Rover MEGA PROJECT

RoboTech

- **PWM from Raspberry:** Implementation and control of Pulse Width Modulation (PWM) signals using a Raspberry Pi for precise motor control.
- **Switching Circuit:** Design and analysis of an efficient switching circuit to manage high-power loads.
- **DC:** Exploration of DC motor characteristics and performance under various operating conditions.
- **IMU:** Integration of an Inertial Measurement Unit (IMU) for real-time orientation and motion tracking.
- **SPI:** Configuration of the Serial Peripheral Interface (SPI) for fast and reliable communication between devices.
- **ROS bridge:** Development of a ROS bridge to enable seamless communication between robotic hardware and software.
- **Robot Localization:** Explains the use of IMU, GPS, and wheel odometry, combined with EKF, to accurately determine the rover's position and orientation in real-time.
- **ROS Nodes Used:** Describes the purpose and core functionalities of each ROS node in the system, covering data acquisition, processing, and communication tasks essential for rover operation.
- **Integration and Launching All Nodes:** Describes the purpose and core functionalities of each ROS node in the system, covering data acquisition, processing, and communication tasks essential for rover operation.

Introduction

This is a brief of our research on each topic concerning the rover. It will include the purpose and functionality of each topic along its relevance to the rover.

Prerequisites

- **Install Pip (Python Package Installer)**

```
sudo apt update
sudo apt install python3-pip
pip3 --version
```

- **Install ROS Melodic**

```
sudo apt update
sudo apt install ros-melodic-desktop-full
```

- **Install ROSDEP (For Managing Dependencies)**

```
sudo apt install python3-rosdep
sudo rosdep init
rosdep update
```

- **Install Robot Localization Package**

```
sudo apt install ros-melodic-robot-localization
```

- **Install ROSbridge Server Package**

```
sudo apt-get update
sudo apt-get install ros-melodic-rosbridge-server
```

- **Install Smbus Library For I2C Devices Interfacing**

```
sudo apt update
sudo apt install python3-smbus
```

- **Install Spidev Library For SPI Devices Interfacing**

```
pip install spidev
```

1. PWM from Raspberry

1.1. Introduction

Pulse Width Modulation "PWMïs a means of modulating the signal wave. This is done by changing the duty cycle of the wave.

1.2. Package Installation

To integrate the PWM, Rassperry GPIO library is needed.

```
sudo apt install python3-rpi.gpio
```

Or

```
sudo apt install python3-pip
pip3 install RPi.GPIO
```

1.3. Test Code

```
import RPi.GPIO as GPIO
import time

# Pin setup
PWM_PIN = 18  # GPIO18 supports hardware PWM
GPIO.setmode(GPIO.BCM)  # Broadcom pin-numbering scheme
GPIO.setup(PWM_PIN, GPIO.OUT)

# Initialize PWM
frequency = 100  # Frequency in Hz
pwm = GPIO.PWM(PWM_PIN, frequency)
pwm.start(0)  # Start PWM with 0% duty cycle

try:
    while True:
        # Gradually increase brightness
        for duty_cycle in range(0, 101, 5):  # 0 to 100%
            pwm.ChangeDutyCycle(duty_cycle)
            time.sleep(0.1)
        # Gradually decrease brightness
        for duty_cycle in range(100, -1, -5):
            pwm.ChangeDutyCycle(duty_cycle)
            time.sleep(0.1)
except KeyboardInterrupt:
    print("Exiting program.")
finally:
    pwm.stop()
    GPIO.cleanup()
```

2. Switching Circuit

2.1. Introduction

Switching Circuit is a circuit responsible for stable switching through multiple components in a system. This circuit helps stabilize and maintain good performance for adequate accessibility.

The Switching Circuit used in the system is an H-Bridge. This circuit is designed to switch between DC motors. An H-Bridge is essential for precise motor control in robotics. It supports diverse applications, specifically the ones that need bi-directional motor operation. Also, it provides speed control to the motors by the 'EN' Pin in H-Bridge. This allows a speed control on the motor with "PWM".

2.2. H-Bridge Concept

An H-Bridge contains multiple numbers of transistors. They help with switching and controlling the required direction of a DC motor. The commonly used types are:

- **MOSEFT**
- **BJT**

2.3. H-Bridge Selection

There are specific kinds of H-Bridges which are commonly known and used in robotics such as:

- **L298N:** Dual H-Bridge motor driver; can drive two DC motors.
- **L293D:** Another dual H-Bridge motor driver similar to the L298N.
- **L293:** A basic motor driver similar to L293D, suitable for low-power applications.

A preferred suggestion for the selection is the "L298N"DC motor driver. It provides 2 DC motor control and connection. In the DC motor section, the arrangement and installation of motors are clarified.

3. DC Motors

3.1. Introduction

DC motors are the backbone of the system as they drive the whole ROVER towards its target. In this section, motor arrangement will be clarified and how to install them.

3.2. DC Motors with H-Bridges

In this section, more explanation for DC motor pins and how to control the direction and speed of the motor.

3.2.1. DC Motor Pins

DC motors have mainly 2 pins:

- **Positive (+ve) Pin**
- **Negative (-ve) Pin**

3.2.2. DC Motor Direction Control

From these pins' configuration, we can control the direction by setting and resetting the 2 pins. This way, the motors can be driven with constant speed without variation.

3.2.3. DC Motor Speed Control

From the previous section, the motors can't change their speed, only their direction. To control their speed, we need an external component and it's "H-Bridge".

As previously mentioned, an H-bridge includes an 'EN' pin. That permits controlling the speed by Pulse Width Modulation "PWM". This is done by changing the duty cycle of the signal sent to the motor. In the end of this section, an explanation code is provided to comprehend how to control the direction and modulate the direction.

3.3. DC Motor Arrangement

In the system, there are 4 DC motors. All of the motors are positioned on the corners of the ROVER. There are a couple of means and arrangements for controlling the DC motors:

- **4-Separate DC Motors:** The 4 DC motors will be controlled individually by an H-Bridge . This is not as efficient as it is supposed to be due to:
  1. **Money-Wise Consideration**
  2. **Hardware Integration Complexity**
- **2 pairs of DC Motors:** Each pair of DC motors will connect with a single H-Bridge. This is an effective solution and approach due to various reasons:
  1. **Cost-Effective Solution**
  2. **Less Component Integration**

  After observing both arrangements, the second one is preferable for its better performance and compatibility.

  In that motor arrangement, the 2 motors on right are connected with an H-Bridge "L298N.ªnd other 2 are connected with an another H-Bridge "L298N".

  In this arrangement, an external power supply is needed as the voltage driving the DC motors is different from the voltage of the ESP32 or Rassperry pi.

3.4. Test Code

The proceeding code is provided to ensure a well-rounded level of comprehension.

```
import RPi.GPIO as GPIO
import time

# GPIO Pins
MOTOR_IN1 = 17  # GPIO pin for IN1
MOTOR_IN2 = 27  # GPIO pin for IN2
MOTOR_ENA = 22  # GPIO pin for ENA (PWM control)

# GPIO setup
GPIO.setmode(GPIO.BCM)  # Use BCM pin numbering
```

```
GPIO.setup(MOTOR_IN1, GPIO.OUT)
GPIO.setup(MOTOR_IN2, GPIO.OUT)
GPIO.setup(MOTOR_ENA, GPIO.OUT)

# Setup PWM
pwm = GPIO.PWM(MOTOR_ENA, 100)  # Set PWM to 100 Hz
pwm.start(0)  # Start with duty cycle 0 (motor off)

def motor_forward(speed=50):
    """Move motor forward at the specified speed (0-100)."""
    GPIO.output(MOTOR_IN1, GPIO.HIGH)
    GPIO.output(MOTOR_IN2, GPIO.LOW)
    pwm.ChangeDutyCycle(speed)

def motor_backward(speed=50):
    """Move motor backward at the specified speed (0-100)."""
    GPIO.output(MOTOR_IN1, GPIO.LOW)
    GPIO.output(MOTOR_IN2, GPIO.HIGH)
    pwm.ChangeDutyCycle(speed)

def motor_stop():
    """Stop the motor."""
    GPIO.output(MOTOR_IN1, GPIO.LOW)
    GPIO.output(MOTOR_IN2, GPIO.LOW)
    pwm.ChangeDutyCycle(0)

try:
    print("Testing DC Motor")
    while True:
        print("Motor Forward")
        motor_forward(50)  # 50% speed
        time.sleep(2)      # Run for 2 seconds

        print("Motor Backward")
        motor_backward(50)  # 50% speed
        time.sleep(2)

        print("Motor Stop")
        motor_stop()
        time.sleep(2)

except KeyboardInterrupt:
    print("Test stopped by user")

finally:
    pwm.stop()
    GPIO.cleanup()
```

```
        print("GPIO cleaned up")
```

4. Inertial Measurement Unit (IMU)

4.1. Introduction

An Inertial Measurement Unit (IMU) is a critical sensor in rover systems, providing essential data for navigation, control, and stability. Here are the main uses of the IMU in the rover:

- **Orientation and Attitude Monitoring:** The IMU provides real-time data on the rover's roll, pitch, and yaw, which represent its orientation in 3D space. This information is crucial for maintaining stability, especially when traversing uneven terrains or slopes.
- **Navigation:** Especially path correction, the IMU's gyroscope data is used to detect deviations from the intended trajectory, allowing the rover to adjust its movement and stay on course.
- **Obstacle Avoidance and Terrain Adaptation:** The IMU helps the rover detect and adapt to sudden changes in terrain, such as inclines or declines, by adjusting speed and direction accordingly. It works in conjunction with other sensors like ultrasonic sensors or cameras to navigate around obstacles.

4.2. IMU Configuration and Components

Mostly we only use the accelerometer (to measure the acceleration vector) and the gyroscope (to measure angular velocity), the temperature sensor is not really used. Here are the details of the IMU that we will use in our rover:

- **Module:** MPU6050 (6-axis, accelerometer + gyroscope).
- **Arduino IDE Library:** Adafruit_MPU6050.
- **Communication with the ESP32:** I2C, using "Wire.h" library.

4.3. Test Code

```
#include <Wire.h>
#include <MPU6050.h>

MPU6050 mpu;

void setup() {
  Serial.begin(115200);  // Initialize serial communication
  Wire.begin();          // Initialize I2C communication

  // Initialize MPU6050
  mpu.initialize();
  if (mpu.testConnection()) {
    Serial.println("MPU6050 connection successful!");
```

```
  } else {
    Serial.println("MPU6050 connection failed!");
    while (1);  // Stop execution if IMU fails
  }
}

void loop() {
  // Read accelerometer and gyroscope values
  int16_t ax, ay, az, gx, gy, gz;
  mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

  // Print data to Serial Monitor
  Serial.print("Ax: "); Serial.print(ax);
  Serial.print(" Ay: "); Serial.print(ay);
  Serial.print(" Az: "); Serial.print(az);
  Serial.print(" | Gx: "); Serial.print(gx);
  Serial.print(" Gy: "); Serial.print(gy);
  Serial.print(" Gz: "); Serial.println(gz);

  delay(500);  // Wait for 500ms before the next reading
}
```

4.4. IMU in Rover

The IMU provides information about the rover's orientation and movement. The key data needed for stabilization are:

- **Pitch:** Forward or backward tilt.
- **Roll:** Side-to-side tilt.
- **Yaw:** Rotational movement (not critical for stabilization but useful for navigation).

1. Process IMU Data Read and process IMU data to calculate the rover's current orientation.
    - Collect raw IMU data (accelerometer and gyroscope readings).
    - Calculate pitch and roll.

```
pitch = atan2(accelY, sqrt(accelX * accelX + accelZ * accelZ)) * 180 / PI;
roll = atan2(-accelX, accelZ) * 180 / PI;
```

2. Analyze IMU Data for Stability The Raspberry Pi receives the IMU data and checks the rover's tilt:
    - If pitch is too high (forward tilt), motors should adjust to prevent tipping over.
    - If roll is too high (side tilt), motors should adjust to balance the rover sideways.

For example:

- **Positive Pitch:** Increase back motor speed or decrease front motor speed.
- **Negative Pitch:** Increase front motor speed or decrease back motor speed.

3. Apply Stabilization Logic Use stabilization algorithms to decide how to adjust the motor speeds:
   - **PID Control:**
     - Implement a Proportional-Integral-Derivative (PID) controller for smooth corrections.
     - Calculate an error term: $Error = DesiredPitch - CurrentPitch$
     - Use PID gains (Kp, Ki, Kd) to compute the correction: $Correction = KpError + KiIntegral(Error) + KdDerivative(Error)$
   - **Output Correction:** Use the correction value to adjust motor speeds.

5. Serial Peripheral Interface (SPI)

5.1. Introduction

Serial Peripheral Interface (SPI) is a synchronous serial communication protocol used to transfer data between a master device and one or more peripheral devices. It operates by shifting bits one at a time through a full-duplex connection, with a clock signal synchronizing the data transmission.

SPI typically uses four main lines:

- **MOSI (Master Out Slave In):** Transmits data from the master to the slave.
- **MISO (Master In Slave Out):** Transmits data from the slave to the master.
- **SCLK (Serial Clock):** Provides a clock signal for synchronization.
- **CS/SS (Chip Select/Slave Select):** Used to select which peripheral device is being communicated with at any given time.

5.2. SPI in ESP32

The ESP32 microcontroller supports SPI communication on several GPIO pins, which can be configured as per the requirements of the application, in our case the communication of the ESP32 with the Raspberry Pi. The ESP32 has hardware SPI support, which provides efficient data transfer with minimal CPU involvement. This makes it ideal for high-speed applications.

Key Features:

- **Multiple SPI Buses:** The ESP32 has three SPI hardware peripherals (SPI0, SPI1, SPI2) that can be used independently, allowing simultaneous communication with multiple peripherals.
- **Flexible Pin Configuration:** The ESP32 allows the user to assign different pins to the SPI interface, which provides flexibility in designing circuits.
- **DMA Support:** The ESP32 supports Direct Memory Access (DMA) for SPI communication, which can offload data transfer to a peripheral and improve performance by reducing CPU usage.

5.2.1. Test Code in (cpp)

```
#include <SPI.h>

// SPI pins
#define SCK  18  // Clock
#define MISO 19  // Master In Slave Out
#define MOSI 23  // Master Out Slave In
#define SS   5   // Slave Select

void setup() {
  Serial.begin(115200);
  SPI.begin(SCK, MISO, MOSI, SS);  // Initialize SPI
  pinMode(SS, OUTPUT);             // Set SS as output
  digitalWrite(SS, HIGH);          // Start with SS high
}

void loop() {
  String data = "Hello, Pi!";  // Example data
  sendSPIData(data);
  delay(1000);
}

void sendSPIData(String data) {
  digitalWrite(SS, LOW);  // Pull SS low to start communication
  for (int i = 0; i < data.length(); i++) {
    SPI.transfer(data[i]);  // Send each character
  }
  digitalWrite(SS, HIGH);  // Pull SS high to end communication
}
```

5.3. SPI in Raspberry Pi

5.3.1. Setting up Raspberry Pi: SPI Slave

1. Enable SPI on the Raspberry Pi:

   ```
   sudo raspi-config
   ```

2. Install Python libraries for SPI:

   ```
   sudo apt-get install python3-dev python3-spidev
   ```

5.3.2. Test Code (in python)

```
import spidev
import time

# SPI initialization
spi = spidev.SpiDev()
spi.open(0, 0)  # Open SPI bus 0, chip select 0
spi.max_speed_hz = 50000  # Set SPI speed

def read_spi():
    while True:
        # Read data (use 0x00 as dummy byte to clock SPI)
        received_data = spi.xfer2([0x00] * 10)
        print("Received:", ''.join([chr(x) for x in received_data]))
        time.sleep(1)

try:
    read_spi()
except KeyboardInterrupt:
    spi.close()
```

6. ROS bridge

To integrate the data into ROS, use a Python ROS node on the Raspberry Pi. ROSbridge server provides a WebSocket-based interface to the ROS system. This allows a communication between ROS nodes and non-ROS applications like web-based GUIs, mobile apps, or remote systems.

1. Install the rosbridge_suite package to enable communication between ROS and external devices.

   ```
   sudo apt-get install ros-<distro>-rosbridge-server
   ```

2. Run the rosbridge_server:

   ```
   roslaunch rosbridge_server rosbridge_websocket.launch
   ```

3. Create a ROS node in Python to publish received SPI data to a topic.

   ```
   #!/usr/bin/env python3
   import rospy
   from std_msgs.msg import String
   import spidev

   # SPI initialization
   ```

```
spi = spidev.SpiDev()
spi.open(0, 0)
spi.max_speed_hz = 50000

def spi_to_ros():
    rospy.init_node('spi_publisher', anonymous=True)
    spi_pub = rospy.Publisher('spi_data', String, queue_size=10)
    rate = rospy.Rate(10)  # 10 Hz

    while not rospy.is_shutdown():
        received_data = spi.xfer2([0x00] * 10)
        message = ''.join([chr(x) for x in received_data]).strip('\x00')
        rospy.loginfo("Publishing: %s", message)
        spi_pub.publish(message)
        rate.sleep()

if __name__ == '__main__':
    try:
        spi_to_ros()
    except rospy.ROSInterruptException:
        spi.close()
```

7. Robot Localization

It is necessary to have the co-ordinates of the ROVER. To help locate the ROVER, we need some helping sensors to get the current characteristics.

There are sensors that are commonly used among the robotics systems.

- **IMU**
- **GPS**
- **Wheel Odometry**

  These sensors are used to get all needed parameters to define the co-ordinates for the ROVER.

  - **Position (x, y, z)**
  - **Orientation (Roll, Pitch, Yaw)**
  - **Linear Velocity (u, v, w)**
  - **Angular Velocity**
  - **Accelerations**

Each of them can handle something and others can't. Maybe some of them can handle multiple characteristics at once.

In the ROVER, using Encoders and IMU to estimate the co-ordinates for the ROVER more accurately.

By using EKF, it will help improve the accuracy of the ROVER positions.

There will be an IMU node and an Odometry node. Each node will publish their values to the topics. Then, the EFK will get this values and publish the filtered values to a topic /odometry/filtered". We can utilize from this topic to get another Node to present these values or implement them in other ascepts of the code.

8. ROS Nodes Used

1. `encoders.py`

Overview

This file implements a ROS node to handle encoder data for odometry calculations, which is crucial for tracking the rover's movement.

Core Features

- Uses GPIO to interface with hall-effect sensors to count wheel rotations.
- Calculates distance traveled based on the wheel diameter and encoder counts.
- Updates and publishes odometry data, including the rover's position (`x`, `y`) and orientation (`yaw`).
- Publishes odometry data to `/encoder/odom` for use in Extended Kalman Filter (EKF) fusion.
- Includes a stub class for simulating encoder data for testing purposes.

2. `imu.py`

Overview

This file provides a ROS node to handle and publish IMU (Inertial Measurement Unit) data, particularly for the MPU6050 sensor.

Core Features

- Interfaces with the MPU6050 sensor using the `board` and `busio` modules.
- Publishes:
  - Raw IMU data (linear acceleration and angular velocity) to `/imu/data`.
  - Yaw angle to `/imu/yaw` by integrating the gyroscope's z-axis data.
- Operates at a rate of 50 Hz to ensure high-frequency updates.
- Includes a stub class to generate and publish dummy IMU data for testing.

3. filtered_values_presenterTest.py

Overview

This file processes filtered odometry data and extracts meaningful insights for presentation.

Core Features

- Subscribes to the `/odometry/filtered` topic to receive odometry data.
- Converts orientation from quaternion to Euler angles (roll, pitch, yaw).
- Logs and displays position and orientation values in a readable format.
- Useful for debugging and verifying filtered odometry data.

4. localization_stack.py

Overview

This file implements a ROS node to manage and publish a localization stack, combining mine flags and odometry data.

Core Features

- Subscribes to the `/rostoweb/mine` topic to receive mine flags.
- Subscribes to the `/odometry/filtered` topic to track position data (`x`, `y`).
- Maintains a stack containing the mine flag and corresponding position.
- Publishes the current stack to `/localization_stack_status`.
- Logs stack updates and position changes for debugging purposes.

5. motor.py

Overview

This file provides a ROS node for motor control, enabling movement commands for the rover.

Core Features

- Interfaces with motor driver hardware to control the rover's motors.
- Implements movement commands such as forward, backward, right, and left.
- Subscribes to `webtoros` to receive movement instructions (L3x, L3y).
- Includes a safety mechanism to stop motors in case of command loss or error.

6. spi_sender.py

Overview

This file initializes a ROS node that sends SPI data based on controller input.

Core Features

- Initializes the ROS node called `spi_sender`.
- Creates a publisher for sending SPI data to the `spi_send_topic`.
- Subscribes to the `webtoros` topic to receive `controller` messages (L3X and L3Y).
- Sets up an SPI connection using `spidev` (bus 0, device 0) with a clock speed of 50 kHz and mode 0.
- Extracts L3X and L3Y values from controller data, converts them to `uint8` (0-255), and sends them via SPI.
- Publishes the sent data in an `SPI_data` message to the `spi_send_topic`.
- Switches to simulation mode and logs the failure if SPI initialization fails.

7. spi_receiver.py

Overview

This Python script initializes a ROS node that listens for SPI data and publishes a mine flag.

Core Features

- Initializes the ROS node called `spi_receiver`.
- Sets up a publisher to send `mine` messages to the `rostoweb/mine` topic.
- Establishes an SPI connection using `spidev` to receive data from an SPI bus.
- Reads 8 bytes from the SPI connection (expected to be an int64 flag indicating a mine).
- Logs the received mine flag and publishes it as a `mine` message to the `rostoweb/mine` topic.
- If SPI connection fails, switches to simulation mode and logs the failure.

Key ROS Topics Used

- `/webtoros`: Receives controller data for SPI communication.
- `/rostoweb/mine`: Receives mine flags for the localization stack.
- `/imu/data`: Publishes raw IMU sensor data.
- `/imu/yaw`: Publishes the yaw angle calculated from the gyroscope.

- `/odometry/filtered`: Processes and presents fused odometry data.
- `/encoder/odom`: Publishes encoder-based odometry data.
- `/localization_stack_status`: Publishes the current localization stack status.
- `/spi_send_topic`: Publishes SPI data to other nodes.

Each file plays a specific role in a robotics system:

- `imu.py`: Handles sensor data acquisition and processing.
- `encoders.py`: Calculates and tracks the rover's movement.
- `filtered_values_presenterTest.py`: Provides insights into processed odometry data for analysis or debugging.
- `localization_stack.py`: Combines mine flags and odometry data into a structured stack for localization.
- `motor.py`: Enables motor control and movement commands.
- `spi_sender.py`: Facilitates SPI communication based on controller inputs.
- `spi_receiver.py`: Listens for SPI data and publishes mine flags.

9. Integration and Launching All Nodes

It is possible that one can run each ROS node individually; although, it is not applicable and would take a lot of effort and time for big applications. Therefore, it is logical to perform an additional step to integrate all these used nodes into a single launching command or order.

Consequently, in our system, a launch file is provided to launch all the required files for the system, whether they are actual nodes or servers. The file is named "full setup.launch".

- **To Launch The File:** you need to be in the directory that includes your ROS package.

```
source devel/setup.bash
roslaunch <your_pkg> <your_launch_file>.launch
```

In our case, the package and launch files are as the proceeding command

```
roslaunch mega_rover_project_pkg full_setup.launch
```

Conclusion

The overall communication workflow of the rover will be something like this:

1. **ESP32:**
    - Gets data from IMU via I2C.
    - Sends IMU data to the Raspberry Pi via SPI.
2. **Rasberry Pi:**
    - Receives data from ESP32 via SPI.
    - Runs a ROS node that:
        - Publishes the IMU data from the ESP32 on a topic.
        - Processes the data (stabilizing logic or PID control).
    - Subscribes to this topic using motor control nodes.
3. **Motor Controllers:**
    - Subscribe to topics to receive control signals.
    - Act on those commands to adjust motor speed or direction.