Helwan University

Faculty of Engineering

Computer Engineering Department

Fire Inspection Rover

A Thesis Presented to Faculty of Engineering,

Helwan University

Submitted in Partial Fulfilment of the Requirements for

a bachelor's degree in computer engineering.

By

Ziad Hesham Salah

Hosam Magdy Lotfy

Dina Salah Salim

Tarek Alaa Gaber

Supervised

By

Prof. Dr. Amr El Sayed

July 2023

Team Contribution:

Names	Chapter	Sub-Chapter
Ziad	introduction	Abstract and introduction
	Requirements	use Cases (unity, SLAM)
	Background	(Unity, SLAM)
	Technologies and Tools	(Unity, SLAM)
	Testing	unit testing for unity
	Results	(Unity, SLAM)
Dina	introduction	Acknowledgement and Literature Review
	Requirements	Functional requirements, Nonfunctional requirements, Stack holders, Actors and goals, Use case description, Use case diagram, Ros system design, Ros class diagram and Ros sequence diagram
	Testing	Ros unit testing, Ros integration testing and System testing
	Results	ROS System
Tarek	Design	Documentation of All layers, Embedded sequence diagram and Embedded Class Diagram
	Testing	introduction (unit testing & integration testing) for embedded system and conclusion
	Results	Embedded system
	Conclusion	
Hosam	Background	Embedded system
	Technologies and Tools	Embedded C (STM Blackpill) and Embedded Linux (Raspberry pi 4)
	Design	Block diagram for all system, Layered Architecture and RTOS for embedded system
	Testing	System Testing for Embedded System

Contents

Figures

1. Chapter 1 :Introduction

1.1 Acknowledgement

We would like to express my heartfelt gratitude to Prof. Dr. Amr El Sayed for his invaluable guidance, unwavering support, and profound expertise throughout the journey of completing this thesis. His insightful feedback and constructive criticism have played a pivotal role in shaping the research and refining the content. His dedication to fostering academic excellence and commitment to nurturing our growth as scholars have been truly inspiring.

1.2 Abstract

The Fire Inspection Rover presents a groundbreaking initiative aimed at revolutionizing firefighting operations and advancing the overall quality of fire response efforts. The primary motivation behind this project is to tackle the pressing challenges faced by firefighters during fire incidents, with the goal of reducing firefighter fatalities, injuries, and property damage while safeguarding the environment.

The project's core objectives encompass multiple facets, including the enhancement of firefighter safety, optimization of search and rescue missions, and support for firefighting operations in complex and hazardous environments. Through the implementation of cutting-edge technology and autonomous systems, the fire inspection rover acquires crucial environmental data from fire sites. The collected data is then analyzed and utilized to generate real-time maps, providing firefighters with invaluable insights for strategic and effective rescue planning.

One of the fundamental limitations faced by firefighters is the lack of

comprehensive visibility when entering fire sites. The fire inspection rover effectively addresses this issue by acting as an autonomous data acquisition platform, capturing crucial data on fire dynamics, temperature gradients, and structural integrity. With the provision of real-time data and mapping, firefighters gain the ability to make informed decisions, which significantly minimizes the risk of injury and maximizes their efficiency in rescue operations.

Moreover, the project recognizes the profound impact of fires not only on human lives but also on the environment. Fires contribute to environmental degradation through the release of harmful pollutants and destruction of ecosystems. To combat this, the fire inspection rover strives to minimize environmental damage by facilitating faster and more efficient firefighting responses. By reducing the extent of fire propagation and preventing prolonged exposure to hazardous materials, the rover aims to mitigate environmental deterioration and promote sustainable firefighting practices.

The incorporation of advanced communication and collaboration features in the fire inspection rover fosters seamless coordination among firefighting teams. The real-time data transmission capabilities allow multiple units to share critical information, synchronize actions, and work cohesively in addressing complex fire scenarios. This collaborative approach not only enhances the effectiveness of firefighting efforts but also promotes safer working conditions for all team members involved.

Furthermore, the project addresses the need for effective crisis management by streamlining firefighting operations. The rover's autonomous movement and data acquisition, coupled with the SLAM (Simultaneous Localization and Mapping) technology, facilitate rapid assessment and planning. The reduction in manual labor requirements allows firefighters to focus more on strategic decision-making, ultimately leading to better crisis management and an

increased ability to control and extinguish fires promptly.

In addition to its active firefighting applications, the fire inspection rover plays a pivotal role in training and preparing firefighters for diverse fire situations. The integration of a sophisticated Unity simulator provides an immersive and realistic training environment, where firefighters can practice and refine their skills in tackling simulated fire scenarios and dynamic environments. This training not only enhances their proficiency but also fosters adaptability and readiness for unforeseen challenges.

In conclusion, the Fire Inspection Rover project represents an ambitious endeavor towards improving firefighting operations and ensuring the safety of both firefighters and fire victims. By combining cutting-edge technology, autonomous systems, and data-driven decision-making, the rover significantly elevates the efficacy and efficiency of firefighting efforts. With its focus on enhancing safety, reducing environmental degradation, and facilitating comprehensive training, the project aims to leave an enduring impact on firefighting practices and ensure a safer and more sustainable future for communities worldwide.

1.3 Introduction

Fires have been a formidable adversary for humanity since time immemorial, threatening lives, property, and the environment. As fire incidents continue to pose serious challenges for firefighting professionals, there is a pressing need for innovative solutions that enhance their effectiveness and minimize risks. In response to this imperative, the Fire Inspection Rover project endeavors to bring about a transformative shift in firefighting operations, leveraging advanced technologies to improve response capabilities and ensure the safety of both firefighters and the public.

In 2017, Egypt witnessed the occurrence of 45,700 fires, followed by 46,323 incidents in 2018, an increase of 1.5% and fires continued to rise in 2019, going up to 50,662 an increase of 9.4%. These same fires were responsible for 252 civilian deaths including three firefighters. In the same year, 54 firefighters were fatally injured and 1,203 experienced nonfatal injuries.(1)

Egyptian state announced that the economic cost of fires in Egypt amounted to EGP 3.6 bn over the last ten years, according to the Central Agency for Public Mobilization and Statistics (CAPMAS). In 2017, the cost of fires was EGP 457.5m, followed by EGP 391.2m in 2018, and EGP 523m in 2019.(2)

The motivation behind the Fire Inspection Rover project stems from the stark realities faced by firefighters on the frontlines. Every year, numerous firefighters lose their lives or suffer injuries while battling fires, and vast amounts of property are ravaged, inflicting significant economic losses. The project aims to address these challenges and mitigate the impact of fires on society by developing a cutting-edge autonomous system capable of collecting environmental data, providing real-time insights, and optimizing rescue planning.

Furthermore, the project places considerable emphasis on crisis management. The fire inspection rover's autonomous movement, data acquisition, and SLAM technology converge to streamline firefighting operations and optimize crisis response efforts. By reducing the demand for manual labor in hazardous and life-threatening operations, the rover empowers firefighters to focus on strategic decision-making, resulting in better crisis management and improved control over fire incidents.

1.4 Literature Review

1.4.1 Autonomous Embedded System Vehicle Design on Environmental, Mapping and Human Detection Data Acquisition for Firefighting Situations By : Armando Pinales and Damian Valles (Ingram School of Engineering)(3)

The paper proposes an autonomous embedded system vehicle (AESV) design that can be used to assist firefighters in firefighting situations. The AESV is equipped with a variety of sensors, including thermal sensors, gas sensors, and LiDAR sensors. These sensors allow the AESV to collect data about the environment, such as the temperature, the presence of harmful gases, and the layout of the building. The AESV also has a human detection system that can be used to identify the presence of people in the building.

The AESV is controlled by a microcontroller system that processes the data collected by the sensors. The microcontroller system also controls the communication between the AESV and the firefighters. The AESV can transmit data to the firefighters, such as the temperature, the presence of harmful gases, and the layout of the building. The firefighters can also control the AESV remotely, such as by sending it to a specific location or by instructing it to avoid a particular area.

The paper evaluates the performance of the AESV in a simulated firefighting situation. The results show that the AESV was able to successfully collect data about the environment and identify the presence of people in the building. The AESV was also able to transmit data to the firefighters and receive instructions from them.

The paper concludes that the AESV has the potential to be a valuable tool for firefighters. The AESV can help firefighters to gather information about the environment and to identify the presence of people in the building. This information can be used by firefighters to make better decisions about how to fight a fire.

The paper also discusses some of the challenges that need to be addressed before the AESV can be deployed in real-world firefighting situations. These challenges include the development of more robust sensors, the development of a more sophisticated communication system, and the development of a more user-friendly interface.

Paper limitations:

1. The paper uses ZigBee protocol which is based on IEEE 802.15.4 standard ZigBee protocol have a limited range of 100M also ZigBee is not as fast as other wireless protocols, such as Wi-Fi.
2. The paper implied no means of robot-to-robot communication or across machine communication.
3. The paper had no means of using a simulator engine to simulate robots in different environments and fire situations.

1.4.2 A Fire Reconnaissance Robot Based on SLAM Position, Thermal Imaging Technologies, and AR Display By: Sen Li, Chunyong Feng, Yunchen Niu, Long Shi, Zeqi Wu and Huaitao Song(4)

The paper introduces the problem of fire reconnaissance and the challenges that it poses. Fire reconnaissance is the process of gathering information about a fire, such as the location of the fire, the size of the fire, and the presence of people or

animals. This information is used by firefighters to make decisions about how to fight the fire.

The paper then discusses the challenges of fire reconnaissance. These challenges include:

The environment: Fires often occur in dangerous environments, such as burning buildings or collapsed structures. These environments can be hazardous for firefighters, and they can make it difficult to gather information about the fire.

Smoke can obscure visibility and make it difficult to see the fire. This can make it difficult to determine the location of the fire, the size of the fire, and the presence of people or animals.

The heat: Fires can produce high temperatures, which can make it difficult for firefighters to operate in the environment. This can also make it difficult to use sensors and other equipment to gather information about the fire.

Proposed Solution

The paper proposes a fire reconnaissance robot that can help to address these challenges. The robot is equipped with a variety of sensors, including a thermal imaging camera, a LiDAR scanner, and an inertial measurement unit (IMU). These sensors allow the robot to gather information about the environment, such as the temperature, the presence of smoke, and the layout of the building.

The robot also uses a simultaneous localization and mapping (SLAM) algorithm to build a map of the environment. This map can be used by the robot to navigate the environment and to avoid obstacles.

The robot also has an augmented reality (AR) display that can be used to display information to firefighters. This information can include the location of the fire, the size of the fire, and the presence of people or animals.

Figure 7. The operation of the developed fire reconnaissance robot.

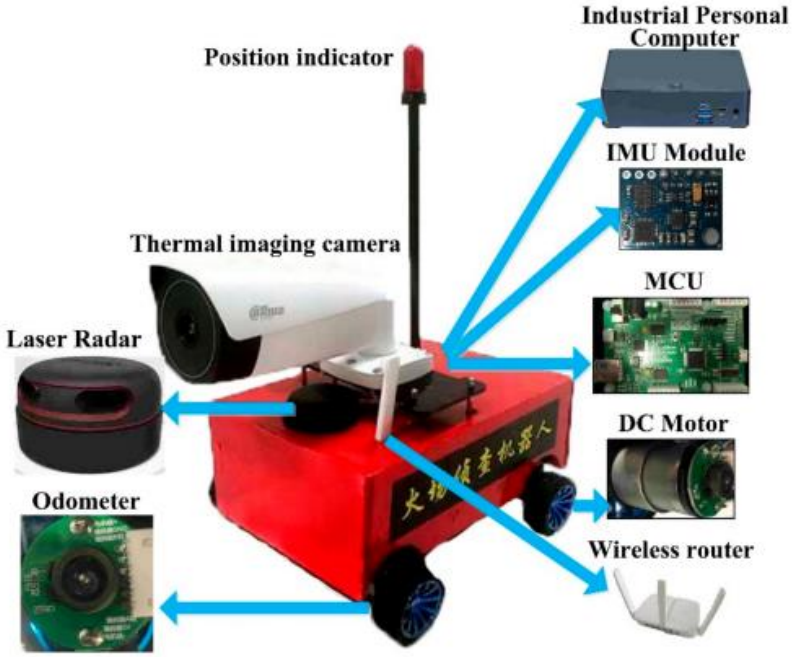

Figure 1:AR Robot Paper

The paper evaluates the performance of the robot in a simulated fire scenario. The results show that the robot was able to successfully gather information about the environment and to navigate the environment. The robot was also able to display information to firefighters in a timely and accurate manner. The paper concludes that the proposed robot has the potential to be a valuable tool for firefighters. The robot can help firefighters to gather information about the environment and to identify the presence of people or animals. This information can be used by firefighters to make better decisions about how to fight a fire.

Paper limitations:

The paper uses AR so that a fireman could see inside the fire sight from a safe distance that will necessarily require a fireman pilot for every robot as the robot have no means of autonomous navigation or mapping the data is Visualized through the AR glass only so it's difficult to visualize a large-scale fire sight. The paper had no means of using a simulator engine to simulate robots in different environments and fire situations.

1.4.3 Multi-Agent Systems for Search and Rescue Applications by Daniel S. Drew(5)

The paper discusses the use of multi-agent systems (MAS) for search and rescue (SAR) applications. MAS are a type of distributed system that consists of a collection of autonomous agents that cooperate to achieve a common goal.

The paper begins by discussing the challenges of SAR. SAR is a complex and challenging task, as it often involves searching large and dangerous environments for survivors. The paper then discusses how MAS can be used to address these challenges.

Multi-Agent Systems for SAR

MAS can be used to address the challenges of SAR in several ways. First, MAS can be used to distribute the workload of SAR. This can be done by dividing the search area into smaller sub-areas and assigning each sub-area to a different agent.

Second, MAS can be used to coordinate the actions of the agents. This can be done by using a communication protocol to allow the agents to share information and to coordinate their actions.

Third, MAS can be used to adapt to changes in the environment. For example, if the environment becomes too dangerous for one agent, the other agents can be re-tasked to continue the search.

Evaluation of MAS for SAR

The paper evaluates the use of MAS for SAR by discussing several case studies. Case studies show that MAS can be used to effectively improve the performance of SAR.

Conclusion

The paper concludes by discussing the future of MAS for SAR. The paper argues that MAS has the potential to revolutionize SAR, and that they will become increasingly important in the future.

In addition to the above, here are some other key points from the paper:

MAS can be used to improve the efficiency of SAR by distributing the workload and coordinating the actions of the agents. MAS can be used to improve the effectiveness of SAR by adapting to changes in the environment. MAS can be used to improve the safety of SAR by reducing the risk to human operators.

The paper concludes by discussing the challenges that need to be addressed before MAS can be widely deployed for SAR. These challenges include the development of more robust MAS platforms, the development of more effective communication protocols, and the development of more user-friendly interfaces.

2. Chapter 2 : Analysis and Requirements:

The analysis and requirements chapter focuses on outlining the systematic study and evaluation of the project's objectives, constraints, and user needs.

This chapter serves as a foundation for the entire development process, providing a clear understanding of the problem domain and guiding the subsequent design and implementation phases.

In the context of the fire inspection rover, the analysis and requirements chapter focus on gathering and analyzing information related to the requirements and constraints of the rover's functionality and performance.

In addition to creating possible scenarios and use cases of the project to verify that each requirement is fulfilled by these cases.

2.1 Problem Statement

In 2016, the United States experienced a staggering number of fires, with approximately 1,342,000 incidents attended to by fire departments. Tragically, these fires resulted in the loss of 3,390 civilian lives and caused significant harm, with 62,085 nonfatal injuries sustained by firefighters and 69 fatal injuries among them. These statistics highlight the urgent need for improved fire rescue and containment activities.

Fire rescue operations pose numerous challenges, including extreme weather conditions, poor air quality, limited visibility, unidentified person locations, and complex floor designs in affected buildings. These factors contribute to the difficulty and time-consuming nature of rescue operations, increasing the risk of injury or even death for both civilians and first responders.

One major issue faced by firefighters is the reliance on general-purpose

equipment that may not be suitable for specific fire sites. This mismatch between equipment and site requirements can create life-threatening situations for firefighters. Additionally, the excessive gear worn by firefighters leads to restricted movement and increased physical strain, hindering their ability to access critical areas within the fire site.

To mitigate these challenges and improve firefighting and rescue operations, it is crucial to enhance the strategies used by firefighters before entering a fire site. Relying solely on visual data, fire emissions, and audible calls for help is insufficient. There is a need for real-time, on-site data and robust statistical analysis to provide firefighters with accurate and comprehensive information.

By incorporating in-site data and leveraging robust real-time statistics, firefighters can make more informed decisions and execute effective firefighting and rescue plans. This data-driven approach would enable firefighters to assess the situation more accurately, identify potential hazards or trapped individuals, and adapt their strategies accordingly. Ultimately, it would enhance the safety and efficiency of fire rescue operations, reducing the risks faced by both civilians and first responders.

2.2 System Requirements

2.2.1 Functional Requirements

Table 1

Label	Description
REQ-1	• Environment Monitoring: The rover should be equipped with sensors to monitor environmental conditions such as temperature, Gas contents, and robot orientation within the fire-affected area.

REQ-2	• Remote Operation: The rover should support remote control and monitoring capabilities to allow firefighters or operators to intervene and control its movements and operations when necessary.
REQ-3	• Remote Sensing and Imagery: The rover should be able to capture and transmit images, maps, or other sensor data to provide visual and enumerated information about the fire's location, size, and spread.
REQ-4	• Real-time Communication: The rover should have a communication system to relay data and map feed in real-time to the control station.
REQ-5	• Fire Detection: The rover should have sensors and algorithms to detect the presence and intensity of fires. This includes visual cameras, smoke detectors, and gas sensors.
REQ-6	• Mapping and Localization: The rover should be capable of mapping the fire-affected area and localizing itself within the map. This involves using technologies like SLAM (Simultaneous Localization and Mapping).
REQ-7	• Autonomous Navigation: The rover should be able to navigate autonomously through fire-affected areas, avoiding obstacles and adapting to changing terrain conditions.
REQ-8	• Emergency Response Assistance: The rover should be capable of providing emergency response assistance by performing tasks like locating trapped individuals, relaying information to rescue teams, or creating safe paths for evacuation.

REQ-9	• User Interface: The rover's control interface should be user-friendly, allowing operators to monitor sensor data, maps and control the rover's functions easily.

2.2.2 Non-Functional Requirements

Table 2

Label	Description
REQ-10	Performance: • Real-time Response: The rover should exhibit minimal latency in detecting fires, transmitting data, and responding to commands. • Speed and Agility: The rover should be capable of navigating through the fire-affected area efficiently, ensuring quick response times.
REQ-11	Reliability: • Fault Tolerance: The system should be resilient to failures and able to handle partial system failures without compromising critical functionalities. • Alternative planning: The system should be able to compensate for missing or faulty data in a range.
REQ-12	Safety: • Emergency Shutdown: The rover should have mechanisms for emergency shutdown or stoppage in case of critical failures or hazardous situations. • Point of Retrieve: In case of system failure the rover should be

	able to retract to a safe point where it could be retrieved.
REQ-13	Usability: • Intuitive User Interface: The control interface should be user-friendly, enabling firefighters to easily understand and interact with the rover's functionalities. • Remote Operation: The rover should support remote operation capabilities, allowing operators to control and monitor the system from a safe distance. • Data Visualization: The system should provide clear and meaningful visualizations of fire-related data to data analysis and decision making.
REQ-14	Scalability: • Expandability: The rover should have provisions for future upgrades or modifications to accommodate additional functionalities or sensor integration. • Multiagent: If multiple rovers are deployed simultaneously, the system should support coordination and cooperation among them to enhance efficiency and coverage.
REQ-15	Maintainability: • Serviceability: The rover should be designed for ease of maintenance and repair, with accessible components and well-documented procedures. • Modularity: The system architecture should promote modularity, allowing individual components to be replaced or upgraded independently.

2.3 Functional Requirements Specification

2.3.1 Stakeholders

I. Fire Departments, Egypt Fire Association and Firefighters: Being the end-users of the Fire Inspection Rover, Fire departments and firefighters are primary stakeholders. They rely on the rover's capabilities to enhance their firefighting operations and improve firefighters' perception during fire incidents.

II. Emergency Management Agencies: Stakeholders from emergency management agencies, such as local or national disaster management authorities, have an interest in the rover's role for fire inspection and response. They may provide guidance, funding, or regulatory requirements for the rover's deployment.

III. Robotics Engineers and Researchers: Professionals in robotics and researchers in related disciplines are stakeholders involved in the design, development, and improvement of the fire inspection rover. They contribute to enhancing the rover's technical capabilities, autonomy, and reliability.

IV. Government and Regulatory Bodies: Government entities such as the Ministry of interiors and regulatory bodies play a role in setting standards, regulations, and guidelines for the use of robots in fire inspection and emergency response. They ensure compliance with safety, privacy, and ethical considerations.

V. System Integrators and Manufacturers: Companies specializing in robotics and autonomous systems are stakeholders involved in the integration, manufacturing, and commercialization of fire inspection rovers. They contribute to the rover's development, production, and distribution.

VI. Insurance Companies: Insurance companies have an interest in fire inspection rovers as they can help in assessing fire risks, preventing damage, and minimizing insurance claims. They may provide support or incentives for the adoption of fire inspection rovers.

VII. Public Safety Agencies: Public safety agencies, including police and emergency medical services, have a stake in fire inspection rovers as they often collaborate with fire departments during emergencies. They may benefit from the rover's capabilities for shared situational awareness and coordinated response.

VIII. Public and Community: The public and local communities can be stakeholders, as they are indirectly affected by the efficiency and effectiveness of fire inspection rovers. They may have concerns about privacy, safety, and the impact of the rover's operations on their environment.

IX. Equipment Suppliers: Suppliers of sensors, communication systems, batteries, and other components used in fire inspection rovers are stakeholders with an interest in promoting their technologies and solutions for the rover's integration.

X. Environmental and Safety Organizations: Environmental organizations and safety-focused groups may have an interest in using the fire inspection rover's data to study the environmental impact of the fire and promote safe practices during fire incidents.

2.3.2 Actors and goals

In the context of a fire inspection rover, the actors and their goals can be identified as follows:

2.3.2.1 Initiating Actors

a) Firefighter:

- Type: Initiating
- Goals:
  - Perform remote control and monitoring of the rover during fire inspection operations.
  - Obtain and assess real-time information about the fire incident, including temperature, smoke, and structural conditions.
  - Guide the rover's movements and actions to effectively assess the situation and plan firefighting strategies.
  - Ensure the rover's safe navigation through the fire-affected area.

b) Chief commander:

- Type: Initiating
- Goals:
  - Oversee and coordinate the fire inspection operation.
  - Gain situational awareness by monitoring the rover's data, Kinect stream, and sensor readings.
  - Make informed decisions based on the information provided by the rover.
  - Construct a rescue and fire extinguishing plan based on data assessed from the robot and the constructed map.

c) Control Station Operator:

- Type: Initiating

– Goals:

- Operate and control the rover from a control station.
- Monitor the rover's navigation, sensor data, and Kinect feed.
- Ensure smooth communication between the rover and the control station.
- Troubleshoot any technical issues or emergencies during rover operations.

2.3.2.2 Participating Actors

a) Fire Inspection Rover:

– Type: Participating

– Goals:

- Navigate through fire-affected areas.
- Gather and transmit relevant sensor data to the control station.
- Use SLAM algorithm to construct and transmit the Map of fire location.
- Detect fires, assess their intensity, and spread, and provide real-time information to the initiating actors.

b) Communication Systems:

– Type: Participating

– Goals:

- Facilitate real-time communication between the rover, control station, and initiating actors.
- Ensure reliable transmission of data, commands, and video feed.
- Maintain seamless connectivity in potentially challenging and hazardous environments.

c) Sensor Systems:

- Type: Participating
- Goals:
  - Collect and provide accurate and reliable sensor data related to fire, temperature, smoke, and environmental conditions.
  - Transmit the sensor data to the rover's control station or other relevant systems for analysis and decision-making.

The identified initiating actors, including firefighters, chief commanders, control station operators, and actively interact with the fire inspection rover to achieve their respective goals.

The participating actors, namely the fire inspection rover, sensor systems, Kinect and communication systems, collaborate to fulfill the needs and goals of the initiating actors.

2.3.3 Use-cases

2.3.3.1 Use case Description.

Use Case 1: Autonomous Navigation

- Description: The rover autonomously navigates through the fire affected area to collect data and detect fire location tasks.
- Actors: Fire Inspection Rover, Kinect, Sensor's system, Control station operator, Firefighter.
- Tools: Autonomous navigation algorithms, obstacle detection and avoidance, terrain adaptation.

Use Case 2: Fire Detection

- Description: The rover detects the presence and intensity of fires using Kinect and open cv algorithms.
- Actors:
- Tools: Fire detection sensors (visual, temperature, or gas), fire assessment algorithms.

Use Case 3: Remote Operation

- Description: Operator remotely controls and monitors the rover's movements and operations.
- Tools: ROS Master communication, real-time Kinect feed, command transmission using Joystick, operator interface.

Use Case 4: Environment Monitoring

- Description: The rover monitors environmental conditions within the fire-affected area, such as temperature, Gas, and air quality.
- Tools: Sensor's suit, data collection and analysis using STM microcontroller, real-time data collection.

Use Case 5: Real-time Communication

- Description: The rover relays data and Kinect feed in real-time to the control station, enabling situational awareness and decision-making.
- Tools: ROS Master Communication system, reliable data transmission, real-time Kinect streaming.

Use Case 6: Mapping and Localization

- Description: The rover maps the fire-affected area, transmits the map to the control station and localizes itself within the map to navigate the

plane effectively.

- Tools: Mapping algorithms, SLAM or GPS localization, map creation and updating, Kinect, RVIZ.

Use Case 7: Remote Sensing and Imagery

- Description: The rover captures and transmits images, maps, or other sensor data to provide visual information about the fire's location, size, and spread.
- Requirements: sensors, Kinect streaming and transmission, real-time visual analysis.

Use Case 8: User Interface

- Description: The rover's control interface is user-friendly, allowing operators to monitor and control the rover's functions easily.
- Requirements: Intuitive user interface, clear visualization of rover status and data, user-friendly controls.

2.3.3.2 Use case Diagram.

Use Case 1: Autonomous Navigation

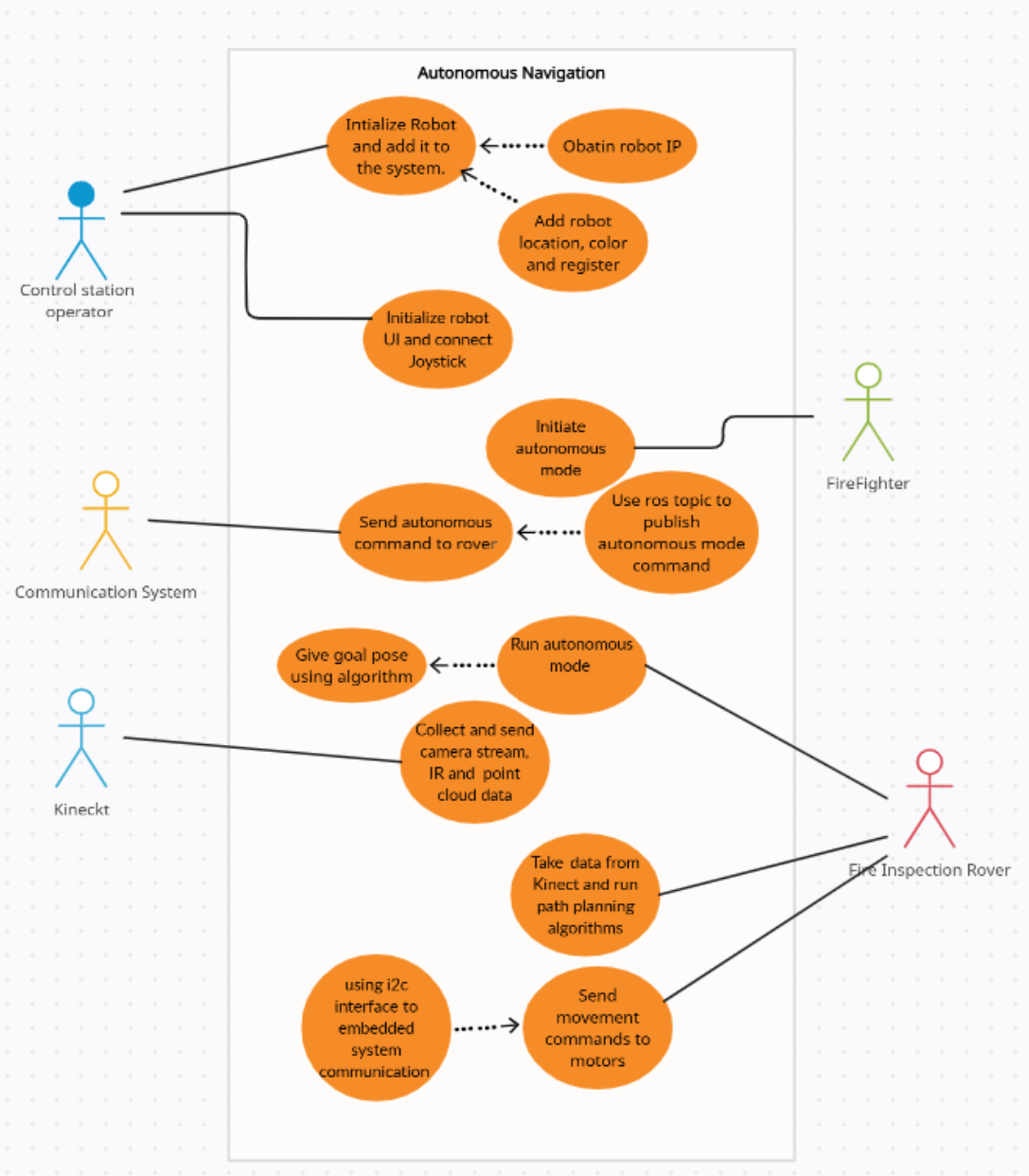

Figure 2:usecase1

Use Case 2: Remote Operation

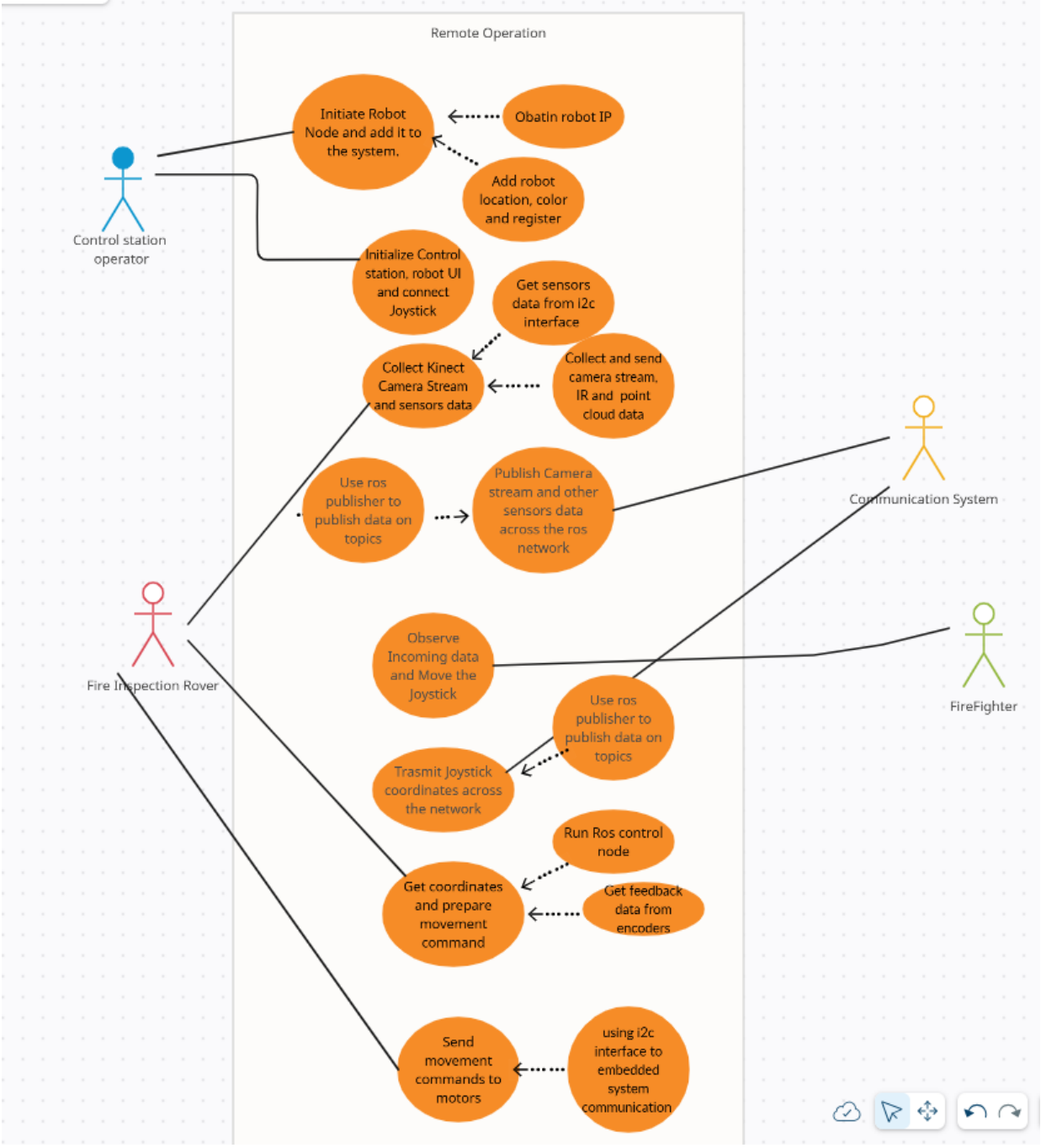

Figure 3:usecase2

Use Case 3: Environment Monitoring

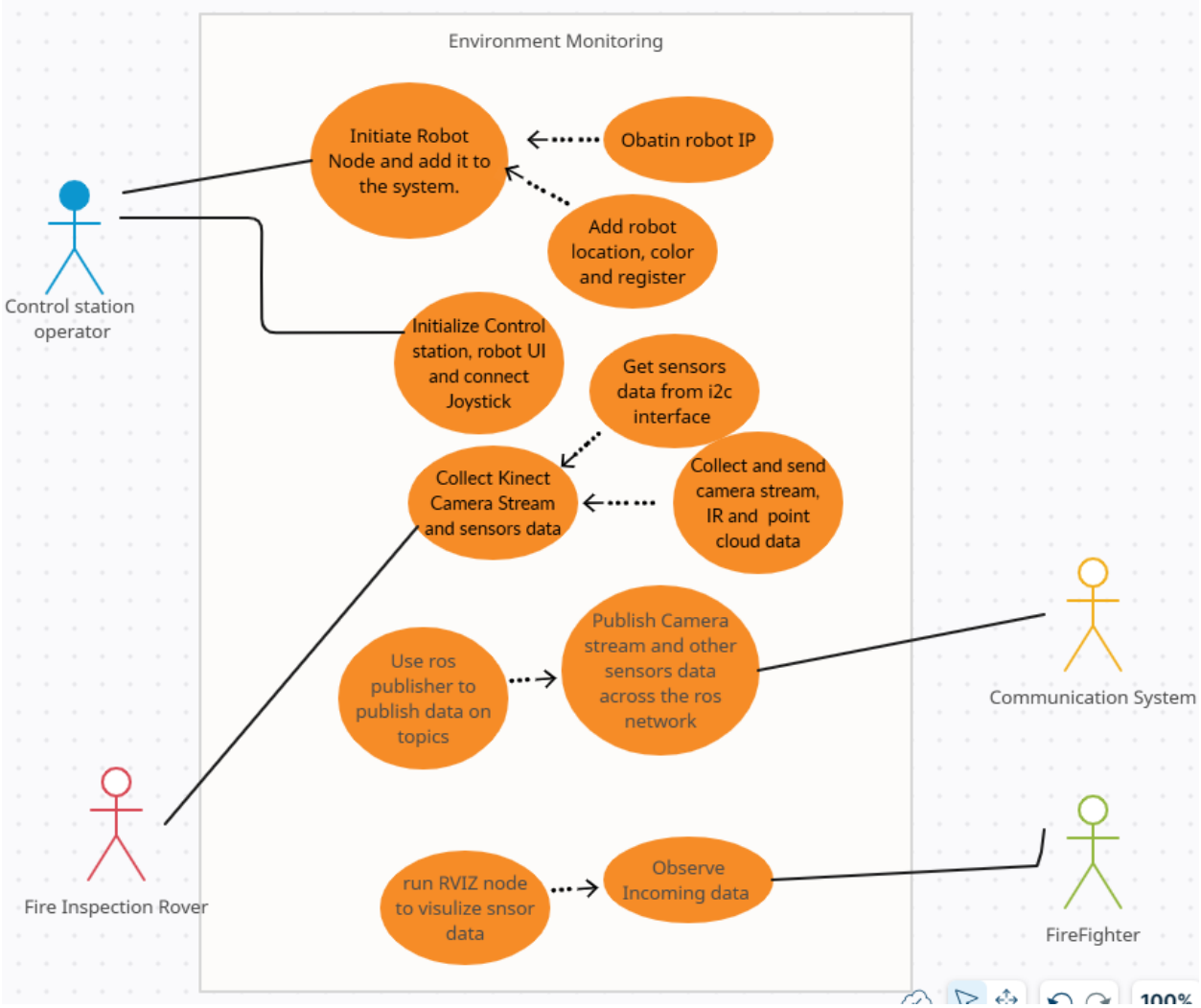

Figure 4:usecase3

Use Case 4: Real-time Communication

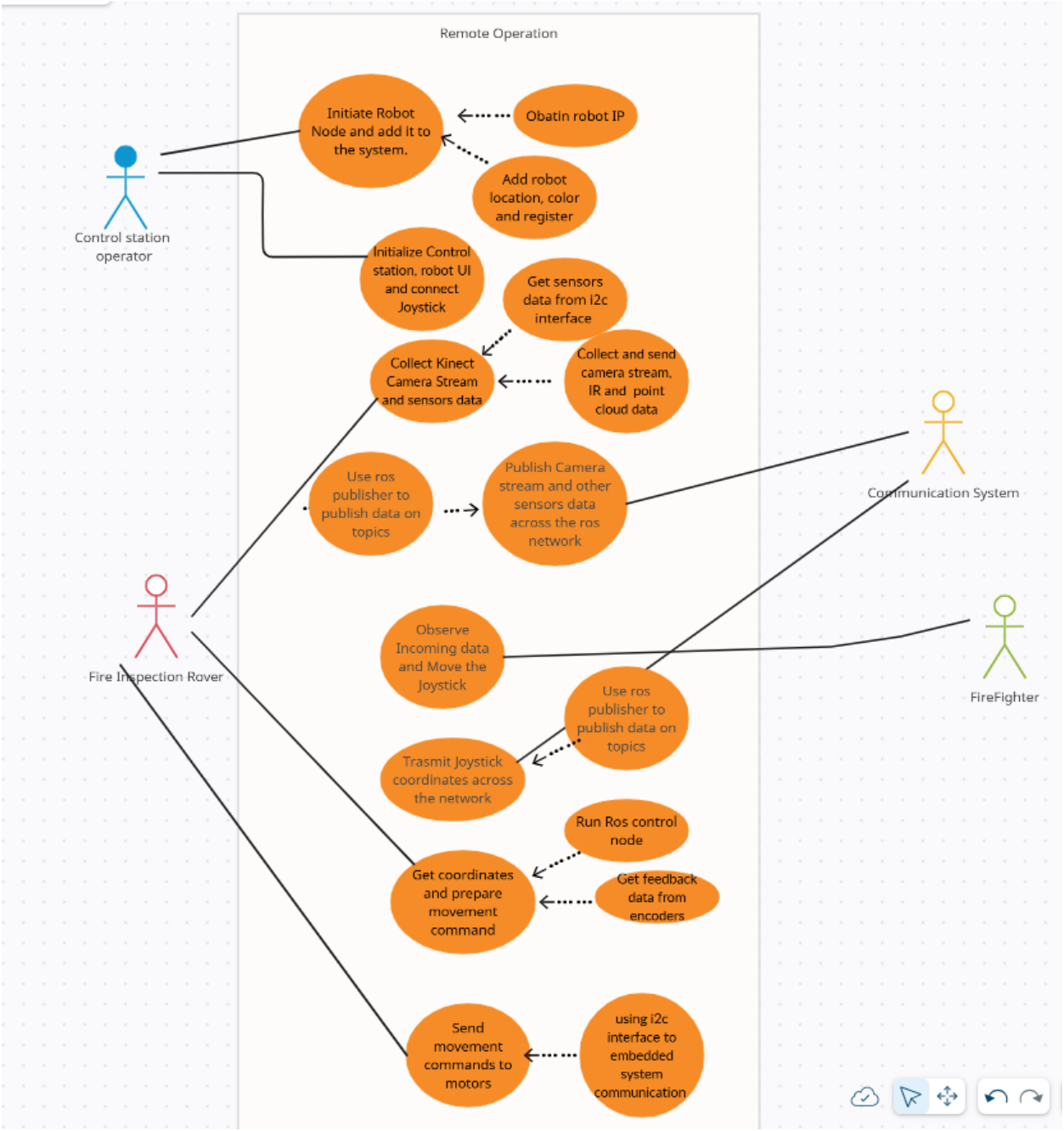

Figure 5:usecase4

Use Case 5: Mapping and Localization

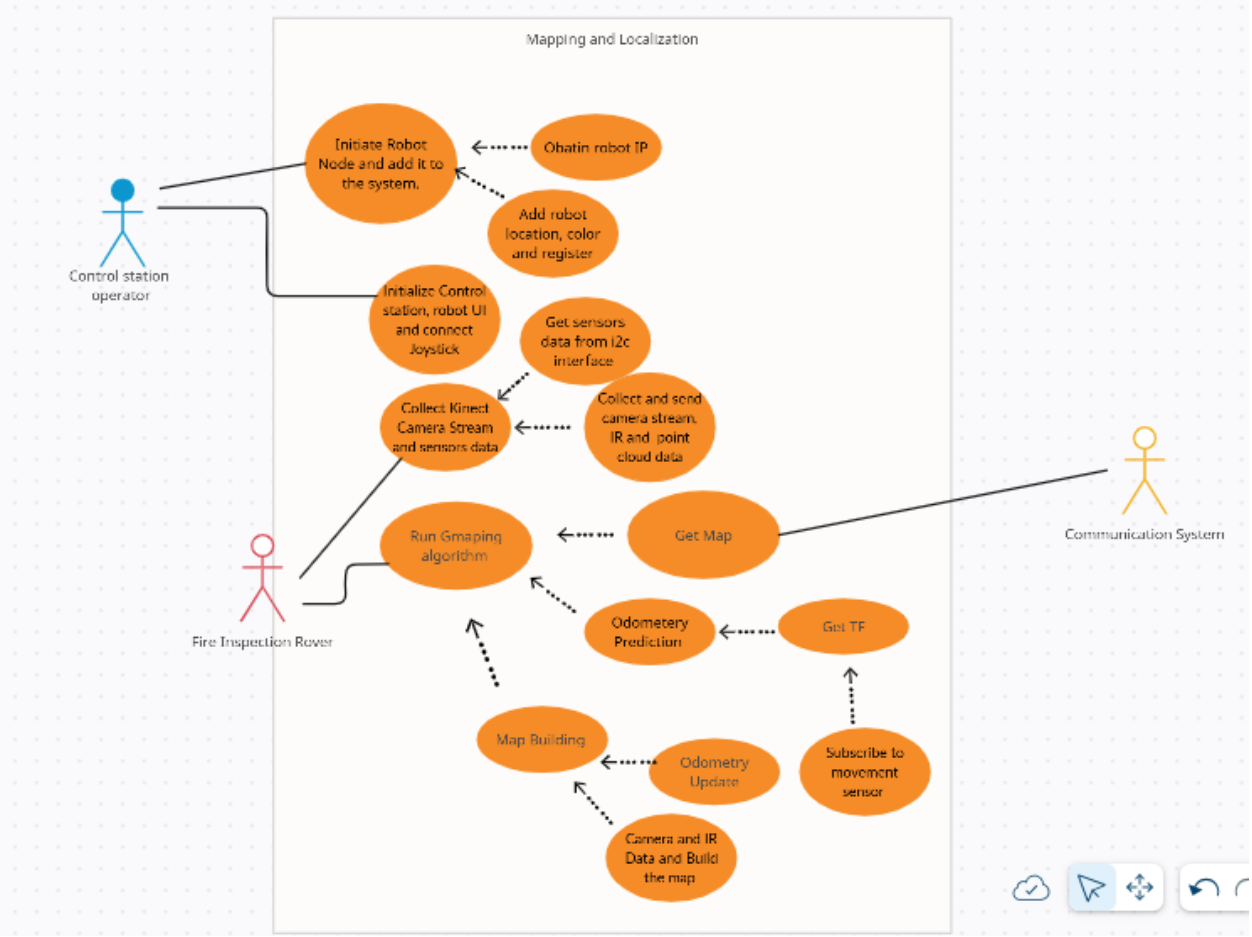

Figure 6:usecase5

Use Case 6: Remote Sensing and Imagery

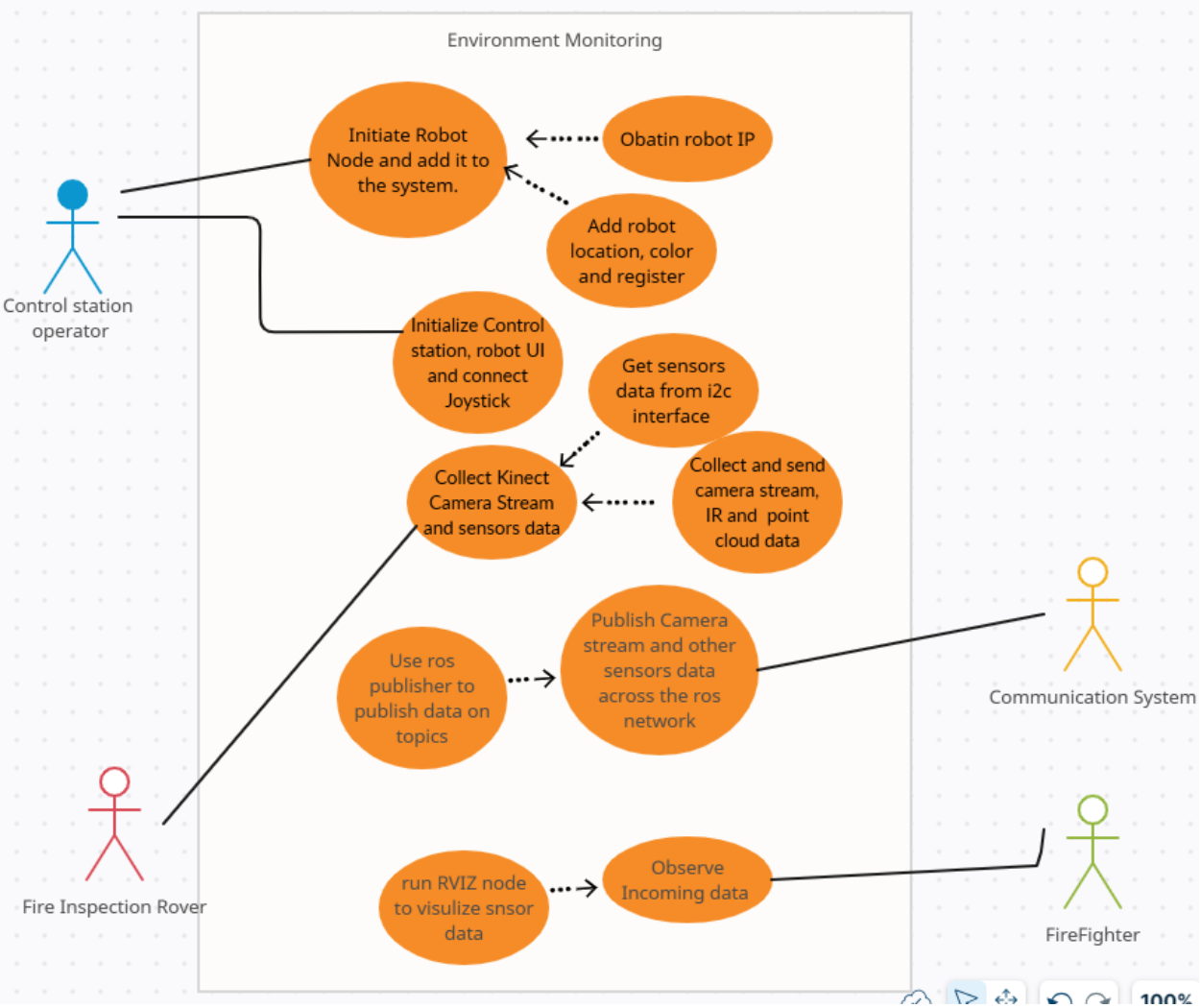

Figure 7:usecase6

Use Case 7: User Interface

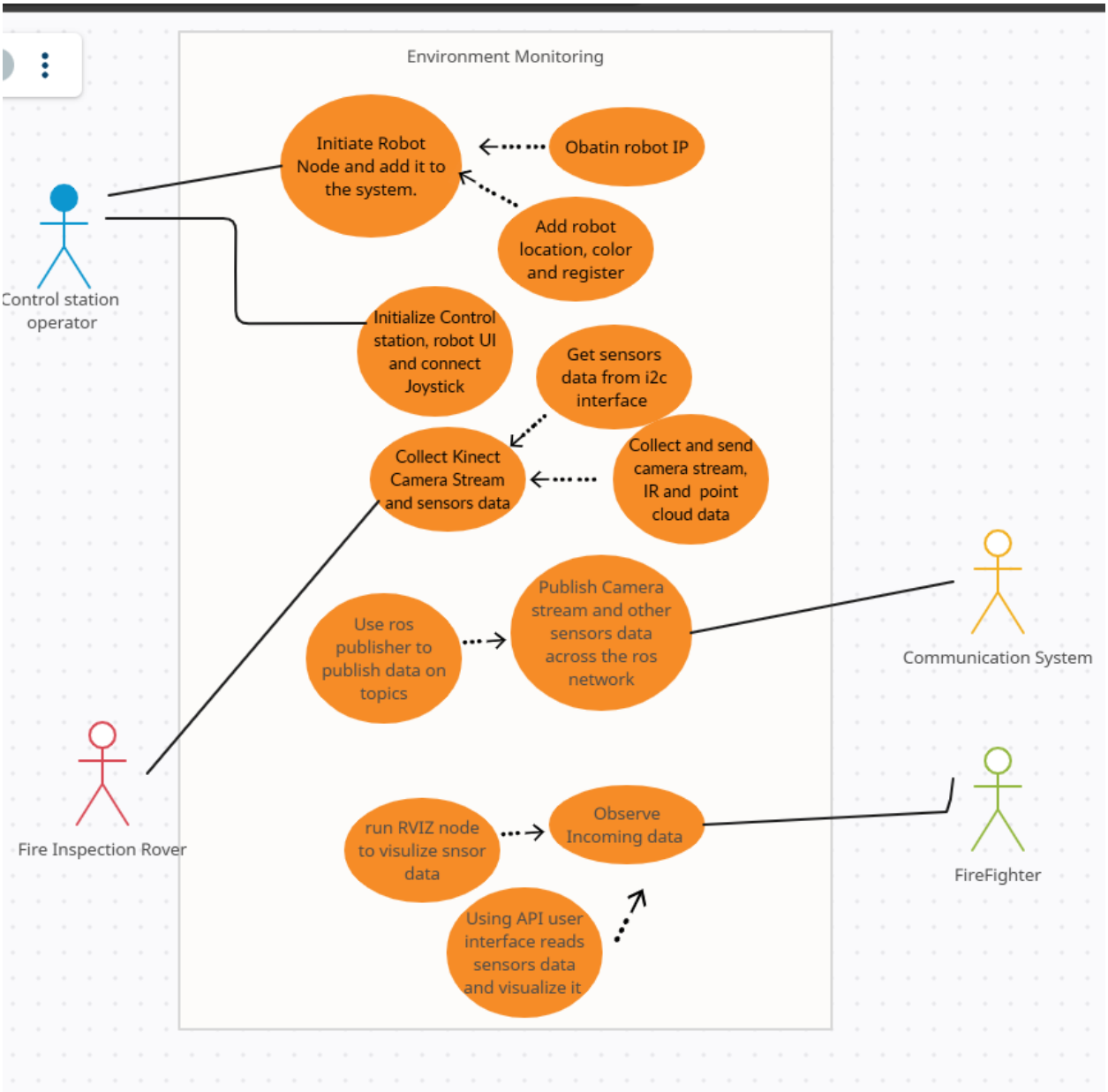

Figure 8:usecase7

In conclusion, the analysis and requirements chapter for the fire inspection rover has identified the challenges faced by firefighters and the need for improved technology in fire inspection and rescue operations. The chapter emphasizes the importance of addressing limitations in current equipment and strategies and highlights the potential benefits of incorporating advanced technologies. The

identified requirements, such as robust communication systems, real-time sensor data, and intelligent decision-making algorithms, provide a foundation for the development of an effective fire inspection rover.

3. Chapter 3: Background

3.1 Embedded System

An embedded system is a combination of hardware and software designed for a specific purpose or task within a larger system. It is typically a computer system that is embedded and integrated into a larger device or system, rather than being a standalone computer. Embedded systems are often used in applications requiring real-time computing, control and monitoring, and specialized functionality. They can be found in a wide range of devices, such as automobiles, appliances, medical devices, industrial machines, and consumer electronics.

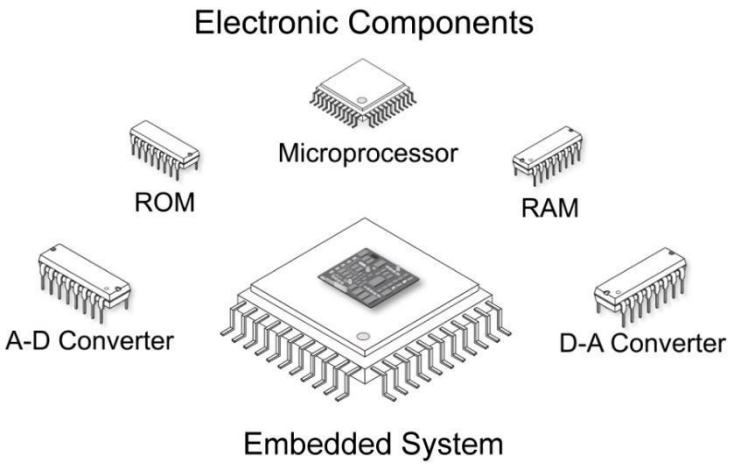

Figure 9:embeddedSystem

- Hardware: The hardware component of an embedded system consists of microprocessors or microcontrollers, memory (ROM and RAM), input/output (I/O) devices, sensors, actuators, and other peripherals. The

hardware is designed to be compact, power-efficient, and tailored for the specific application.

- Software: Embedded systems use a combination of software components, including firmware and operating systems. The firmware is typically stored in non-volatile memory (ROM) and provides the necessary instructions for the system to operate. The operating system (OS) manages the resources, supports communication between hardware and software components, and provides a platform for application development.
- Real-time operation: Many embedded systems require real-time operation where tasks must be completed within strict timing constraints. This is commonly seen in systems that control machinery, monitor critical parameters, or process time-sensitive data. Real-time operating systems (RTOS) are often used to ensure deterministic behavior and meet timing requirements.
- Connectivity: Some embedded systems are designed to be connected to other devices or networks. This enables them to exchange data, receive updates, or be remotely controlled. Connectivity options can include wired (UART, I2C, SPI, USB) or wireless communication protocols.
- Power constraints: Embedded systems are typically designed to operate on limited power sources, such as batteries or low-power supplies. Power management techniques, such as sleep modes, power gating, and dynamic voltage scaling, are used to optimize power consumption and extend battery life.
- Specialized functionality: Embedded systems are often designed to perform specific functions or tasks. For example, in automotive applications, embedded systems control engine management, anti-lock braking systems, and airbag deployment. In medical devices, embedded

systems are used for monitoring patient vital signs or delivering precise doses of medication.

- Development process: Developing embedded systems involves a combination of hardware and software design. It includes activities such as system modeling, hardware circuit design, firmware development, software programming, integration testing, and validation. Specialized development tools, programming languages, and debugging techniques are used to facilitate the development process.
- Overall, embedded systems play a critical role in various industries, enabling efficient control, monitoring, and automation in a wide range of applications.

Layered Architecture in Embedded Systems

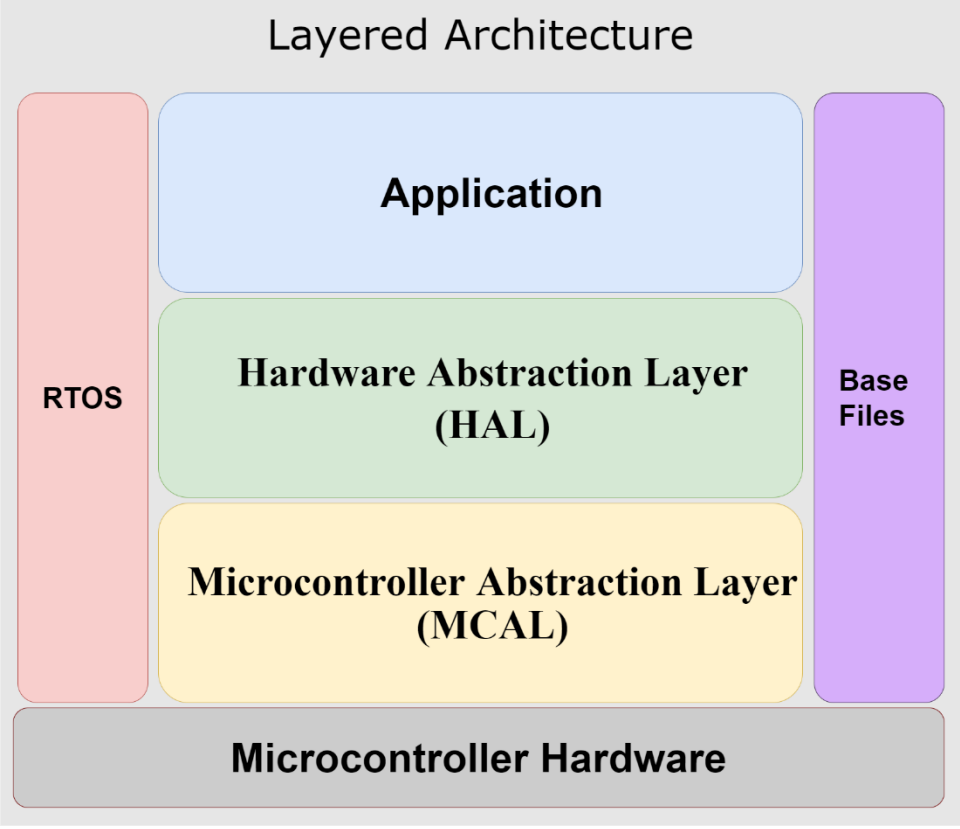

Figure 10:layered Arch

Layered architecture is a commonly used design pattern in embedded systems,

providing a structured and modular approach to system development. It involves dividing the embedded system into distinct layers, each responsible for specific functionalities and interactions. Here are the typical layers found in embedded systems:

I. Application Layer: The topmost layer in the architecture is the application layer, which contains the user-facing functionality of the embedded system. It includes the application software and user interfaces that enable interaction and control of the system. This layer is responsible for processing user input, generating output, and coordinating with other layers to fulfill system requirements.

II. Hardware Abstraction Layer (HAL): The hardware abstraction layer (HAL) is responsible for providing a uniform interface to the underlying hardware components. It abstracts the specific details of different hardware platforms, allowing the rest of the system to remain independent of hardware variations. The HAL provides functions and APIs that enable the software layers to interact with the hardware accurately and efficiently.

III. Microcontroller Abstraction Layer (MCAL): The Microcontroller Abstraction Layer (MCAL) is a specific layer that sits between the HAL layer and the hardware layer. The MCAL layer provides a standardized interface and abstraction for the microcontroller's hardware peripherals. The main purpose of the MCAL layer is to abstract the low-level details of the microcontroller peripherals, such as GPIO (General Purpose Input/Output), timers, ADC (Analog-to-Digital Converter), UART (Universal Asynchronous Receiver-Transmitter), and I2C. It provides a consistent and uniform set of APIs (Application Programming Interfaces) that the higher-level software layers can use to interact with the microcontroller hardware, regardless of the specific microcontroller

model or vendor.

IV. Hardware Layer: This is the lowest layer of the architecture and represents the actual physical hardware components of the embedded system. It includes the microcontrollers, processors, memory and other electronics that form the foundation of the system. The hardware layer provides the necessary computational power, data storage, and I/O capabilities required by the upper layers to perform their tasks.

V. Operating System Layer (RTOS): The operating system layer, also known as the kernel layer, handles the management and coordination of hardware resources. It provides services such as task scheduling, memory management, device drivers, and input/output (I/O) operations. The operating system layer acts as an intermediary between the hardware and the application layer, ensuring proper allocation and utilization of system resources.

The importance of the layered architecture in the embedded systems

The layered architecture in embedded systems is crucial for several reasons:

i. Modularity and Reusability: The layered architecture promotes modularity by dividing the system into distinct layers, with each layer responsible for specific functionalities. This modular structure allows for easy modification, replacement, and reusability of components. Developers can update or replace individual layers without affecting the entire system, enhancing flexibility and maintainability.

ii. Abstraction and Independence: Each layer in the architecture provides a well-defined interface and abstraction to the layer above it. This abstraction shields upper layers from the complexities of lower layers, allowing them to interact with the system in a standardized and consistent way. The independence provided by layered architecture enables

developers to design, develop, and test each layer separately, facilitating efficient collaboration and development of complex systems.

iii. Scalability and Performance Optimization: The layered architecture supports scalability, allowing the system to evolve and adapt to changing requirements. New functionalities or features can be added by inserting new layers or modifying existing ones without affecting the overall system. Additionally, layering enables performance optimization by allowing developers to focus on specific areas of the system, applying targeted optimizations to improve efficiency and response time.

iv. System Stability and Fault Isolation: The layered architecture helps isolate faults and enhances system stability. If a problem occurs in one layer, it can be contained within that layer without affecting the other layers. This isolation prevents cascading failures and facilitates easier debugging and troubleshooting. The layered structure also improves fault tolerance as individual components or layers can be replaced or modified independently.

v. Interoperability and Portability: The layered architecture promotes interoperability and portability by providing standard interfaces between layers. This enables components from different vendors or sources to work together seamlessly if they adhere to the agreed-upon interfaces. The standardized interfaces also facilitate the porting and migration of embedded systems across different platforms or microcontroller architectures.

vi. Development and Maintenance Efficiency: The layered architecture simplifies the development and maintenance processes. Each layer can be developed, tested, and debugged independently, shortening development cycles and reducing dependencies. Additionally, layered architectures allow for better code reuse, as components can be shared or used across different projects or systems, reducing development time and effort.

vii. Future Proofing and Technology Updates: The layered architecture enables future proofing and accommodating technology updates. By keeping each layer separate, it becomes easier to swap out obsolete or outdated components or add support for new technologies. This flexibility allows embedded systems to adapt to advancements in hardware, software, and communication protocols over time.

Overall, the layered architecture in embedded systems provides a structured and efficient approach to system design, development, and maintenance. It promotes modularity, scalability, fault isolation, performance optimization, and facilitates interoperability and portability. These benefits contribute to the stability, versatility, and longevity of embedded systems across a wide range of applications.

RTOS

RTOS, short for Real-Time Operating System, is a specialized operating system designed specifically for embedded systems that require deterministic and time-critical response. Unlike general-purpose operating systems like Windows or Linux, RTOS is optimized for real-time tasks and provides features to ensure timely and predictable execution of processes in the system.

The characteristics of an RTOS:

I. Determinism: An RTOS guarantees deterministic behavior, meaning that tasks and processes are executed within specific time constraints. It ensures that tasks with higher priority or time-critical requirements are given priority and executed promptly.
II. Task Scheduling: An RTOS employs a scheduler that determines the order and timing of task execution. The scheduler is designed to efficiently allocate CPU resources and manage task priorities, allowing

for smooth and predictable execution of tasks.

III. Task Management: RTOS provides mechanisms for task creation, suspension, and termination. It manages the execution of multiple tasks concurrently, ensuring that system resources are allocated appropriately and that each task is given sufficient processing time.

IV. Interrupt Handling: Interrupt handling is crucial in real-time systems, as hardware events and time-sensitive inputs need to be responded to promptly. RTOS provides mechanisms to handle interrupts efficiently, allowing for quick context switches to handle time-critical events.

V. Resource Management: An RTOS manages system resources such as memory, peripherals, and I/O devices. It provides mechanisms for resource sharing, allocation, and synchronization, ensuring that tasks have the necessary resources to execute and communicate without conflicts.

VI. Timing Services: RTOS often provides timing services and tools to measure and control the timing requirements of tasks. This can include features such as timers, periodic scheduling, and precise timing functions, allowing developers to enforce strict time constraints on critical operations.

VII. Communication and Synchronization: RTOS provides communication mechanisms, such as message queues, semaphores, and event flags, to facilitate inter-task communication and synchronization. These mechanisms enable tasks to exchange data, coordinate actions, and ensure effective collaboration within the system.

VIII. Small Footprint: Embedded systems often have limited resources, including memory and processing power. Therefore, RTOSs are designed to have a small footprint, optimizing resource usage and minimizing overhead.

RTOSs are commonly used in applications that require real-time behavior, such as industrial automation, robotics, automotive systems, medical devices, and aerospace systems. They enable precise control, predictable timing, and reliable operation, ensuring that critical tasks and processes meet their specific timing requirements.

3.2 ROS

Starting as a personal project and ending up being one of the basic tools used in each robotic company.

ROS, which stands for Robot Operating System, is an open-source framework that provides a collection of software libraries and tools to help developers build robot applications. It provides a flexible and distributed architecture for creating robotic systems, allowing for modular development, communication between different components, and reusability of code.

To understand ROS fully a set of basic concepts must be identified:

1. Architecture: ROS follows a distributed architecture, where the system is composed of multiple nodes that communicate with each other. Nodes are independent software modules that perform specific tasks or functions within the robot system. Communication between nodes is achieved through ROS messages, which are structured data structures that are passed between nodes over topics.
2. Communication: ROS uses a publish-subscribe messaging model for inter-node communication. Nodes can publish messages on a specific topic based on developer needs, and other nodes that are interested in that topic can subscribe to receive those messages. This decoupled communication enables loose coupling between components, making it easier to develop and maintain complex robotic systems.

3. Package System: ROS uses a package system to organize and distribute software components. A package is a collection of related nodes, libraries, configuration files, and resources. The package system allows for easy sharing and reuse of code across different projects.
4. Tools and Libraries: ROS provides a wide range of tools and libraries to aid in development, simulation, visualization, and debugging of robot applications. Some notable tools include the roscore (the ROS master), relaunch (for launching multiple nodes and configurations), rviz (a 3D visualization tool), and rosbag (for recording and playing back ROS messages).
5. Language Support: ROS supports multiple programming languages, including C++, Python, and more recently, JavaScript and Julia. This allows developers to choose the language that best suits their preferences and expertise and is best suited to their used hardware.

From the previous we conclude that ROS provides a powerful framework for building robotic systems by abstracting common functionalities, facilitating communication between components, and fostering code reuse.

- **ROS Master Communication:**

  In the context of ROS (Robot Operating System), the ROS Master is a crucial component that facilitates communication between various nodes in a distributed ROS system. The ROS Master acts as a centralized server that helps nodes discover each other, establish connections, and exchange information.

- **How ROS Master facilitates communication:**
  - Node Registration: When a ROS node starts, it registers with the ROS Master by providing information such as its name, topics it publishes or subscribes to, and services it offers or uses.

- Topic Publishing: Nodes that produce data publish messages on specific topics. They inform the ROS Master about the topics they publish, and the message types associated with them.
- Topic Subscribing: Nodes that require certain data subscribe to topics of interest. They inform the ROS Master about the topics they want to subscribe to and the message types they expect.
- Service Offering: Nodes can offer services to perform specific tasks. They inform the ROS Master about the services they provide, and the service types associated with them.
- Service Requesting: Nodes that need a particular task to be executed can request services from other nodes. They inform the ROS Master about the services they need and the service types they expect.
- ROS Master as a Directory: The ROS Master acts as a directory service, keeping track of which nodes are publishing or subscribing to specific topics and which nodes offer or use specific services. It maintains an updated record of the system's current state.
- Node-to-Node Communication: Nodes use the information provided by the ROS Master to establish direct communication channels with each other. They can send messages on topics or request and respond to services directly which lay down the ground framework for ROS multiagent systems.
- Message Routing: The ROS Master helps in routing messages between nodes that are publishing and subscribing to the same topic. It ensures that messages are properly delivered to the intended recipients.
- Parameter Server: The ROS Master also manages a parameter server, which is a central repository for storing and sharing configuration parameters across nodes in the ROS system. Nodes

can retrieve or update parameters from the parameter server through the ROS Master.

- It could be concluded from the above that the ROS Master plays a critical role in facilitating communication and coordination among nodes in a distributed ROS system. It provides a centralized mechanism for node registration, topic and service discovery, message routing, and parameter management, enabling effective and efficient communication between the components of a robotic system.

  ROS (Robot Operating System) supports communication across multiple machines, allowing you to distribute the computational load and resources of your robotic system. This distributed setup enables the cooperation of multiple nodes running on different machines to perform complex tasks collectively. Allowing for remotely operated devices to perform tasks and communicate data over to the master machine. Here's an overview of how ROS can be used across machines:

- Network Setup: First step is to ensure that all machines are connected to the same network, either through Ethernet or Wi-Fi. Machines should be able to communicate with each other using their IP addresses or hostnames.
- ROS Master: Selecting one machine to act as the ROS Master. This machine will serve as the centralized coordination point for all nodes in the system. By setting the ROS_MASTER_URI environment variable on all other machines to point to the ROS Master machine's IP address.
- ROS Nodes: The ROS nodes are distributed across the machines based on their functionality and computational requirements. Nodes can be responsible for various tasks such as perception,

planning, control, or monitoring.

- Node Communication: Nodes running on different machines can communicate with each other through ROS topics, services, and actions. Publishers and subscribers can be distributed across machines, allowing for the exchange of messages between different components of the system.
- Message Passing: Messages sent between nodes are serialized using the ROS message format and passed over the network. The underlying transport mechanism can be TCP/IP or UDP/IP, depending on the ROS configuration.
- Launch Files: Use launch files to start multiple nodes simultaneously across different machines. Launch files define the configuration and startup parameters for each node, making it easier to manage the distributed system.
- Parameter Server: The ROS Parameter Server can be utilized to store and share configuration parameters across machines. Nodes can retrieve or update parameters from the Parameter Server, ensuring consistent configuration across the distributed system.
- Visualization and Logging: Tools like RViz and rqt_graph can be used to visualize the system's topology and monitor the communication between nodes running on different machines. Logging and debugging mechanisms are available to aid in diagnosing issues in a distributed setup.
- ROS Tools: ROS provides various tools and utilities to facilitate distributed communication and system management, such as rostopic, rosservice, and roslaunch. These tools enable you to inspect, control, and coordinate the behavior of nodes across machines.
- By leveraging the distributed capabilities of ROS, you can design

and implement complex robotic systems that take advantage of the computational power and resources available across multiple machines. Distributing nodes allows for scalability, fault tolerance, and efficient utilization of hardware resources, enabling you to build more capable and robust robotic applications.

3.3 Unity

Why uses a simulator?

The utilization of a simulator in our fire inspection rover project provides a multitude of advantages. Firstly, simulators offer us the capability to simulate and evaluate different control algorithms, allowing us to assess their effectiveness in various scenarios. This enables us to fine-tune the rover's control system and optimize its performance before deploying it in real-world situations. Secondly, by employing a simulator, we can experiment with and test a wide range of fire-fighting techniques. This facilitates the exploration of different strategies and the evaluation of their integration with firefighters, ensuring seamless coordination and synergy between the rover and human responders. Moreover, simulators enable us to replicate diverse environments and simulate different fire situations, empowering us to train the rover and its pilot in a comprehensive and controlled manner. This training prepares them for various challenges they may encounter during actual firefighting operations. By leveraging simulators, we can enhance the capabilities and efficiency of our fire inspection rover, ultimately contributing to more effective fire prevention and response efforts.

Why use unity?

Using the Unity game engine for our fire inspection rover project offers

numerous advantages specifically tailored to this task. Unity provides a powerful and versatile platform that enables us to create realistic and interactive simulations, making it an ideal choice for simulating fire-related scenarios.

Firstly, Unity's robust physics engine allows us to accurately simulate the movement and behavior of the rover in response to different environmental factors. This includes modeling realistic fire dynamics, smoke propagation, and the interaction between the rover and fire elements. By incorporating these realistic simulations, we can thoroughly test the rover's capabilities and optimize its performance in challenging firefighting scenarios.

The versatility, realism, and accessibility provided by Unity make it an ideal choice for our fire inspection rover project. By leveraging Unity's capabilities, we can create highly realistic and immersive simulations, test and optimize the rover's performance, and enhance training for the pilot and firefighters involved in fire inspection and response operations.

3.4 SLAM

What is SLAM?

- SLAM (Simultaneous Localization and Mapping) is a fundamental problem in the field of robotics and computer vision. It involves the task of creating a map of an unknown environment while simultaneously determining the robot's location within that environment. SLAM has gained significant attention due to its wide range of applications in various domains, including autonomous navigation, augmented reality, and surveillance systems.
- In autonomous navigation, SLAM enables robots or vehicles to explore and navigate unknown environments with limited or no prior information.

By using sensor data, such as camera images, LIDAR scans, or range measurements, SLAM algorithms can construct a map of the environment and estimate the robot's position relative to the map. This capability is crucial for autonomous vehicles, drones, and robots operating in complex, dynamic environments.

- Overall, SLAM represents a critical technology for enabling robots and autonomous systems to operate in unknown environments, providing accurate localization and mapping capabilities. Its applications span across diverse domains, from autonomous navigation to augmented reality and surveillance systems, making it an active area of research and development with significant implications for various industries.

Different SLAM algorithms

There are several types of SLAM algorithms, each with its own strengths and limitations. The two main categories of SLAM algorithms are filter-based methods (such as Extended Kalman Filter, EKF) and optimization-based methods (such as Graph-based SLAM and Bundle Adjustment).

Filter-based methods, like EKF-SLAM, are computationally efficient and suitable for real-time applications. They use an estimation filter to maintain a belief of the robot's pose and the map. However, these methods rely on linearization assumptions and can suffer from linearization errors, leading to suboptimal performance in highly nonlinear environments. They are also sensitive to data association problems, where incorrect associations between measurements and landmarks can lead to map inconsistencies. Another class of SLAM algorithms is known as Visual SLAM, which primarily relies on visual input from cameras. Visual SLAM methods, such as ORB-SLAM and LSD-SLAM, offer advantages in terms of scalability, low-cost sensors, and portability. They can operate in various environments, including

indoor and outdoor scenarios. However, they can be sensitive to lighting conditions, feature ambiguities, and occlusions, which can affect their accuracy and robustness.

SLAM vs VSLAM

Filter-based methods are efficient but may suffer from linearization errors and data association issues. Optimization-based methods provide global consistency but are computationally demanding. Visual SLAM methods offer scalability and cost-effectiveness but may be sensitive to environmental factors. The choice of SLAM algorithm depends on the specific requirements of the application, considering factors such as computational resources, real-time performance, and environmental conditions

SLAM devices.

SLAM devices, also known as SLAM systems or SLAM sensors, are hardware devices specifically designed to facilitate Simultaneous Localization and Mapping (SLAM) tasks. These devices integrate various sensors and technologies to capture and process data, enabling the creation of maps and simultaneous estimation of the device's position within the environment.

SLAM devices typically incorporate a combination of the following sensors:

Cameras: Visual cameras capture images or video streams, which are used for extracting visual features, tracking landmarks, and estimating the device's motion and position.
Stereo camera, IR camera, Kinect.

LIDAR (Light Detection and Ranging): LIDAR sensors emit laser beams and measure the time it takes for the laser pulses to return after reflecting off objects in the environment. This data is used to create 3D point clouds, which are

valuable for mapping and obstacle detection.

Inertial Measurement Units (IMUs): IMUs consist of accelerometers and gyroscopes that measure acceleration and angular velocity, respectively. By integrating these measurements over time, IMUs provide information about the device's motion, orientation, and trajectory.

Ultrasonic Sensors: Ultrasonic sensors emit high-frequency sound waves and measure the time it takes for the waves to reflect back after hitting objects. These sensors are useful for short-range obstacle detection and collision avoidance.

GPS (Global Positioning System): GPS receivers provide global position information based on signals received from satellites. While GPS alone is not sufficient for accurate SLAM, it can be used in combination with other sensors for coarse localization and to improve the overall positioning accuracy.

4. Chapter 4 : Design

The system design chapter of the Fire Inspection Rover focuses on the architecture, components, and integration of an embedded system using STM32 microcontrollers, ROS (Robot Operating System), and SLAM (Simultaneous Localization and Mapping) techniques. This chapter outlines the design principles and considerations involved in creating a robust and efficient system for fire inspection and rescue operations.

The chapter begins by outlining the system's overall architecture, including the hardware and software components as shown in figure.

The STM32 microcontrollers serve as the central processing units, responsible for controlling the various subsystems and executing the embedded software. The choice of STM32 is based on its reliability, real-time capabilities, and low

power consumption, making it suitable for deployment in demanding scenarios.

ROS is introduced as the underlying framework for communication, control, and coordination within the system. It enables modular development and seamless integration of different software modules, allowing for flexibility and scalability. The chapter explains the use of ROS nodes, topics, and messages for inter-component communication, as well as the benefits of leveraging the extensive ROS ecosystem and community support.

SLAM techniques play a crucial role in enabling the Fire Inspection Rover to map its surroundings and navigate autonomously. The chapter discusses the integration of SLAM algorithms, such as visual SLAM, to build accurate maps of the fire scene and localize the robot within the environment. This enables the robot to navigate efficiently, avoid obstacles, and provide firefighters with real-time situational awareness.

The chapter also addresses the integration of various sensors, such as cameras, Kinect, thermal sensors, and gas detectors, to gather relevant data about the fire scene and environmental conditions. These sensors provide critical information for decision-making, hazard detection, and victim localization. The design considerations for sensor selection, placement, and data fusion are discussed in detail.

Overall System Diagram:

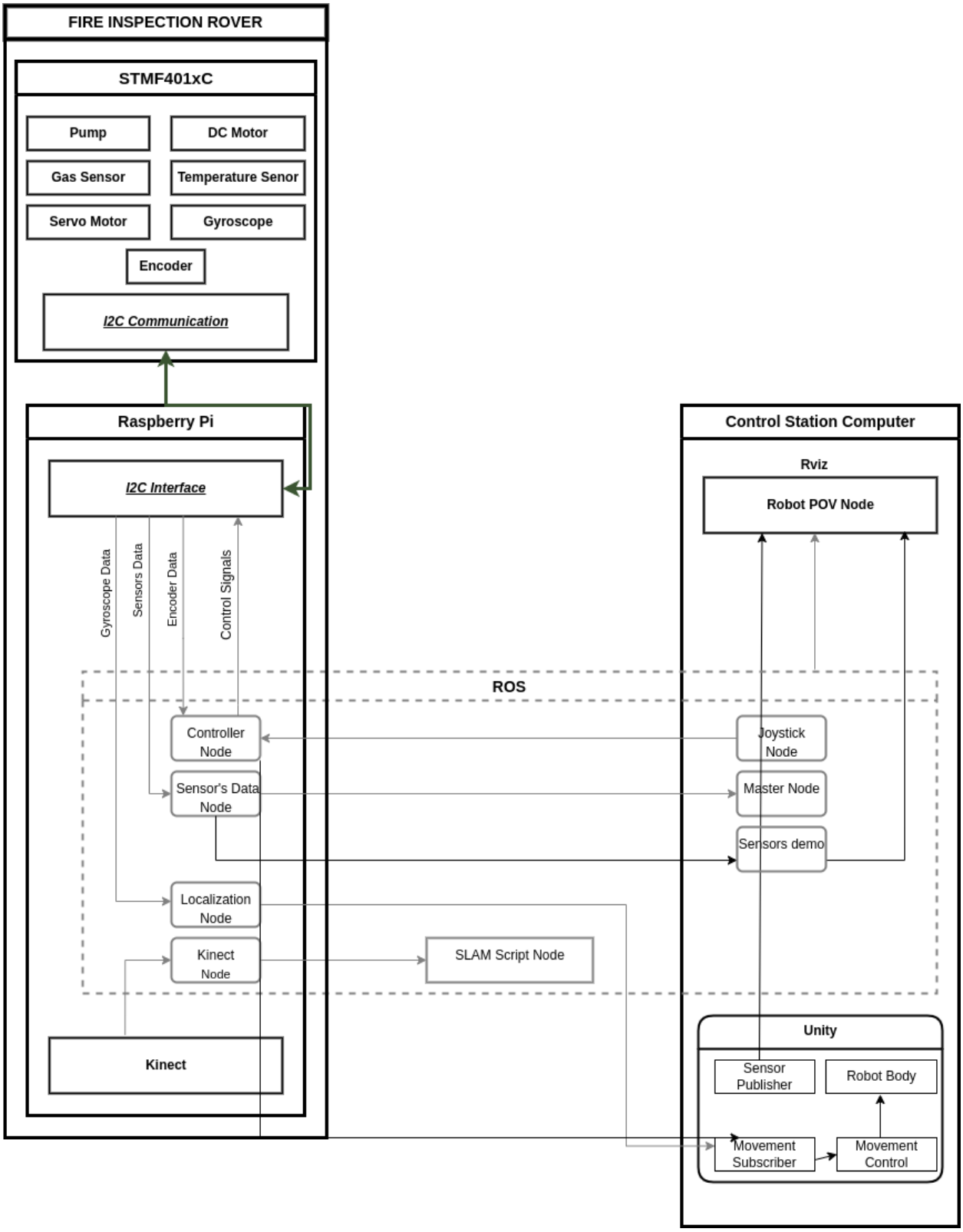

Figure 11:system Diagram

4.1 Embedded System

The layered architecture in embedded systems

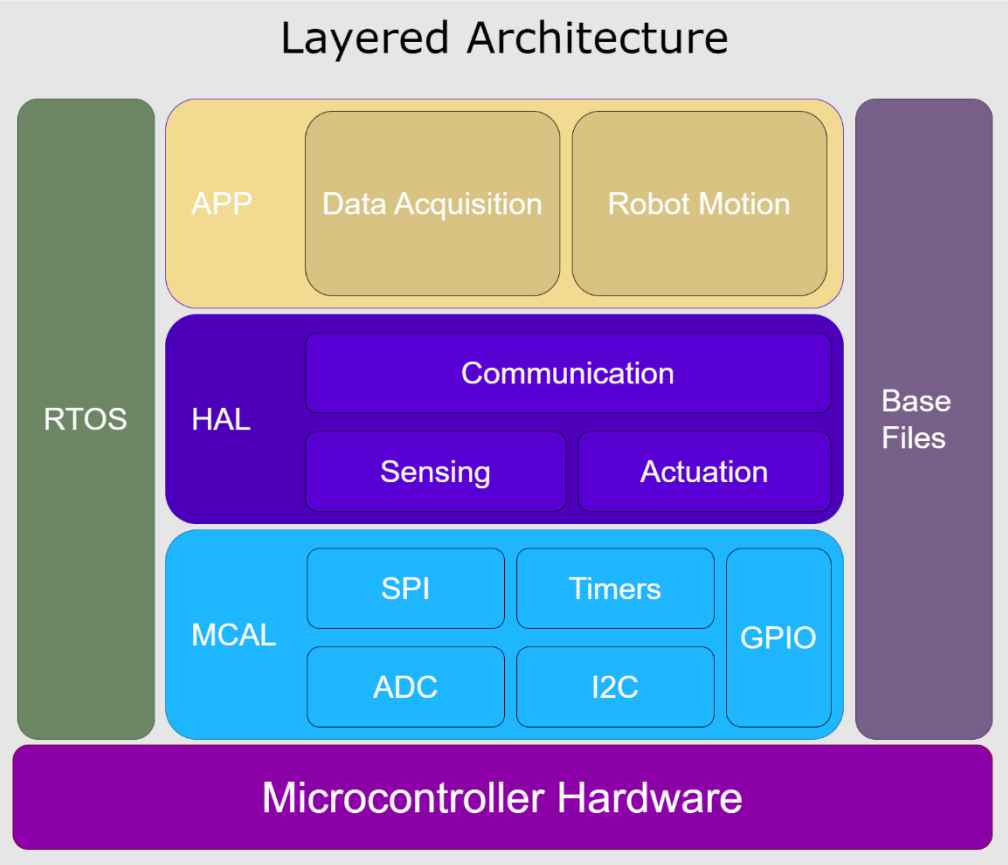

Figure 12:our Layered Arch

Layered architecture is a commonly used design pattern in embedded systems, providing a structured and modular approach to system development. It involves dividing the embedded system into distinct layers, each responsible for specific functionalities and interactions. Here are the typical layers found in embedded systems:

- Application Layer: The topmost layer in the architecture is the application layer, which contains the user-facing functionality of the embedded system. It includes the application software and user interfaces that enable interaction and control of the system. This layer is responsible for

processing user input, generating output, and coordinating with other layers to fulfill system requirements. In our system, it contains two modules, one for collecting the required data from the sensors using the Sensing module in the HAL layer and transmitting it to the Raspberry Pi through the Communication module in the HAL layer. The other module is implemented for receiving the required direction and speed from the Raspberry Pi through the Communication module in the HAL layer to actuate the motors responsible for the motion of the Robot using the Actuation module in the HAL layer and transmitting back the feedback of motors to the Raspberry Pi.

- Hardware Abstraction Layer (HAL): The hardware abstraction layer (HAL) is responsible for providing a uniform interface to the underlying hardware components. It abstracts the specific details of different hardware platforms, allowing the rest of the system to remain independent of hardware variations. The HAL provides functions and APIs that enable the software layers to interact with the hardware accurately and efficiently. There are three main modules in this layer. These modules are using the needed peripherals in the microcontroller through the functions and the APIs in the MCAL layer for a successful interface with the different sensors, actuators in the systems and for achieving the connectivity with the Raspberry Pi.
- Microcontroller Abstraction Layer (MCAL): The Microcontroller Abstraction Layer (MCAL) is a specific layer that sits between the HAL layer and the hardware layer. The MCAL layer provides a standardized interface and abstraction for the microcontroller's hardware peripherals. The main purpose of the MCAL layer is to abstract the low-level details of the microcontroller peripherals, such as GPIO (General Purpose Input/Output), timers, ADC (Analog-to-Digital Converter), UART (Universal Asynchronous Receiver-Transmitter), and I2C. It provides a

consistent and uniform set of APIs (Application Programming Interfaces) that the higher-level software layers can use to interact with the microcontroller hardware, regardless of the specific microcontroller model or vendor.

- Hardware Layer: This is the lowest layer of the architecture and represents the actual physical hardware components of the embedded system. It includes the microcontrollers, processors, memory, and other electronics that form the foundation of the system. The hardware layer provides the necessary computational power, data storage, and I/O capabilities required by the upper layers to perform their tasks.
- Operating System Layer (RTOS): The operating system layer, also known as the kernel layer, handles the management and coordination of hardware resources. It provides services such as task scheduling, memory management, device drivers, and input/output (I/O) operations. The operating system layer acts as an intermediary between the hardware and the application layer, ensuring proper allocation and utilization of system resources.

The importance of the layered architecture in the embedded systems

The layered architecture in embedded systems is crucial for several reasons:

I. Modularity and Reusability: The layered architecture promotes modularity by dividing the system into distinct layers, with each layer responsible for specific functionalities. This modular structure allows for easy modification, replacement, and reusability of components. Developers can update or replace individual layers without affecting the entire system, enhancing flexibility and maintainability.

II. Abstraction and Independence: Each layer in the architecture provides a well-defined interface and abstraction to the layer above it. This

abstraction shields upper layers from the complexities of lower layers, allowing them to interact with the system in a standardized and consistent way. The independence provided by layered architecture enables developers to design, develop, and test each layer separately, facilitating efficient collaboration and development of complex systems.

III. Scalability and Performance Optimization: The layered architecture supports scalability, allowing the system to evolve and adapt to changing requirements. New functionalities or features can be added by inserting new layers or modifying existing ones without affecting the overall system. Additionally, layering enables performance optimization by allowing developers to focus on specific areas of the system, applying targeted optimizations to improve efficiency and response time.

IV. System Stability and Fault Isolation: The layered architecture helps isolate faults and enhances system stability. If a problem occurs in one layer, it can be contained within that layer without affecting the other layers. This isolation prevents cascading failures and facilitates easier debugging and troubleshooting. The layered structure also improves fault tolerance as individual components or layers can be replaced or modified independently.

V. Interoperability and Portability: The layered architecture promotes interoperability and portability by providing standard interfaces between layers. This enables components from different vendors or sources to work together seamlessly if they adhere to the agreed-upon interfaces. The standardized interfaces also facilitate the porting and migration of embedded systems across different platforms or microcontroller architectures.

VI. Development and Maintenance Efficiency: The layered architecture simplifies the development and maintenance processes. Each layer can be developed, tested, and debugged independently, shortening development

cycles, and reducing dependencies. Additionally, layered architectures allow for better code reuse, as components can be shared or used across different projects or systems, reducing development time and effort.

VII. Future Proofing and Technology Updates: The layered architecture enables future proofing and accommodating technology updates. By keeping each layer separate, it becomes easier to swap out obsolete or outdated components or add support for new technologies. This flexibility allows embedded systems to adapt to advancements in hardware, software, and communication protocols over time.

Overall, the layered architecture in embedded systems provides a structured and efficient approach to system design, development, and maintenance. It promotes modularity, scalability, fault isolation, performance optimization, and facilitates interoperability and portability. These benefits contribute to the stability, versatility, and longevity of embedded systems across a wide range of applications.

HAL Layer:

Sensing Module:

I. MQ-7 Gas Sensor

The MQ-7 is a commonly used gas sensor for detecting carbon monoxide (CO) in the air. It is suitable for applications such as gas leakage detection, industrial safety, and environmental monitoring.

Features:

- High sensitivity to carbon monoxide gas.
- Stable and long-term operation.
- Fast response and recovery time.

- Analog output signal that can be easily interfaced with microcontrollers or another circuitry.
- Low power consumption.

Specifications:

- Sensing Gas: Carbon Monoxide (CO)
- Detection Range: 10 to 500 ppm CO
- Operating Voltage: 5V DC
- Heater Resistance: 31 Ω ± 3 Ω at room temperature
- Load Resistance: Adjustable (typically 10 kΩ)
- Dimensions: Varies depending on the manufacturer and package (e.g., 32mm x 20mm x 22mm)

Pinout:

The MQ-7 sensor typically has 4 or 6 pins. The pinout may vary depending on the manufacturer, but here is a common configuration:

- VCC: Connect to the positive terminal of the power supply (5V).

Figure 13:MQ7

- GND: Connect to the ground of the power supply.
- AOUT: Analog output that provides a voltage proportional to the detected CO gas concentration.

- DOUT: Digital output that can be used to trigger an alarm or interface with a microcontroller.
- (Optional) HOUT: Heating voltage output for driving an external heater.

Interfacing:

To use the MQ-7 gas sensor, you can follow these steps:

- Connect the VCC pin to a 5V power supply and the GND pin to ground.
- Connect the AOUT pin to an analog input pin of a microcontroller or an ADC (Analog-to-Digital Converter) module.
- Optionally, connect the DOUT pin to a digital input pin of the microcontroller if you want to use the digital output for threshold-based detection.
- Provide a suitable load resistance (e.g., 10 kΩ) between the heater pins (HOUT) to regulate the heater voltage.

Note:

The sensor requires a warm-up time (usually a few minutes) before accurate readings can be obtained.

Calibration may be necessary to ensure accurate gas concentration measurements.

Please note that this is a general overview of the MQ-7 gas sensor. For detailed specifications, circuit diagrams, and specific usage instructions, it's recommended to refer to the datasheet or documentation provided by the manufacturer of the sensor you are using, as different manufacturers may have slightly different specifications and pinouts.

APIs:

- Function Name: SEN_vidInit

Description:

This function initializes a global variable GLOB_u8GasPercentage to zero and initializes a local structure variable LOC_strADCConfigMQ7 with the configuration values for an ADC (Analog-to-Digital Converter). The ADC configuration values are likely used for configuring the ADC to read data from a gas sensor.

Syntax: void SEN_vidInit(void);

Input: This function does not take any input parameters.

Output: This function does not return any value.

Example Usage:

SEN_vidInit(); // Initializes the global variable for gas percentage to zero and the ADC configuration values for the gas sensor.

Pre-requisites:

The SEN_vidInit function should be called after the gas sensor is properly connected, and any necessary libraries or drivers have been initialized.

Post-requisites:

After the SEN_vidInit function is called, the GLOB_u8GasPercentage variable will be set to zero, and the LOC_strADCConfigMQ7 structure variable will be initialized with the ADC configuration values for reading data from the gas sensor.

- Function Name: SEN_vidUpdateSensorsData

Description:

This function updates the value of the global variable GLOB_u8GasPercentage by reading data from an MQ-7 gas sensor. It first initializes and configures an ADC for the gas sensor using the configuration values stored in the local structure variable LOC_strADCConfigMQ7. It then reads the analog value from the gas sensor using the ADC, and converts it to a gas percentage value, which is stored in the GLOB_u8GasPercentage variable.

Syntax: void SEN_vidUpdateSensorsData(void);

Input: This function does not take any input parameters.

Output: This function does not return any value.

Example Usage:

SEN_vidUpdateSensorsData(); // Updates the gas percentage value based on readings from the MQ-7 gas sensor

Pre-requisites:

The SEN_vidUpdateSensorsData function should be called after the gas sensor is properly connected and initialized, and any necessary libraries or drivers have been initialized.

Post-requisites:

After the SEN_vidUpdateSensorsData function is called, the GLOB_u8GasPercentage variable will be updated with the percentage of gas detected by the MQ-7 gas sensor.

Note:

the ADC_vidInit and ADC_u16GetADCValue functions are defined and declared somewhere in the code. If these functions are not defined or declared, the function will not compile. Additionally, the conversion factor of 41 used in the function to convert the analog value to a gas percentage is likely specific to the MQ-7 gas sensor and may need to be adjusted for other types of gas sensors.

- Function Name: SEN_u8GetGasPercentage

Description:

This function returns the current value of the global variable GLOB_u8GasPercentage, which represents the percentage of gas detected by a gas sensor.

Syntax: u8 SEN_u8GetGasPercentage(void);

Input: This function does not take any input parameters.

Output:

This function returns an 8-bit unsigned integer value representing the percentage of gas detected by the gas sensor.

Example Usage:

u8 gasPercentage = SEN_u8GetGasPercentage(); // Gets the current percentage of gas detected by the gas sensor

Pre-requisites:

The SEN_u8GetGasPercentage function should be called after the gas sensor has been initialized and SEN_vidUpdateSensorsData function has been called at least once to update the gas percentage value.

Post-requisites:

After the SEN_u8GetGasPercentage function is called, the current percentage of gas detected by the gas sensor, stored in the GLOB_u8GasPercentage variable, is returned.

Note:

It is important to note that the gas percentage value returned by this function may not be accurate if SEN_vidUpdateSensorsData has not been called recently to update the gas percentage value.

- **LM35 temperature sensor**

The LM35 is a precision integrated-circuit temperature sensor that outputs a linearly proportional voltage to the Celsius temperature. It is suitable for applications such as Thermometers, Temperature controllers, HVAC systems, Industrial process control, Environmental monitoring.

Features:

- Low cost
- Small size
- Wide temperature range
- High accuracy
- Low self-heating
- Linear output

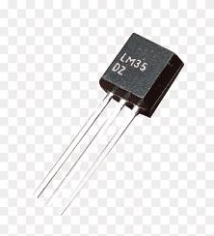

Figure 14:LM Temp sensor

Specifications:

- Type: Integrated circuit temperature sensor
- Operating temperature range: -55°C to 150°C
- Accuracy: ±0.5°C at room temperature, ±1°C over full range
- Output voltage: 10 mV/°C
- Power consumption: 60 µA
- Package: TO-92, TO-220, SOIC

Pinout:

The LM35 has three pins:

- Vout: The output voltage, which is linearly proportional to the Celsius temperature.
- GND: The ground pin.
- Vs: The supply voltage, which can be between 4 and 30 V.

Interfacing:

The LM35 can be easily interfaced with microcontrollers and other electronic devices. The output voltage of the sensor can be directly measured with a multimeter, or it can be converted to a digital signal using an ADC.

APIs:

- Function: SEN_vidInit

  Description:

This function initializes a global variable LOC_u16TemperatureDegree to zero and initializes a local structure variable LOC_strADCConfigLM35 with the configuration values for an ADC (Analog-to-Digital Converter). The ADC

configuration values are likely used for configuring the ADC to read data from a temperature sensor.

Syntax: void SEN_vidInit(void);

Input: This function does not take any input parameters.

Output: This function does not return any value.

Example Usage:

SEN_vidInit(); // Initializes the global variable for temperature and the ADC configuration values for the temperature sensor

Pre-requisites:

The SEN_vidInit function should be called after the temperature sensor is properly connected, and any necessary libraries or drivers have been initialized.

Post-requisites:

After the SEN_vidInit function is called, the LOC_u16TemperatureDegree variable will be set to zero, and the LOC_strADCConfigLM35 structure variable will be initialized with the ADC configuration values for reading data from the temperature sensor.

- Function: SEN_vidUpdateSensorsData

Description:

This function updates the value of the global variable LOC_u16TemperatureDegree by reading data from an LM35 temperature sensor. It first initializes and configures an ADC for the temperature sensor using the configuration values stored in the local structure variable LOC_strADCConfigLM35. It then reads the analog value from the temperature

sensor using the ADC and stores it in the LOC_u16TemperatureDegree variable.

Syntax: void SEN_vidUpdateSensorsData(void);

Input: This function does not take any input parameters.

Output: This function does not return any value.

Example Usage:

SEN_vidUpdateSensorsData(); // Updates the temperature value based on readings from the LM35 temperature sensor

Pre-requisites:

The SEN_vidUpdateSensorsData function should be called after the temperature sensor is properly connected and initialized, and any necessary libraries or drivers have been initialized.

Post-requisites:

After the SEN_vidUpdateSensorsData function is called, the LOC_u16TemperatureDegree variable will be updated with the current temperature reading from the LM35 temperature sensor.

Note:

the ADC_vidInit and ADC_u16GetADCValue functions are defined and declared somewhere in the code. If these functions are not defined or declared, the function will not compile.

- Function: SEN_u16GetTemperature

  Description:

This function returns the current value of the global variable LOC_u16TemperatureDegree, which represents the temperature reading from an LM35 temperature sensor.

Syntax: u16 SEN_u16GetTemperature(void);

Input: This function does not take any input parameters.

Output:

This function returns a 16-bit unsigned integer value representing the temperature reading from the LM35 temperature sensor.

Example Usage:

u16 temperature = SEN_u16GetTemperature(); // Gets the current temperature reading from the LM35 temperature sensor

Pre-requisites:

The SEN_u16GetTemperature function should be called after the temperature sensor has been initialized and SEN_vidUpdateSensorsData function has been called at least once to update the temperature value.

Post-requisites:

After the SEN_u16GetTemperature function is called, the current temperature reading from the LM35 temperature sensor, stored in the LOC_u16TemperatureDegree variable, is returned.

Note:

It is important to note that the temperature value returned by this function may not be accurate if SEN_vidUpdateSensorsData has not been called recently to update the temperature value.

- **Encoder**

An encoder is a sensor that measures the rotation angle or linear displacement of a shaft or other rotating or reciprocating object. Encoders are used in a wide variety of applications, including robotics, machine tools, and industrial automation.

Features:

- Resolution: The resolution of an encoder is the smallest change in position that the encoder can detect. The resolution of an encoder is typically measured in counts per revolution (CPR).
- Accuracy: The accuracy of an encoder is the maximum error that the encoder can make. The accuracy of an encoder is typically measured in degrees or millimeters.
- Repeatability: The repeatability of an encoder is the ability of the encoder to return to the same position after being moved. The repeatability of an encoder is typically measured in degrees or millimeters.
- Sensitivity: The sensitivity of an encoder is the ability of the encoder to detect changes in position. The sensitivity of an encoder is typically measured in counts per degree or counts per millimeter.
- Bandwidth: The bandwidth of an encoder is the frequency at which the encoder can measure position changes. The bandwidth of an encoder is typically measured in hertz.

Figure 15:encoder

Pinout:

encoder has four pins:

- power supply pin.
- ground pin.
- The A output pin.
- The B output pin.

Interfacing:

- Connect the encoder to the controller or device.
- Configure the controller or device.
- Start the encoder.
- Read the encoder's output signals.

APIs:

- Function: SEN_vidInit

  Description:

This function initializes the sensor module. It sets the initial values of the global variables LOC_u32LeftMotorRPM and LOC_u32RightMotorRPM to 0 and starts the input capture mode for two timer channels (TIM_CHANNEL1 and

TIM_CHANNEL2) using the TIM_vidICStart function.

Syntax: void SEN_vidInit(void);

Input: This function does not take any input parameters.

Output: This function does not return any value.

Example Usage: SEN_vidInit(); // Initializes the sensor module

Pre-requisites:

The necessary hardware and software components for the input capture mode should be properly configured.

Post-requisites:

After the SEN_vidInit function is called, the input capture mode will be started for TIM_CHANNEL1 and TIM_CHANNEL2, and the global variables LOC_u32LeftMotorRPM and LOC_u32RightMotorRPM will be set to 0.

Note:

This function is typically called at the beginning of the program or system initialization to set up the necessary components for the sensor module.

- Function: SEN_vidUpdateEncoders

Description:

This function reads the number of pulses captured by the left and right encoders, calculates the RPM (revolutions per minute) of the corresponding motors, and updates the global variables LOC_u32LeftMotorRPM and LOC_u32RightMotorRPM with the calculated RPM values.

Syntax: void SEN_vidUpdateEncoders(void);

Input: This function does not take any input parameters.

Output: This function does not return any value.

Example Usage: SEN_vidUpdateEncoders(); // Updates the encoder values and calculates the motor RPMs

Pre-requisites:

The necessary hardware and software components for the input capture mode should be properly configured, and the SEN_vidInit function should be called before this function.

Post-requisites:

After the SEN_vidUpdateEncoders function is called, the global variables LOC_u32LeftMotorRPM and LOC_u32RightMotorRPM will be updated with the current RPM values of the left and right motors, respectively.

Note:

The scaling factor used for the RPM calculation is based on the number of pulses captured by the encoder (531), the time interval over which the pulses were captured (10ms), and the number of encoder cycles per motor revolution (30). This scaling factor may need to be adjusted depending on the specific hardware configuration.

- Function: SEN_u8GetLeftMotorRPM

Description:

This function returns the current RPM (revolutions per minute) value of the left motor.

Syntax: u32 SEN_u8GetLeftMotorRPM(void);

Input: This function does not take any input parameters.

Output: This function returns an unsigned 32-bit integer value representing the RPM of the left motor.

Example Usage: u32 leftRPM = SEN_u8GetLeftMotorRPM(); // Gets the RPM value of the left motor

Pre-requisites:

The SEN_vidInit and SEN_vidUpdateEncoders functions should be called before this function.

Post-requisites:

This function does not modify any values.

Note:

The RPM value returned by this function is stored in the global variable LOC_u32LeftMotorRPM.

- Function: SEN_u8GetRightMotorRPM

Description:

This function returns the current RPM (revolutions per minute) value of the right motor.

Syntax: u32 SEN_u8GetRightMotorRPM(void);

Input: This function does not take any input parameters.

Output:

This function returns an unsigned 32-bit integer value representing the RPM of

the right motor.

Example Usage:

u32 rightRPM = SEN_u8GetRightMotorRPM(); // Gets the RPM value of the right motor

Pre-requisites:

The SEN_vidInit and SEN_vidUpdateEncoders functions should be called before this function.

Post-requisites:

This function does not modify any values.

Note:

The RPM value returned by this function is stored in the global variable LOC_u32RightMotorRPM.

- **MPU6050 Accelerometer and Gyroscope Sensor**

MPU6050 sensor module is complete 6-axis Motion Tracking Device. It combines 3-axis Gyroscope, 3-axis Accelerometer and Digital Motion Processor all in small package. Also, it has additional feature of on-chip Temperature sensor. It has I2C bus interface to communicate with the microcontrollers.

Specifications:

Gyroscope:

- 3-axis sensing with a full-scale range of ±250, ±500, ±1000, or ±2000 degrees per second (dps)
- Sensitivity of 131, 65.5, 32.8, or 16.4 LSBs per dps

- Output data rate (ODR) range of 8kHz to 1.25Hz

Accelerometer:

- 3-axis sensing with a full-scale range of ±2g, ±4g, ±8g, or ±16g
- Sensitivity of 16384, 8192, 4096, or 2048 LSBs per g
- ODR range of 8kHz to 1.25Hz
- Temperature Sensor:
- Operating range of -40°C to +85°C
- Sensitivity of 340 LSBs per degree Celsius
- Accuracy of ±3°C

Supply Voltage:

- Operating voltage range of 2.375V to 3.46V for the MPU-6050, and 2.375V to 5.5V for the MPU-6050A
- Communication Interface:
- I2C serial interface with a maximum clock frequency of 400kHz
- 8-bit and 16-bit register access modes

Other Features:

- Digital Motion Processor (DMP) for complex motion processing
- On-chip 16-bit ADCs for accurate analog-to-digital conversion
- Programmable digital filters for improved noise performance
- Interrupts for triggering events based on specific motion conditions
- Low-power consumption (3.9mA for full operation)

Accelerometer values in g (g force)

- Acceleration along the X axis = (Accelerometer X axis raw data/16384) g.

- Acceleration along the Y axis = (Accelerometer Y axis raw data/16384) g.
- Acceleration along the Z axis = (Accelerometer Z axis raw data/16384) g.

Gyroscope values in °/s (degree per second)

- Angular velocity along the X axis = (Gyroscope X axis raw data/131) °/s.
- Angular velocity along the Y axis = (Gyroscope Y axis raw data/131) °/s.
- Angular velocity along the Z axis = (Gyroscope Z axis raw data/131) °/s.

Pinout:

The MPU-6050 module has 8 pins:

- **INT:** Interrupt digital output pin.
- **AD0:** I2C Slave Address LSB pin. This is 0th bit in 7-bit slave address of device. If connected to VCC then it is read as logical one and slave address changes.

Figure 16:MPU 8050

- **XCL:** Auxiliary Serial Clock pin. This pin is used to connect other I2C interface enabled sensors SCL pin to MPU-6050.
- **XDA:** Auxiliary Serial Data pin. This pin is used to connect other I2C interface enabled sensors SDA pin to MPU-6050.

- **SCL:** Serial Clock pin. Connect this pin to the microcontroller's SCL pin.
- **SDA:** Serial Data pin. Connect this pin to the microcontroller's SDA pin.
- **GND:** Ground pin. Connect this pin to ground connection.
- **VCC:** Power supply pin. Connect this pin to +5V DC supply.
- MPU-6050 module has Slave address (When AD0 = 0, i.e. it is not connected to Vcc) as,
- **Slave Write address(SLA+W)**: 0xD0
- **Slave Read address(SLA+R)**: 0xD1

APIs:

- Function: SEN_vidMPU6050Init

Description:

This function initializes an MPU6050 sensor by performing the following steps:

- Check if the sensor is working properly by reading the WHO_AM_I register.
- Wakes up the sensor by writing 0x00 to the PWR_MGMT_1 register.
- Sets the data output rate (sample rate) by writing to the SMPLRT_DIV register.
- Configures the full scale range of the accelerometer and gyroscope by writing to the ACCEL_CONFIG and GYRO_CONFIG registers.

Syntax: static void SEN_vidMPU6050Init(void);

Input: This function does not take any input parameters.

Output: This function does not return any value.

Example Usage:

This function is typically called internally within the implementation of a larger system and is not usually called directly.

Pre-requisites:

The MPU6050 sensor should be properly connected to the system, and any necessary libraries or drivers should be initialized.

Post-requisites:

After the SEN_vidMPU6050Init function is called, the MPU6050 sensor will be properly initialized and ready to start providing data.

Note:

This function is called in function SEN_vidInit. In case the WHO_AM_I register does not return the expected value of MPU6050_ADDRESS_7_BITS, the function does not perform any further initialization.

- Function: SEN_vidReadAccel

Description:

This function reads the raw accelerometer data from an MPU6050 sensor and converts it to floating-point values representing acceleration in the X, Y, and Z directions. The raw data is read from the ACCEL_XOUT_H, ACCEL_XOUT_L, ACCEL_YOUT_H, ACCEL_YOUT_L, ACCEL_ZOUT_H, and ACCEL_ZOUT_L registers of the sensor. The conversion from raw data to floating-point values is performed by dividing the raw data by a scaling factor of 16384.0.

Syntax: static void SEN_vidReadAccel(void);

Input: This function does not take any input parameters.

Output: This function does not return any value.

Example Usage:

This function is typically called internally within the implementation of a larger system and is not usually called directly.

Pre-requisites:

The MPU6050 sensor should be properly connected to the system, and any necessary libraries or drivers should be initialized.

Post-requisites:

After the SEN_vidReadAccel function is called, the global variables GLOB_f32AccelX, GLOB_f32AccelY, and GLOB_f32AccelZ will be updated with the current accelerometer data in the X, Y, and Z directions, respectively.

Note:

This function is called in function SEN_vidUpdateSensorsData. The global variables GLOB_f32AccelX, GLOB_f32AccelY, and GLOB_f32AccelZ are defined and declared somewhere in the code. If these variables are not defined or declared, the function will not compile.

- Function: SEN_vidReadGyro

Description:

This function reads the raw gyroscope data from an MPU6050 sensor and converts it to floating-point values representing angular velocity in the X, Y, and Z directions. The raw data is read from the GYRO_XOUT_H, GYRO_XOUT_L, GYRO_YOUT_H, GYRO_YOUT_L, GYRO_ZOUT_H, and GYRO_ZOUT_L registers of the sensor. The conversion from raw data to

floating-point values is performed by dividing the raw data by a scaling factor of 131.0.

Syntax: static void SEN_vidReadGyro(void);

Input: This function does not take any input parameters.

Output: This function does not return any value.

Example Usage:

This function is typically called internally within the implementation of a larger system and is not usually called directly.

Pre-requisites:

The MPU6050 sensor should be properly connected to the system, and any necessary libraries or drivers should be initialized.

Post-requisites:

After the SEN_vidReadGyro function is called, the global variables GLOB_f32GyroX, GLOB_f32GyroY, and GLOB_f32GyroZ will be updated with the current gyroscope data in the X, Y, and Z directions, respectively.

Note:

This function is called in function SEN_vidUpdateSensorsData. The global variables GLOB_f32GyroX, GLOB_f32GyroY, and GLOB_f32GyroZ are defined and declared somewhere in the code. If these variables are not defined or declared, the function will not compile.

Actuation Module:

- **Motors**

Precision metal gear motors are a type of DC motor that uses precision gears to transmit power. This type of motor is often used in applications where high precision and low noise are required, such as in robotics, medical devices, and CNC machines.

Figure 17:motor

Features:

- High precision.
- Low noise.
- High efficiency.
- Long lifespan.

Specifications:

- Voltage: The voltage rating of a precision metal gear motor is the amount of voltage that the motor can safely operate at.
- Current: The current rating of a precision metal gear motor is the amount of current that the motor can safely draw.
- Speed: The speed of a precision metal gear motor is the number of revolutions per minute (RPM) that the motor can spin.
- Torque: The torque of a precision metal gear motor is the amount of force that the motor can exert.
- Efficiency: The efficiency of a precision metal gear motor is the percentage of electrical energy that is converted into mechanical energy.

- Rated load: The rated load of a precision metal gear motor is the maximum amount of torque that the motor can produce continuously without overheating.
- Insulation class: The insulation class of a precision metal gear motor is a measure of its ability to withstand heat.
- Life expectancy: The life expectancy of a precision metal gear motor is the number of hours that the motor can be expected to operate before it fails.

Pinout:

- Signal pin: This pin is used to send a signal to the motor, which tells the motor how to rotate. The signal pin is typically a PWM signal, which is a type of digital signal that varies in width. The width of the signal determines the speed of the motor.
- VCC pin: This pin provides power to the motor. The voltage of the VCC pin must be within the voltage range specified by the motor manufacturer.
- Ground pin: This pin is connected to the ground of the circuit.

Interfacing:

The signal pin is connected to the controller that will be used to control the motor. The VCC pin is connected to the power supply that will be used to power the motor. The ground pin is connected to the ground of the circuit.

Once the motor is wired up, it can be controlled using the controller. The controller will send a PWM signal to the motor, which will tell the motor how to rotate. The speed of the motor can be controlled by adjusting the width of the PWM signal.

APIs:

- Function Name: ACT_vidActuateMotor

Function Description:

The ACT_vidActuateMotor function is used to control the speed and direction of a motor connected to a microcontroller. The function takes three parameters: the motor to be actuated, the direction of rotation, and the speed of rotation. The function uses Pulse Width Modulation (PWM) to control the motor speed and sets the direction of rotation by setting the output of two GPIO pins connected to the motor driver.

Parameters:

- Copy_enuMotor: An enumeration of the motor to be actuated. This parameter can take two values: LEFT_MOTOR or RIGHT_MOTOR.
- Copy_enuDirection: An enumeration of the direction of rotation. This parameter can take two values: FORWARD_ACT or BACKWARD_ACT.
- Copy_u8SpeedPercentage: The speed of rotation of the motor, specified as a percentage of the maximum speed. This parameter can take a value between 0 and 100.

Return Type: The function does not have a return value.

Input:
The function takes three input parameters: Copy_enuMotor, Copy_enuDirection, and Copy_u8SpeedPercentage. These parameters are used to control the speed and direction of the motor.

Output:
The function does not have any output, but it changes the speed and direction of the motor by controlling the output of two GPIO pins and using PWM to adjust

the motor speed.

Function Logic:

The function starts by checking if the Copy_u8SpeedPercentage parameter is greater than 100. If it is, the function sets the value to 100. Otherwise, it continues without any action.

Next, the function uses a switch statement to determine which motor to actuate based on the Copy_enuMotor parameter. If the value is LEFT_MOTOR, the function starts the PWM signal on Timer 2 Channel 3 with a period of 250, and a duty cycle specified by the Copy_u8SpeedPercentage parameter. Then, the function switches on the Copy_enuDirection parameter, and if it is FORWARD_ACT, the function sets the output of GPIO pin PA4 to HIGH, which causes the motor to rotate in the forward direction. If it is BACKWARD_ACT, the function sets the output of GPIO pin PA4 to LOW, which causes the motor to rotate in the backward direction.

If the value of Copy_enuMotor is RIGHT_MOTOR, the function starts the PWM signal on Timer 2 Channel 4 with the same period and duty cycle as the left motor. Then, the function switches on the Copy_enuDirection parameter, and if it is FORWARD_ACT, the function sets the output of GPIO pin PA5 to HIGH, which causes the motor to rotate in the forward direction. If it is BACKWARD_ACT, the function sets the output of GPIO pin PA5 to LOW, which causes the motor to rotate in the backward direction.

- **Valve**

Valves are mechanical devices that are used to control the flow of fluids or gases in a system. They are used in a wide range of applications, such as chemical processing, power generation, water treatment, and oil and gas

production. Valves can be operated manually or automatically, depending on the requirements of the system.

Features:

- Precise control: Valve actuators can be used to precisely control the flow of fluids or gases. This is important in many applications, such as process control and safety systems.
- Remote operation: Valve actuators can be operated remotely, which can be useful in hazardous or remote locations. This is especially useful in applications where it is not safe or practical for humans to be in close proximity to the valve.
- Durability: Valve actuators are typically very durable and can withstand harsh environments. This is important in applications where the actuator may be exposed to extreme temperatures, pressures, or chemicals.
- Safety: Valve actuators can be used to prevent the release of hazardous materials. This is especially important in safety systems, where the actuator must be able to operate quickly and reliably in the event of an emergency.

Specifications:

- Size: Valve actuators come in a variety of sizes, from small actuators that can fit in the palm of your hand to large actuators that can be several feet tall.
- Weight: Valve actuators also come in a variety of weights, from lightweight actuators that can be easily moved to heavy actuators that require lifting equipment.
- Power requirements: Valve actuators can be powered by a variety of sources, including compressed air, electricity, or hydraulic fluid.

- Materials: Valve actuators are typically made from durable materials, such as steel, aluminum, or plastic.
- Operating temperature: Valve actuators can operate in a wide range of temperatures, from very cold to very hot.
- Operating pressure: Valve actuators can operate in a wide range of pressures, from very low to very high.

Figure 18:valve

Pinout:

- Power: Supplies power to the actuator
- Signal: Controls the actuator
- Feedback: Provides feedback on the position of the actuator
- Ground: Provides a common ground for the actuator

Interfacing:

- Identify the pins on the valve actuator. The first step is to identify the pins on the valve actuator that will be used to control it. These pins are typically labeled as "power," "signal," "feedback," and "ground."
- Connect the valve actuator to the microcontroller

APIs:

- Function Name: ACT_vidActuateValve

Function Description:

The ACT_vidActuateValve function is used to control the state of a valve by setting the output of a GPIO pin connected to the valve actuator. The function takes a single parameter, which is an enumeration of the valve state. If the valve state is TRUE, the function sets the GPIO pin output to HIGH, which causes the valve actuator to open the valve. If the valve state is FALSE, the function sets the GPIO pin output to LOW, which causes the valve actuator to close the valve.

Parameter:

Copy_enuValveState: An enumeration of the valve state. This parameter can take two values: TRUE or FALSE.

Return Type: The function does not have a return value.

Input:

The function takes a single input parameter of type tenuValveState. This parameter is an enumeration that defines the state of the valve that needs to be actuated.

Output:

The function does not have any output, but it changes the state of the valve by setting the output of a GPIO pin connected to the valve actuator.

Function Logic:

The function starts with a switch statement that checks the value of the Copy_enuValveState parameter. If the value is TRUE, the function calls the GPIO_voidSetPinValue function with the arguments PORT_B, PIN0, and HIGH. This sets the output of the GPIO pin connected to the valve actuator to

HIGH, which opens the valve. If the value is FALSE, the function calls the GPIO_voidSetPinValue function with the arguments PORT_B, PIN0, and LOW. This sets the output of the GPIO pin connected to the valve actuator to LOW, which closes the valve.

Communication Module:

- Function: COM_vidSendToRaspBerryPi

Description:
This function sends a message to the Raspberry Pi using I2C communication. The message is formatted as a string and contains data related to either sensor readings or encoder readings.

Syntax:

void COM_vidSendToRaspBerryPi(tstrRaspberryPiMsg Copy_strMsg, tenuMsgType Copy_enuMsgType);

Input:

Copy_strMsg: A structure containing the message data to be sent to the Raspberry Pi. The structure has the following members:

- s8GyroX: A signed 8-bit integer representing the X-axis gyroscope reading.
- s8GyroY: A signed 8-bit integer representing the Y-axis gyroscope reading.
- s8GyroZ: A signed 8-bit integer representing the Z-axis gyroscope reading.
- s16AccelX: A signed 16-bit integer representing the X-axis accelerometer reading.

- s16AccelY: A signed 16-bit integer representing the Y-axis accelerometer reading.
- s16AccelZ: A signed 16-bit integer representing the Z-axis accelerometer reading.
- u8GasPercentage: An unsigned 8-bit integer representing the gas percentage.
- u8Temperature: An unsigned 8-bit integer representing the temperature.
- s8LeftMotorRPM: A signed 8-bit integer representing the left motor RPM.
- s8RightMotorRPM: A signed 8-bit integer representing the right motor RPM.

Copy_enuMsgType: An enumeration type variable representing the type of message to be sent. The two possible values are:

- SENSORS_COMM: Indicates that the message contains sensor readings.
- MOTION_COMM: Indicates that the message contains encoder readings.

Output: This function does not return any value.

Example Usage:

```
tstrRaspberryPiMsg message = {0};

message.s8GyroX = 10;

message.s8GyroY = 20;

message.s8GyroZ = 30;

message.s16AccelX = 100;

message.s16AccelY = 200;
```

```
message.s16AccelZ = 300;

message.u8GasPercentage = 50;

message.u8Temperature = 25;

message.s8LeftMotorRPM = 100;

message.s8RightMotorRPM = 200;
```

COM_vidSendToRaspBerryPi(message, SENSORS_COMM); // Sends sensor data to Raspberry Pi

Pre-requisites:
This function requires the initialization of the I2C communication interface and the implementation of the I2C_vidSlaveTX function.

Post-requisites:
This function sends a message to the Raspberry Pi.

Note:
This function formats the message as a string and sends it to the Raspberry Pi using the I2C_vidSlaveTX function. The message contains sensor or encoder readings, depending on the value of the Copy_enuMsgType parameter.

- Function: COM_vidRecFromRaspBerryPi

Description:
This function receives a message from the Raspberry Pi using I2C communication. The message is formatted as a string and contains data related to motor speeds and valve position.

Syntax:

```
void COM_vidRecFromRaspBerryPi(tstrStmMsg *Copy_pstrMsg);
```

Input:

Copy_pstrMsg: A pointer to a tstrStmMsg structure that will store the received message data. The structure has the following members:

- s8RightMotorSpeed: A signed 8-bit integer representing the right motor speed.
- s8LeftMotorSpeed: A signed 8-bit integer representing the left motor speed.
- u8Valve: An unsigned 8-bit integer representing the valve position.

Output:
This function does not return any value but updates the Copy_pstrMsg structure with the received message data.

Example Usage:

tstrStmMsg message = {0};

COM_vidRecFromRaspBerryPi(&message); // Receives message from Raspberry Pi

Pre-requisites:
This function requires the initialization of the I2C communication interface and the implementation of the I2C_vidSlaveRX and s32ToInteger functions.

Post-requisites:
This function receives a message from the Raspberry Pi and stores the received data in the Copy_pstrMsg structure.

Note:
This function receives a message from the Raspberry Pi using the I2C_vidSlaveRX function and stores the received data in the Copy_pstrMsg

structure. The received message contains data related to motor speeds and valve position. The function uses the s32ToInteger function to convert the ASCII characters representing the motor speeds and valve position into their corresponding integer values.

APP Layer:

DataAcquisition Module:

- Function: DAQ_vidInit

Description:
This function initializes the LOC_strMsg structure with default values for the gyroscope readings, accelerometer readings, gas percentage, and temperature.

Syntax: void DAQ_vidInit(void);

Input:
This function does not take any input.

Output:
This function does not return any value but initializes the LOC_strMsg structure with default values.

Example Usage:

DAQ_vidInit(); // Initializes the LOC_strMsg structure with default values

Pre-requisites:
This function assumes that the LOC_strMsg structure has been defined and that it has the required members for gyroscope readings, accelerometer readings, gas percentage, and temperature.

Post-requisites:

This function initializes the LOC_strMsg structure with default values.

Note:
This function sets the default values for the gyroscope readings (s8GyroX, s8GyroY, and s8GyroZ), the accelerometer readings (s16AccelX, s16AccelY, and s16AccelZ), the gas percentage (u8GasPercentage), and the temperature (u8Temperatue) members of the LOC_strMsg structure.

- Function: DAQ_vidCollectData

Description:
This function collects data from the sensors and formats it into a message to be sent to the Raspberry Pi.

Syntax: void DAQ_vidCollectData(void);

Input: This function does not take any input.

Output:
This function does not return any value but sends the sensor data to the Raspberry Pi using the COM_vidSendToRaspBerryPi function.

Example Usage:

DAQ_vidCollectData(); // Collects data from the sensors and sends it to the Raspberry Pi

Pre-requisites:
This function assumes that the COM_vidSendToRaspBerryPi function and the LOC_strMsg structure have been defined.

Post-requisites:
This function sends the sensor data to the Raspberry Pi using the COM_vidSendToRaspBerryPi function.

Note:
This function collects sensor data using the functions provided by the SEN module and formats it into a message to be sent to the Raspberry Pi. The function uses SEN_vidUpdateSensorsData to update the sensor data, SEN_u16GetTemperature to get the temperature reading, SEN_u8GetGasPercentage to get the gas percentage reading, and SEN_vidGetGyroAccel to get the gyroscope and accelerometer readings. The sensor data is then stored in the LOC_strMsg structure and sent to the Raspberry Pi using the COM_vidSendToRaspBerryPi function with the SENSORS_COMM message type.

RobotMotion Module:

- Function: RMO_vidInit

Description:
This function initializes the LOC_strRaspMsg and LOC_strStmMsg structures with default values for the left and right motor RPM and speed.

Syntax: void RMO_vidInit(void);

Input: This function does not take any input.

Output:
This function does not return any value but initializes the LOC_strRaspMsg and LOC_strStmMsg structures with default values.

Example Usage:

RMO_vidInit(); // Initializes the LOC_strRaspMsg and LOC_strStmMsg structures with default values.

Pre-requisites:

This function assumes that the LOC_strRaspMsg and LOC_strStmMsg structures have been defined and that they have the required members for left and right motor RPM and speed.

Post-requisites:
This function initializes the LOC_strRaspMsg and LOC_strStmMsg structures with default values.

Note:
This function sets the default values for the left and right motor RPM (s8LeftMotorRPM and s8RightMotorRPM) members of the LOC_strRaspMsg structure and the left and right motor speed (s8LeftMotorSpeed and s8RightMotorSpeed) members of the LOC_strStmMsg structure.

RTOS

What is RTOS?

RTOS, short for Real-Time Operating System, is a specialized operating system designed specifically for embedded systems that require deterministic and time-critical response. Unlike general-purpose operating systems like Windows or Linux, RTOS is optimized for real-time tasks and provides features to ensure timely and predictable execution of processes in the system.

The characteristics of an RTOS:

- Determinism: An RTOS guarantees deterministic behavior, meaning that tasks and processes are executed within specific time constraints. It ensures that tasks with higher priority or time-critical requirements are given priority and executed promptly.
- Task Scheduling: An RTOS employs a scheduler that determines the order and timing of task execution. The scheduler is designed to

efficiently allocate CPU resources and manage task priorities, allowing for smooth and predictable execution of tasks.

- Task Management: RTOS provides mechanisms for task creation, suspension, and termination. It manages the execution of multiple tasks concurrently, ensuring that system resources are allocated appropriately and that each task is given sufficient processing time.
- Interrupt Handling: Interrupt handling is crucial in real-time systems, as hardware events and time-sensitive inputs need to be responded to promptly. An RTOS provides mechanisms to handle interrupts efficiently, allowing for quick context switches to handle time-critical events.
- Resource Management: An RTOS manages system resources such as memory, peripherals, and I/O devices. It provides mechanisms for resource sharing, allocation, and synchronization, ensuring that tasks have the necessary resources to execute and communicate without conflicts.
- Timing Services: RTOS often provides timing services and tools to measure and control the timing requirements of tasks. This can include features such as timers, periodic scheduling, and precise timing functions, allowing developers to enforce strict time constraints on critical operations.
- Communication and Synchronization: RTOS provides communication mechanisms, such as message queues, semaphores, and event flags, to facilitate inter-task communication and synchronization. These mechanisms enable tasks to exchange data, coordinate actions, and ensure effective collaboration within the system.
- Small Footprint: Embedded systems often have limited resources, including memory and processing power. Therefore, RTOSs are designed

to have a small footprint, optimizing resource usage and minimizing overhead.

RTOSs are commonly used in applications that require real-time behavior, such as industrial automation, robotics, automotive systems, medical devices, and aerospace systems. They enable precise control, predictable timing, and reliable operation, ensuring that critical tasks and processes meet their specific timing requirements.

In our system, RTOS is used to manage the two main tasks in our system, the Data Acquisition task, and the Robot Motion task. The system tick is 100ms and the two tasks is executed every 200ms.

OS Module:

- Function: OS_vidInit

Description:
This function initializes the operating system by setting the state to OS_INIT, setting the LOC_u8Periodicity array with the periodicity values of each task, and starting the SysTick timer with the OS_vidSystickISR function as the interrupt service routine.

Syntax: void OS_vidInit(void);

Input: This function does not take any input.

Output: This function does not return any value but initializes the operating system.

Example Usage: OS_vidInit(); // Initializes the operating system

Pre-requisites:
This function assumes that the GLOB_astrTasks array, LOC_u8Periodicity

array, OS_NUM_OF_TASKS constant, OS_TICK_MILLISECOND constant, SYSTICK_vidStartAsync function, and OS_vidSystickISR function have been defined.

Post-requisites:
This function initializes the operating system.

Note:
This function sets the state of the operating system to OS_INIT, sets the LOC_u8Periodicity array with the periodicity values of each task, and starts the SysTick timer with the OS_vidSystickISR function as the interrupt service routine. The LOC_u8Periodicity array is initialized to the periodicity values of each task in the GLOB_astrTasks array, with the exception of the first task, which is decremented by one. This ensures that the first task is executed immediately after initialization. The SYSTICK_vidStartAsync function is used to start the SysTick timer with a period of 2000 * OS_TICK_MILLISECOND to generate interrupts every OS_TICK_MILLISECOND milliseconds.

- Function: OS_vidScheduler

Description:
This function performs the scheduling of the tasks in the operating system based on their periodicity.

Syntax: void OS_vidScheduler(void);

Input: This function does not take any input.

Output:
This function does not return any value but executes the tasks in the operating system.

Example Usage:

OS_vidScheduler(); // Performs the scheduling of the tasks in the operating system

Pre-requisites:
This function assumes that the GLOB_astrTasks array, LOC_u8Periodicity array, OS_NUM_OF_TASKS constant, and TASK_RUNNING, TASK_READY enum values have been defined.

Post-requisites:
This function executes the tasks in the operating system.

Note:
This function performs the scheduling of the tasks in the operating system based on their periodicity. The function first checks the state of the operating system and performs different actions depending on the state. If the state is OS_INIT, the function sets the state to OS_WAITING, executes all tasks once, and sets their state to TASK_READY. If the state is OS_EXECUTING, the function sets the state to OS_WAITING, executes tasks that are ready to run based on their periodicity, and sets their state to TASK_READY. If the state is OS_WAITING, the function simply waits for the next SysTick interrupt. The LOC_u8Periodicity array is used to keep track of the remaining time until each task should be executed again. The periodicity values of the tasks are decremented each time the scheduler runs, and when a periodicity value reaches zero, the task is marked as ready to run again and its periodicity value is reset.

- Function: OS_vidSystickISR

Description:
This function is the interrupt service routine for the SysTick timer and sets the state of the operating system to OS_EXECUTING.

Syntax: void OS_vidSystickISR(void);

Input: This function does not take any input.

Output:
This function does not return any value but sets the state of the operating system.

Example Usage:

OS_vidSystickISR(); // Sets the state of the operating system to OS_EXECUTING

Pre-requisites:
This function assumes that the LOC_enuOSState and OS_EXECUTING enum value have been defined.

Post-requisites:
This function sets the state of the operating system.

Note:
This function is the interrupt service routine for the SysTick timer and sets the state of the operating system to OS_EXECUTING. When the SysTick timer generates an interrupt, this function is called and sets the state of the operating system to OS_EXECUTING. This state indicates that the scheduler should run and execute the tasks that are ready to run based on their periodicity.

Class diagram

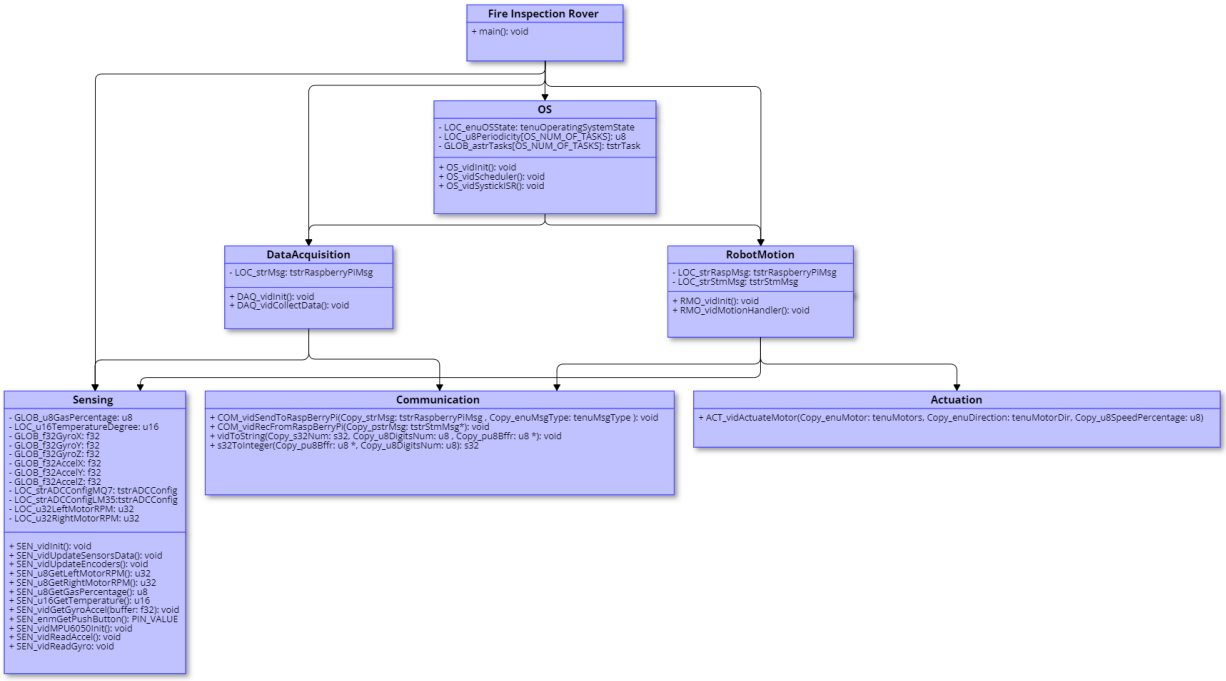

Figure 19:embedded Class diagram

Sequence diagram:

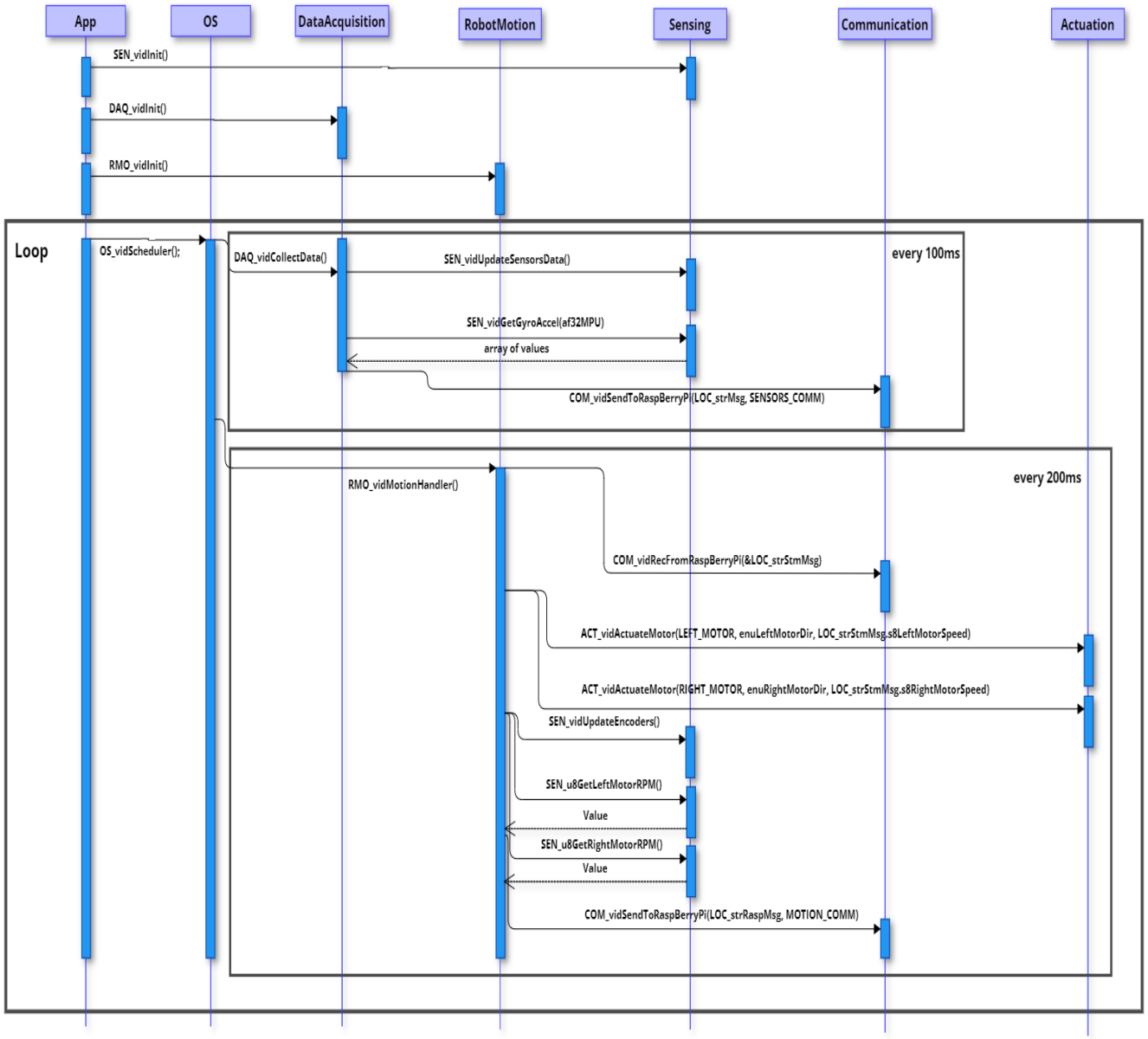

Figure 20:Embedded sequence diagram

4.2 ROS

ROS System Design:

ROS can be summarized into two main components,

1) ROS Across Machines:

ROS across machines in our project is used to connect The control station with the Fire Inspection Rover or any number of needed rovers, but meanwhile the control station acts as the master of the system connecting all the nodes to each other and to it.

All the topics and services are shared by all the nodes in the network as shown in the figure.

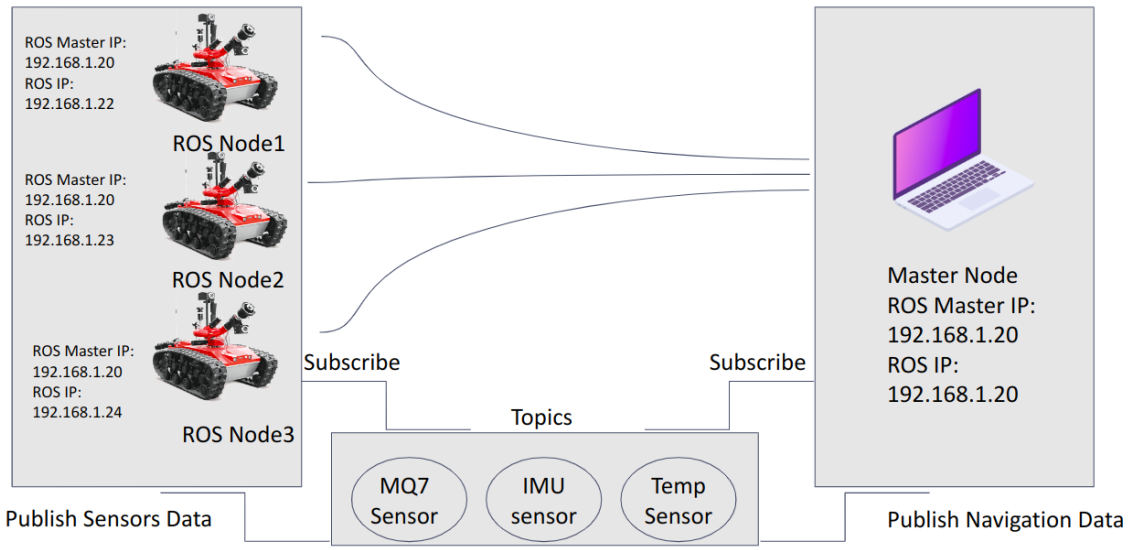

Figure 21:ROS across Machines

As for the nodes in the system, topics, and their connections they can be better represented by the following table:

Table 3:Ros Topics

Node	Topics	Data source	Subscriber Nodes
Sensors_data	/imu	STM32 through i2c interface on the rover	differential_drive_controller
	/temp		
	/encoder_left_wheel		sensors_rviz
	/encoder_right_wheel /mq7		
Log_node	/joystick_data	Control station	differential_drive_controller
differential_drive_controller	/right_wheel_effort_controller	Rover controllers	Sensors_data
	/left_wheel_effort_controller		

sensors_rviz	/tf_publisher	Control station	

2- ROS Control:

The ROS Control package provides a framework for controlling robots in the Robot Operating System (ROS). It consists of several key components that work together to achieve robot control:

The used controllers are.

- joint_state_controller: This controller reads all joint positions and publishes them on to the topic "/joint_states". This controller does not send any command to the actuators. Used as "joint_state_controller/JointStateController".
- JointEffortController : Used as "effort_controllers/JointEffortController". This controller plugin accepts the effort [force (or) torque] values as input. The input effort is simply forwarded as output effort command to the joint actuator, passing by a PID Controller first.

The effort controllers are two called:

- One for right wheel : right_wheel_effort_controller
- One for left wheel: left_wheel_effort_controller

The following rqt_graph shows system nodes and topics and their connections:

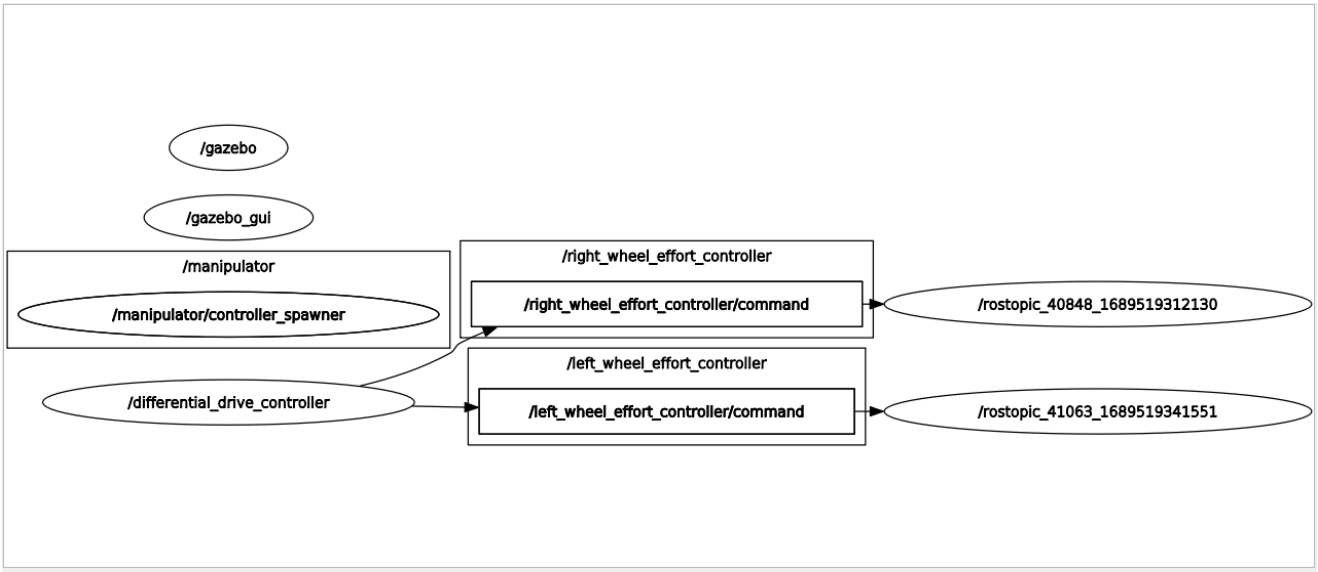

Figure 22:ROS dataflow

ROS Sequence Diagrams:

1- Data communication:

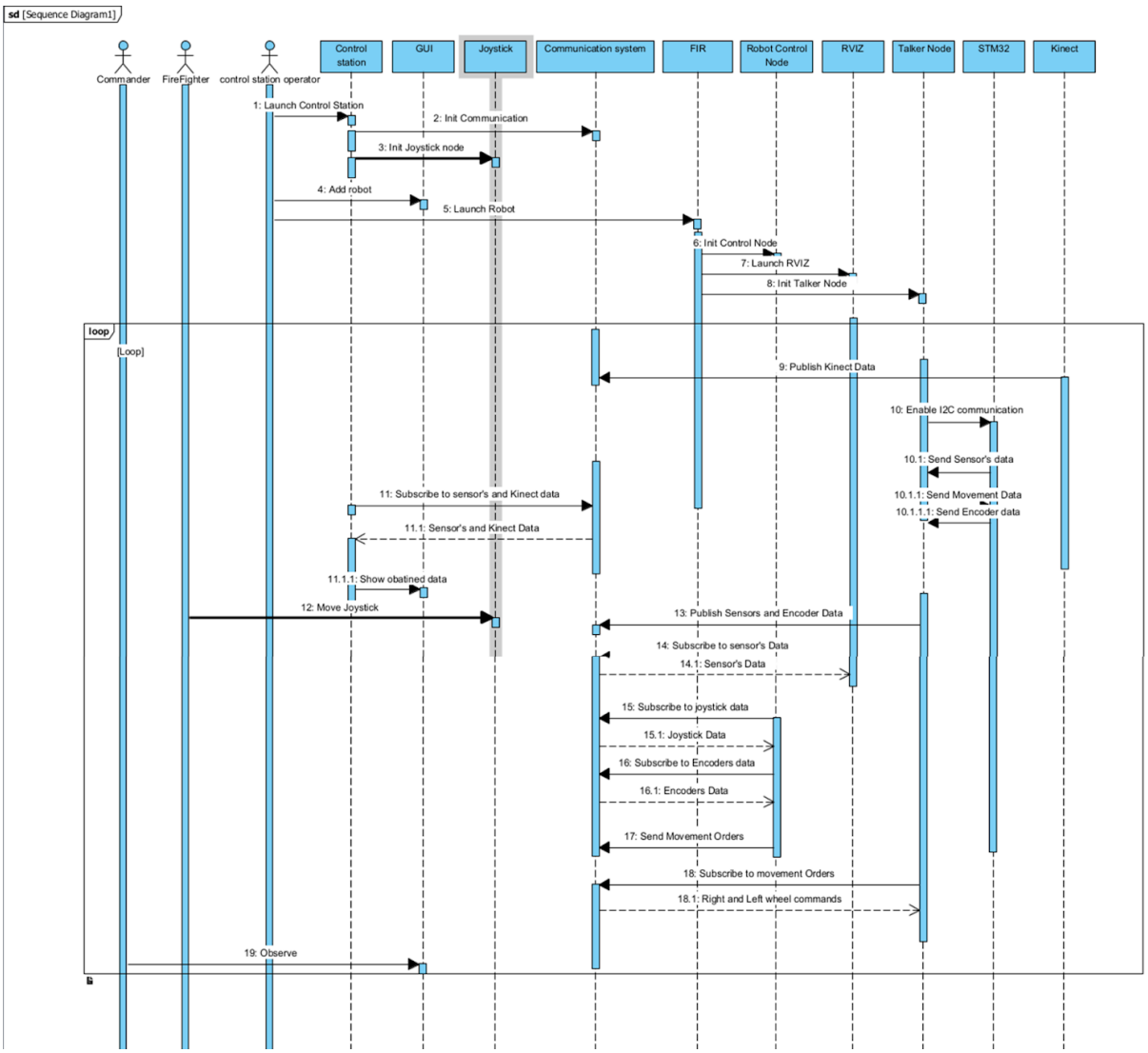

Figure 23:Ros sequence diagram

2- Localization and mapping sequence diagram:

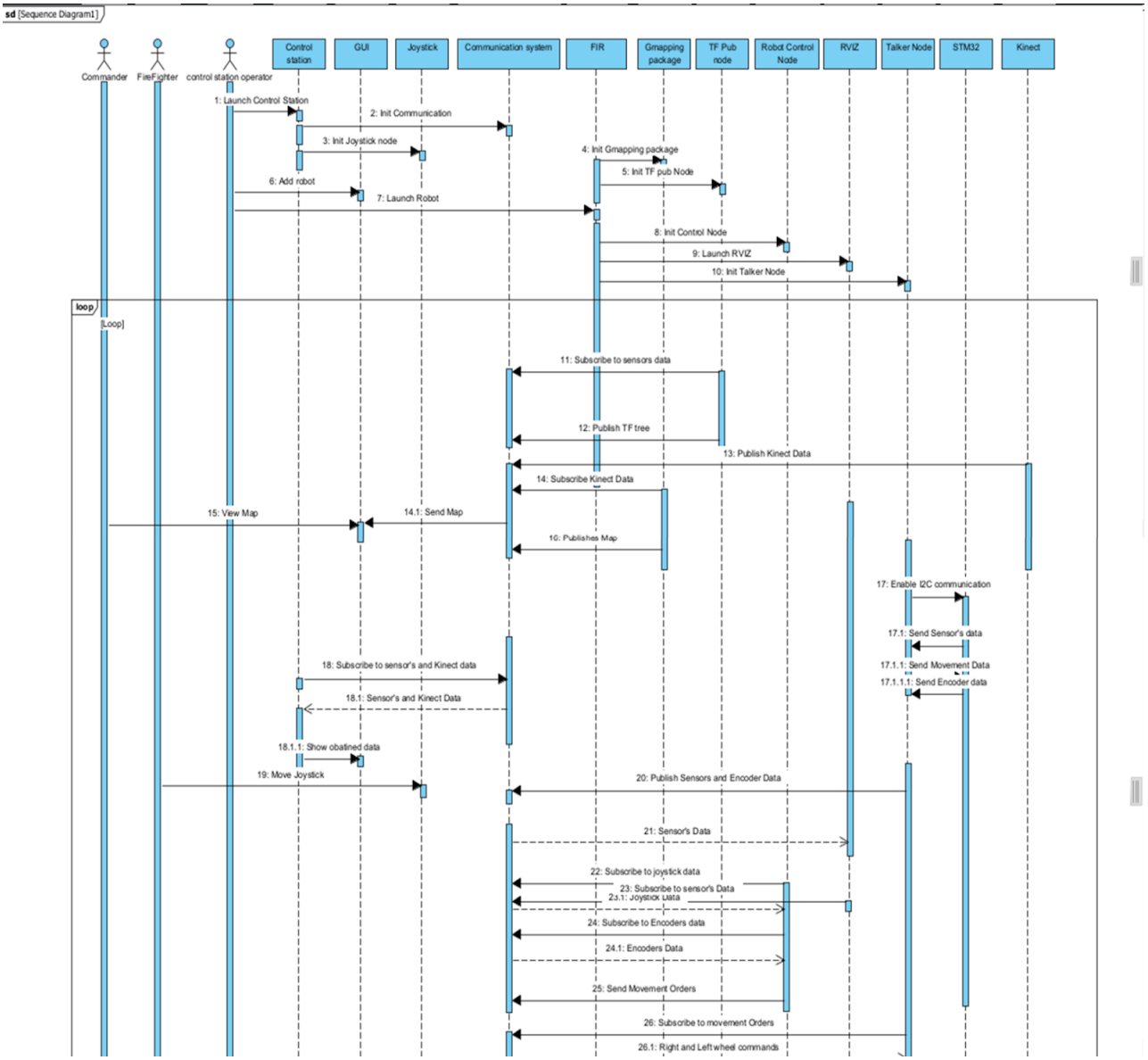

Figure 24:Ros sequence diagram2

ROS class diagram:

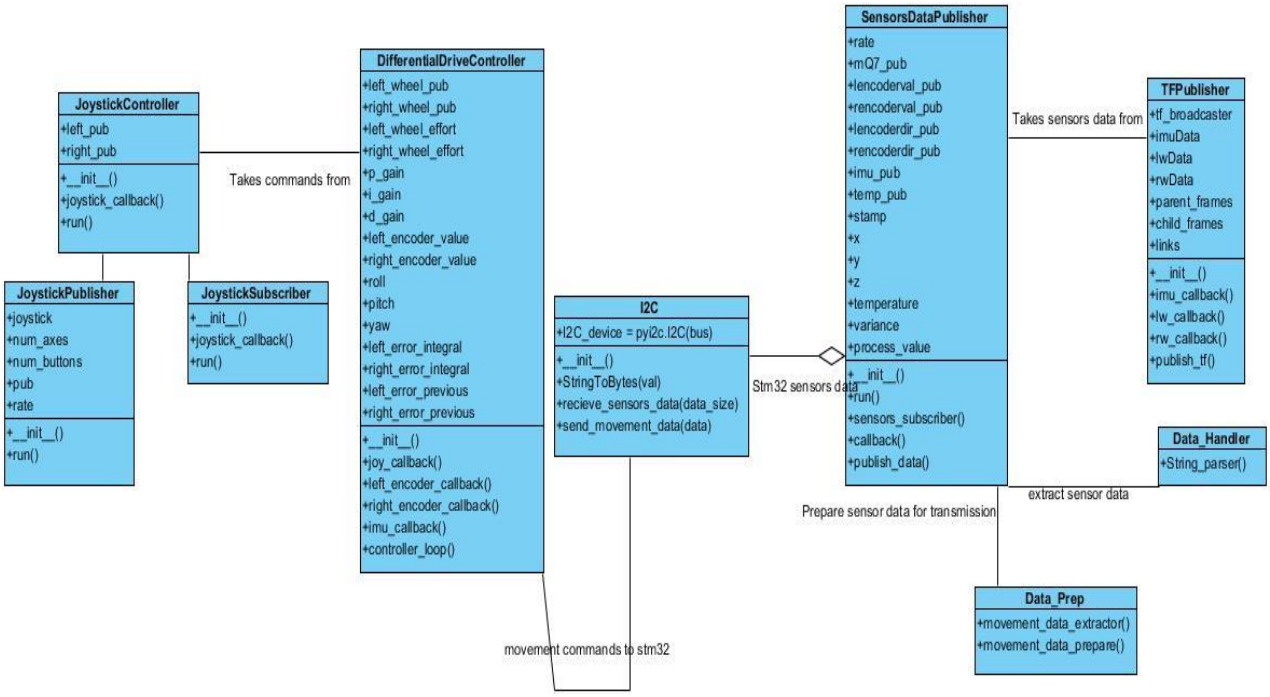

Figure 25:Ros Class diagram

4.3 Unity

Unity Class diagram

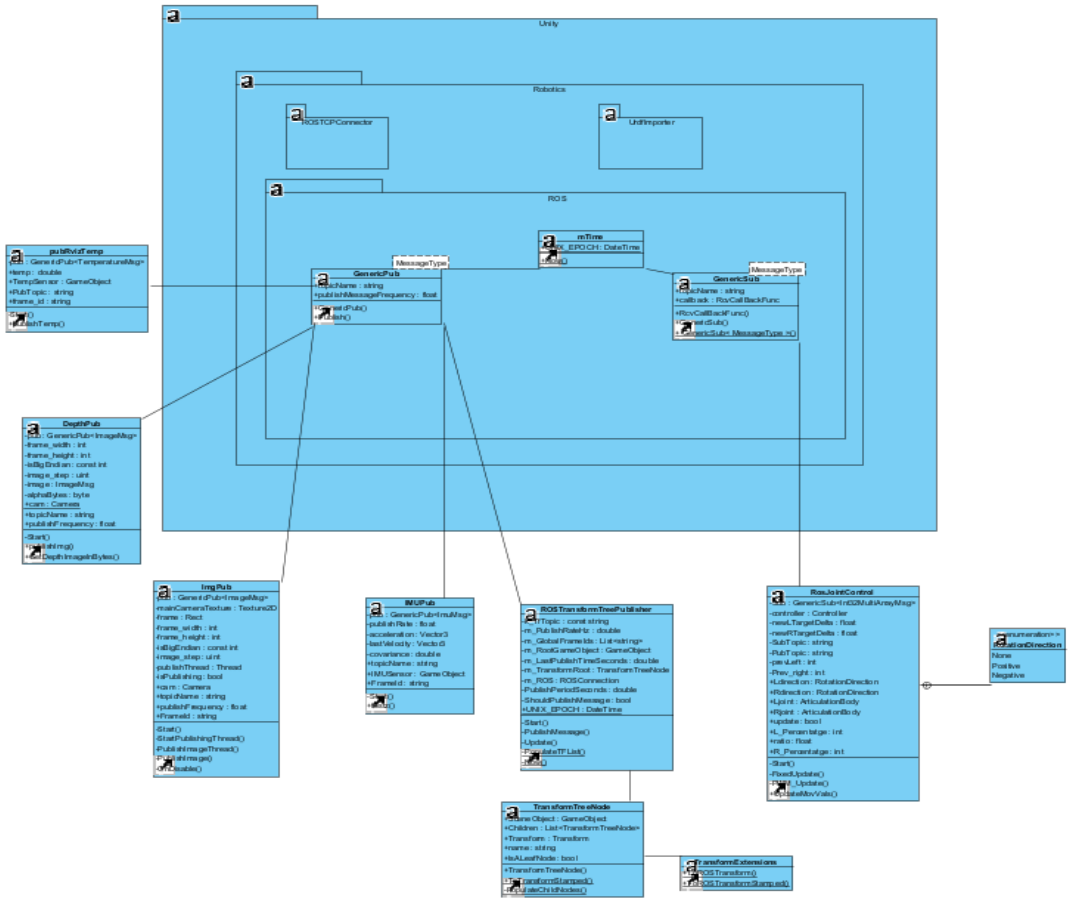

Figure 26:unity Class diagram

Sequence Diagram

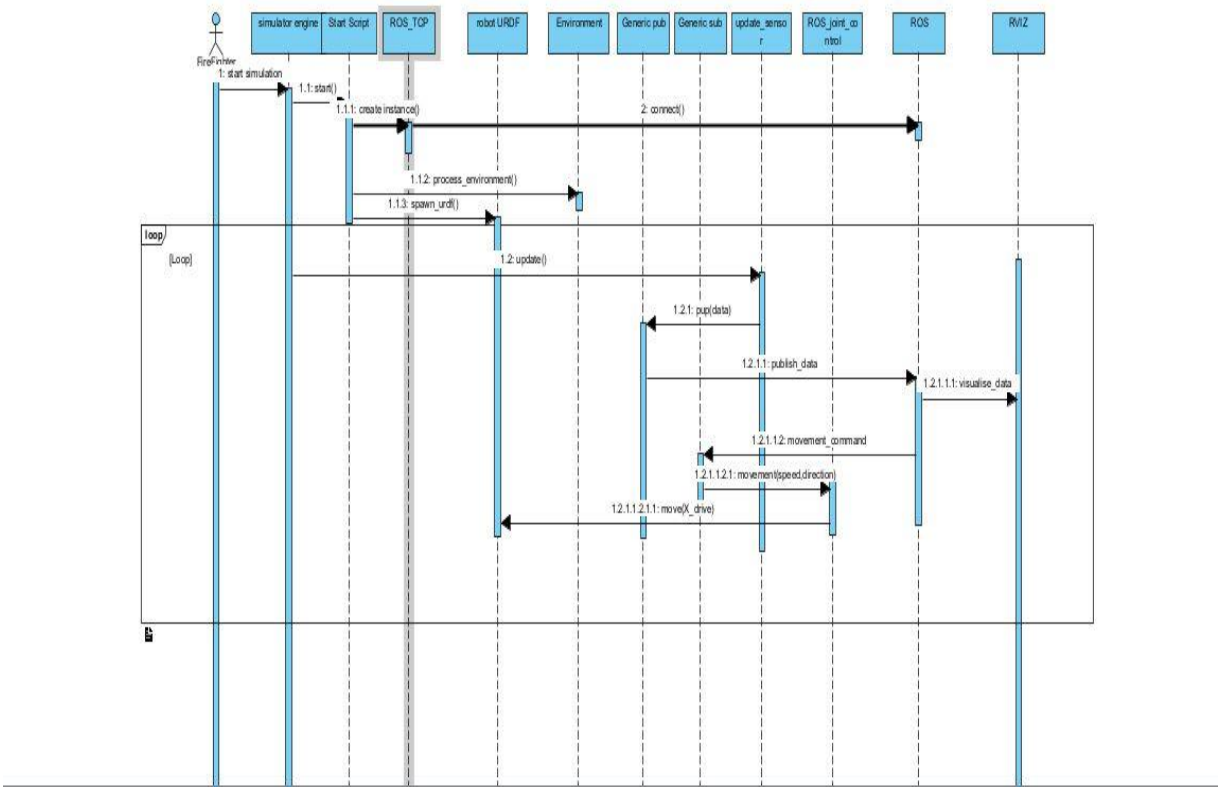

Figure 27:unity sequence diagram

Unity Start function

On start simulation the simulator engine call start function of every script in the project

The start function in sensorPublisher scripts do the following:-

II. Create a publisher instance from genericPublisher class and pass to it the topic name and the publishing frequency
III. Get the Object reference of the sensor on the robot

The start function in movementSubscriper script do the following:-

I. Pass the topic to subscribe to

II. Define and pass the callback function to be called when receiving a message

The start function in the genericPub class start a ros connection instance for every publisher and register it as well as defining the message type

The start function in the genericSub class register a subscriber in ros system the GenericSub constructor take the topic and the callback function as arguments

The start function also respawn the robot urdf and add its collider (articulation body) so that the physics engine can calculate the robot movement collision drag and speed

Unity Update function
The Update function in every script is called periodically for every script

The update function in sensorPublisher scripts do the following:-

1)create a new Ros Message according to the sensor type and publish it to ROS
The update function in MovementSubscriber scripts do the following:-

1)check for incoming message and call callback function for it

The update function for jointControlScript (movement controller) take the new movement commands and calculate robot Xdrive and calculate its collision

5. Chapter 5 : Technologies and Tools

5.1 Embedded System

5.1.1 Embedded C (STM Blackpill)

In addition to the utilization of embedded C programming, the Blackpill development board, and Eclipse IDE, we efficiently handled hardware interfacing and processing on STM microcontrollers. These microcontrollers played a crucial role in managing the input/output operations and processing of data within our project.

By leveraging STM microcontrollers, we were able to interface with various sensors, actuators, and peripheral devices seamlessly. The STM microcontrollers provided us with extensive I/O capabilities and a robust ecosystem of libraries and drivers, making it easier to communicate with and control external hardware components.

To transmit the processed data, we adopted the I2C (Inter-Integrated Circuit) communication protocol. This protocol facilitated reliable and efficient data transmission by utilizing a two-wire interface. We encapsulated the processed data within I2C frames, ensuring the secure and orderly transfer of information between the STM microcontrollers and other devices in the system.

By handling hardware interfacing and processing on STM microcontrollers and using the I2C communication protocol, we achieved efficient and reliable data transfer within our project. This enabled seamless integration between the embedded C code running on the microcontrollers and other components of the system, ultimately contributing to the overall functionality and success of our graduation project.

In Embedded C we developed a Real time operating system on the STM microcontroller to schedule the motor control and sensor readings tasks (developed in C language)

5.1.2 Embedded Linux (Raspberry pi 4)

We used Raspberry Pi as an embedded Linux platform, operating on Ubuntu MATE 20. The Raspberry Pi's versatility allowed us to execute Python scripts for seamless communication with the STM microcontrollers. This facilitated efficient data exchange and control signals between the two components. Moreover, we integrated the power of ROS (Robot Operating System) on the Raspberry Pi, enabling us to effectively manage all robot communication. ROS served as a central hub, streamlining data flow between various modules, sensors, and actuators, simplifying the development and coordination of the entire system. Additionally, we successfully interfaced a Kinect sensor with the Raspberry Pi, expanding the system's capabilities with depth information. This combination of Raspberry Pi, Ubuntu MATE 20, Python scripts, ROS, and Kinect integration.

We used Ubuntu OS on the Raspberry Pi with ROS system installed and wrote C++ and python scripts for ROS nodes and RPI STM I2C communication.

5.2 ROS

One of the key technologies we utilized was ROS (Robot Operating System) 1 Melodic. ROS served as a fundamental framework that facilitated seamless communication, data sharing, and resource integration across various components of our project.

Firstly, ROS enabled efficient communication and coordination between different scripts and modules within our system. Through its messaging

infrastructure, we were able to establish reliable data exchange, enabling real-time information flow between different parts of the project. This allowed us to integrate and synchronize the functionalities of multiple scripts, ensuring effective collaboration and synergy.

Secondly, ROS facilitated communication across different machines within our setup. By leveraging ROS's distributed architecture, we could seamlessly exchange data and commands between multiple computers or with computer and raspberry pi, enabling distributed computing and enabling our project to scale as needed. This capability proved especially valuable when dealing with complex tasks that required the coordination of multiple devices or when implementing distributed systems.

Additionally, we leveraged ROS for its extensive library of control packages and ready-made drivers. This allowed us to easily interface with robotic platforms. By utilizing ROS's pre-existing drivers and control packages, we saved significant development time and effort, focusing instead on implementing higher-level functionalities specific to our project.

ROS played a crucial role in our graduation project by providing a powerful and flexible framework for communication, data sharing, and resource integration. Its ability to facilitate communication between scripts, support cross-machine communication, and offer a rich set of control packages and drivers greatly contributed to the success and efficiency of our project implementation.

In ROS system as a means of communication between different machines and with Raspberry Pi we wrote scripts in both Python, C++ and C# languages.

5.3 Unity

Unity played a crucial role as a powerful simulation tool, allowing us to

simulate the robot in various fire situations. We utilized two essential packages from Unity Robotics to enhance our simulation capabilities.

I. The Unity URDF Importer package enabled us to simulate the robot's URDF (Unified Robot Description Format) in Unity accurately. With this package, we could visualize the robot's physical components and kinematic structure, making it easier to experiment with different control algorithms and assess their efficiency in a controlled virtual environment. This simulation allowed us to fine-tune the robot's behavior before implementing it in real-world scenarios, ensuring optimal performance during firefighting operations.

II. The ROS Unity TCP package facilitated seamless communication between the simulated robot in Unity and the ROS (Robot Operating System) system. This integration enabled us to bridge the gap between the virtual and physical worlds, enabling data exchange between the simulated robot and the rest of the ROS ecosystem. With this connection, we could test the robot's responses to commands and sensor data from the ROS system, ensuring a high level of accuracy and realism in our simulations.

By leveraging Unity and the Unity Robotics packages, we could create realistic and dynamic fire situations, assess different control algorithms, and validate the robot's behavior within the ROS environment. This comprehensive simulation approach significantly contributed to the success and effectiveness of our fire inspection robot project, ensuring the robot's readiness for real-world firefighting scenarios.

All unity scripts are written in C#.

5.4 SLAM

We used the GMAPPING package which is based on Rao-Blackwellized filters are employed to achieve robust and accurate Simultaneous Localization and Mapping (SLAM) results. These filters combine the strengths of two estimation techniques: a particle filter and a Kalman filter.

Rao-Blackwellized filters in GMAPPING utilize a particle filter for handling the spatial uncertainty in mapping. It maintains a set of particles, each representing a possible map of the environment. These particles are updated based on sensor measurements, which allows the algorithm to create a probabilistic representation of the map.

The two stages of estimation and update are as follows: In the estimation stage, the algorithm predicts the robot's future position and updates the particle filters accordingly. Then, during the update stage, it corrects the particle weights based on the sensor data received from the environment. By combining these stages iteratively, GMAPPING continuously refines the map and robot's position estimates, achieving a reliable and accurate SLAM solution for our project's navigation and mapping requirements.

6. Chapter 6 : Testing

6.1 Introduction

This chapter discusses the testing and validation process that was performed on the software. The chapter begins with an overview of the testing and validation process, followed by a description of the different types of testing that were performed. The chapter then presents the results of the testing and validation and discusses any problems that were encountered during the process.

The testing and validation process was an important part of the development of the software. The process helped to ensure that the software is of high quality and that it meets the requirements. The process also helped to identify any problems with the software, and these problems were subsequently resolved.

Types of testing:

- Unit testing
- Integration testing
- System testing
- Acceptance testing

6.2 Unit Testing

Unit testing is the lowest level of testing, and it involves testing individual units of code. Unit tests are typically written by the developers themselves, and they are used to ensure that each unit of code works as expected.

The plan for unit testing typically includes a list of all the units of code that need to be tested, as well as the expected results of each test.

The strategy for unit testing typically involves using a unit testing framework to automate the testing process.

Test Plan

- Test Objective: To test the individual units of code to ensure that they work as expected.
- Test Scope: All units of code that are critical to the system.
- Test Methods: Unit testing framework, such as JUnit or gtest.
- Expected Results: All units of code should pass the unit tests.
- Test Procedures:

- Create a unit test for each unit of code.
- Run the unit tests and verify that they pass.
- If a unit test fails, investigate the cause of the failure and fix the code.

Unit testing for Embedded System:

For Sensing module:

Test Case ID	SEN_vidMPU6050Init()		
Test Case Description	initializes the MPU6050 sensor.	**Test Priority**	High
Pre-Requisite	The sensor should be connected to the system. should be powered on. should be in the default state.	**Post-Requisite**	The sensor should be initialized correctly. should be working correctly. should be in the configured state.

Test Execution Steps:

TC. No	**Inputs**	**Expected Output**	**Test Result**
1	I2C interface connected to MPU6050 sensor	Sensor is initialized and returns a WHO_AM_I value of MPU6050_ADDRESS_7_BITS. The sensor is also	Pass

		woken up, the sample rate is set to 1 KHz, and the full scale range for the accelerometer and gyroscope is set to ±2g and ±250 °/s, respectively.	
2	No sensor connected to I2C interface	The function does not initialize the sensor and no changes are made to the I2C bus.	Pass
3	Sensor connected to I2C interface but not powered on	The function does not initialize the sensor and no changes are made to the I2C bus.	Pass
4	Sensor connected to I2C interface and in different orientation	The function initializes the sensor and returns valid WHO_AM_I value. The sensor is also woken up, the sample rate is set to 1 KHz, and the full scale range for the accelerometer and gyroscope is set correctly for the new orientation.	Pass
5	Sensor connected to I2C interface and subjected to vibration or shock	The function initializes the sensor and returns valid WHO_AM_I value. The sensor is also woken up, the sample rate is set to 1 KHz, and the full scale range for the	Pass

		accelerometer and gyroscope is set correctly. The sensor readings are stable and do not exceed the maximum range of ±2g and ±250 °/s.	

Test Case ID	SEN_vidReadGyro()		
Test Case Description	Reads the gyroscope data from the MPU6050 sensor.	**Test Priority**	High
Pre-Requisite	The sensor is connected to the system. The sensor is powered on. The sensor is in the configured state.	**Post-Requisite**	The gyroscope data is read correctly. The values of GLOB_f32GyroX, GLOB_f32GyroY, and GLOB_f32GyroZ should be within the range of the gyroscope.

Test Execution Steps:

TC. No	**Inputs**	**Expected Output**	**Test Result**

1	I2C bus connected to MPU6050 gyroscope, gyroscope registers GYRO_XOUT_H to GYRO_ZOUT_L, 6 bytes of data	Function reads 6 bytes of data and stores the values in global variables GLOB_f32GyroX, GLOB_f32GyroY, and GLOB_f32GyroZ. Values are within expected range.	Pass
2	No I2C bus connected to the system	Function does not read any data and does not modify any global variables.	Pass
3	I2C bus connected to the system, no sensor connected to the I2C bus	Function does not read any data and does not modify any global variables.	Pass
4	MPU6050 gyroscope connected to I2C bus, gyroscope registers GYRO_XOUT_H to GYRO_ZOUT_L, sensor not powered on	Function does not read any data and does not modify any global variables.	Pass
5	I2C bus connected to MPU6050 gyroscope, gyroscope registers GYRO_XOUT_H to GYRO_ZOUT_L, 6 bytes of data from a different orientation	Function reads 6 bytes of data and stores the values in global variables GLOB_f32GyroX, GLOB_f32GyroY, and GLOB_f32GyroZ. Values are within expected range.	Pass
6	I2C bus connected to MPU6050 gyroscope, gyroscope registers GYRO_XOUT_H to GYRO_ZOUT_L, 6 bytes of data under vibration or shock	Function reads 6 bytes of data and stores the values in global variables GLOB_f32GyroX, GLOB_f32GyroY, and GLOB_f32GyroZ. Values are within expected range and remain stable despite the vibration or shock.	Pass

Test Case ID	SEN_vidReadAccel()		
Test Case Description	Reads the accelerometer data from the MPU6050 sensor.	**Test Priority**	High
Pre-Requisite	The sensor is connected to the system. The sensor is powered on. The sensor is in the configured state.	**Post-Requisite**	The accelerometer data is read correctly. The values of GLOB_f32AccelX, GLOB_f32Accel, and GLOB_f32AccelZ should be within the range of the accelerometer.

Test Execution Steps:

TC. No	**Inputs**	**Expected Output**	**Test Result**
1	I2C interface connected to sensor	GLOB_f32AccelX, GLOB_f32AccelY, and GLOB_f32AccelZ are valid acceleration readings within the range of -2.0 to 2.0 m/s²	Pass
2	No sensor connected to I2C interface	GLOB_f32AccelX, GLOB_f32AccelY, and GLOB_f32AccelZ are all zero	Pass
3	MPU6050 sensor connected but not powered on	GLOB_f32AccelX, GLOB_f32AccelY, and GLOB_f32AccelZ are all zero	Pass
4	MPU6050 sensor connected and in motion	GLOB_f32AccelX, GLOB_f32AccelY, and GLOB_f32AccelZ change in response to the motion of the sensor	Pass

5	MPU6050 sensor connected and subjected to vibration or shock	GLOB_f32AccelX, GLOB_f32AccelY, and GLOB_f32AccelZ change in response to the vibration or shock, but do not exceed the maximum range of -16.0 to 16.0 m/s²	Pass
6	MPU6050 sensor connected and subjected to temperature changes	GLOB_f32AccelX, GLOB_f32AccelY, and GLOB_f32AccelZ change in response to the temperature changes, but remain within the expected range	Pass

Test Case ID	SEN_vidInit ()		
Test Case Description	initializes all of the sensors that are used in the application. This includes the MQ-7 gas sensor, the LM35 temperature sensor, the MPU6050 gyroscope, and the two motor encoders.	**Test Priority**	High
Pre-Requisite	The sensors must be connected to the system. The sensors must be powered on. The sensor configuration must be correct.	**Post-Requisite**	All of the sensors must be initialized correctly. The sensors must be ready to be used.

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result
1	None	Initialize global variables GLOB_u8GasPercentage,	Pass

		LOC_u16TemperatureDegree, GLOB_f32AccelX, GLOB_f32AccelY, GLOB_f32AccelZ, GLOB_f32GyroX, GLOB_f32GyroY, GLOB_f32GyroZ, LOC_u32LeftMotorRPM, and LOC_u32RightMotorRPM to 0. Initialize LOC_strADCConfigMQ7 with the specified settings. Initialize LOC_strADCConfigLM35 with the specified settings. Initialize TIM3 with input capture for channels 1 and 2. Call SEN_vidMPU6050Init function.	

Test Case ID	SEN_vidUpdateSensorsData()		
Test Case Description	updates the data for all of the sensors that are used in the application. This includes the gas percentage, temperature, accelerometer, and gyroscope data.	**Test Priority**	High
Pre-Requisite	The sensors must be connected to the system.	**Post-Requisite**	The gas percentage, temperature, accelerometer, and

	The sensors must be powered on. The sensor configuration must be correct.		gyroscope data must be updated correctly.

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result
1	None	Initialize ADC with LOC_strADCConfigMQ7 settings. Read ADC value from channel CHANNEL0_ADC, divide by 41, and assign the result to GLOB_u8GasPercentage. Initialize ADC with LOC_strADCConfigLM35 settings. Read ADC value from channel CHANNEL1_ADC and assign the result to LOC_u16TemperatureDegree. Call SEN_vidReadAccel function to read accelerometer data and store the values in GLOB_f32AccelX, GLOB_f32AccelY, and GLOB_f32AccelZ. Call SEN_vidReadGyro function to read gyroscope data and store the values in GLOB_f32GyroX, GLOB_f32GyroY, and GLOB_f32GyroZ.	Pass

Test Case ID	SEN_vidUpdateEncoders()

Test Case Description	updates the RPMs for the two motors that are used in the application. The RPMs are calculated based on the number of pluses that have been captured by the encoders in a given time period.	**Test Priority**	High
Pre-Requisite	The motors must be connected to the system. The motors must be powered on. The timer must be configured correctly.	**Post-Requisite**	The motor RPMs must be updated correctly.

Test Execution Steps:

TC. No	**Inputs**	**Expected Output**	**Test Result**
1	None	Read the number of pulses captured by channel 1 of TIM3 with a delay of 10ms and assign the result to u32PlusesLeft. Read the number of pulses captured by channel 2 of TIM3 with a delay of 10ms and assign the result to u32PlusesRight. Calculate the left motor RPM by multiplying u32PlusesLeft by 30 and then dividing the result by 531. Assign the result to LOC_u32LeftMotorRPM. Calculate the right motor RPM by multiplying u32PlusesRight by 30 and then dividing the result by 531. Assign the result to LOC_u32RightMotorRPM.	Pass

For Actuation module:

Test Case ID	ACT_vidActuateMotor()		
Test Case Description	actuates the specified motor in the specified direction at the specified speed. The speed is a percentage value from 0 to 100, where 0 is stopped and 100 is full speed.	**Test Priority**	High
Pre-Requisite	The motor must be connected to the system. The motor must be powered on. The timer must be configured correctly.	**Post-Requisite**	The motor should be rotating in the specified direction at the specified speed.

Test Execution Steps:

TC. No	**Inputs**	**Expected Output**	**Test Result**
1	Left motor, forward, 50%	The left motor should start rotating forward at 50% speed.	Pass
2	Left motor, backward, 50%	The left motor should start rotating backward at 50% speed.	Pass
3	Right motor, forward, 50%	The right motor should start rotating forward at 50% speed.	Pass
4	Right motor, backward,	The right motor should start rotating backward at	Pass

	50%	50% speed.	
5	Left motor, forward, 100%	The left motor should start rotating forward at 100% speed.	Pass
6	Left motor, backward, 100%	The left motor should start rotating backward at 100% speed.	Pass
7	Right motor, forward, 100%	The right motor should start rotating forward at 100% speed.	Pass
8	Right motor, backward, 100%	The right motor should start rotating backward at 100% speed.	Pass

Test Case ID	ACT_vidActuateValve(tenuValveState Copy_enuValveState)		
TestCase Description	actuating the valve based on the valve state received as an input parameter. It sets the corresponding pin on Port B to HIGH or LOW depending on the valve state.	**Test Priority**	High
Pre-Requisite	The HAL module must be initialized and configured to enable the use of the GPIO pins. The valve must be connected to the correct GPIO pin on Port B. The Actuation module must be	**Post-Requisite**	The valve will be actuated based on the valve state received as

	operational.		an input parameter. The corresponding pin on Port B will be set to HIGH or LOW depending on the valve state.

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result	
1	TRUE	The valve pin on Port B, Pin 0 will be set to HIGH.	Pass	
2	FALSE	The valve pin on Port B, Pin 0 will be set to LOW.	Pass	

For communication module:

Test Case ID	vidToString (s32 Copy_s32Num, u8 Copy_u8DigitsNum, u8 *Copy_pu8Bffr)		
Test Case Description	converts a signed integer to a string. The number of digits in the string is specified by the Copy_u8DigitsNum parameter. The string is stored in the buffer pointed to by the	**Test Priority**	High

	Copy_pu8Bffr parameter.		
Pre-Requisite	The buffer pointed to by the Copy_pu8Bffr parameter must be large enough to hold the expected output.	**Post-Requisite**	The buffer pointed to by the Copy_pu8Bffr parameter will contain the string representation of the number.

Test Execution Steps:

TC. No	**Inputs**	**Expected Output**	**Test Result**
1	100, 2, '+', '1', '0', '0'	The buffer should contain the string "100".	Pass
2	-100, 2, '-', '1', '0'	The buffer should contain the string "-10".	Pass
3	0, 2, '0', '0'	The buffer should contain the string "0".	Pass
4	1, 1, '1'	The buffer should contain the string "1".	Pass
5	-1, 1, '-', '1'	The buffer should contain the string "-1".	Pass

6	999, 3, '9', '9', '9'	The buffer should contain the string "999".	Pass
7	-999, 3, '-', '9', '9', '9'	The buffer should contain the string "-999".	Pass
8	12345, 5, '1', '2', '3', '4', '5'	The buffer should contain the string "12345".	Pass
9	-12345, 5, '-', '1', '2', '3', '4', '5'	The buffer should contain the string "-12345".	Pass

Test Case ID	s32ToInteger(u8 * Copy_pu8Bffr, u8 Copy_u8DigitsNum)		
Test Case Description	converts a string to a signed integer. The string is expected to be in the form of a number with a maximum of Copy_u8DigitsNum digits. The integer is returned as the function's output.	**Test Priority**	High
Pre-Requisite	The string must be in the form of a number with a maximum of Copy_u8DigitsNum digits. The string must not contain any non-numeric characters.	**Post-Requisite**	The function will return the integer value represented by the string.

Test Execution Steps:

TC.	Inputs	Expected Output	Test Result

No			
1	'1', '2', '3'	123	Pass
2	'-', '1', '2', '3'	-123	Pass
3	'0'	0	Pass
4	'1', '0'	10	Pass
5	'-', '1', '0'	-10	Pass
6	'9', '9', '9'	999	Pass
7	'-', '9', '9', '9'	-999	Pass
8	'1', '2', '3', '4', '5'	12345	Pass
9	'-', '1', '2', '3', '4', '5'	12345	Pass

Test Case ID	COM_vidSendToRaspBerryPi(tstrRaspberryPiMsg Copy_strMsg, tenuMsgType Copy_enuMsgType)		
Test Case Description	sends a message to the Raspberry Pi. The message is formatted as a string that starts with the characters "!!SEND@" and ends with the characters "!!@". The message body contains the sensor readings and motor RPMs, depending on the message type.	**Test Priority**	High
Pre-Requisite	The Raspberry Pi must be connected to the I2C bus. The Raspberry Pi must be running a program that can receive messages from the microcontroller.	**Post-Requisite**	The Raspberry Pi should receive the message and process it

			accordingly.

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result
1	SENSORS_COMM, tstrRaspberryPiMsg with valid values	The Raspberry Pi should receive the sensor readings.	Pass
2	MOTION_COMM, tstrRaspberryPiMsg with valid values	The Raspberry Pi should receive the motor RPMs.	Pass
3	SENSORS_COMM, tstrRaspberryPiMsg with invalid values	The Raspberry Pi should not receive the sensor readings.	Pass
4	MOTION_COMM, tstrRaspberryPiMsg with invalid values	The Raspberry Pi should not receive the motor RPMs.	Pass

Test Case ID	COM_vidRecFromRaspBerryPi(tstrStmMsg *Copy_pstrMsg)		
Test Case Description	receives a message from the Raspberry Pi. The message is formatted as a string that starts with the characters "!!REC@" and ends with the characters "!!@". The message body contains the motor speeds, separated by the character "@".	**Test Priority**	
Pre-Requisite	The Raspberry Pi must be connected to the I2C bus.	**Post-Requisite**	

	The Raspberry Pi must be running a program that can send messages to the microcontroller.		

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result	
1	Valid message	The function should correctly decode the message and store the motor speeds in the tstrStmMsg structure.	Pass	
2	Invalid message	The function should not decode the message and the tstrStmMsg structure should be unchanged.	Pass	
3	Message with corrupted data	The function should not decode the message and the tstrStmMsg structure should be unchanged.	Pass	

Unit testing for ROS:

Test Case ID	ROS Across Machine Communication – Communication test		
Test Case Description	Test That one ROS master runs on a machine and another ROS Node runs on a separate machine	Test Priority	High

Prerequisite	Both devices are connected on the same network. Both devices' IP addresses are known. - Master device exported as ROS_MASTER on both machines. - A functioning ROS node	**Post-Requisite**	The other machine should run ROS from the master device with Ros parameter server registered on the master device's IP. The node topic should be visible to both machine as well as data published in this topic

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result
1	- Both Machines connected to the same network. - The Master machine's IP address should be exported "ROS_MASTER_IP=:http//mastter's_ip_address":1133 - Node on the other machine is launched	The launched node registers to the ROS Master	**Pass**
2	Both Machines are not connected to the same network.	No node registration from the other machine.	**Pass**

3	Master IP address isn't exported on the other machines.	No node registration from the other machine.	**Pass**

Test Case ID	**ROS Across Machine Communication- Topic Publishing and Subscribing**		
Test Case Description	Test That one ROS master runs on a machine and another ROS Node runs on a separate machine and data is exchanged between them efficiently.	**Test Priority**	**High**
Prerequisite	Both devices are connected on the same network. Both devices' IP addresses are known. Master device exported as ROS_MASTER on both machines. A functioning ROS node	**Post-Requisite**	The other machine should run ROS from the master device with Ros parameter server registered on the master device's IP. The node topic should be visible to both machine as well as data published in this topic
Test Execution Steps:			
TC. No	**Inputs**	**Expected**	**Test**

		Output	**Result**
1	- Both Machines connected to the same network. - The Master machine's IP address should be exported "ROS_MASTER_IP=:http//mastter's_ip_address":1133 - Node on the other machine is launched. - Node publishes on topic.	Master Machine can see the topic and view the data published on it correctly.	**Pass**
2	- Both Machines are not connected to the same network. - The Master machine's IP address not exported - Node on the other machine is launched. - Node publishes on topic.	Master machine can't see the topic and can't view published data.	**Pass**
3	Both Machines are connected to the same network but no Internet connection.	The master can see the node but can't receive published data.	**Pass**

4	- Both Machines connected to the same network. - The Master machine's IP address should be exported "ROS_MASTER_IP=:http//mastter's_ip_address":1133 - Node on the other machine is launched. - Node publishes on topic. - Second Node is launched. -Second Node publishes on topic	Each topic receives its data without running any interference .	**Pass**

Test Case ID	**ROS Across Machine Communication- Performance and Scalability:**		
Test Case Description	Test the performance and scalability of ROS across machines by simulating a high load scenario.	**Test Priority**	**High**
Prerequisite	multiple devices are connected on the same network. all devices IP addresses are known. Master device exported as ROS_MASTER on all machines. 7 functioning ROS nodes	**Post-Requisite**	The other machine should run ROS from the master device with Ros parameter server registered on the master device's IP. The node topic should be visible to both machines as

			well as data published in this topic with minimum delay and no data loss.
Test Execution Steps:			
TC. No	**Inputs**	**Expected Output**	**Test Result**
1	- All Machines connected to the same network. - The Master machine's IP address should be exported "ROS_MASTER_IP=:http//mastter's_ip_address":1133 - 7 Nodes on the other machine are launched. - Publish rate is set to 5 MHz - 7 Nodes publish on topic.	Master Machine can see all the topics and view the data published on it correctly with minimum delay.	**Pass**

2	- All Machines connected to the same network. - The Master machine's IP address should be exported "ROS_MASTER_IP=:http//mastter's_ip_address":1133 - 7 Nodes on the other machine are launched. - Publish rate is set to 10 MHz - 7 Nodes publish on topic.	Master Machine can see all the topics and view the data published on it correctly with minimum delay.	**Pass**

Test Case ID	ROS Control- Controller Startup		
Test Case Description	- Add appropriate controllers in the control package launch file. - Test that the controller can be initialized and started successfully. - Use test inputs to generate commands representing various control scenarios (e.g., position, velocity, or effort commands).	**Test Priority**	High

Pre-Requisite	-Setup Master communication. -Run control nodes on the machine.	**Post-Requisite**	Controllers should start and load all the parameters successfully.

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result
1	Load controllers in the launch file with setting all the parameters.	Controller Initialized and ran successfully.	Pass
2	Load controllers and leave it without usage for more than an hour.	Controller should function normally.	Fail

Test Case ID	ROS Control- Controller

	Functionality		
Test Case Description	- Add appropriate controllers in the control package launch file. - Test that the controller can be initialized and started successfully. - Test that controllers send and receive commands correctly.	**Test Priority**	High
Pre-Requisite	-Setup Master communication. -Run control nodes on the machine. - Setup up joystick to input effort commands to controller. -launch controller.	**Post-Requisite**	Controllers should start successfully. And publish control commands based on input data.

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result
1	Load controllers in the launch file with setting all the	The controllers interpret the commands correctly and extract the relevant control information	Pass

	parameters. Setup the joystick and make it send the control command to the controller.		
2	Input known parameters for known model and compare the results with the expected behavior for this model, RRBOT model was used.	The output commands for the controller should be the same as the known model output.	Pass

Test Case ID	ROS Communication - Publisher		
Test Case Description	- Create a test case to initialize the publisher with the appropriate topic and message type. - Publish a test message to the topic. - Use a subscriber to receive the published message. - Verify that the received message matches the expected values. - Ensure that the publisher publishes messages without any errors or exceptions.	**Test Priority**	High
Pre-Requisite	Have a running ROS core	**Post-Requisite**	Publisher should be initialized successfully. Messages published should be correct without any error or exceptions

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result

1	Launch Publisher.	The publisher node should be listed in the network as ros node. And the topic to be published on should appear on ros topics.	Pass
2	Publish "Hello World" on topic "/Message" at rate 10 Hz	The message is received at the correct rate and without any errors.	Pass

Test Case ID	ROS Communication - Subscriber		
Test Case Description	- Initialize the subscriber with the appropriate topic and message type. - Publish a test message to the topic. - Verify that the subscriber receives the published message. - Check that the received message matches the expected values. - Ensure that the subscriber can handle multiple messages	**Test Priority**	High

	and updates correctly.		
Prerequisite	Have a running ROS core and publisher node publishing at topic "/Message"	**Post-Requisite**	Publisher should be initialized successfully. Messages from topics subscribed to should be correct without any error or exceptions

Test Execution Steps:

TC. No	**Inputs**	**Expected Output**	**Test Result**
1	Launch Publisher. Launch Subscriber. Set publisher rate at 10 Hz. Set subscriber rate at 10 Hz. Publish "Hello world" on topic "/message".	Data read from the publisher should be "Hello World" and received at 10 Hz rate.	Pass

	Subscribe to topic "/message" Read data from the topic.		
2	Launch Publisher. Launch Subscriber. Set publisher rate at 10 Hz. Set subscriber rate at 10 Hz. Publish numbers from 1 to 10 on topic "/message". Subscribe to topic "/Message" Read data from the	The message is received at the correct rate and without any errors.	Pass

	topic.		

Test Case ID	ROS Communication - Message Queue		
Test Case Description	- Create a test case to test message queuing and buffering. - Publish multiple messages to the topic at a faster rate than the subscriber can process. - Verify that the subscriber can handle the incoming messages and process them in the correct order. - Check that the subscriber's message queue doesn't overflow or lose messages. - Ensure that the subscriber can catch up and process all queued messages even if there is a backlog.	**Test Priority**	High

Pre-Requisite	Have a running ROS core Publisher and Subscriber nodes running and publishing on topic "/messages"	**Post-Requisite**	At different publish rates messages should be queued without any data loss.

Test Execution Steps:

TC. No	**Inputs**	**Expected Output**	**Test Result**
1	Launch Publisher. Launch Subscriber. Set publisher rate at 10 Hz. Set subscriber rate at 5 Hz. Publish numbers from 1 to 10 on topic "/message". Subscribe to topic	All numbers should be received on the topic and subscribed to at 5 Hz rate without any message loss, incoming data should be Queued and buffered until subscribed to.	Pass

	"/Message" Read data from the topic.		
2	Launch Publisher. Launch Subscriber. Set publisher rate at 10 Hz. Set subscriber rate at 5 Hz. Publish numbers from 1 to 100 on topic "/message". Subscribe to topic "/Message" Read data from the topic.	Subscribers Queue shouldn't overflow.	Pass

Test Case ID	ROS Communication - Subscriber Callback function		
Test Case Description	- Initialize the subscriber with the appropriate topic and message type and callback function. - Publish a test message to the topic. - Verify that the subscriber receives the published message. -Let the callback function print the received data on the terminal. - Check that the received message matches the expected values. - Ensure that the subscriber can handle multiple messages and updates correctly.	**Test Priority**	High

Prerequisite	Have a running ROS core and publisher node publishing at topic "/Message"	**Post-Requisite**	Publisher should be initialized successfully. Messages from topics subscribed to should be correct without any error or exceptions and printed on the terminal.

Test Execution Steps:

TC. No	**Inputs**	**Expected Output**	**Test Result**
1	Launch Publisher. Launch Subscriber with a callback function to print received data. Set publisher rate at 10 Hz. Set subscriber rate at 10 Hz.	Data read from the publisher should be printed on the terminal as "Hello World" and received at 10 Hz rate.	Pass

	Publish "Hello world" on topic "/message". Subscribe to topic "/message" Read data from the topic.		
2	Launch Publisher. Launch Subscriber with a call back function to multiply received data by 10 and print it on terminal.. Set publisher rate at 10 Hz. Set subscriber	The numbers from "10 to 100" are printed on the terminal.	Pass

	rate at 10 Hz. Publish numbers from 1 to 10 on topic "/message". Subscribe to topic "/Message" Read data from the topic.		

Unit testing for Unity:

Test Case ID	Unity ROS TCP connection		
Test Case Description	start a connection between ros and unity	**Test Priority**	High
Pre-Requisite	ROS system should be configured and installed unity ros tcp connection should be imported roslaunch of ros tcp package valid Master IP and port parameters passed	**Post-Requisite**	Unity published and subscribed topic should be registered in ros unity nodes

			should be visible in ros system

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result
1	Valid IP and port	The connection should be established successfully	Pass
2	Invalid IP or ports	The connection should not be established successfully	Pass
3	configure ROS in unity on ROS1 while running tcp node on ROS 1	The connection should print error of no master registered .	Pass

Test Case ID	Unity-ROS navigation		
Test Case Description	receive a navigation message to move robot	**Test Priority**	
Pre-Requisite	ROS system should be configured and installed a connection between ros and unity should be established	**Post-Requisite**	

Test Execution Steps:			
TC. No	**Inputs**	**Expected Output**	**Test Result**
1	valid move twist message	the robot moves with the direction and speed of the message	Pass
2	Invalid movement due to an obstacle	the robot move collide with obstacle	Pass
3	invalid message type	the robot does not move and a message type error arise	Pass

Test Case ID	Unity-ROS data publishing	
Test Case Description	publish simulated sensor data	**Test Priority**
Pre-Requisite	ROS system should be configured and installed a connection between ros and unity should be established a visualization node should subscribe to sensor data in ROS	**Post-Requisite**

Test Execution Steps:			
TC. No	**Inputs**	**Expected Output**	**Test Result**
1	collect valid data and publish	data should be visualized on ros visualization node	Pass

2	Invalid data collection	data is published but cannot be visualized	Pass
3	invalid publishing topics	data cannot be published	Pass

6.3 Integration Testing

Integration testing involves testing how different units of code interact with each other. Integration tests are typically written by the QA team, and they are used to ensure that the different components of the software work together properly.

The plan for integration testing typically includes a list of all the units of code that need to be integrated, as well as the expected results of each integration test.

The strategy for integration testing typically involves using a test harness to automate the testing process.

Test Plan

- Test Objective: To test how individual units of code interact with each other to ensure that they work together correctly.
- Test Scope: All units of code that are critical to the system.
- Test Methods: Test harness, such as Google Test.
- Expected Results: All units of code should work together correctly.
- Test Procedures:
    - Identify all the units of code that need to be integrated.
    - Create a test harness that integrates the units of code.
    - Run the test harness and verify that it works correctly.

- If the test harness fails, investigate the cause of the failure and fix the code.

Integration testing for Embedded Systems:

for DataAcquistion module:

The DataAcquisition module depends on the following modules:

1. Sensing: This module provides functions to collect sensor data from the MPU sensors and other sensors.

2. Communication: This module provides functions to send and receive data to and from the Raspberry Pi.

The DataAcquisition module interacts with these modules to collect sensor data, send it to the Raspberry Pi. Therefore, the DataAcquisition module's functionality is closely tied to the functionality of these modules, and any changes made to these modules can potentially affect the DataAcquisition module's behavior.

Test Case ID	DAQ_vidInit(void)		
TestCase Description	initializes the variables in the LOC_strMsg structure to 0. The LOC_strMsg structure is a global structure that contains the sensor data.		
Pre-Requisite	A piece of code that runs before the function void DAQ_vidInit(void) is called. set up the environment for the function, such as setting the values of variables or initializing the hardware.	**Post-Requisite**	A piece of code that runs after the function void DAQ_vidInit(void) is called. clean up the environment

			after the function has run, such as releasing resources or closing files.

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result	
1	N/A	All variables should be initialized to 0.	Pass	

Test Case ID	DAQ_vidCollectData(void)
TestCase Description	collects the sensor data from the MPU-6050 and the gas sensor, and then sends the data to the Raspberry Pi.

Pre-Requisite	The Sensing module must be initialized and ready to collect sensor data. The Communication module must be initialized and ready to send and receive data. The necessary variables and data structures, such as af32MPU, u16Temperature, and LOC_strMsg, must be declared and initialized.	**Post-Requisite**	The SEN_vidUpdateSensorsData function from the Sensing module will be called successfully, and the sensor data will be updated. The sensors data from the sensing module will be called and stored successfully. Communication module will be called successfully, and the collected data in LOC_strMsg will be sent to the Raspberry Pi.

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result	
1	af32MPU = {0, 0, 0, 0, 0, 0}; u16Temperature = 0	LOC_strMsg is populated with correct sensor data and sent to the Raspberry Pi.	Pass	
2	af32MPU = {-0.456, 0.789, -1.251, -852, -97, 2}; u16Temperature	LOC_strMsg is populated with correct sensor data and sent to the Raspberry Pi.	Pass	

	= 50		

For RobotMotion module:

The RobotMotion module depends on the following modules:

1. Sensing module - This module provides sensor data to the RobotMotion module, such as encoder data for the left and right motors, which is used to calculate the RPM values for the motors.
2. Communication module - This module provides communication between the RobotMotion module and the Raspberry Pi, allowing the RobotMotion module to receive motor speed commands and send motor RPM values back to the Raspberry Pi.
3. Actuation module - This module is responsible for actuating the left and right motors based on the speed and direction commands received from the Raspberry Pi through the RobotMotion module.

Therefore, the RobotMotion module depends on the Sensing, Communication, and Actuation modules to perform its tasks of handling the motion of the robot.

Test Case ID	RMO_vidInit(void)
TestCase Description	initializing the LOC_strRaspMsg and LOC_strStmMsg structures with default values. These structures are used to store motor RPM and speed data and are used by other functions in the RobotMotion module to control the motion of the robot.

Pre-Requisite	There are no specific prerequisites required to call the RMO_vidInit function.	**Post-Requisite**	The LOC_strRaspMsg.s8LeftMotorRPM and LOC_strRaspMsg.s8RightMotorRPM variables will be set to 0. The LOC_strStmMsg.s8LeftMotorSpeed and LOC_strStmMsg.s8RightMotorSpeed variables will be set to 0. The RobotMotion module will be in a stable state and ready to receive commands from the Raspberry Pi to control the motion of the robot.

Test Execution Steps:

TC. No	**Inputs**	**Expected Output**	**Test Result**	
1	None	LOC_strRaspMsg.s8LeftMotorRPM and LOC_strRaspMsg.s8RightMotorRPM are set to 0, and LOC_strStmMsg.s8LeftMotorSpeed and LOC_strStmMsg.s8RightMotorSpeed are set to 0.	Pass	

Test Case ID	RMO_vidInit(void)		
TestCase Description	handling the motion of the robot based on the motor speed commands received from the Raspberry Pi through the LOC_strStmMsg structure. It calculates the direction and speed for the left and right motors, actuates the motors using the ACT_vidActuateMotor function, updates the encoder values using the SEN_vidUpdateEncoders function, and sends back the RPM values of the left and right motors to the Raspberry Pi through the LOC_strRaspMsg structure.		
Pre-Requisite	The LOC_strStmMsg structure must contain valid motor speed commands received from the Raspberry Pi. The Communication, Actuation, and Sensing modules must be operational. The hardware components, such as the Raspberry Pi, motors, and encoders, must be connected and functioning properly.	**Post-Requisite**	The left and right motors will move in the correct direction and speed based on the motor speed commands received from the Raspberry Pi through the LOC_strStmMsg structure. The encoder values for the left and right motors will be updated using the SEN_vidUpdateEncoders function. The RPM values of the left and right motors will be sent back to the Raspberry Pi through the

			LOC_strRaspMsg structure.
Test Execution Steps:			
TC. No	**Inputs**	**Expected Output**	**Test Result**
1	LOC_strStmMsg with positive motor speed values	Left and right motors will move forward at the specified speed, and LOC_strRaspMsg will be sent back to the Raspberry Pi with the correct RPM values.	Pass

2	LOC_strStmMsg with negative motor speed values	Left and right motors will move backward at the specified speed, and LOC_strRaspMsg will be sent back to the Raspberry Pi with the correct RPM values.	Pass
3	LOC_strStmMsg with mixed positive and negative motor speed values	Left and right motors will move in the correct direction at the specified speed, and LOC_strRaspMsg will be sent back to the Raspberry Pi with the correct RPM values.	Pass

Integration testing for ROS:

For ROS Integration testing ROS nodes had to interact with each other and with other system components such as I2C on raspberry pi, and with the control station.

1- I2C bus - ROS integration:

Test Case ID	ROS - I2C on stm32 linking receiving only.

Test Case Description	- Verify that ROS nodes can establish a successful connection with the I2C bus and communicate with I2C devices. - Send test data through ROS messages and verify that the data is correctly transmitted and received by the I2C device.	**Test Priority**	High
Pre-Requisite	Running Roscore Have an i2c device (stm32) transmitting data 64 bytes. Publisher and Subscriber with a callback function to show received data.	**Post-Requisite**	Data should be received from stm32 successfully and sent to it

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result
1	-Run ros core -Give permission for the i2c script to access i2c bus on raspberry pi. - send 64 bytes of data through stm32.	Data read from i2c should be correct.	Pass

2	-Run ros core -Give permission for the i2c script to access i2c bus on raspberry pi. - send 60 bytes of data through stm32.	Ros node will compensate for lost data with zeros.	Fail
3	-Run ros core -Give permission for the i2c script to access i2c bus on raspberry pi. - send 60 bytes of data through stm32.	Ros node will be stuck waiting for the rest of the data.	Pass

Test Case ID	ROS - I2C on stm32 linking sending only.		
Test Case Description	- Verify that ROS nodes can establish a successful connection with the I2C bus and communicate with I2C devices. - Send test data through ROS	**Test Priority**	High

	messages and verify that the data is correctly transmitted and received by the I2C device.		
Pre-Requisite	Running Roscore Have an i2c device (stm32) waiting to receive data 21 bytes. Publisher and Subscriber with a callback function to show received data.	**Post-Requisite**	Data should be sent to from stm32 successfully and sent to it

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result
1	-Run ros core -Give permission for the i2c script to access i2c bus on raspberry pi. - Send 21 bytes of data.	Stm32 should replay with acknowledge indicating data received successfully.	Pass
2	-Run ros core -Give permission for the i2c script to access i2c bus on	Stm32 should replay with not acknowledge indicating data received wasn't complete.	Fail

	raspberry pi. - Send 20 bytes of data.		

Test Case ID	ROS - I2C on stm32 linking sending and receiving sequence.		
Test Case Description	- Verify that ROS nodes can establish a successful connection with the I2C bus and communicate with I2C devices. - Send test data through ROS messages and verify that the data is correctly transmitted and received by the I2C device.	**Test Priority**	High
Prerequisite	Running Roscore Have an i2c device (stm32) transmitting data 64 bytes. Have an i2c device (stm32) waiting to receive data 21 bytes. Publisher and Subscriber with a callback function to show received data.	**Post-Requisite**	Data should be received from stm32 successfully and sent to it
Test Execution Steps:			

TC. No	**Inputs**	**Expected Output**	**Test Result**

1	-Run ros core -Give permission for the i2c script to access i2c bus on raspberry pi. - Run ros node with i2c data collection script in the main function. Stm32 sends sensor data on the i2c bus.	Data read from i2c should be read successfully.	Pass
2	-Run ros core -Give permission for the i2c script to access i2c bus on raspberry pi. - Run publisher and subscriber nodes with i2c data collection script in the main function. Stm32 sends 64 bytes of sensor data on the i2c bus.	64 bytes Data received from i2c bus should be published on topic data and subscribed to successfully without any data loss.	Pass
3	-Run ros core -Give permission for the i2c script to access i2c bus on raspberry pi. - Run publisher and subscriber nodes with i2c data sending script in the main function. Stm32 receives 21 bytes data on the	Data should be sent to stm32 and verified by an acknowledge form stm32.	Pass

	i2c bus.		
4	-Run ros core -Give permission for the i2c script to access i2c bus on raspberry pi. - Run publisher and subscriber nodes with i2c data sending and receiving sequence agreed on by both devices script in the main function. Stm32 receives 21 bytes data on the i2c bus.	The data is transmitted and received successfully on both ends.	Pass

2- ROS controller - ROS publisher subscriber Integration:

Test Case ID	ROS Control- ROS Publisher linking		
Test Case Description	- Add appropriate controllers in the control package launch file. - Test that the controller can be initialized and started successfully.	**Test Priority**	High

	- Test that controllers send and receive commands correctly. - ROS nodes should be able to publish encoder data to the controller.		
Pre-Requisite	-Setup Master communication. -Run control nodes on the machine. - Setup a publisher node to publish effort commands to the controller on topic"/encoders". -launch controller. -launch publisher node.	**Post-Requisite**	Controllers should start successfully. And publish control commands based on data from the publisher.

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result
1	Encoder data on topic"/encoders" isn't published.	Controller runs and assumes missing parameters 0.	Pass

2	Encoder data published.	Controller runs and output effort commands successfully.	Pass

Test Case ID	ROS Control- ROS Publisher Subscriber linking		
Test Case Description	- Add appropriate controllers in the control package launch file. - Test that the controller can be initialized and started successfully. - Test that controllers send and receive commands correctly. - ROS nodes should be able to publish encoder data to the controller. ROS Node should be able to subscribe to effort data. Callback function on subscriber should print effort commands.	**Test Priority**	High
Pre-Requisite	-Setup Master communication. -Run control nodes on the machine publishing controller commands on topic	**Post-Requisite**	Controllers should start successfully. And publish control commands based on data

	"/effort_commands". - Setup a publisher node to publish effort commands to the controller on topic"/encoders". -launch controller. -launch publisher node. -launch subscriber node to subscribe on topic "/effort_commands". - callback function on the subscriber should print effort data.		from the publisher. Control commands are subscribed to and printed on screen.

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result
1	Encoder data on topic"/encoders" isn't published.	Controller runs and assumes missing parameters 0.	Pass
2	Encoder data published.	Controller runs and effort commands are subscribed to successfully.	Pass

Test Case ID	ROS Control- ROS Publisher

	subscriber Data Synchronization.		
Test Case Description	- Add appropriate controllers in the control package launch file. - Test that the controller can be initialized and started successfully. - Test that controllers send and receive commands correctly. - ROS nodes should be able to publish encoder data to the controller. - Test each node at different rates and its interaction with other node	**Test Priority**	Moderate
Pre-Requisite	-Setup Master communication. -Run control nodes on the machine publishing controller commands on topic "/effort_commands". - Setup a publisher node to publish effort commands to the controller on topic"/encoders".	**Post-Requisite**	Process should run successfully and mismatched rates shouldn't affect data integrity.

	-launch controller. -launch publisher node. -launch subscriber node to subscribe on topic "/effort_commands". - callback function on the subscriber should print effort data.		

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result
1	Publisher rate 10 Hz. Control rate 10 Hz. Subscriber rate 10 Hz.	All data is exchanged in real time and output is correct.	Pass
2	Publisher rate 10 Hz. Control rate 10 Hz. Subscriber rate 5 Hz.	A delay in outputting effort commands, but no losses in data.	Pass
3	Publisher rate 10 Hz. Control rate 5 Hz.	controller buffers data from publisher, but controller and subscriber output data	Pass

	Subscriber rate 5 Hz.	simultaneously.	
4	Publisher rate 5 Hz. Control rate 5 Hz. Subscriber rate 10 Hz.	subscriber takes the same data twice, but controller and publisher run simultaneously.	Pass

6.4 System Testing

System testing involves testing the entire software system as a whole. System tests are typically written by the QA team, and they are used to ensure that the software meets all of the requirements.

The plan for system testing typically includes a list of all the system requirements, as well as the expected results of each system test.

The strategy for system testing typically involves using a test plan to document the testing process.

Test Plan

- Test Objective: To test the entire system to ensure that it meets its requirements.
- Test Scope: All system requirements.
- Test Methods: Manual testing, automated testing, and stress testing.
- Expected Results: The system should meet all of its requirements.
- Test Procedures:
    - Create a test plan that documents all of the system tests that need to be performed.
    - Perform the system tests manually and/or with an automated testing tool.

- Verify that the system meets all of its requirements.
- If the system does not meet its requirements, investigate the cause of the failure and fix the code.

System testing for Embedded Systems:

OS Testing:

The OS includes two tasks, one for data acquisition and one for robot motion control. The OS_vidInit function initializes the OS and starts the SYSTICK timer with a tick interval of 2 milliseconds. The OS_vidScheduler function implements the core logic of the OS scheduler, which executes the tasks based on their periodicity and state. The OS_vidSystickISR function is the interrupt service routine for the SYSTICK timer, which sets the OS state to OS_EXECUTING to indicate that the scheduler should run.

OS testing is an essential aspect of software development for embedded systems, as it helps to ensure the reliability, stability, and performance of the OS and the system it is designed for.

Test Case ID	OS_vidInit(void)
TestCase Description	initializes the operating system (OS) and starts the system timer (SYSTICK) with a specified tick interval. It also initializes the periodicity of the tasks and sets the OS state to OS_INIT.

Pre-Requisite	defines the number of tasks (OS_NUM_OF_TASKS) and the tick interval in milliseconds (OS_TICK_MILLISECOND) for the SYSTICK timer. The DataAcquisition.h and RobotMotion.h files should be included, which define the task functions for data acquisition and robot motion control, respectively.	**Post-Requisite**	The SYSTICK timer is started with the tick interval defined in OS_TICK_MILLISECOND. The periodicity of all tasks is initialized based on the values defined in GLOB_astrTasks and LOC_u8Periodicity arrays. The OS state is set to OS_INIT, and the OS scheduler can be executed to start executing the tasks.

Test Execution Steps:

TC. No	**Inputs**	**Expected Output**	**Test Result**
1	None	OS state is initialized to OS_INIT	Pass
2	OS_TICK_MILLISECOND = 1, OS_NUM_OF_TASKS = 1, period = 1	SYSTICK timer is started with a tick interval of 1 millisecond and periodicity of the task is set to	Pass

		0	
3	OS_TICK_MILLISECOND = 10, OS_NUM_OF_TASKS = 2, period = {2, 1}	SYSTICK timer is started with a tick interval of 20 milliseconds and periodicity of the tasks are set to {1, 2}	Pass

Test Case ID	OS_vidScheduler(void)
TestCase Description	It checks the OS state and executes the tasks based on their periodicity and state. If the OS state is OS_INIT, it executes all tasks once and sets their state to TASK_READY. If the OS state is OS_EXECUTING, it executes the tasks that have a periodicity of 1 and are ready to be executed, updates their periodicity value, and sets their state to TASK_READY. If the OS state is OS_WAITING, it does not execute any tasks.

Pre-Requisite	defines the number of tasks (OS_NUM_OF_TASKS) and the tick interval in milliseconds (OS_TICK_MILLISECOND) for the SYSTICK timer. The DataAcquisition.h and RobotMotion.h files should be included	**Post-Requisite**	The OS state is updated based on the execution of the tasks. If the OS state is OS_INIT, it is set to OS_WAITING. If the OS state is OS_EXECUTING, it is set to OS_WAITING. If the OS state is OS_WAITING, it remains unchanged. The periodicity of each task is updated based on its execution and state.

Test Execution Steps:

TC. No	**Inputs**	**Expected Output**	**Test Result**	
1	LOC_enuOSState = OS_INIT, GLOB_astrTasks = {task1, task2}, LOC_u8Periodicity	OS state is set to OS_WAITING. All tasks are executed and their state is set to TASK_READY. periodicity of task1 is set to 1, and periodicity	Pass	

	= {2, 1}	of task2 is set to 2.	
2	LOC_enuOSState = OS_EXECUTING, GLOB_astrTasks = {task1, task2}, LOC_u8Periodicity = {1, 2}	OS state is set to OS_WAITING. task1 is executed and its state is set to TASK_READY. periodicity of task1 is set to 2, and periodicity of task2 is set to 1.	Pass
3	LOC_enuOSState = OS_WAITING, GLOB_astrTasks = {task1, task2}, LOC_u8Periodicity = {0, 1}	OS state is set to OS_WAITING. No tasks are executed. periodicity of task1 is set to -1, and periodicity of task2 is set to 0.	Pass

System testing for All System:

1- Robot with joystick

In this case we had already tested embedded systems communication with ros, and ros across machines with the station, the next step is to test the robot's ability to maneuver manually.

Test Case ID	Robot Movement with joystick

TestCase Description	Test that the robot can be manually controlled using a joystick mounted on the control station.		
Prerequisite	• Initialize system OS on stm32 • Initialize ROS Nodes on the robot:Sensor's_data, differntial_drive_control. • Initialize ROS Master and Nodes on control station: Log_node.	**Post-Requisite**	Robot Moves according to coordinates coming from the joystick.

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result
1	Joystick sends coordinates to move forward	Controller node outputs movement forward commands and sends them to motors.	Pass
2	Joystick sends coordinates to move backward	Controller node outputs movement backward commands and sends them to motors.	Pass

3	Joystick sends coordinates to move right,	Controller node outputs movement right commands and sends them to motors.	Pass
4	Joystick sends coordinates to move left.	Controller node outputs movement forward commands and sends them to motors.	Pass
5	Joystick sends command to stop.	Controller node outputs 0 commands and sends them to motors.	Pass

2- Mapping and localization

Test Case ID	Robot Mapping and localization.
TestCase Description	Test that the robot can maneuver the field, map the environment around it using Kinect and send the map to the station.

Prerequisite	• Initialize system OS on stm32 • Initialize ROS Nodes on the robot:Sensor's_data, differntial_drive_control, launch_slam. • Initialize ROS Master and Nodes on control station: Log_node, and Gmaping.	**Post-Requisite**	The Robot maneuvers the field, builds the map and publishes it to the station.

Test Execution Steps:

TC. No	Inputs	Expected Output	Test Result
1	All nodes start correctly.	Kinect data is sent to the station and a map is built.	Pass
2	launch_slam node doesn't start.	No data is collected and no map is built	Pass

3- Autonomous:

Test Case ID	Autonomous movement.
TestCase	Test that the robot can move autonomously according to the built

Description	map.		
Prerequisite	• Initialize system OS on stm32 • Initialize ROS Nodes on the robot:Sensor's_data, differntial_drive_control, launch_slam, move_base. • Initialize ROS Master and Nodes on control station: Log_node, and Gmaping.	**Post-Requisite**	The Robot can move autonomously in the field.

Test Execution Steps:

TC. No	**Inputs**	**Expected Output**	**Test Result**
1	All nodes start correctly.	Kinect data is sent to the station and a map is built, map is used to plan movement through move_base package.	Pass
2	launch_slam node doesn't start.	No data is collected and no map is built and the robot doesn't move.	Pass

6.5 Conclusion

In conclusion, the testing phase of our project has been crucial in ensuring the quality and functionality of our product. Through rigorous testing methods and careful analysis of the data, we have gained valuable insights into the strengths and limitations of our product. Our testing process has helped us identify and address any defects or issues, ensuring that our product meets the needs and expectations of our users.

Based on our findings, we have identified several key areas for improvement. For instance, our testing has revealed that certain features could be streamlined to improve user experience, and that additional testing may be needed to ensure compatibility with various devices. We have also identified opportunities to enhance the product's security features. Moving forward, we will be addressing these areas of improvement and continuing to refine our product to better meet the needs of our users.

Overall, the testing phase has been integral to the success of our project. Through careful analysis and refinement, we have created a product that is both functional and user-friendly. We believe that our findings and insights will be valuable to future researchers and developers in the field, and we look forward to leveraging this knowledge to continue improving our product.

7. Chapter 7 : Results

7.1 Embedded System

At the beginning of the main function, the system initializes the used modules in MCAL, HAL and APP layers of the system.

```
void main(void)
{

    /* ********************** MCAL LAYER INITIALIZATION ********************** */
    RCC_voidInit();

    RCC_voidEnablePeripheralClock(ADC1);
    RCC_voidEnablePeripheralClock(GPIOA);
    RCC_voidEnablePeripheralClock(GPIOB);
    RCC_voidEnablePeripheralClock(TIMER2);
    RCC_voidEnablePeripheralClock(TIMER3);
    RCC_voidEnablePeripheralClock(TIMER10);
    RCC_voidEnablePeripheralClock(TIMER11);
    RCC_voidEnablePeripheralClock(I2C1);
    RCC_voidEnablePeripheralClock(I2C2);

    GPIO_voidSetPinsConfig();

    TIM_vidInit();
    I2C_vidInit();
    SYSTICK_vidInit();
    /* ************************************************************************ */
```

Figure 28:Embedded System Results 1

```
    /* ********************** HAL LAYER INITIALIZATION ********************** */
    SEN_vidInit();
    /* ************************************************************************ */


    /* ********************** APP LAYER INITIALIZATION ********************** */
    DAQ_vidInit();
    RMO_vidInit();
    /* ************************************************************************ */
```

Figure 29:Embedded System Results2

Then it calls the initializing function of the RTOS to ensure that the system is ready to operate successfully.

```
    /* ********************** OS LAYER INITIALIZATION ********************** */
    OS_vidInit();
    /* ************************************************************************ */
```

The Scheduler of the RTOS is called to allow the two main tasks of the system to operate, so the system begins to work. The two tasks are executed every fixed

system tick of 100ms.

```
case OS_EXECUTING:

    LOC_enuOSState = OS_WAITING;

    for(u8Counter = 0; u8Counter < OS_NUM_OF_TASKS; u8Counter++)
    {
        if( (1 == LOC_u8Periodicity[u8Counter])
              &&
            (TASK_READY == GLOB_astrTasks[u8Counter].enuTaskState))
        {
            GLOB_astrTasks[u8Counter].enuTaskState = TASK_RUNNING;
            GLOB_astrTasks[u8Counter].pTaskFunction();
            GLOB_astrTasks[u8Counter].enuTaskState = TASK_READY;

            LOC_u8Periodicity[u8Counter] = GLOB_astrTasks[u8Counter].u8Periodicity + 1;
        }
    }
```

Figure 30:Embedded System Results3

The Data Acquisition task is responsible for collecting the required data from the sensor and transmitting it to the Raspberry Pi.

```
void DAQ_vidCollectData(void)
{
    f32 af32MPU[6] ={0};
    u16 u16Temperature = 0;

    SEN_vidUpdateSensorsData();

    u16Temperature = SEN_u16GetTemperature();
    u16Temperature = (((f32)u16Temperature)* 0.05);

    LOC_strMsg.u8GasPercentage = SEN_u8GetGasPercentage();
    LOC_strMsg.u8Temperatue = (u8)u16Temperature;
    SEN_vidGetGyroAccel(af32MPU);

    LOC_strMsg.s8GyroX = ((s8)af32MPU[3]);
    LOC_strMsg.s8GyroY = ((s8)af32MPU[4]);
    LOC_strMsg.s8GyroZ = ((s8)af32MPU[5]);
    LOC_strMsg.s16AccelX = ((s16)(af32MPU[0]*1000));
    LOC_strMsg.s16AccelY = ((s16)(af32MPU[1]*1000));
    LOC_strMsg.s16AccelZ = ((s16)(af32MPU[2]*1000));

    COM_vidSendToRaspBerryPi(LOC_strMsg, SENSORS_COMM);

}
```

Figure 31:Embedded System Results4

The Robot Motion task is responsible for receiving the required direction and speed to actuate the motors achieving the required position. It also transmits the feedback of the encoders to allow controlling the speed of motors for the

accuracy of motion.

```
void RMO_vidMotionHandler(void)
{
    tenuMotorDir enuLeftMotorDir, enuRightMotorDir;

    COM_vidRecFromRaspBerryPi(&LOC_strStmMsg);

    if ( LOC_strStmMsg.s8LeftMotorSpeed < 0)
    {
        LOC_strStmMsg.s8LeftMotorSpeed  *= -1;
        enuLeftMotorDir = BACKWARD_ACT;
    }
    else
    {
        enuLeftMotorDir = FORWARD_ACT;
    }
    if ( LOC_strStmMsg.s8RightMotorSpeed < 0)
    {
        LOC_strStmMsg.s8RightMotorSpeed  *= -1;
        enuRightMotorDir = BACKWARD_ACT;
    }
```

Figure 32:Embedded System Results5

```
    else
    {
        enuLeftMotorDir = FORWARD_ACT;
    }
    if ( LOC_strStmMsg.s8RightMotorSpeed < 0)
    {
        LOC_strStmMsg.s8RightMotorSpeed  *= -1;
        enuRightMotorDir = BACKWARD_ACT;
    }
    else
    {
        enuRightMotorDir = FORWARD_ACT;
    }


    ACT_vidActuateValve(LOC_strStmMsg.u8Valve);
    ACT_vidActuateMotor(LEFT_MOTOR, enuLeftMotorDir, LOC_strStmMsg.s8LeftMotorSpeed);
    ACT_vidActuateMotor(RIGHT_MOTOR, enuRightMotorDir, LOC_strStmMsg.s8RightMotorSpeed);

    //SEN_vidUpdateEncoders();
    LOC_strRaspMsg.s8LeftMotorRPM = SEN_u8GetLeftMotorRPM();
    LOC_strRaspMsg.s8RightMotorRPM = SEN_u8GetRightMotorRPM();
```

```
62
63      if(enuLeftMotorDir == BACKWARD_ACT)
64      {
65          LOC_strRaspMsg.s8LeftMotorRPM *= -1;
66      }
67      else
68      {
69      }
70
71      if(enuRightMotorDir == BACKWARD_ACT)
72      {
73          LOC_strRaspMsg.s8RightMotorRPM *= -1;
74      }
75      else
76      {
77      }
78
79      COM_vidSendToRaspBerryPi(LOC_strRaspMsg, MOTION_COMM);
80  }
81
```

Figure 33:Embedded System Results6

7.2 ROS

Based on the specified requirements and design the following results were obtained.

1- ROS across machine:

Having “ubuntu” representing the system on Fire inspection rover and “dodzz” as the control station

From the results it was shown that “ubuntu” was able to register as a ROS node for the master “dodzz” and both machines were able to view the nodes, topics, and services of the other machine clearly.

From the following graph it could be shown that “dodzz” registered as Master with parameter server at IP address “192.168.120”

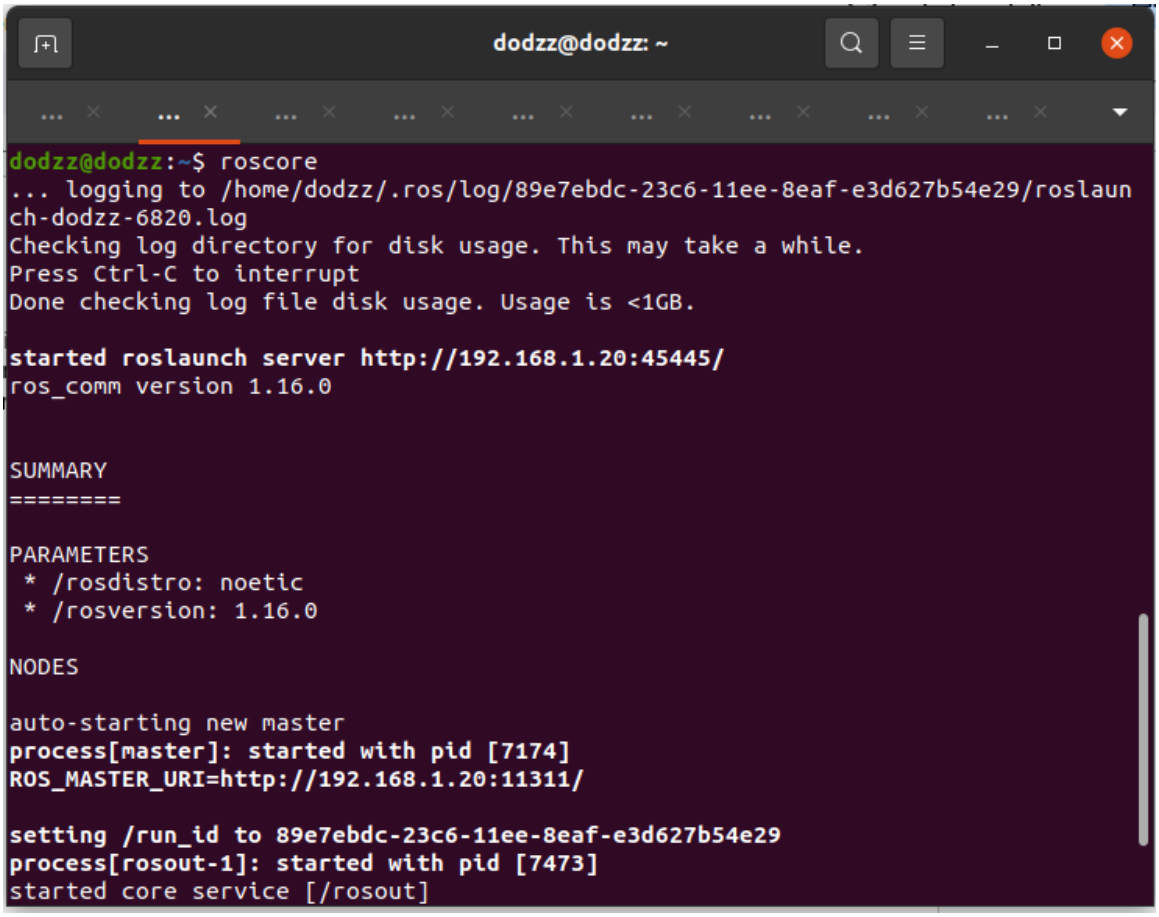

Figure 34:ROS results 1

In the next rqt-graph it could be shown that nodes running on both systems can be shown as one coherent system even it is running on two machines:

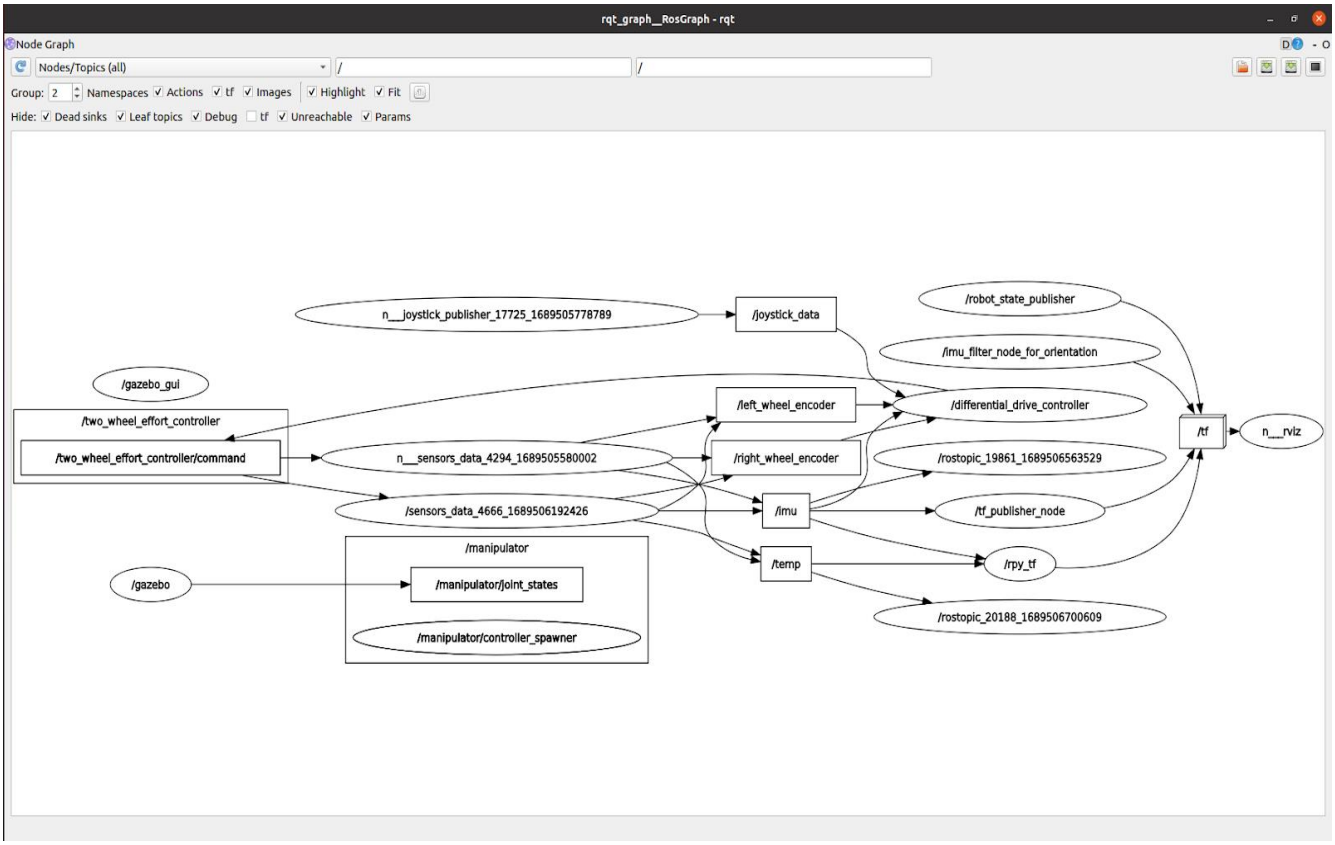

Figure 35:ROS results 2

2- ROS STM32 I2C Communication:

From the results of this section it was shown that ROS could access Raspberry PI I2C bus and receive data transmitted on I2C bus.

SYnchronization between the STM32 and ROS was done using A constant rate clock on the ROS node (10 Hz) and Operating system running on the STM32.

Finally Acknowledge was done using a sequential transmission order as show in

the following:

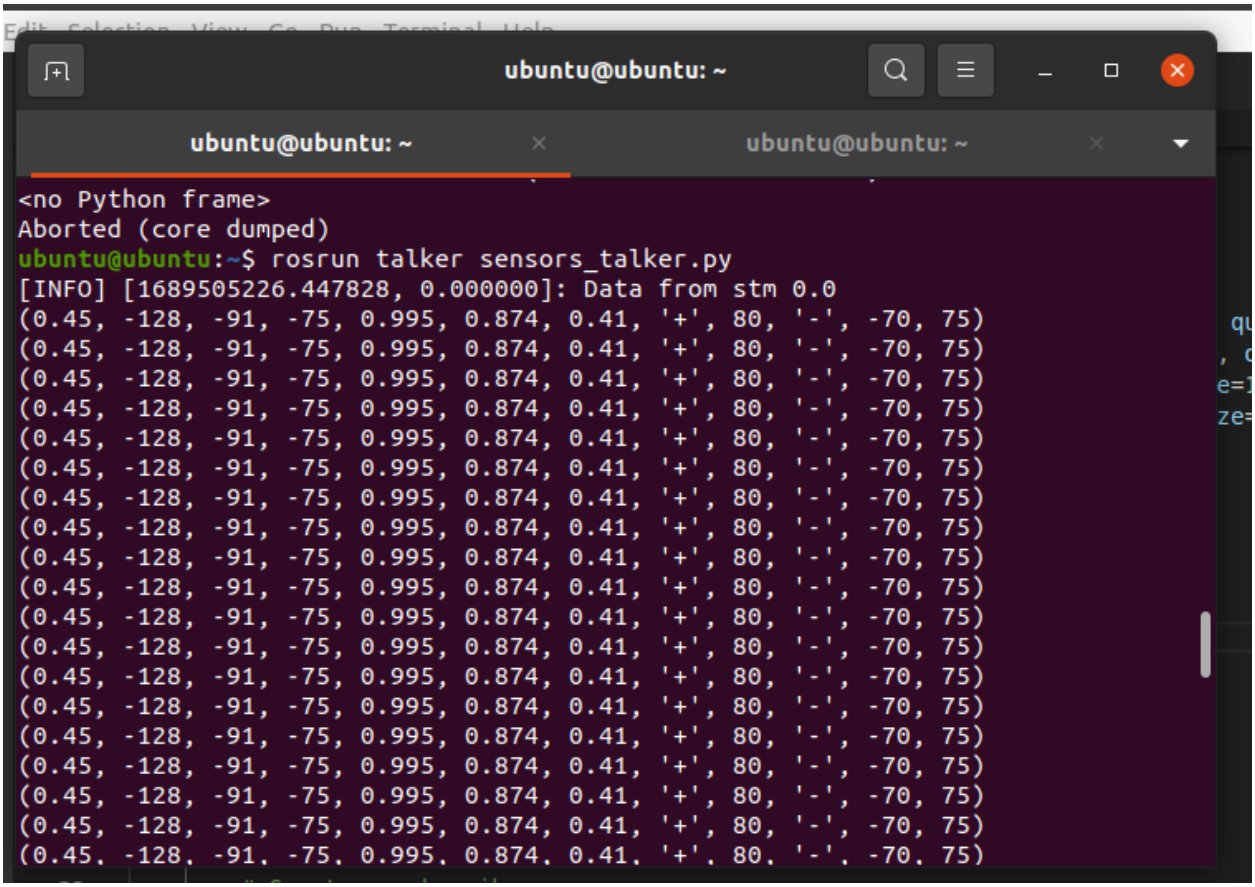

Figure 36:ROS results 3

Each data bit represents sensor reading:

1- MQ7 2- Gyroscope X 3- Gyroscope Y 4- Gyroscope Z

5- Accelerometer x 6-Accelerometer y 7- Accelerometer z8- Right Encoder Direction 9- Right Encoder Value 10- Left Encoder Direction 11- Left Encoder Value

12- Temperature

The following figure shows data coming from the controller and sent to STM32 using ROS Node controlling the I2C bus, with the agreed upon frame between STM32 and the Robot:

M_R: Right Wheel RPM and direction.

M_L: Left Wheel RPM and direction.

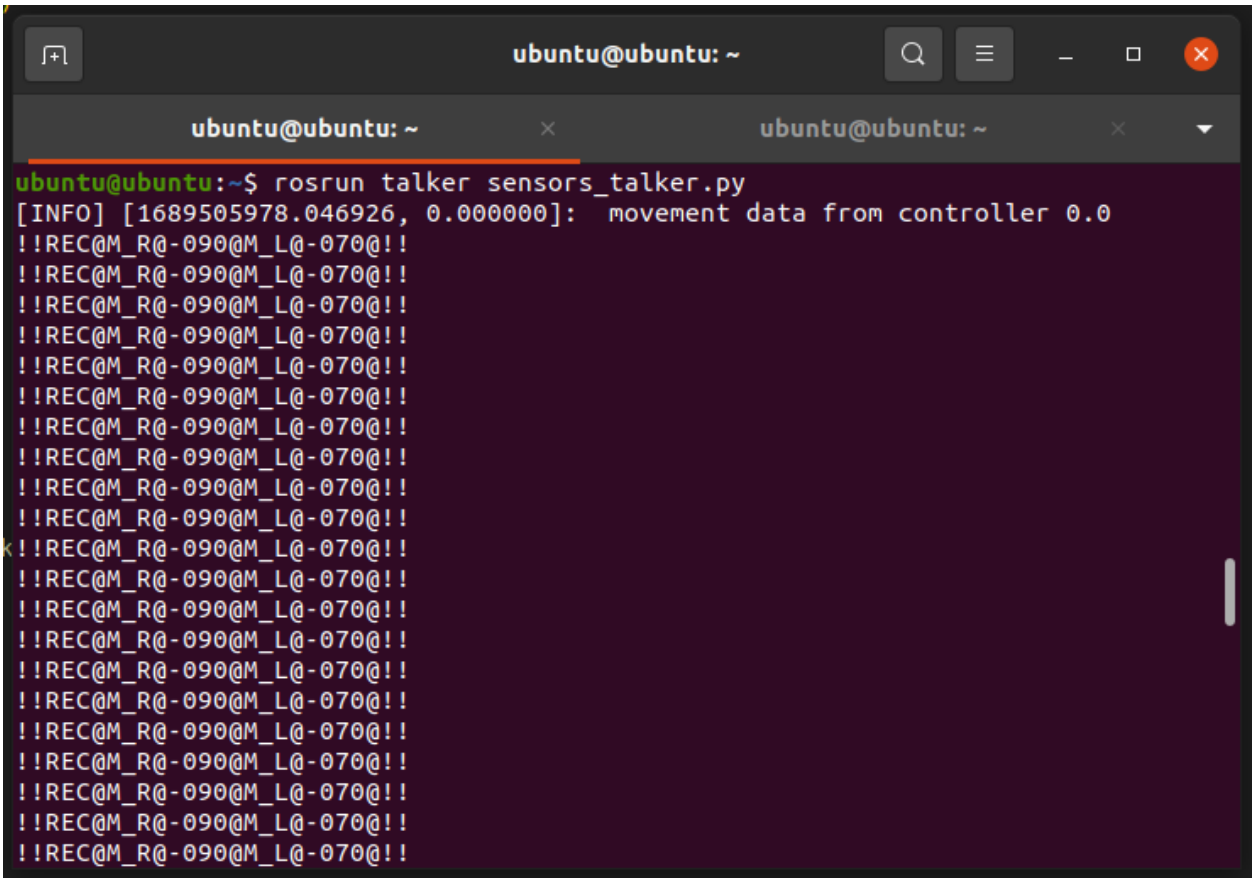

Figure 37:ROS results 4

3- ROS Sensors_data node publishing sensors information on specified topics to be used by all system functions and represented on RVIZ.

For example IMU sensor data on topic “/imu”

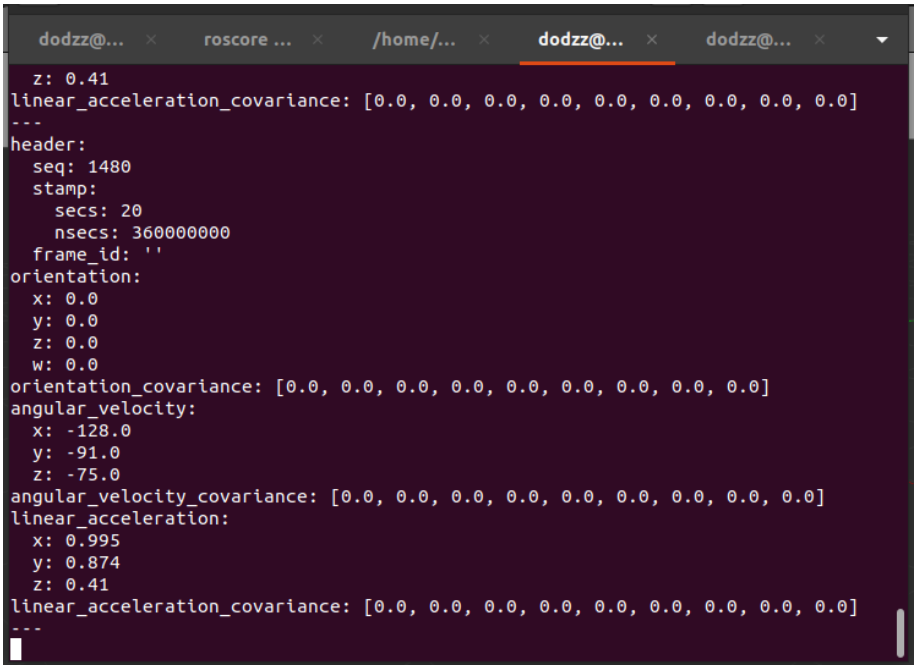

Figure 38:ROS Results 5

And represented on RVIZ:

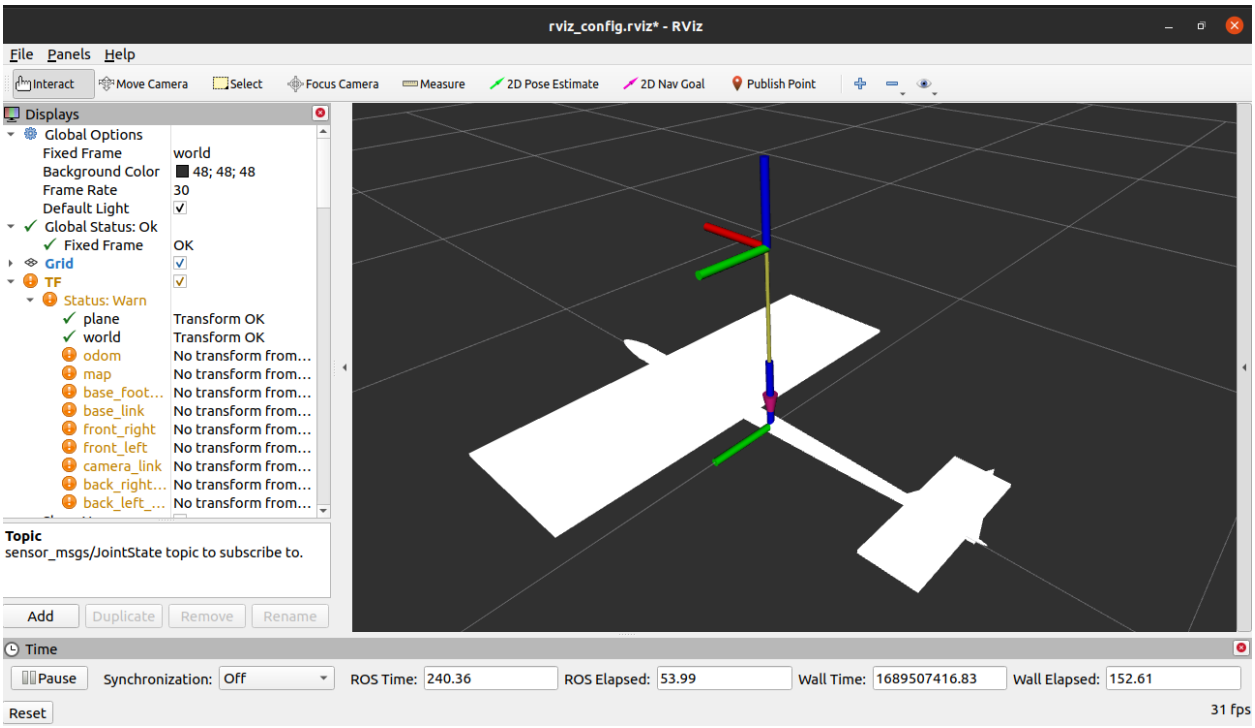

Figure 39:ROS results 6

3- ROS control package:

By running “differential_drive_controller” node and launching 3 controllers:

- ‘right_wheel_effort_control’
- ‘left_Wheel_effort_control’
- ‘joint_state_controller’

The effort controllers receive the joystick data as input and process it to calculate the appropriate control signals or outputs using encoders data from the STM32 on the following topics:

- ‘left_wheel_encoder’
- ‘right_wheel_encoder’

These control signals are then sent to the actuators or control systems responsible for executing the desired actions on the following topics:

- ‘left_wheel_effort_controller/command'
- 'right_wheel_effort_controller/command'
- 'two_wheel_effort_controller/command'

Then the control signals are passed as a frame as shown previously in the I2C communication results part.

The following figure shows as controllers starting on Gazebo simulation:

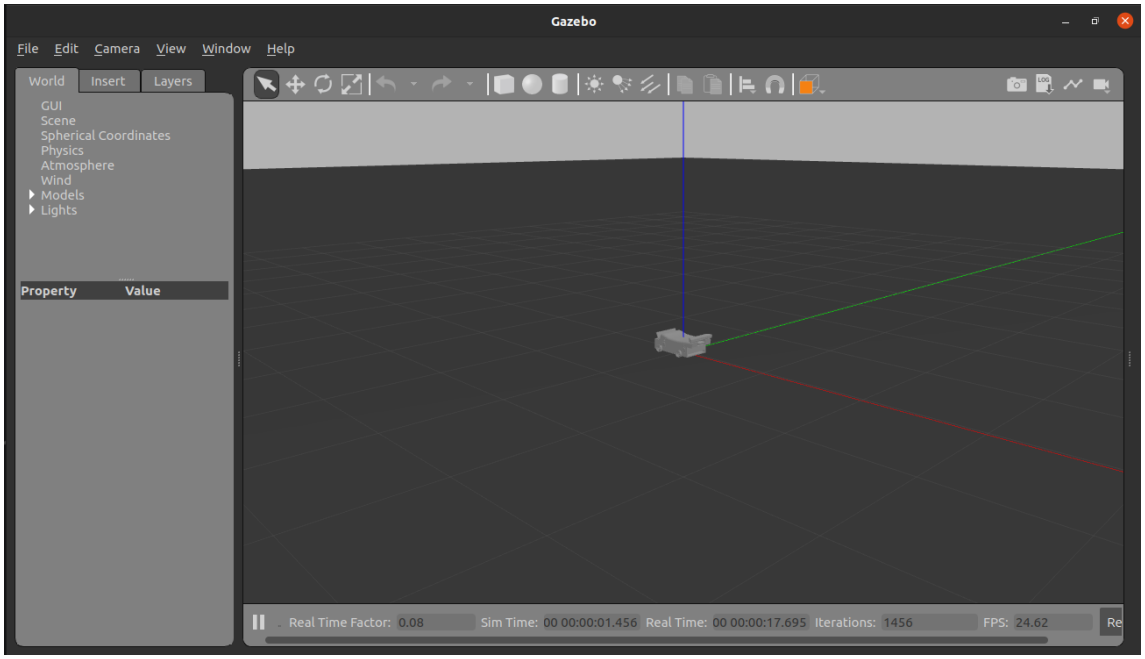

Figure 40:ROS results 7

The following figure show as the two wheels commands are published:

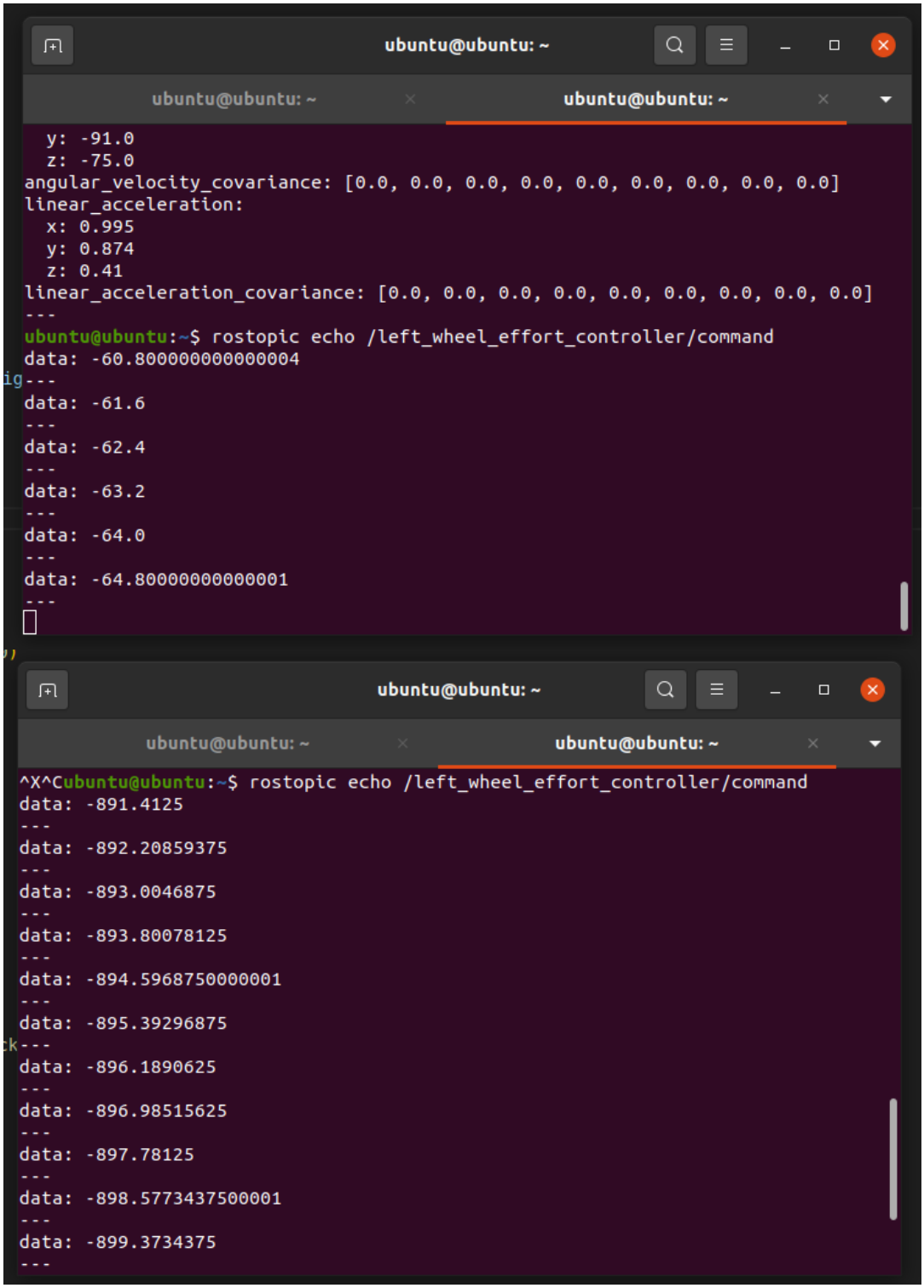

Figure 41:ROS results 8

7.3 Unity

Firstly we simulate the robot in unity to get accurate assumptions of unity physical engine to see how the robot well behave with different colliders and how to properly control its movement

The robot mechanical design

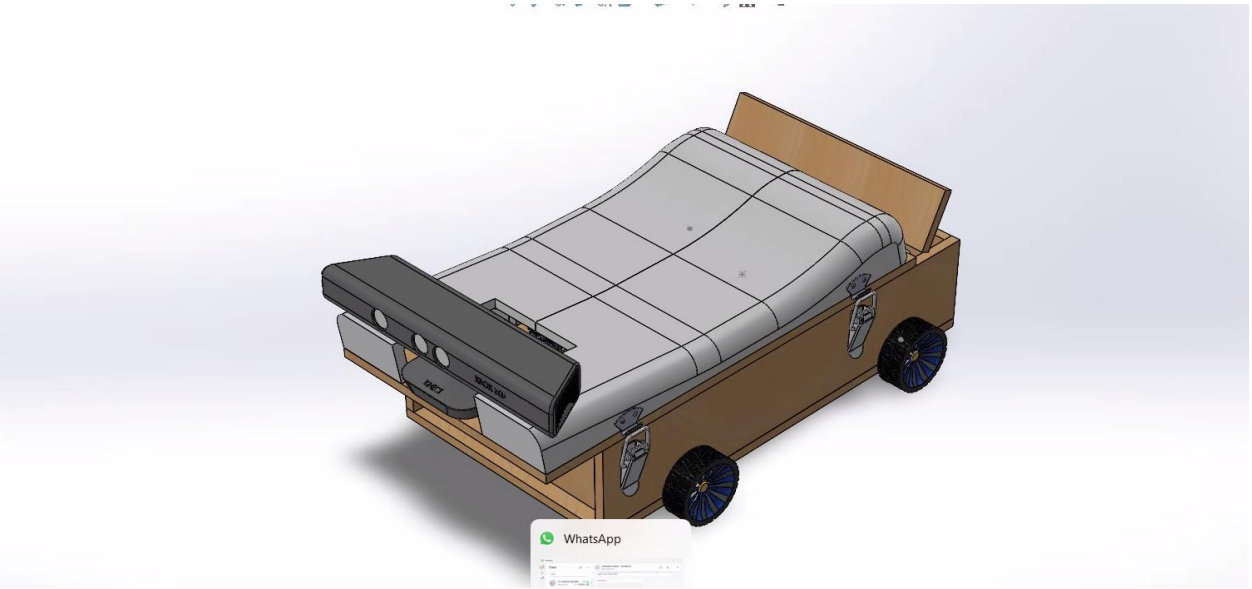

Figure 42:unity results 1

The robot simulation prefab on unity

Figure 43:unity results2

One of the simulations goals is to test the robot behavior in different environment

We chose a railway yard as our environment

Figure 44:unity results3

Figure 45:unity results4

Robot in fire situations

Figure 46:Unity Results5

Figure 47: Unity Results 6

Unity and ROS communication

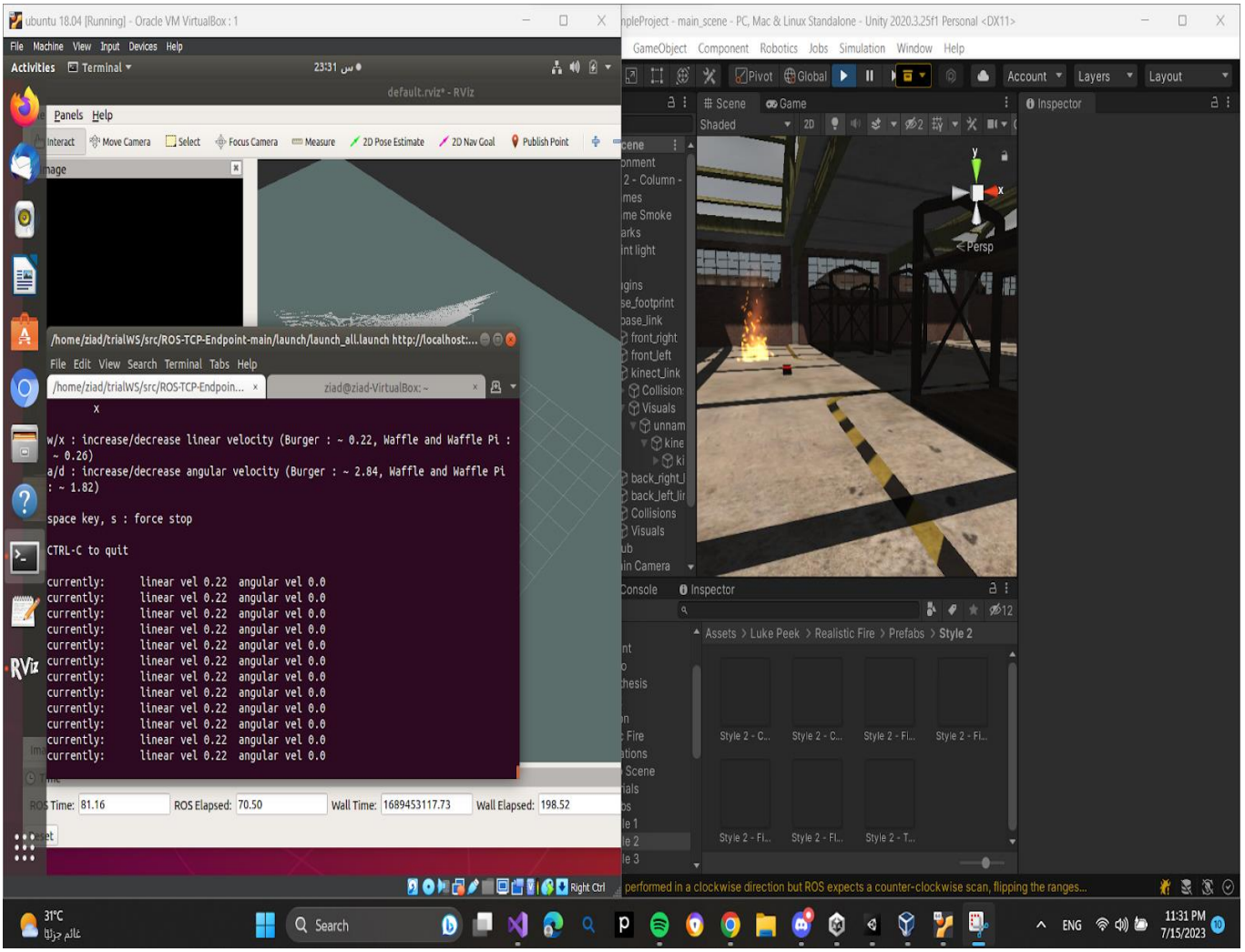

Figure 48:unity results7

Start Mapping

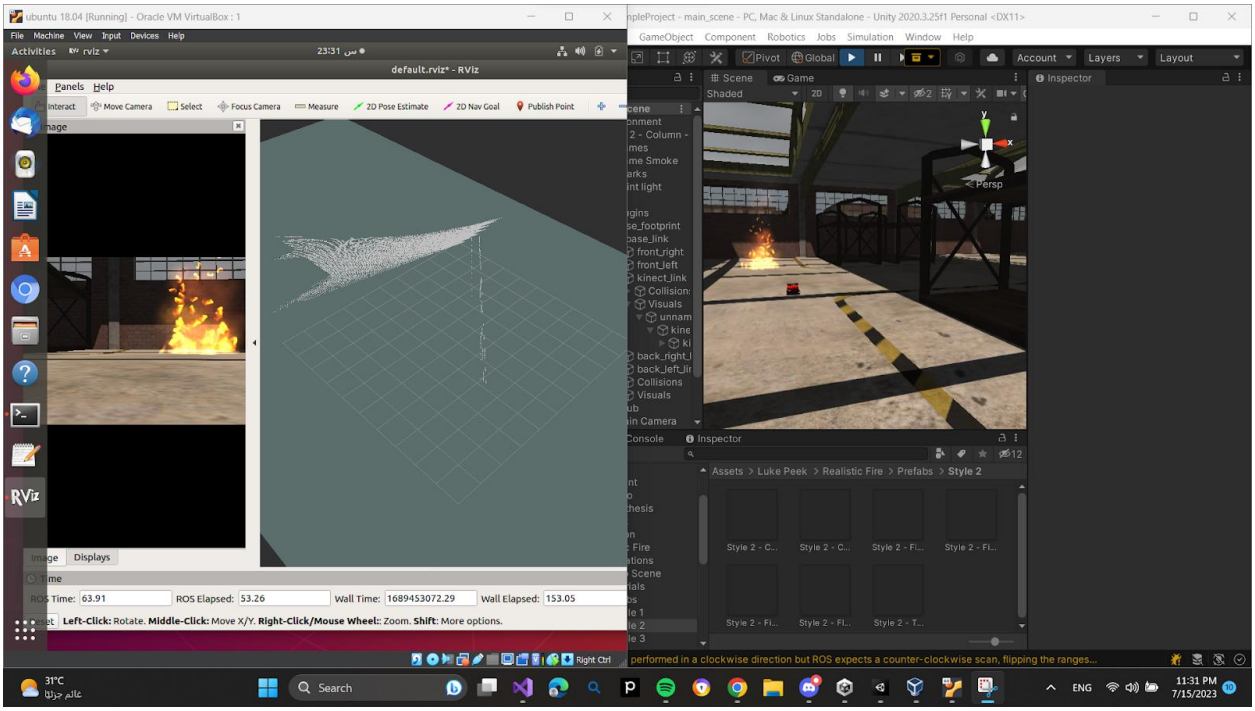

Figure 49:unity results8

Map building

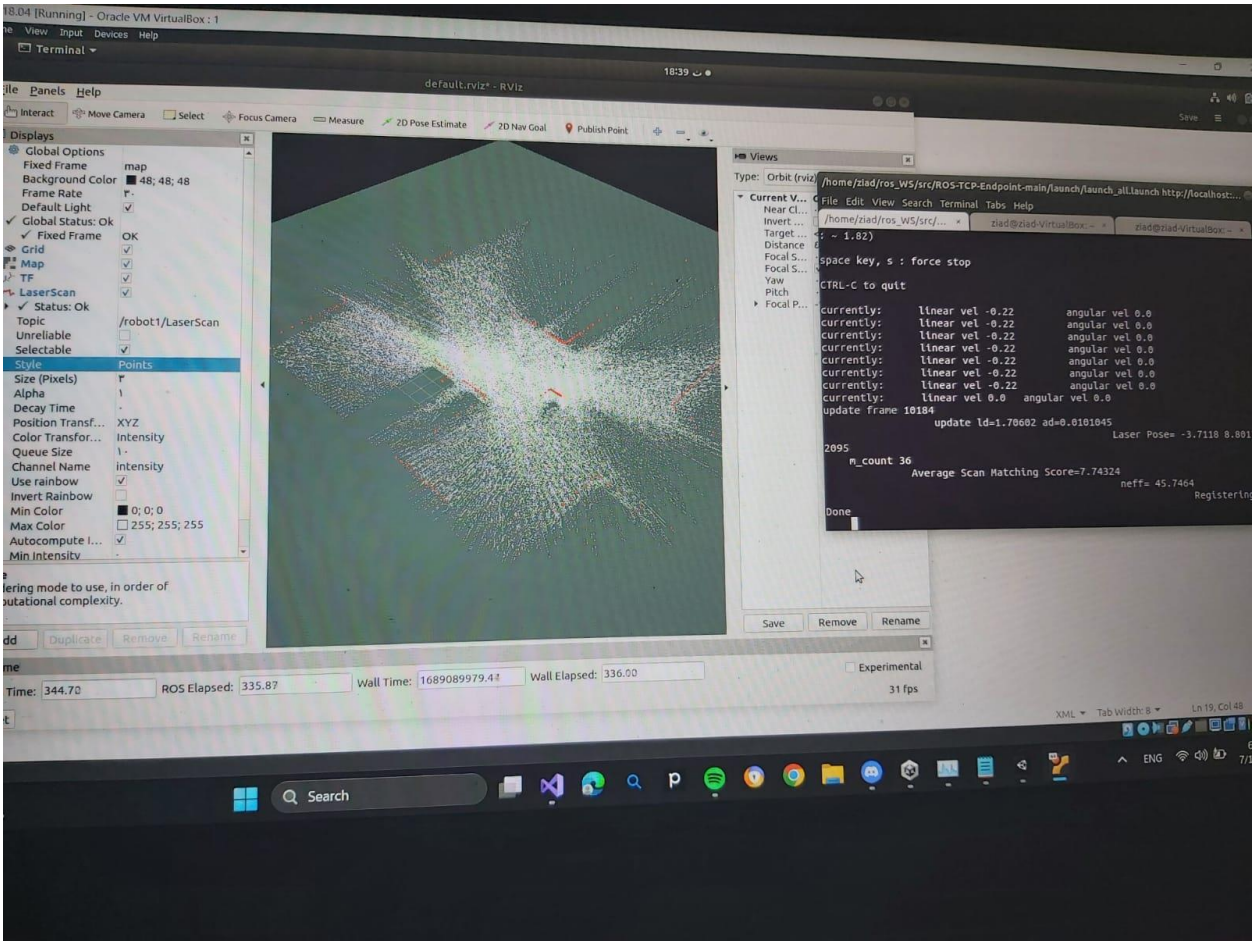

Figure 50:unity results9

7.4 SLAM

SLAM device (kinect)

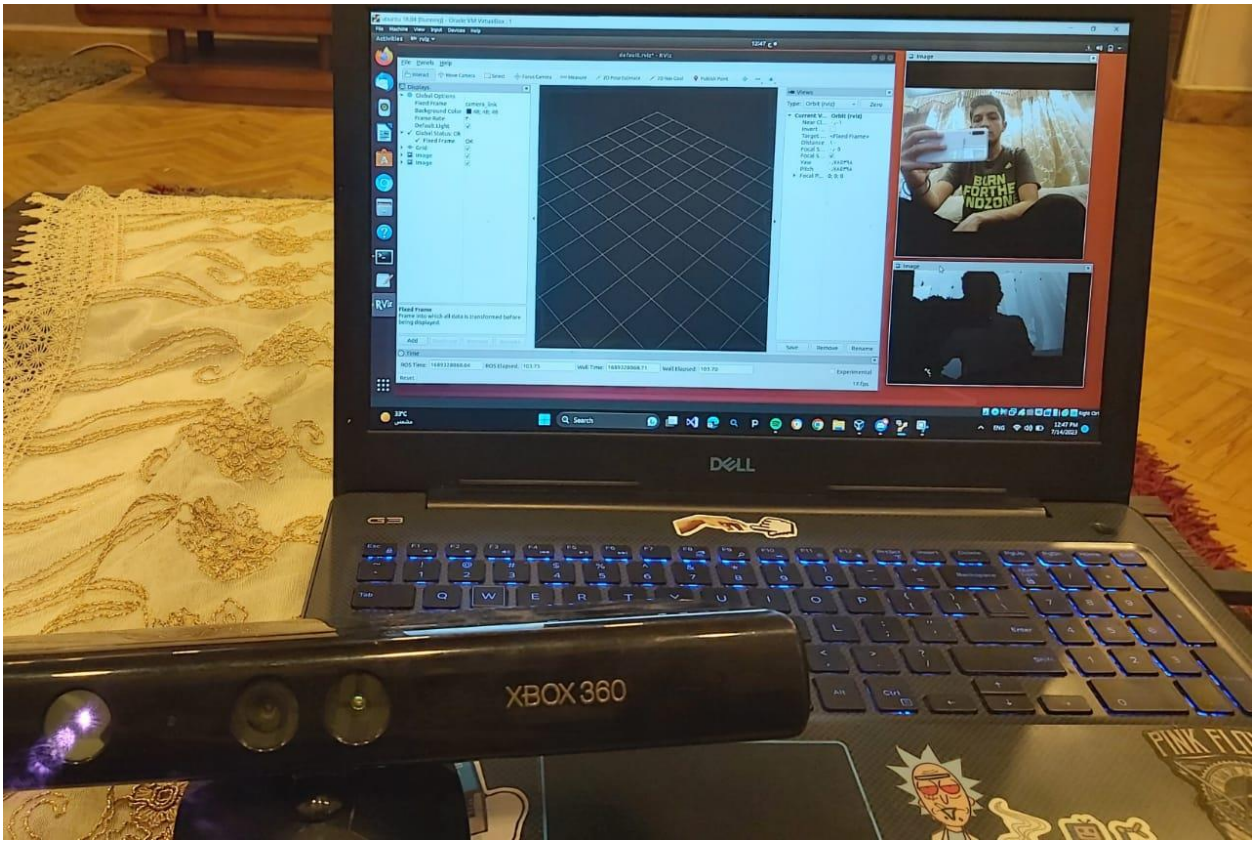

Figure 51:SLAM results 1

Kinect Data Visualisation

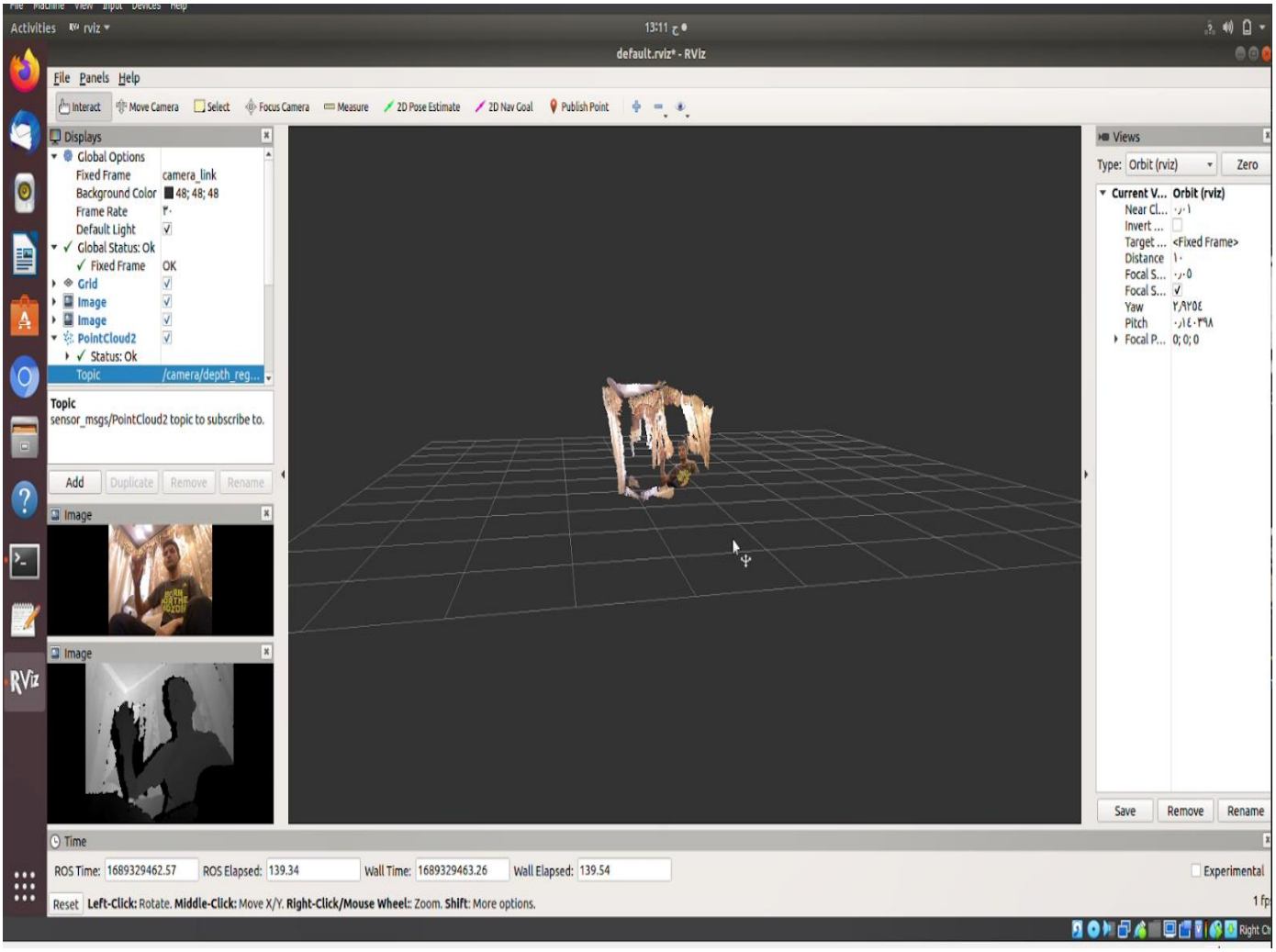

Figure 52:SLAM results 2

8. Chapter 8 : Conclusion

our project has successfully achieved its main objective of creating an applicable solution for fire-fighting operations. By developing a prototype that can be easily turned into a product using existing components, we have ensured that our solution can be readily deployed in the field.

In terms of operation, we have designed our solution to be highly accessible and easy to use. By installing the necessary software on any laptop with a Linux operating system, we have made it simple for firefighters to connect with the robot, collect data, and visualize it in real-time. The joystick interface allows for intuitive manual operation, while the robot's sensors and autonomous mode ensure that it can continue mapping the environment and avoiding obstacles even in the event of a lost connection.

Our solution also has the capability to scale up to multiple robots, allowing for greater coverage and data collection in larger fire sights. By transmitting data between robots and back to the base station, we have created a robust and resilient system that can adapt to changing conditions in the field.

In our solution, we emphasized building a highly modular system where each component can function independently or integrate seamlessly with existing systems. The data acquisition system can be implemented on various platforms, from rovers to drones. The SLAM system is designed to generate maps and assist autonomous navigation in conjunction with different mechanical designs. Additionally, the Unity engine-based simulation allows firefighters to train in diverse environments and with any robots already in use.

Overall, we believe that our solution has the potential to revolutionize fire-fighting operations and improve the safety of firefighters on the front lines. By providing valuable data and insights into the fire sight, we are enabling

firefighters to make more informed decisions and develop more effective strategies for combating fires. We look forward to seeing our solution implemented in the field and making a positive impact on fire-fighting operations.

9. Chapter 9 : Reference

[1] Fire statistic in Egypt for 2022. Central Data Catalog. (n.d.). https://censusinfo.capmas.gov.eg/metadata-en-v4.2/index.php/catalog

[2] Fire statistic in Egypt for 2022. Central Data Catalog. (n.d.). https://censusinfo.capmas.gov.eg/metadata-en-v4.2/index.php/catalog

[3] Autonomous Embedded System Vehicle Design on Environmental, Mapping and Human Detection Data Acquisition for Firefighting Situations
By : Armando Pinales and Damian Valles (Ingram School of Engineering)

[4] A Fire Reconnaissance Robot Based on SLAM Position, Thermal Imaging Technologies, and AR Display By: Sen Li, Chunyong Feng, Yunchen Niu, Long Shi, Zeqi Wu and Huaitao Song

[5] Multi-Agent Systems for Search and Rescue Applications by Daniel S. Drew