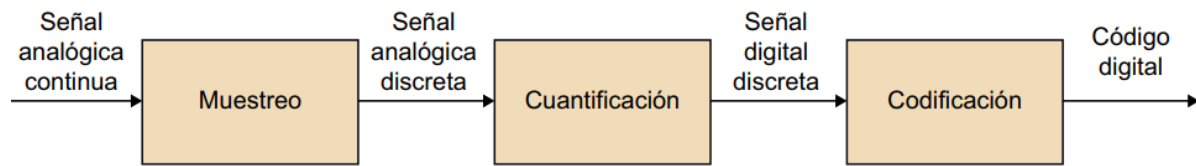


Figura 1. Módulo ADC

Muestreo: La señal analógica de entrada se mide en intervalos regulares de tiempo, como se puede observar en la figura 2 que representa el proceso de muestreo. Cada punto rojo en el eje del tiempo indica un momento de muestreo, donde se toma una medición de la señal analógica. Es importante que la frecuencia de muestreo (Sampling rate) sea al menos el doble que la frecuencia de la señal para evitar el efecto aliasing (frecuencias erróneas en la señal) y garantizar una representación precisa de la señal digitalizada. A medida que aumenta la frecuencia de muestreo, se mejora la precisión de la conversión.



Figura 2. Módulo ADC: fase de muestreo.

Cuantificación: Durante esta etapa, el valor de la señal analógica muestreada se transforma en un valor digital mediante la división del rango de la señal en un número finito de pasos discretos, como se ejemplifica en la figura 3 en relación con la figura 2. A mayor cantidad de pasos discretos en esta división, mayor será la resolución del ADC y, consecuentemente, su precisión en la conversión de la señal analógica a digital.



Figura 3. Módulo ADC: fase de cuantificación.

Codificación: Se refiere al proceso de representar la señal digitalizada en un formato que sea adecuado para su transmisión, almacenamiento o procesamiento posterior. Después de que la señal analógica ha sido muestreada y cuantificada, se convierte en una secuencia de números digitales. Estos números pueden ser codificados utilizando diferentes técnicas, como codificación de pulsos, codificación de delta o codificación Huffman, dependiendo de los requisitos del sistema y el tipo de señal que se está tratando. La codificación es esencial para garantizar que la señal digitalizada se pueda transmitir, almacenar o procesar de manera eficiente y precisa.

Compresión: Por último la compresión se refiere al proceso de reducir el tamaño de los datos digitales sin perder información importante. En este contexto la compresión se utiliza para reducir la cantidad de datos necesarios para representar una señal digitalizada sin comprometer su calidad o integridad. Esto se logra eliminando redundancias o información no esencial de los datos digitales.

Existen diferentes técnicas de compresión, como la compresión sin pérdida y la compresión con pérdida. La compresión sin pérdida preserva toda la información de la señal original, mientras que la compresión con pérdida sacrifica cierta información para lograr una mayor reducción en el tamaño de los datos. La compresión es útil para ahorrar espacio de almacenamiento y ancho de banda en la transmisión de datos, lo que resulta especialmente importante en aplicaciones donde los recursos son limitados

Dado que la ESP32 cuenta con pines digitales, carece de la capacidad para leer señales analógicas de manera directa. En consecuencia, cuando se requiere controlar un motor u otro dispositivo que opera con señales analógicas, como ajustar la velocidad o dirección de un motor, es indispensable emplear un módulo ADC. Este componente se encarga de convertir las señales analógicas en formatos digitales comprensibles para la ESP32. De esta manera, la ESP32 puede interpretar y aprovechar la información analógica para controlar el motor con precisión y eficacia en diversas aplicaciones robóticas y de automatización.

Por último, el PWM (Modulación por Ancho de Pulso, por sus siglas en inglés) es una técnica utilizada para controlar la cantidad de energía entregada a dispositivos como motores, luces LED y servomotores. Funciona generando pulsos eléctricos de voltaje constante con una duración variable. Estos pulsos tienen dos estados: alto (encendido) y bajo (apagado).

La proporción del tiempo durante el cual el pulso está en estado alto con respecto al tiempo total del ciclo determina la cantidad de energía entregada al dispositivo controlado. El ciclo de trabajo, expresado como un porcentaje, representa la relación entre el tiempo durante el cual la señal está en estado alto y el período total del ciclo. Por ejemplo, un ciclo de trabajo del 50% significa que la señal está encendida la mitad del tiempo y apagada la otra mitad.

En el caso de motores eléctricos, el PWM se utiliza para controlar la velocidad del motor ajustando el ciclo de trabajo de la señal PWM. Cuanto mayor sea el ciclo de trabajo, mayor será la potencia entregada al motor y, por lo tanto, mayor será su velocidad.

En resumen, se emplea la ESP32 como plataforma principal para implementar micro-ROS y controlar un motor con encoder. Por un lado se utiliza un módulo ADC para convertir las señales analógicas del potenciómetro en digitales comprensibles para la ESP32, mientras que por otro, se utiliza un módulo PWM para ajustar la velocidad y dirección del motor mediante señales PWM generadas por la ESP32. El conjunto de estos componentes posibilitan la integración de micro-ROS en la ESP32 y la creación de nodos y tópicos para la comunicación en un sistema robótico distribuido.

SOLUCIÓN DEL PROBLEMA

Tal y como mencionamos anteriormente, en muchas ocasiones la cantidad de tareas y operaciones que se deben realizar para un proyecto pueden ser bastantes que resulta necesario hacer uso de microcontroladores para “distribuir” el trabajo. Ahora bien, para trabajar y crear nodos en distintos microcontroladores será necesario hacer uso de micro-ros y, para llegar a ello, debemos realizar lo siguiente:

1. Instalar los binarios para micro-ROS

Para poder hacer uso de micro-ROS, antes que nada será necesario realizar la instalación de los binarios que nos permitirán configurar y preparar nuestro entorno para posteriormente hacer uso de micro-ROS sin presentar ningún problema. Esto lo realizamos de la siguiente manera:

a) Crearemos un directorio nuevo para realizar la configuración de micro-ROS. Dentro de este directorio clonamos el repositorio de github para micro-ROS:

```
mkdir -p ros2_utilities_ws/src
cd ~/ros2_utilities_ws/src
git clone -b $ROS_DISTRO https://github.com/micro-ROS/micro_ros_setup.git
```

b) Actualizamos los paquetes del sistema:

```
sudo apt update -y
sudo apt upgrade -y
sudo apt full-upgrade -y
sudo apt autoremove -y
sudo apt autoclean -y
sudo apt purge -y
```

c) Posteriormente, realizamos la compilación del proyecto y realizamos la configuración respectiva para hacer uso de micro-ros:

```

cd ~/ros2_utilities_ws/
rosdep update
rosdep install --from-paths src --ignore-src -y
sudo apt install python3-pip -y
cd ~/ros2_utilities_ws/
colcon build
source install/local_setup.sh
cd ~/ros2_utilities_ws/
ros2 run micro_ros_setup create_agent_ws.sh
ros2 run micro_ros_setup build_agent.sh
source install/local_setup.sh
echo "source ~/ros2_utilities_ws/install/local_setup.bash" >>
~/.bashrc

```

d) Finalmente, es importante configurar los permisos para poder utilizar el puerto en el que estará conectado nuestro microcontrolador:

```

sudo dmesg | grep tty
sudo chmod a+rw /dev/tty*
sudo usermod -a -G dialout $USER

```

2. Agregar las librerías al IDE de Arduino

Para poder utilizar la librería de microROS en arduino tendremos que entrar al siguiente link para poder descargar el zip que se encuentra dentro del repositorio:

https://github.com/micro-ROS/micro_ros_arduino/releases/tag/v2.0.7-humble

Posterior a ello, dentro de nuestro arduino , debemos añadir la librería descargada en la sección de *Sketch*, tal y como se muestra a continuación:

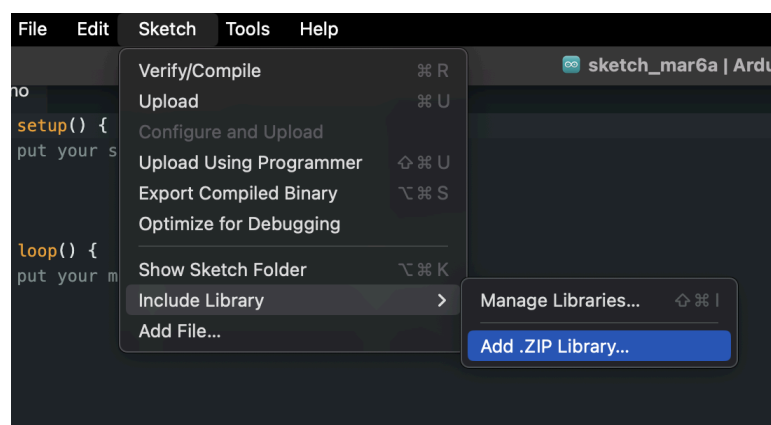


Figura 4. Instalación de la librería en Arduino

De esta forma, te abrirá una ventana nueva en la que deberás dirigirte al directorio donde se encuentra tu librería descargada. Posteriormente, dentro de arduino podrás ver el mensaje

Processing micro_ros_arduino-2.0.7-humble.zip y, finalmente, te arrojará un mensaje indicando que la librería ha sido correctamente instalada (*Successfully installed library from micro_ros_arduino-2.0.7-humble.zip archive*).

3. Implementación de código

Para dar solución a este problema, nos hemos apoyado en los recursos compartidos por Manchester Robotics. Tal y como se solicita en las instrucciones del reto número 3, se espera el desarrollo de un programa en micro-ROS sobre una tarjeta de desarrollo ESP32. Se controla un motor mediante terminal haciendo uso de tópicos y nodos. Se reciben valores a través de un suscriptor para velocidad y dirección.

```
rcl_node_t node_handle;

rcl_publisher_t publisher_raw_pot;
rcl_publisher_t publisher_voltage;
rcl_subscription_t duty_cycle_subscriber;
rcl_subscription_t direction_motor;

rcl_allocator_t allocator;
rcl_executor_t executor;
rcl_support_t support;
```

El programa inicia con la declaración de variables globales y handles, los cuales actúan como un tipo de referencia a distintas entidades dentro del ambiente de ROS, tales como nodos, publicadores, suscriptores, entre otros.

A continuación, se incluye la sección de declaración de pines para configuración inicial. Se incluyen pines de entrada y salida en función del pinout del Esp32. Además, se incluyen otras variables como la frecuencia del PWM y resolución de 8 bits en el tipo de dato correspondiente.

```
//Pin led debugger:
#define LED_PIN 2
//Pin del potenciómetro:
#define POT_PIN 4
//PWM:
#define EnA 26
#define In1 14
#define In2 27
//Configuración:
#define frequency 5000
#define resolution 8
#define PWM1_Ch 0
//Variable para la lectura del potenciómetro
int pot = 0;
int pwm = 0;
int dir_ = 0;
```

Las siguientes funciones tienen como propósito verificar que el retorno de funciones en micro-ROS funcionen de forma adecuada, retornando un valor booleano. Como en la práctica de ejemplo (publisher), nuevamente se tiene una función *error_loop* que parpadea un LED en caso de existir un error. Suele ser muy útil este tipo de funciones para realizar un debug en cada proceso del programa.

```
#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 para indicar un error:
void error_loop(){
    while(1){
        digitalWrite(LED_PIN, !digitalRead(LED_PIN));
        delay(100);
    }
}
```

Las funciones de callback tienen el propósito de ejecutar una tarea cada cierto tiempo determinado. En este programa se utilizan dos: una se encarga de leer la lectura analógica del potenciómetro y la otra de hacer un tipo de mapeo de la lectura del potenciómetro a 3.3 voltios. Una vez calculadas, se publican en su respectivo *publisher*.

```
//Callback para el timer de 10ms:
void timer_10ms_callback(rcl_timer_t * timer, int64_t last_call_time)
{
    RCLC_UNUSED(last_call_time);
    if (timer != NULL) {
        pot = analogRead(POT_PIN);
        //pot++;
    }
}

void timer_100ms_callback(rcl_timer_t * timer, int64_t last_call_time)
{
    RCLC_UNUSED(last_call_time);
    if (timer != NULL) {
        raw_pot.data = pot;
        voltage.data = pot*3.3/4095;
        RCSOFTCHECK(rcl_publish(&publisher_raw_pot, &raw_pot, NULL));
        RCSOFTCHECK(rcl_publish(&publisher_voltage, &voltage, NULL));
    }
}
```


Del otro lado, se crean funciones callback pero para los suscriptores. Una se suscribe a duty cycle, cambiando temporalmente la velocidad del motor. Por su parte, la segunda suscripción, se encarga de darle dirección al motor, cambiando la escritura de los pines asignados en el puente H.

```
void subscription_pwm_callback(const void * msgin)
{
    const std_msgs__msg__Float32 * duty_cycle = (const
std_msgs__msg__Float32 *)msgin;
    pwm = (duty_cycle->data)*255; //El valor de entrada va de 0 a 1,
mapeamos
    ledcWrite(PWM1_Ch, pwm); //Escribimos sobre el canal de pwm la
velocidad deseada
}

void subscription_direction(const void * msgin)
{
    const std_msgs__msg__Int32 * dir = (const std_msgs__msg__Int32
*)msgin;
    dir_ = dir->data; //Obtenemos el valor y lo guardamos en dir_

    //Realizamos las comparaciones
    if(dir_ == 1){
        digitalWrite(In1, LOW); // 1: Derecha/Izquierda
        digitalWrite(In2, HIGH); // -1: Izquierda/Derecha
    } // 0: Se detiene

    else if (dir_ == -1){
        digitalWrite(In1, HIGH);
        digitalWrite(In2, LOW);
    }

    else if (dir_ == 0){
        digitalWrite(In1, LOW);
        digitalWrite(In2, LOW);
    }
}
```

Función setup()

Además de indicar configuraciones de pines para controlar velocidad y dirección de giro del motor, dentro de esta función se encuentra la construcción de nodos. EL nodo puede tener el nombre que el usuario desee, siendo en este caso `micro_ros_esp32_node`. Se crean

también los publisher, donde se debe indicar explícitamente el tipo de dato que se publicará, en embargo, por cuestiones de sintaxis el mensaje siempre debe ser llamado *msg*. Cada dato que se publique debe estar en un publisher por separado.

```
//Creamos el publisher 1, el cual publicara el dato crudo del
potenciometro
RCCHECK(rclc_publisher_init_default(
    &publisher_raw_pot, //IMPORTANTE indicar el handle al publisher
    &node_handle,       //y al nodo al que pertenece
    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Float32), //Tipo de dato
    "micro_ros_esp32/raw_pot")); //Nombre del topico

//Creamos el publisher 2, el cual publicará el voltaje
RCCHECK(rclc_publisher_init_default(
    &publisher_voltage,
    &node_handle,
    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Float32), //Tipo de dato
    "micro_ros_esp32/voltage")); //Nombre del topico
```

Las funciones de timer se encargan de crear temporizadores que expiran cada cierto tiempo, siendo 10 y 100 milisegundos respectivamente. Se encargan de completar tareas en periodos regulares en el tiempo especificado. Una vez que terminan, se manda a llamar a la función *timer_callback*.

```
//Create timer 10 ms:
const unsigned int timer_10ms_timeout = 10;
RCCHECK(rclc_timer_init_default(
    &timer_10ms,
    &support,
    RCL_MS_TO_NS(timer_10ms_timeout),
    timer_10ms_callback));

//Create timer 100 ms:
const unsigned int timer_100ms_timeout = 100;
RCCHECK(rclc_timer_init_default(
    &timer_100ms,
    &support,
    RCL_MS_TO_NS(timer_100ms_timeout),
    timer_100ms_callback));
```

Se crean los suscriptores en que se publican los valores anteriores. El primero se suscribe al topico de *duty_cycle* (*micro_ros_esp32/pwm_duty_cycle*), que lee un dato de tipo *Int32*. El

otro suscriptor hacia el tópic de direccionamiento del motor (`micro_ros_esp32/direction`).

```
//Creamos la subscripción al topico que publicara el duty_cycle
RCCHECK(rclc_subscription_init_default(
    &duty_cycle_subscriber, //Handle
    &node_handle, //Handle al nodo
    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Float32), //Tipo de dato
    "micro_ros_esp32/pwm_duty_cycle")); //Nombre del topico

//Creamos la subscripción al topico que publicara el sentido o
direccion del motor
RCCHECK(rclc_subscription_init_default(
    &direction_motor, //Handle a subscription
    &node_handle, //Handle al nodo
    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32), //Tipo de dato
    "micro_ros_esp32/direction"));
```

El ejecutor se encarga de enumerar el número de callbacks que tiene el programa. Indispensable para el funcionamiento adecuado del código.

```
RCCHECK(rclc_executor_init(&executor, &support.context, 4,
    &allocator));
```

Finalmente, como buena práctica, se ha mantenido un loop casi vacío, únicamente incluyendo el uso de la función `rclc_executor_spin_some` para hacerle saber al microcontrolador que debe ejecutar indefinidamente lo que se le ha asociado al executor anteriormente.

```
void loop() {
    delay(100);
    RCSOFTCHECK(rclc_executor_spin_some(&executor, RCL_MS_TO_NS(100)));
}
```

RESULTADOS

Para observar lo que nuestro programa arroja como resultado, antes es necesario cargar el archivo a nuestra ESP32, tal y como usualmente lo hacemos dentro del IDE de Arduino, seleccionando la tarjeta y el puerto correspondiente. Sin embargo, es importante tomar en cuenta las siguientes consideraciones:

1. Previamente, además de la instalación de las librerías, es necesario haber descargado dentro de *Board Manager* los paquetes correspondientes para hacer uso de la ESP32.

- Es muy probable que se presenten problemas al intentar cargar el programa, esto debido a que el puerto designado a la ESP32 no cuenta con los permisos necesarios. Para arreglar esto, en una nueva terminal se deberán ejecutar los siguientes comandos:

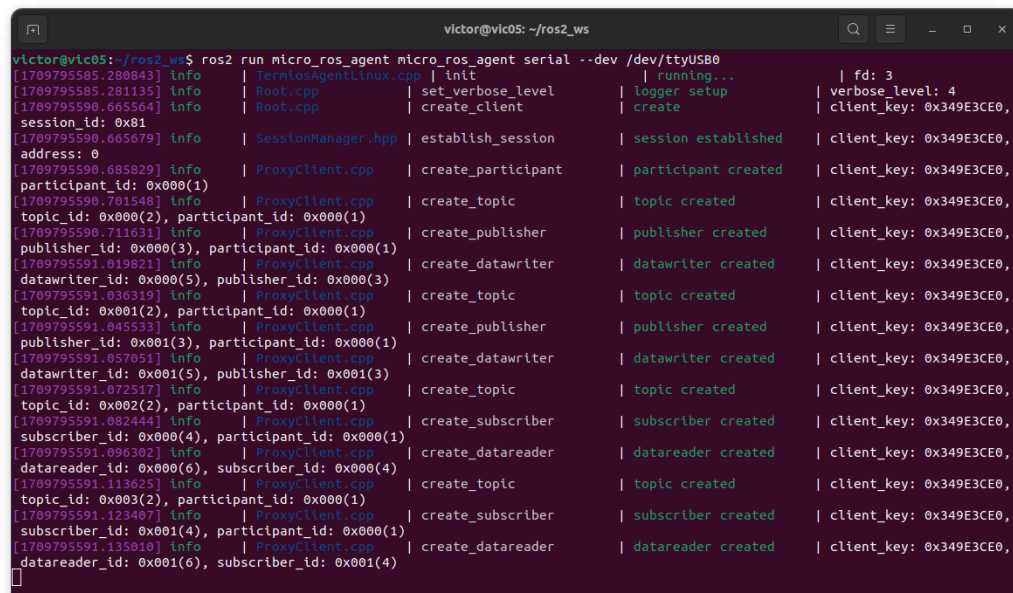
```
cd /dev
sudo chmod 666 /dev/ttyACM*
sudo chmod 666 /dev/ttyUSB*
```

De esta forma, con el primer comando, podremos identificar el puerto designado a la ESP32 y, posteriormente, usando el comando 2 o 3 (dependiendo el puerto) nos aseguraremos de otorgar los permisos correspondientes. Una vez hecho esto, podremos cargar el archivo .ino sin problemas.

Ahora bien, tras haber cargado programado nuestra ESP32 es probable que el LED que hemos conectado nos indique que hay errores presentes. Para arreglar eso, realizaremos lo siguiente en terminal:

```
cd ros2_ws Necesario para usar los comandos de ros
source install/setup.bash
ros2 run micro_ros_agent micro_ros_agent serial --dev /dev/ttyUSB0
```

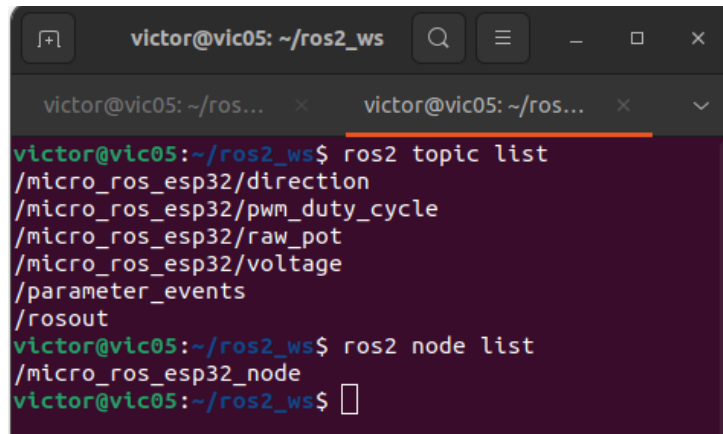
Ahora bien, una vez ejecutado el último comando, presionaremos el botón reset o enable de nuestra ESP32 para “eliminar” los errores (flasheo del LED) y, finalmente, podremos trabajar con los nodos creados:



```
victor@vic05: ~/ros2_ws
victor@vic05:~/ros2_ws$ ros2 run micro_ros_agent micro_ros_agent serial --dev /dev/ttyUSB0
[1709795585.280843] Info | TermiosAgentLinux.cpp | init | running... | fd: 3
[1709795585.281135] Info | Root.cpp | set_verbose_level | logger setup | verbose_level: 4
[1709795590.665564] Info | Root.cpp | create_client | create | client_key: 0x349E3CE0,
session_id: 0x81
[1709795590.665679] Info | SessionManager.hpp | establish_session | session established | client_key: 0x349E3CE0,
address: 0
[1709795590.685829] Info | ProxyClient.cpp | create_participant | participant created | client_key: 0x349E3CE0,
participant_id: 0x000(1)
[1709795590.781548] Info | ProxyClient.cpp | create_topic | topic created | client_key: 0x349E3CE0,
topic_id: 0x000(2), participant_id: 0x000(1)
[1709795590.711631] Info | ProxyClient.cpp | create_publisher | publisher created | client_key: 0x349E3CE0,
publisher_id: 0x000(3), participant_id: 0x000(1)
[1709795591.019821] Info | ProxyClient.cpp | create_datawriter | datawriter created | client_key: 0x349E3CE0,
datawriter_id: 0x000(5), publisher_id: 0x000(3)
[1709795591.036319] Info | ProxyClient.cpp | create_topic | topic created | client_key: 0x349E3CE0,
topic_id: 0x001(2), participant_id: 0x000(1)
[1709795591.045533] Info | ProxyClient.cpp | create_publisher | publisher created | client_key: 0x349E3CE0,
publisher_id: 0x001(3), participant_id: 0x000(1)
[1709795591.057051] Info | ProxyClient.cpp | create_datawriter | datawriter created | client_key: 0x349E3CE0,
datawriter_id: 0x001(5), publisher_id: 0x001(3)
[1709795591.072517] Info | ProxyClient.cpp | create_topic | topic created | client_key: 0x349E3CE0,
topic_id: 0x002(2), participant_id: 0x000(1)
[1709795591.082444] Info | ProxyClient.cpp | create_subscriber | subscriber created | client_key: 0x349E3CE0,
subscriber_id: 0x000(4), participant_id: 0x000(1)
[1709795591.096302] Info | ProxyClient.cpp | create_datareader | datareader created | client_key: 0x349E3CE0,
datareader_id: 0x000(6), subscriber_id: 0x000(4)
[1709795591.113625] Info | ProxyClient.cpp | create_topic | topic created | client_key: 0x349E3CE0,
topic_id: 0x003(2), participant_id: 0x000(1)
[1709795591.123407] Info | ProxyClient.cpp | create_subscriber | subscriber created | client_key: 0x349E3CE0,
subscriber_id: 0x001(4), participant_id: 0x000(1)
[1709795591.135810] Info | ProxyClient.cpp | create_datareader | datareader created | client_key: 0x349E3CE0,
datareader_id: 0x001(6), subscriber_id: 0x001(4)
```

Figura 5. Salida en terminal tras la configuración de ESP32.

Posterior a esto, abriremos una nueva terminal en la que podremos hacer uso de nuestros nodos y tópicos. Comenzaremos por solicitar a ros la lista de nodos y tópicos vivos en el momento, utilizando el comando `ros2 topic/node list`:

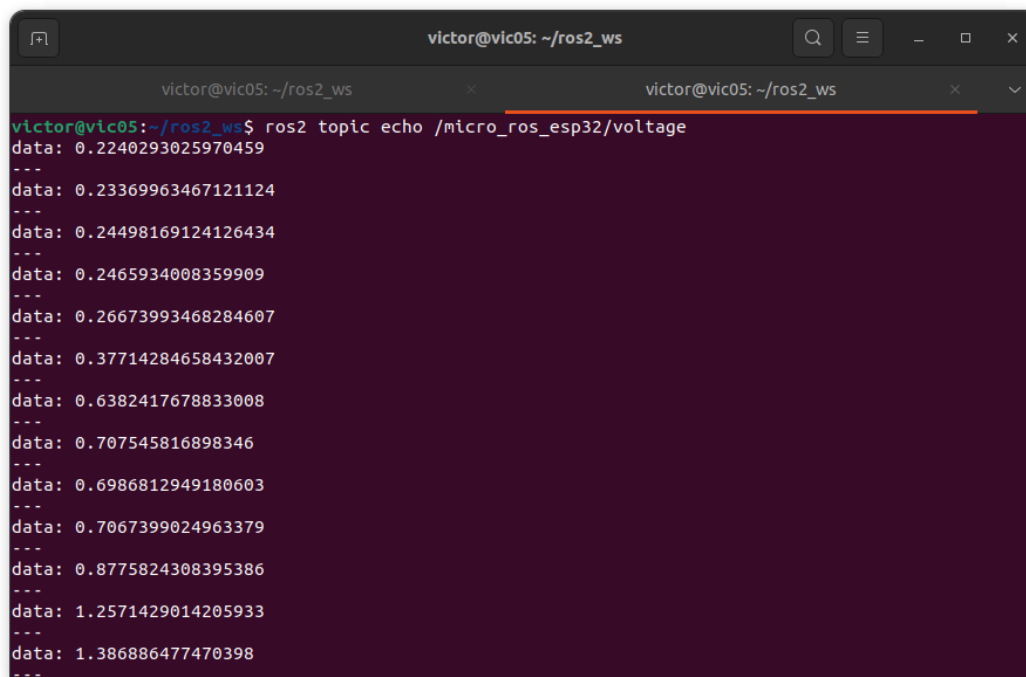


```
victor@vic05: ~/ros2_ws
victor@vic05: ~/ros... x victor@vic05: ~/ros... x v
victor@vic05:~/ros2_ws$ ros2 topic list
/micro_ros_esp32/direction
/micro_ros_esp32/pwm_duty_cycle
/micro_ros_esp32/raw_pot
/micro_ros_esp32/voltage
/parameter_events
/rosout
victor@vic05:~/ros2_ws$ ros2 node list
/micro_ros_esp32_node
victor@vic05:~/ros2_ws$
```

Figura 6. Lista de tópicos y nodos

Tal y cómo podemos observar en la *Figura 6*, contamos con cuatro tópicos, siendo dos los que pertenecen a los publisher que creamos dentro de nuestro código:

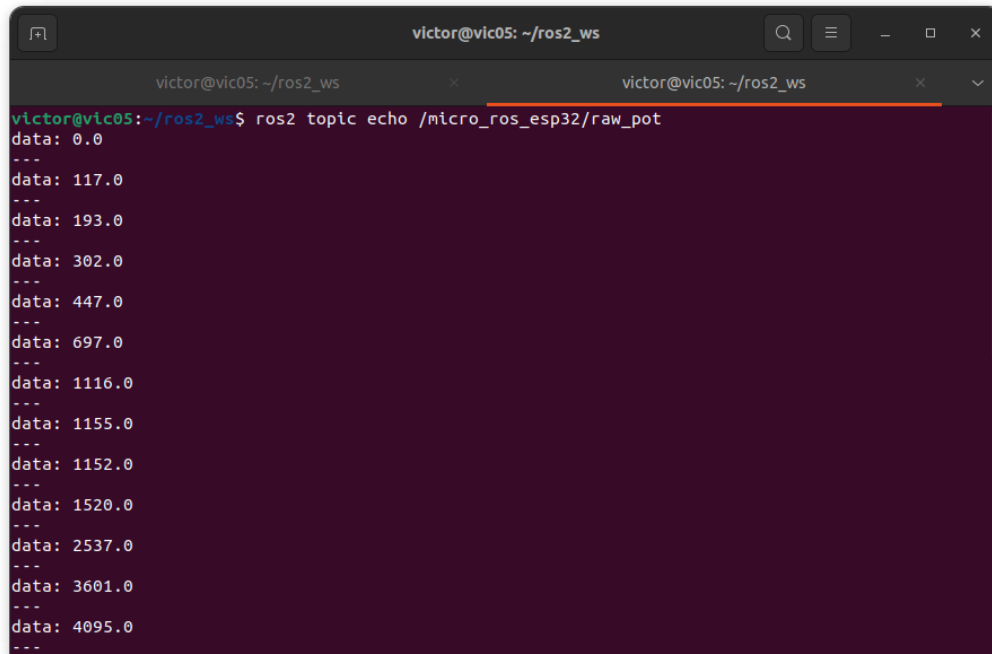
topic /micro_ros_esp32/voltage. Este tópico contiene el valor del potenciómetro mapeado en valores de voltaje, tomando valores desde 0 hasta 3.3V. A continuación, mostraremos lo que arroja en terminal mientras jugamos un poco con la perilla del potenciómetro. Para “escuchar” estos datos haremos uso del comando `ros2 topic echo /micro_ros_esp32/voltage`:



```
victor@vic05: ~/ros2_ws
victor@vic05: ~/ros2_ws x victor@vic05: ~/ros2_ws x v
victor@vic05:~/ros2_ws$ ros2 topic echo /micro_ros_esp32/voltage
data: 0.2240293025970459
---
data: 0.23369963467121124
---
data: 0.24498169124126434
---
data: 0.2465934008359909
---
data: 0.26673993468284607
---
data: 0.37714284658432007
---
data: 0.6382417678833008
---
data: 0.707545816898346
---
data: 0.6986812949180603
---
data: 0.7067399024963379
---
data: 0.8775824308395386
---
data: 1.2571429014205933
---
data: 1.386886477470398
---
```

Figura 7. Valores arrojados por el tópico micro_ros_esp32/voltage

topic /micro_ros_esp32/raw_plot. Como mencionamos en la solución, este tópico contiene el dato crudo del potenciómetro, el cual estaría dentro del rango 0 - 4095. Para visualizar los datos usaremos el comando echo utilizado anteriormente:



```
victor@vic05: ~/ros2_ws
victor@vic05: ~/ros2_ws
victor@vic05:~/ros2_ws$ ros2 topic echo /micro_ros_esp32/raw_pot
data: 0.0
---
data: 117.0
---
data: 193.0
---
data: 302.0
---
data: 447.0
---
data: 697.0
---
data: 1116.0
---
data: 1155.0
---
data: 1152.0
---
data: 1520.0
---
data: 2537.0
---
data: 3601.0
---
data: 4095.0
---
```

Figura 8. Valores arrojados por el tópico micro_ros_esp32/raw_pot

Ahora bien, antes de continuar, haremos uso de rqt_plot para poder visualizar los datos de una mejor manera, comparando raw_pot con voltage. Cabe aclarar que para esta parte, se realizó una modificación momentánea al código para poder visualizar el comportamiento de mejor manera, ajustando el valor de raw_pot para que fuera representado con la notación $\times 10^3$:

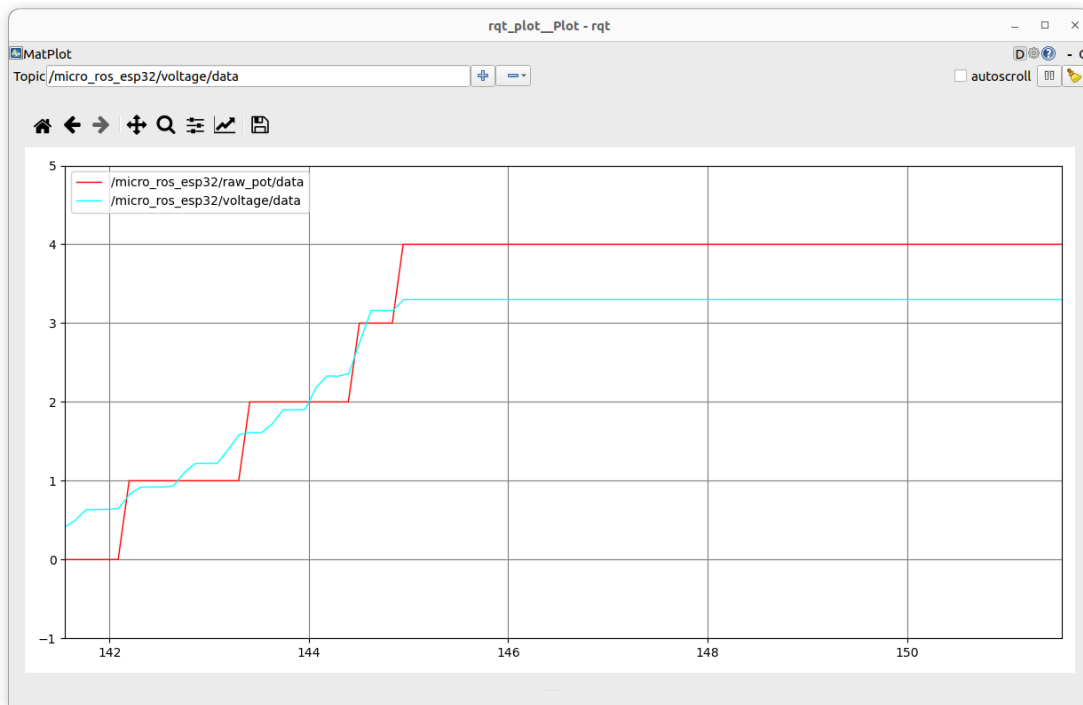


Figura 9. Voltage vs raw_pot.

En esta gráfica podemos observar como el valor de raw pot varía desde 0 hasta 4095 (escalado en $\times 10^3$). Como era de esperarse, al aumentar el raw_pot el valor de voltage igualmente aumenta. Esto debido a que son directamente proporcionales ya que uno depende del otro.

Finalmente, los siguientes tópicos que analizaremos serán los relacionados con los suscriptores de nuestro nodo en el microcontrolador.

topic /micro_ros_esp32/direction. Como ya se mencionó, nuestro nodo está suscrito a estos tópicos, esperando a recibir un dato para su procesamiento. Ahora bien, para publicar un dato dentro de un tópico haremos uso del siguiente comando:

```
ros2 topic pub /micro_ros_esp32/direction std_msgs/msg/Int32 "data: 1" --once
```

Haciendo uso de este comando, estamos mandando un 1 a través del tópico direction, lo que provocará que nuestro motor gire en un sentido en específico.

topic /micro_ros_esp32/pwm_duty_cycle. Por último, hemos llegado al tópico que nos permitirá cambiar la velocidad del motor. Para lograr esto, de manera similar al caso anterior, publicaremos un dato a través del tópico correspondiente. Por ejemplo:

```
ros2 topic pub /micro_ros_esp32/pwm std_msgs/msg/Float32 "data: 0.5" --once
```

A continuación mostramos un pequeño video del uso de este tópico en el que se cambia la velocidad del motor pasando de 0 a 0.55 y, finalmente, 0.9:

CONCLUSIONES

Elaborar ésta práctica tuvo sus complicaciones como cualquier otra, pues realizar la configuración del entorno para poder hacer uso de micro-ROS puede llegar a ser un problema bastante tedioso. En un inicio, algunos de los integrantes de nuestro equipo presentaron errores y problemas al intentar descargar los paquetes correspondientes de micro-ROS.

Dejando de lado un poco la parte de la configuración de micro-ROS, es importante hablar y reflexionar sobre los objetivos alcanzados a lo largo de esta pequeña práctica. Tal y como pudimos observar en puntos anteriores, logramos implementar en su totalidad lo que se nos solicitaba. De igual forma, resulta interesante mencionar que realizar la codificación de los nodos, publishers y subscribers en micro-ROS fue mucho más sencillo de lo esperado, ya que la sintaxis y lógica que se sigue es muy similar a lo que habíamos trabajado ya en semestres anteriores, como es el caso de FreeRTOS. Sin embargo, pese a entender y comprender de manera general el cómo realizar la programación, debemos reconocer que hay ciertas instrucciones de las que no tenemos mucha noción de su utilidad, ya que nos apoyamos bastante de los ejemplos que nos proporciona la librería. Es por ello, que consideramos necesario investigar y leer más al respecto para poder comprender y conocer mucho mejor cada una de las instrucciones que se implementan en micro-ROS, lo que nos permitirá crear códigos más complejos y eficientes en un futuro.

Ahora bien, durante la implementación de esta práctica utilizamos diferentes pines a los que se nos solicitaba, esto más que nada por la comodidad y practicidad al momento de realizar las conexiones. Sin embargo, durante las pruebas que realizamos notamos que en algunas ocasiones el valor que se leía del potenciómetro era siempre 0. Después de indagar y buscar un poco al respecto nos dimos cuenta que el problema se debía a que no estábamos usando pines que contarán con un convertidor analógico-digital.

Cómo retroalimentación a la práctica en general, creo que sería interesante agregar algún otro elemento, como un LED, que parpadeara a una velocidad dependiendo del valor del potenciómetro, y es que a pesar de no ser una gran novedad, considero que sería una buena mejora ya que como se pudo ver previamente, el potenciómetro y la velocidad del motor no están relacionadas, ya que esta última se cambia desde terminal. Considerando esta modificación, podría ser una mejora “visual” que nos ayudaría a identificar visualmente el valor del potenciómetro.

Finalmente, consideramos que el mayor aprendizaje que nos llevamos es que ahora siempre tomaremos en consideración si los pines que estamos utilizando están diseñados de tal manera que nos funcionen apropiadamente, ya que como nos pudimos dar cuenta, precipitar las conexiones sin revisar antes el pinout del microcontrolador puede llegar a costarnos tiempo valioso.

REFERENCIAS

Finocchiario F. (27 de Agosto del 2020). *Puerto micro-ROS a ESP32. micro-ROS*.
<https://micro.ros.org/blog/2020/08/27/esp32/>

FIRST y WPILib Contributors. (2024). *PWM Motor Controllers in Depth*.
<https://docs.wpilib.org/en/stable/docs/software/hardware-apis/motors/pwm-controllers.html>

Graña, C. Q. (2008). Estructuras avanzadas de convertidores analógico-digital (Doctoral dissertation, UNED. Universidad Nacional de Educación a Distancia).

Ingeniería Mecafenix. (18 de Mayo del 2021). Como funciona el convertidor analógico a digital.
<https://www.ingmecafenix.com/electronica/componentes/convertidor-analogico-a-digital/>

Manchester Robotics. (2024). Micro-ROS [Diapositivas de PowerPoint].
https://github.com/ManchesterRoboticsLtd/TE3001B_Robotics_Foundation_2024/blob/main/Week%203/Slides/4%20-%20MCR2_ROS_microRos.pdf

MicroPython Documentation.(Última actualización 07 de Marzo del 2024.). *Pulse Width Modulation*. <https://docs.micropython.org/en/latest/esp8266/tutorial/pwm.html>

Tech. (18 de Mayo del 2020). *Introduction to the micro-ROS Framework*. Fiware.
<https://www.fiware.org/2020/05/18/introduction-to-the-micro-ros-framework/>

Vazquez Gallego F. (2017). *Conversión analógico-digital*.
https://openaccess.uoc.edu/bitstream/10609/141046/10/PLA3_Conversi%C3%B3n%20anal%C3%B3gico-digital.pdf