Design Document - AMR Controller

210019E - Abithan A.
210292G - Kirushanth G
210498T - Priyankan V.
210624E - Sundarbavan T.

Submitted as a requirement for an assignment in the course module EN2160
at the Electronic and Telecommunication Department, University of Moratuwa
Colombo, Sri Lanka

10 July 2024

Contents

1 Design Details - Introduction **6**
1.1 Introduction . . . 6
1.2 Objective of our task . . . 6

2 PCB Design **7**
2.1 Introduction . . . 7
2.2 Component selection . . . 7
2.2.1 Integrated Circuits (ICs) . . . 7
2.2.2 Passive Components . . . 7
2.2.3 Connectors . . . 8
2.3 PCB Routing . . . 9
2.4 PCB Testing . . . 12

3 Solidworks Design **16**
3.1 Introduction . . . 16
3.2 The Platform . . . 16
3.2.1 Design Considerations . . . 16
3.2.2 Construction . . . 17
3.3 The Motor Driver Enclosure . . . 18
3.3.1 Design Considerations . . . 18
3.3.2 Construction . . . 18
3.3.3 Mold Design . . . 23

4 Hardware Implementation **25**
4.1 Power consumption and Power Supply . . . 25
4.2 Connecting Arduino Nano with Motor Drivers . . . 25
4.2.1 Hardware Configuration . . . 26

5 Software Details **27**
5.1 Introduction . . . 27
5.1.1 Project Overview . . . 27
5.1.2 Objective of the Documentation . . . 27
5.1.3 Target Audience . . . 27
5.1.4 Key Features . . . 27
5.2 System Requirements . . . 27
5.2.1 Hardware Requirements . . . 27
5.2.2 Software Requirements . . . 28
5.3 Initial Setup . . . 28
5.3.1 Software Installation . . . 28
5.3.2 Dependencies and Packages Installation . . . 30
5.3.3 Setting Up Development Environment . . . 31
5.4 ROS Basics . . . 33
5.4.1 ROS Distributions . . . 33
5.4.2 Nodes . . . 33
5.4.3 Topics & Messages . . . 33
5.4.4 Services . . . 33

5.4.5 Parameters & Remapping . . . 33
5.4.6 Launching . . . 34
5.4.7 System Packages (Underlay) . . . 34
5.4.8 Building a Workspace (Overlay) . . . 34
5.4.9 Quality of Service (QoS) . . . 34
5.5 ROS Transform System(TF) . . . 34
5.5.1 Transforms Overview . . . 34
5.5.2 Coordinate Frames . . . 34
5.5.3 Creating Transforms . . . 34
5.5.4 The TF2 Libraries . . . 34
5.5.5 Static and Dynamic Transforms . . . 35
5.5.6 Broadcasting Transforms . . . 35
5.5.7 Visualizing Transforms with RViz . . . 35
5.5.8 Broadcasting Dynamic Transforms . . . 35
5.5.9 Debugging with view_frames . . . 35
5.6 Describing a robot using URDF . . . 36
5.6.1 Introduction . . . 36
5.6.2 Links and Joints . . . 36
5.6.3 Defining Links . . . 36
5.6.4 URDF Syntax . . . 37
5.6.5 Using Xacro . . . 37
5.7 Simulations using Gazebo and RViz . . . 37
5.7.1 Introduction . . . 37
5.7.2 Gazebo Environment Basics . . . 38
5.7.3 Integrating Gazebo with ROS . . . 38
5.7.4 Spawning a Robot in Gazebo . . . 38
5.8 Building our own robot model . . . 39
5.8.1 Introduction . . . 39
5.8.2 Differential Drive Concept . . . 39
5.8.3 Recap of Important Ideas . . . 40
5.8.4 Resulted Robot . . . 47
5.9 Driving the virtual robot . . . 47
5.9.1 Introduction . . . 47
5.9.2 Spawning Our Robot in the Gazebo world . . . 47
5.9.3 Wrapping It Up in a Launch File . . . 47
5.9.4 Control Concept Overview . . . 49
5.9.5 Adding the Control Plugin . . . 50
5.9.6 Keyboard Teleoperation . . . 51
5.9.7 RViz Visualization . . . 58
5.10 ros2_control . . . 59
5.10.1 Introduction . . . 59
5.10.2 The Need for ros2_control . . . 59
5.10.3 Architecture of ros2_control . . . 59
5.10.4 Differential drive kinematics . . . 73
5.10.5 Practical Implementation . . . 74
5.10.6 Running and Managing Controllers . . . 74
5.11 Moving a real robot using ros2_control . . . 75
5.11.1 Overview of the Control System . . . 75
5.11.2 Setting Up the Hardware Interface . . . 76
5.11.3 diffdrive_arduino . . . 76

5.11.4 serial . . . 84
5.11.5 Setting Up the Launch File . . . 96
5.11.6 Testing the Configuration . . . 98
5.12 Interaction Between diff_drive_controller and diffdrive_arduino . . . 98
5.12.1 Control Flow . . . 98
5.12.2 diffdrive_arduino Hardware Interface . . . 99
5.13 Controlling motors . . . 99
5.13.1 Introduction . . . 99
5.13.2 Basic Power and Motor Drivers . . . 99
5.13.3 Motor Controllers: Open and Closed Loop . . . 100
5.13.4 Utilizing an Arduino Nano as a Motor Controller . . . 100
5.13.5 ROS System . . . 100
5.14 SLAM using slam_toolbox . . . 105
5.14.1 Introduction . . . 105
5.14.2 SLAM Overview . . . 105
5.14.3 Using the Map Frame to Account for Wheel Slippage and Other Issues . . . 105
5.14.4 Coordinate Frames in ROS . . . 106
5.14.5 Odometry and Its Limitations . . . 106
5.14.6 Introducing the Map Frame . . . 106
5.14.7 Topics and Data . . . 107
5.14.8 slam_toolbox . . . 107
5.14.9 Algorithms and Techniques . . . 108
5.14.10 Setting Up slam_toolbox . . . 109
5.14.11 Running SLAM in Simulation . . . 109
5.14.12 Saving the Map . . . 110
5.14.13 Using the Map . . . 110
5.14.14 Generated Maps . . . 112
5.15 Navigation using Nav2 Stack . . . 112
5.15.1 Introduction . . . 112
5.15.2 What is Navigation? . . . 112
5.15.3 Nav2 Stack . . . 113
5.15.4 Initialization . . . 113
5.15.5 Localization . . . 113
5.15.6 Global Path Planning . . . 114
5.15.7 Costmaps . . . 114
5.15.8 Local Path Planning and Control . . . 114
5.15.9 Obstacle Avoidance . . . 114
5.15.10 Behavior Tree Execution . . . 114
5.15.11 Detailed Algorithm Steps . . . 115
5.15.12 Installing Nav2 and Preparations . . . 116
5.15.13 Running Nav2 in Gazebo . . . 116
5.15.14 Diagrams of the frames broadcast by tf2 . . . 124
5.15.15 Active Nodes and Topics in the ROS system in our project . . . 125
5.16 Simulation Results . . . 125
5.16.1 Room1 . . . 125
5.16.2 Room2 . . . 125

6 Software Implementation **128**
6.1 Implementation . . . 128
6.1.1 Copying packages . . . 128

6.1.2 Running Simulations . . . 129
6.1.3 Moving a Real Robot . . . 134
6.2 Trobuleshooting . . . 136
6.2.1 Colcon build error . . . 136
6.2.2 Not sourcing underlay . . . 137
6.2.3 Not sourcing overlay . . . 137
6.2.4 Initial pose error . . . 137

A Previously used Arduino Code **138**

B Daily Log Details **140**
B.1 Project Selection and Proposal (01 - 29 February 2024) . . . 140
B.2 Conceptual Design (01 - 07 March 2024) . . . 140
B.3 Overview of SLAM Technologies (08 - 13 March 2024) . . . 140
B.3.1 Cartographer . . . 141
B.3.2 Hector SLAM . . . 141
B.3.3 SLAM Toolbox . . . 141
B.3.4 Choosing SLAM toolbox . . . 142
B.4 Simulation Tools (14 - 17 March 2024) . . . 143
B.4.1 ROS/Gazebo . . . 143
B.4.2 MATLAB . . . 143
B.4.3 Choosing Gazebo . . . 144
B.5 ROS1 or ROS2 (18 - 20 March 2024) . . . 144
B.5.1 ROS1 . . . 145
B.5.2 ROS2 . . . 145
B.5.3 Our Choice - ROS2 . . . 145
B.6 ROS2 installation and Simulation of Environment (21 - 27 March 2024) . . . 145
B.6.1 ROS2 Implementation . . . 145
B.6.2 Environment Simulation . . . 146
B.7 PCB Design and Finalizing (28 March - 04 April 2024) . . . 146
B.8 PCB Manufacturing (05 - 19 April 2024) . . . 146
B.9 Robot Platform Design and Production(20 - 25 April 2024) . . . 146
B.10 Enclosure Design for Motor Controller (26 April - 04 May 2024) . . . 146
B.11 PCB Assembly (05 - 10 May 2024) . . . 146
B.12 Atmega Chip Programming (24 June - 07 July 2024) . . . 147
B.13 Simulation Implementations (28 March 2024 - 07 July 2024) . . . 147

7 Download Links and References **148**

Chapter 1

Design Details - Introduction

1.1 Introduction

This section ensures that all stakeholders have a clear understanding of the AMR controller's basic architecture, the rationale behind design choices, and adherence to relevant industry standards. This provides a solid foundation for detailed design and development in subsequent phases of the project.

1.2 Objective of our task

The main objective of our project is to develop a sophisticated Autonomous Mobile Robot (AMR) controller capable of accurately mapping and autonomously navigating complex industrial environments. The controller is to be integrated with state-of-the-art Simultaneous Localization and Mapping (SLAM) technologies, with a focus on operational efficiency, adaptability, and precision.

Our aim is to tackle the following specific objectives:

1. To Implement Advanced SLAM Capabilities: Integrate robust SLAM algorithms that enable the AMR to interpret and interact with its environment dynamically. The goal is to allow the robot to create real-time maps of its surroundings while concurrently tracking its own position within these maps.
2. To Facilitate Real-time Navigation and Decision-Making: Develop algorithms that process sensor data to guide the AMR through obstacle-rich paths. This involves both real-time processing of environmental data and predictive modeling to anticipate and navigate around potential obstacles.
3. To Optimize Sensor Integration: Seamlessly combine data from various sensors, such as LiDAR and encoders, to provide the AMR with a comprehensive view of its operating environment, enhancing its ability to make informed navigation decisions.

Chapter 2

PCB Design

2.1 Introduction

This documentation outlines the design and implementation of a dual motor driver circuit, an essential component for controlling two motors simultaneously in the project. The PCB design is a critical aspect of this project, ensuring that the dual motor driver operates reliably and effectively. The layout and routing of the PCB have been optimized to handle the power and signal requirements of the motor driver circuit while minimizing noise and ensuring robust performance. This report will detail the key aspects of the PCB design, including component placement, routing strategy, and critical design considerations, showcasing how these elements contribute to the overall functionality and efficiency of the dual motor driver circuit.

2.2 Component selection

The selection process involved careful consideration of various parameters such as current handling capability, voltage ratings, power dissipation, and noise characteristics. Below is an overview of the key components used in the design and the rationale behind their selection.

2.2.1 Integrated Circuits (ICs)

1. **Motor Driver ICs (U2, U3):** The choice of motor driver ICs was critical, as they need to handle the required current and voltage for driving the motors. ICs U2 and U3 were selected for their high current handling capability, built-in protection features (such as thermal shutdown and overcurrent protection), and efficiency. These ICs ensure reliable motor operation under various load conditions.
   Our choice : MC33886PVWR2
   Alternatives considered : L298N H-bridge, L293D motordriver IC
2. **Voltage Regulator (U1):** U1 is a 5V voltage regulator responsible for providing a stable supply voltage to the motor driver ICs and micro controller.
   Our choice : TLV76750DGNR (LDO type regulator)
   Alternatives considered : MAX42402AFLA+ (switching regulator), LV14540DDAR (switching regulator),

2.2.2 Passive Components

1. **Resistors (R1, R2, R3, R4):** Precision resistors were chosen to set and stabilize current levels within the circuit. These resistors were selected based on their power ratings, tolerance, and temperature stability to ensure accurate and reliable performance.
2. **Capacitors (C1, C2, C3, C4, C5, C6):** Capacitors were selected for their roles in noise filtering, signal smoothing, and voltage stabilization.
3. **Flyback Diodes (D3, D4, D5, D6, D7, D8, D9, D10):** Diodes were placed across the motor terminals to protect the motor driver ICs from back electromotive force (EMF) generated by the motors. Schottky diodes were chosen for their fast switching speed and

low forward voltage drop, ensuring efficient protection.
Our choice : SS34-E3/9AT

4. **LEDs (D1, D2):** LEDs D1 and D2 are used to indicate the presence of the 12V power supply and the 5V output from the voltage regulator, respectively. These LEDs provide visual confirmation that the power supply and voltage regulation are functioning correctly, aiding in troubleshooting and ensuring proper operation.

2.2.3 Connectors

1. **Input/Output Connectors (J1, J2, J3, P1, P2, P3):** Connectors were chosen for their durability, ease of use, and reliable electrical connections. They facilitate the connection of power supplies, control signals, and motor outputs, ensuring secure and stable connectivity.

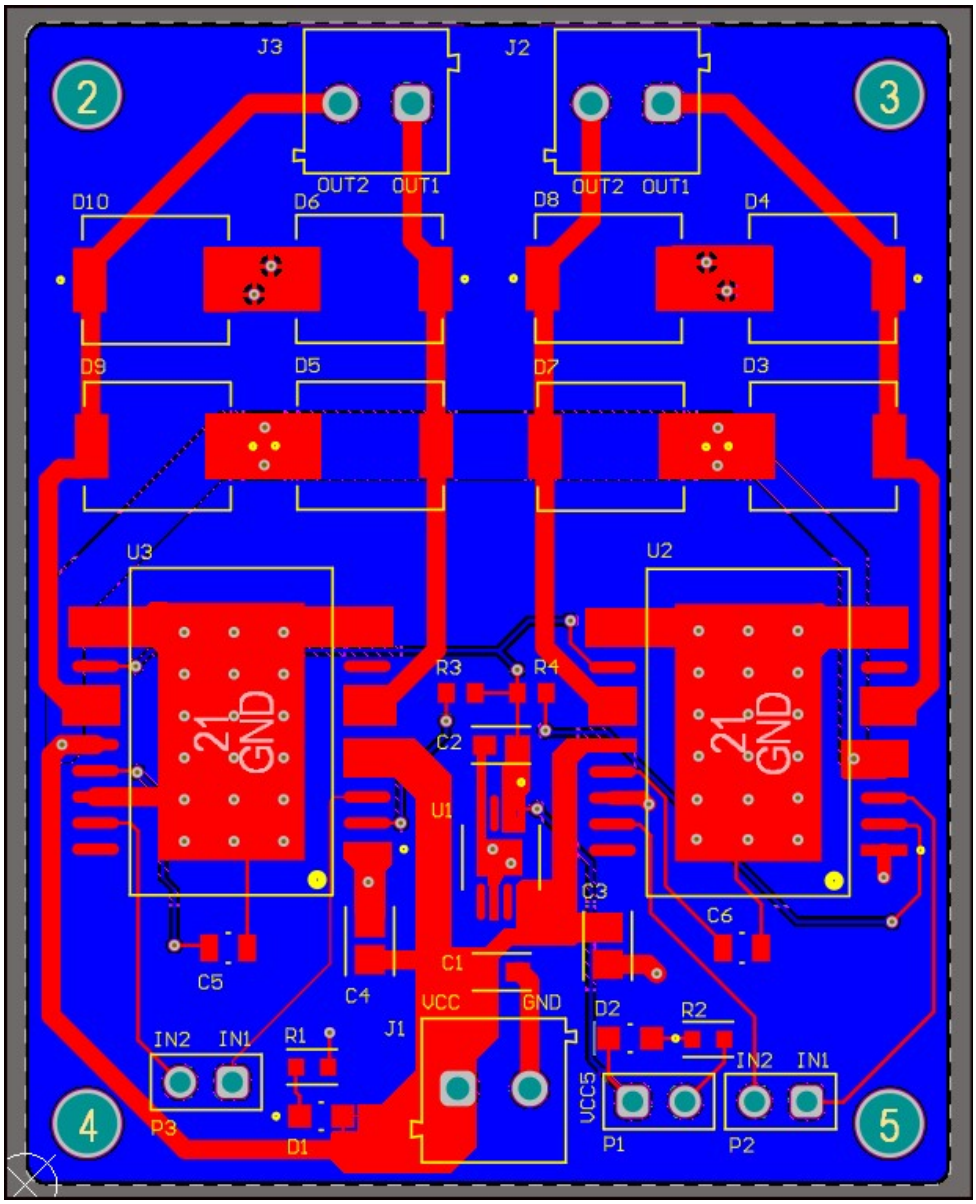


Figure 2.1: Routed PCB

2.3 PCB Routing

Effective routing is a critical aspect of PCB design, directly impacting the performance, reliability, and manufacturability of the dual motor driver circuit. The routing strategy for this PCB was carefully planned to ensure minimal noise, efficient power delivery, and robust signal integrity

1. **Component Placement:** Integrated Circuits (U2, U3) are placed centrally to shorten the length of high-current paths and reduce inductance.Decoupling capacitors (e.g., C1, C2, C3) are placed as close as possible to the power pins of the ICs to filter high-frequency noise and provide stable voltage levels.
2. **Power Traces:** The power traces are routed in the top layer with thicker widths to handle the high current required by the motor drivers. This minimizes voltage drops and ensures efficient power distribution across the PCB.
3. **Ground Plane:** A solid ground plane is used extensively on bottom layer to provide a low-impedance path for return currents and to reduce electromagnetic interference (EMI). The ground plane also aids in heat dissipation, which is crucial for maintaining the thermal stability of the board.
4. **Signal Traces:** Signal traces are kept as short as possible to minimize potential noise and signal degradation. Critical signal paths, especially those controlling the motor drivers, are routed with care to avoid crossing noisy power lines. Adequate spacing between traces is maintained to prevent crosstalk and short circuits.
5. **Thermal Vias:** Thermal vias are used to connect the top and bottom layers, enhancing heat dissipation from components, especially the motor driver ICs (U2, U3), which are likely to generate significant heat during operation.

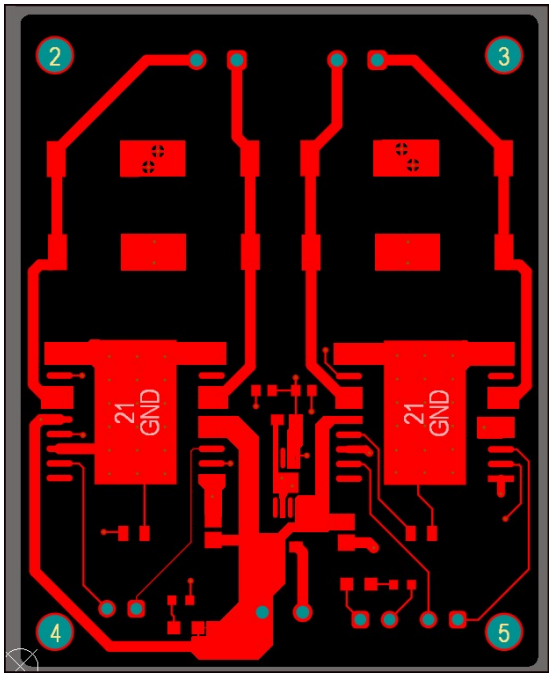


Figure 2.2: Top layer

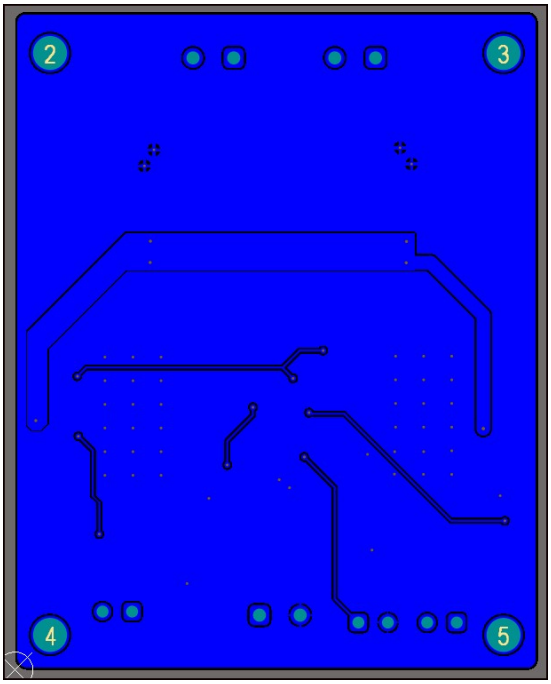


Figure 2.3: Bottom layer

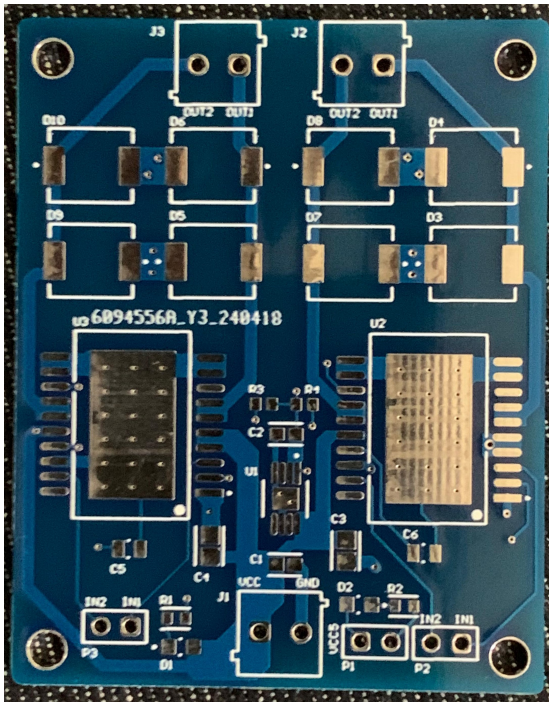


Figure 2.4: Bare PCB Front View

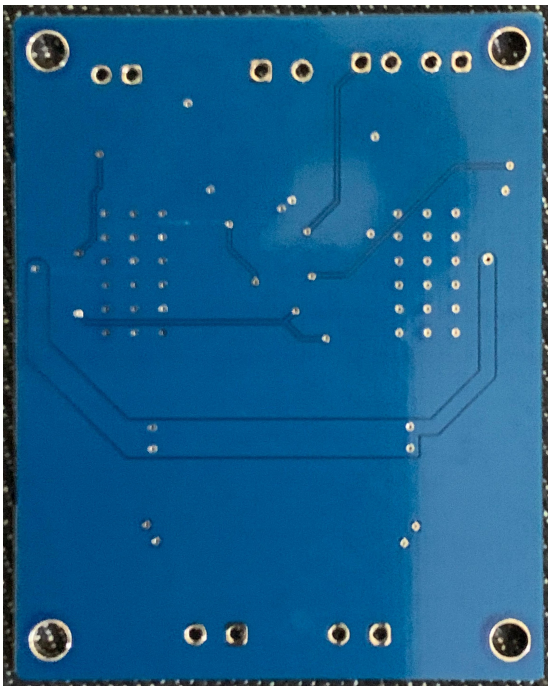


Figure 2.5: Bare PCB Rear View

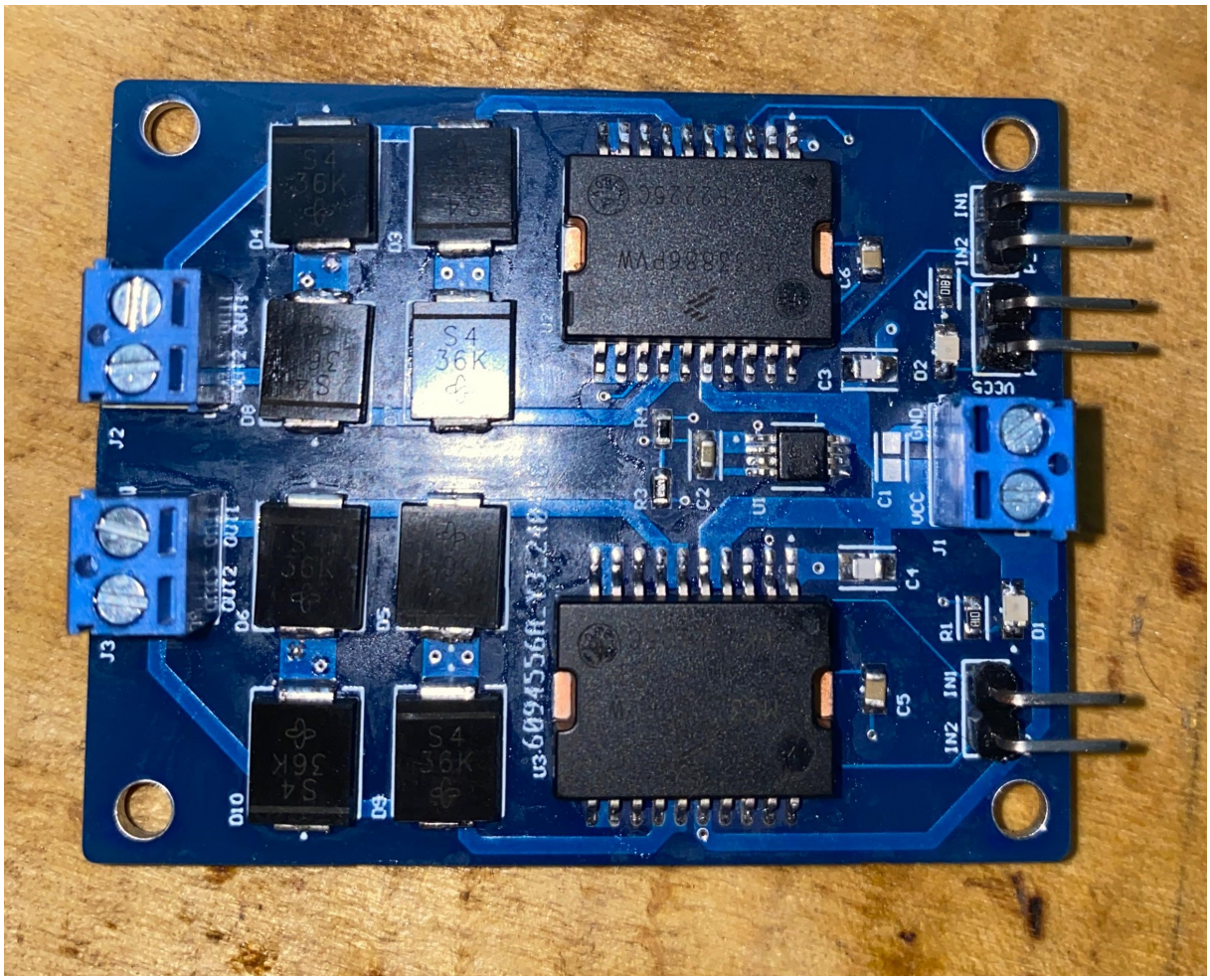
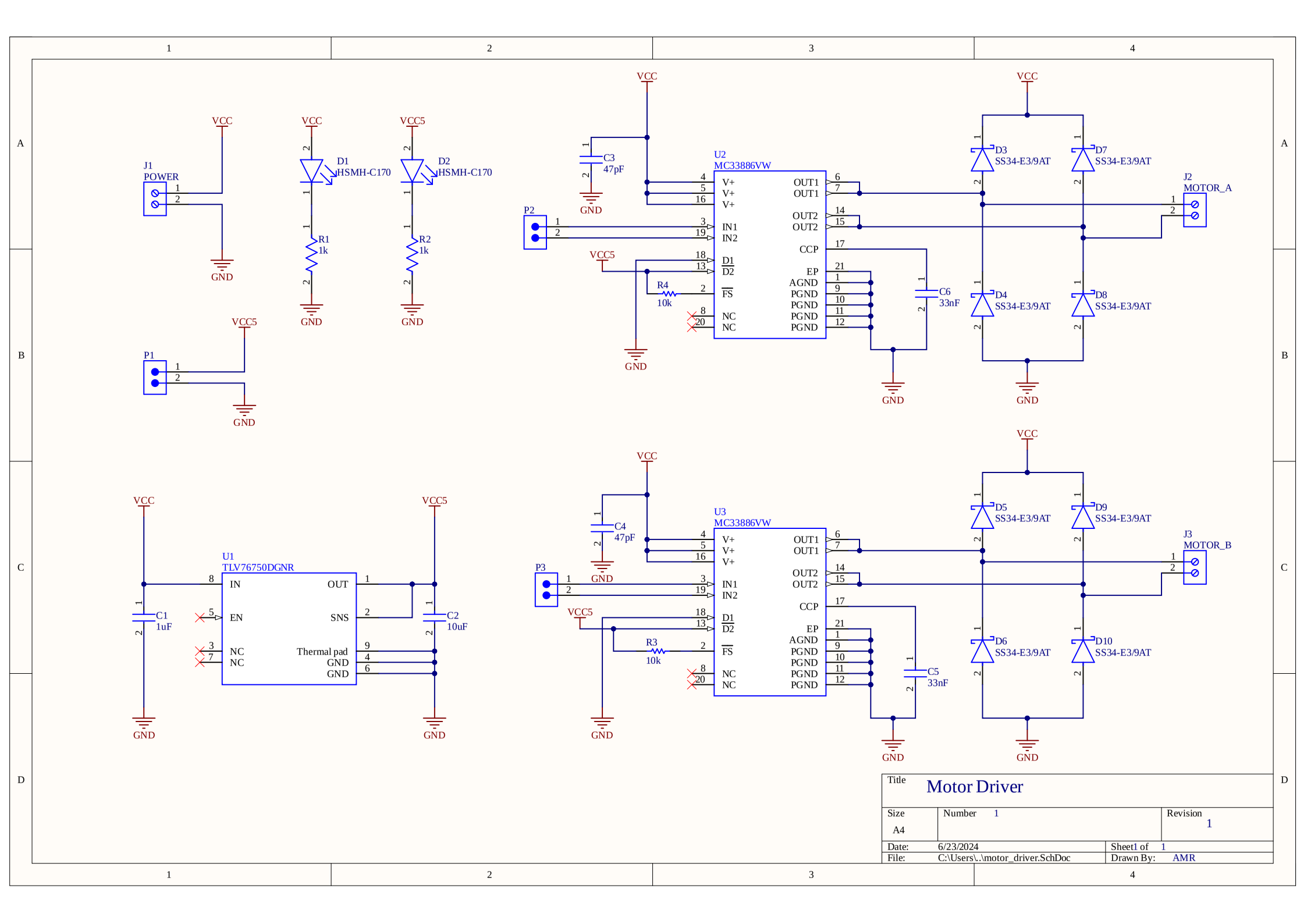


Figure 2.6: Soldered PCB


J1
POWER
VCC
GND
D1
HSMH-C170
R1
1k
D2
HSMH-C170
R2
1k
VCC5
P1
P2
P3
C3
47pF
C4
47pF
U2
MC33886VW
U3
MC33886VW
V+
IN1
IN2
D1
D2
FS
NC
CCP
EP
AGND
PGND
OUT1
OUT2
R4
10k
R3
10k
C6
33nF
C5
33nF
D3
SS34-E3/9AT
D4
SS34-E3/9AT
D7
SS34-E3/9AT
D8
SS34-E3/9AT
D5
SS34-E3/9AT
D6
SS34-E3/9AT
D9
SS34-E3/9AT
D10
SS34-E3/9AT
J2
MOTOR_A
J3
MOTOR_B
U1
TLV76750DGNR
IN
EN
NC
OUT
SNS
Thermal pad
C1
1uF
C2
10uF
Title
Motor Driver
Size
A4
Number
1
Revision
1
Date:
6/23/2024
Sheet1 of
1
File:
C:\Users\..\motor_driver.SchDoc
Drawn By:
AMR

Designator	Description	Manufacturer Part No.
C1	Multilayer Ceramic Capacitors MLCC - SMD/SMT 16 V 0.1uF X7R 0603 10	CC0603KRX7R7BB104
C2	Multilayer Ceramic Capacitors MLCC - SMD/SMT 10 uF 6.3 VDC 20 0603 X5R	GRM188R60J106ME47D
C3, C4	Multilayer Ceramic Capacitors MLCC - SMD/SMT 100V 47pF C0G 0805 5	C0805C470J1GACTU
C5, C6	Multilayer Ceramic Capacitors MLCC - SMD/SMT 25V 0.033uF X7R 0805 10	CC0805KRX7R8BB333
D3 - D10	Schottky Diodes Rectifiers 40 Volt 3.0 Amp 100 Amp IFSM	SS34-E3/9AT
R1, R2	Thick Film Resistors - SMD 1 kOhms 100mW 0603 1	RC0603FR-071KL
R3, R4	Thick Film Resistors - SMD 1/10watt 10Kohms 1	CRCW060310K0FKEA
U1	LDO Voltage Regulators Adjustable- and fixed-output, 1-A, 16-V, positive-voltage low-dropout (LDO) linear regulator 8-HVSSOP -40 to 125	TLV76750DGNR
U2, U3	Motor / Motion / Ignition Controllers Drivers 5.0 A H-BRIDGE	MC33886PVWR2
D1, D2	Standard LEDs - SMD Red Diffused 639nm 17mcd	HSMH-C170

Table 2.1: Motor Driver Components

2.4 PCB Testing

We tested the motor driver by applying different PWM values and recorded the corresponding output voltage.

Testing Setup

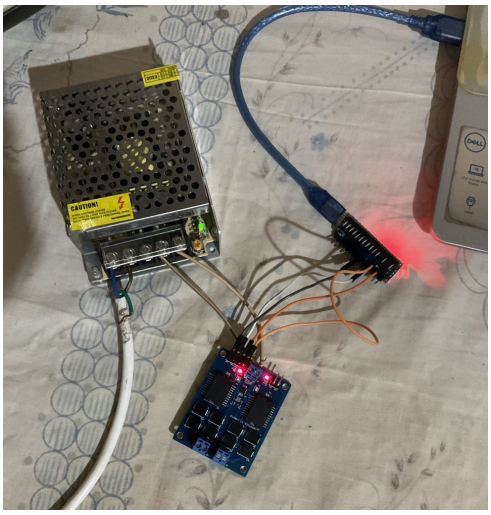


Figure 2.7: Testing Setup

- **Power Supply:** 12V
- **Measurement Tool:** Multimeter
- **Control Signal:** PWM (Pulse Width Modulation)
- **Controller used to give PWM signal:** Arduino Nano

1. **PWM value-50**

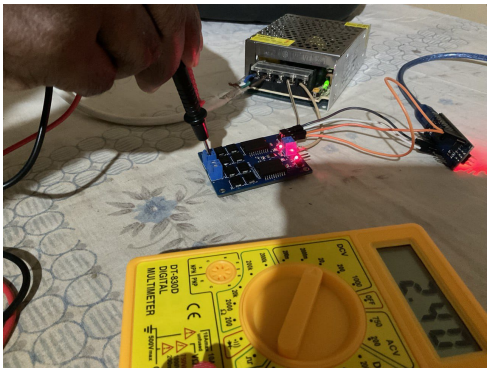


Figure 2.8: PWM = 50

2. **PWM value-100**

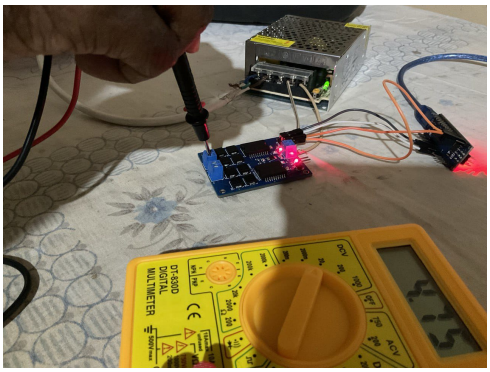


Figure 2.9: PWM = 50

3. **PWM value-150**

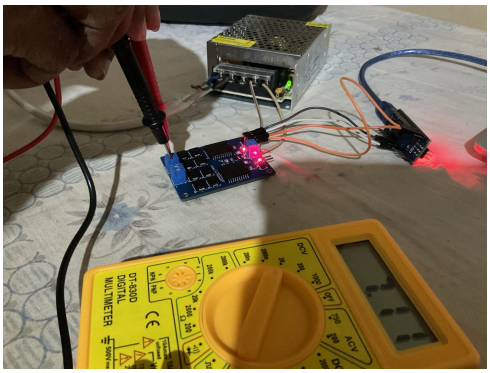


Figure 2.10: PWM = 150

4. **PWM value-200**

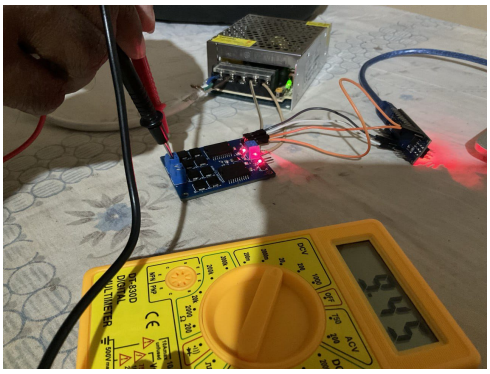


Figure 2.11: PWM = 200

5. **PWM value-250**

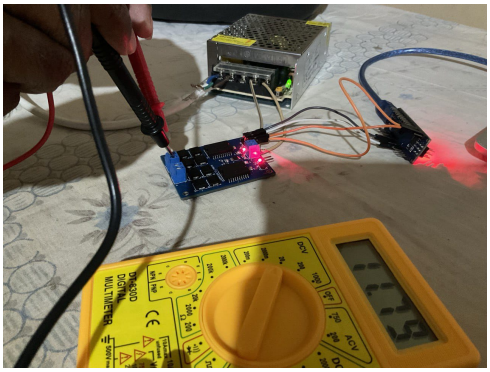


Figure 2.12: PWM = 250

6. **Max Output(PWM = 255)**

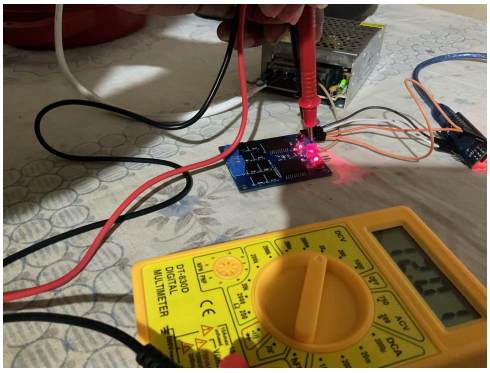


Figure 2.13: Max Output(PWM = 255)

Observation:

As the PWM value increased, the output voltage of the motor driver also increased proportionally. The motor driver responded correctly to the PWM signals, demonstrating its capability to control motor speed efficiently.

The tables below summarizes the results:

Duty cycle	Current	Output Voltage
20	40	2.34
30	52	3.56
40	63	4.76
50	71	5.96
60	75	7.16
70	77	8.36
80	78	9.56
100	84	11.96

Table 2.2: Current and Output Voltage Measurements
IN1 - GND and IN2 - PWM

Duty cycle	Current	Output Voltage
20	39	2.34
30	52	3.55
40	62	4.76
50	70	5.96
60	74	7.16
70	78	8.36
80	78	9.55
100	80	11.96

Table 2.3: Current and Output Voltage Measurements
IN1 - PWM and IN2 - GND

Conclusion:

The motor driver was successfully tested with a 12V power supply. The results show that it performs as expected, with the output voltage varying in accordance with the applied PWM values. This confirms the effectiveness of the motor driver in controlling motor speed. In both configurations, where IN1 is connected to GND and IN2 to PWM, and vice versa, the motor driver exhibits similar behavior in terms of current consumption and output voltage. At each duty cycle from 20% to 100%, the current and output voltage measurements are almost identical. This indicates that the motor driver performs consistently regardless of which pin is connected to PWM or GND, thereby ensuring reliable and predictable operation in either configuration.

Chapter 3

Solidworks Design

3.1 Introduction

For this project, we have designed two solid bodies: one to carry the entire assembly (the platform) and another to enclose the motor driver system. These designs have undergone multiple rounds of refinement and finalization to ensure they meet our needs.

3.2 The Platform

This metal platform is an essential component for the testing and implementation of the AMRC. Once we decided to design the robot platform, we conducted a thorough analysis to ensure we obtained a design suitable for our needs. Following this, we modeled the design using SOLIDWORKS drawings to visualize and refine the structure before fabrication.

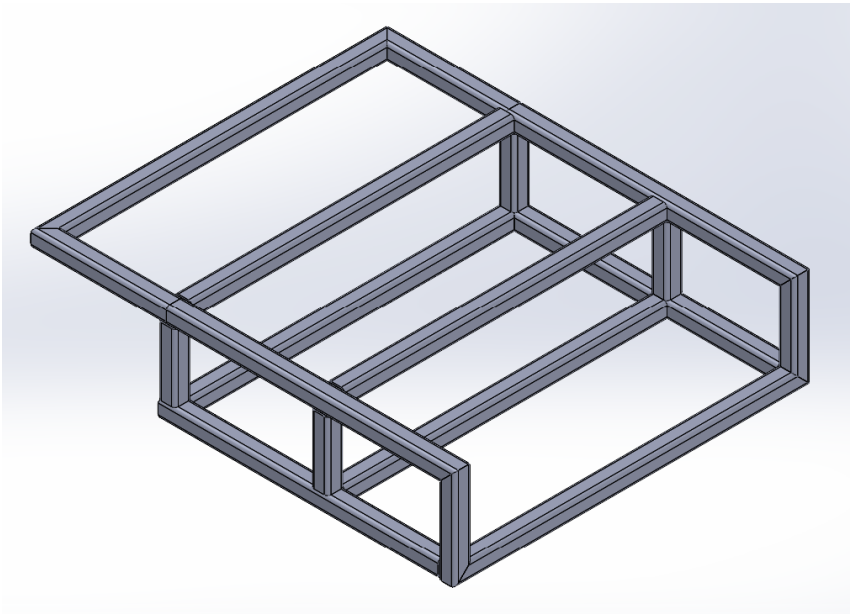


Figure 3.1: Frame

3.2.1 Design Considerations

The platform should be designed to support a laptop and a battery, as these are the heaviest components it will carry. It needs to be compact, ensuring it fits into constrained spaces, and yet stable enough to ensure smooth and reliable operation. To achieve these goals, we meticulously evaluated numerous designs, considering factors such as weight distribution, balance, material strength, and overall stability. After thorough analysis and comparison, we have chosen a design that meets these criteria effectively. This design not only ensures the platform's stability during operation but also maximizes efficiency and durability, providing a reliable solution for our needs.

3.2.2 Construction

We have chosen steel bars (1 cm x 1 cm) to design the platform. Steel bars are capable of supporting our requirements and are cost-effective. Since the steel bars can be welded together, this platform is scalable for future purposes, allowing for easy modifications and expansions as needed.

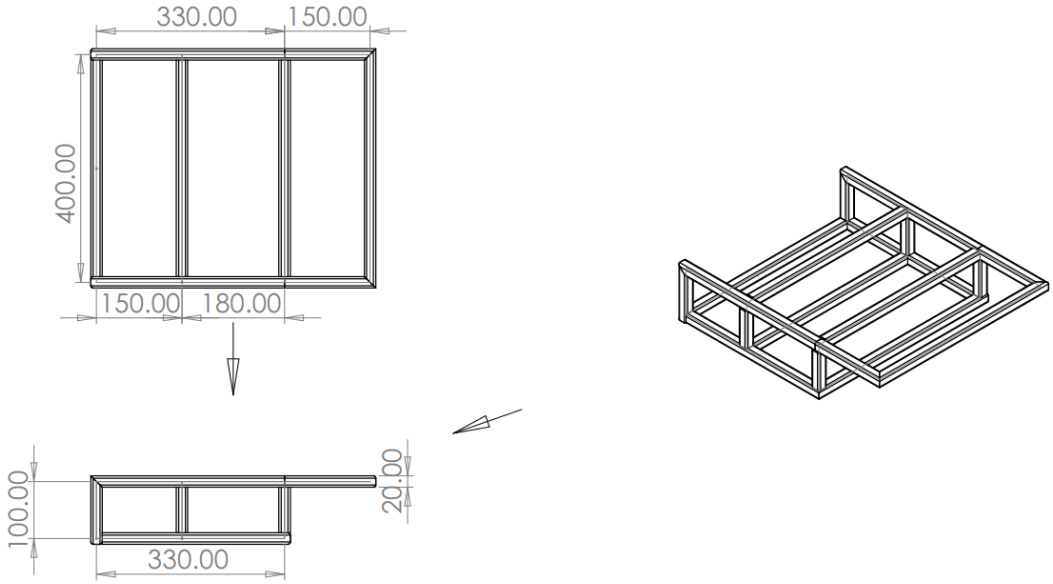


Figure 3.2: Fame Drawings

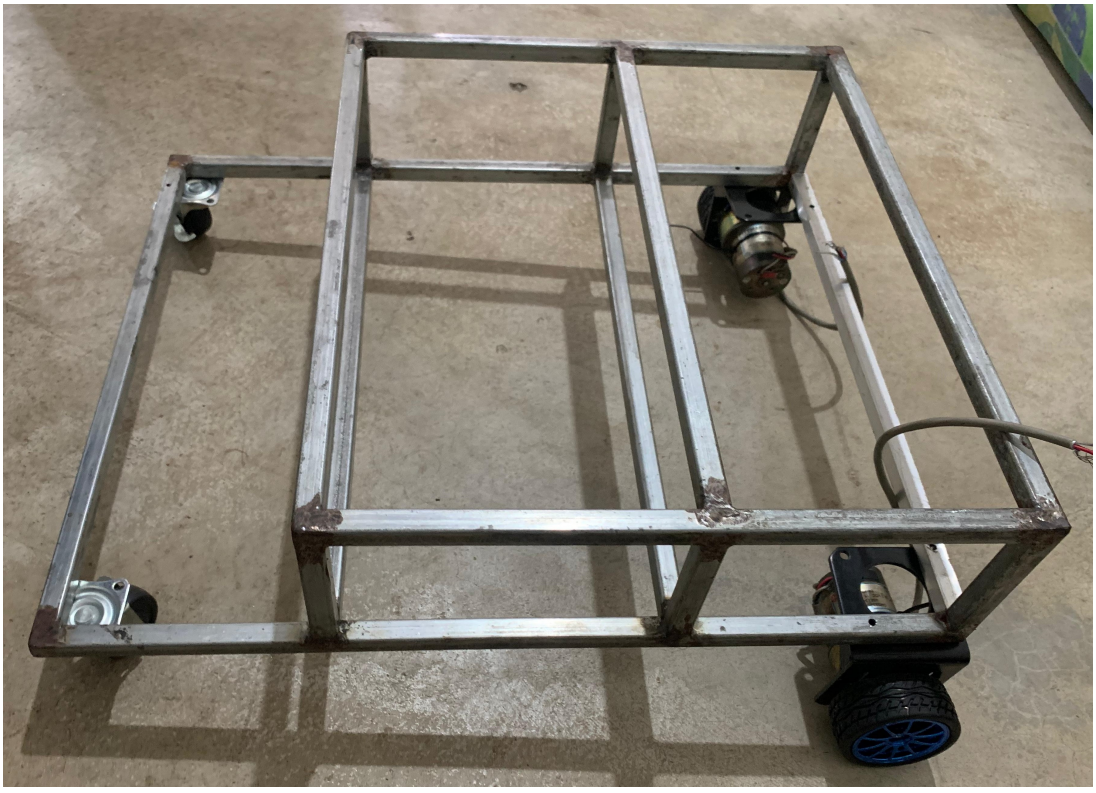


Figure 3.3: Welded Frame

3.3 The Motor Driver Enclosure

The motor driver enclosure serves as a crucial component in our project, providing protection and organization for the intricate electronic components that drive the robot's motors. Here's an overview of its design and functionality:

3.3.1 Design Considerations

- One of the primary purposes of the motor driver enclosure is to shield the delicate electronic components from environmental factors such as dust, moisture, and accidental impact.
- Efficient heat dissipation is essential to prevent overheating of the motor driver system during operation. The enclosure design incorporates ventilation slots to facilitate airflow and cooling.
- The enclosure design must also allow easy access to the motor driver system for maintenance, troubleshooting, and upgrades.
- The enclosure is designed to seamlessly integrate with the overall robot assembly, fitting snugly within the framework while providing sufficient space for wiring and cable management.
- It's also important to ensure that the enclosure can be readily manufactured in large quantities.

3.3.2 Construction

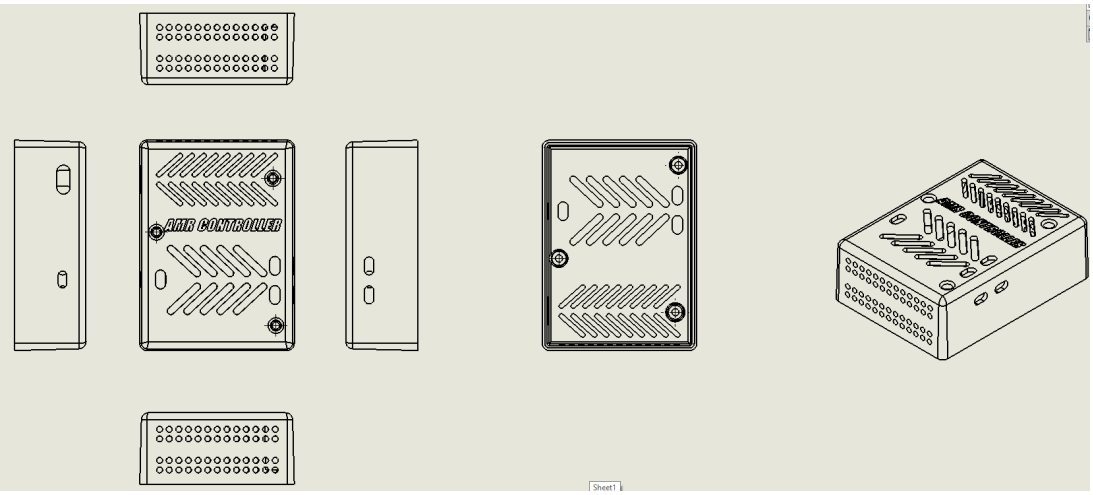


Figure 3.4: Top Enclosure Drawing

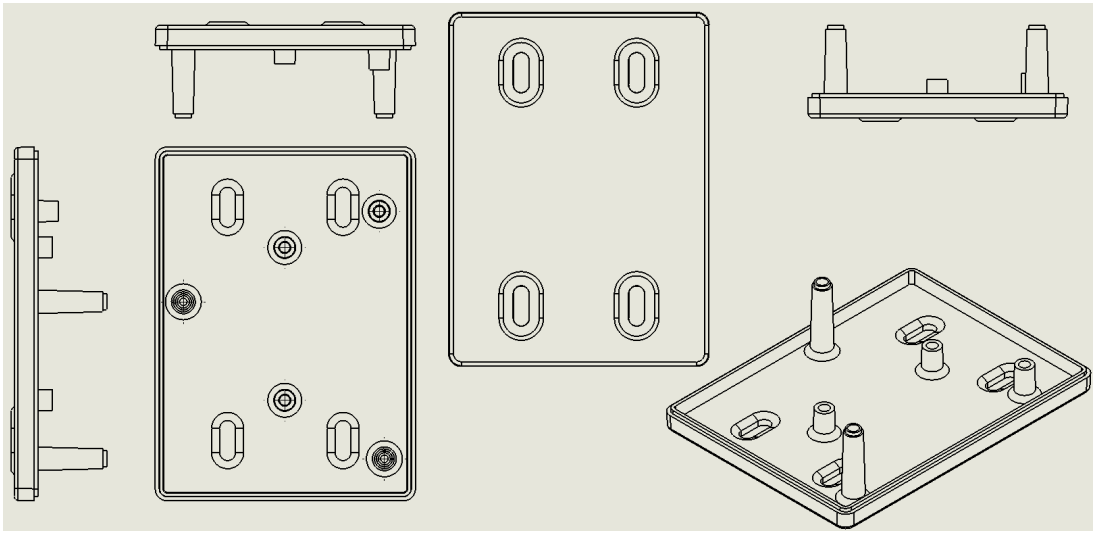


Figure 3.5: Bottom Enclosure Drawing

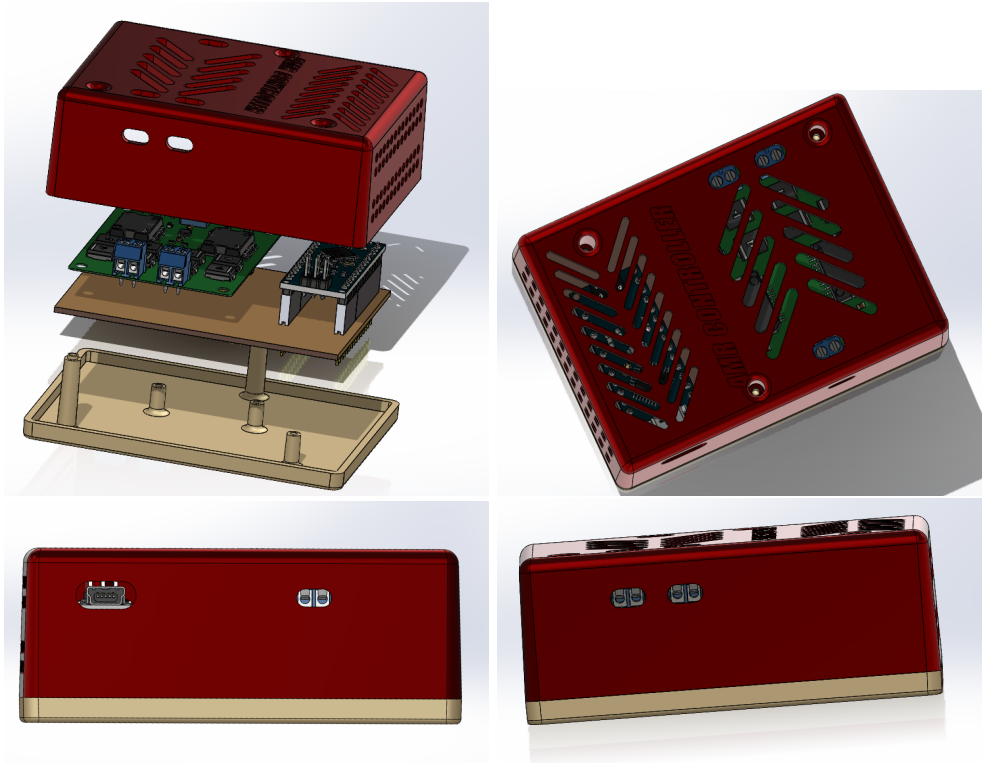


Figure 3.6: Motor Driver Enclosure Design

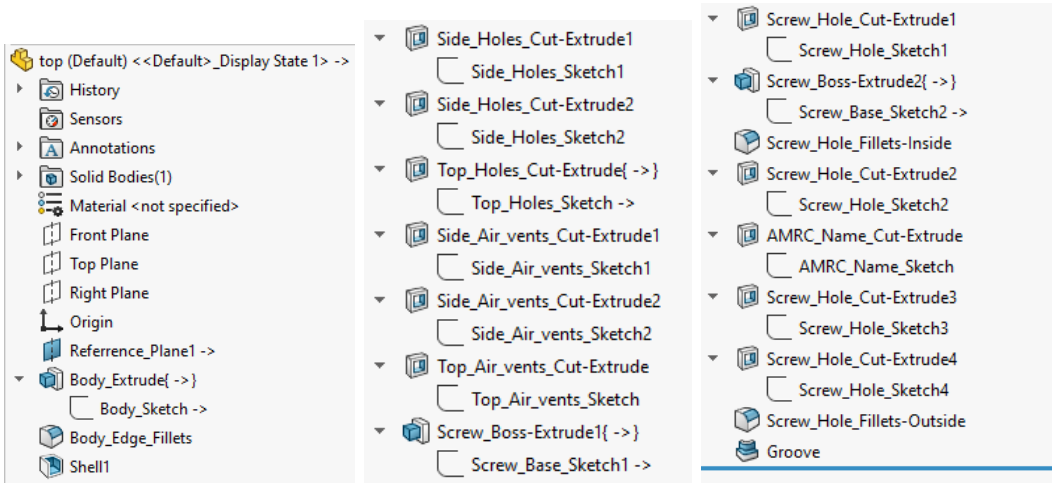


Figure 3.7: Top Part Model Tree

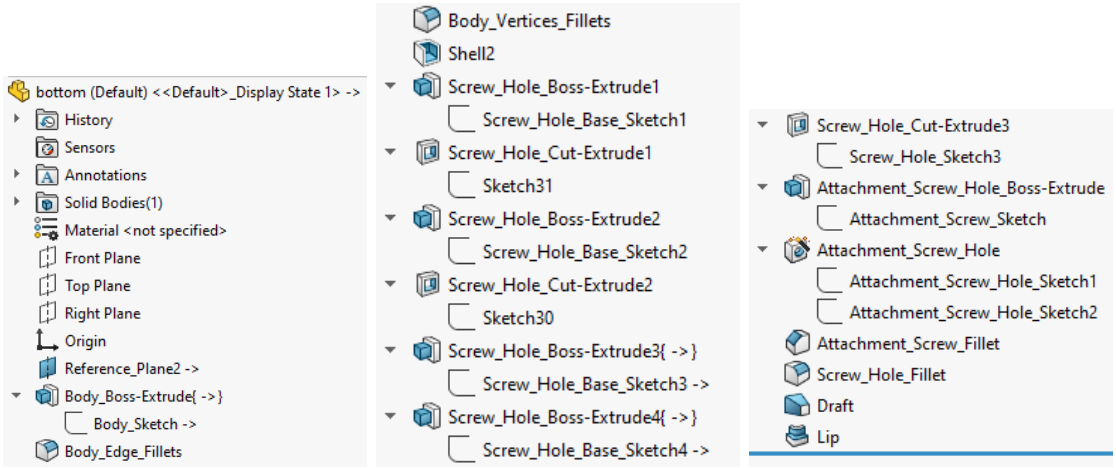
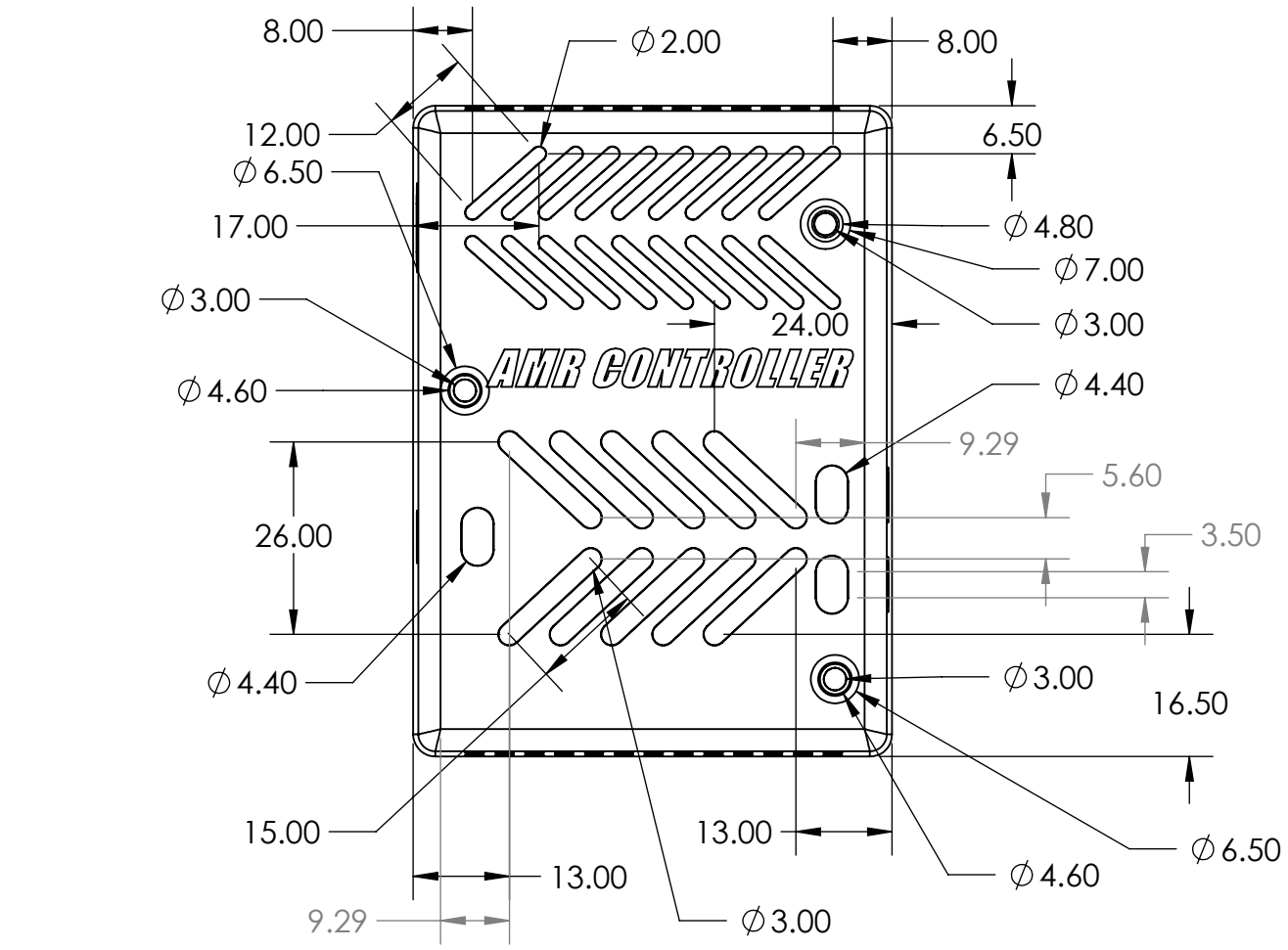


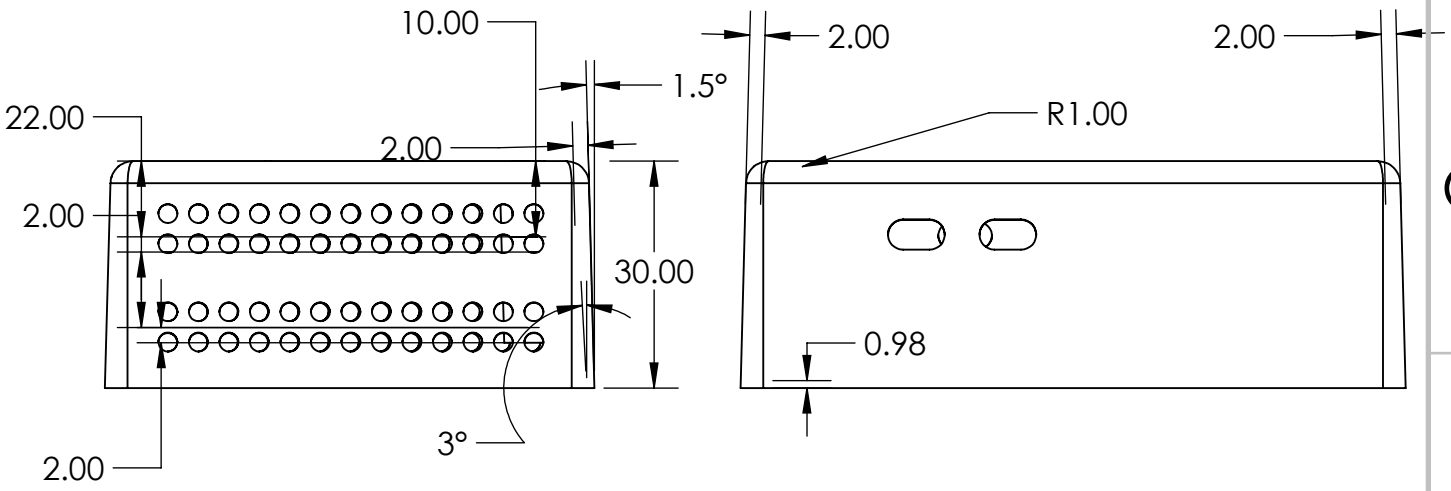
Figure 3.8: Bottom Part Model Tree

The enclosure is constructed using durable material, ABS plastic, chosen for its robustness and suitability for manufacturing processes like injection molding.

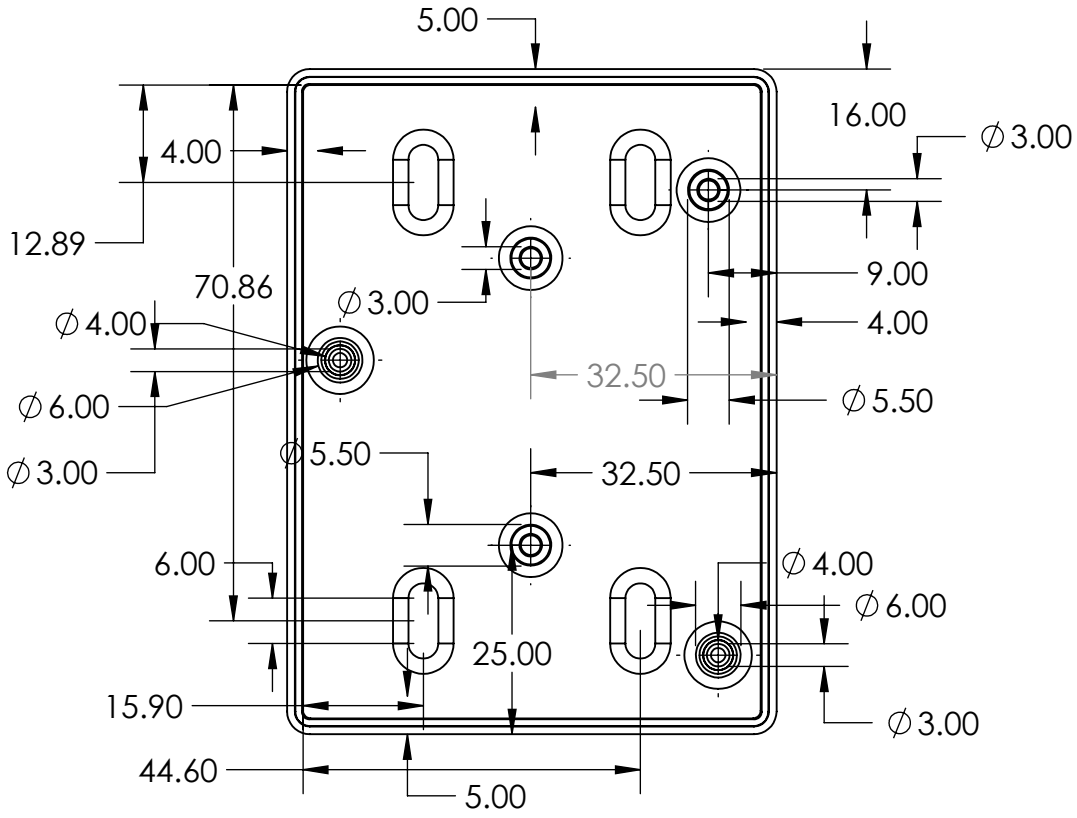
Key features of the enclosure design may include:

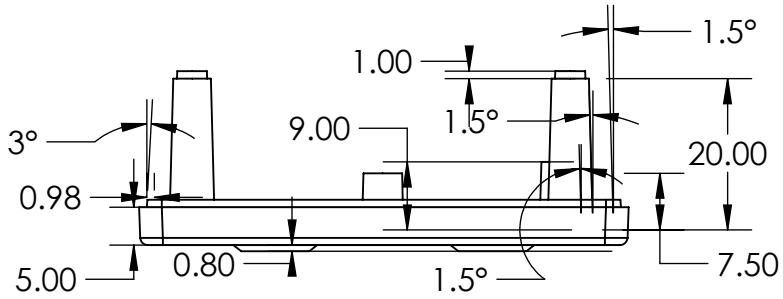
- The enclosure consists of two components that can be assembled and disassembled easily, facilitating manufacturing, assembly, and maintenance processes.
- It includes mounting points to securely fasten the enclosure to the robot platform, ensuring stability and vibration resistance during operation.
- The enclosure design incorporates the necessary space to route and organize cables neatly, reducing the risk of tangling or damage to wiring.

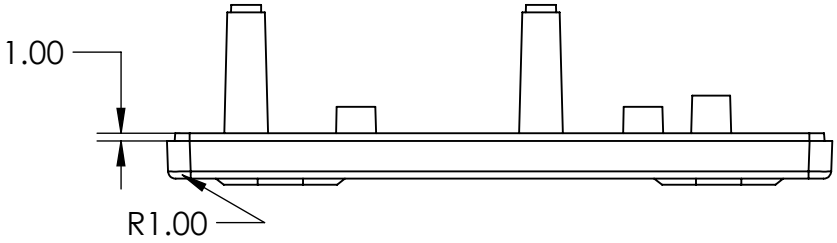




UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH: 06/07/2024		DEBURR AND BREAK SHARP EDGES		DO NOT SCALE DRAWING	REVISION
	NAME	SIGNATURE	DATE				TITLE:
DRAWN	Abithan A.		04/05/2024				Top Part
CHK'D							
APPV'D							
MFG							
Q.A				MATERIAL:		DWG NO.	A4
				ABS		01	
				WEIGHT:		SCALE: 1:1	SHEET 1 OF 1







UNLESS OTHERWISE SPECIFIED: DIMENSIONS ARE IN MILLIMETERS SURFACE FINISH: TOLERANCES: LINEAR: ANGULAR:		FINISH:		DEBURR AND BREAK SHARP EDGES		DO NOT SCALE DRAWING	REVISION
		06/07/2024					
	NAME	SIGNATURE	DATE			TITLE:	
DRAWN	Abithan A.		04/05/2024			Bottom Part	
CHK'D							
APPV'D							
MFG							
Q.A				MATERIAL:		DWG NO.	A4
				ABS		02	
				WEIGHT:		SCALE: 1:1	SHEET 1 OF 1

3.3.3 Mold Design

The mold design is tailored to ensure precise and consistent manufacturing, meeting all specifications and quality standards. It incorporates features that facilitate efficient molding processes, reduce production time, and enhance the structural integrity of the enclosure. This comprehensive approach ensures that the final product is both robust and reliable, maintaining the necessary standards for housing sensitive electronic components.

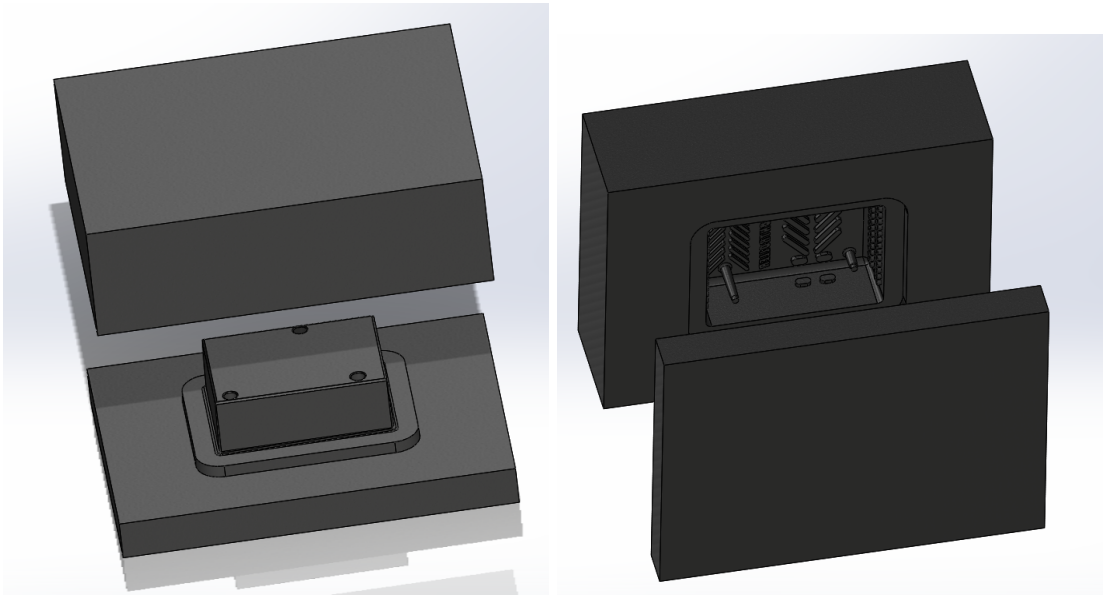


Figure 3.9: Top Enclosure Mold Design

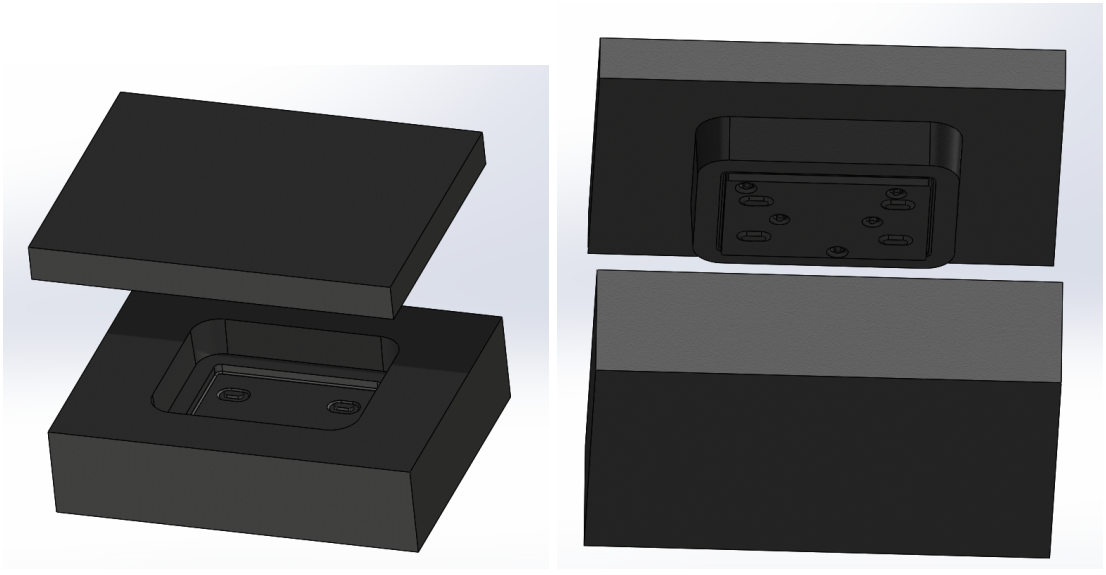


Figure 3.10: Bottom Enclosure Mold Design

Figure 3.11: Enclosure Exterior View

Figure 3.12: Enclosure Interior View

Chapter 4

Hardware Implementation

4.1 Power consumption and Power Supply

To ensure our robot operates effectively, we need to carefully consider the power consumption of its components and choose an appropriate power supply. Below is the breakdown of the power requirements for each major component:

1. **Motor**
    - Maximum Current Consumption of a Motor at 12V = 1A
    - Number of Motors = 2
    - Total Current Consumption for Motors = 2×1A=2A
2. **Arduino Nano**
    - Max Current consumption of Arduino nano = 200mA
3. **LiDAR Sensor**
    - Expected Current consumption of LiDAR Sensor at 12V = 2.5A

Considering all these components, the total current consumption can be calculated as follows:

$$\textit{Total Current Consumption} = 2A + 0.2A + 2.5A = 4.7A$$

To power our robot, we need a battery that can supply at least 4.7A at 12V. A motorcycle battery is a suitable choice for this purpose as it can provide the required current rating efficiently. Therefore, we are using a motorcycle battery to ensure our robot has a reliable power supply.

Additionally, it is important to note that the laptop used for controlling the robot can run on its own battery backup. This ensures that the laptop's power consumption does not add to the overall power requirements of the robot, thus simplifying the power management of the robotic system.

By selecting a motorcycle battery, we ensure that our robot has a stable and adequate power source to operate all its components efficiently in the long run, while the laptop remains independent with its own power supply.

4.2 Connecting Arduino Nano with Motor Drivers

In the implementation of the Automatic Mobile Robot Controller, the Arduino Nano serves as the intermediary between the ROS2-controlled software environment and the physical motor drivers that power the robot's movement. This section details the process of connecting the Arduino Nano to the motor drivers using the MC33886PVWR2 motor driver IC, ensuring successful translation of software commands into mechanical actions.

4.2.1 Hardware Configuration

The Arduino Nano is connected to the MC33886PVWR2 motor driver IC, which is capable of controlling up to two motors. Here's how these components are interconnected:

Motor Driver Pins: The motor driver IC, MC33886PVWR2, has several pins for controlling the motors. For each motor, two pins are used for PWM signal input, which controls the speed and direction of the motors.

Arduino to Motor Driver Connection

Motor 1:

- IN1_Motor1 (Pin 6 on Arduino) is connected to the first control input of the motor driver for Motor 1.
- IN2_Motor1 (Pin 2 on Arduino) is connected to the second control input of the motor driver for Motor 1.

Motor 2:

- IN1_Motor2 (Pin 5 on Arduino) is connected to the first control input of the motor driver for Motor 2.
- IN2_Motor2 (Pin 3 on Arduino) is connected to the second control input of the motor driver for Motor 2.

Chapter 5

Software Details

5.1 Introduction

5.1.1 Project Overview

The "Automatic Mobile Robot Controller" (AMR Controller) is decided to be designed to offer seamless integration with any sensor, motor, or actuator. Its primary purpose is to provide robust SLAM (Simultaneous Localization and Mapping) and navigation capabilities in industrial environments such as warehouses. The AMR Controller enables automated mobile robots to map environments using LiDAR sensors and autonomously navigate to specified goals while avoiding obstacles.

5.1.2 Objective of the Documentation

This document provides a comprehensive guide to the software implementation of the AMR Controller. It contains all necessary software details, troubleshooting steps, and other essential information to empower others to implement and maintain this solution independently. Additionally, this documentation will serve as a valuable resource for anyone continuing the project's development in the future.

5.1.3 Target Audience

This documentation targets the industrial sector, specifically warehouse-level industries in Sri Lanka. The AMR Controller aims to improve operational efficiency and automate critical workflows for these industries.

5.1.4 Key Features

1. **SLAM and Navigation:** Environment Mapping: Leverages a LiDAR sensor to map the surrounding environment. (In our project we use the LiDAR sensor from the Gazebo simulated environment)
2. **Autonomous Navigation:** Safely navigates to user-specified goals, avoiding barriers and obstacles in real-time.

5.2 System Requirements

5.2.1 Hardware Requirements

For the "Automatic Mobile Robot Controller" software to operate optimally on Ubuntu Linux - Jammy Jellyfish (22.04), ensure the system meets the following hardware specifications:

Minimum Requirements

RAM: 2GB
Disk Space: 25GB
Processor: 64-bit

Recommended Requirements

RAM: 4GB or more
Disk Space: 40GB or more
Processor: 64-bit

Arduino Nano

Arduino Nano is also used in our project as a interconnection between Ubuntu System and the motor driver.

5.2.2 Software Requirements

Operating System

Ubuntu Linux - Jammy Jellyfish (22.04): The desktop image can be downloaded from the Official Ubuntu web page (https://releases.ubuntu.com/jammy/) and install it following Ubuntu's standard installation guidelines. Note that ROS2 Humble is only supported on this distribution.

Robot Operating System (ROS2) Distribution

ROS2 Humble: This particular version is selected for simulation, SLAM, and navigation purposes. Although other distributions are supported in ROS2, Humble is purposefully chosen for this project due to its comprehensive features.

Alternative Platforms

ROS2 can also be installed on other platforms too. More details about other installations platforms can be found on the official ROS2 documentation here.https://docs.ros.org/en/humble/Installation.html

AVR Studio

In order to upload necessary codes to Arduino Nano hardware, AVR Studio is needed. It can be downloaded by following the instructions from here.https://www.microchip.com/en-us/tools-resources/archives/avr-sam-mcus

5.3 Initial Setup

5.3.1 Software Installation

ROS2 Humble Installation

As previously stated, the ROS2 Humble distribution was selected and installed on Ubuntu Linux - Jammy Jellyfish (22.04) using Debian packages, which is the recommended approach. Follow the steps below to install ROS2 Humble and set up the development environment:

Step 1: Set Locale

First, open a terminal on Ubuntu and ensure that your system locale supports UTF-8. You can verify and modify this setting using the following commands:

```
locale  # check for UTF-8

sudo apt update && sudo apt install locales
sudo locale-gen en_US en_US.UTF-8
sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8
export LANG=en_US.UTF-8

locale  # verify settings
```

Step 2: Set Up Sources

Add the ROS2 apt repository to the system by enabling the Ubuntu Universe repository first:

```
sudo apt install software-properties-common
sudo add-apt-repository universe
```

Now, add the ROS2 GPG key to the apt package manager:

```
sudo apt update && sudo apt install curl -y
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key \
-o /usr/share/keyrings/ros-archive-keyring.gpg
```

Then, add the ROS2 repository to the sources list:

```
echo "deb [arch=$(dpkg --print-architecture) \
signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] \
http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo \
$UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list > \
/dev/null
```

Step 3: Update and Upgrade Packages

After setting up the repository, update and upgrade the apt package list:

```
sudo apt update
```

```
sudo apt upgrade
```

Step 4: Install ROS2 Humble Desktop Version

With the repositories properly configured, install the ROS2 Humble desktop version:

```
sudo apt install ros-humble-desktop
```

Install Development Tools

Lastly, ensure that the necessary development tools, compilers, and utilities are installed for building ROS2 packages:

```
sudo apt install ros-dev-tools
```

With ROS2 Humble installed and the development tools in place, our environment is ready to support simulation and control functions for the Automatic Mobile Robot Controller.

For any more details about the ROS2 installtion refer here.

Using colcon to build packages

colcon is an iteration on the ROS build tools catkin_make, catkin_make_isolated, catkin_tools and ament_tools. We are using colcon to build our ros2 packages here.

Install Colcon

```
sudo apt install python3-colcon-common-extensions
```

5.3.2 Dependencies and Packages Installation

The following packages are needed for our project

- ros-humble-gazebo-ros-pkgs
- ros-humble-xacro
- ros-humble-navigation2
- ros-humble-nav2-bringup
- ros-humble-twist-mux
- ros-humble-turtlebot3*

The above packages can be installed by executing the following command.

```
sudo apt install ros-humble-navigation2 ros-humble-nav2-bringup \
ros-humble-twist-mux ros-humble-turtlebot3* ros-humble-gazebo-ros-pkgs \
ros-humble-xacro
```

5.3.3 Setting Up Development Environment

Background

The Robot Operating System 2 (ROS2) employs the concept of workspaces managed via the shell environment. A workspace is a location on the system where development activities related to ROS2 take place. The primary or "core" ROS2 workspace is known as the underlay, while subsequent local workspaces that extend it are called overlays. When developing with ROS2, it's common to have multiple workspaces concurrently active.

Combining these workspaces simplifies the process of working with different versions or package sets of ROS2. This feature makes it possible to install several ROS2 distributions (e.g., Dashing, Eloquent) on the same system and switch between them seamlessly.

Access to ROS2 commands, packages, and resources depends on sourcing setup files, which must be done every time a new shell is opened. Alternatively, sourcing can be permanently added to the shell startup script to simplify this process.

Step 1: Source the Setup Files

Ensure ROS2 Humble is already installed before proceeding. To access ROS2 commands in every new shell session, execute the following command

```
# Replace ".bash" with your shell if you're not using bash.
# Possible values are: setup.bash, setup.sh, setup.zsh
source /opt/ros/humble/setup.bash
```

Alternative method for the Automatic Mobile Robot Controller project, sourcing the setup file automatically at startup is preferable, given the use of only the Humble distribution. This can be achieved by adding the source command to the shell startup script:

```
echo "source /opt/ros/humble/setup.bash" >> ~/.bashrc
```

Make sure to adjust the sourcing command when using a shell other than bash (e.g., setup.zsh for Zsh users).

Step 2: Create an Overlay Workspace

An overlay workspace allows developers to add new packages without interfering with the core ROS2 workspace (underlay). The underlay workspace must contain dependencies for all the packages in the overlay, and any packages in the overlay will override those in the underlay.

This layered approach enables developers to maintain multiple overlays on top of one or more underlays.

To set up the overlay workspace for this project:

Create a new directory named amr_ws and a src directory within it:

```
mkdir -p ~/amr_ws/src
```

The src directory is where packages related to our projects will be placed within the overlay workspace.

Navigate to the amr_ws directory:

```
cd ~/amr_ws
```

Build the workspace with the following command:

```
colcon build --symlink-install
```

The –symlink-install option prevents unnecessary rebuilding by only recompiling packages when new dependencies, files, or packages are added to the workspace. **Whenever new dependencies, files, or packages are added to the workspace worksapce should be rebuilt using above command.**

Verify the build was successful. If so, the following message will be displayed:

```
Summary: 0 packages finished [5.58s]
```

This command also creates three directories: build, install, and log.

Finally, remember to source the overlay workspace whenever using packages from this workspace. This is done by executing the following command from within the amr_ws directory **whenever opening a new terminal.**

```
source install/setup.bash
```

With the overlay workspace set up and sourced, packages and resources from the overlay will be accessible for development. For more details and explanations about overlay workspace refer here. https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Creating-A-Workspace/Creating-A-Workspace.html

Step 2: Create a new package

A package is a collection of related files, including nodes, launch files, and documentation. Packages also list dependencies on other packages. For instance, a scanner driver package might contain a node for laser communication and a launch file to run the node with common parameters.

To create a new package called "my_bot" in the workspace , execute the following command in src directory.

```
ros2 pkg create --build-type ament_cmake my_bot
```

5.4 ROS Basics

5.4.1 ROS Distributions

ROS, which stands for Robot Operating System, is not actually an OS but a set of common libraries and tools for building complex robotic systems. New versions of ROS, called distributions, are released annually and are designed to run on the latest LTS version of Ubuntu. While ROS can run on other Linux distributions and even Mac and Windows, beginners should start with Ubuntu for the best support. ROS2 is the current version, following a major upgrade from ROS1. Although many concepts have remained the same between ROS1 and ROS2, the actual programs and commands differ.

5.4.2 Nodes

A ROS system is composed of numerous smaller programs called nodes, each designed to perform a specific task. Nodes can read data from sensors, send control signals to motors, receive operator controls, display visualizations, or compute trajectories. These nodes form a ROS network, allowing for distributed systems across multiple machines. Key commands for working with nodes include running a node and listing running nodes.

5.4.3 Topics & Messages

Nodes communicate using topics and messages. A topic is a named location where nodes (publishers) can publish messages, and other nodes (subscribers) can receive these messages. This system allows for modular and flexible communication between nodes.

5.4.4 Services

Services in ROS provide a request-reply communication model, useful for tasks such as stopping a motor to save battery. Service commands include listing available services and manually calling a service.

5.4.5 Parameters & Remapping

Parameters allow users to change the behavior of a node, while remapping enables renaming of topics, services, and other named entities. This is particularly useful for avoiding naming conflicts and managing multiple sensors.

5.4.6 Launching

Launch files in ROS facilitate the execution of multiple nodes with specified parameters and remappings. In ROS2, launch scripts are written using Python for greater power and flexibility.

5.4.7 System Packages (Underlay)

ROS packages are installed in a directory within `/opt/ros`, with a `setup.bash` file to add the ROS directory to the system paths. This base installation, known as the underlay, is supplemented by user-created packages in a workspace.

5.4.8 Building a Workspace (Overlay)

A ROS workspace contains user-created packages. The build tool `colcon` handles building and installing these packages, creating a `setup.bash` file for the workspace. This workspace, known as the overlay, is sourced to make its packages available to the ROS system.

5.4.9 Quality of Service (QoS)

QoS in ROS2 allows publishers and subscribers to agree on communication parameters. Incompatibilities in QoS settings can cause communication issues, making it essential to understand and configure these parameters correctly.

5.5 ROS Transform System(TF)

5.5.1 Transforms Overview

Transforms help us understand the relationship between different coordinate frames in a robotic system. For example, if we have multiple robots exploring a room, one robot might find an object of interest and determine its location relative to itself. However, to understand where that object is within the overall room or relative to another robot, we need to use transforms.

5.5.2 Coordinate Frames

To use transforms, we first assign coordinate reference points, called frames, to various elements such as the world, robots, and sensors. In ROS, the convention is to use red for the X direction, green for the Y direction, and blue for the Z direction.

5.5.3 Creating Transforms

A transform defines how to change from one frame to another. These transforms can be used bidirectionally, allowing us to convert coordinates from one frame to another as long as there is a continuous line of transforms between them. To avoid conflicts, transforms should form a tree structure, meaning each frame is defined with respect to only one other frame, without any closed loops.

5.5.4 The TF2 Libraries

The ROS transform system, called tf2 (Transform Version 2), allows nodes to broadcast and listen to transforms. By broadcasting transforms, nodes can create a transform tree, enabling conversion between any frames within the tree.

5.5.5 Static and Dynamic Transforms

Transforms can be static (unchanging over time) or dynamic (changing over time). Static transforms are assumed to be always correct once created, while dynamic transforms need regular updates. Broadcasting nodes must keep dynamic transforms up-to-date, and listeners can check the recency of the data to ensure accuracy.

5.5.6 Broadcasting Transforms

Nodes can use tf2 libraries to broadcast transforms. ROS provides built-in nodes for common broadcasting tasks, such as the static transform publisher, which broadcasts static transforms. This is useful for learning transforms, quick prototypes, or acting as a glue in larger systems.

5.5.7 Visualizing Transforms with RViz

RViz is a ROS visualization tool that can display different types of data, including transform data. To visualize transforms, we add the TF display in RViz and set the reference frame to the appropriate frame (e.g., world). RViz will then show the frames and transforms, allowing us to see the relationships between different elements.

5.5.8 Broadcasting Dynamic Transforms

To broadcast dynamic transforms, we need a URDF (Unified Robot Description Format) file that specifies the physical characteristics of the robot's components. The robot_state_publisher node reads the URDF file and broadcasts the transforms for the robot's joints. It subscribes to a joint_states topic for dynamic joints, publishing their positions, velocities, or efforts. es.

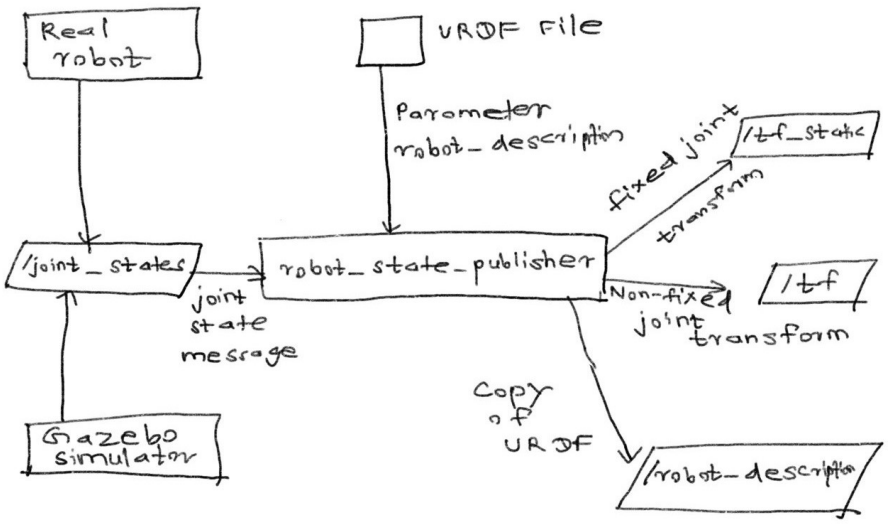


Figure 5.1: Nodes and Topics related to TF

5.5.9 Debugging with view_frames

The view_frames tool helps debug transform issues by generating a PDF file that visually represents the transform tree. This can be useful for understanding the relationships between frames and ensuring that all transforms are correctly defined.

The ROS transform system is a powerful tool for managing coordinate frames and transforms in robotic systems. By leveraging tf2 libraries and tools like RViz and view_frames, we can simplify the process of handling transforms, allowing us to focus on higher-level problem-solving in robotics. Whether working with mobile robots, manipulators, or simulations, the transform system is an essential part of the ROS toolkit.

5.6 Describing a robot using URDF

5.6.1 Introduction

When designing robotic systems, it is beneficial to have a common language to describe the robot. Different software components often need to know about the physical characteristics of the robot, and maintaining this information in a single location ensures simplicity and consistency. In ROS, this information is stored in a format called URDF, which stands for Unified Robot Description Format.

5.6.2 Links and Joints

Breaking Down the Robot

The first step in describing a robot using URDF is to break down its physical structure into separate components called links. Each link represents a distinct part of the robot, such as the base, arms, sensors, or any removable modules.

- Independent Motion: If two parts can move independently, they need to be separate links.
- Logical Separation: If it makes logical sense to separate two parts, such as a sensor that can be removed, they should be different links.

5.6.3 Defining Links

We choose the origin for each link's coordinate system, typically placing it at the pivot point for rotating links.

Example Links

- Base Link
- Slider Link
- Arm Link
- Camera Link

Each link is joined to another through joints, which define their spatial relationship, including position and rotation.

Joints and Motion Types

Joints specify how links are connected and their type of motion. The four common types of joints are:

- Revolute: Rotational motion with fixed start and stop angles (e.g., robotic arms).
- Continuous: Rotational motion without fixed limits (e.g., wheels).

- Prismatic: Linear translational motion (e.g., linear actuators).
- Fixed: No relative motion between parent and child links.

Joint Characteristics

Each joint needs:

- Name: A unique identifier.
- Type: One of the motion types.
- Parent and Child Links: Defines the relationship between the two links.
- Origin: Initial position and orientation before any motion is applied.
- Axis and Limits (for non-fixed joints): Specifies the axis of motion and physical limitations.

5.6.4 URDF Syntax

URDF is based on XML, with each file starting with an XML declaration and containing a root tag, typically the `<robot>` tag. Inside this, we have `<link>` and `<joint>` tags.

Link Tag

A `<link>` tag represents one link and can specify:

- Visual Properties: Geometry, origin, and material for visualization in RViz and Gazebo.
- Collision Properties: Geometry and origin used for collision detection.
- Inertial Properties: Mass, center of mass, and inertia matrix for physics calculations.

Joint Tag

A `<joint>` tag specifies a joint between two links and includes:

- Name
- Type
- Parent and Child Links
- Origin
- Axis and Limits (for movable joints)

5.6.5 Using Xacro

Xacro (XML Macros) is a tool to simplify URDF files by allowing the inclusion of macros, properties, and mathematical operations.

5.7 Simulations using Gazebo and RViz

5.7.1 Introduction

Simulations are one of the most interesting and essential parts of robotic development. A robust simulation environment is invaluable, as it allows for testing algorithms safely and cost-effectively without risking real hardware. Gazebo, a free robotic simulation environment developed by Open

Robotics, integrates seamlessly with ROS, enabling developers to simulate robots, sensors, and environments accurately. This section will explore how to use Gazebo, simulate a virtual robot, and integrate it with ROS for comprehensive testing and development.

5.7.2 Gazebo Environment Basics

Gazebo provides a 3D simulation environment where we can interact with and manipulate models.

Gazebo's world file format (SDF) describes the environment and models within it. Models can be individual SDF files, making it easy to reuse components across different simulations.

5.7.3 Integrating Gazebo with ROS

Gazebo uses plugins to interact with ROS. These plugins enable communication between the simulation and ROS nodes, allowing for control and data exchange.

Converting URDF to SDF

Gazebo can convert URDF (Unified Robot Description Format) files to SDF (Simulation Description Format) automatically, making it easy to integrate ROS-based robot descriptions into the simulation.

Running Gazebo with ROS Integration

To launch Gazebo with ROS integration, use the following command:

```
ros2 launch gazebo_ros gazebo.launch.py
```

This command starts Gazebo with an empty world, ready for further configuration.

5.7.4 Spawning a Robot in Gazebo

To spawn a robot in Gazebo, we can use the spawn_entity script provided by the gazebo_ros package:

```
ros2 run gazebo_ros spawn_entity -topic robot_description -entity my_bot
```

This script reads the robot description from the specified ROS topic and spawns the robot in the Gazebo world.

.

Gazebo, combined with ROS, provides a powerful simulation environment for testing and developing robotic systems. By simulating robots, sensors, and environments, developers can refine their algorithms and systems safely and efficiently. This integration allows for realistic testing and accelerates the development process, making it an invaluable tool in the robotics field.

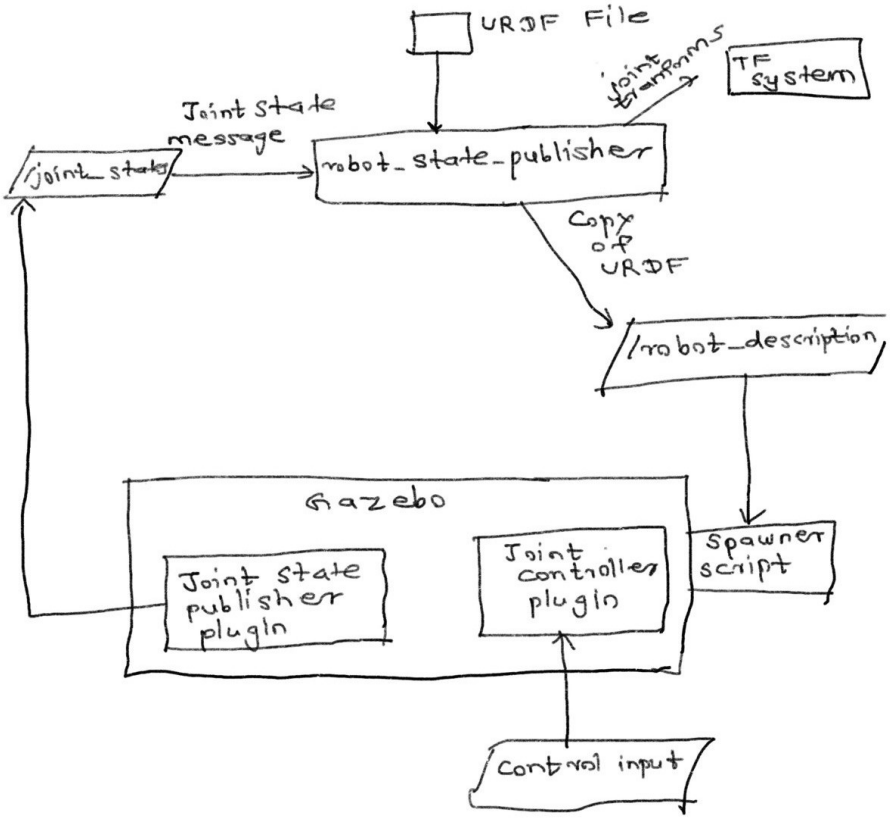


Figure 5.2: Block Diagram Including Gazebo plugins

5.8 Building our own robot model

5.8.1 Introduction

In this section, we'll focus on building our own mobile robot model using URDF (Unified Robot Description Format) and how to set it up for simulation. The robot we'll be building is a differential drive robot, which is a common type of robot with two independently driven wheels. This setup allows for easy control and high maneuverability.

5.8.2 Differential Drive Concept

A differential drive robot has two driven wheels on the left and right sides responsible for all motion control. Any additional wheels are caster wheels, which spin freely to provide stability. This design is prevalent because it is easy to understand, build, and control. It also allows the robot to turn on the spot, unlike a car, which needs space for a U-turn or a three-point turn.

In ROS, the main coordinate frame for a mobile robot should be called `base_link`, with the orientation set as follows:

- X-axis pointing forward
- Y-axis pointing to the left
- Z-axis pointing up

5.8.3 Recap of Important Ideas

- **URDF Files:** Describe the robot's structure.
- **Xacro:** Combines multiple configuration files into a single URDF.
- **Robot State Publisher:** Makes the URDF data available on the `robot_description` topic and broadcasts transforms.
- **Joint State Publisher:** Provides input values for movable joints.

URDF Files in the Project

The URDF description for our robot is divided into multiple files for better organization and modularity. Here, we'll look at the main URDF file (robot.urdf.xacro) and its included components.

Main URDF File: robot.urdf.xacro

This is the primary URDF file for the robot, including other essential files and configurations.

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="robot">
    <xacro:include filename="robot_core.xacro" />
    <xacro:include filename="lidar.xacro" />
</robot>
```

Core Robot Description: robot_core.xacro

This file defines the core structure of the robot, including its links, joints, and physical properties.

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

    <!-- Include inertial macros -->
    <xacro:include filename="inertial_macros.xacro" />

    <!-- Define properties for chassis and wheels -->
    <xacro:property name="chassis_length" value="0.34" />
    <xacro:property name="chassis_width" value="0.27" />
    <xacro:property name="chassis_height" value="0.14" />
    <xacro:property name="chassis_mass" value="1.1" />
    <xacro:property name="wheel_radius" value="0.035" />
    <xacro:property name="wheel_thickness" value="0.027" />
    <xacro:property name="wheel_mass" value="0.055" />
    <xacro:property name="wheel_offset_x" value="0.23" />
    <xacro:property name="wheel_offset_y" value="0.15" />
    <xacro:property name="wheel_offset_z" value="0.011" />
    <xacro:property name="caster_wheel_radius" value="0.011" />
    <xacro:property name="caster_wheel_mass" value="0.012" />
    <xacro:property name="caster_wheel_offset_x" value="0.08" />
```

```
<xacro:property name="caster_wheel_offset_z" value="${wheel_offset_z -
    wheel_radius + caster_wheel_radius}" />

<!-- Define materials -->
<material name="white">
    <color rgba="1 1 1 1" />
</material>

<material name="green">
    <color rgba="0 1 0 1" />
</material>

<material name="blue">
    <color rgba="0.2 0.2 1 1" />
</material>

<material name="black">
    <color rgba="0 0 0 1" />
</material>

<material name="yellow">
    <color rgba="1 1 0 1" />
</material>

<!-- Base link -->
<link name="base_link" />

<!-- Chassis link and joint -->
<joint name="chassis_joint" type="fixed">
    <parent link="base_link" />
    <child link="chassis" />
    <origin xyz="${-wheel_offset_x} 0 ${-wheel_offset_z}" />
</joint>

<link name="chassis">
    <visual>
        <origin xyz="${chassis_length/2} 0 ${chassis_height/2}" />
        <geometry>
            <box size="${chassis_length} ${chassis_width}
                ${chassis_height}" />
        </geometry>
        <material name="green" />
    </visual>
    <collision>
        <origin xyz="${chassis_length/2} 0 ${chassis_height/2}" />
        <geometry>
            <box size="${chassis_length} ${chassis_width}
                ${chassis_height}" />
        </geometry>
```

```
        </collision>
        <xacro:inertial_box mass="${chassis_mass}" x="${chassis_length}"
            y="${chassis_width}" z="${chassis_height}">
            <origin xyz="${chassis_length/2} 0 ${chassis_height/2}" rpy="0 0
                0" />
        </xacro:inertial_box>
    </link>

    <gazebo reference="chassis">
        <material>Gazebo/Green</material>
    </gazebo>

    <!-- Left wheel link and joint -->
    <joint name="left_wheel_joint" type="continuous">
        <parent link="base_link" />
        <child link="left_wheel" />
        <origin xyz="0 ${wheel_offset_y} 0" rpy="-${pi/2} 0 0" />
        <axis xyz="0 0 1" />
    </joint>

    <link name="left_wheel">
        <visual>
            <geometry>
                <cylinder radius="${wheel_radius}" length="${wheel_thickness}"
                    />
            </geometry>
            <material name="blue" />
        </visual>
        <collision>
            <geometry>
                <sphere radius="${wheel_radius}" />
            </geometry>
        </collision>
        <xacro:inertial_cylinder mass="${wheel_mass}"
            length="${wheel_thickness}" radius="${wheel_radius}">
            <origin xyz="0 0 0" rpy="0 0 0" />
        </xacro:inertial_cylinder>
    </link>

    <gazebo reference="left_wheel">
        <material>Gazebo/Blue</material>
    </gazebo>

    <!-- Right wheel link and joint -->
    <joint name="right_wheel_joint" type="continuous">
        <parent link="base_link" />
        <child link="right_wheel" />
        <origin xyz="0 ${-wheel_offset_y} 0" rpy="${pi/2} 0 0" />
        <axis xyz="0 0 -1" />
```

```
    </joint>

    <link name="right_wheel">
        <visual>
            <geometry>
                <cylinder radius="${wheel_radius}" length="${wheel_thickness}"
                    />
            </geometry>
            <material name="blue" />
        </visual>
        <collision>
            <geometry>
                <sphere radius="${wheel_radius}" />
            </geometry>
        </collision>
        <xacro:inertial_cylinder mass="${wheel_mass}"
            length="${wheel_thickness}" radius="${wheel_radius}">
            <origin xyz="0 0 0" rpy="0 0 0" />
        </xacro:inertial_cylinder>
    </link>

    <gazebo reference="right_wheel">
        <material>Gazebo/Blue</material>
    </gazebo>

    <!-- Caster wheels links and joints -->
    <joint name="caster_wheel_joint_" type="fixed">
        <parent link="chassis" />
        <child link="caster_wheel" />
        <origin xyz="${caster_wheel_offset_x} 0 ${caster_wheel_offset_z}" />
    </joint>

    <link name="caster_wheel">
        <visual>
            <geometry>
                <sphere radius="${caster_wheel_radius}" />
            </geometry>
            <material name="yellow" />
        </visual>
        <collision>
            <geometry>
                <sphere radius="${caster_wheel_radius}" />
            </geometry>
        </collision>
        <xacro:inertial_sphere mass="${caster_wheel_mass}"
            radius="${caster_wheel_radius}">
            <origin xyz="0 0 0" rpy="0 0 0" />
        </xacro:inertial_sphere>
    </link>
```

```
    <gazebo reference="caster_wheel">
        <material>Gazebo/Yellow</material>
        <mu1 value="0.001" />
        <mu2 value="0.001" />
    </gazebo>

</robot>
```

Sensor Description: lidar.xacro

This file describes the LiDAR sensor and its integration into the robot.

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

    <!-- Define the laser joint -->
    <joint name="laser_joint" type="fixed">
        <parent link="chassis" />
        <child link="laser_frame" />
        <origin xyz="0.12 0 0.21" rpy="0 0 0" />
    </joint>

    <!-- Define the laser frame link -->
    <link name="laser_frame">
        <visual>
            <geometry>
                <cylinder radius="0.055" length="0.045" />
            </geometry>
            <material name="black" />
        </visual>
        <visual>
            <origin xyz="0 0 -0.055" />
            <geometry>
                <cylinder radius="0.015" length="0.11" />
            </geometry>
            <material name="black" />
        </visual>
        <collision>
            <geometry>
                <cylinder radius="0.055" length="0.045" />
            </geometry>
        </collision>
        <xacro:inertial_cylinder mass="0.12" length="0.045" radius="0.055">
            <origin xyz="0 0 0" rpy="0 0 0" />
        </xacro:inertial_cylinder>
    </link>
```

```
    <!-- Gazebo-specific properties for the laser frame -->
    <gazebo reference="laser_frame">
        <material>Gazebo/Black</material>

        <!-- Define the laser sensor -->
        <sensor name="laser" type="ray">
            <pose>0 0 0 0 0 0</pose>
            <visualize>false</visualize>
            <update_rate>10</update_rate>
            <ray>
                <scan>
                    <horizontal>
                        <samples>370</samples>
                        <min_angle>-3.13</min_angle>
                        <max_angle>3.13</max_angle>
                    </horizontal>
                </scan>
                <range>
                    <min>0.25</min>
                    <max>12.5</max>
                </range>
            </ray>
            <plugin name="laser_controller"
                filename="libgazebo_ros_ray_sensor.so">
                <ros>
                    <argument>~/out:=scan</argument>
                </ros>
                <output_type>sensor_msgs/LaserScan</output_type>
                <frame_name>laser_frame</frame_name>
            </plugin>
        </sensor>
    </gazebo>

</robot>
```

Inertial properties: intertial_macros.xacro

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

    <!-- Define standard inertial calculations -->
    <!-- Refer to https://en.wikipedia.org/wiki/List_of_moments_of_inertia for
        more details -->
    <!-- Utilize xacro's mathematical functionality -->

    <!-- Macro for calculating inertial properties of a sphere -->
```

```
    <xacro:macro name="inertial_sphere" params="mass radius *origin">
        <inertial>
            <xacro:insert_block name="origin" />
            <mass value="${mass}" />
            <inertia ixx="${(2/5) * mass * (radius * radius)}" ixy="0.0"
                ixz="0.0"
                    iyy="${(2/5) * mass * (radius * radius)}" iyz="0.0"
                    izz="${(2/5) * mass * (radius * radius)}" />
        </inertial>
    </xacro:macro>

    <!-- Macro for calculating inertial properties of a box -->
    <xacro:macro name="inertial_box" params="mass x y z *origin">
        <inertial>
            <xacro:insert_block name="origin" />
            <mass value="${mass}" />
            <inertia ixx="${(1/12) * mass * (y * y + z * z)}" ixy="0.0"
                ixz="0.0"
                    iyy="${(1/12) * mass * (x * x + z * z)}" iyz="0.0"
                    izz="${(1/12) * mass * (x * x + y * y)}" />
        </inertial>
    </xacro:macro>

    <!-- Macro for calculating inertial properties of a cylinder -->
    <xacro:macro name="inertial_cylinder" params="mass length radius *origin">
        <inertial>
            <xacro:insert_block name="origin" />
            <mass value="${mass}" />
            <inertia ixx="${(1/12) * mass * (3 * radius * radius + length *
                length)}" ixy="0.0" ixz="0.0"
                    iyy="${(1/12) * mass * (3 * radius * radius + length *
                        length)}" iyz="0.0"
                    izz="${(1/2) * mass * (radius * radius)}" />
        </inertial>
    </xacro:macro>

</robot>
```

These 4 files are included in the ”description” directory which is created within the ”src” directory.

URDF provides a standardized way to describe robotic systems, ensuring that all software components can reference the same physical characteristics. By using links and joints, and leveraging tools like Xacro, we can create detailed and reusable robot descriptions. This approach simplifies the development and maintenance of robotic systems, enhancing consistency and collaboration.

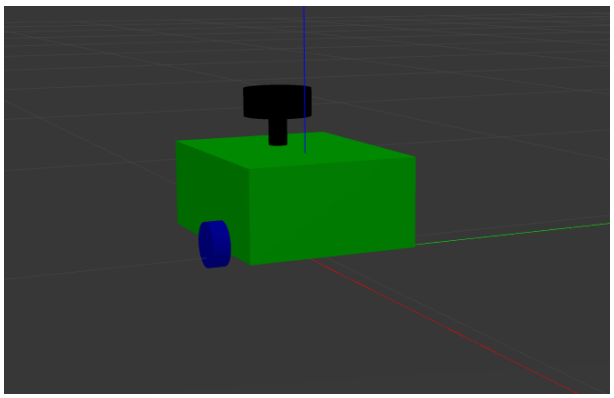


Figure 5.3: Virtual Robot

5.8.4 Resulted Robot

5.9 Driving the virtual robot

5.9.1 Introduction

In this section, we will continue from the URDF file creation and work on simulating the robot in Gazebo. In this section, we will create a 3D simulated version of our robot that we can drive around in a virtual environment.

5.9.2 Spawning Our Robot in the Gazebo world

We'll start by spawning our robot into Gazebo using the URDF we've created. This involves three steps:

1. Running the Robot State Publisher in simulation mode.
2. Launching Gazebo with ROS compatibility.
3. Running the Gazebo robot spawner script to add our robot to the simulation.

5.9.3 Wrapping It Up in a Launch File

To make our lives easier, we will create a single launch file to handle starting the Robot State Publisher, Gazebo, and the robot spawner all at once. First , we include the "robot_state_publisher" node in the following `rsp.launch.py` launch file.

```
import os

from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch.substitutions import LaunchConfiguration, Command
from launch.actions import DeclareLaunchArgument
from launch_ros.actions import Node

import xacro
```

```
def generate_launch_description():

    # Get configurations for simulation time and ROS2 control
    use_sim_time = LaunchConfiguration('use_sim_time')

    # Process the URDF file using xacro
    pkg_path = os.path.join(get_package_share_directory('my_bot'))
    xacro_file = os.path.join(pkg_path, 'description', 'robot.urdf.xacro')
    robot_description_config = Command(['xacro', xacro_file, 'sim_mode:=',
        use_sim_time])

    # Set parameters for the robot_state_publisher node
    params = {'robot_description': robot_description_config, 'use_sim_time':
        use_sim_time}
    node_robot_state_publisher = Node(
        package='robot_state_publisher',
        executable='robot_state_publisher',
        output='screen',
        parameters=[params]
    )

    # Return the LaunchDescription with declared arguments and the node
    return LaunchDescription([
        DeclareLaunchArgument(
            'use_sim_time',
            default_value='false',
            description='Use simulation time if true'),
        DeclareLaunchArgument(
            'use_ros2_control',
            default_value='true',
            description='Use ROS2 control if true'),
        node_robot_state_publisher
    ])
```

Here is the launch file named `launch_robot_sim.launch.py` we created to include the above launch file and other essential nodes. We will continue to update this code along the way.

```
import os
from ament_index_python.packages import get_package_share_directory
from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch_ros.actions import Node

def generate_launch_description():
    # Get the package share directory for 'my_bot'
    pkg_share = get_package_share_directory('my_bot')
```

```
    # Include the robot state publisher launch file with simulation time
        enabled
    rsp = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(pkg_share, 'launch', 'rsp.launch.py')
        ),
        launch_arguments={'use_sim_time': 'true'}.items()
    )

    # Include the Gazebo launch file
    gazebo = IncludeLaunchDescription(
        PythonLaunchDescriptionSource(
            os.path.join(get_package_share_directory('gazebo_ros'), 'launch',
                'gazebo.launch.py')
        )
    )

    # Node to spawn the robot entity in Gazebo
    spawn_entity = Node(
        package='gazebo_ros',
        executable='spawn_entity.py',
        arguments=['-topic', 'robot_description', '-entity', 'robot'],
        output='screen'
    )

    # Return the LaunchDescription with the included launch files and node
    return LaunchDescription([
        rsp,
        gazebo,
        spawn_entity,
    ])
```

Save this file and execute the following command to visulaize the virtual robot:

```
ros2 launch my_bot launch_robot_sim.launch.py
```

This will start everything needed for our simulation.

With the simulation environment set up, we can now drive your robot around a virtual room. In the next steps, we will look at integrating sensors and further refining the simulation.

5.9.4 Control Concept Overview

Before diving into the implementation of control systems for our robot, it's essential to understand a few fundamental concepts. Later in this project, we will use the ROS2 control library to manage our control code. This library offers the advantage of a unified codebase that works for both the simulated and real robot, minimizing discrepancies between the two environments.

Although ROS2 control is robust, its setup can be complex. For now, we'll utilize a simpler differential drive control system provided by Gazebo to control our robot.

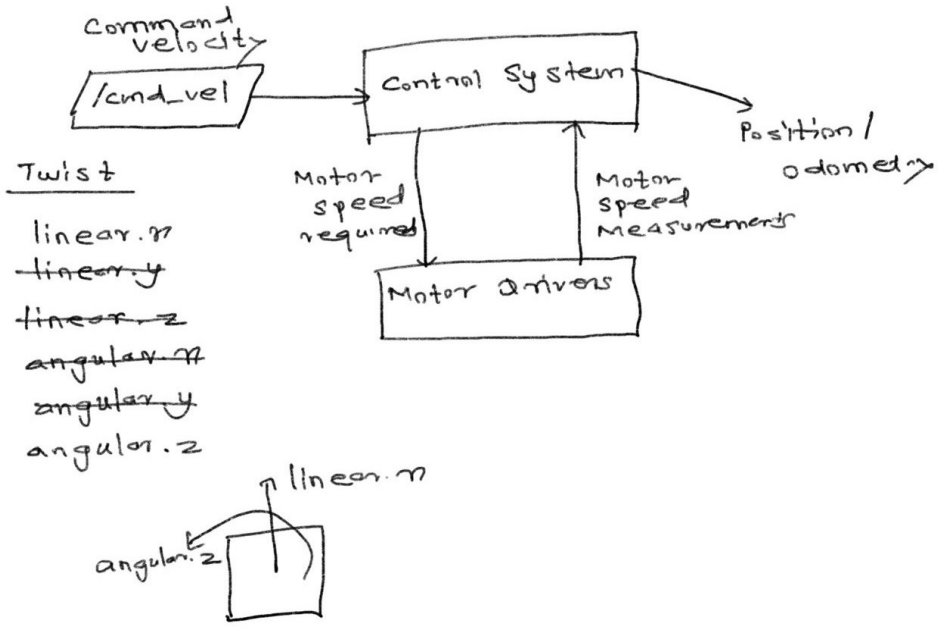


Figure 5.4: Control Concept

Key Concepts

- **Command Velocity Input:** The control system takes command velocity inputs, determining how fast the robot should move. These commands are translated into motor commands for the motor drivers, which are then used to drive the motors.
- **Odometry:** By reading the actual motor speeds, the control system can calculate the robot's true velocity and estimate its position through a process called dead reckoning. This position estimate is known as odometry.
- **Twist Messages:** In ROS, the command velocity is published on the **cmd_vel** topic as a Twist message. This message contains six components: linear velocities (x, y, z) and angular velocities (around x, y, z axes). For a differential drive robot, we only use the linear velocity in the x-axis (forward/backward) and angular velocity around the z-axis (turning), with the other components set to zero.
- **Transforms:** The control plugin broadcasts a transform from a frame called **odom** (representing the robot's start position) to the **base_link** frame (representing the robot's current position). This transform provides the current position estimate of the robot.

5.9.5 Adding the Control Plugin

To drive our robot, we need to add a control plugin to our URDF file. Instead of cluttering the main URDF file, we create a new include file specifically for the Gazebo control configuration.

Steps to Add the Control Plugin

1. Create a new Gazebo control include file:

```
<!-- gazebo_control.xacro -->
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

    <gazebo>
        <plugin name="diff_drive" filename="libgazebo_ros_diff_drive.so">

            <!-- Wheel Information -->
            <left_joint>left_wheel_joint</left_joint>
            <right_joint>right_wheel_joint</right_joint>
            <wheel_separation>0.30</wheel_separation>
            <wheel_diameter>0.07</wheel_diameter>

            <!-- Limits -->
            <max_wheel_torque>200</max_wheel_torque>
            <max_wheel_acceleration>10.0</max_wheel_acceleration>

            <!-- Output -->
            <odometry_topic>odom3</odometry_topic>
            <odometry_frame>odom3</odometry_frame>
            <robot_base_frame>base_link</robot_base_frame>

            <publish_odom>true</publish_odom>
            <publish_odom_tf>true</publish_odom_tf>
            <publish_wheel_tf>true</publish_wheel_tf>

        </plugin>
    </gazebo>

</robot>
```

2. Include the Gazebo control file in the main URDF:

```
<!-- robot.urdf.xacro -->
<xacro:include filename="gazebo_control.xacro" />
```

5.9.6 Keyboard Teleoperation

For manual control of the robot during testing, instead of using the standard `teleop_twist_keyboard` package, which allows sending velocity commands through keyboard inputs, we are going to modify that code to fit our use case.

Create a new package called ”tele” inside the ”src” directory by executing the following command.

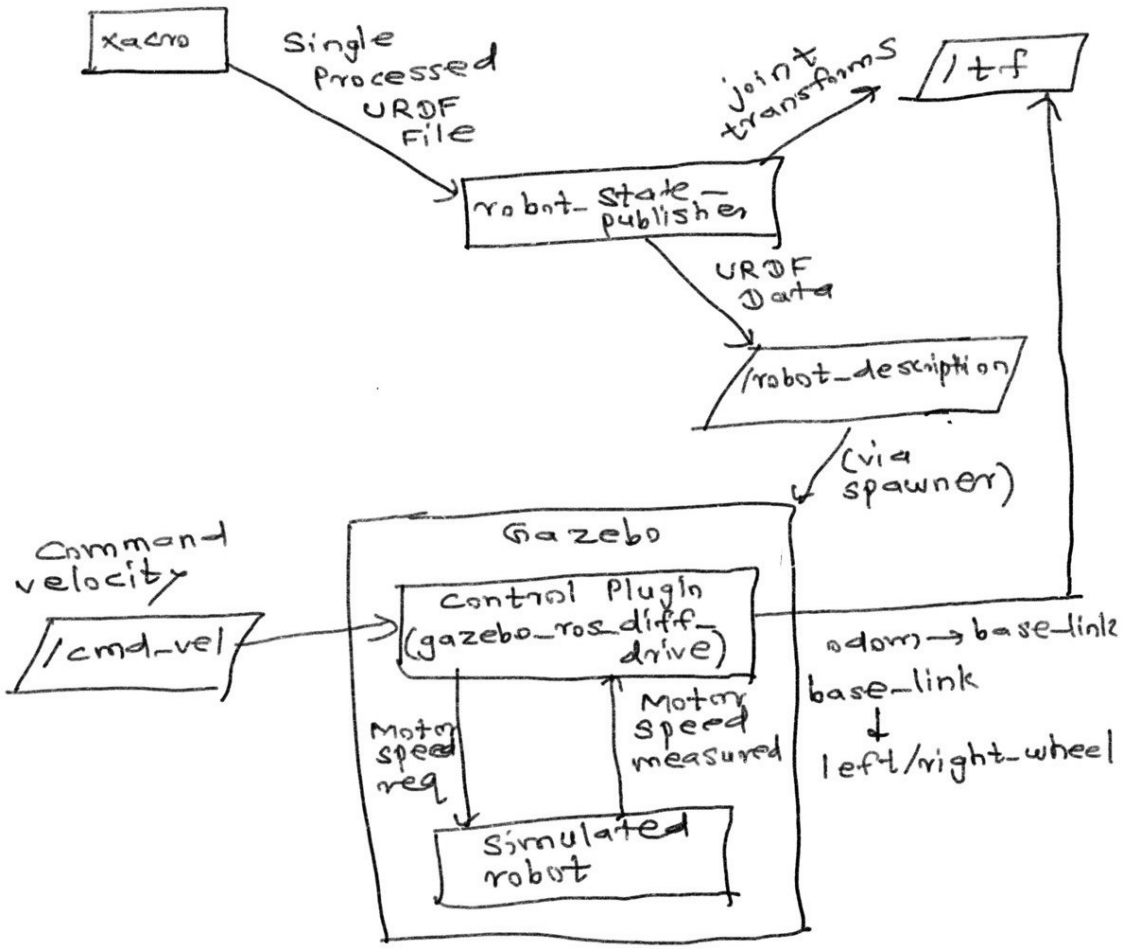


Figure 5.5: Control Plugin Added

```
ros2 pkg create --build-type ament_cmake tele
```

Inside the "tele" directory there are 3 files,

```
#setup.py
from setuptools import setup

package_name = 'tele'

setup(
    name=package_name,
    version='0.0.1',
    packages=[package_name],
    data_files=[
        ('share/ament_index/resource_index/packages',
```

```
            ['resource/' + package_name]),
        ('share/' + package_name, ['package.xml']),
    ],
    install_requires=['setuptools'],
    entry_points={
        'console_scripts': [
            'teleop_twist_keyboard = tele.teleop_twist_keyboard:main'
        ],
    },
)
```

```
#setup.cfg
[develop]
script_dir=$base/lib/tele
[install]
install_scripts=$base/lib/tele
```

```
#teleop_twist_keyboard.py

# All rights reserved.

# Software License Agreement (BSD License 2.0)

# Redistribution and use in source and binary forms, with or without
# modification, are permitted provided that the following conditions
# are met:

#  * Redistributions of source code must retain the above copyright
#    notice, this list of conditions and the following disclaimer.
#  * Redistributions in binary form must reproduce the above
#    copyright notice, this list of conditions and the following
#    disclaimer in the documentation and/or other materials provided
#    with the distribution.
#  * Neither the name of the copyright holder nor the names of its
#    contributors may be used to endorse or promote products derived
#    from this software without specific prior written permission.

# THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
# "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
# LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
# FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
# COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
# INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
# BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
# LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
# CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
# LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
```

```
# ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
# POSSIBILITY OF SUCH DAMAGE.

# Modified by Sundarbavan T.

import sys
import threading
import geometry_msgs.msg
import rclpy

# Platform-specific imports for keyboard input
if sys.platform == 'win32':
    import msvcrt
else:
    import termios
    import tty

# Instructions for the user
instructions = """
This node captures keypresses from the keyboard and publishes them
as Twist/TwistStamped messages. It is optimized for a US keyboard layout.
---------------------------
Movement keys:
   u    i    o
   j    k    l
   m    ,    .

For Holonomic mode (strafing), hold down the shift key:
---------------------------
   U    I    O
   J    K    L
   M    <    >

t : up (+z)
b : down (-z)

other keys : stop

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%

CTRL-C to quit
"""

# Movement bindings
move_bindings = {
    'i': (1, 0, 0, 0),
    'o': (1, 0, 0, -1),
```

```
    'j': (0, 0, 0, 1),
    'l': (0, 0, 0, -1),
    'u': (1, 0, 0, 1),
    ',': (-1, 0, 0, 0),
    '.': (-1, 0, 0, 1),
    'm': (-1, 0, 0, -1),
    'O': (1, -1, 0, 0),
    'I': (1, 0, 0, 0),
    'J': (0, 1, 0, 0),
    'L': (0, -1, 0, 0),
    'U': (1, 1, 0, 0),
    '<': (-1, 0, 0, 0),
    '>': (-1, -1, 0, 0),
    'M': (-1, 1, 0, 0),
    't': (0, 0, 1, 0),
    'b': (0, 0, -1, 0),
}

# Speed adjustment bindings
speed_bindings = {
    'q': (1.1, 1.1),
    'z': (.9, .9),
    'w': (1.1, 1),
    'x': (.9, 1),
    'e': (1, 1.1),
    'c': (1, .9),
}

# Timer for automatic stopping
timer = None
stop_time = 0.1  # seconds

def get_key(settings):
    """
    Get a single keypress from the user.
    """
    if sys.platform == 'win32':
        return msvcrt.getwch()
    else:
        tty.setraw(sys.stdin.fileno())
        key = sys.stdin.read(1)
        termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)
        return key

def save_terminal_settings():
    """
    Save the current terminal settings.
    """
    if sys.platform == 'win32':
```

```
        return None
    return termios.tcgetattr(sys.stdin)

def restore_terminal_settings(old_settings):
    """
    Restore the terminal settings to their previous state.
    """
    if sys.platform == 'win32':
        return
    termios.tcsetattr(sys.stdin, termios.TCSADRAIN, old_settings)

def display_velocities(speed, turn):
    """
    Display the current speed and turn rates.
    """
    return f'currently:\tspeed {speed}\tturn {turn}'

def stop_robot(pub, twist_msg, stamped):
    """
    Publish a message to stop the robot.
    """
    if stamped:
        twist_msg.header.stamp = node.get_clock().now().to_msg()
        twist_msg.twist.linear.x = 0.0
        twist_msg.twist.linear.y = 0.0
        twist_msg.twist.linear.z = 0.0
        twist_msg.twist.angular.x = 0.0
        twist_msg.twist.angular.y = 0.0
        twist_msg.twist.angular.z = 0.0
    else:
        twist_msg.linear.x = 0.0
        twist_msg.linear.y = 0.0
        twist_msg.linear.z = 0.0
        twist_msg.angular.x = 0.0
        twist_msg.angular.y = 0.0
        twist_msg.angular.z = 0.0
    pub.publish(twist_msg)

def reset_timer(pub, twist_msg, stamped):
    """
    Reset the timer to stop the robot after a certain duration of inactivity.
    """
    global timer
    if timer is not None:
        timer.cancel()
    timer = threading.Timer(stop_time, stop_robot, [pub, twist_msg, stamped])
    timer.start()

def main():
```

```
    """
    Main function to initialize the node and handle keyboard input.
    """
    settings = save_terminal_settings()

    rclpy.init()

    global node
    node = rclpy.create_node('teleop_twist_keyboard')

    # Parameters for whether to use TwistStamped messages and the frame ID
    stamped = node.declare_parameter('stamped', False).value
    frame_id = node.declare_parameter('frame_id', '').value

    TwistMsg = geometry_msgs.msg.TwistStamped if stamped else
        geometry_msgs.msg.Twist
    pub = node.create_publisher(TwistMsg, 'cmd_vel', 10)

    # Thread to keep the node spinning
    spinner = threading.Thread(target=rclpy.spin, args=(node,))
    spinner.start()

    speed = 0.5
    turn = 1.0
    status = 0.0

    twist_msg = TwistMsg()
    if stamped:
        twist_msg.header.frame_id = frame_id

    try:
        print(instructions)
        print(display_velocities(speed, turn))
        while True:
            key = get_key(settings)
            if key in move_bindings:
                x, y, z, th = move_bindings[key]
                reset_timer(pub, twist_msg, stamped)
            elif key in speed_bindings:
                speed *= speed_bindings[key][0]
                turn *= speed_bindings[key][1]
                print(display_velocities(speed, turn))
                if status == 14:
                    print(instructions)
                status = (status + 1) % 15
            else:
                x = y = z = th = 0
                if key == '\x03':  # CTRL-C
                    break
```

```
            if stamped:
                twist_msg.header.stamp = node.get_clock().now().to_msg()
                twist_msg.twist.linear.x = float(x) * speed
                twist_msg.twist.linear.y = float(y) * speed
                twist_msg.twist.linear.z = float(z) * speed
                twist_msg.twist.angular.z = float(th) * turn
            else:
                twist_msg.linear.x = float(x) * speed
                twist_msg.linear.y = float(y) * speed
                twist_msg.linear.z = float(z) * speed
                twist_msg.angular.z = float(th) * turn

            pub.publish(twist_msg)

    except Exception as e:
        print(e)

    finally:
        rclpy.shutdown()
        spinner.join()
        restore_terminal_settings(settings)

if __name__ == '__main__':
    main()
```

After add these files use "colcon build" and build the package as mentioned earlier in the documentation.

Running the Teleop Node

```
ros2 run tele teleop_twist_keyboard
```

This node maps keyboard keys to movement commands, enabling easy manual control of the robot. We can press the keys and move the robot around.

5.9.7 RViz Visualization

RViz serves as the visual interface to observe and interact with the robot's simulation environment.

Configuring RViz

Launch RViz and configure it to display the robot's state and movement within the simulated environment.

```
rviz2
```

Add necessary visualization elements like the Robot Model and Transform displays to monitor the robot's movements and sensor data.

By integrating these control and visualization tools, we can effectively manage and observe the robot's operations within Gazebo, setting the stage for more advanced applications such as autonomous navigation and interactive tasks. This foundation is crucial for developing robust and versatile robotic systems using ROS2.

5.10 ros2_control

5.10.1 Introduction

Control is a fundamental aspect of robotics, encompassing the transformation of input signals into commands for actuators such as motors and hydraulics. Given the diversity of actuators and control methodologies, a standardized system becomes essential to avoid repetitive and inefficient custom solutions. This section introduces ros2_control, a framework designed to streamline the control process across various hardware and control scenarios, and discusses its practical application in both simulated and real-world environments.

5.10.2 The Need for ros2_control

Robotic control systems must manage a vast array of actuators and interfaces. Traditionally, each new project might involve rewriting custom controllers and drivers, leading to inefficiencies and redundancy. Instead, a unified framework where hardware drivers and control algorithms communicate in a standardized manner can save significant development time and resources. While ROS topics could theoretically handle this by defining nodes for hardware drivers and controllers, the need for speed and real-time responsiveness necessitates a more efficient approach. This is where ros2_control excels, providing a robust and flexible system for real-time control.

5.10.3 Architecture of ros2_control

Hardware Interface

Different hardware setups require specific control methodologies. The ros2_control framework uses hardware interfaces to standardize communication with various hardware components. These interfaces abstract the specifics of hardware control into command and state interfaces:

- **Command Interfaces:** These are writable interfaces for controlling the robot, such as setting motor velocities.
- **State Interfaces:** These are readable interfaces for monitoring the robot, such as reading encoder positions or velocities.

Each robot may have multiple hardware interfaces, depending on its complexity and modularity. These interfaces are defined in the robot's URDF (Unified Robot Description Format) file, specifying which joints and sensors are available for control and monitoring.

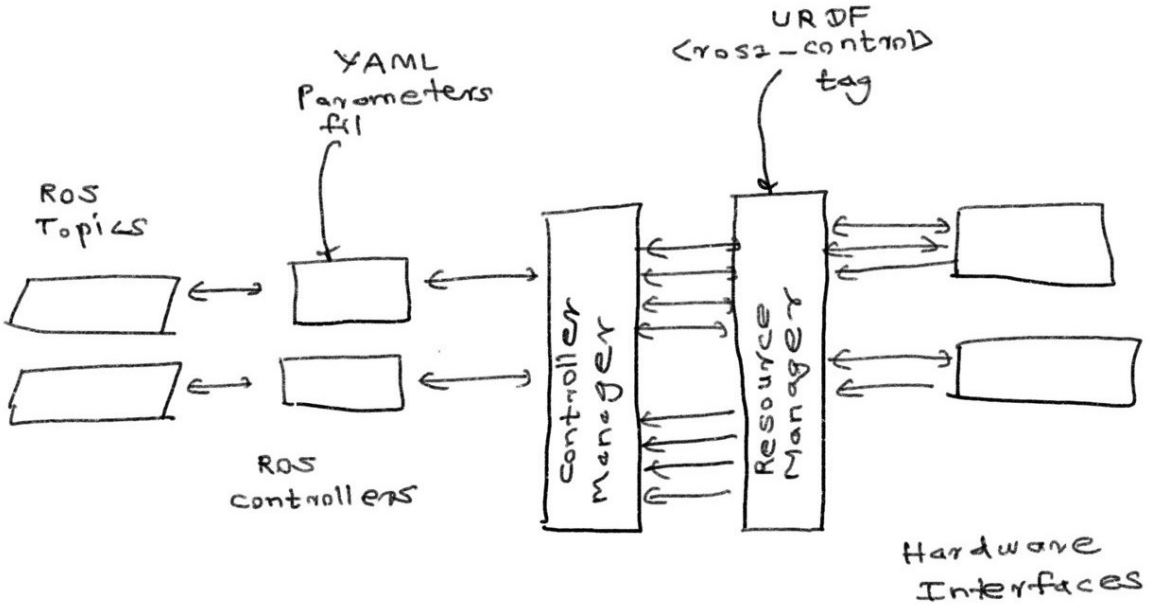


Figure 5.6: ros2_control concept

Controller Interface

Controllers in ros2_control manage the logic of transforming high-level commands into specific hardware actions. These controllers:

- Listen for input commands on ROS topics.
- Calculate the necessary actions for the actuators.
- Communicate these actions to the hardware interfaces via the resource manager.

The framework includes various pre-built controllers for common robotic applications, such as differential drive robots and robotic arms. Users can also develop custom controllers if needed.

In our project we use the "diff_drive_controller" to perform necessary calculations to convert the "cmd_vel" (command velocity - linear.x velocity and angular.z velocity) to indivual wheel rotational velocities.

The source code for the "diff_drive_controller" is given below.

```
// Copyright 2020 PAL Robotics S.L.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```cpp
// See the License for the specific language governing permissions and
// limitations under the License.

/*
 * Author: Bence Magyar, Enrique Fernández, Manuel Meraz
 */


#include <memory>
#include <queue>
#include <string>
#include <utility>
#include <vector>

#include "diff_drive_controller/diff_drive_controller.hpp"
#include "hardware_interface/types/hardware_interface_type_values.hpp"
#include "lifecycle_msgs/msg/state.hpp"
#include "rclcpp/logging.hpp"
#include "tf2/LinearMath/Quaternion.h"

namespace
{
// Default topics used by the controller
constexpr auto DEFAULT_COMMAND_TOPIC = "~/cmd_vel";
constexpr auto DEFAULT_COMMAND_OUT_TOPIC = "~/cmd_vel_out";
constexpr auto DEFAULT_ODOMETRY_TOPIC = "~/odom";
constexpr auto DEFAULT_TRANSFORM_TOPIC = "/tf";
}  // namespace

namespace diff_drive_controller
{
using namespace std::chrono_literals;
using controller_interface::interface_configuration_type;
using controller_interface::InterfaceConfiguration;
using hardware_interface::HW_IF_POSITION;
using hardware_interface::HW_IF_VELOCITY;
using lifecycle_msgs::msg::State;

// Constructor
DiffDriveController::DiffDriveController() :
    controller_interface::ControllerInterface() {}

// Determine feedback type based on parameter
const char * DiffDriveController::feedback_type() const
{
  return params_.position_feedback ? HW_IF_POSITION : HW_IF_VELOCITY;
}

// Initialization callback
```

```
controller_interface::CallbackReturn DiffDriveController::on_init()
{
  try
  {
    // Create the parameter listener and get the parameters
    param_listener_ = std::make_shared<ParamListener>(get_node());
    params_ = param_listener_->get_params();
  }
  catch (const std::exception & e)
  {
    fprintf(stderr, "Exception during init stage: %s \n", e.what());
    return controller_interface::CallbackReturn::ERROR;
  }

  return controller_interface::CallbackReturn::SUCCESS;
}

// Configuration for command interfaces
InterfaceConfiguration DiffDriveController::command_interface_configuration()
    const
{
  std::vector<std::string> conf_names;
  for (const auto & joint_name : params_.left_wheel_names)
  {
    conf_names.push_back(joint_name + "/" + HW_IF_VELOCITY);
  }
  for (const auto & joint_name : params_.right_wheel_names)
  {
    conf_names.push_back(joint_name + "/" + HW_IF_VELOCITY);
  }
  return {interface_configuration_type::INDIVIDUAL, conf_names};
}

// Configuration for state interfaces
InterfaceConfiguration DiffDriveController::state_interface_configuration()
    const
{
  std::vector<std::string> conf_names;
  for (const auto & joint_name : params_.left_wheel_names)
  {
    conf_names.push_back(joint_name + "/" + feedback_type());
  }
  for (const auto & joint_name : params_.right_wheel_names)
  {
    conf_names.push_back(joint_name + "/" + feedback_type());
  }
  return {interface_configuration_type::INDIVIDUAL, conf_names};
}
```

```
// Update function called periodically
controller_interface::return_type DiffDriveController::update(
  const rclcpp::Time & time, const rclcpp::Duration & period)
{
  auto logger = get_node()->get_logger();
  if (get_state().id() == State::PRIMARY_STATE_INACTIVE)
  {
    if (!is_halted)
    {
      halt();
      is_halted = true;
    }
    return controller_interface::return_type::OK;
  }

  // Get the last command message
  std::shared_ptr<Twist> last_command_msg;
  received_velocity_msg_ptr_.get(last_command_msg);

  if (last_command_msg == nullptr)
  {
    RCLCPP_WARN(logger, "Received null velocity message.");
    return controller_interface::return_type::ERROR;
  }

  const auto age_of_last_command = time - last_command_msg->header.stamp;
  // Brake if cmd_vel has timed out
  if (age_of_last_command > cmd_vel_timeout_)
  {
    last_command_msg->twist.linear.x = 0.0;
    last_command_msg->twist.angular.z = 0.0;
  }

  // Command may be limited further by SpeedLimiter
  Twist command = *last_command_msg;
  double & linear_command = command.twist.linear.x;
  double & angular_command = command.twist.angular.z;

  previous_update_timestamp_ = time;

  // Apply multipliers
  const double wheel_separation = params_.wheel_separation_multiplier *
  params_.wheel_separation;
  const double left_wheel_radius = params_.left_wheel_radius_multiplier
  * params_.wheel_radius;
  const double right_wheel_radius = params_.right_wheel_radius_multiplier *
      params_.wheel_radius;

  if (params_.open_loop)
```

```
{
  odometry_.updateOpenLoop(linear_command, angular_command, time);
}
else
{
  double left_feedback_mean = 0.0;
  double right_feedback_mean = 0.0;
  for (size_t index = 0; index < static_cast<size_t>(wheels_per_side_);
      ++index)
  {
    const double left_feedback =
        registered_left_wheel_handles_[index].feedback.get().get_value();
    const double right_feedback =
        registered_right_wheel_handles_[index].feedback.get().get_value();

    if (std::isnan(left_feedback) || std::isnan(right_feedback))
    {
      RCLCPP_ERROR(logger, "Invalid feedback for wheel %zu", index);
      return controller_interface::return_type::ERROR;
    }

    left_feedback_mean += left_feedback;
    right_feedback_mean += right_feedback;
  }
  left_feedback_mean /= static_cast<double>(wheels_per_side_);
  right_feedback_mean /= static_cast<double>(wheels_per_side_);

  if (params_.position_feedback)
  {
    odometry_.update(left_feedback_mean, right_feedback_mean, time);
  }
  else
  {
    odometry_.updateFromVelocity(
      left_feedback_mean * left_wheel_radius * period.seconds(),
      right_feedback_mean * right_wheel_radius * period.seconds(), time);
  }
}

// Update orientation
tf2::Quaternion orientation;
orientation.setRPY(0.0, 0.0, odometry_.getHeading());

bool should_publish = false;
try
{
  if (previous_publish_timestamp_ + publish_period_ < time)
  {
    previous_publish_timestamp_ += publish_period_;
```

```
    should_publish = true;
  }
}
catch (const std::runtime_error &)
{
  // Handle exceptions when the time source changes and initialize publish
      timestamp
  previous_publish_timestamp_ = time;
  should_publish = true;
}

if (should_publish)
{
  // Publish odometry message
  if (realtime_odometry_publisher_->trylock())
  {
    auto & odometry_message = realtime_odometry_publisher_->msg_;
    odometry_message.header.stamp = time;
    odometry_message.pose.pose.position.x = odometry_.getX();
    odometry_message.pose.pose.position.y = odometry_.getY();
    odometry_message.pose.pose.orientation.x = orientation.x();
    odometry_message.pose.pose.orientation.y = orientation.y();
    odometry_message.pose.pose.orientation.z = orientation.z();
    odometry_message.pose.pose.orientation.w = orientation.w();
    odometry_message.twist.twist.linear.x = odometry_.getLinear();
    odometry_message.twist.twist.angular.z = odometry_.getAngular();
    realtime_odometry_publisher_->unlockAndPublish();
  }

  // Publish transform
  if (params_.enable_odom_tf &&
      realtime_odometry_transform_publisher_->trylock())
  {
    auto & transform =
        realtime_odometry_transform_publisher_->msg_.transforms.front();
    transform.header.stamp = time;
    transform.transform.translation.x = odometry_.getX();
    transform.transform.translation.y = odometry_.getY();
    transform.transform.rotation.x = orientation.x();
    transform.transform.rotation.y = orientation.y();
    transform.transform.rotation.z = orientation.z();
    transform.transform.rotation.w = orientation.w();
    realtime_odometry_transform_publisher_->unlockAndPublish();
  }
}

// Apply speed limits
auto & last_command = previous_commands_.back().twist;
auto & second_to_last_command = previous_commands_.front().twist;
```

```
  limiter_linear_.limit(
    linear_command, last_command.linear.x, second_to_last_command.linear.x,
        period.seconds());
  limiter_angular_.limit(
    angular_command, last_command.angular.z, second_to_last_command.angular.z,
        period.seconds());

  previous_commands_.pop();
  previous_commands_.emplace(command);

  // Publish limited velocity if required
  if (publish_limited_velocity_ && realtime_limited_velocity_publisher_-
  >trylock())
  {
    auto & limited_velocity_command =
        realtime_limited_velocity_publisher_->msg_;
    limited_velocity_command.header.stamp = time;
    limited_velocity_command.twist = command.twist;
    realtime_limited_velocity_publisher_->unlockAndPublish();
  }

  // Compute wheel velocities
  const double velocity_left = (linear_command - angular_command *
      wheel_separation / 2.0) / left_wheel_radius;
  const double velocity_right = (linear_command + angular_command *
      wheel_separation / 2.0) / right_wheel_radius;

  // Set wheel velocities
  for (size_t index = 0; index < static_cast<size_t>(wheels_per_side_);
  ++index)
  {
    registered_left_wheel_handles_[index].velocity.get().set_value
    (velocity_left);
    registered_right_wheel_handles_[index].velocity.get().set_value
    (velocity_right);
  }

  return controller_interface::return_type::OK;
}

// Callback for configuring the controller
controller_interface::CallbackReturn DiffDriveController::on_configure
(
  const rclcpp_lifecycle::State &)
{
  auto logger = get_node()->get_logger();

  // Update parameters if they have changed
  if (param_listener_->is_old(params_))
```

```
{
  params_ = param_listener_->get_params();
  RCLCPP_INFO(logger, "Parameters updated");
}

if (params_.left_wheel_names.size() != params_.right_wheel_names.size())
{
  RCLCPP_ERROR(
    logger, "Mismatch between left and right wheel counts: %zu left,
    %zu right",
    params_.left_wheel_names.size(), params_.right_wheel_names.size());
  return controller_interface::CallbackReturn::ERROR;
}

const double wheel_separation = params_.wheel_separation_multiplier *
params_.wheel_separation;
const double left_wheel_radius = params_.left_wheel_radius_multiplier
* params_.wheel_radius;
const double right_wheel_radius = params_.right_wheel_radius_multiplier *
    params_.wheel_radius;

odometry_.setWheelParams(wheel_separation, left_wheel_radius,
    right_wheel_radius);
odometry_.setVelocityRollingWindowSize(params_.velocity_rolling_window
_size);

cmd_vel_timeout_ =
    std::chrono::milliseconds{static_cast<int>(params_.cmd_vel_timeout *
    1000.0)};
publish_limited_velocity_ = params_.publish_limited_velocity;

limiter_linear_ = SpeedLimiter(
  params_.linear.x.has_velocity_limits,
      params_.linear.x.has_acceleration_limits,
  params_.linear.x.has_jerk_limits, params_.linear.x.min_velocity,
      params_.linear.x.max_velocity,
  params_.linear.x.min_acceleration, params_.linear.x.max_acceleration,
      params_.linear.x.min_jerk,
  params_.linear.x.max_jerk);

limiter_angular_ = SpeedLimiter(
  params_.angular.z.has_velocity_limits,
      params_.angular.z.has_acceleration_limits,
  params_.angular.z.has_jerk_limits, params_.angular.z.min_velocity,
  params_.angular.z.max_velocity, params_.angular.z.min_acceleration,
  params_.angular.z.max_acceleration, params_.angular.z.min_jerk,
      params_.angular.z.max_jerk);

if (!reset())
```

```
{
  return controller_interface::CallbackReturn::ERROR;
}

// Set the number of wheels per side
wheels_per_side_ = static_cast<int>(params_.left_wheel_names.size());

if (publish_limited_velocity_)
{
  limited_velocity_publisher_ =
      get_node()->create_publisher<Twist>(DEFAULT_COMMAND_OUT_TOPIC,
      rclcpp::SystemDefaultsQoS());
  realtime_limited_velocity_publisher_ =
      std::make_shared<realtime_tools::RealtimePublisher<Twist>>
  (limited_velocity_publisher_);
}

// Initialize the received velocity message
const Twist empty_twist;
received_velocity_msg_ptr_.set(std::make_shared<Twist>(empty_twist));

// Fill the previous commands queue with default commands
previous_commands_.emplace(empty_twist);
previous_commands_.emplace(empty_twist);

// Initialize the command subscriber
velocity_command_subscriber_ = get_node()->create_subscription<Twist>
(
  DEFAULT_COMMAND_TOPIC, rclcpp::SystemDefaultsQoS(),
  [this](const std::shared_ptr<Twist> msg) -> void
  {
    if (!subscriber_is_active_)
    {
      RCLCPP_WARN(get_node()->get_logger(), "Cannot accept new commands,
          subscriber is inactive");
      return;
    }
    if ((msg->header.stamp.sec == 0) && (msg->header.stamp.nanosec == 0))
    {
      RCLCPP_WARN_ONCE(get_node()->get_logger(), "Received TwistStamped with
          zero timestamp, setting to current time. This message will only be
          shown once");
      msg->header.stamp = get_node()->get_clock()->now();
    }
    received_velocity_msg_ptr_.set(std::move(msg));
  });

// Initialize the odometry publisher and message
odometry_publisher_ = get_node()->create_publisher<nav_msgs::msg::Odometry>
```

```
  (DEFAULT_ODOMETRY_TOPIC, rclcpp::SystemDefaultsQoS());
  realtime_odometry_publisher_ =
      std::make_shared<realtime_tools::RealtimePublisher<nav_msgs::msg::Odom
  etry>>(odometry_publisher_);

  // Initialize transform publisher and message
  odometry_transform_publisher_ =
      get_node()->create_publisher<tf2_msgs::msg::TFMessage>
  (DEFAULT_TRANSFORM_TOPIC, rclcpp::SystemDefaultsQoS());
  realtime_odometry_transform_publisher_ =
      std::make_shared<realtime_tools::RealtimePublisher<tf2_msgs::msg::TFMe
  ssage>>(odometry_transform_publisher_);

  // Initialize odometry values
  auto & odometry_message = realtime_odometry_publisher_->msg_;
  odometry_message.header.frame_id = params_.odom_frame_id;
  odometry_message.child_frame_id = params_.base_frame_id;
  odometry_message.twist =
      geometry_msgs::msg::TwistWithCovariance(rosidl_runtime_cpp::MessageIni
  tialization::ALL);
  for (size_t index = 0; index < 6; ++index)
  {
    odometry_message.pose.covariance[index * 7] =
        params_.pose_covariance_diagonal[index];
    odometry_message.twist.covariance[index * 7] =
        params_.twist_covariance_diagonal[index];
  }

  // Initialize transform values
  auto & transform =
      realtime_odometry_transform_publisher_->msg_.transforms.front();
  transform.header.frame_id = params_.odom_frame_id;
  transform.child_frame_id = params_.base_frame_id;

  previous_update_timestamp_ = get_node()->get_clock()->now();
  return controller_interface::CallbackReturn::SUCCESS;
}

// Callback for activating the controller
controller_interface::CallbackReturn DiffDriveController::on_activate(
  const rclcpp_lifecycle::State &)
{
  const auto left_result = configure_side("left", params_.left_wheel_names,
      registered_left_wheel_handles_);
  const auto right_result = configure_side("right", params_.right_wheel_names,
      registered_right_wheel_handles_);

  if (left_result == controller_interface::CallbackReturn::ERROR ||
      right_result == controller_interface::CallbackReturn::ERROR)
```

```
  {
    return controller_interface::CallbackReturn::ERROR;
  }

  if (registered_left_wheel_handles_.empty() ||
      registered_right_wheel_handles_.empty())
  {
    RCLCPP_ERROR(get_node()->get_logger(), "Either left or right wheel
        interfaces are missing");
    return controller_interface::CallbackReturn::ERROR;
  }

  is_halted = false;
  subscriber_is_active_ = true;

  RCLCPP_DEBUG(get_node()->get_logger(), "Subscriber and publisher are now
      active.");
  return controller_interface::CallbackReturn::SUCCESS;
}

// Callback for deactivating the controller
controller_interface::CallbackReturn DiffDriveController::on_deactivate(
  const rclcpp_lifecycle::State &)
{
  subscriber_is_active_ = false;
  if (!is_halted)
  {
    halt();
    is_halted = true;
  }
  registered_left_wheel_handles_.clear();
  registered_right_wheel_handles_.clear();
  return controller_interface::CallbackReturn::SUCCESS;
}

// Callback for cleaning up the controller
controller_interface::CallbackReturn DiffDriveController::on_cleanup(
  const rclcpp_lifecycle::State &)
{
  if (!reset())
  {
    return controller_interface::CallbackReturn::ERROR;
  }

  received_velocity_msg_ptr_.set(std::make_shared<Twist>());
  return controller_interface::CallbackReturn::SUCCESS;
}

// Callback for handling errors
```

```
controller_interface::CallbackReturn DiffDriveController::on_error(const
    rclcpp_lifecycle::State &)
{
  if (!reset())
  {
    return controller_interface::CallbackReturn::ERROR;
  }
  return controller_interface::CallbackReturn::SUCCESS;
}

// Reset the controller
bool DiffDriveController::reset()
{
  odometry_.resetOdometry();

  // Release the old queue
  std::queue<Twist> empty;
  std::swap(previous_commands_, empty);

  registered_left_wheel_handles_.clear();
  registered_right_wheel_handles_.clear();

  subscriber_is_active_ = false;
  velocity_command_subscriber_.reset();

  received_velocity_msg_ptr_.set(nullptr);
  is_halted = false;
  return true;
}

// Callback for shutting down the controller
controller_interface::CallbackReturn DiffDriveController::on_shutdown(
  const rclcpp_lifecycle::State &)
{
  return controller_interface::CallbackReturn::SUCCESS;
}

// Halt the controller
void DiffDriveController::halt()
{
  const auto halt_wheels = [](auto & wheel_handles)
  {
    for (const auto & wheel_handle : wheel_handles)
    {
      wheel_handle.velocity.get().set_value(0.0);
    }
  };

  halt_wheels(registered_left_wheel_handles_);
```

```
  halt_wheels(registered_right_wheel_handles_);
}

// Configure wheel handles for a side
controller_interface::CallbackReturn DiffDriveController::configure_side(
  const std::string & side, const std::vector<std::string> & wheel_names,
  std::vector<WheelHandle> & registered_handles)
{
  auto logger = get_node()->get_logger();

  if (wheel_names.empty())
  {
    RCLCPP_ERROR(logger, "No '%s' wheel names specified", side.c_str());
    return controller_interface::CallbackReturn::ERROR;
  }

  // Register handles
  registered_handles.reserve(wheel_names.size());
  for (const auto & wheel_name : wheel_names)
  {
    const auto interface_name = feedback_type();
    const auto state_handle = std::find_if(
      state_interfaces_.cbegin(), state_interfaces_.cend(),
      [&wheel_name, &interface_name](const auto & interface)
      {
        return interface.get_prefix_name() == wheel_name &&
               interface.get_interface_name() == interface_name;
      });

    if (state_handle == state_interfaces_.cend())
    {
      RCLCPP_ERROR(logger, "Unable to obtain joint state handle for %s",
          wheel_name.c_str());
      return controller_interface::CallbackReturn::ERROR;
    }

    const auto command_handle = std::find_if(
      command_interfaces_.begin(), command_interfaces_.end(),
      [&wheel_name](const auto & interface)
      {
        return interface.get_prefix_name() == wheel_name &&
               interface.get_interface_name() == HW_IF_VELOCITY;
      });

    if (command_handle == command_interfaces_.end())
    {
      RCLCPP_ERROR(logger, "Unable to obtain joint command handle for %s",
          wheel_name.c_str());
      return controller_interface::CallbackReturn::ERROR;
```

```
    }

    registered_handles.emplace_back(
      WheelHandle{std::ref(*state_handle), std::ref(*command_handle)});
  }

  return controller_interface::CallbackReturn::SUCCESS;
}
}  // namespace diff_drive_controller

#include "class_loader/register_macro.hpp"

// Register the controller with the class loader
CLASS_LOADER_REGISTER_CLASS(diff_drive_controller::DiffDriveController,
    controller_interface::ControllerInterface)
```

5.10.4 Differential drive kinematics

Differential drive kinematics is a widely used method for controlling mobile robots equipped with two wheels on a single axis. The following section explains how thethe linear and angular velocities of such a robot are converted into the velocities of the left and right wheels in the above code.

Definitions

- v: Linear velocity of the robot (m/s) (linear command - linear.x velocity)
- ω: Angular velocity of the robot (rad/s)
- R: Radius of the wheels (m) (angular command - angular.z velocity)
- L: Distance between the two wheels (wheelbase) (m)
- v_l: Linear velocity of the left wheel (m/s)
- v_r: Linear velocity of the right wheel (m/s)

Formulas

The relationship between the robot's linear and angular velocities and the velocities of its wheels can be described by the following equations:

Wheel Velocities

Given the linear velocity v and angular velocity ω of the robot, the velocities of the left and right wheels (v_l and v_r) can be calculated as follows:

$$v_l = v - \frac{L}{2} \cdot \omega, \tag{5.1}$$

$$v_r = v + \frac{L}{2} \cdot \omega. \tag{5.2}$$

Robot Velocities

Conversely, if the velocities of the left and right wheels are known, the linear and angular velocities of the robot can be computed by:

$$v = \frac{v_l + v_r}{2}, \tag{5.3}$$

$$\omega = \frac{v_r - v_l}{L}. \tag{5.4}$$

5.10.5 Practical Implementation

Setting Up ros2_control in Gazebo Simulation

To illustrate the implementation of ros2_control, we begin by setting it up in a Gazebo simulation environment. This involves several steps:

Installation

```
sudo apt install ros-foxy-ros2-control ros-foxy-ros2-controllers \
ros-foxy-gazebo-ros2-control
```

Updating URDF

Replace existing Gazebo control tags with ros2_control tags. Define the hardware interfaces and specify the command and state interfaces for each joint.

Controller Configuration

Create a YAML file to define the controllers and their parameters, such as update rates and specific joint configurations. Include the path to this configuration file in the URDF.

Launching the Simulation

Modify the launch file to start the controller manager and spawn the robot in Gazebo. Use ROS commands to list and start the controllers.

5.10.6 Running and Managing Controllers

Once the configurations are in place, the simulation can be launched, and the controllers managed using ROS commands. For example:

```
ros2 control list_hardware_interfaces
ros2 control list_controllers
ros2 run controller_manager spawner.py diff_drive_controller
ros2 run controller_manager spawner.py joint_state_broadcaster
```

These commands ensure that the correct controllers are running, allowing for real-time interaction with the simulated robot.

ros2_control provides a comprehensive and efficient framework for managing robotic control systems. By standardizing hardware interfaces and controllers, it allows for modular, reusable, and maintainable control systems, applicable to both simulated and real-world robots. Integrating ros2_control into your projects can significantly enhance development efficiency and control precision, making it an essential tool for modern robotics.

In our project, we did not use ros2_control in the Gazebo simulation, since our goal was to drive both the virtual robot and the real robot at the same time, there was difficulty in using ros2_control for both virtual robot and real robot. Therefore we stick with the previously written "gazebo_control.xacro" file for the Gazebo system. We only use ros2_control for the real robot.

5.11 Moving a real robot using ros2_control

This section provides a detailed guide on using ros2_control to drive a real robot, focusing on the practical application and configuration necessary to transition from a simulated environment to a physical one.

5.11.1 Overview of the Control System

For controlling a physical robot, the architecture involves the following components:

- **Physical Robot:** Equipped with two drive wheels controlled via velocity commands.
- **Command Velocity:** A Twist message containing the desired velocities in linear X and angular Z directions.
- **Controller Manager:** A node managing controllers and linking command velocities to hardware interfaces.
- **Differential Drive Controller:** Converts command velocities into wheel velocities.
- **Hardware Interface:** Translates wheel velocities into motor commands.
- **Joint State Broadcaster:** Publishes motor encoder positions for generating transforms.

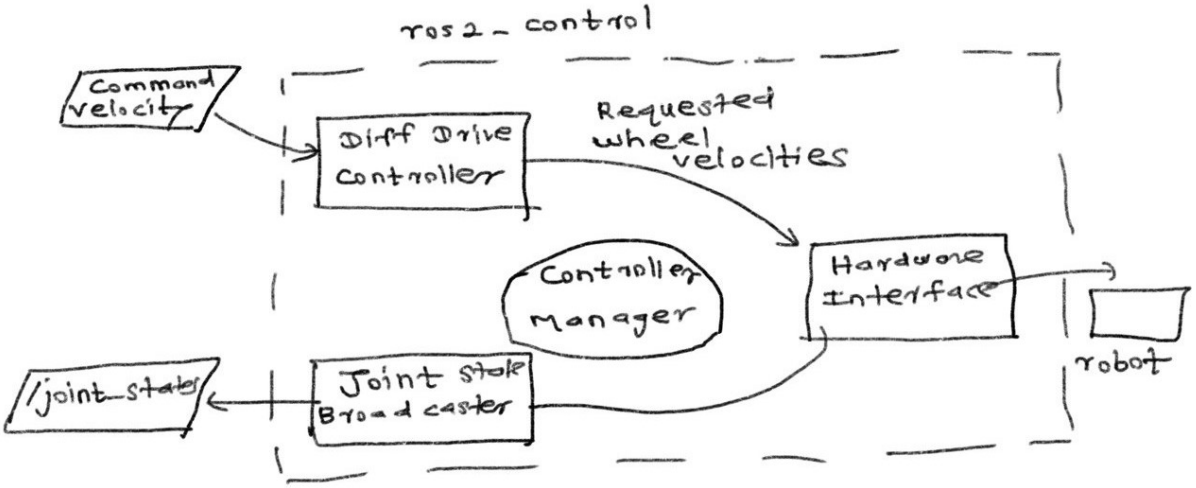


Figure 5.7: ros2_control used in real robot

5.11.2 Setting Up the Hardware Interface

Installing Necessary Packages

Before configuring the hardware interface, we need two more additional packages "serial" and "diffdrive_arduino" package.

5.11.3 diffdrive_arduino

Purpose: This package is essential for converting velocity commands from ROS2 into PWM signal commands that can be understood by the Arduino, which then drives the motors. It plays a critical role in ensuring that the physical movements of the robot correspond accurately to the commands issued in the simulation.

Functionality: It takes wheel velocities from "diff_drive_controller" and translates them into PWM values which are sent via serial communication to the Arduino.

The needed source code of the "diffdrive_arduino" package is given below. Use this code and create the package and build the package as intructed earlier.

```
/* diffdrive_arduino.cpp */

/* BSD 3-Clause License

Copyright (c) 2020, Josh Newans
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
    this
   list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its
   contributors may be used to endorse or promote products derived from
   this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
    ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
```

```
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

/* Modified by Sundarbavan T. */

#include "diffdrive_arduino/diffdrive_arduino.h"
#include "hardware_interface/types/hardware_interface_type_values.hpp"

// Constructor
DiffDriveArduino::DiffDriveArduino()
    : logger_(rclcpp::get_logger("DiffDriveArduino"))
{}

// Initialization callback
hardware_interface::CallbackReturn DiffDriveArduino::on_init(const
    hardware_interface::HardwareInfo & info)
{
  // Call the base class on_init method
  if (hardware_interface::SystemInterface::on_init(info) !=
      CallbackReturn::SUCCESS)
  {
    return CallbackReturn::ERROR;
  }

  RCLCPP_INFO(logger_, "Configuring...");

  // Initialize the current time
  time_ = std::chrono::system_clock::now();

  // Read hardware parameters from the configuration
  cfg_.left_wheel_name = info_.hardware_parameters["left_wheel_name"];
  cfg_.right_wheel_name = info_.hardware_parameters["right_wheel_name"];
  cfg_.loop_rate = std::stof(info_.hardware_parameters["loop_rate"]);
  cfg_.device = info_.hardware_parameters["device"];
  cfg_.baud_rate = std::stoi(info_.hardware_parameters["baud_rate"]);
  cfg_.timeout = std::stoi(info_.hardware_parameters["timeout"]);
  cfg_.enc_counts_per_rev_left =
      std::stoi(info_.hardware_parameters["enc_counts_per_rev_left"]);
  cfg_.enc_counts_per_rev_right =
      std::stoi(info_.hardware_parameters["enc_counts_per_rev_right"]);

  // Set up the wheels with their respective names and encoder counts
  l_wheel_.setup(cfg_.left_wheel_name, cfg_.enc_counts_per_rev_left);
  r_wheel_.setup(cfg_.right_wheel_name, cfg_.enc_counts_per_rev_right);

  // Set up the Arduino with the device configuration
  arduino_.setup(cfg_.device, cfg_.baud_rate, cfg_.timeout);
```

```
  RCLCPP_INFO(logger_, "Finished Configuration");

  return CallbackReturn::SUCCESS;
}

// Export state interfaces for the wheels
std::vector<hardware_interface::StateInterface>
    DiffDriveArduino::export_state_interfaces()
{
  std::vector<hardware_interface::StateInterface> state_interfaces;

  // Create velocity and position state interfaces for each wheel
  state_interfaces.emplace_back(hardware_interface::StateInterface(
  l_wheel_.name, hardware_interface::HW_IF_VELOCITY, &l_wheel_.vel));
  state_interfaces.emplace_back(hardware_interface::StateInterface
  (l_wheel_.name, hardware_interface::HW_IF_POSITION, &l_wheel_.pos));
  state_interfaces.emplace_back(hardware_interface::StateInterface
  (r_wheel_.name, hardware_interface::HW_IF_VELOCITY, &r_wheel_.vel));
  state_interfaces.emplace_back(hardware_interface::StateInterface
  (r_wheel_.name, hardware_interface::HW_IF_POSITION, &r_wheel_.pos));

  return state_interfaces;
}

// Export command interfaces for the wheels
std::vector<hardware_interface::CommandInterface>
    DiffDriveArduino::export_command_interfaces()
{
  std::vector<hardware_interface::CommandInterface> command_interfaces;

  // Create velocity command interfaces for each wheel
  command_interfaces.emplace_back(hardware_interface::CommandInterface
  (l_wheel_.name, hardware_interface::HW_IF_VELOCITY, &l_wheel_.cmd));
  command_interfaces.emplace_back(hardware_interface::CommandInterface
  (r_wheel_.name, hardware_interface::HW_IF_VELOCITY, &r_wheel_.cmd));

  return command_interfaces;
}

// Activation callback
hardware_interface::CallbackReturn DiffDriveArduino::on_activate(const
    rclcpp_lifecycle::State & /*previous_state*/)
{
  RCLCPP_INFO(logger_, "Starting Controller...");

  // Send an empty message to Arduino and set PID values
  arduino_.sendEmptyMsg();
  arduino_.setPidValues(30, 20, 0, 100);
```

```
  return CallbackReturn::SUCCESS;
}

// Deactivation callback
hardware_interface::CallbackReturn DiffDriveArduino::on_deactivate(const
    rclcpp_lifecycle::State & /*previous_state*/)
{
  RCLCPP_INFO(logger_, "Stopping Controller...");

  return CallbackReturn::SUCCESS;
}

// Read function to update wheel states
hardware_interface::return_type DiffDriveArduino::read(
  const rclcpp::Time & /*time*/, const rclcpp::Duration & /*period*/)
{
  // Calculate time delta
  auto new_time = std::chrono::system_clock::now();
  std::chrono::duration<double> diff = new_time - time_;
  double deltaSeconds = diff.count();
  time_ = new_time;

  // Check if Arduino is connected
  if (!arduino_.connected())
  {
    return return_type::ERROR;
  }

  // Read encoder values from Arduino
  arduino_.readEncoderValues(l_wheel_.enc, r_wheel_.enc);

  // Update left wheel position and velocity
  double pos_prev = l_wheel_.pos;
  l_wheel_.pos = l_wheel_.calcEncAngle();
  l_wheel_.vel = (l_wheel_.pos - pos_prev) / deltaSeconds;

  // Update right wheel position and velocity
  pos_prev = r_wheel_.pos;
  r_wheel_.pos = r_wheel_.calcEncAngle();
  r_wheel_.vel = (r_wheel_.pos - pos_prev) / deltaSeconds;

  return return_type::OK;
}

// Write function to send commands to the wheels
hardware_interface::return_type DiffDriveArduino::write(
  const rclcpp::Time & /*time*/, const rclcpp::Duration & /*period*/)
{
  if (!arduino_.connected())
```

```
  {
    return return_type::ERROR;
  }

  // Example scaling factors, these should be adjusted based on your hardware
      specifics
  const double max_velocity = 1.0;  // Max velocity in rad/s that corresponds
      to max PWM (needs to calcaulted since we are using open loop control)
  const int max_pwm = 255;  // Maximum PWM value
  double left_pwm = (l_wheel_.cmd / max_velocity) * max_pwm;
  double right_pwm = (r_wheel_.cmd / max_velocity) * max_pwm;

  // Clamping the PWM values to ensure they are within the valid range
  left_pwm = std::clamp(left_pwm, static_cast<double>(0),
      static_cast<double>(max_pwm));
  right_pwm = std::clamp(right_pwm, static_cast<double>(0),
      static_cast<double>(max_pwm));

  // Send PWM values to Arduino
  arduino_.setMotorValues(static_cast<int>(left_pwm),
      static_cast<int>(right_pwm));

  return return_type::OK;
}

// Register the plugin with the class loader
#include "pluginlib/class_list_macros.hpp"
PLUGINLIB_EXPORT_CLASS(
  DiffDriveArduino,
  hardware_interface::SystemInterface
)
```

Given below are the used header files for above source code.

```
// arduino_comms.h

#ifndef DIFFDRIVE_ARDUINO_ARDUINO_COMMS_H
#define DIFFDRIVE_ARDUINO_ARDUINO_COMMS_H

#include <serial/serial.h>
#include <cstring>

class ArduinoComms
{
public:
  // Default constructor
  ArduinoComms() { }
```

```
  // Constructor with parameters for serial device, baud rate, and timeout
  ArduinoComms(const std::string &serial_device, int32_t baud_rate, int32_t
      timeout_ms)
      : serial_conn_(serial_device, baud_rate,
          serial::Timeout::simpleTimeout(timeout_ms))
  { }

  // Setup function to initialize the serial connection
  void setup(const std::string &serial_device, int32_t baud_rate, int32_t
      timeout_ms);

  // Function to send an empty message to Arduino
  void sendEmptyMsg();

  // Function to read encoder values from Arduino
  void readEncoderValues(int &val_1, int &val_2);

  // Function to set motor values
  void setMotorValues(int val_1, int val_2);

  // Function to set PID values
  void setPidValues(float k_p, float k_d, float k_i, float k_o);

  // Function to check if the serial connection is open
  bool connected() const { return serial_conn_.isOpen(); }

  // Function to send a message to Arduino and optionally print the output
  std::string sendMsg(const std::string &msg_to_send, bool print_output =
      false);

private:
  serial::Serial serial_conn_;  ///< Underlying serial connection
};

#endif // DIFFDRIVE_ARDUINO_ARDUINO_COMMS_H
```

```
// config.h

#ifndef DIFFDRIVE_ARDUINO_CONFIG_H
#define DIFFDRIVE_ARDUINO_CONFIG_H

#include <string>

// Configuration struct for the differential drive Arduino setup
struct Config
{
```

```
  std::string left_wheel_name = "left_wheel";       // Name of the left wheel
  std::string right_wheel_name = "right_wheel";     // Name of the right wheel
  float loop_rate = 30;                             // Control loop rate in Hz
  std::string device = "/dev/ttyUSB0";              // Serial device path for
      Arduino
  int baud_rate = 57600;                            // Baud rate for serial
      communication
  int timeout = 1000;                               // Timeout for serial
      communication in milliseconds
  int enc_counts_per_rev_left = 6533;               // Encoder counts per
      revolution for the left wheel
  int enc_counts_per_rev_right = 6533;              // Encoder counts per
      revolution for the right wheel
};

#endif // DIFFDRIVE_ARDUINO_CONFIG_H
```

```
// diffdrive_arduinio.h

#ifndef DIFFDRIVE_ARDUINO_REAL_ROBOT_H
#define DIFFDRIVE_ARDUINO_REAL_ROBOT_H

#include <cstring>
#include "rclcpp/rclcpp.hpp"

#include "hardware_interface/system_interface.hpp"
#include "hardware_interface/handle.hpp"
#include "hardware_interface/hardware_info.hpp"
#include "hardware_interface/types/hardware_interface_return_values.hpp"
#include "rclcpp_lifecycle/state.hpp"

#include "config.h"
#include "wheel.h"
#include "arduino_comms.h"

using hardware_interface::return_type;

// Class for the differential drive Arduino interface
class DiffDriveArduino : public hardware_interface::SystemInterface
{
public:
  // Constructor
  DiffDriveArduino();

  // Override initialization callback
  CallbackReturn on_init(const hardware_interface::HardwareInfo & info)
      override;
```

```
  // Export state interfaces for the wheels
  std::vector<hardware_interface::StateInterface> export_state_interfaces()
      override;

  // Export command interfaces for the wheels
  std::vector<hardware_interface::CommandInterface>
      export_command_interfaces() override;

  // Override activation callback
  CallbackReturn on_activate(const rclcpp_lifecycle::State & previous_state)
      override;

  // Override deactivation callback
  CallbackReturn on_deactivate(const rclcpp_lifecycle::State & previous_state)
      override;

  // Read function to update wheel states
  hardware_interface::return_type read(
    const rclcpp::Time & time, const rclcpp::Duration & period) override;

  // Write function to send commands to the wheels
  hardware_interface::return_type write(
    const rclcpp::Time & time, const rclcpp::Duration & period) override;

private:
  Config cfg_;  // Configuration for the differential drive system
  ArduinoComms arduino_;  // Arduino communication interface
  Wheel l_wheel_;  // Left wheel
  Wheel r_wheel_;  // Right wheel
  rclcpp::Logger logger_;  // Logger for the class
  std::chrono::time_point<std::chrono::system_clock> time_;  // Time point for
      tracking time differences
};

#endif // DIFFDRIVE_ARDUINO_REAL_ROBOT_H
```

```
// wheels.h

#ifndef DIFFDRIVE_ARDUINO_WHEEL_H
#define DIFFDRIVE_ARDUINO_WHEEL_H

#include <string>

// Class representing a wheel of the differential drive robot
class Wheel
{
public:
    // Public member variables representing the state and command of the
        wheel
```

```
    std::string name = "";        // Name of the wheel
    int enc = 0;                  // Encoder value
    double cmd = 0;               // Command value (e.g., velocity command)
    double pos = 0;               // Position of the wheel
    double vel = 0;               // Velocity of the wheel
    double eff = 0;               // Effort applied to the wheel (not used in
        this example)
    double velSetPt = 0;          // Velocity setpoint (not used in this
        example)
    double rads_per_count = 0;    // Radians per encoder count

    // Default constructor
    Wheel() = default;

    // Constructor with parameters for wheel name and encoder counts per
        revolution
    Wheel(const std::string &wheel_name, int counts_per_rev);

    // Setup function to initialize the wheel with its name and encoder counts
        per revolution
    void setup(const std::string &wheel_name, int counts_per_rev);

    // Function to calculate the angle of the wheel based on the encoder
        value
    double calcEncAngle();
};

#endif // DIFFDRIVE_ARDUINO_WHEEL_H
```

5.11.4 serial

Purpose: Handles the serial communication between the ROS2 environment and the Arduino. This package is crucial for the transmission of PWM signal commands and receiving feedback if necessary.

Functionality: It manages the data flow over the USB connection, ensuring that messages are properly sent and received without errors or delays.

The needed source code of the "serial" package is given below. Use this code and create the package and build the package as intructed earlier.

```
/* serial.cc */
/* The MIT License

Copyright (c) 2012 William Woodall, John Harrison

Permission is hereby granted, free of charge, to any person obtaining a copy
    of this software and associated documentation files (the "Software"), to
    deal in the Software without restriction, including without limitation the
    rights to use, copy, modify, merge, publish, distribute, sublicense,
    and/or sell copies of the Software, and to permit persons to whom the
    Software is furnished to do so, subject to the following conditions:
```

```
The above copyright notice and this permission notice shall be included in all
    copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
    IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
    FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
    THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
    LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
    FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
    DEALINGS IN THE SOFTWARE.

 */

#include <algorithm>

#if !defined(_WIN32) && !defined(__OpenBSD__) && !defined(__FreeBSD__)
# include <alloca.h>
#endif

#if defined (__MINGW32__)
# define alloca __builtin_alloca
#endif

#include "serial/serial.h"

#ifdef _WIN32
#include "serial/impl/win.h"
#else
#include "serial/impl/unix.h"
#endif

using std::invalid_argument;
using std::min;
using std::numeric_limits;
using std::vector;
using std::size_t;
using std::string;

using serial::Serial;
using serial::SerialException;
using serial::IOException;
using serial::bytesize_t;
using serial::parity_t;
using serial::stopbits_t;
using serial::flowcontrol_t;

class Serial::ScopedReadLock {
public:
```

```
  ScopedReadLock(SerialImpl *pimpl) : pimpl_(pimpl) {
    this->pimpl_->readLock();
  }
  ~ScopedReadLock() {
    this->pimpl_->readUnlock();
  }
private:
  // Disable copy constructors
  ScopedReadLock(const ScopedReadLock&);
  const ScopedReadLock& operator=(ScopedReadLock);

  SerialImpl *pimpl_;
};

class Serial::ScopedWriteLock {
public:
  ScopedWriteLock(SerialImpl *pimpl) : pimpl_(pimpl) {
    this->pimpl_->writeLock();
  }
  ~ScopedWriteLock() {
    this->pimpl_->writeUnlock();
  }
private:
  // Disable copy constructors
  ScopedWriteLock(const ScopedWriteLock&);
  const ScopedWriteLock& operator=(ScopedWriteLock);
  SerialImpl *pimpl_;
};

Serial::Serial (const string &port, uint32_t baudrate, serial::Timeout
    timeout,
                bytesize_t bytesize, parity_t parity, stopbits_t stopbits,
                flowcontrol_t flowcontrol)
 : pimpl_(new SerialImpl (port, baudrate, bytesize, parity,
                                          stopbits, flowcontrol))
{
  pimpl_->setTimeout(timeout);
}

Serial::~Serial ()
{
  delete pimpl_;
}

void
Serial::open ()
{
  pimpl_->open ();
}
```

```
void
Serial::close ()
{
  pimpl_->close ();
}

bool
Serial::isOpen () const
{
  return pimpl_->isOpen ();
}

size_t
Serial::available ()
{
  return pimpl_->available ();
}

bool
Serial::waitReadable ()
{
  serial::Timeout timeout(pimpl_->getTimeout ());
  return pimpl_->waitReadable(timeout.read_timeout_constant);
}

void
Serial::waitByteTimes (size_t count)
{
  pimpl_->waitByteTimes(count);
}

size_t
Serial::read_ (uint8_t *buffer, size_t size)
{
  return this->pimpl_->read (buffer, size);
}

size_t
Serial::read (uint8_t *buffer, size_t size)
{
  ScopedReadLock lock(this->pimpl_);
  return this->pimpl_->read (buffer, size);
}

size_t
Serial::read (std::vector<uint8_t> &buffer, size_t size)
{
  ScopedReadLock lock(this->pimpl_);
```

```
  uint8_t *buffer_ = new uint8_t[size];
  size_t bytes_read = 0;

  try {
    bytes_read = this->pimpl_->read (buffer_, size);
  }
  catch (const std::exception &e) {
    delete[] buffer_;
    throw;
  }

  buffer.insert (buffer.end (), buffer_, buffer_+bytes_read);
  delete[] buffer_;
  return bytes_read;
}

size_t
Serial::read (std::string &buffer, size_t size)
{
  ScopedReadLock lock(this->pimpl_);
  uint8_t *buffer_ = new uint8_t[size];
  size_t bytes_read = 0;
  try {
    bytes_read = this->pimpl_->read (buffer_, size);
  }
  catch (const std::exception &e) {
    delete[] buffer_;
    throw;
  }
  buffer.append (reinterpret_cast<const char*>(buffer_), bytes_read);
  delete[] buffer_;
  return bytes_read;
}

string
Serial::read (size_t size)
{
  std::string buffer;
  this->read (buffer, size);
  return buffer;
}

size_t
Serial::readline (string &buffer, size_t size, string eol)
{
  ScopedReadLock lock(this->pimpl_);
  size_t eol_len = eol.length ();
  uint8_t *buffer_ = static_cast<uint8_t*>
                              (alloca (size * sizeof (uint8_t)));
```

```
  size_t read_so_far = 0;
  while (true)
  {
    size_t bytes_read = this->read_ (buffer_ + read_so_far, 1);
    read_so_far += bytes_read;
    if (bytes_read == 0) {
      break; // Timeout occured on reading 1 byte
    }
    if (string (reinterpret_cast<const char*>
         (buffer_ + read_so_far - eol_len), eol_len) == eol) {
      break; // EOL found
    }
    if (read_so_far == size) {
      break; // Reached the maximum read length
    }
  }
  buffer.append(reinterpret_cast<const char*> (buffer_), read_so_far);
  return read_so_far;
}

string
Serial::readline (size_t size, string eol)
{
  std::string buffer;
  this->readline (buffer, size, eol);
  return buffer;
}

vector<string>
Serial::readlines (size_t size, string eol)
{
  ScopedReadLock lock(this->pimpl_);
  std::vector<std::string> lines;
  size_t eol_len = eol.length ();
  uint8_t *buffer_ = static_cast<uint8_t*>
    (alloca (size * sizeof (uint8_t)));
  size_t read_so_far = 0;
  size_t start_of_line = 0;
  while (read_so_far < size) {
    size_t bytes_read = this->read_ (buffer_+read_so_far, 1);
    read_so_far += bytes_read;
    if (bytes_read == 0) {
      if (start_of_line != read_so_far) {
        lines.push_back (
          string (reinterpret_cast<const char*> (buffer_ + start_of_line),
            read_so_far - start_of_line));
      }
      break; // Timeout occured on reading 1 byte
    }
```

```
    if (string (reinterpret_cast<const char*>
         (buffer_ + read_so_far - eol_len), eol_len) == eol) {
      // EOL found
      lines.push_back(
        string(reinterpret_cast<const char*> (buffer_ + start_of_line),
          read_so_far - start_of_line));
      start_of_line = read_so_far;
    }
    if (read_so_far == size) {
      if (start_of_line != read_so_far) {
        lines.push_back(
          string(reinterpret_cast<const char*> (buffer_ + start_of_line),
            read_so_far - start_of_line));
      }
      break; // Reached the maximum read length
    }
  }
  return lines;
}

size_t
Serial::write (const string &data)
{
  ScopedWriteLock lock(this->pimpl_);
  return this->write_ (reinterpret_cast<const uint8_t*>(data.c_str()),
                       data.length());
}

size_t
Serial::write (const std::vector<uint8_t> &data)
{
  ScopedWriteLock lock(this->pimpl_);
  return this->write_ (&data[0], data.size());
}

size_t
Serial::write (const uint8_t *data, size_t size)
{
  ScopedWriteLock lock(this->pimpl_);
  return this->write_(data, size);
}

size_t
Serial::write_ (const uint8_t *data, size_t length)
{
  return pimpl_->write (data, length);
}

void
```

```
Serial::setPort (const string &port)
{
  ScopedReadLock rlock(this->pimpl_);
  ScopedWriteLock wlock(this->pimpl_);
  bool was_open = pimpl_->isOpen ();
  if (was_open) close();
  pimpl_->setPort (port);
  if (was_open) open ();
}

string
Serial::getPort () const
{
  return pimpl_->getPort ();
}

void
Serial::setTimeout (serial::Timeout &timeout)
{
  pimpl_->setTimeout (timeout);
}

serial::Timeout
Serial::getTimeout () const {
  return pimpl_->getTimeout ();
}

void
Serial::setBaudrate (uint32_t baudrate)
{
  pimpl_->setBaudrate (baudrate);
}

uint32_t
Serial::getBaudrate () const
{
  return uint32_t(pimpl_->getBaudrate ());
}

void
Serial::setBytesize (bytesize_t bytesize)
{
  pimpl_->setBytesize (bytesize);
}

bytesize_t
Serial::getBytesize () const
{
  return pimpl_->getBytesize ();
```

```
}

void
Serial::setParity (parity_t parity)
{
  pimpl_->setParity (parity);
}

parity_t
Serial::getParity () const
{
  return pimpl_->getParity ();
}

void
Serial::setStopbits (stopbits_t stopbits)
{
  pimpl_->setStopbits (stopbits);
}

stopbits_t
Serial::getStopbits () const
{
  return pimpl_->getStopbits ();
}

void
Serial::setFlowcontrol (flowcontrol_t flowcontrol)
{
  pimpl_->setFlowcontrol (flowcontrol);
}

flowcontrol_t
Serial::getFlowcontrol () const
{
  return pimpl_->getFlowcontrol ();
}

void Serial::flush ()
{
  ScopedReadLock rlock(this->pimpl_);
  ScopedWriteLock wlock(this->pimpl_);
  pimpl_->flush ();
}

void Serial::flushInput ()
{
  ScopedReadLock lock(this->pimpl_);
  pimpl_->flushInput ();
```

```
}

void Serial::flushOutput ()
{
  ScopedWriteLock lock(this->pimpl_);
  pimpl_->flushOutput ();
}

void Serial::sendBreak (int duration)
{
  pimpl_->sendBreak (duration);
}

void Serial::setBreak (bool level)
{
  pimpl_->setBreak (level);
}

void Serial::setRTS (bool level)
{
  pimpl_->setRTS (level);
}

void Serial::setDTR (bool level)
{
  pimpl_->setDTR (level);
}

bool Serial::waitForChange()
{
  return pimpl_->waitForChange();
}

bool Serial::getCTS ()
{
  return pimpl_->getCTS ();
}

bool Serial::getDSR ()
{
  return pimpl_->getDSR ();
}

bool Serial::getRI ()
{
  return pimpl_->getRI ();
}

bool Serial::getCD ()
```

```
{
  return pimpl_->getCD ();
}
```

Configuring the Hardware Interface

Edit the ros2_control section of the URDF file to include the hardware interface for the real robot:

Inorder to add the ros2_control section, let's create ros2_control2.xacro file and include that in the main robot.urdf.xacro file as mentioned before.

```
<!-- ros2_control2.xacro -->

<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro">

    <!-- ROS 2 Control System Configuration for the RealRobot -->
    <ros2_control name="RealRobot" type="system">
        <hardware>
            <!-- Plugin for the DiffDriveArduino hardware interface -->
            <plugin>diffdrive_arduino/DiffDriveArduino</plugin>

            <!-- Hardware parameters for the differential drive system -->
            <param name="left_wheel_name">left_wheel_joint</param>
            <param name="right_wheel_name">right_wheel_joint</param>
            <param name="loop_rate">30</param>                  <!-- Control
                loop rate in Hz -->
            <param name="device">/dev/ttyUSB0</param>           <!-- Serial
                device path for Arduino -->
            <param name="baud_rate">9600</param>               <!-- Baud rate
                for serial communication -->
            <param name="timeout">1000</param>                  <!-- Timeout
                for serial communication in milliseconds -->
        </hardware>

        <!-- Joint configuration for the left wheel -->
        <joint name="left_wheel_joint">
            <!-- Command interface for controlling the velocity of the left
                wheel -->
            <command_interface name="velocity">
                <param name="min">-10</param>                   <!-- Minimum
                    velocity command -->
                <param name="max">10</param>                    <!-- Maximum
                    velocity command -->
            </command_interface>
            <!-- State interfaces for reading the velocity and position of the
                left wheel -->
```

```
            <state_interface name="velocity" />
            <state_interface name="position" />
        </joint>

        <!-- Joint configuration for the right wheel -->
        <joint name="right_wheel_joint">
            <!-- Command interface for controlling the velocity of the right
                wheel -->
            <command_interface name="velocity">
                <param name="min">-10</param>                  <!-- Minimum
                    velocity command -->
                <param name="max">10</param>                   <!-- Maximum
                    velocity command -->
            </command_interface>
            <!-- State interfaces for reading the velocity and position of the
                right wheel -->
            <state_interface name="velocity" />
            <state_interface name="position" />
        </joint>
    </ros2_control>
</robot>
```

The configuration for the controller is given below, we use an .yaml file to configure the controller

```
# my_controllers2.yaml

controller_manager:
  ros__parameters:
    update_rate: 30  # Update rate for the controller manager in Hz

    # Configuration for the differential drive controller
    diff_cont2:
      type: diff_drive_controller/DiffDriveController  # Type of the
          controller

diff_cont2:
  ros__parameters:

    publish_rate: 50.0  # Rate at which to publish odometry and transform
        information in Hz

    base_frame_id: base_link  # Frame ID for the base link of the robot

    # Names of the joints for the left and right wheels
    left_wheel_names: ['left_wheel_joint']
    right_wheel_names: ['right_wheel_joint']
```

```
    wheel_separation: 0.3  # Distance between the left and right wheels in
        meters
    wheel_radius: 0.035  # Radius of the wheels in meters

    use_stamped_vel: false  # Whether to use stamped velocity messages

    odom_frame_id: "/odom2"  # Frame ID for the odometry frame

    enable_odom_tf: false  # Whether to publish the transform from the
        odometry frame to the base frame
```

5.11.5 Setting Up the Launch File

Update the existing "launch_robot_sim.launch.py" launch file to start the controller manager and hardware interface on the real robot:

```
import os
from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch.actions import IncludeLaunchDescription, TimerAction,
    RegisterEventHandler
from launch.launch_description_sources import PythonLaunchDescriptionSource
from launch.substitutions import Command
from launch.event_handlers import OnProcessStart

from launch_ros.actions import Node

def generate_launch_description():
    package_name = 'my_bot'  # Name of your package

    # Include the robot_state_publisher launch file from your package
    rsp = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([
            os.path.join(get_package_share_directory(package_name), 'launch',
                'rsp.launch.py')
        ]),
        launch_arguments={'use_sim_time': 'false', 'use_ros2_control':
            'true'}.items()
    )

    # Load the Gazebo parameters file
    gazebo_params_file =
        os.path.join(get_package_share_directory(package_name), 'config',
        'gazebo_params.yaml')

    # Include the Gazebo launch file from the gazebo_ros package
```

```
    gazebo = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([
            os.path.join(get_package_share_directory('gazebo_ros'), 'launch',
                'gazebo.launch.py')
        ]),
        launch_arguments={'extra_gazebo_args': '--ros-args --params-file ' +
            gazebo_params_file}.items()
    )

    # Node to spawn the robot entity in Gazebo
    spawn_entity = Node(
        package='gazebo_ros',
        executable='spawn_entity.py',
        arguments=['-topic', 'robot_description', '-entity', 'my_bot'],
        output='screen'
    )

    # Command to get the robot description parameter
    robot_description = Command(['ros2 param get --hide-type
        /robot_state_publisher robot_description'])

    # Load the controller parameters file
    controller_params_file =
        os.path.join(get_package_share_directory(package_name), 'config',
        'my_controllers2.yaml')

    # Node for the controller manager
    controller_manager2 = Node(
        package="controller_manager",
        executable="ros2_control_node",
        parameters=[{'robot_description': robot_description},
            controller_params_file]
    )

    # Delay the start of the controller manager
    delayed_controller_manager = TimerAction(period=3.0,
        actions=[controller_manager2])

    # Node to spawn the differential drive controller
    diff_drive_spawner = Node(
        package="controller_manager",
        executable="spawner",
        arguments=["diff_cont2"],
    )

    # Delay the start of the differential drive spawner until the controller
        manager starts
    delayed_diff_drive_spawner = RegisterEventHandler(
        event_handler=OnProcessStart(
```

```
            target_action=controller_manager2,
            on_start=[diff_drive_spawner],
        )
    )

    # Launch all the nodes and actions
    return LaunchDescription([
        rsp,
        twist_mux,
        delayed_controller_manager,
        delayed_diff_drive_spawner,
        spawn_entity,
        gazebo
    ])
```

5.11.6 Testing the Configuration

Prop the robot to prevent it from moving unexpectedly during initial tests. Run the launch file and ensure all components are working correctly:

```
ros2 launch my_bot launch_robot_sim.launch.py
```

Verify that the controllers are started and the robot can be controlled using teleop_twist_keyboard:

```
ros2 run tele teleop_twist_keyboard
```

By following these steps, we can configure and use ros2_control to drive a real robot, transitioning from a simulation environment to physical hardware. This setup provides a robust and flexible control system, leveraging the powerful features of ros2_control to manage complex robotic behaviors efficiently.

5.12 Interaction Between diff_drive_controller and diff-drive_arduino

5.12.1 Control Flow

Command Reception and Processing

The `diff_drive_controller` receives velocity commands (cmd_vel topics) which dictate how the robot should move. It processes these commands based on the robot's configuration (wheel radius, separation, etc.).

Velocity Calculation

It calculates the necessary velocities for each wheel to achieve the desired movement. These calculations are performed in each cycle of the controller's update function, ensuring that the latest commands are accurately reflected in the wheel velocities.

Hardware Commands

Once the velocities are determined, these are sent down to the hardware level—in this case, the `diffdrive_arduino` module. This module interacts directly with the physical robot hardware (e.g., motor drivers, encoders).

5.12.2 diffdrive_arduino Hardware Interface

Interface Setup

It sets up command and state interfaces for the wheels. This includes interfaces for both position and velocity, which allows the higher-level controller (`diff_drive_controller`) to command wheel velocities and read back positions or velocities as feedback.

Command Execution

The write function in `diffdrive_arduino` takes the velocity commands processed by `diff_drive_controller` and applies them to the motors through the Arduino connection. This function translates the desired wheel velocities into motor commands, considering the specific characteristics of the hardware like encoder counts per revolution and loop rate.

Feedback Processing

The read function gathers feedback from the hardware (e.g., encoder readings) and updates the internal state of the wheels (position and velocity). This feedback is critical for closed-loop control operations where the `diff_drive_controller` might adjust commands based on the difference between desired and actual wheel movements.

5.13 Controlling motors

5.13.1 Introduction

Integrating motors, controllers, and an Arduino Nano with ROS (Robot Operating System) presents a complex challenge. However, this complexity can be effectively managed using a layered approach. This section will detail the theory behind motor control layers and the practical steps to wire a motor control circuit.

5.13.2 Basic Power and Motor Drivers

At the core of our system is the motor. For a brushed DC motor, applying the appropriate supply voltage initiates movement. However, controlling speed, direction, and enabling remote control necessitates additional layers.

A motor driver is essential as it amplifies a low voltage, low current signal from a controller (such as a Raspberry Pi or Arduino) to create a higher voltage, higher current feed for the motors. This is typically achieved using Pulse Width Modulation (PWM), where the duty cycle of the PWM signal determines the motor's speed and direction.

5.13.3 Motor Controllers: Open and Closed Loop

A motor controller generates the PWM input signal for the motor driver. The simplest type is an open-loop controller, where a predefined map translates input signals into output signals. However, this method lacks robustness due to varying factors such as load and battery charge.

A more reliable method is closed-loop control, where the motor's actual speed is measured and fed back into the controller. The controller adjusts the input to achieve the desired speed, often using Proportional-Integral-Derivative (PID) control to minimize the error between the desired and actual speeds.

However, since we couldn't manage to get proper encoders for our motors, we are using the open loop control now.

5.13.4 Utilizing an Arduino Nano as a Motor Controller

Communication between the ROS system and the motor controller (Arduino Nano) is typically conducted via a serial link. The motor controller receives speed requests and provides feedback on motor performance, ensuring precise control. (In our case, this will be one way communication, since the feedback on motor performance is not monitored)

5.13.5 ROS System

The robot's control software (ROS) is responsible for calculating the required motor speeds and communicating these to the motor controller. This is divided into two components: the robot controller, which handles high-level control logic, and the driver software, which translates these commands into motor-specific instructions.

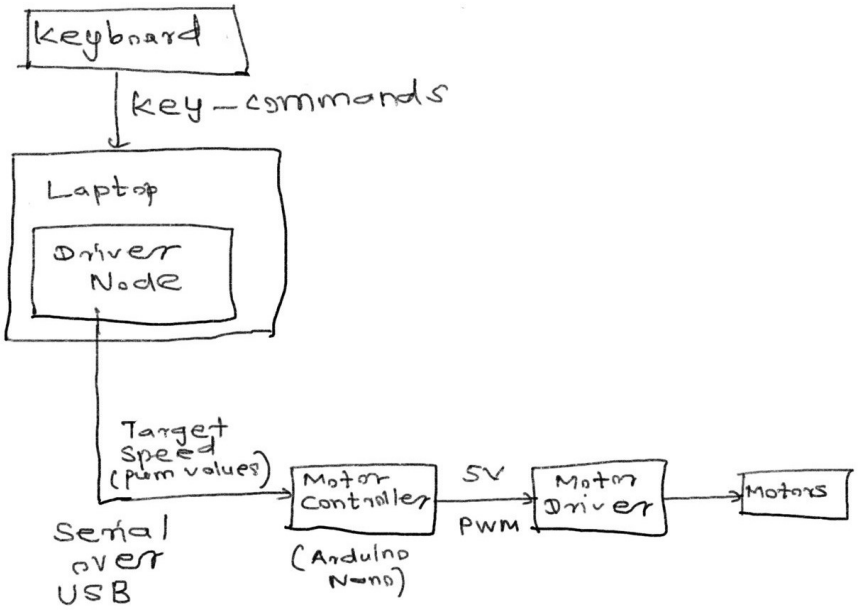


Figure 5.8: Motor Control

By layering the motor control system, integrating a motor driver, controller, and feedback loop, and utilizing ROS for high-level control, precise and reliable motor control is achieved. This

foundation allows for further enhancements, such as adding sensors and advanced control algorithms, to build a fully functional autonomous robot.

Integration of Arduino and Motor Driver

In our project, we utilize the ROS2 framework to manage and control a real robot by converting the velocity commands from an Rviz simulation into PWM (Pulse Width Modulation) signal commands (value between 0 and 255). These commands are then transmitted to an Arduino Nano via serial communication over a USB port. The Arduino Nano is programmed to interpret these PWM commands to control the motors' speed and direction, thus mirroring the movements of the simulated robot in Gazebo.

AVR microcontroller Code Explanation

The provided AVR microcontroller code is crucial for bridging the gap between digital commands from ROS2 and the physical movement of the robot's motors. Here is a breakdown of the key components of the code:

Below is the AVR microcontroller code for motor control:

```
#define F_CPU 16000000UL  // Define CPU clock speed
#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdlib.h>
#include <util/delay.h>

#define MOTOR1_PWM PD6 // OC0A (Output Compare pin for Timer0, Channel A)
#define MOTOR1_DIR PD2
#define MOTOR2_PWM PD5 // OC0B (Output Compare pin for Timer0, Channel B)
#define MOTOR2_DIR PD3

int pwm1 = 0, pwm2 = 0;

void uart_init(unsigned int ubrr) {
    // Set baud rate
    UBRR0H = (unsigned char)(ubrr>>8);
    UBRR0L = (unsigned char)ubrr;
    // Enable receiver and transmitter
    UCSR0B = (1<<RXEN0) | (1<<TXEN0);
    // Set frame format: 8 data bits, 1 stop bit
    UCSR0C = (1<<UCSZ01) | (1<<UCSZ00);
}

unsigned char uart_receive(void) {
    // Wait for data to be received
    while (!(UCSR0A & (1<<RXC0)));
    // Get and return received data from buffer
    return UDR0;
}

void uart_print(char* str) {
```

```
    while (*str) {
        while (!(UCSR0A & (1<<UDRE0)));
        UDR0 = *str++;
    }
}

void uart_print_int(int value) {
    char buffer[10];
    itoa(value, buffer, 10);
    uart_print(buffer);
}

void parse_and_set_speeds(char* input) {
    // Parse the input string for two integers
    int speed1, speed2;
    if (sscanf(input, "%d %d", &speed1, &speed2) == 2) {
        // Print the received PWM values
        uart_print("Received PWM values: ");
        uart_print_int(speed1);
        uart_print(" ");
        uart_print_int(speed2);
        uart_print("\r\n");

        // Set motor speeds
        set_motor_speed(1, speed1);
        set_motor_speed(2, speed2);
    } else {
        uart_print("Error: Invalid input format.\r\n");
    }
}

void pwm_init() {
    // Set PWM pins as output
    DDRD |= (1<<MOTOR1_PWM) | (1<<MOTOR2_PWM);
    DDRD |= (1<<MOTOR1_DIR) | (1<<MOTOR2_DIR);

    // Set Fast PWM mode, non-inverted for Timer0
    TCCR0A = (1<<WGM00) | (1<<WGM01) | (1<<COM0A1) | (1<<COM0B1);
    // Set prescaler to 64 and start PWM for Timer0
    TCCR0B = (1<<CS01) | (1<<CS00);

    // Set Fast PWM mode, non-inverted for Timer1
    TCCR1A = (1<<WGM10) | (1<<COM1A1) | (1<<COM1B1);
    TCCR1B = (1<<WGM12) | (1<<CS11) | (1<<CS10); // Prescaler 64
}

void set_motor_speed(int motor, int speed) {
    if (motor == 1) {
        if (speed >= 0) {
```

```
            PORTD &= ~(1<<MOTOR1_DIR); // Set direction to forward
            OCR0A = speed; // Set PWM duty cycle
        } else {
            PORTD |= (1<<MOTOR1_DIR); // Set direction to backward
            OCR0A = -speed; // Set PWM duty cycle
        }
    } else if (motor == 2) {
        if (speed >= 0) {
            PORTD &= ~(1<<MOTOR2_DIR); // Set direction to forward
            OCR0B = speed; // Set PWM duty cycle
        } else {
            PORTD |= (1<<MOTOR2_DIR); // Set direction to backward
            OCR0B = -speed; // Set PWM duty cycle
        }
    }
}

int main(void){
    char input[20];
    int inputIndex = 0;
    char receivedChar;

    uart_init(103); // Initialize UART with baud rate 9600
    pwm_init(); // Initialize PWM

    sei(); // Enable global interrupts

    while(1){
        // Receive characters and build the input string
        receivedChar = uart_receive();
        if (receivedChar == '\n' || receivedChar == '\r') {
            input[inputIndex] = '\0'; // Null-terminate the string
            parse_and_set_speeds(input); // Parse and set speeds
            inputIndex = 0; // Reset input index for next input
        } else {
            input[inputIndex++] = receivedChar; // Add char to input string
        }
    }
}
```

Definitions, Includes and Global Variables:
The code defines pin numbers connected to the motor driver IC MC33886PVWR2, for two motors.
Each motor has two pins designated for PWM control:

- F_CPU defines the CPU clock speed.
- Includes necessary AVR libraries for input/output, interrupts, standard libraries, and delay functions.

- Defines the pins used for PWM and direction control for two motors.
- pwm1 and pwm2 are variables to store PWM values for the motors.

UART Initialization:

- Initializes UART for serial communication with a given baud rate.

UART Receive and Transmit Functions:

- uart_receive waits for data to be received and returns it.
- uart_print sends a string over UART.

Parsing Input and Setting Motor Speeds:

- Parses the input string to extract two integers representing motor speeds.
- Prints the received PWM values and sets the motor speeds accordingly(for debugging purpose).

PWM Initialization:

- Configures the pins for PWM output.
- Sets up Timer0 and Timer1 for Fast PWM mode with a prescaler of 64.

Setting Motor Speed:

- Sets the speed and direction for the specified motor.
- Uses PWM to control speed and sets the direction pin accordingly.

Main Function:

- Initializes UART and PWM.
- Enables global interrupts.
- Continuously receives characters via UART to build an input string.
- Parses and sets motor speeds when a complete command is received.

Programming interface setup

Step 1: Download and Install AVR Studio

- Download and install AVR Studio using the given link https://www.microchip.com/en-us/tools-resources/archives/avr-sam-mcus

Step 2: Create a New Project

- Go to File > New > Project
- Select GCC C Executable Project, enter a name and location for your project, then click OK.

Step 3: Select the Device

- In the Device Selection window, select ATmega328P and click OK.

Step 4: Write and Build the program

- Write the given code in the window and click on Build > Build selection

Step 5: Upload the Code to Arduino nano

- Download and install AVRdude using the given link https://github.com/ZakKemble/AVRDUDESS/releases/tag/v2.17
- Select the required Baud rate, port number and browse the complied hex file of the program in the flash box and click program

Motor Driver IC

The motor control is executed using the MC33886PVWR2 motor driver IC, which responds to the PWM signals by adjusting the voltage and, consequently, the speed of the motors. This setup allows precise control over the robot's movement based on the simulation data.

5.14 SLAM using slam_toolbox

5.14.1 Introduction

This section provides a comprehensive guide on using the slam_toolbox package to perform Simultaneous Localization and Mapping (SLAM) with a 2D LiDAR sensor on a robot. We will go through the basics of SLAM, how it integrates into ROS, and the practical steps to implement it using slam_toolbox. By the end, we will be able to generate a map of the environment and localize the robot within that map.

5.14.2 SLAM Overview

SLAM stands for Simultaneous Localization and Mapping. It involves building a map of the environment while simultaneously keeping track of the robot's location within that map. This is crucial for autonomous navigation, especially in environments where GPS is unavailable or unreliable.

Localization and Mapping

- **Mapping:** Creating a map of the environment by noting landmarks and features as the robot moves.
- **Localization:** Determining the robot's position within a pre-existing map by recognizing landmarks and features.

SLAM combines these two processes, allowing the robot to build a map from scratch while keeping track of its position within that map.

5.14.3 Using the Map Frame to Account for Wheel Slippage and Other Issues

In robotics, accurate localization and mapping are crucial for autonomous navigation. However, wheel slippage, sensor noise, and other factors can cause discrepancies in the robot's odometry, leading to inaccurate position estimates. To address these issues, we introduce a coordinate frame called the map frame, which helps in correcting these errors and maintaining accurate localization. This section explains how the map frame works and how it helps account for wheel slippage and other inaccuracies.

5.14.4 Coordinate Frames in ROS

Before diving into the map frame, it's essential to understand the basic coordinate frames used in ROS (Robot Operating System):

- **Base Link Frame:** This frame is attached to the robot and moves with it.
- **Odom Frame:** Represents the origin of the robot's odometry. It is calculated based on the wheel encoders and provides a smooth but potentially inaccurate estimate of the robot's position.
- **Map Frame:** A global reference frame used for accurate positioning and mapping. This frame is updated using SLAM (Simultaneous Localization and Mapping) or other localization systems.

5.14.5 Odometry and Its Limitations

Odometry involves estimating the robot's position by integrating the velocities over time. This method relies on wheel encoders, which measure the rotation of the wheels. However, odometry has inherent limitation

- **Wheel Slippage:** Wheels can slip on the surface, leading to errors in distance measurement.
- **Cumulative Drift:** Small errors accumulate over time, causing the robot's estimated position to drift away from the true position.

5.14.6 Introducing the Map Frame

To correct for the drift and slippage inherent in odometry, we introduce the map frame. The map frame provides a more accurate and global reference for the robot's position, updated through SLAM or GPS. The transform from the map frame to the odom frame accounts for the discrepancies and ensures accurate localization.

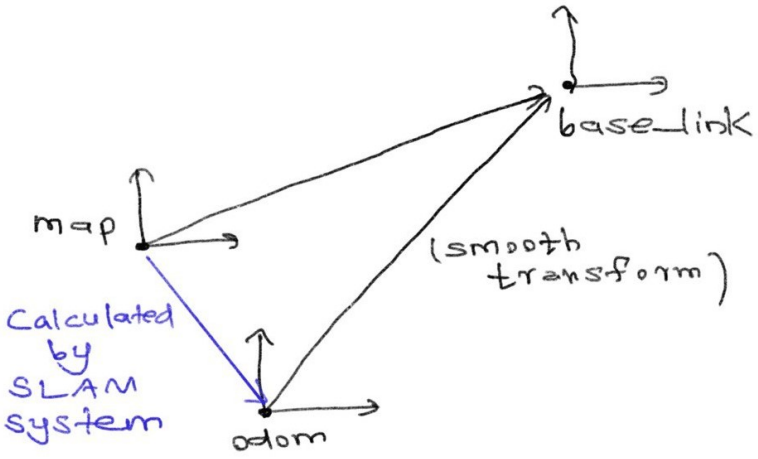


Figure 5.9: map frame

Transform Calculations

The key transforms involved are:

- **Odom to Base Link:** Provides the robot's estimated position based on odometry.
- **Map to Odom:** Corrects the odometry estimate using the SLAM position estimate.

Transform Logic

- **Initial State:** At startup, both odom and base_link frames originate at (0, 0, 0).
- **Movement:** As the robot moves, the odom frame tracks the estimated position based on wheel encoders.
- **SLAM Correction:** SLAM provides a more accurate global position of the robot. This position is used to calculate the map to odom transform..
- **Final Position:** The robot's accurate position in the map frame is obtained by combining the map to odom and odom to base_link transforms.

Example

Consider a scenario where the robot drives forward for 2 meters. Due to wheel slippage, the odometry might only report 1.9 meters. The transforms would be:

- **Odom to Base Link:** (1.9, 0, 0)
- **SLAM Position Estimate:** (2.0, 0, 0)
- **Map to Odom:** Adjusts for odometry drift, correcting base_link to (2.0, 0, 0) in the map frame.

5.14.7 Topics and Data

- **Odom Topic:** Contains the odometry data, including position and velocity estimates.
- **Map Topic:** Contains the occupancy grid map, representing the environment and obstacles.

Additional Frames

Base Footprint Frame: A 2D projection of the base link frame, used for 2D SLAM applications. It ensures that the SLAM process ignores vertical movements and focuses on the horizontal plane.

The map frame is an essential component in robotic localization and mapping, providing a global reference that corrects the errors inherent in odometry. By integrating the map frame into the SLAM process, we can ensure accurate and reliable navigation, accounting for wheel slippage and other inaccuracies. Understanding and implementing these coordinate frame conventions in ROS enhances the robustness and precision of autonomous robotic systems.

5.14.8 slam_toolbox

The SLAM Toolbox, developed by Steve Macenski, is a robust ROS2 package designed for implementing SLAM with 2D LIDAR data. This document explores the key algorithms and techniques utilized by the SLAM Toolbox to provide efficient and accurate SLAM capabilities.

5.14.9 Algorithms and Techniques

The SLAM Toolbox employs several advanced algorithms and techniques that enhance its SLAM performance, including grid-based SLAM, scan matching, loop closure, and pose graph optimization.

Grid-Based SLAM

Grid-based SLAM represents the environment as a discrete grid, where each cell indicates whether it is occupied, free, or unknown. This method is particularly effective for 2D mapping.

- **Occupancy Grid:** A grid where each cell represents the occupancy probability.
- **Cell Update:** LIDAR scans are used to update the occupancy probabilities of the cells based on the sensor readings.

Scan Matching

Scan matching involves aligning new LIDAR scans with existing scans or maps to accurately determine the robot's movement between scans. Techniques like Iterative Closest Point (ICP) are used to efficiently match scans.

- **Iterative Closest Point (ICP):** A technique to minimize the difference between two clouds of points by iteratively aligning them.
- **Probability Maximization:** Techniques to maximize the likelihood of scan alignment.

Loop Closure

Loop closure is vital for correcting drift in the SLAM process. It adjusts the map and trajectory when the robot revisits previously mapped areas.

- **Drift Correction:** Adjusting the map and trajectory to correct accumulated errors.
- **Recognition:** Identifying previously visited locations to facilitate loop closure.

Pose Graph Optimization

Pose graph optimization minimizes the errors in pose estimates by optimizing a graph where nodes represent robot poses and edges represent transformations between these poses.

- **Pose Graph:** A graphical representation of the robot's poses and transformations.
- **Optimization:** Techniques like sparse pose adjustment (SPA) refine the pose graph to reduce errors and improve map accuracy.

SLAM Toolbox Modes

The SLAM Toolbox supports various operational modes to accommodate different scenarios:

- **Online Asynchronous Mapping:** Real-time mapping while the robot is in motion, processing scan data asynchronously.
- **Localization:** Utilizing a pre-built map for localization purposes without updating the map, ideal for operations in known environments.

- **Lifelong Mapping:** Continuously updating the map to accommodate changes in dynamic environments over extended periods.

The SLAM Toolbox provides a powerful suite of tools for implementing SLAM in robotic systems. Its versatility in handling different mapping and localization scenarios makes it an essential tool for developing autonomous navigation capabilities in robots.

5.14.10 Setting Up slam_toolbox

Installing slam_toolbox

First, install slam_toolbox:

```
sudo apt install ros-humble-slam-toolbox
```

Configuring the URDF

Add the base_footprint link to the robot's URDF file:

```
<joint name="base_footprint_joint" type="fixed">
  <parent link="base_link"/>
  <child link="base_footprint"/>
</joint>
<link name="base_footprint"/>
```

SLAM Parameters

Copy the sample parameter file for slam_toolbox and "online_async_launch.py" launch file and place them in the "my_bot" directory. And build the package again.

Edit the parameter file as needed. For our project, we mostly use the default settings, meanwhile adjusting some necessary frames.

5.14.11 Running SLAM in Simulation

Launch the Robot Simulation:

```
ros2 launch my_bot launch_robot_sim.launch.py
```

Run slam_toolbox:

```
ros2 launch my_bot online_async_launch.py
```

Visualize in RViz:

- Add the Map display and set the topic to /map.
- Set the fixed frame to map.

Drive the robot around to see the map being generated in real-time.

By following these steps, we can implement SLAM on our robot using slam_toolbox, generate maps, and use those maps for localization. This setup provides a robust framework for autonomous navigation, which will be further enhanced in the next section with the integration of the nav2 stack.

5.14.12 Saving the Map

After generating a map with the SLAM Toolbox in RViz, the next important step is to save this map for future use, whether for navigation tasks, further processing, or simply as a record. This process involves using the SLAM Toolbox Plugin integrated within RViz, which provides a user-friendly interface for managing the SLAM session and saving the resultant map.

Using the SLAM Toolbox Plugin in RViz

To save the generated map from your SLAM session, follow these steps:

1. **Add the SLAM Toolbox Plugin in RViz:** Open RViz and go to the "Displays" panel. Click on the "Add" button, navigate to "By topic", and then select "SLAM Toolbox". This action adds the plugin's display to your RViz workspace.
2. **Save the Map:** Once the plugin is added, locate the "Save Map" button within the SLAM Toolbox Plugin panel. Clicking this button will prompt you to choose a location and file name for saving your map.
3. **Specify the File Location:** Enter the desired path and filename in the dialog box that appears. The map will be saved in two parts:
   - A `.pgm` file: This file contains the occupancy grid data, which visually represents the map.
   - A `.yaml` file: This configuration file includes metadata about the map, such as the map's dimensions, resolution, and the origin coordinate.

5.14.13 Using the Map

Once the map is saved, it can be used for various applications such as autonomous navigation, environment monitoring, or as a foundation for further environmental modeling. To use the map in a new RViz session or within a navigation stack, we will need to:

1. Load the map using the Map Server provided by the ROS navigation stack.
2. Configure any navigation nodes to use this map for planning and executing movements.
3. Optionally, adjust parameters such as the robot's starting position on the map, which can be set using the "2D Pose Estimate" tool in RViz for initial localization.

The slam_toolbox package source codes are availble in this repo. https://github.com/SteveMacenski/slam_toolbox

Used "online_async_launch.py" file is given below.

```
import os

from launch import LaunchDescription
```

```
from launch.actions import DeclareLaunchArgument, LogInfo
from launch.conditions import UnlessCondition
from launch.substitutions import LaunchConfiguration, PythonExpression
from launch_ros.actions import Node
from ament_index_python.packages import get_package_share_directory
from nav2_common.launch import HasNodeParams

def generate_launch_description():
    # Launch configurations
    use_sim_time = LaunchConfiguration('use_sim_time')
    params_file = LaunchConfiguration('params_file')

    # Default parameters file path
    default_params_file = os.path.join(get_package_share_directory("my_bot"),
                                       'config',
                                           'mapper_params_online_async.yaml')

    # Declare the 'use_sim_time' launch argument with default value 'true'
    declare_use_sim_time_argument = DeclareLaunchArgument(
        'use_sim_time',
        default_value='true',
        description='Use simulation/Gazebo clock'
    )

    # Declare the 'params_file' launch argument with the default parameters
        file path
    declare_params_file_cmd = DeclareLaunchArgument(
        'params_file',
        default_value=default_params_file,
        description='Full path to the ROS2 parameters file to use for the
            slam_toolbox node'
    )

    # Check if the provided params file has 'slam_toolbox' parameters
    has_node_params = HasNodeParams(source_file=params_file,
        node_name='slam_toolbox')

    # Determine the actual parameters file to use
    actual_params_file = PythonExpression([
        '"', params_file, '" if ', has_node_params, ' else "',
            default_params_file, '"'
    ])

    # Log info if the provided params file does not contain 'slam_toolbox'
        parameters
    log_param_change = LogInfo(
        msg=[
            'provided params_file ', params_file,
            ' does not contain slam_toolbox parameters. Using default: ',
                default_params_file
```

```
        ],
        condition=UnlessCondition(has_node_params)
    )

    # Start the async_slam_toolbox_node with the determined parameters file
        and 'use_sim_time'
    start_async_slam_toolbox_node = Node(
        parameters=[
            actual_params_file,
            {'use_sim_time': use_sim_time}
        ],
        package='slam_toolbox',
        executable='async_slam_toolbox_node',
        name='slam_toolbox',
        output='screen'
    )

    # Create the launch description and add the actions
    ld = LaunchDescription()
    ld.add_action(declare_use_sim_time_argument)
    ld.add_action(declare_params_file_cmd)
    ld.add_action(log_param_change)
    ld.add_action(start_async_slam_toolbox_node)

    return ld
```

5.14.14 Generated Maps

5.15 Navigation using Nav2 Stack

5.15.1 Introduction

In this section, we will explore how to implement autonomous navigation in our robot using the Nav2 stack. This involves specifying a target location and having the robot autonomously navigate to that location while avoiding obstacles. We'll start with an overview of navigation concepts, set up Nav2 in a Gazebo simulation, and then implement it on a real robot. Finally, we'll cover some tips and tricks for optimizing and customizing the navigation process.

5.15.2 What is Navigation?

Navigation is the process of planning and executing a safe trajectory from an initial pose to a target pose. Essential components required include:

- Accurate Position Estimate: Provided by SLAM or another localization system.
- Obstacle Awareness: Information about obstacles can come from a pre-recorded map, live sensor data, or a combination of both.

These pieces of information are stored in a cost map, which assigns a cost to different areas of the environment based on their navigability.

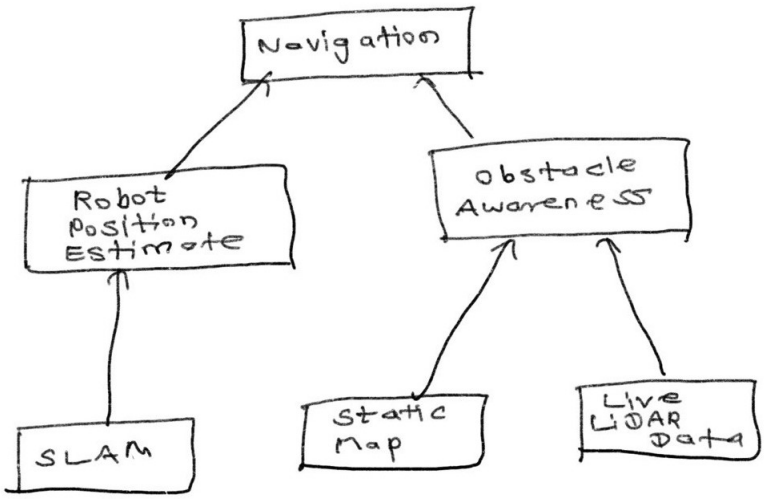


Figure 5.10: Navigation concept

5.15.3 Nav2 Stack

The Navigation2 (Nav2) stack is an advanced robotics software framework designed to facilitate autonomous navigation. This report provides a detailed explanation of the Nav2 stack algorithms, highlighting the key components and processes involved in navigating a robot from a starting point to a goal position. The report covers initialization, localization, path planning, costmap management, local planning and control, obstacle avoidance, and behavior management.

5.15.4 Initialization

Upon initialization, several essential nodes and components are set up:

- Map Server: Provides the global map of the environment.
- AMCL (Adaptive Monte Carlo Localization): Localizes the robot on the map.
- Costmap2D: Maintains both global and local costmaps.
- Planner Server: Generates a global path from the start to the goal.
- Controller Server: Generates local velocity commands to follow the global path.
- Behavior Tree: Manages the overall navigation behavior.

5.15.5 Localization

Localization is achieved through the Adaptive Monte Carlo Localization (AMCL) node:

- AMCL uses a particle filter to estimate the robot's pose based on sensor data (e.g., LiDAR or camera) and the provided map.
- The AMCL node continuously updates the robot's estimated position and orientation on the map.

5.15.6 Global Path Planning

The process of global path planning involves:

- Setting a goal: The user specifies a goal position on the map.
- Path computation: The Planner Server uses algorithms like A* or Dijkstra's algorithm to create a global path from the robot's current location to the goal location.
- Global costmap utilization: The global planner leverages the global costmap, which contains static obstacles, to generate the path.

5.15.7 Costmaps

The Nav2 stack employs two types of costmaps:

- Global Costmap: Represents the entire map with static obstacles and is used for global path planning.
- Local Costmap: Represents the area around the robot and is updated with dynamic obstacles for local planning and obstacle avoidance.

5.15.8 Local Path Planning and Control

The Controller Server handles local path planning and control:

- It receives the global path and computes local velocity commands to follow this path while avoiding obstacles.
- The default local planner is the DWB (Dynamic Window Approach) Local Planner, which operates by sampling possible velocity commands, predicting the robot's trajectory for each command, scoring each trajectory based on factors such as distance to obstacles, alignment with the global path, and velocity, and selecting the best trajectory to generate corresponding velocity commands (linear.x and angular.z).

5.15.9 Obstacle Avoidance

Obstacle avoidance is managed through:

- Continuous updates to the local costmap with sensor data.
- Adjustment of the local path and velocity commands to avoid collisions.
- Triggering replanning or recovery behaviors if an obstacle blocks the path.

5.15.10 Behavior Tree Execution

The behavior tree coordinates the overall navigation task:

- It handles different states such as planning, controlling, and recovery behaviors.
- Defines transitions between tasks, like replanning when a new obstacle is detected or performing recovery actions if the robot gets stuck.

5.15.11 Detailed Algorithm Steps

Setup and Initialization

Initialize the nodes and components. Load the map into the global costmap. Start the AMCL for localization.

Localization and State Estimation

The AMCL node updates the robot's pose on the map using sensor data.

Setting a Goal

The user sets a goal position on the map. The goal is sent to the Planner Server.

Global Path Planning

The Planner Server computes a global path using the global costmap. The planned path is sent to the Controller Server.

Local Path Planning and Control

The Controller Server receives the global path. It uses the DWB Local Planner to compute velocity commands.

- Generate possible velocity samples.
- Predict trajectories for each sample.
- Score each trajectory based on various criteria (e.g., distance to obstacles, alignment with the path).
- Select the best trajectory and generate the corresponding linear.x and angular.z velocities.

The velocities are sent to the robot's motor controllers.

Obstacle Avoidance and Re-planning

Continuously update the local costmap with sensor data. Adjust the local path and velocity commands to avoid obstacles. If an obstacle blocks the path, trigger replanning or recovery behaviors.

Behavior Management

The behavior tree manages the high-level tasks and states. It handles different scenarios like path following, replanning, and recovery. Ensures smooth transitions between different navigation states.

Reaching the Goal

The process continues until the robot reaches the goal position. Once the goal is reached, the behavior tree transitions to the idle state.

The Nav2 stack employs a combination of global and local planning, state estimation, and behavior management to achieve autonomous navigation. The global planner provides a path to the goal, while the local planner generates real-time velocity commands to follow the path and

avoid obstacles. The behavior tree orchestrates the overall navigation process, handling various states and scenarios to ensure successful navigation.

5.15.12 Installing Nav2 and Preparations

First, install Nav2 and Twist Mux:

```
sudo apt install ros-humble-navigation2 ros-humble-nav2-bringup\
ros-humble-twist-mux
```

Setting Up Twist Mux

Twist Mux manages multiple velocity command sources and prioritizes them. Create a configuration file `twist_mux.yaml`:

```
twist_mux:
  ros__parameters:
    topics:
      navigation:
        topic   : cmd_vel
        timeout : 0.5
        priority: 10
      tracker:
        topic   : cmd_vel_tracker
        timeout : 0.5
        priority: 20
```

And include the following code in the "launch_robot_sim.launch.py" file.

```
twist_mux_params = os.path.join(get_package_share_directory(package_name),
'config','twist_mux.yaml')
twist_mux = Node(
        package="twist_mux",
        executable="twist_mux",
        parameters=[twist_mux_params],
        remappings=[('/cmd_vel_out','/diff_cont2/cmd_vel_unstamped')]
)
```

5.15.13 Running Nav2 in Gazebo

And copy Nav2 configuration files and "navigation_launch.py" launch file into the my_bot package for easier customization

Launch the robot simulation:

```
ros2 launch my_bot launch_robot_sim.launch.py
```

Run SLAM Toolbox:

```
ros2 launch my_bot online_async_launch.py
```

Launch Nav2:

```
ros2 launch my_bot navigation_launch.py use_sim_time:=true
```

Visualize in RViz by adding a Global Costmap display and setting the fixed frame to map.

Set the initial pose in RViz using the 2D Pose Estimate tool.

Given below is the "navigation_launch.py" launch file.

```
# Copyright (c) 2018 Intel Corporation
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

#Modified by Sundarbavan T.

import os
from ament_index_python.packages import get_package_share_directory

from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument, GroupAction,
    SetEnvironmentVariable
from launch.conditions import IfCondition
from launch.substitutions import LaunchConfiguration, PythonExpression
from launch_ros.actions import LoadComposableNodes, Node
from launch_ros.descriptions import ComposableNode, ParameterFile
from nav2_common.launch import RewrittenYaml

def generate_launch_description():
    # Get the directory of the 'my_bot' package
    bringup_dir = get_package_share_directory('my_bot')

    # Launch configurations
    namespace = LaunchConfiguration('namespace')
    use_sim_time = LaunchConfiguration('use_sim_time')
    autostart = LaunchConfiguration('autostart')
```

```
    params_file = LaunchConfiguration('params_file')
    use_composition = LaunchConfiguration('use_composition')
    container_name = LaunchConfiguration('container_name')
    container_name_full = (namespace, '/', container_name)
    use_respawn = LaunchConfiguration('use_respawn')
    log_level = LaunchConfiguration('log_level')

    # Nodes that require lifecycle management
    lifecycle_nodes = [
        'controller_server',
        'smoother_server',
        'planner_server',
        'behavior_server',
        'bt_navigator',
        'waypoint_follower',
        'velocity_smoother'
    ]

    # Remappings for tf and tf_static topics
    remappings = [
        ('/tf', 'tf'),
        ('/tf_static', 'tf_static')
    ]

    # Create our own temporary YAML files that include substitutions
    param_substitutions = {
        'use_sim_time': use_sim_time,
        'autostart': autostart
    }

    # Rewritten parameters file with substitutions
    configured_params = ParameterFile(
        RewrittenYaml(
            source_file=params_file,
            root_key=namespace,
            param_rewrites=param_substitutions,
            convert_types=True
        ),
        allow_substs=True
    )

    # Set environment variable for logging
    stdout_linebuf_envvar = SetEnvironmentVariable(
        'RCUTILS_LOGGING_BUFFERED_STREAM', '1'
    )

    # Declare launch arguments
    declare_namespace_cmd = DeclareLaunchArgument(
        'namespace',
```

```
        default_value='',
        description='Top-level namespace'
    )

    declare_use_sim_time_cmd = DeclareLaunchArgument(
        'use_sim_time',
        default_value='false',
        description='Use simulation (Gazebo) clock if true'
    )

    declare_params_file_cmd = DeclareLaunchArgument(
        'params_file',
        default_value=os.path.join(bringup_dir, 'config', 'nav2_params.yaml'),
        description='Full path to the ROS2 parameters file to use for all
            launched nodes'
    )

    declare_autostart_cmd = DeclareLaunchArgument(
        'autostart',
        default_value='true',
        description='Automatically startup the nav2 stack'
    )

    declare_use_composition_cmd = DeclareLaunchArgument(
        'use_composition',
        default_value='False',
        description='Use composed bringup if True'
    )

    declare_container_name_cmd = DeclareLaunchArgument(
        'container_name',
        default_value='nav2_container',
        description='The name of the container that nodes will load in if use
            composition'
    )

    declare_use_respawn_cmd = DeclareLaunchArgument(
        'use_respawn',
        default_value='False',
        description='Whether to respawn if a node crashes. Applied when
            composition is disabled.'
    )

    declare_log_level_cmd = DeclareLaunchArgument(
        'log_level',
        default_value='info',
        description='Log level'
    )
```

```
    # Load the nodes without composition
    load_nodes = GroupAction(
        condition=IfCondition(PythonExpression(['not ', use_composition])),
        actions=[
            Node(
                package='nav2_controller',
                executable='controller_server',
                output='screen',
                respawn=use_respawn,
                respawn_delay=2.0,
                parameters=[configured_params],
                arguments=['--ros-args', '--log-level', log_level],
                remappings=remappings + [('cmd_vel', 'cmd_vel_nav')]
            ),
            Node(
                package='nav2_smoother',
                executable='smoother_server',
                name='smoother_server',
                output='screen',
                respawn=use_respawn,
                respawn_delay=2.0,
                parameters=[configured_params],
                arguments=['--ros-args', '--log-level', log_level],
                remappings=remappings
            ),
            Node(
                package='nav2_planner',
                executable='planner_server',
                name='planner_server',
                output='screen',
                respawn=use_respawn,
                respawn_delay=2.0,
                parameters=[configured_params],
                arguments=['--ros-args', '--log-level', log_level],
                remappings=remappings
            ),
            Node(
                package='nav2_behaviors',
                executable='behavior_server',
                name='behavior_server',
                output='screen',
                respawn=use_respawn,
                respawn_delay=2.0,
                parameters=[configured_params],
                arguments=['--ros-args', '--log-level', log_level],
                remappings=remappings
            ),
            Node(
                package='nav2_bt_navigator',
```

```
            executable='bt_navigator',
            name='bt_navigator',
            output='screen',
            respawn=use_respawn,
            respawn_delay=2.0,
            parameters=[configured_params],
            arguments=['--ros-args', '--log-level', log_level],
            remappings=remappings
        ),
        Node(
            package='nav2_waypoint_follower',
            executable='waypoint_follower',
            name='waypoint_follower',
            output='screen',
            respawn=use_respawn,
            respawn_delay=2.0,
            parameters=[configured_params],
            arguments=['--ros-args', '--log-level', log_level],
            remappings=remappings
        ),
        Node(
            package='nav2_velocity_smoother',
            executable='velocity_smoother',
            name='velocity_smoother',
            output='screen',
            respawn=use_respawn,
            respawn_delay=2.0,
            parameters=[configured_params],
            arguments=['--ros-args', '--log-level', log_level],
            remappings=remappings + [
                ('cmd_vel', 'cmd_vel_nav'),
                ('cmd_vel_smoothed', 'cmd_vel')
            ]
        ),
        Node(
            package='nav2_lifecycle_manager',
            executable='lifecycle_manager',
            name='lifecycle_manager_navigation',
            output='screen',
            arguments=['--ros-args', '--log-level', log_level],
            parameters=[
                {'use_sim_time': use_sim_time},
                {'autostart': autostart},
                {'node_names': lifecycle_nodes}
            ]
        ),
    ]
)
```

```
    # Load the nodes with composition
    load_composable_nodes = LoadComposableNodes(
        condition=IfCondition(use_composition),
        target_container=container_name_full,
        composable_node_descriptions=[
            ComposableNode(
                package='nav2_controller',
                plugin='nav2_controller::ControllerServer',
                name='controller_server',
                parameters=[configured_params],
                remappings=remappings + [('cmd_vel', 'cmd_vel_nav')]
            ),
            ComposableNode(
                package='nav2_smoother',
                plugin='nav2_smoother::SmootherServer',
                name='smoother_server',
                parameters=[configured_params],
                remappings=remappings
            ),
            ComposableNode(
                package='nav2_planner',
                plugin='nav2_planner::PlannerServer',
                name='planner_server',
                parameters=[configured_params],
                remappings=remappings
            ),
            ComposableNode(
                package='nav2_behaviors',
                plugin='behavior_server::BehaviorServer',
                name='behavior_server',
                parameters=[configured_params],
                remappings=remappings
            ),
            ComposableNode(
                package='nav2_bt_navigator',
                plugin='nav2_bt_navigator::BtNavigator',
                name='bt_navigator',
                parameters=[configured_params],
                remappings=remappings
            ),
            ComposableNode(
                package='nav2_waypoint_follower',
                plugin='nav2_waypoint_follower::WaypointFollower',
                name='waypoint_follower',
                parameters=[configured_params],
                remappings=remappings
            ),
            ComposableNode(
                package='nav2_velocity_smoother',
```

```
                plugin='nav2_velocity_smoother::VelocitySmoother',
                name='velocity_smoother',
                parameters=[configured_params],
                remappings=remappings + [
                    ('cmd_vel', 'cmd_vel_nav'),
                    ('cmd_vel_smoothed', 'cmd_vel')
                ]
            ),
            ComposableNode(
                package='nav2_lifecycle_manager',
                plugin='nav2_lifecycle_manager::LifecycleManager',
                name='lifecycle_manager_navigation',
                parameters=[{
                    'use_sim_time': use_sim_time,
                    'autostart': autostart,
                    'node_names': lifecycle_nodes
                }]
            ),
        ],
    )

    # Create the launch description and populate it with actions
    ld = LaunchDescription()

    # Set environment variables
    ld.add_action(stdout_linebuf_envvar)

    # Declare the launch arguments
    ld.add_action(declare_namespace_cmd)
    ld.add_action(declare_use_sim_time_cmd)
    ld.add_action(declare_params_file_cmd)
    ld.add_action(declare_autostart_cmd)
    ld.add_action(declare_use_composition_cmd)
    ld.add_action(declare_container_name_cmd)
    ld.add_action(declare_use_respawn_cmd)
    ld.add_action(declare_log_level_cmd)

    # Add the actions to launch all of the navigation nodes
    ld.add_action(load_nodes)
    ld.add_action(load_composable_nodes)

    return ld
```

By following these steps, we can implement autonomous navigation on our robot using the Nav2 stack. This setup allows our robot to navigate safely and efficiently in various environments, opening up further possibilities for advanced robotics applications.

5.15.14 Diagrams of the frames broadcast by tf2

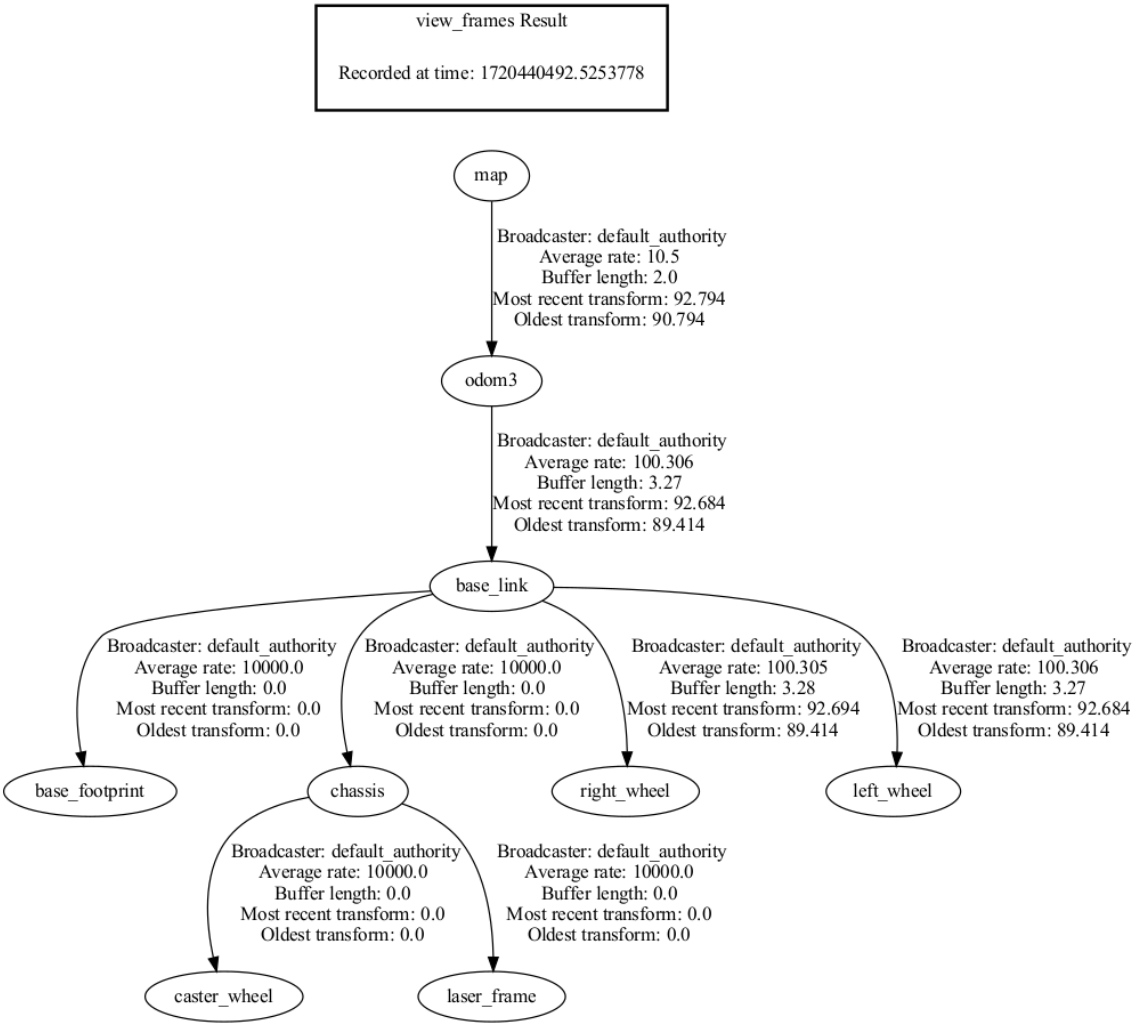


Figure 5.11: Frames

5.15.15 Active Nodes and Topics in the ROS system in our project

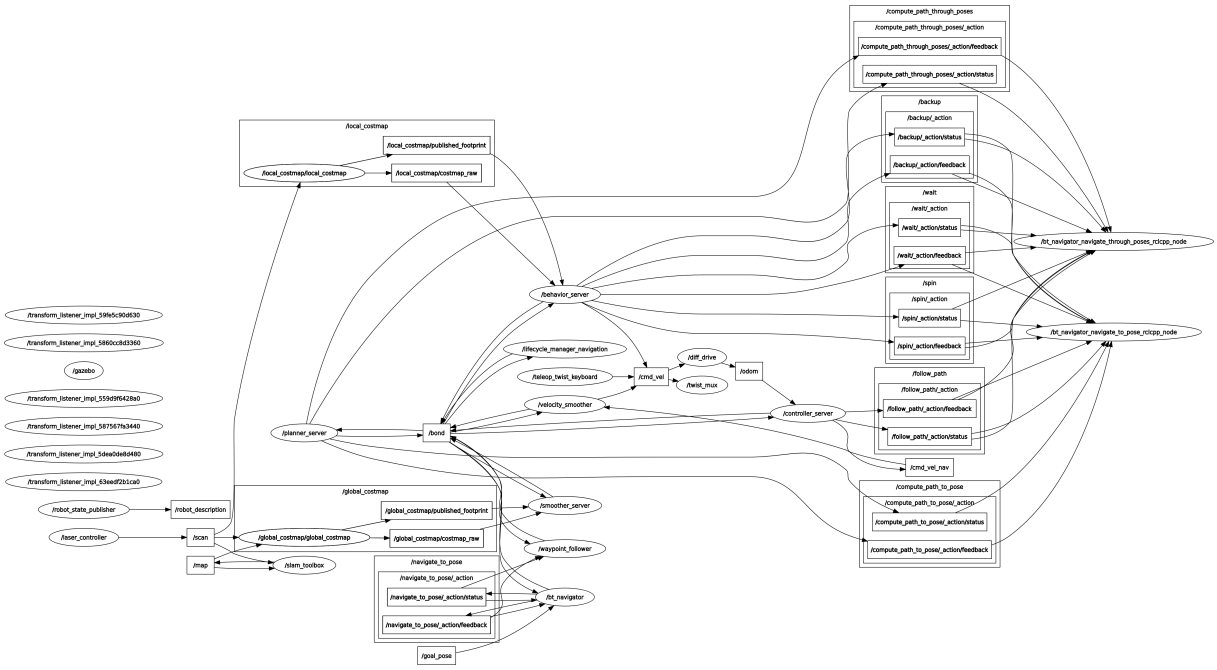


Figure 5.12: Nodes and Topics

5.16 Simulation Results

5.16.1 Room1

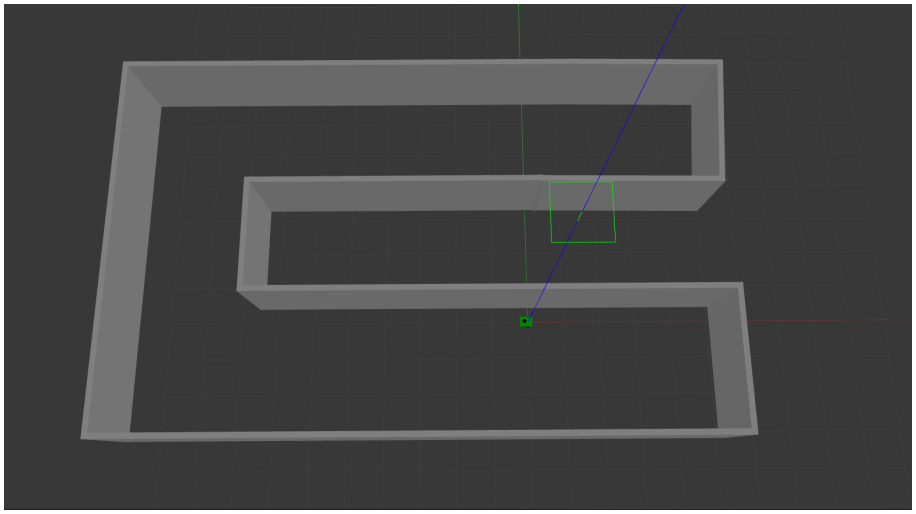


Figure 5.13: Room1 Gazebo

5.16.2 Room2

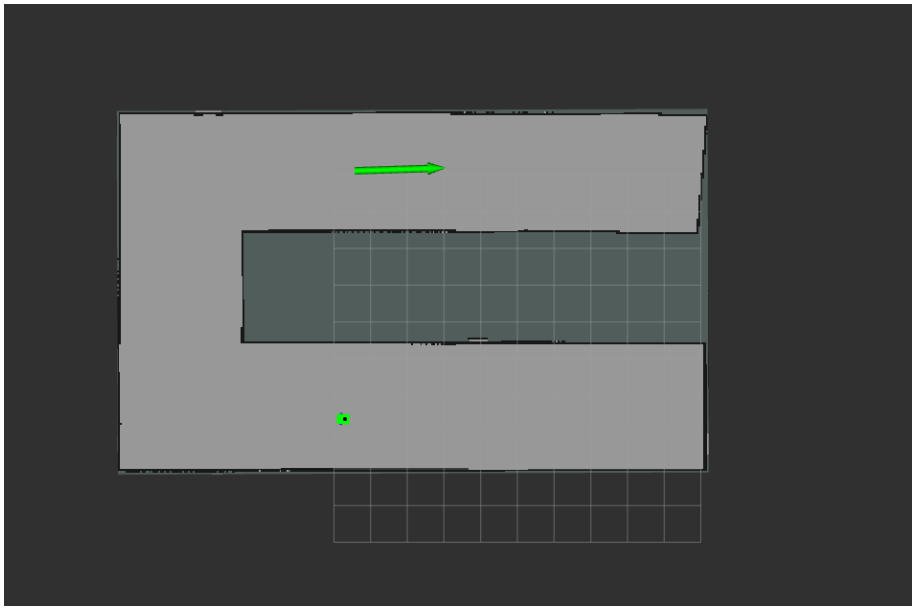


Figure 5.14: Room1 RViz Map and 2D Goal

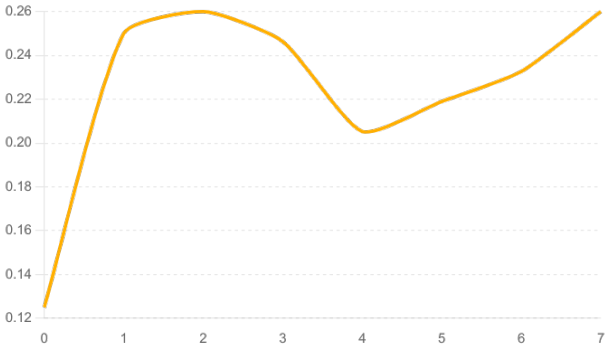


Figure 5.15: linear_x Velocity over Time

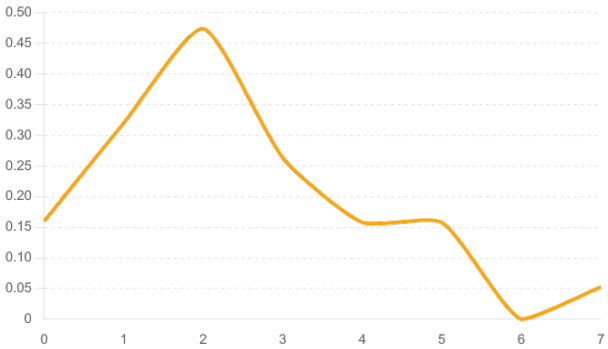


Figure 5.16: angular_z Velocity over Time

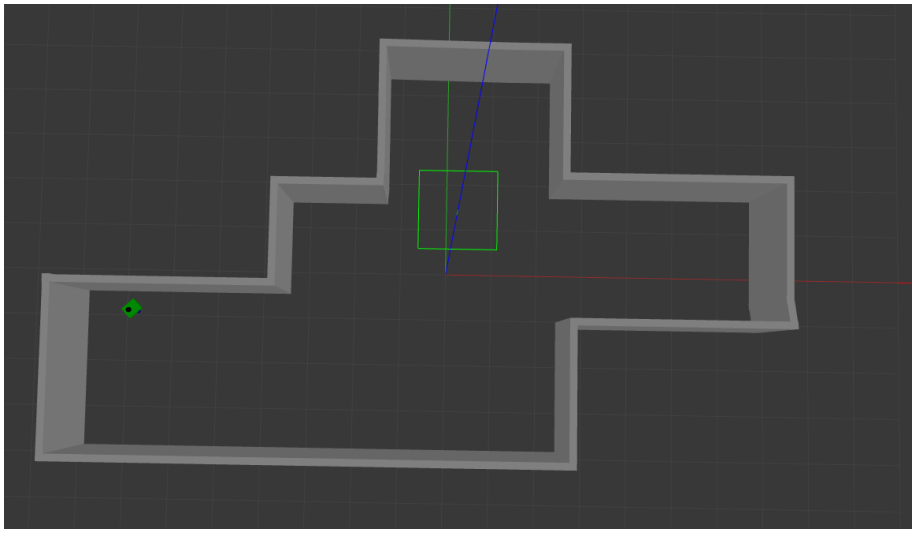


Figure 5.17: Room2 Gazebo

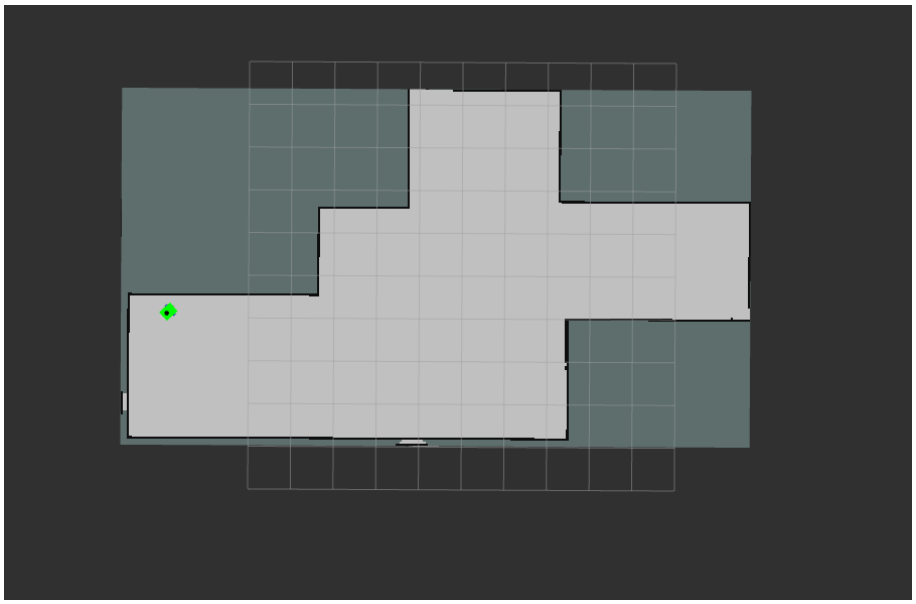


Figure 5.18: Room2 Rviz Map

Chapter 6

Software Implementation

6.1 Implementation

To implement the software for the Automatic Mobile Robot Controller (AMR Controller), we will start by integrating the project-specific packages located in the provided GitHub repository. Follow the steps below to properly clone and build the necessary packages within the workspace:

6.1.1 Copying packages

Navigate to the src Directory

First, access the src directory inside the amr_ws workspace:

```
cd ~/amr_ws/src
```

Clone the GitHub Repository

Clone the project repository to the src directory to add the necessary packages for the AMR Controller:

```
git clone https://github.com/Bavan2002/AMR.git
```

This step will download all project-related packages into our workspace.

Build the Workspace

Navigate back to the root of the workspace:

```
cd ..
```

Rebuild the entire workspace to incorporate the new packages:

```
colcon build --symlink-install
```

Remember that this command should always be executed from the root directory of the workspace (in this case, amr_ws). The –symlink-install flag ensures that minor changes to existing files don't require a complete rebuild, saving time and resources.

After completion

Figure 6.1: After colcon build

Now, with the essential packages cloned and built, the development environment is set up and ready for further customization and testing specific to the Automatic Mobile Robot Controller

After every build, source the overlay workspace by running the following command **whenever opening a new terminal.**

```
source install/setup.bash
```

6.1.2 Running Simulations

Make sure the if the underlay is not automatically sourced at startup, it is important to execute the following command **whenever opening a new terminal.**

```
# Replace ".zsh" with your shell if you're not using zsh.
# Possible values are: setup.bash, setup.sh, setup.zsh
source /opt/ros/humble/setup.bash
cd amr_ws
source install/setup.bash
```

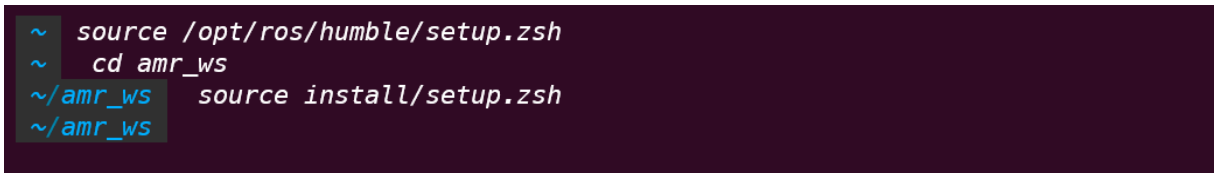


Figure 6.2: whenever opening a new terminal

From now on the commands are being executed from the amr_ws directory unless otherwise stated.

Gazebo Envoironment Simulation

To run the Gazebo simulation of the robot within your desired world, execute the following command in a **new terminal:**

```
ros2 launch my_bot launch_robot_sim.launch.py \
world:=./src/AMR/my_bot/worlds/obstacles.world
```

In this command, my_bot refers to the package name, and launch_robot_sim.launch.py is the launch file for simulating the robot in Gazebo. The world mentioned within this launch file will also be imported into Gazebo. After executing the above command, a simulated gazebo world and the robot will appear in Gazebo app.

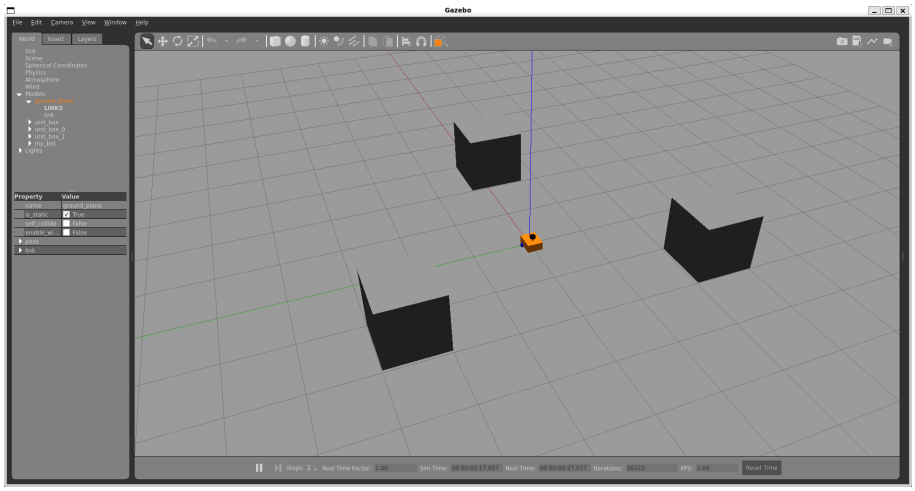


Figure 6.3: Gazebo simulation

Rviz for Odometry and Map Visualization

To visualize the robot's odometry and environment maps in Gazebo, we use Rviz, which displays these aspects in a graphical format. Launch Rviz with our specified configurations by executing the following commands in a **new terminal:**

```
ros2 run rviz2 rviz2 -d src/AMR/my_bot/config/main.rviz \
--ros-args -p use_sim_time:=true
```

Here main.rviz is the specific Rviz configuration file.

Teleop Keyboard Control

To move the robot during the Gazebo simulation, utilize the teleop keyboard, which is implemented via a modified version of the teleop_twist_keyboard tool. Launch it with the following command in a **new terminal:**

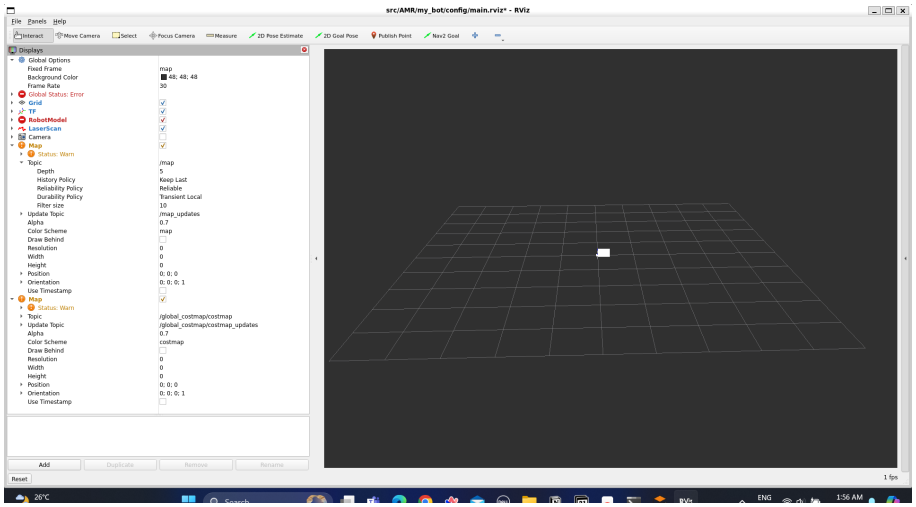


Figure 6.4: RViz

```
ros2 run tele teleop_twist_keyboard
```

When executed in a new terminal, this command enables keyboard control for robot movement. **Ensure the terminal window remains active while using the keyboard** to control the robot. Instructions and key bindings for movement will be displayed in the terminal.

SLAM

To generate this map, run the SLAM node in a **new terminal:**

```
ros2 launch my_bot online_async_launch.py
```

Use the teleop keyboard to move the robot around the Gazebo simulation while mapping the environment via the LiDAR sensor. (teleop keyboard terminal should be the active terminal)

This project uses the SLAM Toolbox with custom parameters that can be configured in the accompanying .yaml file.

Move the robot using the teleop keyboard while the SLAM node is running to map the desired environment.

Saving the Map

After mapping the environment, save the map with the following command in a **new terminal:**

```
ros2 run nav2_map_server map_saver_cli -f my_map
```

Replace my_map with a suitable name for the saved map. The map will be saved in the /amr_ws directory.

```
~  source /opt/ros/humble/setup.zsh
~  cd amr_ws
~/amr_ws  source install/setup.zsh
~/amr_ws  ros2 run tele teleop_twist_keyboard

This node takes keypresses from the keyboard and publishes them
as Twist/TwistStamped messages. It works best with a US keyboard layout.
---------------------------
Moving around:
   u    i    o
   j    k    l
   m    ,    .

For Holonomic mode (strafing), hold down the shift key:
---------------------------
   U    I    O
   J    K    L
   M    <    >

t : up (+z)
b : down (-z)

anything else : stop

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%

CTRL-C to quit

currently:      speed 0.5       turn 1.0
```

Figure 6.5: Teleop twist keyboard

Navigation

We can import the above saved base map and use that for localization and navigation. In order to do that, first stop the SLAM node by pressing Cmd+C in the keyboard of the terminal which was used to run the SLAM node.

After that to import the map and enable loacalization by executing the following command in a **new terminal:**.

```
ros2 launch my_bot localization_launch.py map:=my_map.yaml
```

After importing the above map, in the RViz window initial position of the robot should be selected. Look at the Gazebo world and estimate where could be the robot in RViz map. Then use the **"2D Pose Estimate" button to select the position in the map with the correct direction.**

Then to enable navigation for the robot, execute the following command to launch the Nav2 stack in a **new terminal:**.

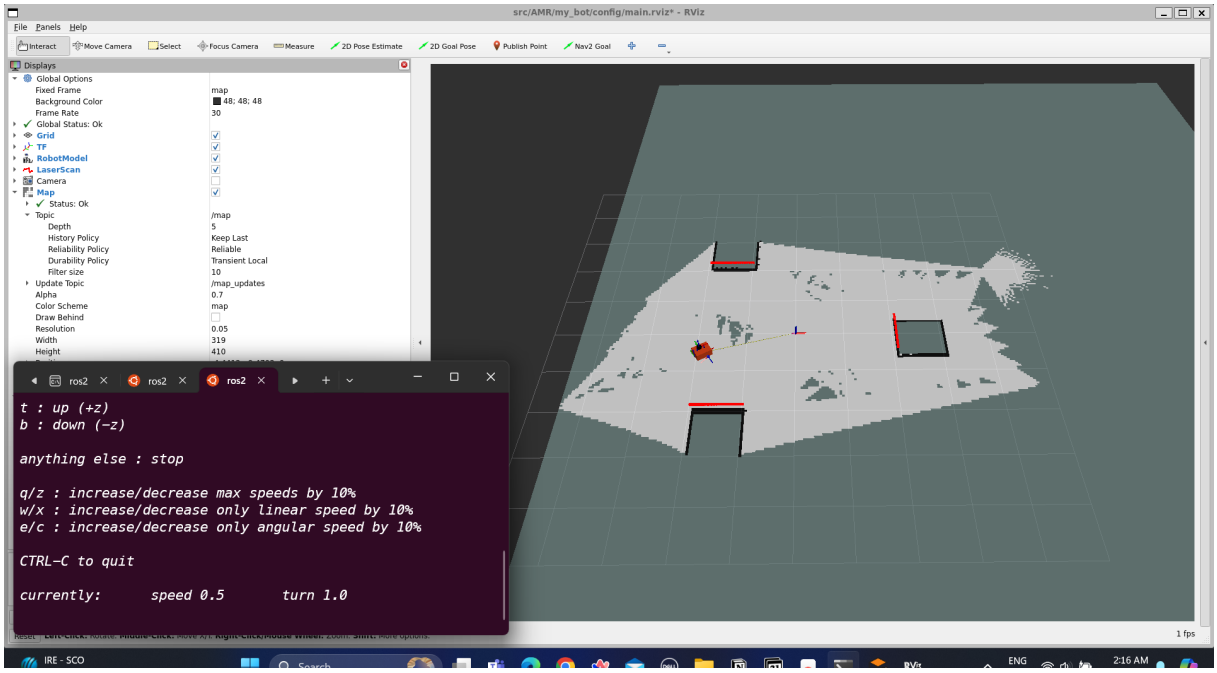


Figure 6.6: While mapping

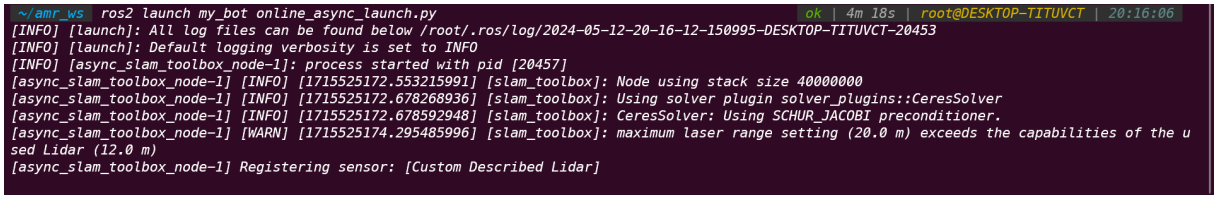


Figure 6.7: SLAM

```
ros2 launch my_bot navigation_launch.py map_subscribe_transient_local:=true
```

By using the above commands, no longer base map will be changed, but if there is new obstacles is presented in the environment, robot will able to avoid that.

After launching, select a goal in Rviz using "2D Goal Pose" button, and the robot should autonomously navigate to that goal while avoiding obstacles. When selecting the goal, the direction of the goal also should be given.

Alternate Navigation using SLAM toolbox

Instead of stopping the SLAM node and running the Localization node as mentioned, we can continue navigating using the SLAM node without the Localization node. In this case, we will not import the map as before. Instead, we will keep the SLAM node running and execute the same command to run the Nav2 Stack in a **new terminal:**.

```
ros2 launch my_bot navigation_launch.py map_subscribe_transient_local:=true
```

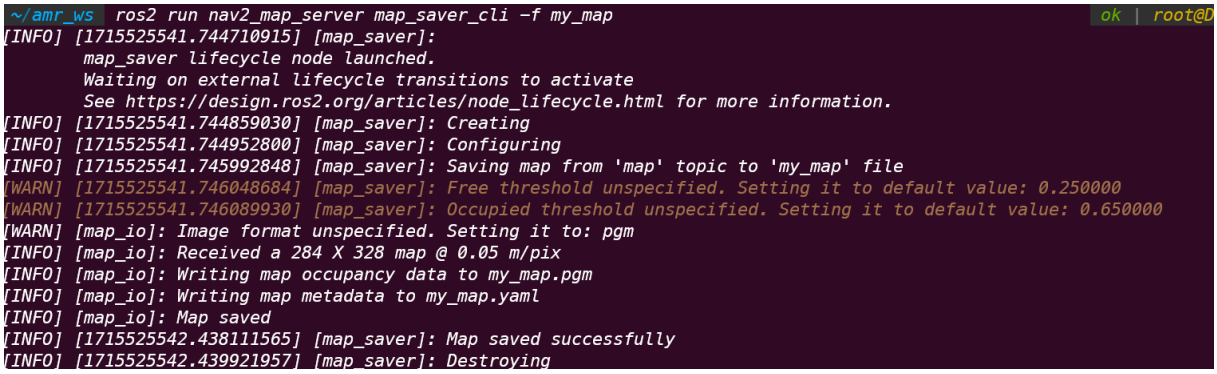


Figure 6.8: Saving Map

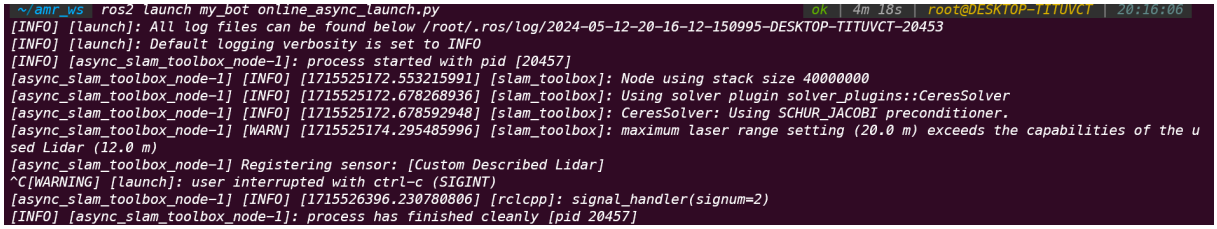


Figure 6.9: Stop SLAM Node

However, since the SLAM node is still running, the base map will be permanently altered, even if temporary obstacles are present in the environment.

6.1.3 Moving a Real Robot

In this section, we'll explore how to synchronize a real robot with the simulated robot from Gazebo. Since we don't have access to an actual LiDAR sensor, we'll control the real robot to mirror the movement of the Gazebo simulation. The real robot will move in the same directions as the simulated one.

System Interaction

The "diffdrive_arduino" plugin available on GitHub is used within the ROS2 system to convert the robot's velocities from the Rviz simulation into PWM signal commands. These PWM commands (values between 0 and 255) are sent via serial communication, which is captured by Arduino and translated into motor actions. This seamless integration ensures that the real robot mimics the movements of its virtual counterpart in the Gazebo environment, providing an effective tool for real-world testing and development of mobile robotic applications.

To achieve this, follow these steps:

Connect the Motors via Arduino Nano

Use an Arduino Nano as an intermediary between the motor drivers and the Ubuntu system.

- Clone the repository containing the Arduino bridge code and upload the **ROSArduinoBridge.uno** file (located in ROSArduinoBridge folder) to the Arduino Nano using the Arduino IDE.
- Disconnect the port from the Arduino IDE (or close the IDE) once the upload is complete.

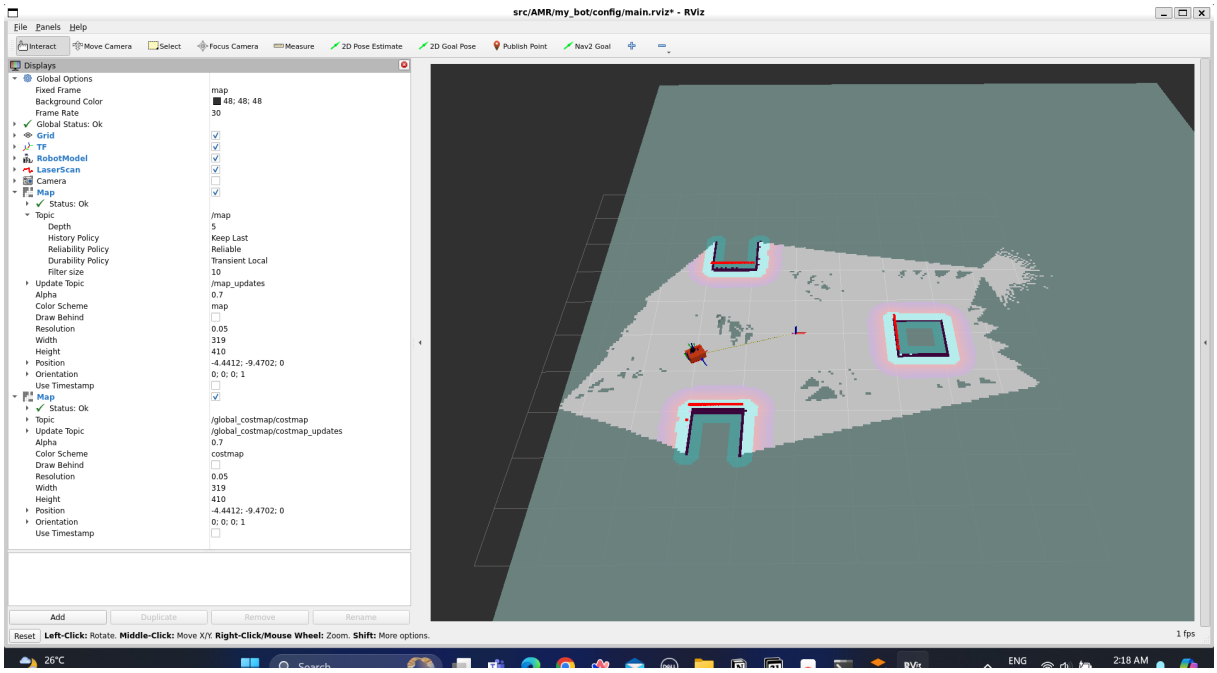


Figure 6.10: Costmap - After running the Nav2 Stack

- Connect the motor drivers to the Arduino Nano and connect the Nano to the Ubuntu system via USB.

Rerun Gazebo Simulation and Keyboard Control Codes

With the Arduino Nano properly set up, execute the same Gazebo simulation and keyboard control codes used for the simulated robot:

Gazebo Environment Simulation:

```
ros2 launch my_bot launch_robot_sim.launch.py \
world:=./src/AMR/my_bot/worlds/obstacles.world
```

Teleop Keyboard Control:

```
ros2 run tele teleop_twist_keyboard
```

Mapping and Navigation: Run the necessary SLAM node and Nav2 stack as before, following the same instructions from the "Running Simulation" section.

By following these steps, both the Gazebo-simulated robot and the real robot should move in the same directions simultaneously, providing real-world movement and mapping in tandem with the simulation.

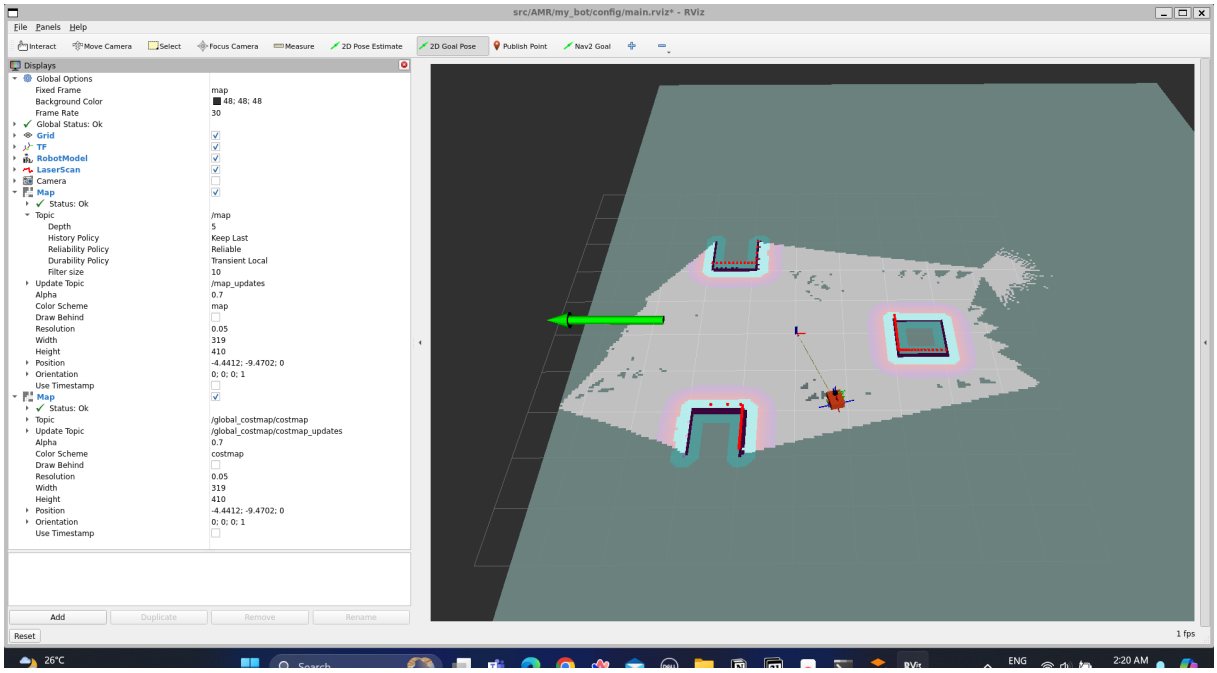


Figure 6.11: 2d Goal Selection

6.2 Trobuleshooting

6.2.1 Colcon build error

Ensuring the underlay is sourced before building is crucial. If the setup is not automatically sourced at startup (refer to Section 1.3.3, Step 1), the underlay must be sourced in every new terminal before running colcon build. Failure to do so may result in the following build error. To avoid this execute the following sourcing command

```
CMake Error at CMakeLists.txt:5 (find_package):
  By not providing "Findament_cmake.cmake" in CMAKE_MODULE_PATH this project
  has asked CMake to find a package configuration file provided by
  "ament_cmake", but CMake did not find one.

  Could not find a package configuration file provided by "ament_cmake" with
  any of the following names:

    ament_cmakeConfig.cmake
    ament_cmake-config.cmake

  Add the installation prefix of "ament_cmake" to CMAKE_PREFIX_PATH or set
  "ament_cmake_DIR" to a directory containing one of the above files.  If
  "ament_cmake" provides a separate development package or SDK, be sure it
  has been installed.
```

Figure 6.12: Error while colcon build

```
# Replace ".bash" with your shell if you're not using bash.
# Possible values are: setup.bash, setup.sh, setup.zsh
```

```
source /opt/ros/humble/setup.bash
```

6.2.2 Not sourcing underlay

If the setup file is not sourced properly following error will occur.

```
~ ros2 launch my_bot launch_robot_sim.launch.py
zsh: command not found: ros2
```

Figure 6.13: Underlay sourcing error

6.2.3 Not sourcing overlay

If the overlay is not sourced properly, the following error will occur, indicating that the package could not be found.

```
~/amr_ws ros2 launch my_bot launch_robot_sim.launch.py
Package 'my_bot' not found: "package 'my_bot' not found, searching: ['/opt/ros/humble']"
```

Figure 6.14: Enter Caption

6.2.4 Initial pose error

If the initial pose of the robot was not selected as mentioned by "2D Pole Estimate" button, the following error will be shown in the Localization terminal

```
[amcl-2] [WARN] [1716234771.062131960] [amcl]: AMCL cannot publish a pose or update the transform. Please set the initial pose...
[amcl-2] [WARN] [1716234773.065743258] [amcl]: AMCL cannot publish a pose or update the transform. Please set the initial pose...
[amcl-2] [WARN] [1716234775.070584393] [amcl]: AMCL cannot publish a pose or update the transform. Please set the initial pose...
[amcl-2] [WARN] [1716234777.074535261] [amcl]: AMCL cannot publish a pose or update the transform. Please set the initial pose...
[amcl-2] [WARN] [1716234779.078228891] [amcl]: AMCL cannot publish a pose or update the transform. Please set the initial pose...
```

Figure 6.15: Enter Caption

Appendix A

Previously used Arduino Code

```
const int motor1PWM = 6; // PWM pin for motor 1
const int motor1Dir = 2; // Direction pin for motor 1
const int motor2PWM = 5; // PWM pin for motor 2
const int motor2Dir = 3; // Direction pin for motor 2
int pwm1 = 0, pwm2 = 0;

void setup() {
  // Initialize serial communication at 9600 bps
  Serial.begin(9600);

  // Set PWM pins as output
  pinMode(motor1PWM, OUTPUT);
  pinMode(motor2PWM, OUTPUT);

  // Set direction control pins as output
  pinMode(motor1Dir, OUTPUT);
  pinMode(motor2Dir, OUTPUT);

  // Set initial PWM values to 0
  analogWrite(motor1PWM, 0);
  analogWrite(motor2PWM, 0);
}

void loop() {
  static char buffer[20];
  static int index = 0;

  // Read PWM values from the serial input
  while (Serial.available() > 0) {
    char c = Serial.read();
    if (c == '\n' || c == '\r') {
      buffer[index] = '\0';

      // Print the received raw buffer to the Serial Monitor
      Serial.print("Received buffer: ");
      Serial.println(buffer);

      // Parse the received PWM values
      char* ptr = strtok(buffer, " ");
      if (ptr != NULL) pwm1 = atoi(ptr);
      ptr = strtok(NULL, " ");
      if (ptr != NULL) pwm2 = atoi(ptr);
```

```
      // Print the received PWM values to the Serial Monitor
      Serial.print("Received PWM1: ");
      Serial.println(pwm1);
      Serial.print("Received PWM2: ");
      Serial.println(pwm2);

      // Set motor speeds
      setMotorSpeed(1, pwm1);
      setMotorSpeed(2, pwm2);

      // Reset the index for the next read
      index = 0;
    } else {
      if (index < sizeof(buffer) - 1) {
        buffer[index++] = c;
      }
    }
  }
}

void setMotorSpeed(int motor, int speed) {
  if (motor == 1) {
    if (speed >= 0) {
      digitalWrite(motor1Dir, LOW); // Set direction to forward
      analogWrite(motor1PWM, speed); // Set PWM duty cycle
    } else {
      digitalWrite(motor1Dir, HIGH); // Set direction to backward
      analogWrite(motor1PWM, -speed); // Set PWM duty cycle
    }
  } else if (motor == 2) {
    if (speed >= 0) {
      digitalWrite(motor2Dir, LOW); // Set direction to forward
      analogWrite(motor2PWM, speed); // Set PWM duty cycle
    } else {
      digitalWrite(motor2Dir, HIGH); // Set direction to backward
      analogWrite(motor2PWM, -speed); // Set PWM duty cycle
    }
  }
}
```

Appendix B

Daily Log Details

B.1 Project Selection and Proposal (01 - 29 February 2024)

After forming our group, we proposed the Autonomous Mobile Robot Controller (AMRC) project to the instructor. This involved presenting a detailed project proposal outlining our objectives, scope, and expected outcomes. Subsequently, we engaged in extensive discussions with the instructor to refine the project goals and ensure alignment with course objectives.

We also conducted a thorough assessment of each group member's skills and expertise relevant to the project. This process included reviewing technical competencies, previous experience, and specific contributions each member could bring to the team.

Following these discussions and evaluations, the project was formally approved by the instructor. This marked the beginning of our journey to develop and implement the AMRC project, leveraging our collective skills and expertise to achieve project success.

B.2 Conceptual Design (01 - 07 March 2024)

Upon commencement of the project, we initiated the exploration of conceptual designs for software implementation, motor driver design, and robot body design. For each component, we developed multiple design alternatives.

In our analysis, we meticulously evaluated the properties, drawbacks, and efficiency of each design option. This involved considering factors such as performance metrics, manufacturability, scalability, and integration with existing systems.

After comprehensive deliberation and comparison, we reached a consensus on the most promising design for each component. This decision was based on our collective assessment of technical feasibility, alignment with project objectives, and potential for meeting performance requirements.

Subsequently, we proceeded to implement the selected designs. This phase involved detailed planning, prototyping where necessary, and initiating the development process. By focusing on our chosen designs, we aimed to optimize the project's development trajectory and ensure robust execution of the AMRC project.

B.3 Overview of SLAM Technologies (08 - 13 March 2024)

Simultaneous Localization and Mapping (SLAM) technologies stand at the forefront of autonomous robotic navigation, providing the dual capability for robots to construct a map of an unknown environment while simultaneously tracking their location within it. In our AMR Controller Project, we delve deep into three SLAM methodologies: Google's Cartographer, SLAM toolbox and Hector SLAM. Each of these offer distinct approaches to SLAM, each with its strengths and challenges suited for different operational demands.

B.3.1 Cartographer

Cartographer is an open-source SLAM system offered by Google, designed to provide real-time 2D and 3D mapping. It employs various sensors and algorithms to create detailed maps that AMRs can utilize for navigation and operation.

Key Features

1. Submap-Based Approach: Cartographer employs a novel submap-based approach, where local consistency is maintained within submaps that are later aligned and merged to form a comprehensive global map.
2. Loop Closure Detection: It incorporates advanced loop closure techniques to correct for drift over time, ensuring maps remain reliable even over extended periods and distances
3. Versatile Sensor Integration: It supports integration with a diverse array of sensors, including LiDAR, radar, and cameras, allowing for multi-modal data assimilation to enhance mapping accuracy.

Applications in Project

1. Industrial Navigation: For environments with repetitive structures or areas that require regular updating of the map, Cartographer's adaptability makes it particularly beneficial.
2. Dynamic Obstacle Avoidance: The high-frequency update rate of Cartographer's maps ensures that AMRs can respond promptly to changes within their operational environment.

B.3.2 Hector SLAM

Hector SLAM takes a unique approach to SLAM. Unlike other SLAM systems, Hector SLAM does not rely on odometry data, which makes it highly suitable for robots equipped with LiDAR sensors but lacking sophisticated wheel encoders or inertial measurement units (IMUs).

Key Features

1. Odometry-Free Navigation: Hector SLAM's independence from odometry allows it to function in scenarios where wheel slippage or alternative forms of locomotion (like flight) render traditional odometry ineffective.
2. Fast Computation: The algorithm's computational efficiency makes it an excellent candidate for low-power hardware or AMRs with limited processing capabilities.

Applications in Project

1. Versatile AMR Designs: Given its odometry-free operation, Hector SLAM allows for a broader variety of AMR designs, expanding the possibilities for our project's scope.
2. Resilience to Environmental Factors: Its robustness to slippery surfaces or varying terrains makes it an ideal choice for industrial AMRs that must operate in less controlled environments.

B.3.3 SLAM Toolbox

SLAM Toolbox is a flexible open-source software for real-time and post-processing mapping, navigation, and loop closure detection for Autonomous Mobile Robots (AMRs). Developed by

Steve Macenski, it is designed to be a versatile solution that integrates seamlessly with ROS (Robot Operating System), providing robust mapping capabilities in both known and unknown environments.

Key Features

1. Lifecycle Management: SLAM Toolbox operates on a lifecycle management system, which enhances control over the active mapping sessions. This allows operators to configure, clean up, and shut down the mapper nodes as required, improving resource efficiency and system reliability.
2. Serializable State: The toolbox can serialize and deserialize its state, enabling long-term operation and recovery from any interruptions without losing the map data. This feature ensures continuity in AMR operations, critical for industrial applications where consistent mapping is essential.
3. Modular Framework: Designed with modularity in mind, it supports multiple mapping modes including mapping, localization, and lifelong mapping. This modularity allows users to adapt the tool to a wide range of applications and environments.

Applications in Project

1. Continuous Mapping: For projects requiring persistent AMR operation across multiple shifts or operational periods, SLAM Toolbox's ability to save and reload maps dynamically is invaluable. This facilitates continuous and consistent AMR functionality without the need for frequent recalibrations.
2. Adaptive Re-Mapping: In environments where the layout changes frequently, such as warehouses or manufacturing facilities, the lifelong mapping feature of SLAM Toolbox allows AMRs to adapt by updating their maps in real-time. This ensures navigation systems remain accurate and efficient, even as the operational landscape evolves.

B.3.4 Choosing SLAM toolbox

The choice of SLAM Toolbox as the SLAM solution for our Autonomous Mobile Robot (AMR) Controller Project is driven by several compelling factors that match the needs of our AMR system in terms of mapping accuracy, sensor integration, and real-time performance.

Lifecycle Management

SLAM Toolbox's lifecycle management system allows for more precise control over mapping sessions. This is particularly beneficial for projects where the robot needs to operate autonomously over extended periods or restart operations after interruptions without losing mapping progress.

Serializable State

The ability to serialize and deserialize the mapping state ensures that your project can maintain continuity in AMR operations, even after shutdowns or disruptions. This feature is crucial for maintaining high levels of accuracy in the maps over time, particularly in industrial settings where downtime can be costly.

Integration with ROS

As SLAM Toolbox is designed to integrate seamlessly with ROS, it benefits from the robustness, community support, and plethora of tools available within the ROS ecosystem. This integration facilitates easier development, testing, and deployment of robotic applications, streamlining the process significantly.

Efficient Resource Management

Given its lifecycle-based operation and the ability to handle operations efficiently, SLAM Toolbox is well-suited for projects where resource management is critical. It can operate effectively on hardware with varying computational power, making it flexible for deployment on different types of AMRs.

B.4 Simulation Tools (14 - 17 March 2024)

Simulation plays a pivotal role in the development of autonomous mobile robots (AMRs). I For our AMR controller project, we assessed two simulation tools—ROS/Gazebo and MATLAB—each offering unique features that cater to different aspects of our simulation requirements.

B.4.1 ROS/Gazebo

ROS paired with Gazebo simulator offers a comprehensive and realistic simulation environment for robotics. Gazebo provides the physics engine for accurate modeling of robot dynamics, sensor simulation, and environmental interaction, which ROS uses to implement and test the functionality of robotic systems.

Key Features

1. High Fidelity Simulations: Gazebo's advanced physics engine enables the realistic simulation of complex mechanical interactions, sensor feedback, and real-world physics laws applied to various terrains and obstacles.
2. Sensor Data Simulation: It can simulate a wide range of sensors, such as LiDAR, cameras, and IMUs, providing valuable data for testing sensor fusion algorithms and SLAM implementations.
3. ROS Integration: The tight integration with ROS ensures that the transition from simulation to real-world deployment is as seamless as possible, with many ROS packages designed to work in both simulated and physical environments without modification.

B.4.2 MATLAB

MATLAB is a multi-paradigm numerical computing environment known for its powerful toolbox for engineers and scientists. Within the robotics domain, MATLAB offers specialized toolboxes for algorithm development, system modeling, and simulation.

Key Features

1. Robust Algorithm Development: MATLAB excels in algorithm development, offering extensive libraries for data analysis, visualization, and a high-level programming environment for implementing SLAM and other robotic algorithms.

2. Simulink for Model-Based Design: Simulink, a MATLAB-based graphical programming environment, provides a model-based design for system simulation, which includes drag-and-drop block diagramming and automatic code generation.
3. Hardware-in-the-Loop Simulation: MATLAB supports hardware-in-the-loop (HIL) simulations, allowing for testing the integration of hardware components and software algorithms in real-time.

B.4.3 Choosing Gazebo

After careful consideration of our project's needs and the capabilities of both ROS/Gazebo and MATLAB, we have chosen to move forward with ROS/Gazebo as our primary simulation environment. The decision is rooted in several key factors that align with the core objectives of our AMR controller project.

Realism and Accuracy

ROS/Gazebo provides a high-fidelity simulation environment that closely replicates the physics and dynamics of the real world. This accuracy is crucial for testing our SLAM algorithms, ensuring that our robot's interactions with the simulated environment will be reflective of its behavior in actual operational settings.

Integration with Robotic Systems

The ecosystem of ROS is designed with robotic systems in mind, offering out-of-the-box support for a variety of robotic hardware and software components. Gazebo, as a part of this ecosystem, enables seamless integration of simulation data with the ROS nodes and topics we will use in the real robot, minimizing the transition time from simulation to real-world testing.

Sensor and Environment Simulation

Gazebo offers an extensive suite of simulated sensors that we can use to generate the diverse range of data our SLAM algorithms require. Its ability to accurately simulate complex environments, including dynamic changes and interactions, allows us to test our AMR controller under various scenarios and conditions without the need for physical prototypes.

In conclusion, ROS/Gazebo stands out as the more suitable option for our AMR controller project, offering a comprehensive simulation platform that meets our current requirements and provides the flexibility to accommodate future advancements. By choosing ROS/Gazebo, we are equipping ourselves with a robust tool that will aid in the development of a reliable, efficient, and cutting-edge AMR controller.

B.5 ROS1 or ROS2 (18 - 20 March 2024)

Robot Operating System (ROS) has long been the standard for robotic software development, offering tools and libraries to create complex and robust robotic applications. However, with the development of ROS2, significant improvements have been made to address some of the limitations of the original ROS. When deciding on the software framework for our AMR controller project, both ROS and ROS2 were considered.

B.5.1 ROS1

1. Maturity: ROS has a large number of packages and a mature set of tools developed over many years. It's been extensively used in academia and industry, creating a large body of knowledge and a strong community.
2. Networking: ROS uses a master-slave architecture which can be a single point of failure in the system. The networking is not designed for distributed systems or low-latency operations.
3. Real-Time: It lacks real-time capabilities, which is a significant limitation for applications requiring precise timing and high reliability.
4. Support: ROS distributions, such as Noetic, have a defined end-of-life, after which the support and updates are limited, posing a challenge for long-term projects.

B.5.2 ROS2

1. Modern Infrastructure: ROS2 is built on the Data Distribution Service (DDS), which is a more robust and flexible communication layer, allowing for distributed and real-time systems.
2. Real-Time Support: ROS2 addresses real-time requirements better than ROS, making it more suitable for applications like AMRs that require consistent and reliable timing.
3. Growing Package Ecosystem: While ROS2 currently has fewer packages than ROS, it is rapidly growing and evolving, with the expectation that it will continue to develop and expand in the coming years.
4. Long-Term Support (LTS): Distributions like ROS2 Humble provide longer support periods. This is important for our project because it guarantees we have a stable and supported framework over the years we expect our AMR system to be in operation.

B.5.3 Our Choice - ROS2

We chose ROS2 for our project primarily because of its longer-term support with the Humble distribution. Even though it may have fewer packages available compared to ROS Noetic, ROS2 is the future of robotic software frameworks, and its advantages align with the needs of our project. The improvements in communication infrastructure, real-time support, and the promise of long-term stability and growth make ROS2 the prudent choice for a project that aims to stay current and maintainable for many years. Additionally, the focus of the ROS community and ongoing development are rapidly transitioning to ROS2, ensuring we are on the cutting edge of robotic software development.

B.6 ROS2 installation and Simulation of Environment (21 - 27 March 2024)

B.6.1 ROS2 Implementation

After deciding to use ROS2, everyone in our group installed ROS2 on Ubuntu. Once the installations were complete, we began initial testing. This involved verifying the installation, configuring the environment, and running basic ROS2 nodes to ensure everything was set up correctly. These initial tests were crucial for familiarizing ourselves with ROS2 and identifying any potential issues early on.

B.6.2 Environment Simulation

To test the controller, we had to design a simulated environment before physical implementation. We were given a room to test the robot with our controller. Using measuring tools like a measuring tape, we obtained the room's dimensions.

After that, we used the 3D design software Blender to model the room and successfully imported it into Gazebo, the simulation software.

B.7 PCB Design and Finalizing (28 March - 04 April 2024)

For the motor controller design, we went through several rounds of refinement to find the exact components for the given motor. After finalizing the components, we started designing the PCB using Altium Designer. This phase also involved multiple review processes to ensure accuracy and functionality.

B.8 PCB Manufacturing (05 - 19 April 2024)

After completing the PCB design, we sent our design files to JLCPCB in China for fabrication. We chose JLCPCB due to their reputation for precision and cost-effective production. The submission process involved preparing the Gerber files, specifying the board's dimensions, layer count, and material, and selecting the desired surface finish and solder mask color. Once the order was placed, JLCPCB provided regular updates on the production status, ensuring transparency and allowing us to track the progress. The fabricated PCBs were then shipped to us for further assembly and testing.

B.9 Robot Platform Design and Production(20 - 25 April 2024)

To test the controller, we needed to design a robot. For this, we designed a platform and manufactured it using weldments. The platform served as the base for the robot, ensuring stability and structural integrity during testing.

B.10 Enclosure Design for Motor Controller (26 April - 04 May 2024)

To house the motor controller, we meticulously designed an enclosure tailored to match the exact dimensions of the PCB while ensuring efficient heat dissipation. The enclosure incorporates strategically placed ventilation slots and integrated heat sinks to manage thermal performance effectively.

B.11 PCB Assembly (05 - 10 May 2024)

After receiving the PCB from JLCPCB and the ordered components from Mouser, we began the assembly process. This involved carefully soldering each component onto the PCB. We used hand soldering to ensure precise placement and secure connections. Each component was verified against the schematic before soldering to prevent errors.

B.12 Atmega Chip Programming (24 June - 07 July 2024)

Initially, we were using the Arduino language to program the Arduino Nano (Atmega328P). However, after discussions with our instructor, we started programming the chip using C language instead. This shift allowed us to have more control over the hardware and utilize the chip's full potential. To facilitate this transition, we referred to many resources, including textbooks, online tutorials, and community forums.

B.13 Simulation Implementations (28 March 2024 - 07 July 2024)

We focused extensively on developing and integrating the SLAM algorithm for our Autonomous Mobile Robot Controller project. Initially, we searched the internet for sample codes and resources related to ROS2 SLAM algorithms. This involved exploring various algorithms to determine the most suitable option for our needs. We eventually found a promising implementation in the SLAM Toolbox. The available code catered to both simulated and real robots, so we proceeded to integrate these codes into a unified system, carefully adjusting the parameters to ensure compatibility. Subsequently, we conducted extensive testing of the simulated robot's mapping and navigation capabilities using Gazebo and Rviz. This allowed us to refine our approach and validate the effectiveness of the integrated SLAM algorithm in a simulated environment.

Chapter 7

Download Links and References

1. Ubuntu 22.04.4 LTS (Jammy Jellyfish) Installation : https://releases.ubuntu.com/jammy/
2. ROS2 Humble distributuion Documentation: https://docs.ros.org/en/humble/index.html
3. AVR Studio: https://www.microchip.com/en-us/tools-resources/archives/avr-sam-mcus
4. AVR programming: The avr microcontroIIer and embedded system using assembly and C book by MUHAMMAD ALI MAZIDI, SARMAD NAIMI and SEPMR NAIMI
5. ROS2 Creating Workspace: https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Creating-A-Workspace/Creating-A-Workspace.html
6. Github repository of teleop-twist-keyboard: https://github.com/ros2/teleop_twist_keyboard
7. Github repository of diff_drive_controller: https://github.com/ros-controls/ros2_controllers/tree/master/diff_drive_controller
8. slam-toolbox: https://github.com/SteveMacenski/slam_toolbox
9. Navigation2 Documentation: https://docs.nav2.org/
10. Project Repository: https://github.com/Bavan2002/AMR
11. ROS Arduino Bridge code: https://github.com/Bavan2002/ros_arduino_bridge.git