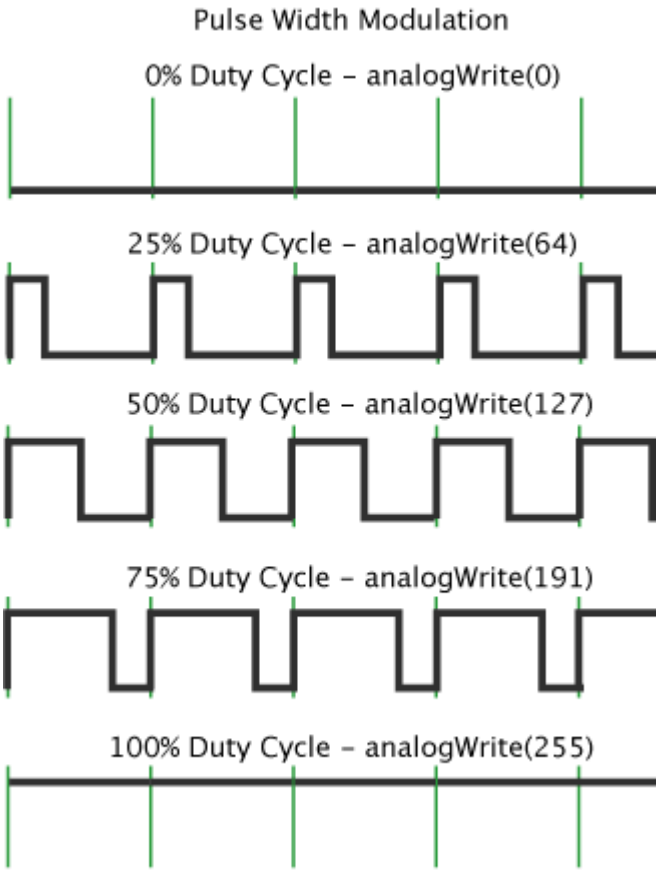
Tutorial 5 ROS Programming – Actuators and PID

Objective

In this tutorial, students learn to develop an Arduino program using an ESP32 microcontroller to regulate an encoder motor's rotation speed via a basic PID control algorithm.

1 Control the encoder motor through Pulse Width Modulation (PWM)

Pulse Width Modulation (PWM) is a technique used to control the power delivered to devices like encoder motors by varying the width of pulses in a digital signal. For encoder motors, which combine a DC motor with an encoder for position or speed feedback, PWM is commonly used to adjust the motor's speed and direction. The PWM signal consists of a series of pulses with a fixed frequency, where the duty cycle—the percentage of time the signal is "on" (high) versus "off" (low)—determines the average power supplied. For example, a 50% duty cycle delivers half the maximum power, slowing the motor, while a 100% duty cycle provides full power.



1. Create a new sketch by File>New Sketch
2. Include the necessary libraries and define the pin number of Encoder motor by adding these codes into the sketch:

```
#include <micro_ros_arduino.h>


#include <stdio.h>
#include <rcl/rcl.h>
#include <rcl/error_handling.h>
#include <rclc/rclc.h>
#include <rclc/executor.h>


#include <std_msgs/msg/int32.h>
#include <std_msgs/msg/string.h>




#define BUZZER_PIN 2
#define STB 13
#define PWMA 5
#define AIN1 4
#define AIN2 15
#define ENCODER_A 34
#define ENCODER_B 35


// Encoder specs
#define PPR 13;
#define GEAR_RATIO 45.0


// Encoder tracking
volatile long encoderCount = 0;
double currentRPM = 0;


unsigned long tBuzzerStarted;
bool bBuzzerStarted = false;
```

3. Define micro-ROS variables and implement the timer callback for publishing encoder counts by these codes:

```
#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)) {} \
  }


rcl_publisher_t publisher_rpm;
std_msgs__msg__Int32 msg_rpm;


rcl_subscription_t subscriber_set_pwm;
std_msgs__msg__Int32 msg_set_pwm;


rclc_executor_t executor;
rclc_support_t support;
rcl_allocator_t allocator;
rcl_node_t node;
rcl_timer_t timer;


void error_loop() {
  while (1) {
    digitalWrite(BUZZER_PIN, !digitalRead(BUZZER_PIN));
    delay(100);
  }
}


void IRAM_ATTR encoderISR() {
  if (digitalRead(ENCODER_A) == digitalRead(ENCODER_B)) {
```

```
    encoderCount++;
  } else {
    encoderCount--;
  }
}

void timer_callback(rcl_timer_t * timer, int64_t last_call_time) {
  RCLC_UNUSED(last_call_time);
  if (timer != NULL) {
    // RPM calculation
    noInterrupts();
    long count = encoderCount;
    encoderCount = 0;
    interrupts();

    double revolutions = (double)count / PPR;
    double rpm_motor = revolutions * (600.0); // 100ms -> 600 Rev
    currentRPM = rpm_motor / GEAR_RATIO;

    msg_rpm.data = currentRPM;
    RCSOFTCHECK(rcl_publish(&publisher_rpm, &msg_rpm, NULL));
  }
}

void callback_set_pwm(const void *msgin) {
  const std_msgs__msg__Int32 *msg = (const std_msgs__msg__Int32 *)msgin;

  int32_t pwm;
  pwm = msg->data;
  if (pwm > 0) {
    // forward
    digitalWrite(AIN1, HIGH);
    digitalWrite(AIN2, LOW);
    pwm = (pwm > 255) ? 255 : pwm;
```

```
    analogWrite(PWMA, pwm);
  } else if (pwm < 0) {
    // backward
    digitalWrite(AIN1, LOW);
    digitalWrite(AIN2, HIGH);
    pwm = (pwm < -255) ? 255 : -pwm;
    analogWrite(PWMA, pwm);
  } else {
    // stop
    analogWrite(PWMA, 0);
  }

  digitalWrite(BUZZER_PIN, HIGH);
  tBuzzerStarted = millis();
  bBuzzerStarted = true;
}
```

4. Create the setup function with these codes:

```
void setup() {
  pinMode(BUZZER_PIN, OUTPUT);
  digitalWrite(BUZZER_PIN, LOW);

  // Initialize motor pins
  pinMode(STB, OUTPUT);
  pinMode(PWMA, OUTPUT);
  pinMode(AIN1, OUTPUT);
  pinMode(AIN2, OUTPUT);
  digitalWrite(STB, HIGH); // Enable motor driver
  digitalWrite(AIN1, LOW);
  digitalWrite(AIN2, LOW);
  analogWrite(PWMA, 0);

  // Initialize encoder pins
```

```
pinMode(ENCODER_A, INPUT);
pinMode(ENCODER_B, INPUT);

// Attach interrupt
attachInterrupt(digitalPinToInterrupt(ENCODER_A), encoderISR, CHANGE);

set_microros_transports();
delay(2000);

allocator = rcl_get_default_allocator();

//create init_options
RCCHECK(rclc_support_init(&support, 0, NULL, &allocator));

// create node
RCCHECK(rclc_node_init_default(
  &node, "hku_dase_micro_ros_node", "", &support));

// create timer
const unsigned int timer_timeout = 100;
RCCHECK(rclc_timer_init_default(
  &timer, &support,
  RCL_MS_TO_NS(timer_timeout),
  timer_callback));

// create publisher
RCCHECK(rclc_publisher_init(
  &publisher_rpm,
  &node,
  ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32),
  "/rpm",
  &rmw_qos_profile_sensor_data));

// create subscriber
```

```
  RCCHECK(rclc_subscription_init_default(
    &subscriber_set_pwm,
    &node,
    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32),
    "set_pwm"));

  // create executor
  // total number of handles = #subscriptions + #timers
  RCCHECK(rclc_executor_init(
    &executor, &support.context, 2, &allocator));
  RCCHECK(rclc_executor_add_timer(&executor, &timer));

  RCCHECK(rclc_executor_add_subscription(
    &executor,
    &subscriber_set_pwm,
    &msg_set_pwm,
    &callback_set_pwm,
    ON_NEW_DATA));
}
```

5. Create the loop function with these codes.

```
void loop() {
  // delay(100);
  RCCHECK(rclc_executor_spin_some(&executor, RCL_MS_TO_NS(10)));

  // Turn off the Buzzer if it is turned on for more than 500ms
  if ((bBuzzerStarted == true) && ((millis() - tBuzzerStarted) > 500)) {
    digitalWrite(BUZZER_PIN, LOW);
    bBuzzerStarted = false;
  }
}
```

6. Select the board by Tools>Board>esp32>ESP32 Dev Module

7. Select the port as "/dev/ttyUSB0" from Tools>Port
8. Verify and compile the sketch by Sketch>Verify/Compile
9. Open a new terminal and run commands.

```
$ ls /dev/tty*

# (After running "ls /dev/tty*", verify if there is an item named

# "/dev/ttyUSB0")

$ sudo chmod 666 /dev/ttyUSB0

$ sudo usermod -aG dialout $USER

$ newgrp docker

$ groups

# (verify if there is "docker" in the output)
```

10. Upload the sketch by Tools>Upload
11. After uploading is done, open a new terminal and turn on the micro-ROS-agent:

```
$ docker run -it --rm -v /dev:/dev --privileged --net=host microros/micro-ros-agent:jazzy serial --dev /dev/ttyUSB0 -v6
```

12. Check the message received in the ROS 2 topic by opening a new terminal and running:

```
$ ros2 topic echo /rpm
```

13. Send one of the following messages to control the motor's PWM in a new terminal:

```
$ # set PWM value to 200

$ ros2 topic pub --once /set_pwm std_msgs/msg/Int32 "{data: 100}"


$ # set PWM value to 0

$ ros2 topic pub --once /set_pwm std_msgs/msg/Int32 "{data: 0}"
```

2 Control the encoder motor through a PID algorithm

Before learning PID control, here is a brief introduction to **open-loop control** and **closed-loop control**.

Open-Loop Control: Open-loop control is a simple method to control a system, like an encoder motor's rotation speed, without feedback. In this approach, a command (e.g., a PWM signal with a fixed duty cycle) is sent to the motor to achieve a desired speed, but there's no mechanism to verify or adjust based on the actual speed. For example, setting a PWM duty cycle to 50% assumes the motor will run at half its maximum speed, but factors like load, voltage variations, or friction aren't accounted for. Open-loop control is straightforward and computationally light but often inaccurate, as it cannot correct for errors or disturbances, making it less reliable for precise speed control in robotics applications.

Closed-Loop Control: Closed-loop control, in contrast, uses feedback to ensure the motor's actual speed matches the desired speed. For an encoder motor, the encoder provides real-time data on the motor's position or velocity. A controller, such as a PID (Proportional-Integral-Derivative) algorithm, compares the desired speed (setpoint) with the actual speed (from encoder feedback) and adjusts the PWM duty cycle to minimize the error. The PID algorithm calculates corrections based on the proportional (current error), integral (accumulated error), and derivative (rate of error change) terms, enabling precise and adaptive speed control. Closed-loop control is more complex but ensures accuracy and robustness, making it ideal for teaching students how to implement precise motor control in ROS2-based robotics projects using an ESP32 and Micro-ROS.

1. In Arduino IDE, create a new sketch by File>New Sketch
2. Include the necessary libraries and define the pin number of Encoder motor by adding these codes into the sketch.

```
#include <micro_ros_arduino.h>

#include <micro_ros_utilities/type_utilities.h>


#include <stdio.h>

#include <rcl/rcl.h>

#include <rcl/error_handling.h>

#include <rclc/rclc.h>

#include <rclc/executor.h>


#include <std_msgs/msg/int32.h>
```

```
#include <std_msgs/msg/float32_multi_array.h>


#define BUZZER_PIN 2

#define STB 13

#define PWMA 5

#define AIN1 4

#define AIN2 15

#define ENCODER_A 34

#define ENCODER_B 35


// Encoder specs

#define PPR 13;

#define GEAR_RATIO 45.0


// Encoder tracking

volatile long encoderCount = 0;

double currentRPM = 0;


unsigned long tBuzzerStarted;

bool bBuzzerStarted = false;
```

3. Declare additional variables and PID parameters for the PID controller. Modify the pidInterval to adjust the update frequency of PID calculation and output.

```
// PID variables

double Setpoint = 0, Input = 0, Output = 0;

double Kp = 1.0, Ki = 0.5, Kd = 0.1;

unsigned long pidInterval = 10; // ms


// PID internal state

double error = 0, previous_error = 0;

double integral = 0, derivative = 0;
```

14. Define micro-ROS variables and implement callback functions, error_loop function, and encoderISR function with these codes:

```
#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)) {} \
  }


rcl_publisher_t publisher_rpm;
std_msgs__msg__Int32 msg_rpm;
rcl_subscription_t subscriber_set_rpm;
std_msgs__msg__Int32 msg_set_rpm;
rcl_subscription_t subscriber_set_pid;
std_msgs__msg__Float32MultiArray msg_set_pid;
rclc_executor_t executor;
rclc_support_t support;
rcl_allocator_t allocator;
rcl_node_t node;
rcl_timer_t timer;


void error_loop() {
  while (1) {
    digitalWrite(BUZZER_PIN, !digitalRead(BUZZER_PIN));
    delay(100);
  }
}


void IRAM_ATTR encoderISR() {
  if (digitalRead(ENCODER_A) == digitalRead(ENCODER_B)) {
    encoderCount++;
  } else {
```

```
    encoderCount--;
  }
}

void timer_callback(rcl_timer_t * timer, int64_t last_call_time) {
  RCLC_UNUSED(last_call_time);
  if (timer != NULL) {
    msg_rpm.data = currentRPM;
    RCSOFTCHECK(rcl_publish(&publisher_rpm, &msg_rpm, NULL));
  }
}

void callback_set_rpm(const void *msgin) {
  const std_msgs__msg__Int32 *msg = (const std_msgs__msg__Int32 *)msgin;

  int32_t rpm;
  rpm = msg->data;

  if (msg->data > 0) {
    // forward
    digitalWrite(AIN1, HIGH);
    digitalWrite(AIN2, LOW);
  } else {
    digitalWrite(AIN1, LOW);
    digitalWrite(AIN2, LOW);
  }

  Setpoint = (rpm > 255) ? 255 : rpm;

  digitalWrite(BUZZER_PIN, HIGH);
  tBuzzerStarted = millis();
  bBuzzerStarted = true;
}
```

```
void callback_set_pid(const void *msgin) {
  const std_msgs__msg__Float32MultiArray *msg =
    (const std_msgs__msg__Float32MultiArray *)msgin;

  if (msg->data.size == 3) {
    Kp = msg->data.data[0];
    Ki = msg->data.data[1];
    Kd = msg->data.data[2];
  }

  digitalWrite(BUZZER_PIN, HIGH);
  tBuzzerStarted = millis();
  bBuzzerStarted = true;
}
```

4. Implement PID control in a new function ControlTask that runs on CPU core 1.

```
void ControlTask(void *pvParameters) {
  TickType_t xLastWakeTime = xTaskGetTickCount();
  const TickType_t xFrequency = pdMS_TO_TICKS(pidInterval);

  while (true) {
    // Wait for next cycle
    vTaskDelayUntil(&xLastWakeTime, xFrequency);

    double dt = pidInterval / 1000.0;

    // RPM calculation
    noInterrupts();
    long count = encoderCount;
    encoderCount = 0;
    interrupts();

    double revolutions = (double)count / PPR;
```

```
    double rpm_motor = revolutions * (60000.0 / pidInterval);
    currentRPM = rpm_motor / GEAR_RATIO;

    // PID calculation
    Input = currentRPM;
    error = Setpoint - Input;
    integral += error * dt;
    derivative = (error - previous_error) / dt;
    Output = Kp * error + Ki * integral + Kd * derivative;
    previous_error = error;

    Output = constrain(Output, 0, 255);
    analogWrite(PWMA, (int)Output);
  }
}
```

5. Create the setup function with these codes.

```
void setup() {
  pinMode(BUZZER_PIN, OUTPUT);
  digitalWrite(BUZZER_PIN, LOW);

  // Initialize motor pins
  pinMode(STB, OUTPUT);
  pinMode(PWMA, OUTPUT);
  pinMode(AIN1, OUTPUT);
  pinMode(AIN2, OUTPUT);
  digitalWrite(STB, HIGH); // Enable motor driver
  digitalWrite(AIN1, LOW);
  digitalWrite(AIN2, LOW);
  analogWrite(PWMA, 0);

  // Initialize encoder pins
  pinMode(ENCODER_A, INPUT);
```

```
  pinMode(ENCODER_B, INPUT);

  // Attach interrupt
  attachInterrupt(digitalPinToInterrupt(ENCODER_A), encoderISR, CHANGE);

  // Start control task on CPU1
  xTaskCreatePinnedToCore(
    ControlTask,       // Task function
    "ControlTask",     // Name
    2048,              // Stack size
    NULL,              // Parameters
    1,                 // Priority
    NULL,              // Task handle
    1                  // CPU core (1)
  );

  set_microros_transports();
  delay(2000);

  allocator = rcl_get_default_allocator();

  //create init_options
  RCCHECK(rclc_support_init(&support, 0, NULL, &allocator));

  // create node
  RCCHECK(rclc_node_init_default(
    &node, "hku_dase_micro_ros_node", "", &support));

  // Automatic allocate message memory using Micro-ROS's default conf.
  // Check website:
// https://micro.ros.org/docs/tutorials/advanced/handling_type_memory/
  static micro_ros_utilities_memory_conf_t conf = {0};
  micro_ros_utilities_create_message_memory(
    ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Float32MultiArray),
```

```
  &msg_set_pid,
  conf
);

// create timer
const unsigned int timer_timeout = 100;
RCCHECK(rclc_timer_init_default(
  &timer, &support,
  RCL_MS_TO_NS(timer_timeout),
  timer_callback));

// create publisher
RCCHECK(rclc_publisher_init(
  &publisher_rpm,
  &node,
  ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32),
  "/rpm",
  &rmw_qos_profile_sensor_data));

// create subscriber
RCCHECK(rclc_subscription_init_default(
  &subscriber_set_rpm,
  &node,
  ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Int32),
  "set_rpm"));

// create subscriber
RCCHECK(rclc_subscription_init_default(
  &subscriber_set_pid,
  &node,
  ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Float32MultiArray),
  "set_pid"));

// create executor
```

```
  // total number of handles = #subscriptions + #timers
  RCCHECK(rclc_executor_init(
    &executor, &support.context, 3, &allocator));
  RCCHECK(rclc_executor_add_timer(&executor, &timer));

  RCCHECK(rclc_executor_add_subscription(
    &executor,
    &subscriber_set_pid,
    &msg_set_pid,
    &callback_set_pid,
    ON_NEW_DATA));

  RCCHECK(rclc_executor_add_subscription(
    &executor,
    &subscriber_set_rpm,
    &msg_set_rpm,
    &callback_set_rpm,
    ON_NEW_DATA));

}
```

6. Create the loop function with these codes:

```
void loop() {
  // delay(100);
  RCCHECK(rclc_executor_spin_some(&executor, RCL_MS_TO_NS(10)));

  // Turn off the Buzzer if it is turned on for more than 500ms
  if ((bBuzzerStarted == true) && ((millis() - tBuzzerStarted) > 500)) {
    digitalWrite(BUZZER_PIN, LOW);
    bBuzzerStarted = false;
  }
}
```

7. Select the board by Tools>Board>esp32>ESP32 Dev Module
8. Select the port as “/dev/ttyUSB0” from Tools>Port

9. Verify and compile the sketch by Sketch>Verify/Compile
10. Open a new terminal:

```
$ ls /dev/tty*

# (After running “ls /dev/tty*”, verify if there is an item named

# “/dev/ttyUSB0”)

$ sudo chmod 666 /dev/ttyUSB0

$ sudo usermod -aG dialout $USER

$ newgrp docker

$ groups

# (verify if there is “docker” in the output)
```

11. Upload the sketch by Tools>Upload
12. After uploading is done, open a new terminal and turn on the micro-ROS-agent:

```
$ docker run -it --rm -v /dev:/dev --privileged --net=host microros/micro-ros-agent:jazzy serial --dev /dev/ttyUSB0 -v6
```

13. Open a new terminal and run the following command to monitor the value of rpm value.

```
$ ros2 topic echo /rpm
```

14. Open a new terminal and send a message to the kit. The motor should move at the specific speed.

```
$ # set to RPM value 200

$ ros2 topic pub --once /set_rpm std_msgs/msg/Int32 "{data: 200}"


$ # stop

$ ros2 topic pub --once /set_rpm std_msgs/msg/Int32 "{data: 0}"
```

15. Open a new terminal and send a message to the kit. The PID controller uses the provided PID parameters to calculate the PID output.

```
$ # Set Kp = 3.0, Ki = 0.2, Kd = 0.0

$ ros2 topic pub --once /set_pid std_msgs/msg/Float32MultiArray "{data:
[3.0, 0.2, 0.0]}"


$ # Set Kp = 3.0, Ki = 0.05, Kd = 0.0

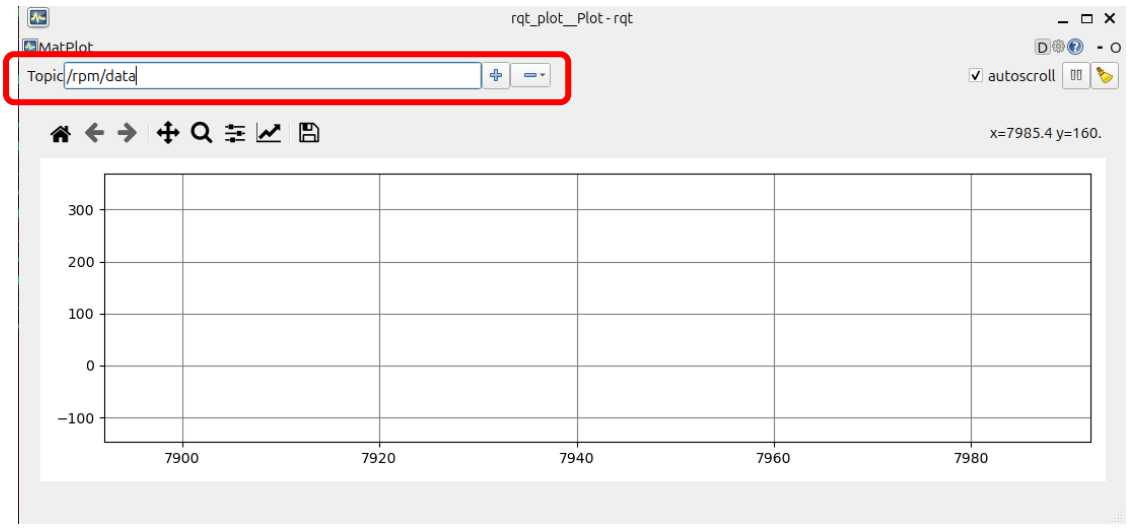
$ ros2 topic pub --once /set_pid std_msgs/msg/Float32MultiArray "{data:
[3.0, 0.05, 0.0]}"
```

16. Open a new terminal and launch rqt_plot for viewing the data as a graphical chart.

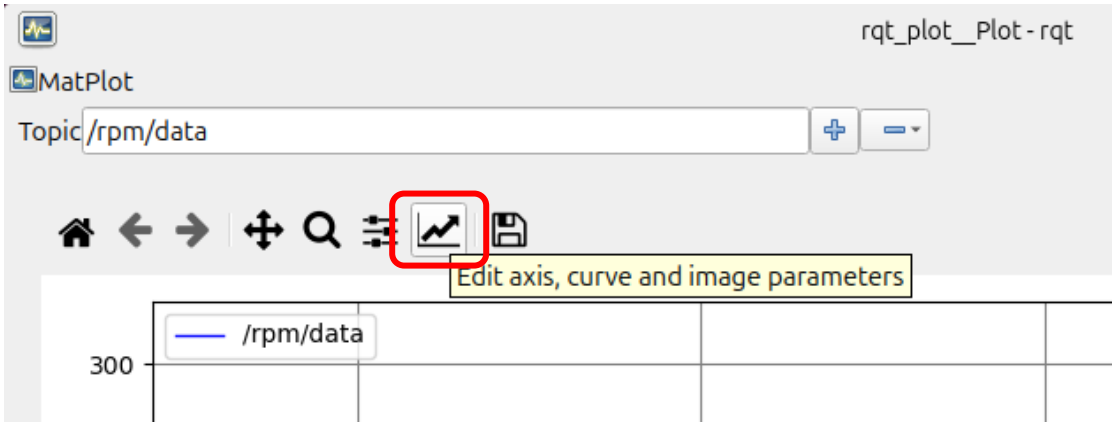
```
$ # launch rqt_plot

$ ros2 run rqt_plot rqt_plot
```

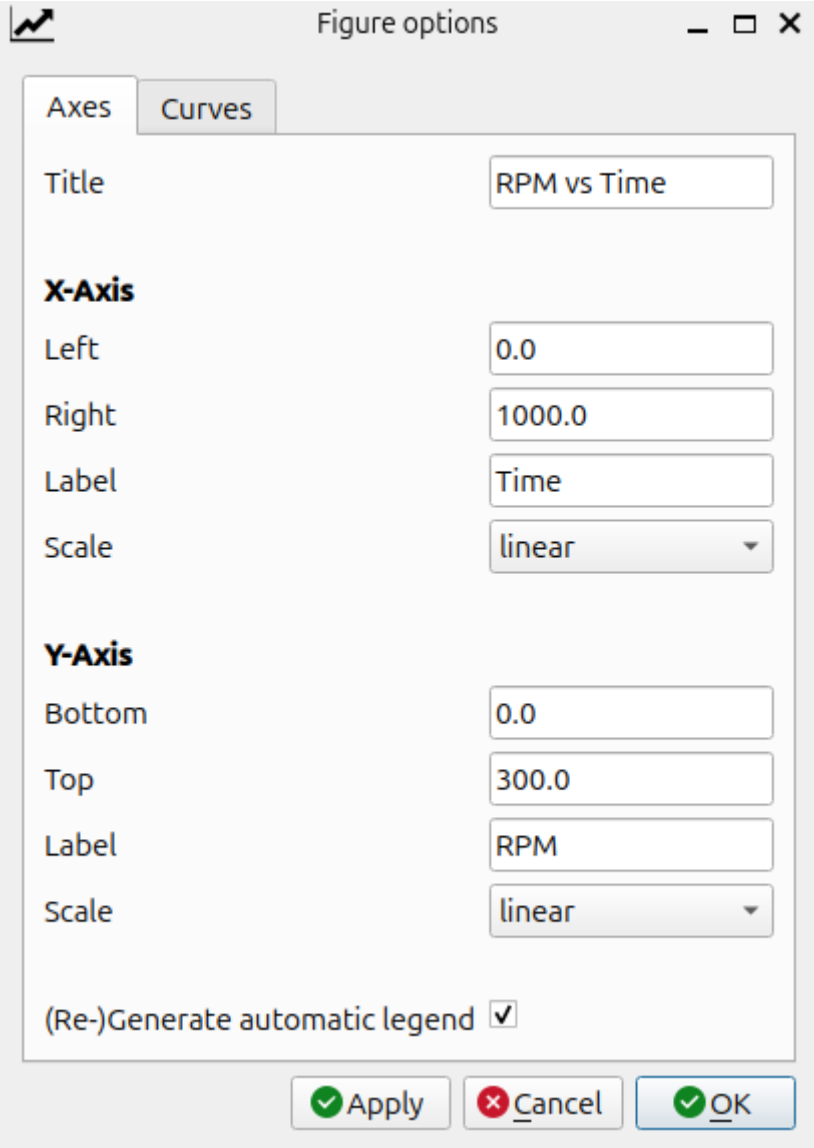
17. Enter "/rpm/data" in the "topic" entry and then click the "+" button.



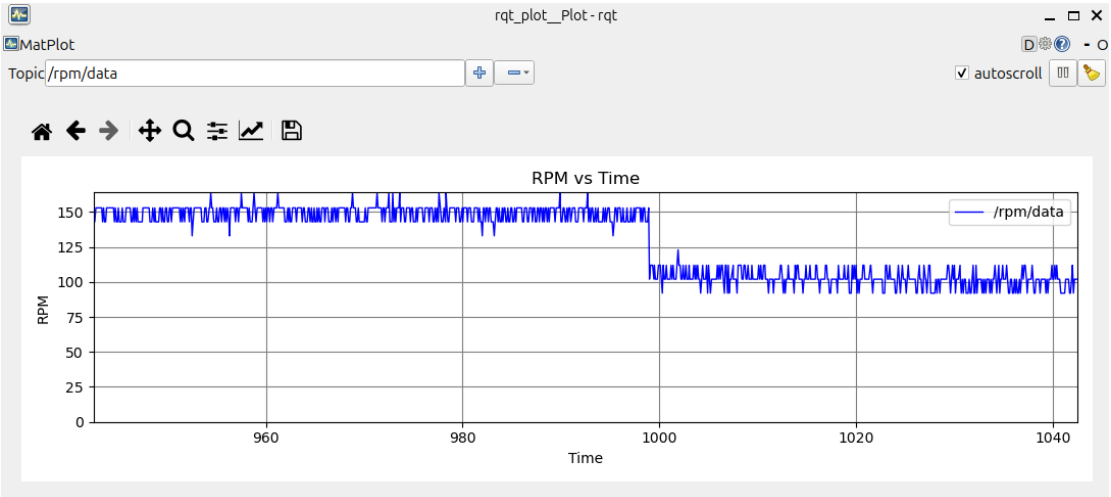
18. Click the "Chart" button to edit the figure options.



19. Input data as shown in the following figure and then click “OK”.



20. The rqt_plot will plot the RPM vs Time. You can try adjusting the RPM and PID values to see if the actual RPM reported changes.



Practice

1. Can you find any appropriate PID value for your tutorial kit that will drive the DC motor from 100 to 200 RPM in the shortest period while maintaining a pretty constant RPM over the long run?

2. How can the precision of RPM measurement be improved?

Acknowledgement

1. Micro-ROS Project. https://micro.ros.org/

References

1. Micro-ROS for Arduino GitHub project.
https://github.com/micro-ROS/micro_ros_arduino

2. ROS2 Jazzy Tutorial.
https://docs.ros.org/en/jazzy/index.html