The Hong Kong Polytechnic University
Department of Electrical and Electronics Engineering

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
電機及電子工程學系

Project ID: FYP_Isaac_1_B

Autonomous Underwater Vehicle Hardware and Control System Development with ROS (Robot Operating System)

by

HE JIAJIE

20058054D

Final Year Project 2023/2024 (A)

Final Report

Bachelor of Engineering (Honours)

In

Electrical Engineering

of

The Hong Kong Polytechnic University

Supervisor: Dr FUNG Yu-fai

Date: 16-March 2024

Abstract

This project centers on developing an autonomous underwater vehicle (AUV), encompassing the design of its hardware system, and implementing a control system utilizing Robot Operating System 2 (ROS2).

The project is executed in two main stages. The initial stage focuses on developing the AUV's hardware system, which includes its electrical architecture and mechanisms crucial for underwater functionality. This stage ensures the AUV's essential controllability and comprehensive operation in submerged conditions.

The second stage is dedicated to advancing the control system within the ROS2 framework. The framework involves creating control nodes and integrating them with other team members' nodes (subsystems) to form a unified autonomous underwater vehicle system. This system is designed to process control commands from autonomous programs and manage the AUV's hardware systems and behaviors accordingly. In this stage, the utilization of ROS2 is first researched and studied, and the programming methods, such as various libraries and object-oriented programming, are frequently practiced to construct the control system. After the ROS2 development, other team members' nodes are consolidated into the control system by designing unified ROS2 communication interfaces.

At the end of the project, this project successfully developed a hardware system for the AUV, which includes the electrical architecture and essential mechanisms for underwater functionality. The AUV control system, built on ROS2, effectively utilizes ROS2 client library APIs to establish a robust communication layer for the AUV's subsystems, such as the autonomous, buoyancy, and thruster subsystems. After the testing of the ROS2-based AUV control system underwater, it is also believed that the control system is effective and confident in handling control commands from autonomous programs to control the hardware system of the AUV.

Apart from the integration between the hardware system and the control system, this project has discussed and introduced practical telecommunication solutions, enabling efficient remote control, configuration, testing, and commissioning of the AUV system. The project also contributes to the development of AUV technology by providing innovative solutions to underwater exploration and automation challenges.

The use of ROS2 as the AUV control system is a novel approach in the academic community, where it has the potential to significantly reduce development costs and accelerate the pace of AUV innovation. The project's development of a control system that draws inspiration from the "reverse proxy" concept represents a unique innovation, allowing for a more efficient and robust management of the AUV's hardware systems and behaviors.

The implications of this project extend beyond the immediate development of an AUV. By utilizing a free and open-source framework like ROS2, the project opens up new possibilities for researchers and developers in the AUV field to collaborate and innovate. This not only lowers the barrier to entry for new participants but also fosters a community-driven approach to AUV development, which can lead to more rapid advancements in the field.

Table of Contents

1. Introduction

1.1 What is an AUV

An Autonomous Underwater Vehicle (AUV) is an underwater vehicle that operates independently of direct human control. Unlike Remotely Operated Vehicles (ROVs), controlled by operators on the surface or onshore, AUVs are self-contained systems capable of performing tasks automatically based on pre-programmed instructions.

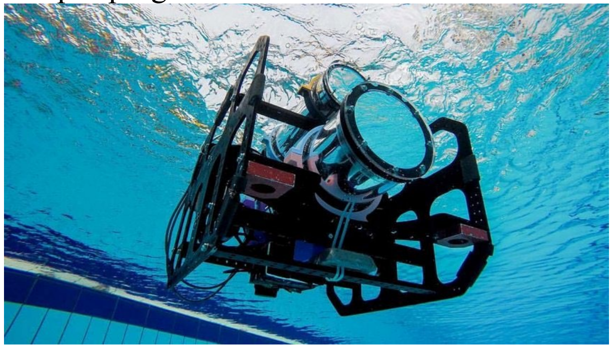


Figure 1: An Autonomous Underwater Vehicle (AUV) [1]

AUVs are crucial in various fields, such as ocean exploration, resource mapping, and environmental monitoring. They are essential in areas where human presence is limited or hazardous, such as deep-sea research or disaster response. AUVs provide a means to gather data and perform tasks in these environments without direct human intervention. Below are some reasons to develop an AUV essentially [2][3][4][5]:

1. **Advancing Ocean Exploration**: AUVs can access areas of the ocean that are otherwise inaccessible to humans, enabling the discovery of new species, mapping of underwater landscapes, improvements in marine science, and the study of aquatic ecosystems.
2. **Resource Mapping and Management**: AUVs are used to map and monitor underwater resources, such as oil and gas deposits, minerals, and fishing grounds. This information is critical for sustainable resource management and economic development.
3. **Environmental Monitoring**: AUVs can monitor the health of marine environments, detect pollution, and study the impact of human activities on the ocean. This data is essential for understanding and addressing environmental challenges.
4. **Disaster Response**: In the event of an oil spill or other underwater disasters, AUVs can be deployed to assess the extent of the damage and assist in cleanup efforts.

Therefore, it is interesting to develop an AUV alongside my team members. It is also believed that this project holds significant implications for ecology, technological advancement, and sustainable development. The prospect of contributing to such a multifaceted endeavour is both intellectually stimulating and personally rewarding.

Beyond the intrinsic value of AUVs in advancing ocean exploration, resource mapping, and environmental monitoring, it is also challenging to design and construct an AUV. The project presents a formidable test of my perseverance and problem-solving skills, characteristics that I embrace and find deeply satisfying. The opportunity to grapple with complex technical issues and witness the fruits of our collective labour drives my enthusiasm for this project.

1.2 Characteristics of an AUV

By referencing from AUV development literature [6] and an AUV competition rule book [7], AUVs may include the following characteristics:

1. **Integrated Computer System**: AUVs have an onboard computer system that processes input and output signals, allowing them to make decisions and execute tasks without external input.
2. **Embedded Power Source**: AUVs are powered by internal energy sources, such as batteries, and do not rely on external power cables (tetherless), allowing them to operate independently underwater.
3. **Specialized Wireless Communication**: AUVs are equipped with wireless communication systems that enable them to transmit data to and receive instructions from operators onshore or surface vessels, particularly during testing, commissioning, or when adjusting mission parameters.
4. **Real-time Image Processing for Autonomous Operation**: Many AUVs can use image recognition algorithms to navigate and avoid obstacles underwater, ensuring safe and efficient operations even in complex underwater environments. Therefore, AUVs can follow pre-programmed missions and carry out tasks without real-time human intervention. They are designed to function without human-made disturbances during normal operations.
5. **Physical Actuators**: AUVs are equipped with physical actuators to perform various underwater movements. This system typically includes thrusters and buoyancy, which allow AUVs to manoeuvre horizontally (i.e. forwards, backward, left, right) and vertically (i.e. up, down, diving, ascending). The control system is responsible for translating the computer’s decisions or commands into actual vehicle movements through the water.
6. **Display Screen**: Although AUVs operate autonomously, they often feature a display system that provides a visual interface for operators during certain phases, such as mission planning, testing, or maintenance. This display can show system status, such as the vehicle’s position, speed, heading, power levels, and other relevant operational data. It may also be used to monitor sensor data and perform diagnostics, ensuring that the AUV functions as intended and allowing operators to intervene if necessary.

1.3 Reasons for Constructing an AUV with ROS2

In underwater robotics, Remotely Operated Vehicles (ROVs) have traditionally been the preferred choice for tasks requiring real-time human control due to their maturity and lower costs. However, as the demand for efficient and cost-effective autonomous solutions grows, there's a burgeoning interest in AUVs.

ROVs necessitate constant human oversight, limiting their autonomy and flexibility. Despite their widespread use, it's important to note that AUVs, while more expensive with costs ranging from US$2-6 million, offer distinct advantages such as a depth rating of up to 3,000 meters and faster survey capabilities, making them an attractive option for various underwater tasks [1].

Recognizing the potential for a more accessible and widespread use of AUVs, this project focuses on developing a cost-effective AUV structure with Robot Operating System 2 (ROS2). ROS2 is a structured communication layer above the operating system. The reason for choosing ROS2 to develop an autonomous program is that ROS2 is an open-source, free, modular robotics software framework. It is also designed to meet the needs of next-generation robotics applications by providing a robust middleware that enables different programs to communicate and exchange data effectively through custom-defined interfaces. Therefore, this architecture facilitates developing, testing, and deploying individual components in complex AUV subsystems and offers a cost-effective solution for creating a robotic solution.

1.4 Project Goal

This project addresses a diverse range of challenges, including the design of an electrical system that meets the AUV's power requirements, the creation of AUV mechanisms, the definition of ROS2 nodes' functions, and the efficient transmission of control commands through nodes to manage the AUV's hardware components within a ROS2-based control system.

1.5 Objectives of this project

The primary objectives of this project are as follows:

1. Design and construct the AUV's hardware, primarily focusing on the electrical system.
2. Develop basic control programs for the AUV's thrusters.
3. Acquire knowledge of Python object-oriented programming and ROS2 application programming interfaces (APIs).
4. Develop a framework for a ROS2-based AUV control system with custom ROS2 interfaces.
5. Define ROS2 interfaces with team members to ensure seamless integration and communication.
6. Consolidate team members' nodes (subsystems) into a unified system.
7. Test and debug nodes to ensure that the AUV control system can effectively handle and process data from team members' systems, thereby controlling the AUV stably and executing autonomous tasks correctly.

1.6 Report Overview

This report begins with a background to the general design considerations for autonomous underwater vehicles (AUVs) and the planned ROS2 control architecture (Section 2). It then delineates the design and control of the electrical system, including thruster control. The report continues with an explanation of how ROS2 nodes are utilized to control the AUV and facilitate communication with other team members' nodes (Section 3). Subsequently, the report presents the analysis of the project results (Section 4). The report then discusses the project's limitations and weaknesses and reflects on potential improvements (Section 5). In conclusion (Section 6), the report summarizes the project achievements and learning outcomes from this project.

2. Background

2.1 General AUV Hardware Development

An AUV hardware development involves both an electrical system and a mechanical system, so a general design of an AUV should identify and consider the following factors: hull design, propulsion, submerging, and electric power [8].

The design of the hull emphasizes the critical need to house AUV components in a dry, watertight environment. It should prioritize easy accessibility and maintainability of components and incorporate modularity to accommodate potential future changes or additions to the AUV structure [3].

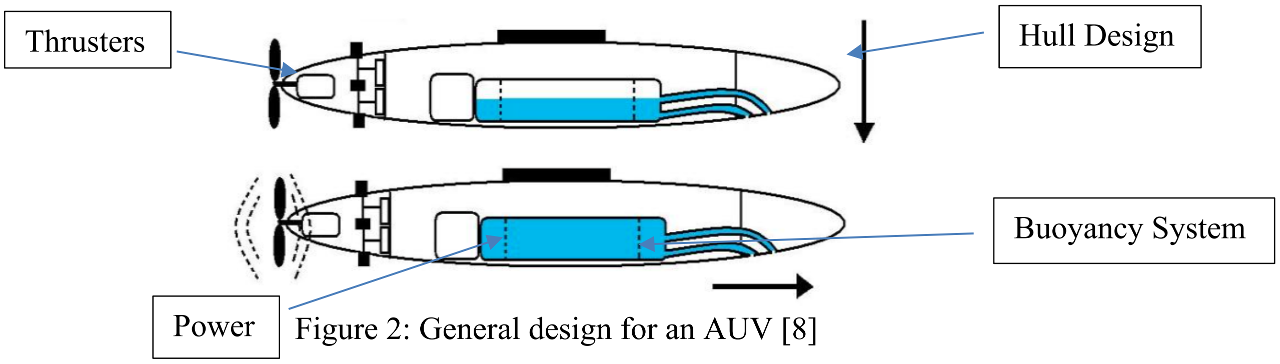


Figure 2: General design for an AUV [8]

Propulsion and submerging can refer to the presence of effective thrusters and a fast-response buoyancy system, respectively, and electric power can be considered a high-performance power source (high current drawing battery).

Generally, the thrusters provide vertical movements, while the buoyancy system provides horizontal movements for the AUVs. Thrusters used in the AUVs operate based on the principle of reaction force to propel the vehicle. In the context of underwater vehicles, they expel water with force in one direction. According to Newton's third law, the vehicle also experiences an equal and opposite reaction force, resulting in a forward motion. The buoyancy system can be a ballast system that pumps water in or out to control buoyancy and the AUV's depth. The primary purpose is to adjust the vehicle's weight, affecting its overall density and ability to float or sink. Therefore, when developing the ROS2-based control system nodes, they should be able to communicate with the thruster and buoyancy systems and implement hardware control on them.

2.2 Electrical System

When developing an AUV electrical system, it is necessary to consider what electronic components should be included in a hull. According to a conference paper about AUV design [9], a simple AUV has the following electrical circuit design:

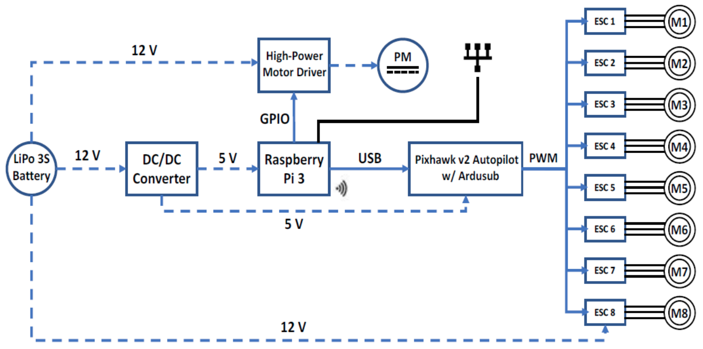


Figure 3: An Example of AUV Electrical System Design [9]

From the above figure, it is observed that the above AUV electrical system includes a LiPo 3S battery, DC-DC converter, Raspberry Pi 3 (RPi 3) computer board, Pixhawk, motor driver, and electronic speed controllers (ESCs). The 12VDC battery powers the DC-DC converter, which steps down a voltage to 5VDC to power the RPi 3 and the Pixhawk, the motor driver, and ESCs. The RPi 3 provides GPIO signals to control the motor driver. The Pixhawk provides PWM signals to ESCs to control thrusters.

Therefore, an AUV electrical system can be designed and conducted based on this reference. For instance, a new AUV electrical system can include the following components:

Input Devices:

1. A camera for computer vision and an AUV autonomous system.
2. A pressure sensor for reading depth values and AUV buoyancy system

Output Devices:

1. ESCs to control thrusters.
2. Thrusters to produce horizontal moving force.
3. Motor drivers/water pumps control the buoyancy system to produce vertical moving force.

Power Devices:

1. A high-current battery
2. A DC-DC converter to provide suitable voltages for electronic devices.
3. Fuse boxes to protect devices if short-circuited.
4. A kill-switch to turn on/off the AUV.

Others:

1. An integrated computer for processing I/O signals

2.3 Mechanical System

The mechanical system design of the AUV involves storing components, constructing frames, and positioning components such as thrusters, electronic components, and buoyancy.

<u>2.3.1 Storing for Components</u>

Electronic components such as the computer board, ESCs, battery, and wires should be stored in a closed space such as inside a cylinder with sealing caps. The rubber gaskets provide tension forces between the sealing cap and the lid of the cylinder to avoid water leakage [10].

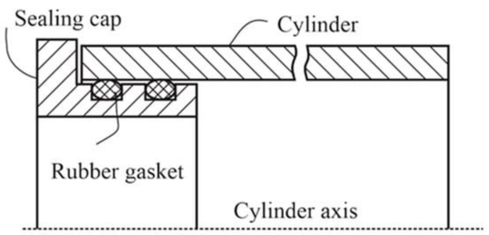


Figure 4: Design of a Waterproof Cylinder [10]

<u>2.3.2 Frame Design</u>

When considering the frame design, it is critical to determine different factors, such as the materials for the frame, the hull design, the frame's weight, and the number of thrusters and cylinders to use.

Since our AUV will use a buoyancy system for the vertical movements of the AUV, we may consider using at least two cylinders in the AUV. One cylinder stores electronic components and the other uses the buoyancy system. Therefore, thrusters can only be used for exerting horizontal forces, resulting in a deduction of thrusters. For example, two thrusters are enough to drive the AUV in this project.

To construct the AUV frame efficiently, it is worth considering using acrylic to manufacture the frame for the AUV. Acrylic is highly amenable to precision machining, including laser cutting, allowing us to achieve accurate and intricate designs that align with our specific requirements. Furthermore, acrylic is both lightweight and robust, making it a suitable choice for the frame of the AUV. These qualities contribute to the overall flexibility and durability of the AUV design.

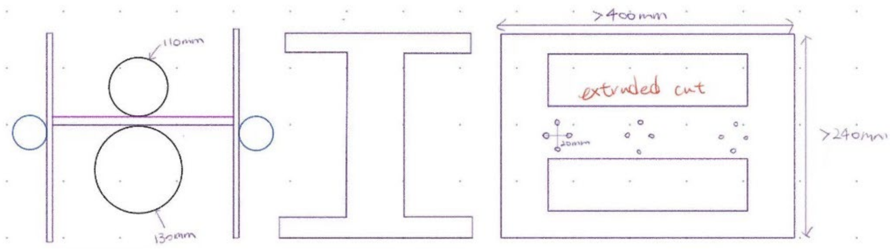


Figure 5: Preliminary AUV frame design

2.4 ROS2 Architecture and Communication Patterns Review

<u>2.4.1 ROS2 Architecture</u>

In this project, ROS2 is used as the communication layer for different AUV subsystems, such as the autonomous subsystem (image processing system), the buoyancy system, the thruster subsystem, and the control system. According to the ROS2 design sheet [11], the ROS2 is referred to as middleware because it provides a set of client library interfaces (APIs) such as

rclpy (Python API library) and **rclpp** (C++ API library) that allow different programs to communicate and exchange data. The ROS2 abstracts the underlying communication details and provides a standardized way for programs (packed as nodes) to send and receive messages. In the bottom layer, ROS2 uses a middleware, a Data Distribution Service (DDS), to handle the actual data communication. DDS is a standard for publish-subscribe messaging that provides low-latency, high-reliability data communication across distributed systems. As a result, ROS2 leverages DDS to translate the data sent between nodes into a format that DDS can understand and then forwards it to the appropriate destination.

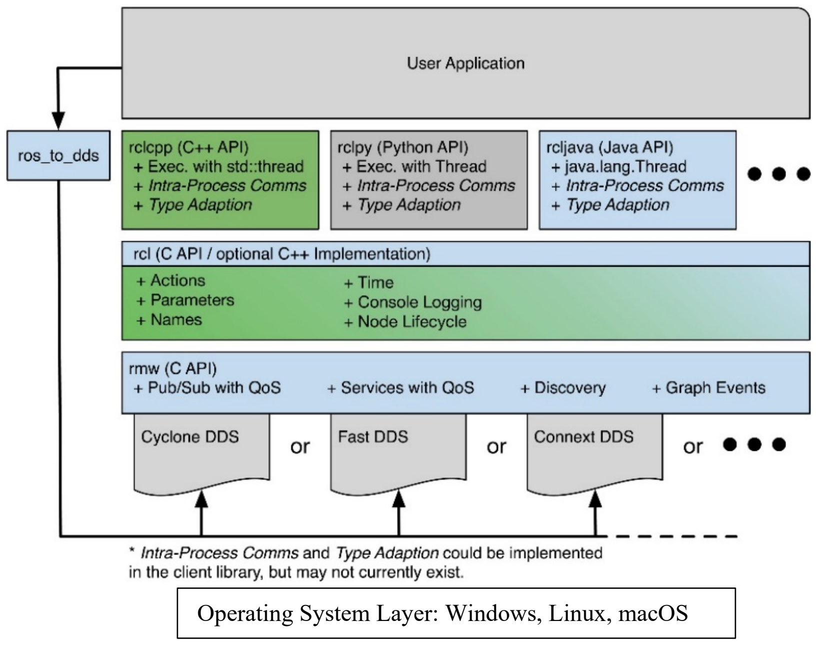


Figure 6: ROS2 Architecture [11]

After understanding the architecture of ROS2, it's also necessary to understand the ROS2 client APIs so that developers can add them into different programs correctly to establish node communications. Since Python is the programming language used to develop the AUV control system in this project, the ROS2 Client Library for Python (rclpy) is utilized. Similar to the rclcpp (ROS2 C++ API) and rcljava (ROS2 Java API), the **rclpy** provides three communication patterns: **topics, services, and actions**, for developers to develop nodes.

2.4.2 Communication Platform

ROS2 offers a comprehensive set of communication layers and application programming interfaces (APIs) such as Topics, Services, and Actions. These enable developers to define the data flow and control between nodes, ensuring coordination and synchronization in distributed systems. As a result, data can be shared and processed between different functional nodes. Therefore, ROS2 provides an advantage in that teammates responsible for other AUV subsystems do not need to be concerned about the methods to transfer data from their subsystems to another subsystem. They can concentrate more on developing their AUV subsystems.

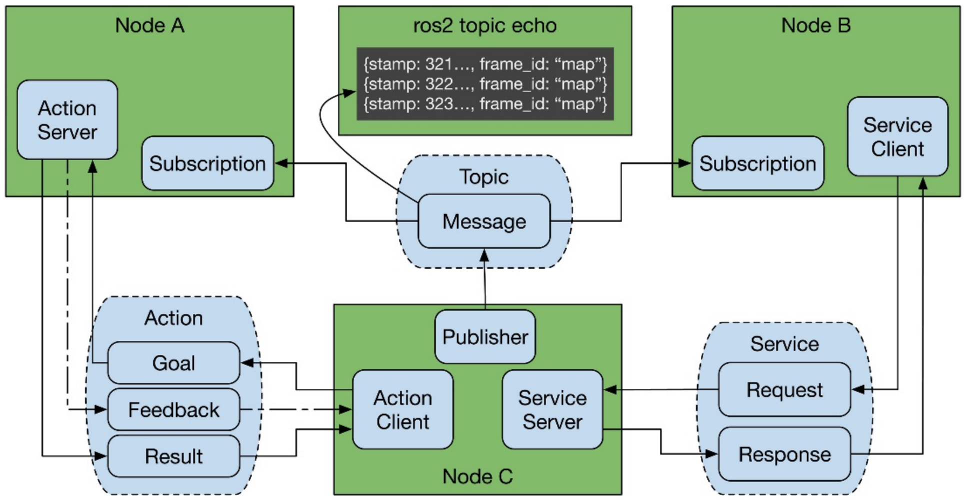


Figure 7: Communication Patterns in ROS2 [11]

2.4.3 Nodes

After understanding the architecture and communication patterns of ROS2, nodes are introduced. In ROS2, nodes are processes that perform computation [12]. They are essentially standalone programs enhanced with ROS2 client library interfaces (APIs) to facilitate standard ROS2 data communications.

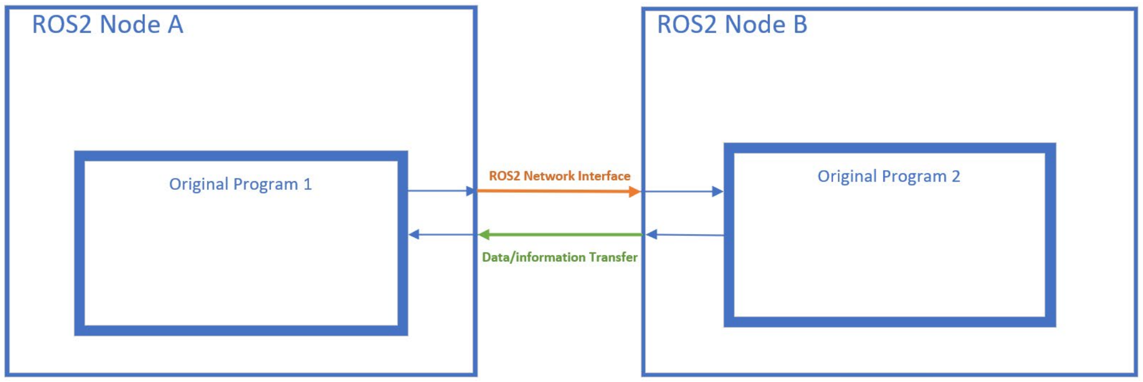


Figure 8: Data Transfer between ROS2 Nodes

Take the above figure as an example. The ROS2 APIs provide standard or customized network interfaces to allow programs to transfer data conveniently so that the complexity of these programs can be reduced. Teammates can, therefore, focus more on developing the AUV subsystems and do not need to spend time building wheels for communication.

2.4.4 Topic

The Topics interface is the simplest communication method in ROS2. It provides a publish-subscribe functionality, similar to Message Queuing Telemetry Transport (MQTT), to allow one node to periodically publish/broadcast messages under a specific topic name and other interested nodes to subscribe to the relevant topics to obtain messages [13]. This communication pattern is generally used in nodes that implement periodic tasks such as recording sensor data and monitoring system status.

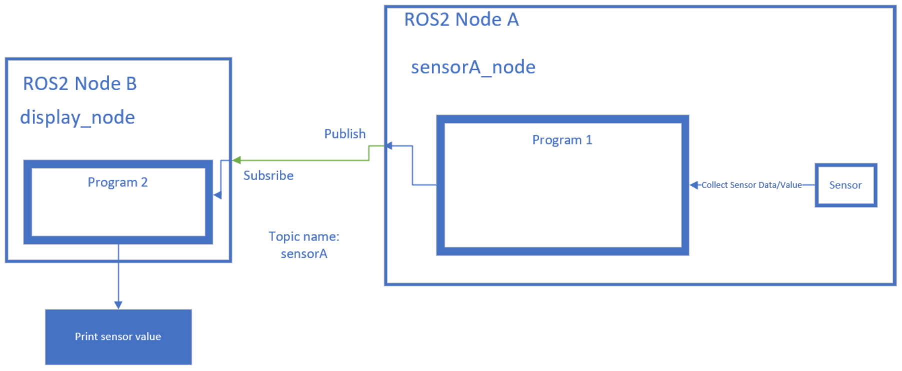


Figure 9: An example of ROS2 Topics

For example, a program developed initially to read data from a pressure sensor can be packed as a node (publisher) after it is added to the ROS2 Topics interface to publish the sensor data messages by a specific topic name (i.e. sensorA_node) periodically. Another node (subscriber), such as the display_node, can be developed by adding the ROS2 Topics interface to subscribe to the messages published from the sensorA_node and print out the obtained data from the pressure sensor.

2.4.5 Services

ROS2 provides a request-response style pattern that allows easy data association between a request and response pair [15]. A request-response communication pattern is commonly used in the Internet protocol, such as a client's Hypertext Transfer Protocol (HTTP) request to request a picture with headers and an HTTP response from a server with headers and contents.

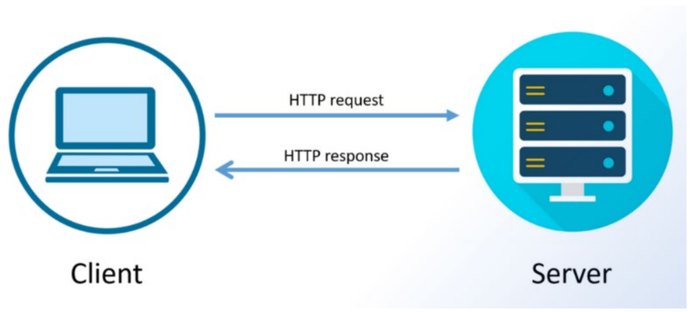


Figure 10: An Example of an HTTP Request and Response [15]

The ROS2 service interface is valuable and important because it can be used to ensure that whether services-based server nodes acknowledge or complete tasks after a request is sent from services-based client nodes. Take the below figure as an example. In the beginning, the control_node (client) announces a “turn on fan” request with a service name “fan_control”. Then, the fan_node (server), which listens to the service name “fan_control”, receives the request (message) from the client and implements the request by turning on the fan. After that, the fan_node program checks that no errors occurred and responds to the control_node to turn the fan on successfully.

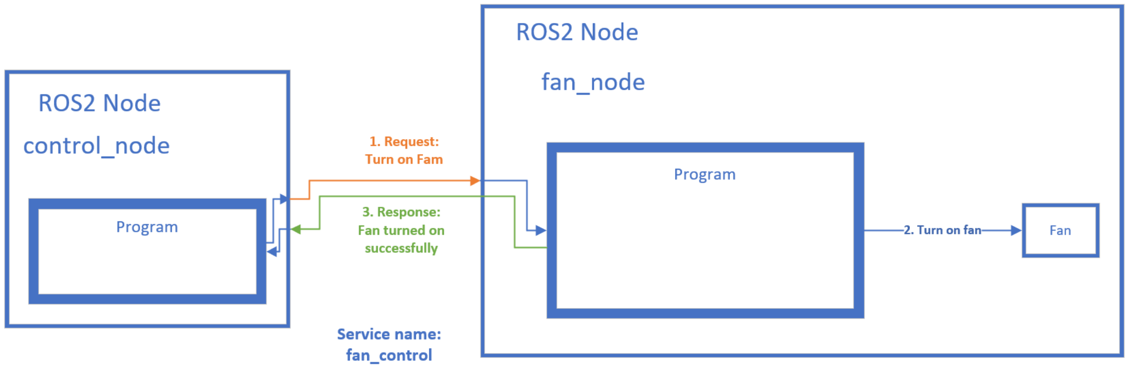


Figure 11: An Example of ROS2 Services-based Nodes

After receiving the response from the fan_node, the control_node knows that the control command has been implemented successfully and allows the next action to be implemented. If errors occur while turning on the fan, the fan_node may respond to a failure message to control_node to tell it that the task cannot be implemented. Hence, the control_node may announce urgent signals to notify users or implement further actions, such as requesting the fan_node to turn on the fan again.

2.4.6 Actions

ROS action interface is a higher-level communication pattern that combines both communication patterns of topics and services [16]. Similar to the standard service's request-response communication pattern, the server side provides Topics-based feedback messages to the client before a response is sent back. Such communication interfaces can be used in robotics applications, such as closed-loop control of the velocity of a vehicle.

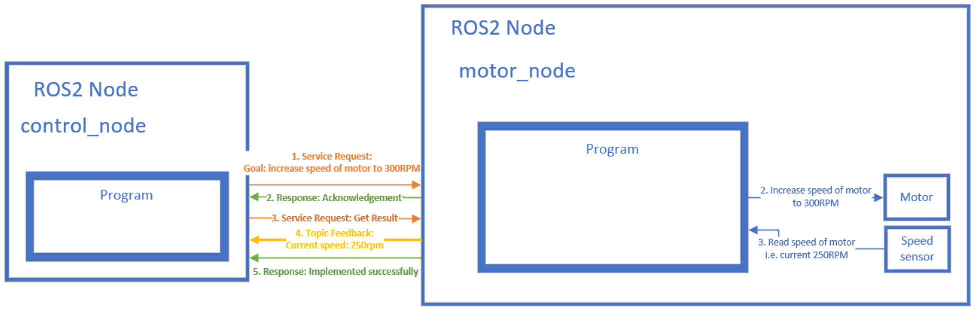

ROS2 Node
control_node
Program
1. Service Request:
Goal: increase speed of motor to 300RPM
2. Response: Acknowledgement
3. Service Request: Get Result
4. Topic Feedback:
Current speed: 250rpm
5. Response: Implemented successfully
ROS2 Node
motor_node
Program
2. Increase speed of motor to 300RPM
Motor
3. Read speed of motor i.e. current 250RPM
Speed sensor

Figure 12: An Example of ROS2 Actions-based Nodes

Take the above figure as an example. When the control_node establishes an action communication with the motor_node, it first sends a service request of a goal to the motor_node to increase the motor's speed (300RPM). If the motor_node receives the request for the goal, it responds with an acknowledgment back to control_node. Afterwards, the control_node sends a second request for results after receiving the acknowledgement. The motor_node receives the second request and responds by publishing frequent feedback on the current motor speed by the ROS2 topics interface back to the control_node (i.e. 250RPM, 255RPM, 263RPM…). The control_node can, therefore, know that the speed of the motor is increasing. Once the motor reaches 300RPM, the motor_node completes the program and responds to the control_node, indicating that the implementation is completed successfully.

2.4.7 Custom-defined Interfaces

While ROS2 streamlines the process of exchanging data through its intuitive communication patterns, it is essential to establish well-defined interfaces for these patterns. These interfaces serve as a contract or norm that specifies the data types being transmitted and received, such as Booleans, integers (i.e. int32), floating point numbers (i.e. float64), strings, etc. [17][18]. The data types ensure that nodes within the system can interact seamlessly by adhering to a common understanding of the data structure. Therefore, different nodes can efficiently import and export variables with correct data types from the ROS2 network, and invalid data type errors can be prevented.

For instance, consider a scenario where we need to define a message interface for a sensor node (i.e. sensorA_node), which publishes depth values from a pressure sensor. The interface may look like this:

```
int32 depth_value
```

This simple interface is defined within a ROS2 **.msg** file, which is then compiled into a language-specific format that nodes can utilize. With this approach, sensorA_node can publish a topic containing an int32 variable named depth_value, and a display_node can subscribe to this topic to receive and process the depth_value presented by a data type of int32.

In addition to creating custom interfaces, developers can leverage ROS2's extensive library of standard message **(.msg)**, service **(.srv)**, and action **(.action)** interfaces to facilitate data transfer. For example, ROS2 provides a standard message interface for images [19] and is widely used in robotics and computer vision applications. These predefined interfaces cover a wide range of common data types and are designed to promote interoperability across different nodes and packages within the ROS2 ecosystem.

In summary, the custom-defined interfaces in ROS2 are vital for maintaining clear communication channels between nodes by outlining the structure and type of data being shared. This practice not only enhances the interoperability of the system but also simplifies the development process by providing a consistent framework for data exchange.

2.5 Advantages of ROS2

Apart from the benefits of using ROS2 as a communication layer for the AUV control system, ROS2 also brings the following advantages [11] to this project:

1. Cross-Platform Support: **ROS2 supports a wide range of operating systems and hardware platforms**, including Linux, Windows, macOS, ARM64, and x86_64, as it leverages DDS to translate the data. This versatility makes it suitable for applications to be built across different environments. Consequently, ROS2 bridges the communication gap between different system versions or architectures, **allowing nodes on various systems to interact with one another once ROS2 is installed**.

2. Quality of Service: ROS2 is built to handle complex distributed system architectures, leveraging the Data Distribution Service (DDS) communication protocol with Quality of Service (QoS) [20]. This **ensures efficient and reliable data transfer even in challenging network conditions.**

3. Community Support: ROS2 benefits from a vibrant open-source community that **provides a wealth of tools, libraries, and documentation**. This support facilitates code sharing and reuse, accelerating the development process. For example, it is free to download different ROS2 tools such as Gazebo (robot simulation) and tf2 (transform library) to advance the robotic system further.

4. Diverse Network: ROS2 nodes can communicate across different computer devices within **the same IP segment of a local area network (LAN)**. This feature simplifies the process for developers to monitor and access the status of various nodes. It also enables the establishment of **master-slave relationships** between different computer boards, such as connecting a main computer board to a companion computer board. This setup allows the companion board to handle specific tasks independently while still being integrated into the more extensive system. For example, the main computer board could be responsible for real-time control and decision-making, while the company board handles data logging, analysis, or **control of external systems**. This division of labour not only enhances system efficiency but also facilitates modular development and easier maintenance.

2.6 Pros and Cons of Using Services instead of Topics in ROS2 Control System

Owing to the time constraints and the appropriate workload associated with the final-year project, this project focuses on primarily employing the ROS2 **services** interface in conjunction with the **topics** interface to construct the ROS2-based control system.

Before establishing an AUV control system framework with ROS2, it is imperative to weigh the advantages and disadvantages of employing either the topic interface or the service interface. As outlined in the official ROS2 documentation [18][21], the distinguishing characteristics of these two communication mechanisms are summarized in the table below:

Aspect	ROS2 Topics	ROS2 Services
Communication	Asynchronous message passing between nodes	Synchronous request-response communication between nodes
Data Flow	One-to-many (publishers to subscribers)	One-to-one (client to service)
Usage	Ideal for continuous data streams, such as sensor data	Suitable for discrete operations, such as setting parameters or performing specific actions by commands
Real-time Feedback	Subscribers receive data as it is published, without direct feedback to publishers	Services provide immediate response or confirmation to clients
Suitability	Well-suited for periodic and time-sensitive data	Better for non-periodic, logical operations that require

		immediate results
Complexity	Simpler architecture for data distribution	More complex as it involves request and response handling
Error Handling	Messages may be lost if not configured with appropriate QoS settings	Guaranteed immediate processing upon request

Table 1: Comparison between ROS2 Topics and Services

When designing an AUV control framework, the choice between the topic interface and the service interface depends on the system's specific requirements. Topics are generally preferred for real-time data streams that require continuous updates, such as sensor readings and control signals. On the other hand, services are more appropriate for tasks that require immediate and transactional operations, such as changing system parameters or initiating certain actions based on specific conditions. Therefore, the ROS2-based AUV control system should use Services to process commands from other subsystems and Topics to publish continuous data, such as the status of the AUV.

2.7 Design Flow of the Project

In this project, the hardware systems for the AUV, such as the electrical and mechanical systems, are first developed. According to AUV hardware design and development literature [22], the design flow for AUV hardware systems can be classified into different stages as follows:

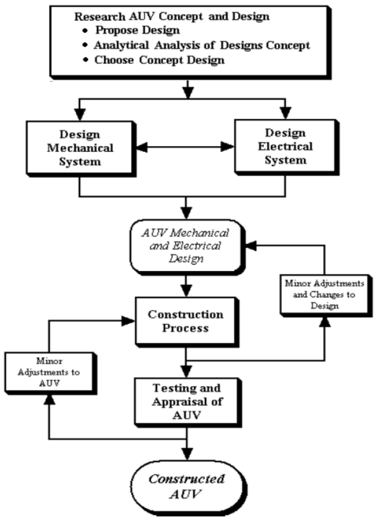

Research AUV Concept and Design
• Propose Design
• Analytical Analysis of Designs Concept
• Choose Concept Design
Design Mechanical System
Design Electrical System
AUV Mechanical and Electrical Design
Minor Adjustments and Changes to Design
Construction Process
Minor Adjustments to AUV
Testing and Appraisal of AUV
Constructed AUV

Figure 13: Design flow for an AUV

This block diagram shows that the AUV development proceeds with the design of the electrical system, such as designing the circuit diagram (power & signals, I/O control), and the design of the mechanical system, such as the AUV's frame, concurrently.

After the electrical system and the mechanical system are designed, the AUV construction begins. Components are bought or manufactured to assemble according to these systems. During the construction, these systems can also be adjusted to fulfil the requirements, such as increasing water resistance for circuit wiring.

When the electrical and mechanical systems are constructed, **basic controllable functions** are developed to test and commission these systems for further improvements, such as offsetting speed differences between AUV thrusters. Once basic control methods are well-developed, ROS2 client library (rclpy) APIs are utilized to develop ROS2 nodes with those control functions to establish the AUV control system based on the ROS2 communication layer.

3. Methodology

3.1 Hardware Development

3.1.1 Electrical System Design

Circuit Diagram

After the project research, this project's AUV contains the following characteristics:

1. An integrated computer system to process I/O data.
2. An embedded power source to power the AUV
3. Autonomous programs to control the AUV
4. Specialized Wireless Communication to communicate with the AUV
5. Physical actuators such as thrusters and buoyancy to control AUV's movements.

Therefore, this project's AUV composites of the following components:

- An Orange Pi 5 Plus computer board (OP5P) with Linux Ubuntu installed for data processing and controlling the AUV.
- A DC-DC Converter for stepping down voltage for electronic components.
- A Kill Switch acts as a relay for switching on/off DC-DC buck converter as well as the AUV.
- Electronic Speed Controllers (ESCs) for controlling AUV thrusters by PWM signals.
- Thrusters for propelling the AUV.
- Jump Wires for connecting pins on the OP5P to electronic components.
- A 4-Cell (4S) Battery for providing 14.8V voltage to the DC-DC Converter and thrusters.
- Fuse boxes to protect the AUV if short-circuited.
- A display screen to display the AUV system status.
- A 27MHz receiver
- A Raspberry Pi computer board (RPi) with Linux Debian installed (AUV buoyancy system)
- A Pressure Sensor for measuring underwater depth and temperature (AUV buoyancy system)
- A Pixhawk (AUV buoyancy system)
- A L298N Motor Driver (AUV buoyancy system)
- A USB Camera (AUV autonomous system).

The finalized AUV circuit design is given in below:

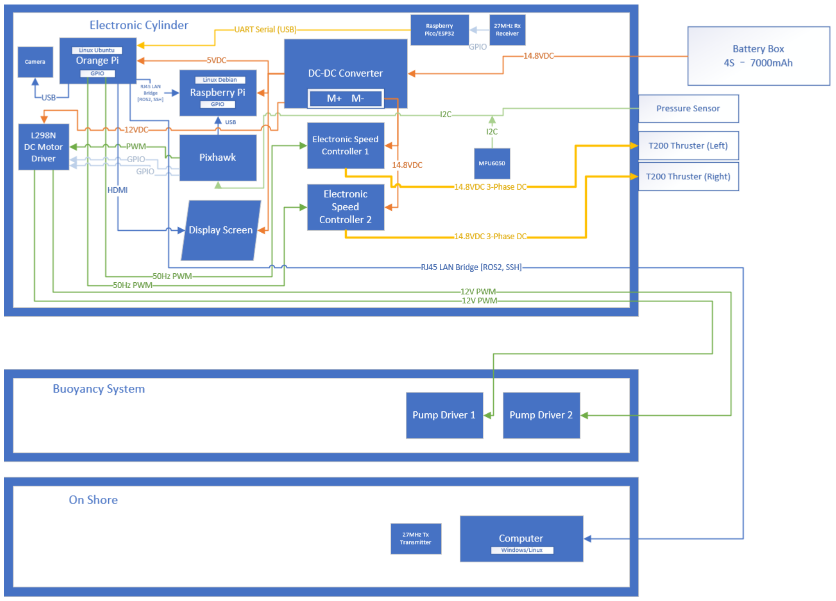


Figure 14: Electrical System for AUV

Figure 14 shows three areas: the area of the electronic cylinder, which stores the main electronic components; the area of the buoyancy system, which controls the floating capability of the AUV; and the area of onshore space. It is noted that electronic components for the AUV buoyancy subsystem and the AUV autonomous subsystem, such as the Pixhawk, L298N motor driver, MPU6050, and camera, are irrelevant to this project. However, they may still be discussed in the following sections for a better interpretation of this project goal.

In the electronic cylinder, the main computer board refers to the Orange Pi 5 Plus (OP5P), which has a high processing performance and operates ROS2 seamlessly. It connects the USB camera to capture images for autonomous programs. It also provides PWM channels for controlling ESCs to drive thrusters with different speeds. The DC-DC converter provides stepped-down output voltages (i.e. 5VDC and 12VDC) from the 4S battery (14.8VDC) for powering computer boards, display screen, and the motor driver for the buoyancy system. It also provides unchanged output voltages (14.8VDC) for ESCs and thrusters. A 27MHz receiver is used to receive 6-bit data from the onshore which is read by GPIO on a Raspberry Pi Pico (RPi Pico)/ESP32. The RPi Pico/ESP32 then transmits the 6-bit value to the OP5P through UART serial communication to establish a simple telecommunication. An internal display is installed inside the electronic cylinder to display console output messages from the OP5P through HDMI signals.

Apart from the electronic cylinder, the AUV's battery is preferably stored in an external box so that it can be replaced or recharged conveniently without accessing electronic components in the electronic cylinder. This approach increases the accessibility of the AUV and improves its hull design.

In the onshore area, the 27MHz transmitter is controlled manually to communicate with the receiver inside the AUV. A computer can also remotely control the OP5P and Raspberry Pi (RPi) by the secure shell (SSH) through an RJ45 ethernet cable.

Power System

The AUV power electronic devices are shown below:

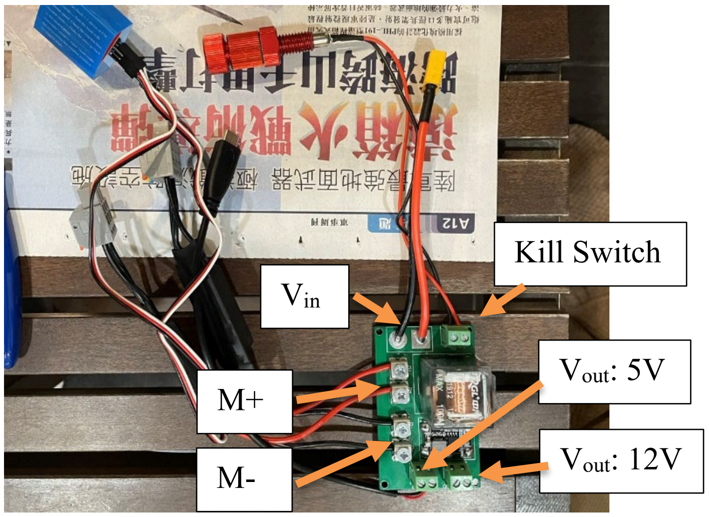


Figure 15: DC-DC Converter

The green board is the AUV DC-DC converter. The converter has the following ports:

1. An input source that is adapted by an XT60 socket. The XT60 socket is further connected to a power source such as a 4-cell battery (14.8VDC).
2. A 5VDC V_{out} terminal provides power to the OP5P, Raspberry Pi, and display screen.
3. A 12VDC V_{out} terminal provides power to the L298N motor driver for the buoyancy system.
4. 2 pairs of M+/M- outputs terminal (14.8VDC) which connects to two ESCs. Two thrusters are further connected to the two ESCs.
5. A kill switch relay is used to switch the DC-DC converter and the AUV on/off.

Furthermore, fuse boxes are installed between the DC-DC converter and the 4S battery. It is crucial to avoid short circuits inside the electronic cylinder. If the circuit is shorted and water leakage is present, water may be hydrolyzed to oxygen and hydrogen. The presence of these gases and short-circuited sparks may cause an explosion and damage to the AUV. Therefore, fuse boxes are equipped to break and protect the electrical circuit immediately once the AUV is shorted.

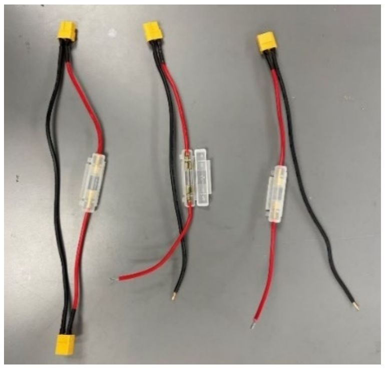


Figure 16: Wires with fuse boxes

Simple Thrusters Control by PWM Channels from OP5P

In this project, one of the objectives is to enable control of the AUV's thrusters through the Orange Pi 5 Plus. It is crucial to understand the working principles of AUV's thrusters first. Generally, AUV's thrusters are brushless (synchronous) motors and are excited by 3-phase DC [23].

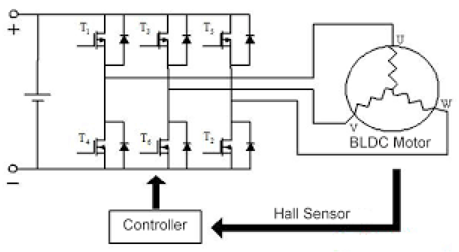


Figure 17: Control of a brushless DC motor [23]

When the 3-phase DC powers AUV's thrusters from the ESCs, their rotating speeds are controlled by PWM signals received on the ESCs. The PWM signals are square waves with specific switching frequencies and duty cycles.

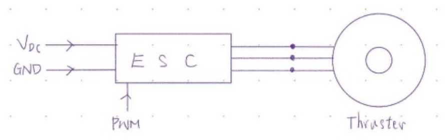
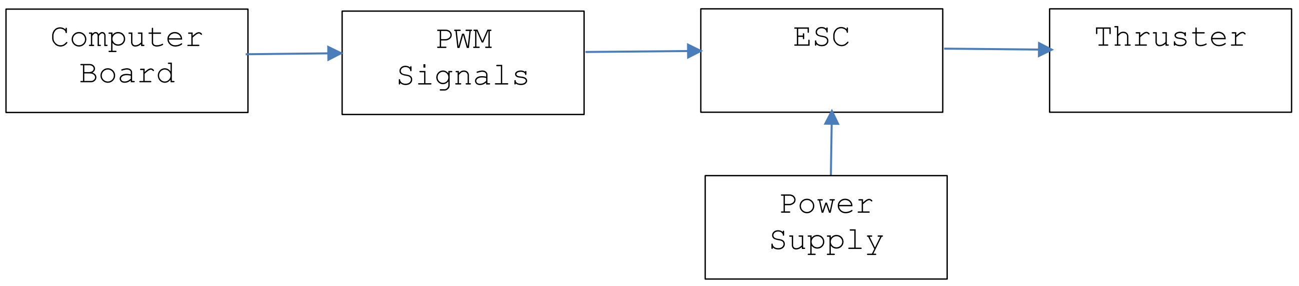


Figure 18: Electrical diagram of an ESC and a thruster.

Given the constant input voltage and constant input PWM switching frequency on an ESC, the thrusters' rotating speed increases when the PWM signal's duty cycle increases.



It is essential to figure out the correct PWM switching frequencies and duty cycles to control AUV thrusters correctly. After the investigation by using a servo tester and a portable oscilloscope, the operating conditions for the ESCs and the thrusters are shown below:

1. Thrusters will initialize and operate after the ESCs receive PWM signals of 50Hz switching frequency (f_s) and ~4.5-5% on-state duty cycle (D_{ON}).
2. Thrusters will rotate at maximum after the ESCs receive PWM signals of 50Hz f_s and ~10% D_{ON}.

Therefore, the OP5P must reserve two pins to output PWM signals with 50Hz f_s and ~5% D_{ON} to ESCs to initialize before controlling the AUV thrusters.

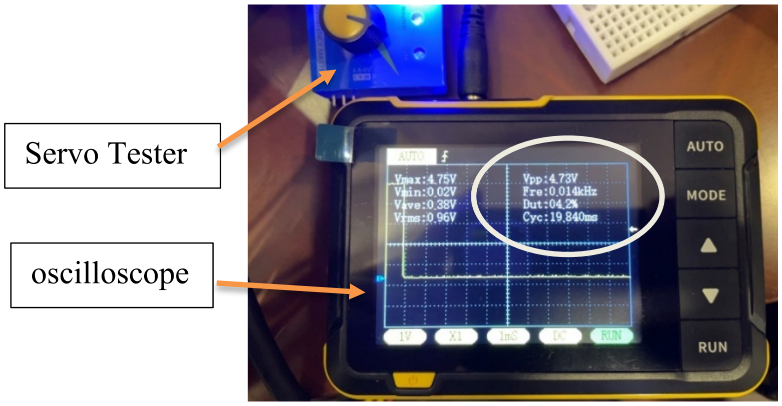
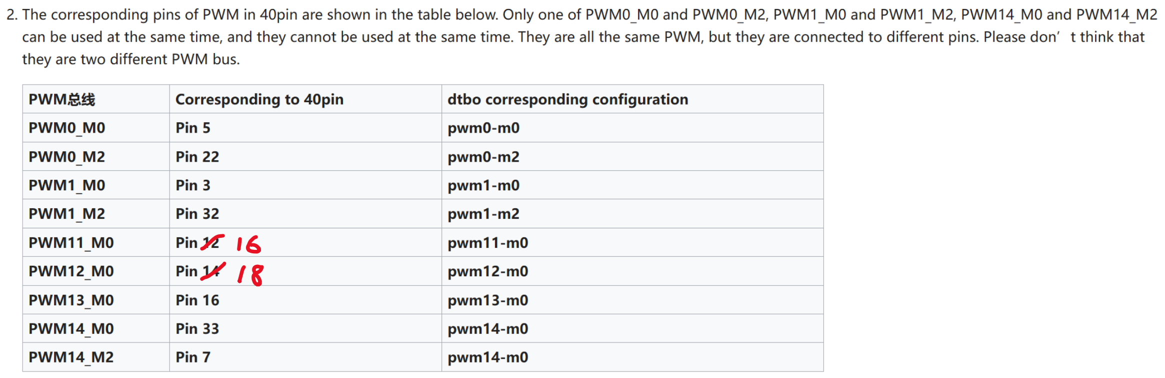


Figure 19: PWM Signal with 50Hz and 4% D_{ON}

According to official documentation [24], at most 6 PWM channels are available in the OP5P:



2. The corresponding pins of PWM in 40pin are shown in the table below. Only one of PWM0_M0 and PWM0_M2, PWM1_M0 and PWM1_M2, PWM14_M0 and PWM14_M2 can be used at the same time, and they cannot be used at the same time. They are all the same PWM, but they are connected to different pins. Please don' t think that they are two different PWM bus.

PWM总线	Corresponding to 40pin	dtbo corresponding configuration
PWM0_M0	Pin 5	pwm0-m0
PWM0_M2	Pin 22	pwm0-m2
PWM1_M0	Pin 3	pwm1-m0
PWM1_M2	Pin 32	pwm1-m2
PWM11_M0	Pin ~~12~~ 16	pwm11-m0
PWM12_M0	Pin ~~14~~ 18	pwm12-m0
PWM13_M0	Pin 16	pwm13-m0
PWM14_M0	Pin 33	pwm14-m0
PWM14_M2	Pin 7	pwm14-m0

Figure 20: PWM channels for OP5P [24]

Pin 16 and Pin 18 are selected for generating PWM signals. Pin 16 (PWM12_M0) is connected to the left thruster's ESC, and Pin 18 (PWM13_M0) is connected to the right thruster's ESC. It is noted that there are some mistakes in the official documentation. PWM12_M0 and PWM13_M0 should refer to Pin 16 and Pin 18, respectively.

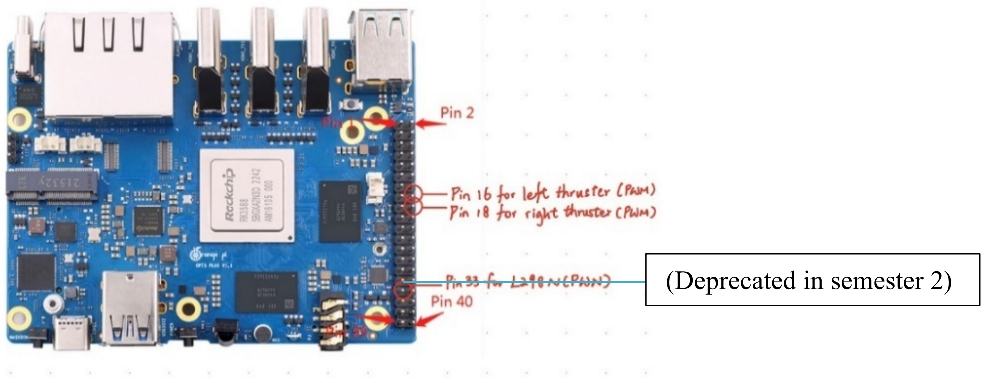
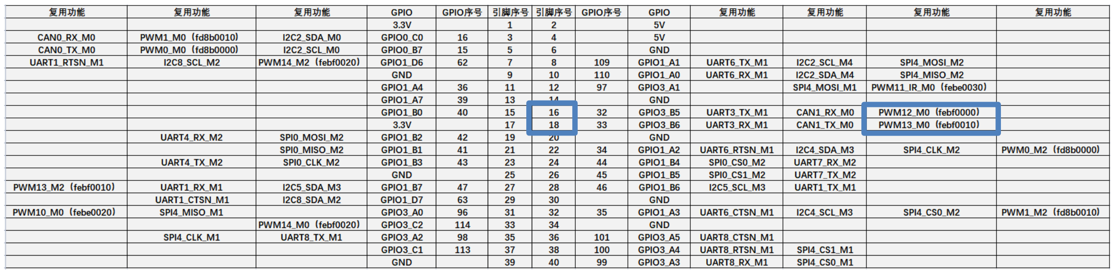

Pin 2
Pin 16 for left thruster (PWM)
Pin 18 for right thruster (PWM)
~~Pin 33 for L298N (PWM)~~
Pin 40
(Deprecated in semester 2)

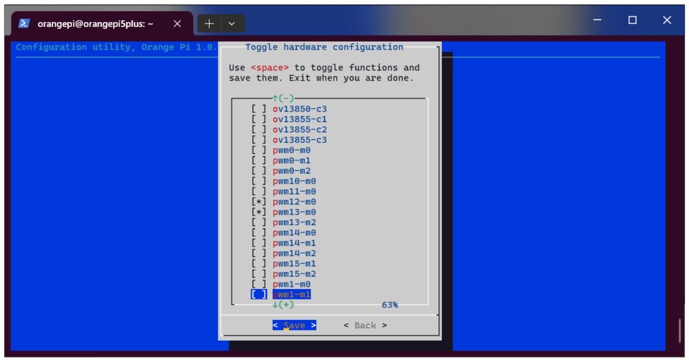
Figure 21: Pin connections for PWM outputs



复用功能	复用功能	复用功能	GPIO	GPIO序号	引脚序号	引脚序号	GPIO序号	GPIO	复用功能	复用功能	复用功能	复用功能
			3.3V		1	2		5V				
CAN0_RX_M0	PWM1_M0（fd8b0010）	I2C2_SDA_M0	GPIO0_C0	16	3	4		5V				
CAN0_TX_M0	PWM0_M0（fd8b0000）	I2C2_SCL_M0	GPIO0_B7	15	5	6		GND				
UART1_RTSN_M1	I2C8_SCL_M2	PWM14_M2（febf0020）	GPIO1_D6	62	7	8	109	GPIO1_A1	UART6_TX_M1	I2C2_SCL_M4	SPI4_MOSI_M2	
			GND		9	10	110	GPIO1_A0	UART6_RX_M1	I2C2_SDA_M4	SPI4_MISO_M2	
			GPIO1_A4	36	11	12	97	GPIO3_A1		SPI4_MOSI_M1	PWM11_IR_M0（febe0030）	
			GPIO1_A7	39	13	14		GND				
			GPIO1_B0	40	15	16	32	GPIO3_B5	UART3_TX_M1	CAN1_RX_M0	PWM12_M0（febf0000）	
			3.3V		17	18	33	GPIO3_B6	UART3_RX_M1	CAN1_TX_M0	PWM13_M0（febf0010）	
	UART4_RX_M2	SPI0_MOSI_M2	GPIO1_B2	42	19	20		GND				
		SPI0_MISO_M2	GPIO1_B1	41	21	22	34	GPIO1_A2	UART6_RTSN_M1	I2C4_SDA_M3	SPI4_CLK_M2	PWM0_M2（fd8b0000）
	UART4_TX_M2	SPI0_CLK_M2	GPIO1_B3	43	23	24	44	GPIO1_B4	SPI0_CS0_M2	UART7_RX_M2		
			GND		25	26	45	GPIO1_B5	SPI0_CS1_M2	UART7_TX_M2		
PWM13_M2（febf0010）	UART1_RX_M1	I2C5_SDA_M3	GPIO1_B7	47	27	28	46	GPIO1_B6	I2C5_SCL_M3	UART1_TX_M1		
	UART1_CTSN_M1	I2C8_SDA_M2	GPIO1_D7	63	29	30		GND				
PWM10_M0（febe0020）	SPI4_MISO_M1		GPIO3_A0	96	31	32	35	GPIO1_A3	UART6_CTSN_M1	I2C4_SCL_M3	SPI4_CS0_M2	PWM1_M2（fd8b0010）
		PWM14_M0（febf0020）	GPIO3_C2	114	33	34		GND				
	SPI4_CLK_M1	UART8_TX_M1	GPIO3_A2	98	35	36	101	GPIO3_A5	UART8_CTSN_M1			
			GPIO3_C1	113	37	38	100	GPIO3_A4	UART8_RTSN_M1	SPI4_CS1_M1		
			GND		39	40	99	GPIO3_A3	UART8_RX_M1	SPI4_CS0_M1		

Figure 22: GPIO Layout and PWM Pins for OP5P [24]

The 2 pins cannot output the PWM signals before they are activated as PWM channels. By setting **orangepi-config** [24] in Linux, the system reserves the 2 pins to generate PWM signals.


orangepi@orangepi5plus: ~
Configuration utility, Orange Pi 1.0.
Toggle hardware configuration

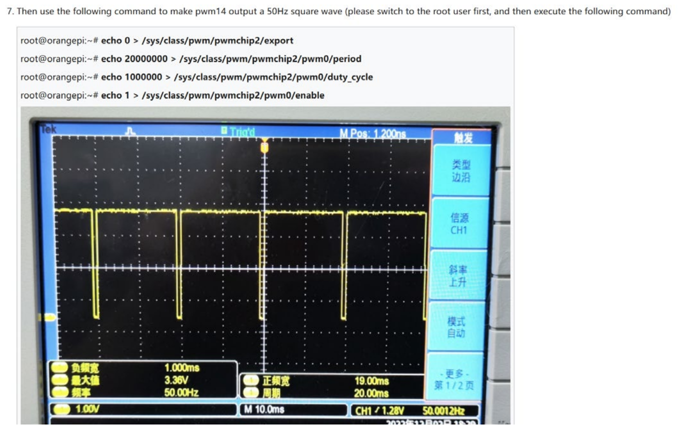
```
Use <space> to toggle functions and
save them. Exit when you are done.

        ↑(-)
    [ ] ov13850-c3
    [ ] ov13855-c1
    [ ] ov13855-c2
    [ ] ov13855-c3
    [ ] pwm0-m0
    [ ] pwm0-m1
    [ ] pwm0-m2
    [ ] pwm10-m0
    [ ] pwm11-m0
    [*] pwm12-m0
    [*] pwm13-m0
    [ ] pwm13-m2
    [ ] pwm14-m0
    [ ] pwm14-m1
    [ ] pwm14-m2
    [ ] pwm15-m1
    [ ] pwm15-m2
    [ ] pwm1-m0
    [ ] pwm1-m1
        ↓(+)                63%

        < Save >      < Back >
```

Figure 23: orangepi-config

After completing the setting, we can follow the official documentation [24] to output PWM signals:



7. Then use the following command to make pwm14 output a 50Hz square wave (please switch to the root user first, and then execute the following command)

```
root@orangepi:~# echo 0 > /sys/class/pwm/pwmchip2/export
root@orangepi:~# echo 20000000 > /sys/class/pwm/pwmchip2/pwm0/period
root@orangepi:~# echo 1000000 > /sys/class/pwm/pwmchip2/pwm0/duty_cycle
root@orangepi:~# echo 1 > /sys/class/pwm/pwmchip2/pwm0/enable
```

Figure 24: Official OP5P Documentation [24]

From the above documentation, four shell commands are required to trigger the PWM signal:

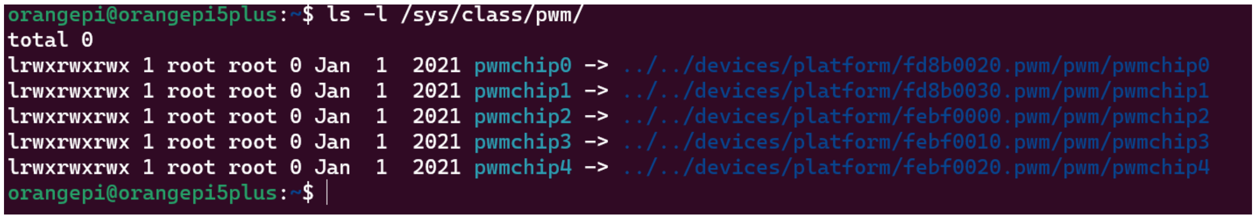
1. The first command is to write 0 into a Linux file named **"export"** to enable the PWM channel so that it is available for configuration and ready to control. We can treat it as an initialization of the PWM channel.
2. The second command is to write 1/50Hz*1E9=20000000 nanoseconds (ns) as a switching period into a Linux file named **"period"**.

3. The third command is to write 1/50Hz*1E9*5%=1000000 ns off-state duty time (T_{OFF}) into a Linux file named **"duty_cycle"**. Therefore, the PWM signal has an on-state duty ratio D_{ON}=95%.
4. The fourth command is to write "1" into a Linux file named **"enable"** to enable the PWM output signal. If "0" is written to this file, the PWM output signal is disabled.

It is noted that the 1000000 ns in the third step refers to the **off-state** duty time (T_{OFF}**)**, but not to the **on-state** time (T_{ON}). Therefore, we need to normalize the PWM duty time by:

$$T_{ON} = \frac{1}{f_s} * 1E9 * (1 - D_{ON})$$

To know which **pwmchip** belongs to which of the pins, we can list out **/sys/class/pwm/** in the bash shell:



```
orangepi@orangepi5plus:~$ ls -l /sys/class/pwm/
total 0
lrwxrwxrwx 1 root root 0 Jan  1  2021 pwmchip0 -> ../../devices/platform/fd8b0020.pwm/pwm/pwmchip0
lrwxrwxrwx 1 root root 0 Jan  1  2021 pwmchip1 -> ../../devices/platform/fd8b0030.pwm/pwm/pwmchip1
lrwxrwxrwx 1 root root 0 Jan  1  2021 pwmchip2 -> ../../devices/platform/febf0000.pwm/pwm/pwmchip2
lrwxrwxrwx 1 root root 0 Jan  1  2021 pwmchip3 -> ../../devices/platform/febf0010.pwm/pwm/pwmchip3
lrwxrwxrwx 1 root root 0 Jan  1  2021 pwmchip4 -> ../../devices/platform/febf0020.pwm/pwm/pwmchip4
orangepi@orangepi5plus:~$ 
```

Figure 25: Listing pwmchipx

To associate the above variables, such as 'fd8b0020' and 'febf00120', with specific pins, we consult the official documentation to identify which 'pwmchipx' corresponds to which pin (Figure []). In this example, 'pwmchip2' is associated with Pin 16, and 'pwmchip3' is associated with Pin 18. Consequently, we can create a straightforward Python test script that utilizes the previously mentioned shell commands to control AUV thrusters.

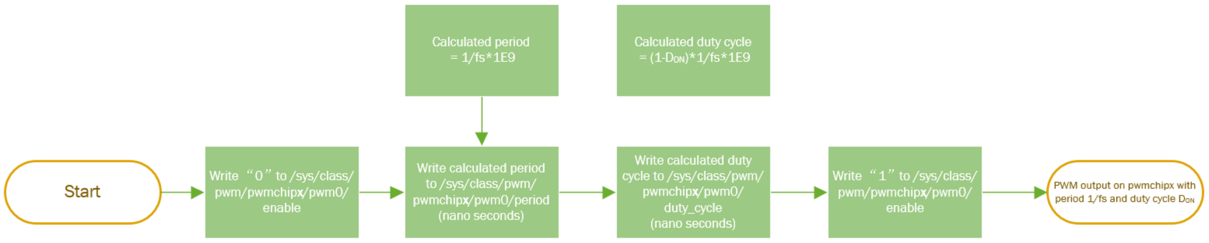
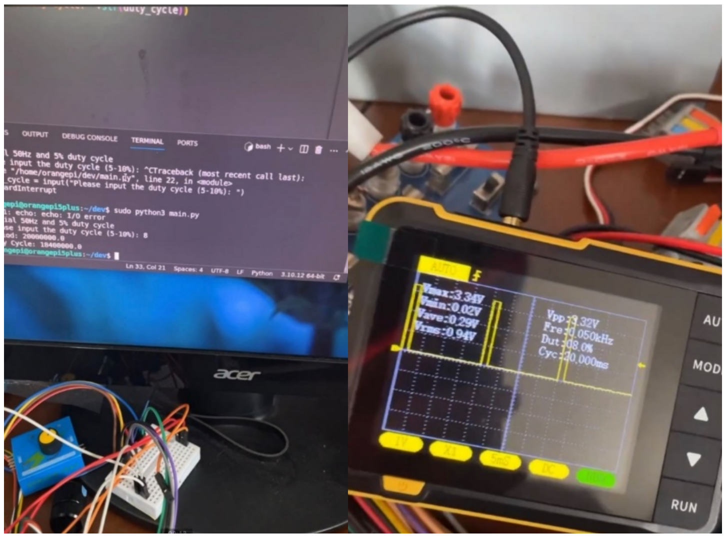

Calculated period = 1/fs*1E9
Calculated duty cycle = (1-D_ON)*1/fs*1E9
Start
Write "0" to /sys/class/pwm/pwmchipx/pwm0/enable
Write calculated period to /sys/class/pwm/pwmchipx/pwm0/period (nano seconds)
Write calculated duty cycle to /sys/class/pwm/pwmchipx/pwm0/duty_cycle (nano seconds)
Write "1" to /sys/class/pwm/pwmchipx/pwm0/enable
PWM output on pwmchipx with period 1/fs and duty cycle D_ON

Figure 26: Process of PWM Generations for OP5P

When the program starts, it firstly enables PWM channels in a pin and produces PWM signals with 50Hz f_s and 5% D_{ON} to initialize a thruster. Then, the program asks a user to input an on-state duty ratio D_{ON}, such as 8%. After inputting, the program calculates a corresponding off-state duty time T_{OFF} and writes these values into the Linux files. The OP5P consequently generates a PWM square wave with 50Hz f_s and 8% D_{ON} eventually.



```
l 50Hz and 5% duty cycle
 input the duty cycle (5-10%): ^CTraceback (most recent call last):
e "/home/orangepi/dev/main.py", line 22, in <module>
_cycle = input("Please input the duty cycle (5-10%): ")
ardInterrupt

gepi@orangepi5plus:~/dev$ sudo python3 main.py
1: echo: echo: I/O error
ial 50Hz and 5% duty cycle
se input the duty cycle (5-10%): 8
iod: 20000000.0
y Cycle: 18400000.0
ngepi@orangepi5plus:~/dev$
```

Figure 27: PWM signal outputs from the OP5P

With this approach, AUV thrusters can be controlled manually. A further function named **thruster_move()** is developed to control all AUV thrusters conveniently and would be used in the ROS2 development. The logic flow-chart of thruster_move() is shown in below:

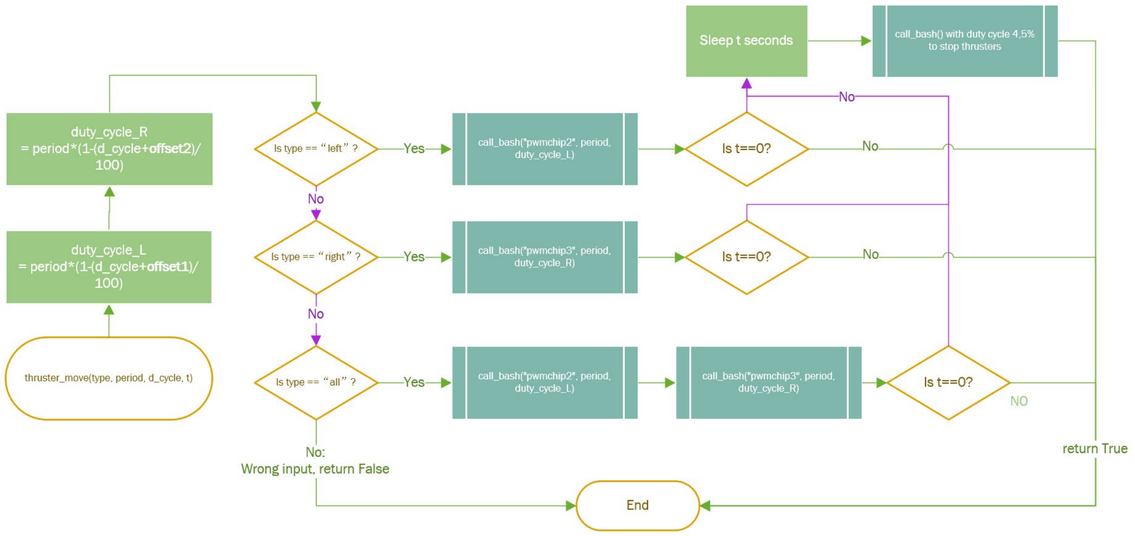
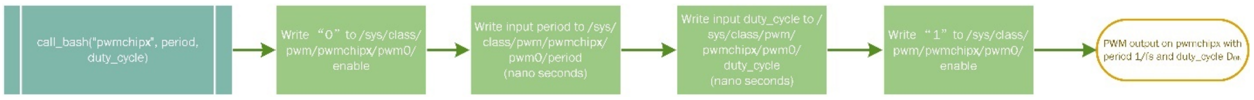

thruster_move(type, period, d_cycle, t)
duty_cycle_L = period*(1-(d_cycle+offset1)/100)
duty_cycle_R = period*(1-(d_cycle+offset2)/100)
Is type == "left" ?
Yes
No
Is type == "right" ?
Yes
No
Is type == "all" ?
Yes
No: Wrong input, return False
call_bash("pwmchip2", period, duty_cycle_L)
call_bash("pwmchip3", period, duty_cycle_R)
call_bash("pwmchip2", period, duty_cycle_L)
call_bash("pwmchip3", period, duty_cycle_R)
Is t==0?
No
Is t==0?
No
Is t==0?
NO
No
Sleep t seconds
call_bash() with duty cycle 4,5% to stop thrusters
return True
End

Figure 28: Flowchart for thruster_move()

Figure 28 shows that the thruster_move() function has 4 parameters: **type of thruster**, **PWM period (nanoseconds)**, **PWM on-state duty ratio D_{ON} (%)**, and **time**. When it is called, it first creates two normalized off-state duty times T_{OFF} (ns) named **d_cycle_L and d_cycle_R** based on the D_{ON} parameter and pre-defined offset values, as thrusters may rotate at different speeds under the same duty cycle. Then, the function determines the type of thrusters, such as **"left", "right", and "all"**. If the type is left, the function calls another function named **call_bash(),** which has parameters of **pwmchip2** (indicating Pin 16), **period** (PWM switching period in ns), and **duty_cycle_L** (normalized T_{OFF} for the AUV left thruster). Once the **call_bash()** is called, it implements similar tasks mentioned in **Figure 26** that write corresponding PWM parameters into the Linux files. Eventually, the AUV left thruster propels.


call_bash("pwmchipx", period, duty_cycle)
Write "0" to /sys/class/pwm/pwmchipx/pwm0/enable
Write input period to /sys/class/pwm/pwmchipx/pwm0/period (nano seconds)
Write input duty_cycle to /sys/class/pwm/pwmchipx/pwm0/duty_cycle (nano seconds)
Write "1" to /sys/class/pwm/pwmchipx/pwm0/enable
PWM output on pwmchipx with period 1/fs and duty_cycle Dnt
Figure 29: Flowchart for call_bash()

Similarly, the AUV right thruster propels when the type in **thruster_move()** is "right". In this case, the **thruster_move()** calls **call_bash()** with another group of parameters, such as **pwmchip3** (indicating Pin 18) and **duty_cycle_R** (normalized T_{OFF} for the AUV right thruster). When the type is "all", the **thruster_move()** calls **call_bash()** twice to propel two thrusters simultaneously.

Apart from these parameters, thruster_move() also provides a time parameter to allow the duration of propulsion. For example, if the time parameter **"t"** is 0, thruster(s) is turned on non-stop. If **t** is 5, thruster(s) will be turned off after 5 seconds.

In conclusion, thruster_move() offers a practical and efficient means of easily controlling AUV thrusters. This modular design not only simplifies the integration process but also facilitates its application in other systems. Furthermore, it can be seamlessly incorporated into subsequent ROS2 development efforts, such as for the more efficient design of an AUV control system based on ROS2 nodes.

3.1.2 Mechanical System Design

AUV Frame

A simple AUV frame was suggested. It was constructed by laser-cutting and assembled with teammates in semester 1.

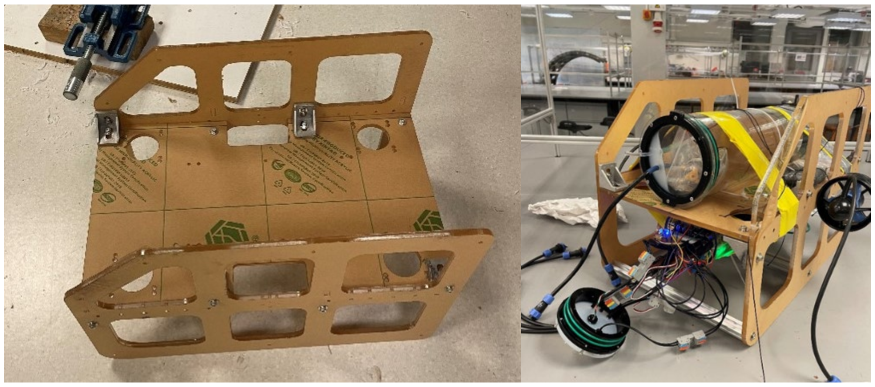


Figure 30: A Simple AUV Frame

Battery Box Design

A battery box made of a food box provides a waterproof connector to allow the 4S battery to power the AUV. This design improves the AUV's hull design.

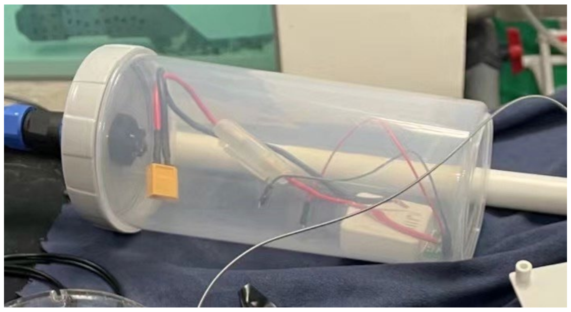


Figure 31: A Battery Box

Electronic Components mounting Rack

To manage the wiring and stabilize circuit connection, designing a mounting rack for electronic components such as the OP5P, the DC-DC converter, and the motor driver is practical.

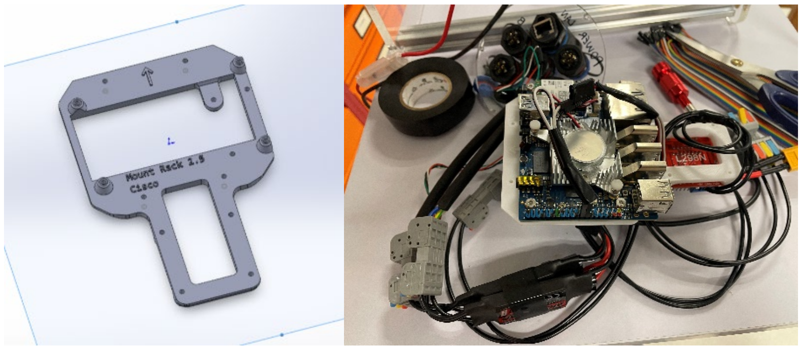


Figure 32: Electronic Components Mounting Rack

3.2 ROS2 Control System Framework

3.2.1 Design Ideas from Reverse Proxy

Before developing an AUV control system based on ROS2, it is crucial to understand the structure of the AUV system designed by other team members. For instance, in this project, the AUV system is classified into four subsystems: **the control system, the buoyancy subsystem, the thruster subsystem, and the autonomous subsystem.** The buoyancy subsystem is developed as a single node, while the autonomous subsystem is designed as a series of nodes, with each node responsible for a specific autonomous task. The AUV control system includes the thruster subsystem to directly control the AUV thrusters. This structure helps ensure seamless integration and effective collaboration within the development team.

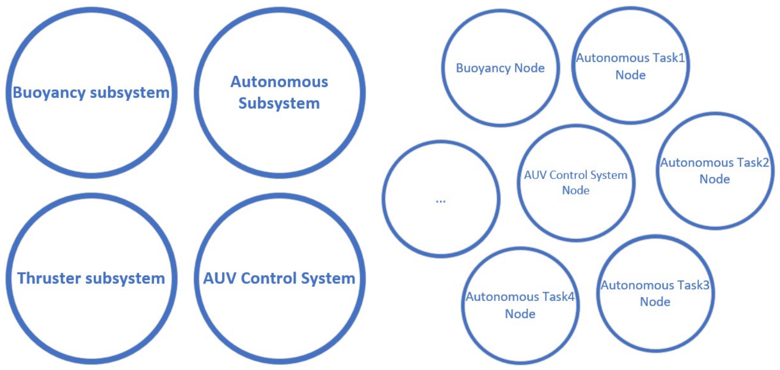


Figure 33: Abstracting AUV Subsystems to Nodes

As mentioned previously in the Background Section, it is more suitable to use the ROS2 Services interface to process logical operations that require immediate results, such as control commands from the autonomous subsystem. Therefore, the control system mainly uses the ROS2 Services interface to communicate with different nodes.

Below is a simple example of a ROS2-based control system with a single task node.

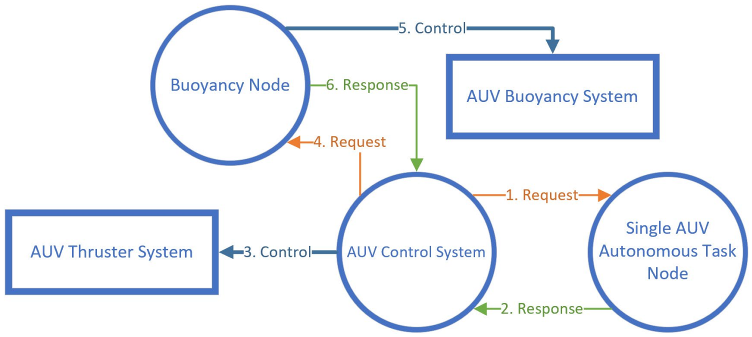


Figure 34: AUV Control System on a Single Autonomous Program (Node)

When the AUV control system uses the ROS2 Services interface to request one of the autonomous task nodes, such as Task1 Node (**Step 1**), the task node receives the request and starts to implement autonomous programs. After the processing, the task node responds with corresponding **control commands** for the AUV thruster subsystem and buoyancy subsystem (**Step 2**). When the control system receives the control commands from the response, it **implements** the commands. If the commands require the AUV to “move left”, “move right”, or “move forward”, the control system implements the previous **thruster_move()** function to turn on the “right” thruster or “left thrusters” or “all thrusters” respectively (**Step 3**) to control the thruster subsystem.

If the commands also require the AUV to “move upwards” or “move downwards”, the control system calculates a new target depth value based on an original depth value and requests the buoyancy node with the target depth value (**Step 4**). When the buoyancy node receives the request with the target depth value, it implements corresponding control for the buoyancy subsystem to allow the AUV to move to the desired depth (**Step 5**). After the implementation, the buoyancy responds with a success flag to acknowledge the control system in which the task is completed (**Step 6**). After the acknowledgement, the control system finishes the first implementation control cycle and commences the next cycle by requesting the autonomous node again. With this approach, the control system can handle commands and control the AUV's movements.

On the other hand, this framework **can only allow a single task node to perform a single autonomous task**. To tackle this problem, a concept of "**reverse proxy**" is introduced to the framework so that the control system can request commands from various task nodes based on specific conditions.

A reverse proxy is a server that sits between client devices and web servers, acting as an intermediary for requests [25]. Its primary function is to receive requests from clients, determine the appropriate web server to handle them, and then forward them accordingly. This process allows the reverse proxy to manage and balance the load on multiple servers, improve performance, and enhance security by providing an additional abstraction layer.

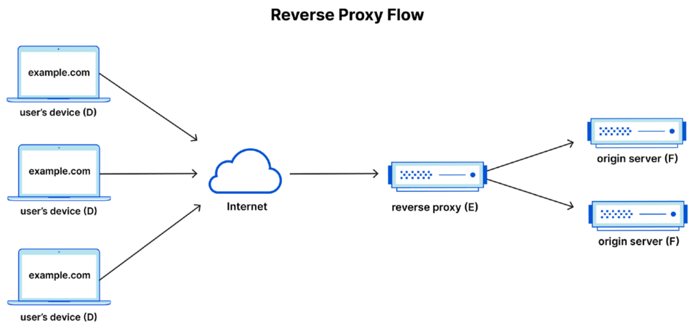


Figure 35: Reverse Proxy Flow [26]

Drawing inspiration from the intermediate connection and forwarding properties of reverse proxies, a similar approach is adopted in the design of a ROS2 framework for the AUV control system. In this framework, t**he AUV control system is separated into two nodes: a Control Node and a Master Node**. The Master Node serves as a central mediator and manager, receiving requests from the Control Node and determining the appropriate task node to handle. The Master Node then requests to the respective task nodes and manages the subsequent flow of responses from the task nodes such as control commands for the Control Node. The Control Node is the one that analysis and processes all control commands and converts them as movements and actions of the AUV by implementing hardware control.

To make the idea easier to interpret, a "matchstick men" example is illustrated below:

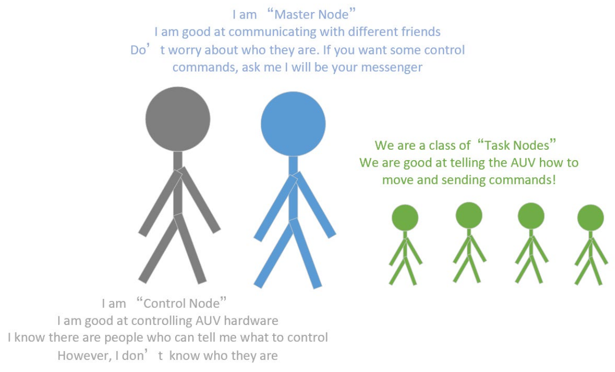

I am "Master Node"
I am good at communicating with different friends
Do't worry about who they are. If you want some control commands, ask me I will be your messenger
We are a class of "Task Nodes"
We are good at telling the AUV how to move and sending commands!
I am "Control Node"
I am good at controlling AUV hardware
I know there are people who can tell me what to control
However, I don't know who they are

Figure 36: Illustration of ROS2-based Nodes for the AUV Control System 1

The "Control Node" matchstick man has strength in controlling the AUV hardware, such as the thrusters and the buoyancy. However, he has a few friends (the Master Node and Buoyancy Node) and only knows how to control the AUV hardware when he receives control commands from others. The "Master Node" matchstick man is a socialholic who meets a lot of friends (task nodes and the Control Node). His main job is to be an intermediator to help the "Control Node" matchstick man ask the correct "Task Nodes" matchstick man (individual autonomous AUV programs) to get a proper command. If the Master Node finds a correct "Task Nodes" matchstick man and asks to obtain a correct control command, he will tell the "Control Node" matchstick man back.

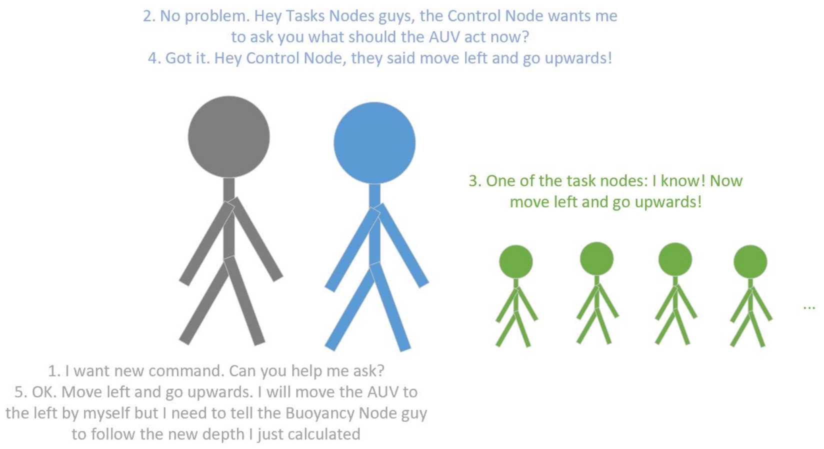

2. No problem. Hey Tasks Nodes guys, the Control Node wants me to ask you what should the AUV act now?
4. Got it. Hey Control Node, they said move left and go upwards!
3. One of the task nodes: I know! Now move left and go upwards!
...
1. I want new command. Can you help me ask?
5. OK. Move left and go upwards. I will move the AUV to the left by myself but I need to tell the Buoyancy Node guy to follow the new depth I just calculated

Figure 37: Illustration of ROS2-based Nodes for the AUV Control System 2

After the "Control Node" matchstick man gets the new control command, he will control the AUV thrusters (thruster subsystem) by himself. However, if the control command requires the AUV to move vertically (buoyancy subsystem), the "Control Node" matchstick man will calculate a new depth value for the AUV and ask the "Buoyancy Node" matchstick man to control the buoyancy subsystem.

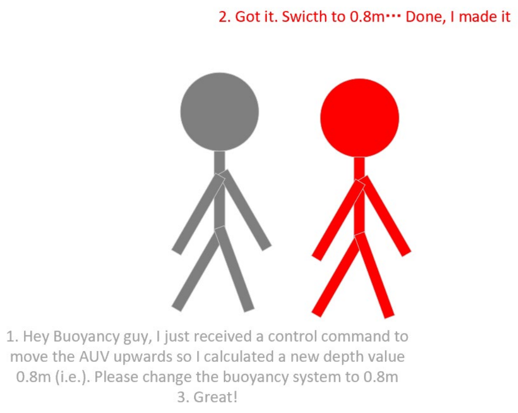

2. Got it. Swicth to 0.8m··· Done, I made it
1. Hey Buoyancy guy, I just received a control command to move the AUV upwards so I calculated a new depth value 0.8m (i.e.). Please change the buoyancy system to 0.8m
3. Great!

Figure 38: Illustration of ROS2-based Nodes for the AUV Control System 3

When the "Buoyancy Node" matchstick man replies to the "Control Node" matchstick man that the implementation is completed, the AUV control system finishes the first implementation cycle, and the "Control Node" matchstick man starts to ask the "Master Node" matchstick man for a new control command again.

Therefore, the AUV control system framework for a single implementation cycle is represented below:

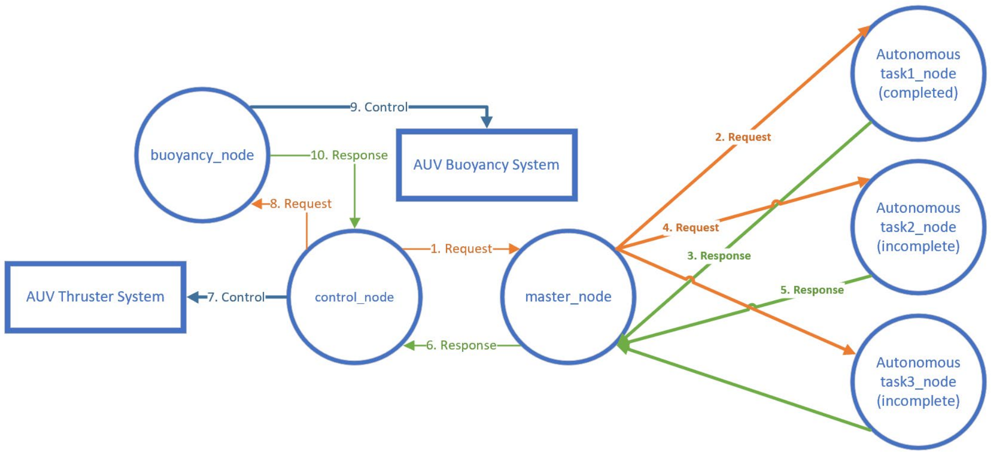


Figure 39: The ROS2-based AUV Control System Framework

There are 6 nodes in Figure 39. When the Control Node (control_node) requests the Master Node (master_node) to get control commands (**Step 1**), the master_node first acknowledges the request and sends another request to the Task 1 Node (task1_node) (**Step 2**). If task1_node has completed its task, it responds with a completed flag and control commands to the master_node (**Step 3**). When the master_node receives the completed flag, it automatically abandons the control commands and requests to the next task node, such as the Task 2 Node (task2_node) (**Step 4**). If the task2_node has not yet been completed, it will respond with an incomplete flag and control commands to the master_node (**Step 5**). When the master_node receives the incomplete flag from the task2_ndoe, it will treat the control commands as useful messages and forward them back to the control_node (**Step 6**). Eventually, the control_node gets the control commands from the task2_node and implements them (**Steps 7-10**).

Several advantages are realised by adopting a reverse proxy-inspired framework in the ROS2-based AUV control system. Firstly, the Master Node acts as a strategic coordinator, ensuring tasks are executed in a predetermined sequence. When the Control Node requests a task to be performed, the Master Node sends a request to the appropriate task node, such as the Task 1 Node. Upon receiving a response indicating completion, the Master Node incrementally requests the next task node, Task 2 Node, in the sequence, maintaining a controlled flow of operations.

This approach not only ensures that the AUV follows a logical and ordered sequence of task implementations but also simplifies the Control Node's responsibilities. The Control Node does not need to be aware of the current task number or the state of individual task nodes (the autonomous subsystem). It simply needs to communicate with the Master Node, which then handles the task distribution and status monitoring. This decoupling allows the Control Node to focus on its core function of controlling the thruster and buoyancy subsystem, while the Master

Node assumes the role of a sophisticated controller or manager within the AUV control system.

In summary, the reverse proxy-inspired framework streamlines the communication between the Control Node and task nodes, providing a structured and orderly execution of tasks. It also significantly reduces the complexity of the Control Node, allowing it to concentrate on hardware control. At the same time, the Master Node takes on the responsibility of orchestrating the data flow and task management within the ROS2 network.

3.2.2 Tree Structure of AUV ROS2 Control System

A simplified structure of the ROS2-based AUV Control System is introduced below.

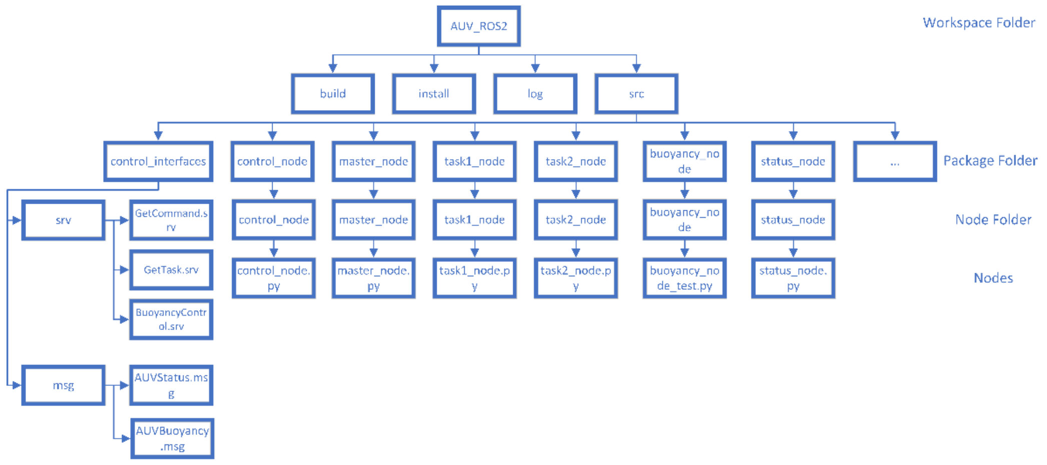
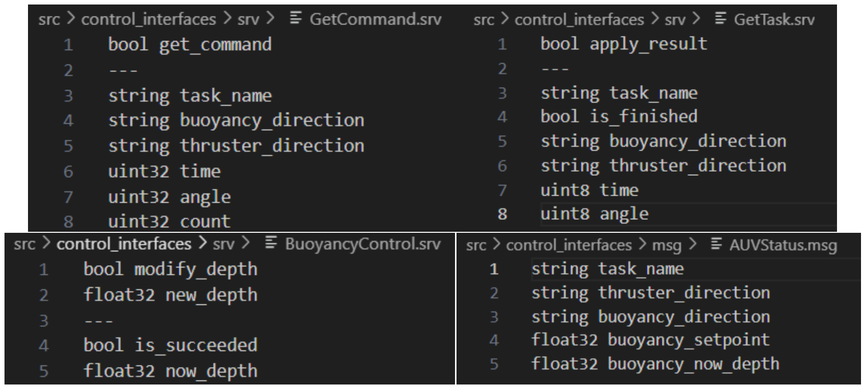


Figure 40: File System of ROS2-based AUV Control System

In the ROS2 development, all source files are stored in a folder named “**src**” under a main workspace folder (AUV_ROS2). Under the **“src”** folder, many package folders are present. One package file represents one node, except the **“control_interfaces”** file, which is used to define network interfaces between the nodes. Inside the node folders, Python programs written with ROS2 APIs are present. These programs are the main logic of the ROS2 nodes.

3.2.2 ROS2 Interfaces for AUV Control System

Customized network interfaces are defined within the **"control_interfaces"** directory to facilitate easy communication and data exchange between various nodes. These interfaces are designed to align with ROS2's conventions, enabling nodes to engage in ROS2 service-based communication using files like **"GetCommand.srv"** and **"GetTask.srv"** under **“srv”** for interactions between the **Control Node** and **Master Node**, and between the **Master Node** and **various task nodes**. Additionally, under the **"msg"** folder, files such as **"AUVStatus.msg"** and **"AUVBuoyancy.msg"** are also created to establish ROS2 topics-based communication channels. By adhering to the predefined data types within these interfaces, nodes can transmit data seamlessly.



src > control_interfaces > srv > GetCommand.srv

```
bool get_command
---
string task_name
string buoyancy_direction
string thruster_direction
uint32 time
uint32 angle
uint32 count
```

src > control_interfaces > srv > GetTask.srv

```
bool apply_result
---
string task_name
bool is_finished
string buoyancy_direction
string thruster_direction
uint8 time
uint8 angle
```

src > control_interfaces > srv > BuoyancyControl.srv

```
bool modify_depth
float32 new_depth
---
bool is_succeeded
float32 now_depth
```

src > control_interfaces > msg > AUVStatus.msg

```
string task_name
string thruster_direction
string buoyancy_direction
float32 buoyancy_setpoint
float32 buoyancy_now_depth
```

Figure 41: Customized ROS2 services-based and topics-based interfaces in the AUV control system.

3.2.3 Functionality of Control Node (control_node)

This section introduces the functionality and implementations of the Control Node in the AUV control system. The Control Node consists of 3 communication interfaces: one services-based interface for requesting control commands to the Master Node, one services-based interface for controlling the AUV buoyancy subsystem (the Buoyancy Node), and one topics-based interface for publishing AUV's overall status (the Status Node).

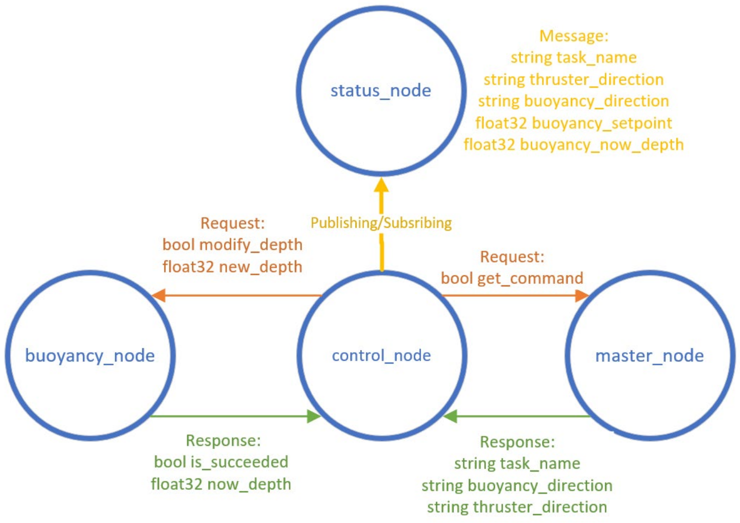


Figure 42: Network Interfaces on Control Node

Since ROS2 utilizes object-oriented programming to develop nodes, a node object and its constructor were created to initialize the three network interfaces.

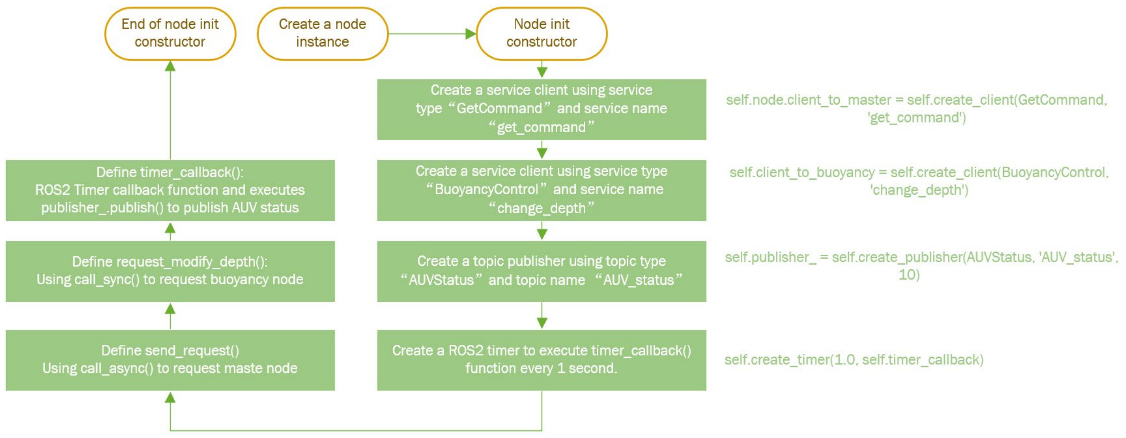

End of node init constructor
Create a node instance
Node init constructor
Create a service client using service type "GetCommand" and service name "get_command"
self.node.client_to_master = self.create_client(GetCommand, 'get_command')
Define timer_callback(): ROS2 Timer callback function and executes publisher_.publish() to publish AUV status
Create a service client using service type "BuoyancyControl" and service name "change_depth"
self.client_to_buoyancy = self.create_client(BuoyancyControl, 'change_depth')
Define request_modify_depth(): Using call_sync() to request buoyancy node
Create a topic publisher using topic type "AUVStatus" and topic name "AUV_status"
self.publisher_ = self.create_publisher(AUVStatus, 'AUV_status', 10)
Define send_request() Using call_async() to request maste node
Create a ROS2 timer to execute timer_callback() function every 1 second.
self.create_timer(1.0, self.timer_callback)

Figure 43: Python Constructor for Control Node

Figure 43 illustrates the constructor for the Control Node. The constructor firstly uses **rclpy** APIs to initialize 2 services-based clients and 1 topics-based publisher (rclpy.node.create_client, rclpy.node.create_publisher). A rclpy API named **timer** is used to create a timer_callback method. This method is triggered in the backend every 1 second to allow the publisher to publish messages periodically. After that, the constructor defines two request methods, such as a send_request() method, which requests the Master Node once with specific request contents (i.e. get_command = True). When any of these request methods are triggered, they will provide a "future" instance that is used to obtain control commands once the server returns a response (future.result).

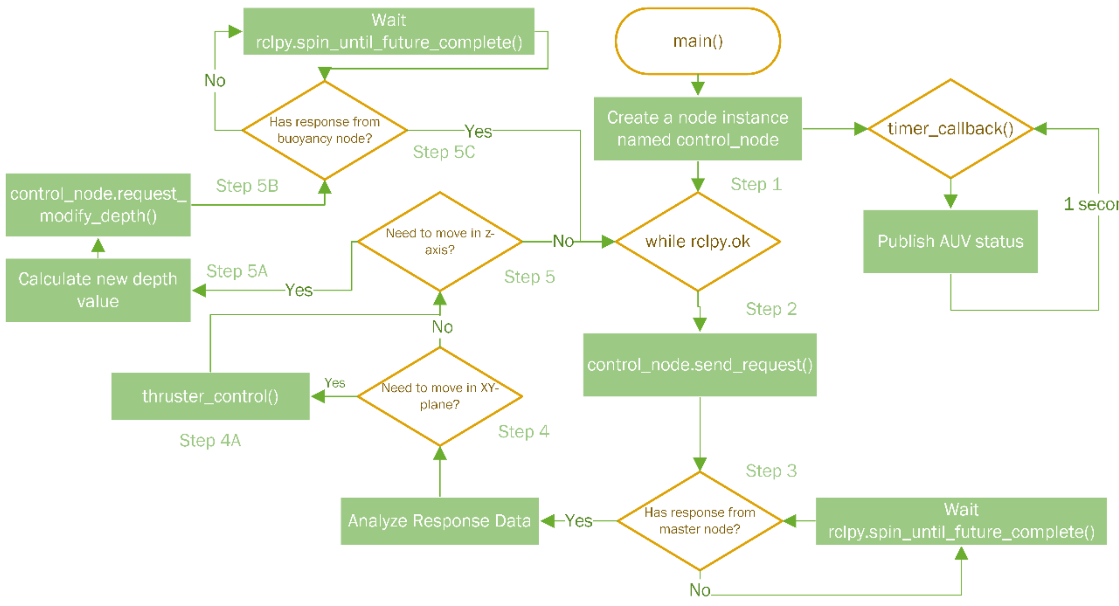

Wait rclpy.spin_until_future_complete()
main()
No
Has response from buoyancy node?
Yes
Step 5C
Create a node instance named control_node
timer_callback()
control_node.request_modify_depth()
Step 5B
Step 1
1 second
Need to move in z-axis?
No
while rclpy.ok
Publish AUV status
Calculate new depth value
Step 5A
Yes
Step 5
No
Step 2
control_node.send_request()
Yes
thruster_control()
Need to move in XY-plane?
Step 4
Step 4A
Step 3
Analyze Response Data
Yes
Has response from master node?
Wait rclpy.spin_until_future_complete()
No

Figure 44: Flowchart of Implementation Cycle for Control Node

Figure 44 illustrates an implementation cycle for the Control Node. When the **main** function starts, a Control Node instance is created (**Step 1**). The **timer_callback** method is therefore activated and publishes the AUV status in the background of the program every 1 second (**Figure 42 and Figure 45**). Hence, the Control Node publishes the AUV's status regularly, and it can be subscribed by any node, such as the **Status Node**, for various applications such as debugging and showing information on the AUV's display.

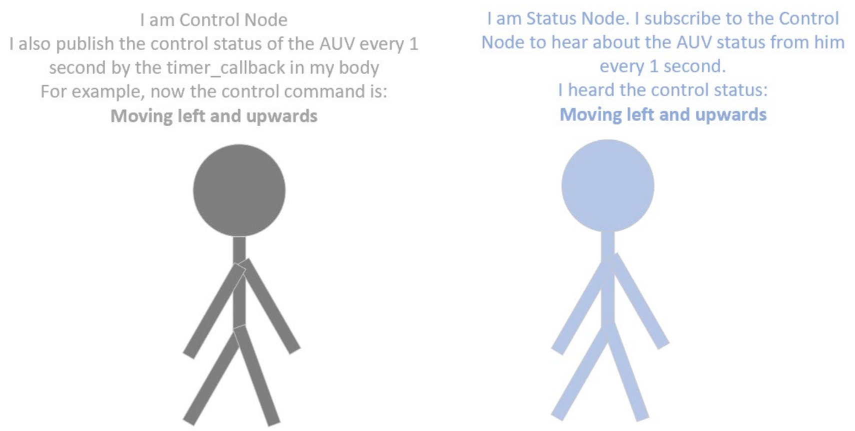

I am Control Node
I also publish the control status of the AUV every 1 second by the timer_callback in my body
For example, now the control command is:
Moving left and upwards
I am Status Node. I subscribe to the Control Node to hear about the AUV status from him every 1 second.
I heard the control status:
Moving left and upwards

Figure 45: Topics Interface between Control Node and Status Node

On the other hand, the main function enters a while-loop (rclpy.ok is **True** when the node is operating normally). In the while-loop, the Control Node first requests the Master Node (control_node.send_reuqest method) by sending a Boolean flag **get_command=True** to tell the master node that it is applying a control command (**Step 2**). The Control Node waits (spins) until it receives the response (**Step 3**). After the Master Node communicates with task nodes, it responds with a control command to the Control Node. The control command includes information in the form of string variables such as current autonomous task name (**task_name**), directions (**thruster_direction: Left/Right/Forward/None**) for AUV to move in the aspect of the xy-coordinate plane, and directions (**buoyancy_direction: Up/Down/Still/Still**) for AUV to move in the aspect of z-axis underwater. The request-response process is illustrated in **Figure 46**.

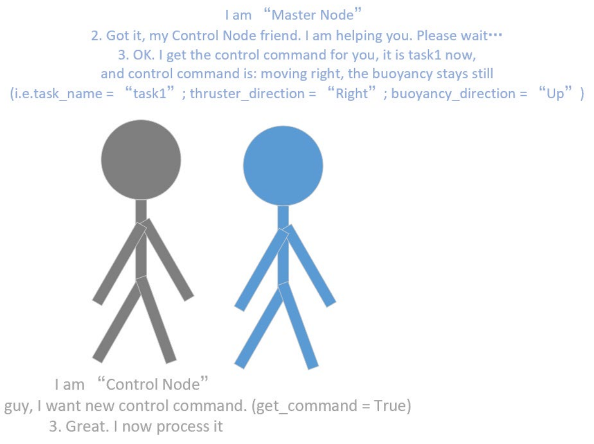

I am "Master Node"
2. Got it, my Control Node friend. I am helping you. Please wait…
3. OK. I get the control command for you, it is task1 now, and control command is: moving right, the buoyancy stays still
(i.e.task_name = "task1" ; thruster_direction = "Right" ; buoyancy_direction = "Up")
I am "Control Node"
1. Hey Master Node guy, I want new control command. (get_command = True)
3. Great. I now process it

Figure 46: Services Interface between Control Node and Master Node

After the main function in the Control Node receives the control command, it starts to analyze and implement it. If the control command requires the AUV to move in the aspect of the xy-coordinate plane (**Step 4**), the Control Node calls the thruster control function **thruster_move()** which is developed in the previous section to establish the movement (**Step 4A**). Suppose the control command requires the AUV to move in the z-axis (**Step 5**). In that case, the Control Node calculates a new depth value by a constant increment (i.e. old depth value

+0.1) or decrement (i.e. old depth value-0.1) (**Step 5A**) and sends a request with a Boolean flag **modify_depth=True** and a floating point number new_depth equivalent to the newest depth value to the Buoyancy Node to apply a modification for the AUV's depth (**Step 5B**). The request-response process is illustrated in **Figure 47**.

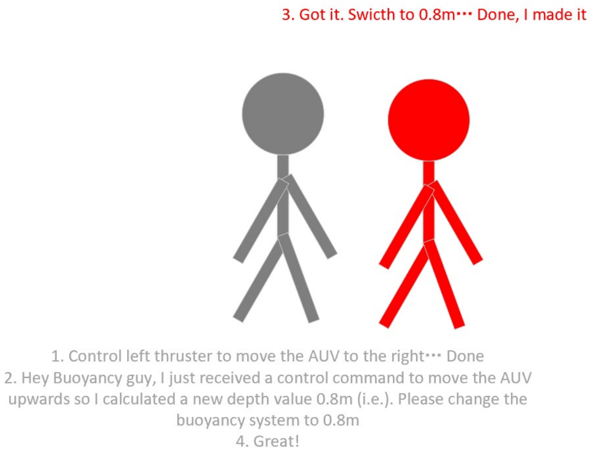

3. Got it. Swicth to 0.8m… Done, I made it
1. Control left thruster to move the AUV to the right… Done
2. Hey Buoyancy guy, I just received a control command to move the AUV upwards so I calculated a new depth value 0.8m (i.e.). Please change the buoyancy system to 0.8m
4. Great!

Figure 47: Services Interface between Control Node and Buoyancy Node

When the Buoyancy Node receives the modification to change the AUV's depth, the Buoyancy Node complies with the command and sets a new depth. It then implements its programs to adjust the buoyancy subsystem, and finally returns a success flag and a current depth value to the control node (**Step 5C**). Once these steps are implemented, the first implementation cycle of the AUV control system is said to be completed. The process continues to repeat until the Control Node is shut down manually.

3.2.4 Functionality of Master Node (master_node)

This section introduces the functionality and implementations of the Master Node in the AUV control system. Firstly, the Master Node consists of 2 communication interfaces: one services-based interface for requesting control commands from task nodes and one for forwarding responses from task nodes to the Control Node. The network interfaces are shown below:

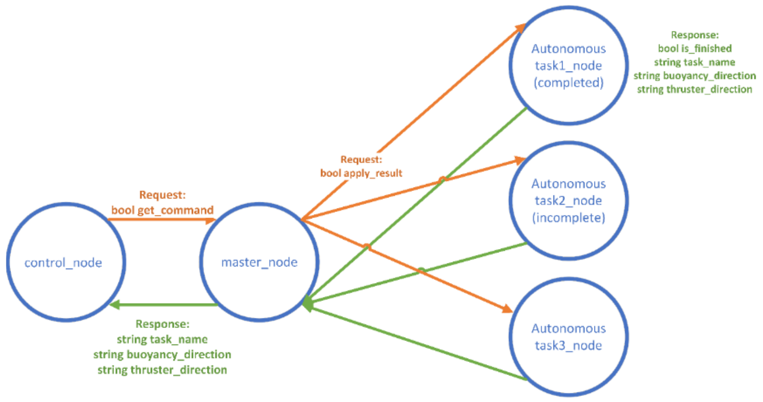
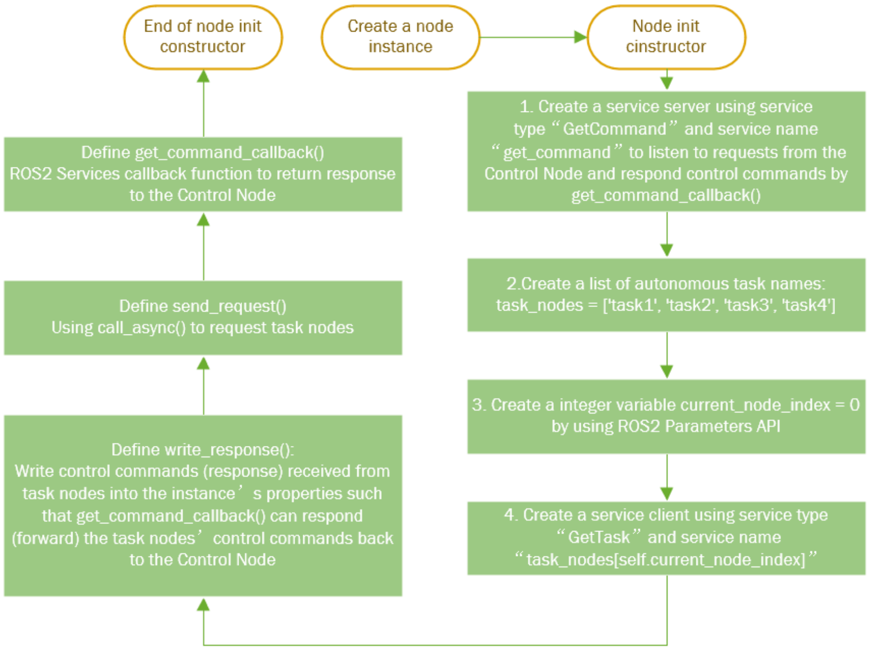

control_node
master_node
Request:
bool get_command
Response:
string task_name
string buoyancy_direction
string thruster_direction
Request:
bool apply_result
Autonomous task1_node (completed)
Autonomous task2_node (incomplete)
Autonomous task3_node
Response:
bool is_finished
string task_name
string buoyancy_direction
string thruster_direction

Figure 48: Network Interfaces on Master Node

Similar to the Control Node, a node object and a constructor are created to initialize the network interfaces in the Master Node when the Master Node starts.


End of node init constructor
Create a node instance
Node init cinstructor
1. Create a service server using service type "GetCommand" and service name "get_command" to listen to requests from the Control Node and respond control commands by get_command_callback()
Define get_command_callback() ROS2 Services callback function to return response to the Control Node
2.Create a list of autonomous task names: task_nodes = ['task1', 'task2', 'task3', 'task4']
Define send_request() Using call_async() to request task nodes
3. Create a integer variable current_node_index = 0 by using ROS2 Parameters API
Define write_response(): Write control commands (response) received from task nodes into the instance's properties such that get_command_callback() can respond (forward) the task nodes' control commands back to the Control Node
4. Create a service client using service type "GetTask" and service name "task_nodes[self.current_node_index]"

```
Step 1. self.srv = self.create_service(GetCommand, 'get_command', self.get_command_callback)
Step 2. self.task_nodes = ['task1', 'flare_detect', 'task3', 'task4']  # Add all task nodes Service names in order
Step 3. self.current_node_index = self.declare_parameter('task_index', 0).value
Step 4. self.client = self.create_client(GetTask, self.task_nodes[self.current_node_index])
```

Figure 49: Python Constructor for Master Node

Figure 49 illustrates the constructor for the Master Node. The constructor first uses **rclpy** APIs to initialize a services-based server (rclpy.node.create_service). This serviced-based server listens to any request with service type "GetCommand" and service name "get_command". If the Master Node receives the request from the Control Node with the same service type and service name, it will implement a callback method named get_command_callback to return a response.

Then, the constructor creates a list named **task_nodes**, which stores all task nodes' ROS2 service names, as each task node contains a unique service name. **The unique service name identifies each node to be requested or responded to.** An integer variable named **current_node_index** is created with a default value 0 by ROS2 Parameters API. This index is then used to **fetch the service name of the current task node from the task_nodes list**. This approach enables the Master Node to **initialize a service-based client that requests to the specified task node for retrieving control commands.** If the Control Node sends a request to the Master Node, the Master Node utilizes this index to decide which task node the request should be routed to. Therefore, the current_node_index plays a crucial role in facilitating the communication between the Master Node and the appropriate task node, ensuring that each request is directed to the correct destination for task execution (flow control).

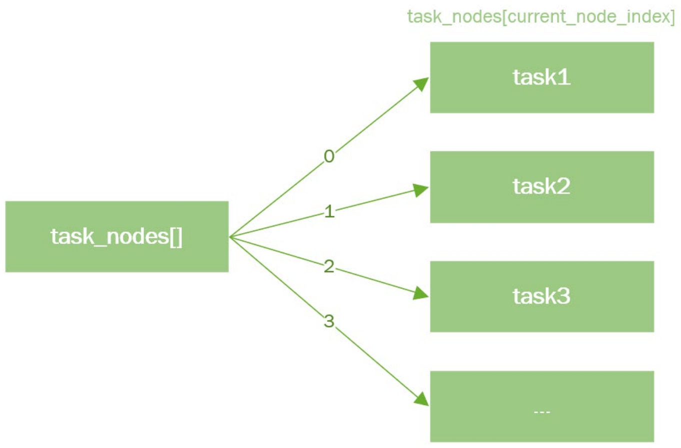
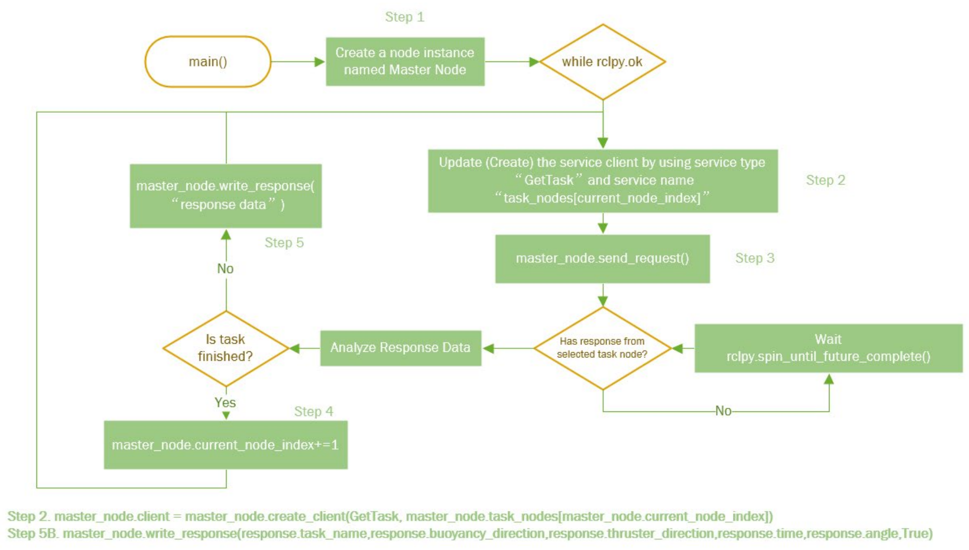


Figure 50: task_nodes List with Service Names

Take **Figure 50** as an example. The first service name to request is “**task1**”. The Master Node will, therefore, create a services-based client with service type “**GetTask**” and service name “**task1**” to request a task node with the same service type and service name to forward control commands to the Control Node.

In addition to the current_node_index variable, a **write_response** method is created within the constructor in the node class. This function accepts multiple parameters, enabling the main function to pass variables stored within it to the properties of the node instance. By doing so, the get_command_callback method can access response data from the task nodes temporarily stored in the node instance. The method then uses the data to formulate a response, which is forwarded back to the Control Node. Therefore, the **write_response** method facilitates the communication between the Master Node and the Control Node by providing a means to store and retrieve response data, ensuring that the Control Node receives the necessary information to proceed with its control operations.

Similar to the Control Node, the send_request method is also created in the constructor to request a specified task node. If triggered, it provides a “future” instance used to obtain control commands when the task node returns a response (future.result).


Step 1
main()
Create a node instance named Master Node
while rclpy.ok
Update (Create) the service client by using service type "GetTask" and service name "task_nodes[current_node_index]"
Step 2
master_node.send_request()
Step 3
Wait rclpy.spin_until_future_complete()
Has response from selected task node?
No
Analyze Response Data
Is task finished?
Yes
Step 4
master_node.current_node_index+=1
No
Step 5
master_node.write_response("response data")

Step 2. master_node.client = master_node.create_client(GetTask, master_node.task_nodes[master_node.current_node_index])
Step 5B. master_node.write_response(response.task_name,response.buoyancy_direction,response.thruster_direction,response.time,response.angle,True)

Figure 51: Flowchart of Implementation Cycle for Control Node

Figure 51 illustrates an implementation cycle for the Master Node. When the main function starts, a Master Node instance is first created (**Step 1**). Then, the main function enters into the "rclpy.ok" while-loop. In the while loop, the main function re-updates the service-based client with the latest service name, even though the client has already been initialized in the constructor (**Step 2**). The Master Node then starts to listen to any request from the Control Node. According to the **task_node[]** list, the main function implements the **send_request** method to request the first task node (task1_node) with a Boolean flag **apply_result=True** to the master node to tell the task node that it is applying a control command. Then, the Master Node spins to wait for the response (**Step 3**). When the task node receives the request, it implements its autonomous programs. After the implementation, the task node responds with a control command back to the Master Node. The control command includes information in the form of a Boolean flag (**is_finished**) which represents the status of the task node (**True if task is completed, and vice versa**), and string variables such as current autonomous task name (**task_name**), directions (**thruster_direction: Left/Right/Forward/None**) for AUV to move in the aspect of xy-coordinate plane, and directions **(buoyancy_direction: Up/Down/Still/Still)** for AUV to move in the aspect of z-axis underwater.

After receiving the response from the task node, the Master Node starts to analyze the response data. If the Master Node finds that the task has been completed (is_finished = True), the main function will **increment the current_node_index by 1, indicating the next service name should be used** (**Step 4**). The Master Node will re-update the service client with the same service type, "GetTask", and a new service name (i.e. task2) in the subsequent implementation cycle so that a new task node will be requested. If the main function finds that the task has not been completed (is_finished=False), the main function implements the **write_response** method (**Step 5**) to store the response data as node's properties, allowing **get_command_callback** method to read the response in the node's properties and forward it to Control Node. Once these steps are implemented, the first implementation cycle of the Master Node is said to be completed. The whole process continues to repeat until the Master Node is shut down.

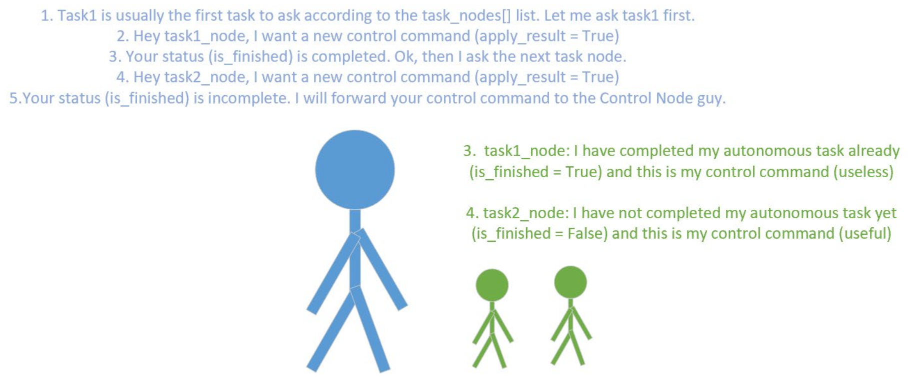

1. Task1 is usually the first task to ask according to the task_nodes[] list. Let me ask task1 first.
2. Hey task1_node, I want a new control command (apply_result = True)
3. Your status (is_finished) is completed. Ok, then I ask the next task node.
4. Hey task2_node, I want a new control command (apply_result = True)
5.Your status (is_finished) is incomplete. I will forward your control command to the Control Node guy.
3. task1_node: I have completed my autonomous task already (is_finished = True) and this is my control command (useless)
4. task2_node: I have not completed my autonomous task yet (is_finished = False) and this is my control command (useful)

Figure 52: Illustration of Implementation Steps 1-5

3.2.5 Simulating Nodes (task1_node, task2_node, buoyancy_node_test)

To verify whether the Control Node can request and receive autonomous control commands from task nodes through the Master Node, it is essential to develop test methods to simulate the control system. Therefore, two simulated task nodes (Task 1 Node and Task 2 Node) were developed for this project.

Although the Task 1 Node (task1_node) and Task 2 Node (task2_node) have different node names and service names, they share the same structure and function, allowing users to input customized control commands to respond to the Master Node. In the constructor's aspect (**Figure 53**), the constructor first uses **rclpy** APIs to initialize a services-based server (rclpy.node.create_service). This serviced-based server listens to any request with service type "**GetTask**" and service name **"task1"**. When the Task 1 Node receives the request from the Master Node with the same service type and service name, it will implement the callback method **get_task_callback** to return a response. Additionally, a **write_response** method is created to allow simulated control commands inputted by users to be stored in the node's properties, allowing the **get_task_callback** method to read the control commands and respond to the Master Node.

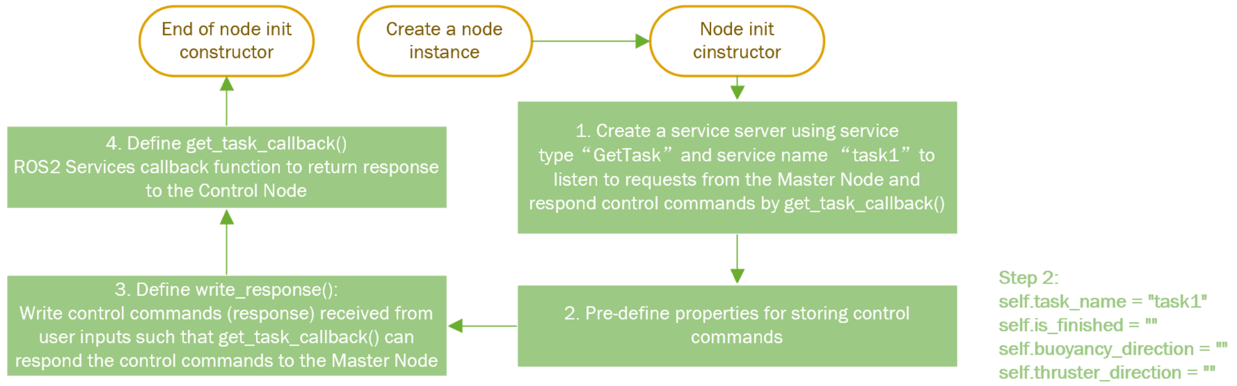

Create a node instance
Node init cinstructor
1. Create a service server using service type "GetTask" and service name "task1" to listen to requests from the Master Node and respond control commands by get_task_callback()
2. Pre-define properties for storing control commands
Step 2:
self.task_name = "task1"
self.is_finished = ""
self.buoyancy_direction = ""
self.thruster_direction = ""
3. Define write_response(): Write control commands (response) received from user inputs such that get_task_callback() can respond the control commands to the Master Node
4. Define get_task_callback() ROS2 Services callback function to return response to the Control Node
End of node init constructor

Figure 53: Python Constructor for Simulating Task 1Node

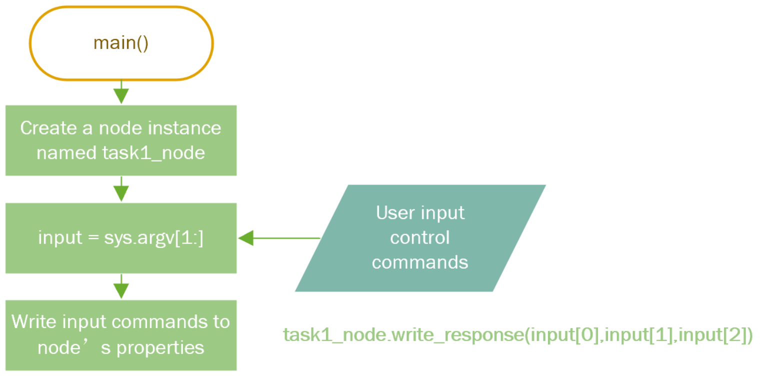

main()
Create a node instance named task1_node
input = sys.argv[1:]
User input control commands
Write input commands to node’s properties
task1_node.write_response(input[0],input[1],input[2])

Figure 54: Flowchart of Implementation Cycle for Task 1 Node

Figure 54 demonstrates the Implementation Cycle for a simulated Task 1 Node. A Task 1 Node instance is first created when the main function starts. After that, it reads 3 arguments from the user input. The first input value refers to a Boolean flag for the task node status (Boolean: True/False indicating the task node is completed/incomplete). The second input value refers to a control command for the buoyancy subsystem (String: Up/Down/Still indicating the AUV moves in the z-axis). The third input value refers to a control command for the thruster subsystem. (String: Left/Right/Forward indicating the AUV moves in the xy-coordinate plane). With this approach, we can simulate a task node and send control commands to the Control Node through the Master Node to control the AUV.

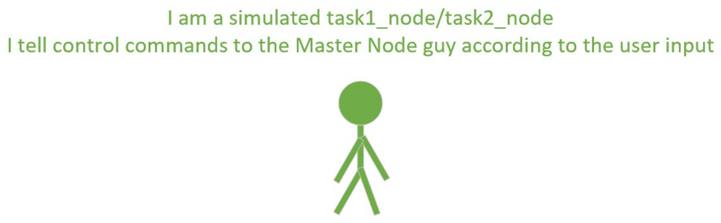

I am a simulated task1_node/task2_node
I tell control commands to the Master Node guy according to the user input

Figure 55: Illustration of Functionality of Simulated Task 1 Node

Additionally, a simulated buoyancy node is developed to receive and respond to requests from the Control Node. The simulated buoyancy node is similar to the above task1_node, which allows users to input two variables to respond to the Control Node (a Boolean flag: True/False to tell the Control Node that the buoyancy task is implemented and a floating point number of the current AUV’s depth value read from a pressure sensor.

3.2.6 Application of ROS2 Parameters API

The ROS2 Parameters API is a powerful feature that allows users to manage and configure parameters for their nodes dynamically. Parameters are key-value pairs that store configuration data for a node, such as sensor calibration values, control values, or other user-defined settings. The Parameters API provides a standardized way to set, get, and monitor these parameters, making it easier to manage and maintain the configuration of a ROS2 system.

In this project, ROS2 Parameters API is utilized in the Master Node. The ROS2 Parameters API establishes the following functions:

- 1. Launching the Master Node to request task nodes in different orders (startup).
- 2. Switching the Master Node to request indicated task nodes (runtime).

<u>Launching Master Node to Request Task Nodes in Different Orders</u>

As mentioned previously, the variable "current_node_index" is assigned by the ROS2 Parameter API which allows users to define a specific value when the Master Node is started up:

self.current_node_index = self.declare_parameter('task_index', 0).value

The above declare_parameter method consists of two parameters. The first parameter is the parameter name of the declaration (task_index), and the second parameter is the parameter's default value (0). The ".value" at the end is used to retrieve the value of the parameter after it has been declared. Therefore, **the current_node_index is assigned to 0 by default**.

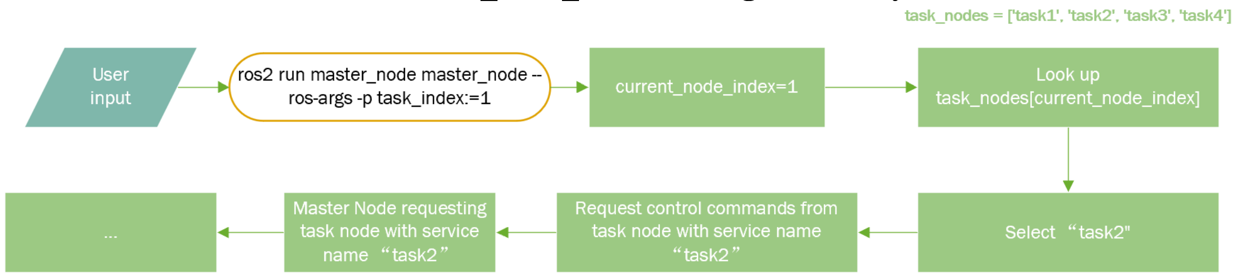

task_nodes = ['task1', 'task2', 'task3', 'task4']
User input
ros2 run master_node master_node --ros-args -p task_index:=1
current_node_index=1
Look up task_nodes[current_node_index]
Select "task2"
Request control commands from task node with service name "task2"
Master Node requesting task node with service name "task2"
...

Figure 56: Flowchart of Using ROS2 Parameters API to Start Master Node

Take **Figure 56** as an example. When users require the Master Node to request the second task node (i.e. task2) firstly, the users can run the following command:

ros2 run master_node master_node --ros-args -p task_index:=1

to configure the value of "current_node_index". In this scenario, this command is used to run a ROS2 node named "master_node" from a package also named "master_node", and it includes a ROS argument (**--ros-args**) that sets the value of a parameter **(-p)** named task_index to 1 (**task_index:=1**) when the node starts. Therefore, the "current_node_index" would be assigned to 1, and the Master Node would request the task node with the services name "task2" (task_nodes[1] = "task2") according to **Figure 50**.

If the users start the Master Node without inputting arguments:

ros2 run master_node master_node

Since the "current_node_index" is assigned to 0 by default, the Master Node will request the task node with the service name "task1" instead of "task2" (task_nodes[0]= "task1").

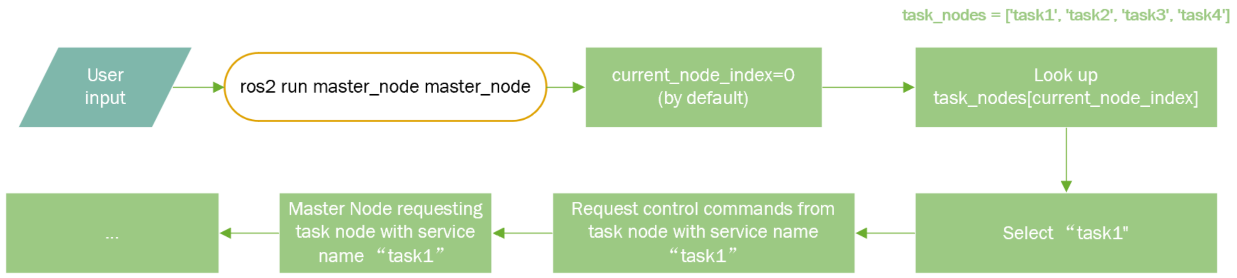

task_nodes = ['task1', 'task2', 'task3', 'task4']
User input
ros2 run master_node master_node
current_node_index=0 (by default)
Look up task_nodes[current_node_index]
Select "task1"
Request control commands from task node with service name "task1"
Master Node requesting task node with service name "task1"
...

Figure 57: Flowchart of Starting Master Node by Default

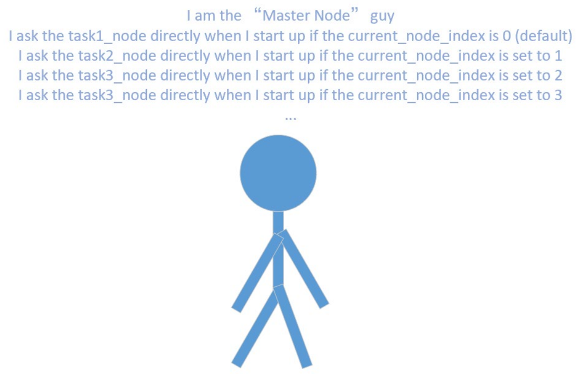

I am the "Master Node" guy
I ask the task1_node directly when I start up if the current_node_index is 0 (default)
I ask the task2_node directly when I start up if the current_node_index is set to 1
I ask the task3_node directly when I start up if the current_node_index is set to 2
I ask the task3_node directly when I start up if the current_node_index is set to 3
...

Figure 58: Illustration of Functionality of ROS2 Parameters on Master Node.

Therefore, the Master Node's requesting order can be configured by using the ROS2 Parameters API when it is started up. The AUV can implement autonomous tasks in different orders according to the operators' requirements.

Switching Master Node to Request Indicated Task Nodes at Runtime

Apart from using the ROS2 Parameters method to configure the Master Node at startup, users can also configure the value of "current_node_index" at runtime, which will automatically update the corresponding task node that the master node should communicate with.

To establish this function, we can use a ROS2 API method **"add_on_set_parameters_callback(parameter_callback)"** to register a callback function that will be triggered whenever a parameter is changed. In this case, a callback function named "parameter_callback" is created. When the value of the parameter "task_index" is modified at run time by inputting the command:

"**ros2 param set /master_node task_index [value]**",

the callback function will assign the "current_node_index" with the new value. This function allows for dynamic reconfiguration of the Maste Node without the need to restart it, enhancing the flexibility and robustness of the system.

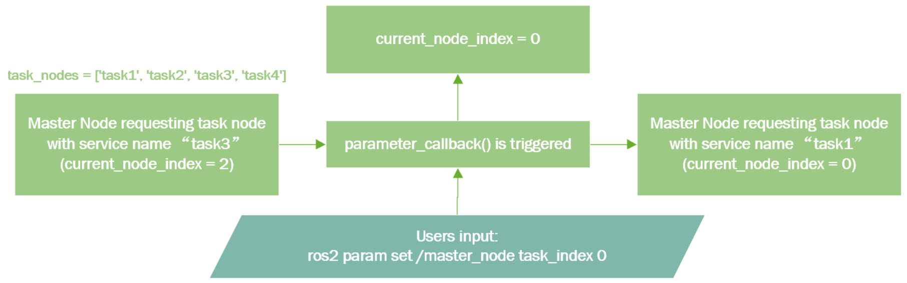
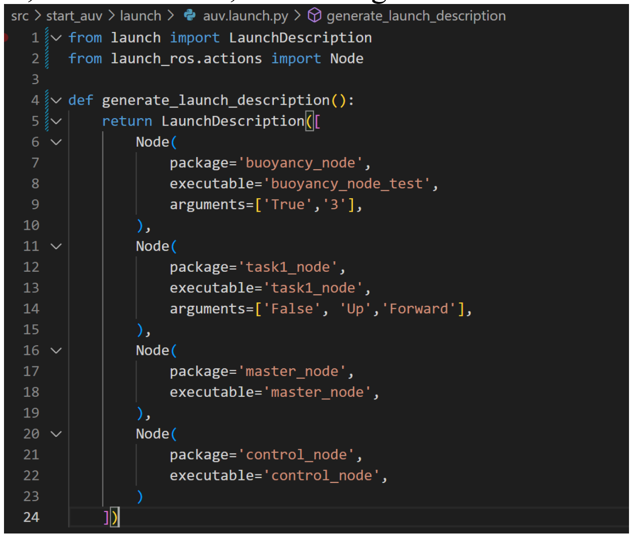

current_node_index = 0
task_nodes = ['task1', 'task2', 'task3', 'task4']
Master Node requesting task node with service name "task3" (current_node_index = 2)
parameter_callback() is triggered
Master Node requesting task node with service name "task1" (current_node_index = 0)
Users input: ros2 param set /master_node task_index 0

Figure 59: Flowchart of Using ROS2 Parameters API to Configure Master Node's Requesting Order at Runtime

In conclusion, the ROS2 Parameters API is a valuable tool for managing and configuring nodes in a ROS2 system. The main benefit of using the ROS2 Parameters API is the ability to dynamically configure and reconfigure nodes at startup and runtime. This means that users can change the behaviour of a node without needing to restart or rebuild the program, which can be particularly useful during development and testing for the AUV's autonomous programs, as the implementing order for autonomous programs can be altered dynamically.

3.2.7 Application of ROS2 Launch Method

After the ROS2 nodes for the AUV control system are developed, it is difficult to start these nodes one by one (one Linux terminal per node). Therefore, the ROS2 Launch method can be used to start all nodes in one command. The ROS2 Launch method requires developers to configure a list of ROS2 nodes to launch. In this configuration, it is needed to configure the package name of the node, the node name, and the argument of the node.


src > start_auv > launch > auv.launch.py > generate_launch_description

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='buoyancy_node',
            executable='buoyancy_node_test',
            arguments=['True','3'],
        ),
        Node(
            package='task1_node',
            executable='task1_node',
            arguments=['False', 'Up','Forward'],
        ),
        Node(
            package='master_node',
            executable='master_node',
        ),
        Node(
            package='control_node',
            executable='control_node',
        )
    ])
```

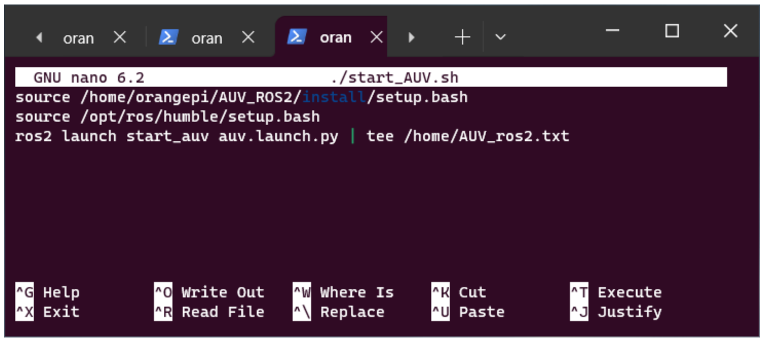
Figure 60: Example of ROS2 Launch Configuration

Once the ROS2 Launch is configured, all desired ROS2 nodes can be theoretically launched in a command:

ros2 launch start_auv auv.launch.py

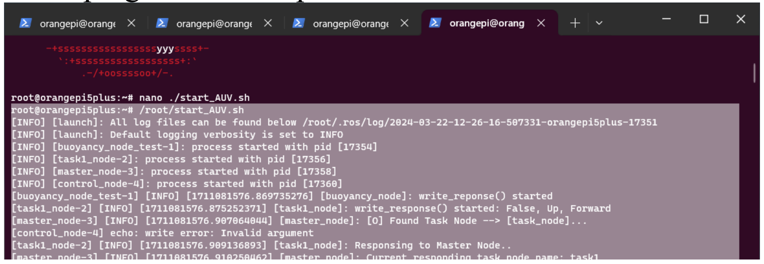
However, since it is found that the Control Node requires **root permissions** to manipulate

PWM hardware control in the OP5P and the /root/.bashrc file does not take effects to source the ROS2 setup files, a simple shell script was created to start the AUV system as the root user in Linux:



```
  GNU nano 6.2                    ./start_AUV.sh
source /home/orangepi/AUV_ROS2/install/setup.bash
source /opt/ros/humble/setup.bash
ros2 launch start_auv auv.launch.py | tee /home/AUV_ros2.txt
```

Figure 61: Developing a Shell Script to Launch ROS2 Nodes in Root Environment



```
root@orangepi5plus:~# nano ./start_AUV.sh
root@orangepi5plus:~# /root/start_AUV.sh
[INFO] [launch]: All log files can be found below /root/.ros/log/2024-03-22-12-26-16-507331-orangepi5plus-17351
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [buoyancy_node_test-1]: process started with pid [17354]
[INFO] [task1_node-2]: process started with pid [17356]
[INFO] [master_node-3]: process started with pid [17358]
[INFO] [control_node-4]: process started with pid [17360]
[buoyancy_node_test-1] [INFO] [1711081576.869735276] [buoyancy_node]: write_reponse() started
[task1_node-2] [INFO] [1711081576.875252371] [task1_node]: write_response() started: False, Up, Forward
[master_node-3] [INFO] [1711081576.907064044] [master_node]: [0] Found Task Node --> [task_node]...
[control_node-4] echo: write error: Invalid argument
[task1_node-2] [INFO] [1711081576.909136893] [task1_node]: Responsing to Master Node..
[master_node-3] [INFO] [1711081576.910250462] [master_node]: Current responding task node name: task1
```

Figure 62: Demonstration of ROS2 Launch

Consequently, The OP5P can start all ROS2 nodes in the root environment. The AUV control system starts to read commands from the autonomous subsystem and manipulate the thruster and buoyancy subsystems.

3.2.8 Frequency Control in Control Node

The Control Node plays a pivotal role in coordinating and managing the various subsystems of an underwater vehicle, particularly the buoyancy subsystem. The buoyancy subsystem, crucial for the vehicle's vertical movement, must be controlled with precision to ensure stable and reliable operation. However, the physical nature of buoyancy adjustments introduces inevitable delays and inertia, which must be accounted for in the Control Node's design.

To address this challenge, the Control Node incorporates a frequency control mechanism. This strategy ensures that buoyancy adjustment commands are not sent to the buoyancy node with every cycle but after every fifth cycle (~3-4 seconds). This delay is intentional and designed to allow the buoyancy system sufficient time to process the previous command and exhibit the desired physical change.

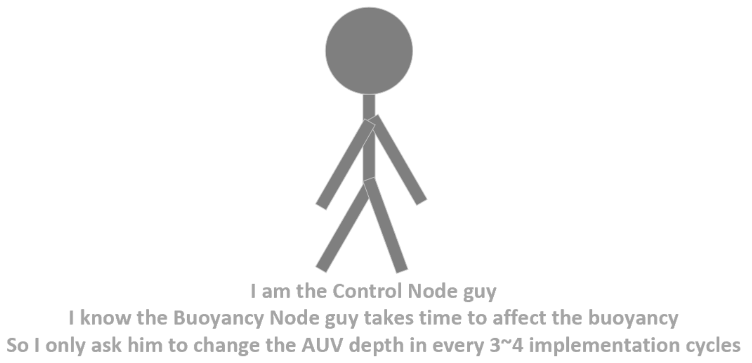

I am the Control Node guy
I know the Buoyancy Node guy takes time to affect the buoyancy
So I only ask him to change the AUV depth in every 3~4 implementation cycles

Figure 63: Illustration of Frequecy Control in Control Node

Take the block diagram in **Figure 64** as an example. A variable named **buoyancy_count=0** is initialized in the constructor of the Control Node. When the Control Node determines whether it is needed to move in the z-axis (Step 5), it checks the value of the buoyancy_count first. If the buoyancy_count is smaller than 5, it is incremented by 1, and the Control Node skips requesting the Buoyancy Node to apply for modifying a depth change. The Control Node therefore ends the current implementation cycle and starts a new cycle. On the other hand, if the buoyancy_count is equal to or greater than 5, the Control Node determines to process control commands for the buoyancy node and request to the buoyancy_node. After that, the Control Node resets the buoyancy_count to 0 before starting a new implementation cycle.

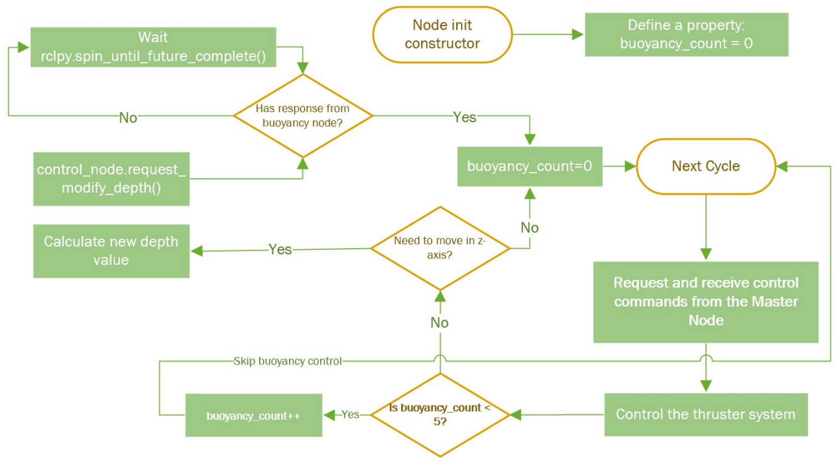

Wait
rclpy.spin_until_future_complete()
Node init
constructor
Define a property:
buoyancy_count = 0
No
Has response from buoyancy node?
Yes
control_node.request_modify_depth()
buoyancy_count=0
Next Cycle
No
Calculate new depth value
Yes
Need to move in z-axis?
Request and receive control commands from the Master Node
No
Skip buoyancy control
buoyancy_count++
Yes
Is buoyancy_count < 5?
Control the thruster system

Figure 64: Flowchart of Frequency Control in Control Node

As a result, the Control Node only deals with the buoyancy's control commands every 5 cycles with this approach, establishing the frequency control mechanism to allow the buoyancy subsystem sufficient time to behave.

3.2.9 Response Control in Master Node

Gate Control in get_command_callback Method

In the previous section, we mentioned that ROS2 Services callback methods, such as the **get_command_callback** in the Master Node and the **get_task_callback** in task nodes, will be triggered if servers receive requests. However, these callback functions need to be implemented in an orderly manner. For example, the get_command_callback method should not be implemented to respond to the Control Node until the get_task_callback methods in task nodes are implemented to respond to the Master Node. Otherwise, the Master Node may respond with incorrect or outdated control commands to the Control Node.

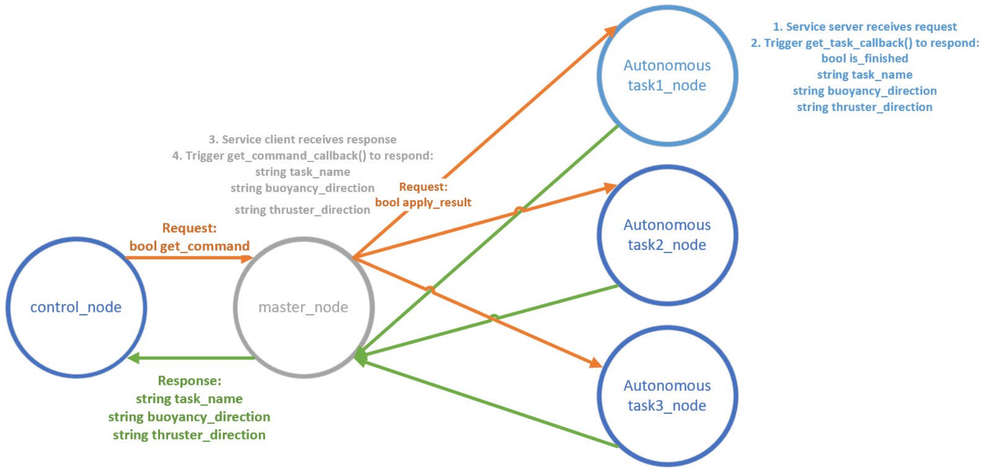
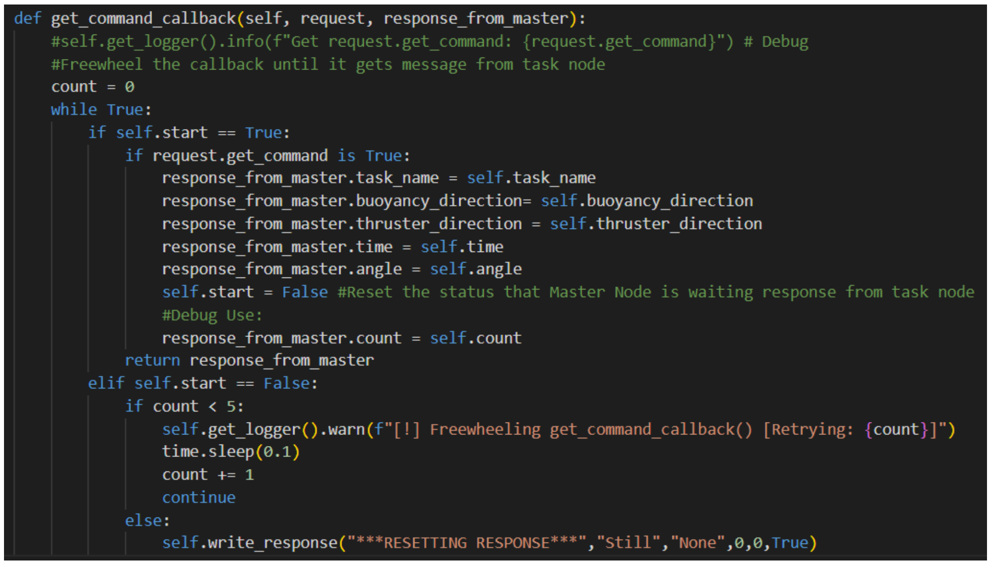

1. Service server receives request
2. Trigger get_task_callback() to respond:
bool is_finished
string task_name
string buoyancy_direction
string thruster_direction
Autonomous task1_node
Autonomous task2_node
Autonomous task3_node
3. Service client receives response
4. Trigger get_command_callback() to respond:
string task_name
string buoyancy_direction
string thruster_direction
Request:
bool apply_result
Request:
bool get_command
control_node
master_node
Response:
string task_name
string buoyancy_direction
string thruster_direction

Figure 65: Implementing Order between get_command_callback and get_task_callback method

To solve this problem, a "**start**" flag is defined in the Master Node (**Figure 66**). This flag acts as a control gate to determine whether the Master Node is allowed to respond to the Control Node.



```
def get_command_callback(self, request, response_from_master):
    #self.get_logger().info(f"Get request.get_command: {request.get_command}") # Debug
    #Freewheel the callback until it gets message from task node
    count = 0
    while True:
        if self.start == True:
            if request.get_command is True:
                response_from_master.task_name = self.task_name
                response_from_master.buoyancy_direction= self.buoyancy_direction
                response_from_master.thruster_direction = self.thruster_direction
                response_from_master.time = self.time
                response_from_master.angle = self.angle
                self.start = False #Reset the status that Master Node is waiting response from task node
                #Debug Use:
                response_from_master.count = self.count
            return response_from_master
        elif self.start == False:
            if count < 5:
                self.get_logger().warn(f"[!] Freewheeling get_command_callback() [Retrying: {count}]")
                time.sleep(0.1)
                count += 1
                continue
            else:
                self.write_response("***RESETTING RESPONSE***","Still","None",0,0,True)
```

Figure 66: Gate Control in get_command_callback Method

With this approach, we can enable the Master Node to respond to the Control Node only when it has received the response (control commands) from task nodes.

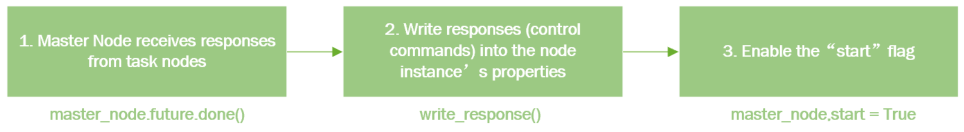

1. Master Node receives responses from task nodes
2. Write responses (control commands) into the node instance' s properties
3. Enable the "start" flag
master_node.future.done()
write_response()
master_node.start = True

Figure 67: Flowchart of Enabling get_command_callback method to Returrn Reponse

In the get_command_callback() method, it **freewheels** by a while-loop to wait until it detects **True** in the "start" flag. Once the "start" flag is **True**, the method reads the node instance's properties (control commands), responds (forward) them to the Control Node, and switches the "starts" back to False.

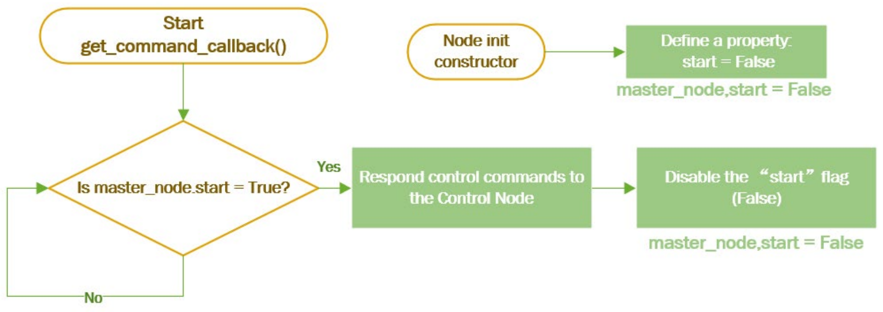

Start
get_command_callback()
Node init constructor
Define a property: start = False
master_node.start = False
Is master_node.start = True?
Yes
No
Respond control commands to the Control Node
Disable the "start" flag (False)
master_node.start = False

Figure 68: Flowchart of Gate Control in get_command_callback method

With this approach, the Master Node can respond with the latest control commands to the Control Node. The Control Node would not receive any outdated control commands.

<u>Possible Request Lose in ROS2</u>

In this ROS2-based control system for the AUV, it is found that the Master Node may fail to receive responses from the task nodes every ~1000-1500 implementation cycles. The reason behind it is still figuring out. However, this results in infinitely waiting in the Master Node, which is disabled to respond to the Control Node owing to a False Boolean status of the "start" flag. The Control Node, therefore, is also waiting for the Master Node to implement control commands and commence the next implementation cycle.

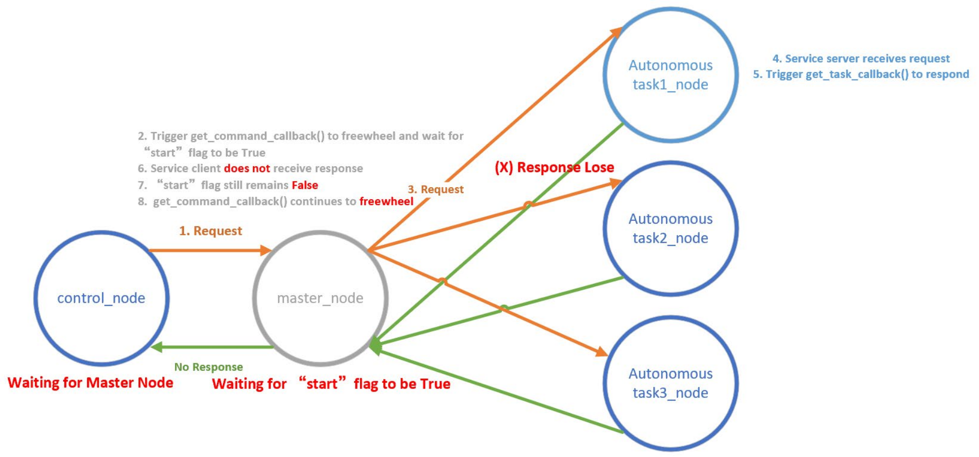

Autonomous task1_node
4. Service server receives request
5. Trigger get_task_callback() to respond
2. Trigger get_command_callback() to freewheel and wait for "start" flag to be True
6. Service client does not receive response
7. "start" flag still remains False
8. get_command_callback() continues to freewheel
3. Request
(X) Response Lose
Autonomous task2_node
1. Request
control_node
master_node
No Response
Waiting for Master Node
Waiting for "start" flag to be True
Autonomous task3_node

Figure 69: Possible Request Lose in ROS2

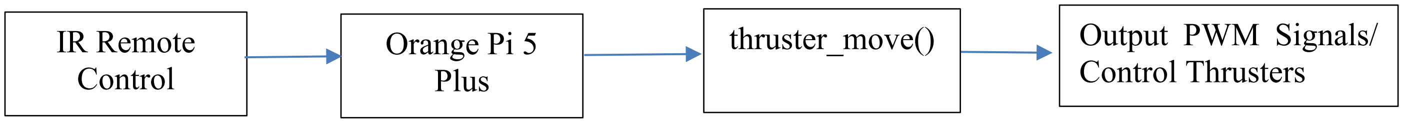
Therefore, we design an if-statement control logic to reset the response if the get_command_callback method has freewheeled for 5 times (0.1*5 = 0.5 seconds), as it is assumed that the task nodes should respond to the Master Node within 0.5 seconds. If the callback method has freewheeled 5 times, the system will automatically respond with **a dummy message with control commands thruster_direction: None and buoyancy_direction: Still** to the Control Node to allow the AUV to do nothing. Once the Control Node receives the control commands, it can directly start the next cycle and continue to request to the Master Node. The Master Node will request the specified task node again and try to retrieve responses from it. As a result, the communication inside the AUV control system will probably be recovered.

3.3 Non-ROS2 Testing and Commissioning Methods for AUV

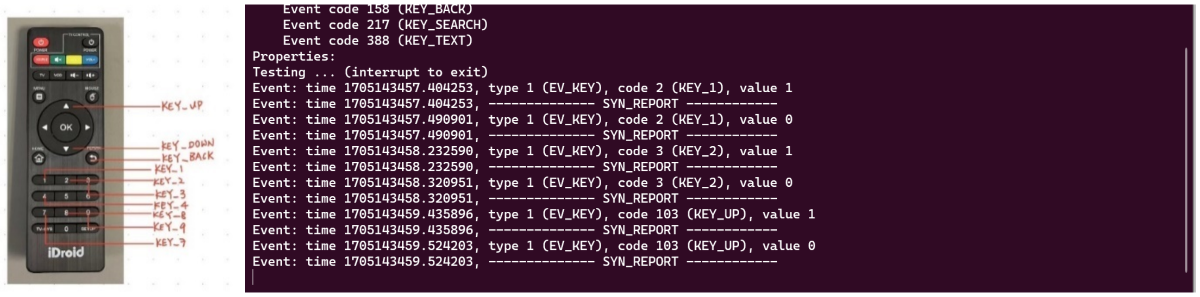
3.3.1 Thruster Control by Infra-red Remote Control

Before testing the AUV autonomous system, designers need to test and commission the AUV's electrical system intuitively, such as offsetting the AUV thrusters' operating duty cycles. Therefore, an infrared (IR) control approach is designed to test the AUV's thrusters.

To test the AUV thruster's speeds, we can design a program that uses an infrared remote control to control thrusters intuitively. The advantage of using infrared remote control is that IR signals can be received well underwater while Wi-Fi signals are almost absorbed by water. This idea will help test the AUV before implementing the autonomous programs.



To control the OP5P by IR signals, an IR remote control for the OP5P is used. According to the official documentation, the OP5P can read key values from the IR remote control by running Event Test (**evtest)** in the Linux terminal.

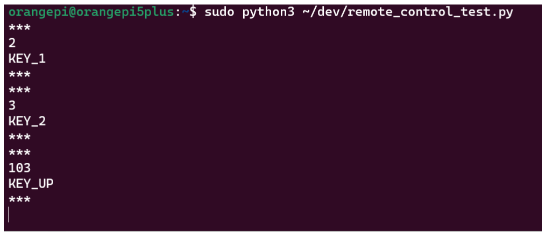


```
    Event code 158 (KEY_BACK)
    Event code 217 (KEY_SEARCH)
    Event code 388 (KEY_TEXT)
Properties:
Testing ... (interrupt to exit)
Event: time 1705143457.404253, type 1 (EV_KEY), code 2 (KEY_1), value 1
Event: time 1705143457.404253, -------------- SYN_REPORT ------------
Event: time 1705143457.490901, type 1 (EV_KEY), code 2 (KEY_1), value 0
Event: time 1705143457.490901, -------------- SYN_REPORT ------------
Event: time 1705143458.232590, type 1 (EV_KEY), code 3 (KEY_2), value 1
Event: time 1705143458.232590, -------------- SYN_REPORT ------------
Event: time 1705143458.320951, type 1 (EV_KEY), code 3 (KEY_2), value 0
Event: time 1705143458.320951, -------------- SYN_REPORT ------------
Event: time 1705143459.435896, type 1 (EV_KEY), code 103 (KEY_UP), value 1
Event: time 1705143459.435896, -------------- SYN_REPORT ------------
Event: time 1705143459.524203, type 1 (EV_KEY), code 103 (KEY_UP), value 0
Event: time 1705143459.524203, -------------- SYN_REPORT ------------
```

Figure 70: IR remote control and outputs of Event Test in Linux

Therefore, we can design a Python program to read these outputs. For example, we can use the Python library **evdev** [27] to read which of the button keys users have pressed:

If the "1", "2", and "up" keys are pressed on the IR remote control, the Linux terminal outputs the corresponding key values:



```
orangepi@orangepi5plus:~$ sudo python3 ~/dev/remote_control_test.py
***
2
KEY_1
***
***
3
KEY_2
***
***
103
KEY_UP
***
```

Figure 71: Read IR signal by Python

With this approach, a Python program is developed to control the AUV thrusters using the IR remote control. This Python program also utilizes the **evdev** library to read the key values the OP5P received from the IR remote control.

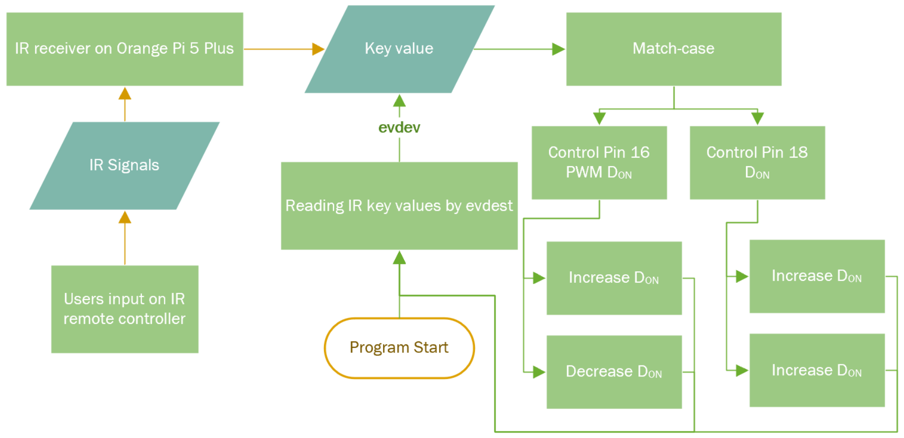


Figure 72: Flowchart of IR Remote Control Method

Take Figure 72 as an example. Firstly, when the user presses a button on the IR remote control, the controller sends a signal to the OP5P. Once the OP5P receives the IR signals, the Python program captures the key value associated with the signal. The program then employs a case-match statement to map different key values to specific actions. These actions may include selecting PWM channels on the OP5P and increasing or decreasing the PWM duty cycles on these channels, thereby controlling the speed of the AUV's thrusters.

3.3.2 6-bit Telecommunication

Apart from the IR remote control method, it is also necessary to study possible solutions to remotely control the AUV, such as controlling the thruster system, the buoyancy system, and ROS2 nodes. Therefore, a simple simplex 6-bit telecommunication method is developed in this project.

The 6-bit telecommunication is inspired by an online submarine project [28], which used a submarine toy's remote control (transmitter) and the toy's receiver to implement the remote control of a LEGO submarine. In this project, the receiver operates at 27MHz and has 6 output signals, which originally are referred to as "forward", "backward", "left", "right", "surface", and "dive". These output signals from the receiver have voltages of around 3.3VDC. Therefore, the receiver's output signals can be read by Pico/ESP32/Arduino's GPIO pins directly. After receiving these signals, they can be converted to an integer by bit shifting.

Receiving End

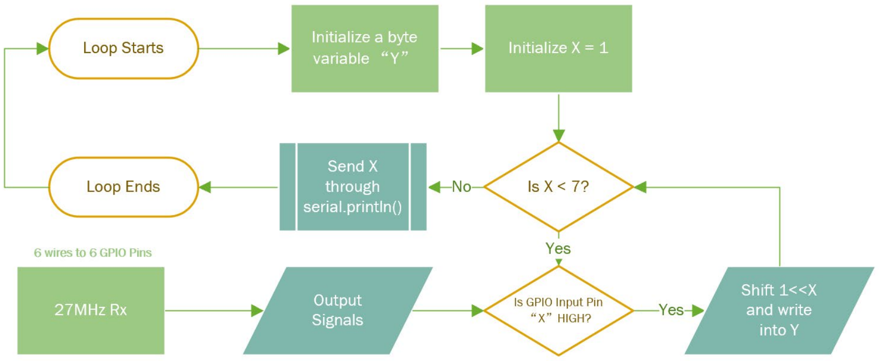

Loop Starts
Initialize a byte variable "Y"
Initialize X = 1
Is X < 7?
No
Send X through serial.println()
Loop Ends
Yes
6 wires to 6 GPIO Pins
27MHz Rx
Output Signals
Is GPIO Input Pin "X" HIGH?
Yes
Shift 1<<X and write into Y

Figure 73: Flowchart of Converting Recceived Signals to an integer

Connecting the receiver with a Raspberry Pi Pico as an example, when the Pico starts, it will declare a byte "Y" and initialize GPIO Pin 1-6 as a low-level input source. When the loop starts, the Pico reads the first pin first. If the 27MHz receiver outputs HIGH to the first pin, then the program executes 1<<1 to "Y", resulting in "00000001". After that, the program reads the second pin. If the 27MHz receiver outputs HIGH to the second pin, then the program executes 1<<2 to "Y", resulting in "00000011". The process continues until Pico finishes reading the last pin. If the resultant "Y" is "00100111", then the Pico sends a value of 39 by **Serial.println(Y)**, which the OP5P then receives through the USB UART Serial. For example, the OP5P can receive 37 by directly reading **/dev/ttyACMx** or **/dev/ttyUSBx** in Linux. After receiving the value from the receiver, the OP5P can implement additional tasks based on it or shift back the byte to look up which of the Pico's Pins is high level, for example, implementing a loop for ((00100111 >> i) & 1 where 1<=i<=6). On the other hand, the OP5P can implement corresponding tasks if it receives the same value (i.e. 37) more than 5 times, as it can determine whether the received signal is valid.

Transmitting End

On the transmitting end, we can directly press buttons of the 27MHz remote control to telecommunicate with the AUV's system. However, this approach is inconvenient and ineffective in pressing the buttons continuously. Therefore, we can connect the polarities of the remote control's buttons to 6 controllable relays and use an Arduino to trigger the "press" action. For example:

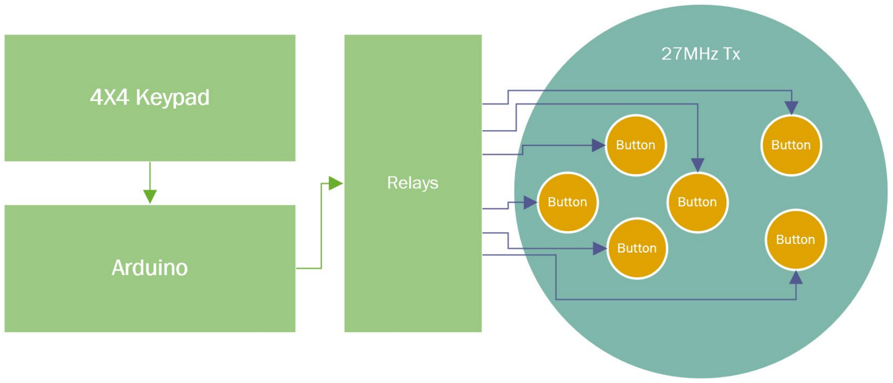


Figure 74: Circuit Connection in Transmitting End of 27MHz Remote Control

An Arduino with a keypad is used for users to press buttons (i.e. 1-9). The Arduino is responsible for processing signals from the keypad's pressed buttons and then turning on corresponding relays to short-circuit the buttons on the 27MHz remote control to transmit signals. With this approach, the telecommunication between the users and the AUV is more stable and convenient.

4. Results

4.1 Demonstration of ROS2 Control System Framework by Simulating Nodes

4.1.1 Demonstration of ROS2 Services between AUV Control System Nodes and Single Task Node

In this section, a basic ROS2-based control system for AUV demonstration is presented. This demonstration verifies whether the Control Node and the Master Node (AUV control system) can function to interact with a simulated task node so that the Control Node can correctly receive control commands from the simulated task node through the Master Node and implement them. The demonstration also presents the frequency control in the Control Node and the freewheeling control in the Master Node. The following diagram illustrates the whole steps of one implementation cycle for the AUV system in this demonstration.

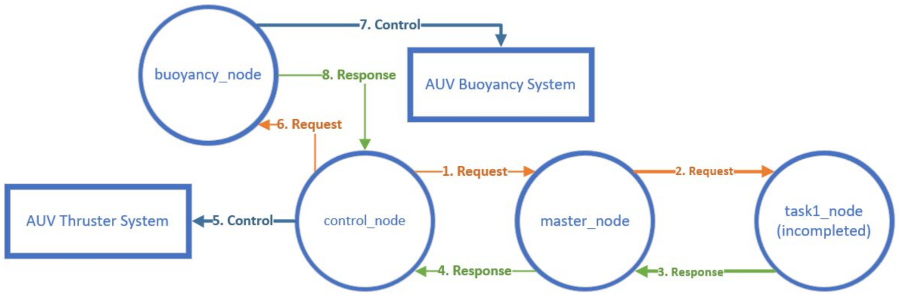
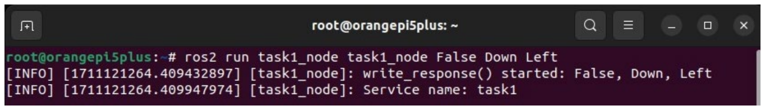


Figure 75: Demonstration of ROS2 Services between AUV Control System Nodes and Single Task Node

Firstly, a simulated task node named **task1_node** was started by running:

ros2 run task1_node task1_node False Down Left

(ros2 run package_name node_name arguments)



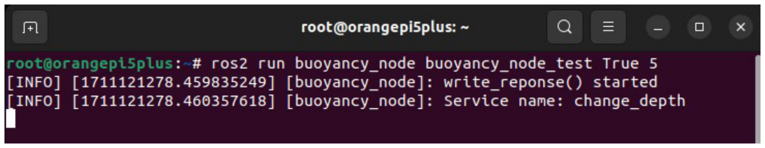
```
root@orangepi5plus: ~
root@orangepi5plus:~# ros2 run task1_node task1_node False Down Left
[INFO] [1711121264.409432897] [task1_node]: write_response() started: False, Down, Left
[INFO] [1711121264.409947974] [task1_node]: Service name: task1
```

Figure 76: Launching a Simulated Task 1 Node

Three arguments were declared in the task1_node. The “False” represents the incomplete status of the current task so that the Master Node would request to this node continuously. The “Down” represents the control command for the buoyancy system so that the Control Node would request the Buoyancy Node with a decreased depth value. The “Left” represents the control command for the thruster system so that the AUV’s right thruster would propel to turn the AUV to the left. After the task1_node was started, its service name “task1” was output.

Secondly, a simulated buoyancy node named **buoyancy_node_test** was started by running:

ros2 run buoyancy_node buoyancy_node_test True 5



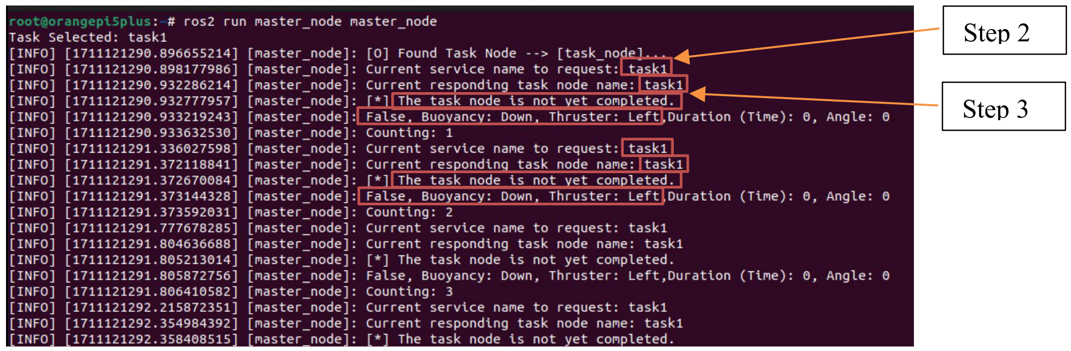
```
root@orangepi5plus:~# ros2 run buoyancy_node buoyancy_node_test True 5
[INFO] [1711121278.459835249] [buoyancy_node]: write_reponse() started
[INFO] [1711121278.460357618] [buoyancy_node]: Service name: change_depth
```

Figure 77: Launching a Simulated Buoyancy Node

Two arguments were declared in the buoyancy_node_test. The "True" represents a simulated result of the successful status of the AUV buoyancy system that has modified the depth value requested by the Control Node. The value of "5" represents a simulated result of the current AUV depth value read from a pressure sensor.

Then, a Master Node (master_node) was started by running:

ros2 run master_node master_node



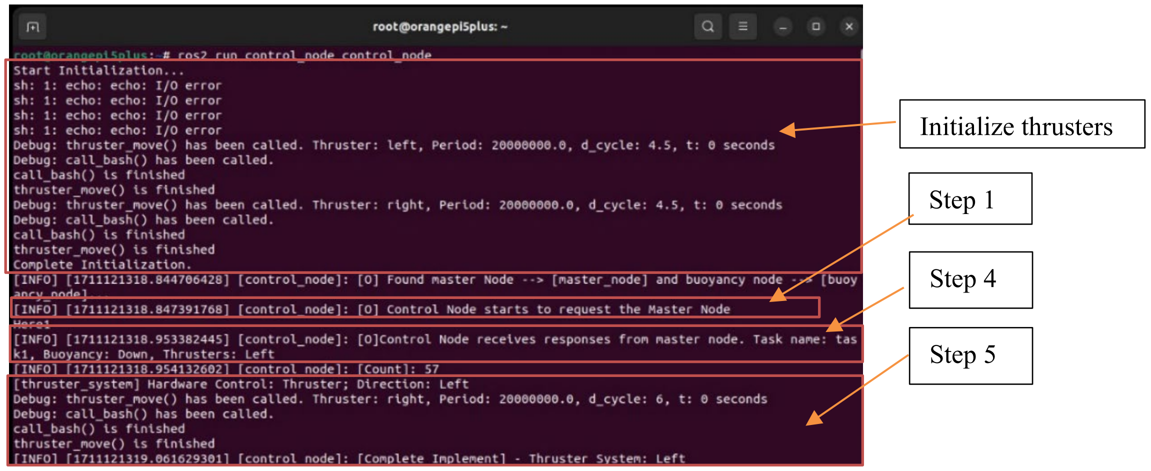
```
root@orangepi5plus:~# ros2 run master_node master_node
Task Selected: task1
[INFO] [1711121290.896655214] [master_node]: [O] Found Task Node --> [task_node]...
[INFO] [1711121290.898177986] [master_node]: Current service name to request: task1
[INFO] [1711121290.932286214] [master_node]: Current responding task node name: task1
[INFO] [1711121290.932777957] [master_node]: [*] The task node is not yet completed.
[INFO] [1711121290.933219243] [master_node]: False, Buoyancy: Down, Thruster: Left,Duration (Time): 0, Angle: 0
[INFO] [1711121290.933632530] [master_node]: Counting: 1
[INFO] [1711121291.336027598] [master_node]: Current service name to request: task1
[INFO] [1711121291.372118841] [master_node]: Current responding task node name: task1
[INFO] [1711121291.372670084] [master_node]: [*] The task node is not yet completed.
[INFO] [1711121291.373144328] [master_node]: False, Buoyancy: Down, Thruster: Left,Duration (Time): 0, Angle: 0
[INFO] [1711121291.373592031] [master_node]: Counting: 2
[INFO] [1711121291.777678285] [master_node]: Current service name to request: task1
[INFO] [1711121291.804636688] [master_node]: Current responding task node name: task1
[INFO] [1711121291.805213014] [master_node]: [*] The task node is not yet completed.
[INFO] [1711121291.805872756] [master_node]: False, Buoyancy: Down, Thruster: Left,Duration (Time): 0, Angle: 0
[INFO] [1711121291.806410582] [master_node]: Counting: 3
[INFO] [1711121292.215872351] [master_node]: Current service name to request: task1
[INFO] [1711121292.354984392] [master_node]: Current responding task node name: task1
[INFO] [1711121292.358408515] [master_node]: [*] The task node is not yet completed.
```

Figure 78: Output Result in Master Node at startup

From the above figure, it is observed that the master_node started to request to the first task node by default (service name: "task1") (**Step 2**), and the master_node successfully retrieved control commands from the task1_node **(Step 3**). The master_node acknowledged that the task1_node was incomplete, so it continues to request the task1_node.

Lastly, a Control Node (control_node) was started by running:

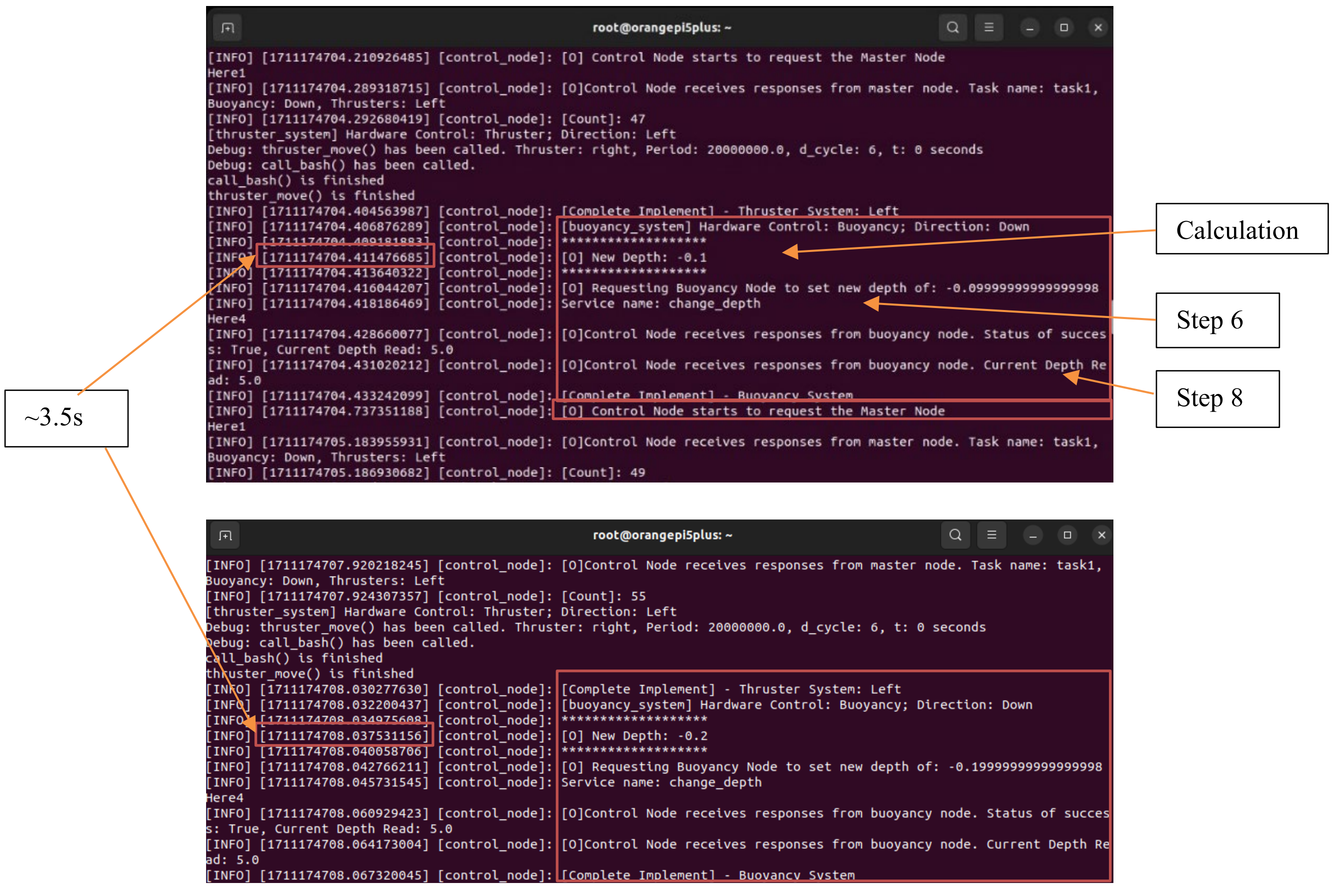
ros2 run control_node control_node



```
root@orangepi5plus:~# ros2 run control_node control_node
Start Initialization...
sh: 1: echo: echo: I/O error
sh: 1: echo: echo: I/O error
sh: 1: echo: echo: I/O error
sh: 1: echo: echo: I/O error
Debug: thruster_move() has been called. Thruster: left, Period: 20000000.0, d_cycle: 4.5, t: 0 seconds
Debug: call_bash() has been called.
call_bash() is finished
thruster_move() is finished
Debug: thruster_move() has been called. Thruster: right, Period: 20000000.0, d_cycle: 4.5, t: 0 seconds
Debug: call_bash() has been called.
call_bash() is finished
thruster_move() is finished
Complete Initialization.
[INFO] [1711121318.844706428] [control_node]: [O] Found master Node --> [master_node] and buoyancy node --> [buoy
ancy_node]...
[INFO] [1711121318.847391768] [control_node]: [O] Control Node starts to request the Master Node
Here1
[INFO] [1711121318.953382445] [control_node]: [O]Control Node receives responses from master node. Task name: tas
k1, Buoyancy: Down, Thrusters: Left
[INFO] [1711121318.954132602] [control_node]: [Count]: 57
[thruster_system] Hardware Control: Thruster; Direction: Left
Debug: thruster_move() has been called. Thruster: right, Period: 20000000.0, d_cycle: 6, t: 0 seconds
Debug: call_bash() has been called.
call_bash() is finished
thruster_move() is finished
[INFO] [1711121319.061629301] [control_node]: [Complete Implement] - Thruster System: Left
```

Figure 79: Output Result in Control Node (thruster subsystem)

In **Figure 79**, the control_node initialized the OP5P's PWM channels first. Then, it called the **thruster_move** function to generate PWM signals with a period of 20000000ns (50Hz f_s) and 4.5% D_{ON} to initialize the AUV thrusters. After the initialization, the control_node started to request to the master_node (**Step 1**) and successfully retrieved control commands from the Master Node (**Step 4**). As the control_node received a command to move left, it called the **thruster_move** to turn on the AUV **right** thruster to propel (**Step 5**).



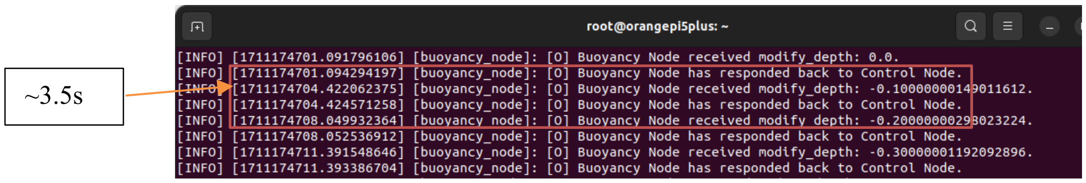
```
[INFO] [1711174704.210926485] [control_node]: [O] Control Node starts to request the Master Node
Here1
[INFO] [1711174704.289318715] [control_node]: [O]Control Node receives responses from master node. Task name: task1,
Buoyancy: Down, Thrusters: Left
[INFO] [1711174704.292680419] [control_node]: [Count]: 47
[thruster_system] Hardware Control: Thruster; Direction: Left
Debug: thruster_move() has been called. Thruster: right, Period: 20000000.0, d_cycle: 6, t: 0 seconds
Debug: call_bash() has been called.
call_bash() is finished
thruster_move() is finished
[INFO] [1711174704.404563987] [control_node]: [Complete Implement] - Thruster System: Left
[INFO] [1711174704.406876289] [control_node]: [buoyancy_system] Hardware Control: Buoyancy; Direction: Down
[INFO] [1711174704.409181883] [control_node]: *******************
[INFO] [1711174704.411476685] [control_node]: [O] New Depth: -0.1
[INFO] [1711174704.413640322] [control_node]: *******************
[INFO] [1711174704.416044207] [control_node]: [O] Requesting Buoyancy Node to set new depth of: -0.09999999999999998
[INFO] [1711174704.418186469] [control_node]: Service name: change_depth
Here4
[INFO] [1711174704.428660077] [control_node]: [O]Control Node receives responses from buoyancy node. Status of succes
s: True, Current Depth Read: 5.0
[INFO] [1711174704.431020212] [control_node]: [O]Control Node receives responses from buoyancy node. Current Depth Re
ad: 5.0
[INFO] [1711174704.433242099] [control_node]: [Complete Implement] - Buoyancy System
[INFO] [1711174704.737351188] [control_node]: [O] Control Node starts to request the Master Node
Here1
[INFO] [1711174705.183955931] [control_node]: [O]Control Node receives responses from master node. Task name: task1,
Buoyancy: Down, Thrusters: Left
[INFO] [1711174705.186930682] [control_node]: [Count]: 49
```

```
[INFO] [1711174707.920218245] [control_node]: [O]Control Node receives responses from master node. Task name: task1,
Buoyancy: Down, Thrusters: Left
[INFO] [1711174707.924307357] [control_node]: [Count]: 55
[thruster_system] Hardware Control: Thruster; Direction: Left
Debug: thruster_move() has been called. Thruster: right, Period: 20000000.0, d_cycle: 6, t: 0 seconds
Debug: call_bash() has been called.
call_bash() is finished
thruster_move() is finished
[INFO] [1711174708.030277630] [control_node]: [Complete Implement] - Thruster System: Left
[INFO] [1711174708.032200437] [control_node]: [buoyancy_system] Hardware Control: Buoyancy; Direction: Down
[INFO] [1711174708.034975608] [control_node]: *******************
[INFO] [1711174708.037531156] [control_node]: [O] New Depth: -0.2
[INFO] [1711174708.040058706] [control_node]: *******************
[INFO] [1711174708.042766211] [control_node]: [O] Requesting Buoyancy Node to set new depth of: -0.19999999999999998
[INFO] [1711174708.045731545] [control_node]: Service name: change_depth
Here4
[INFO] [1711174708.060929423] [control_node]: [O]Control Node receives responses from buoyancy node. Status of succes
s: True, Current Depth Read: 5.0
[INFO] [1711174708.064173004] [control_node]: [O]Control Node receives responses from buoyancy node. Current Depth Re
ad: 5.0
[INFO] [1711174708.067320045] [control_node]: [Complete Implement] - Buoyancy System
```

Figure 80: Output Result in Control Node after Launching (buoyancy subsystem)

In the aspect of the frequency control (**Figure 80**), the control_node only requests the buoyancy_node to change behaviours of the buoyancy system every ~3-4 seconds. Taking the above figures as an example, when the control_node determined to request the buoyancy_node after completing the control for the thruster system, it first calculated the latest depth for the AUV. As the task1_node required the AUV to move down, the control_node calculated a new depth value by decreasing the AUV depth by 0.1 meters (i.e. 0-0.1=-0.1 meters). Then, it requested the buoyancy node with the service name "change_depth" and the new-calculated depth value (**Step 6**). After that, the control_node received the response from the buoyancy_node, telling the control_node that the implementation status was successful and the current depth read from a pressure sensor is 5 meters (simulated response)(**Step 7-8**). Ultimately, the control_node started the next implementation cycle and continued to request the master_node to retrieve control commands.

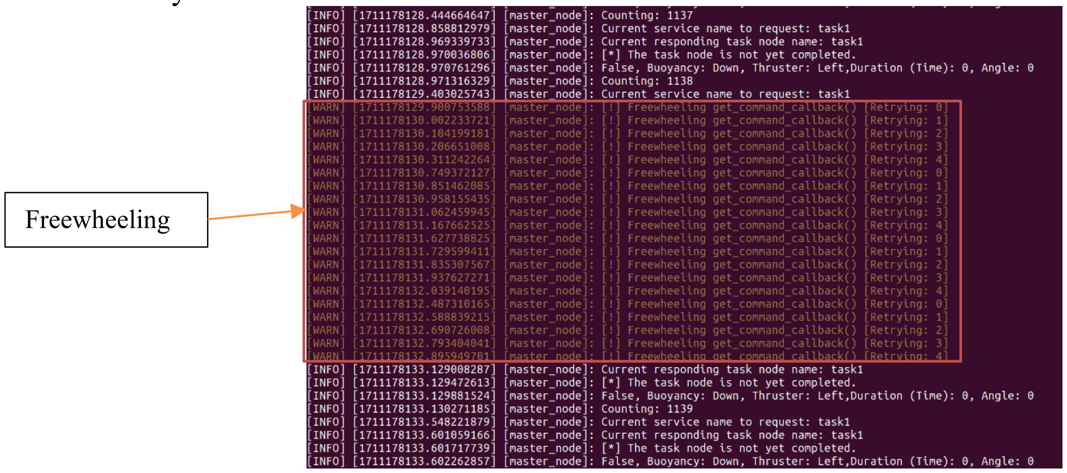
Since the control_node had already requested the buoyancy_node for once, it would temporarily not proceed with any requests for the buoyancy system for followed few control cycles. After the few control cycles were implemented, the control_node determined to calculate a new AUV depth (-0.1-0.1=-0.2 meter) again according to the control command (Down) and requested the buoyancy_node again. Eventually, the buoyanc_node received requests from the control_node every ~3.5 seconds. The control frequency can be adjusted to fit the physical change of the AUV buoyancy system by modifying the **buoyancy_count** in the control_node.



```
[INFO] [1711174701.091796106] [buoyancy_node]: [O] Buoyancy Node received modify_depth: 0.0.
[INFO] [1711174701.094294197] [buoyancy_node]: [O] Buoyancy Node has responded back to Control Node.
[INFO] [1711174704.422062375] [buoyancy_node]: [O] Buoyancy Node received modify_depth: -0.10000000149011612.
[INFO] [1711174704.424571258] [buoyancy_node]: [O] Buoyancy Node has responded back to Control Node.
[INFO] [1711174708.049932364] [buoyancy_node]: [O] Buoyancy Node received modify_depth: -0.20000000298023224.
[INFO] [1711174708.052536912] [buoyancy_node]: [O] Buoyancy Node has responded back to Control Node.
[INFO] [1711174711.391548646] [buoyancy_node]: [O] Buoyancy Node received modify_depth: -0.30000001192092896.
[INFO] [1711174711.393386704] [buoyancy_node]: [O] Buoyancy Node has responded back to Control Node.
```

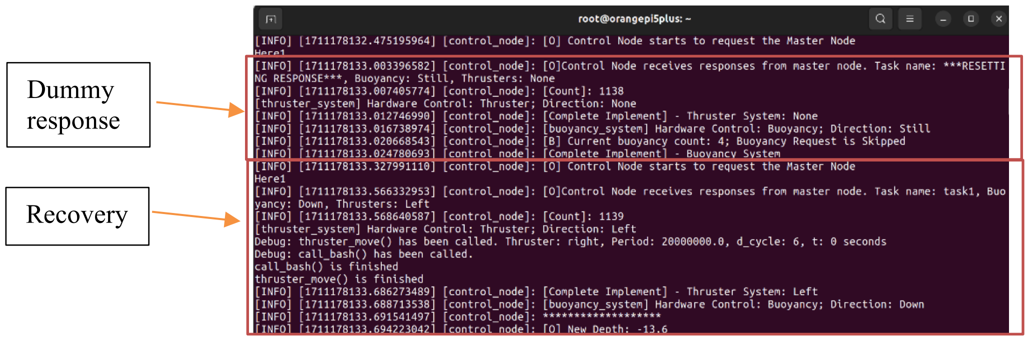
Figure 81: Output Result in Buoyancy Node after Launching

In the aspect of the freewheeling control in the Master Node (**Figure 82**), the Master Node successfully recovered the communication by automatically responding with a dummy control command (**thruster_direction: None, buoyancy_direction: Still**) to the Control Node when the Master Node lost responses from the task node. Taking the below figure as an example, when the master_node encountered freewheel for more than 0.5 seconds, it automatically responded with a dummy control command to the control_node to start a new control cycle. With this approach, the master_node would not be stuck and block the overall system's operations, and communications within the nodes would be recovered automatically. Therefore, the AUV control system is stabilized and more robust.



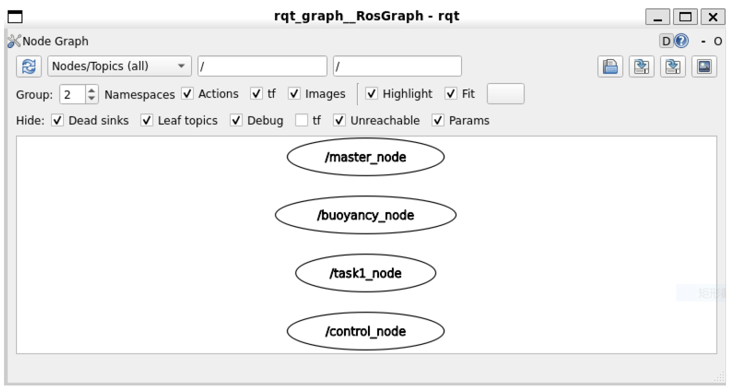
```
[INFO] [1711178128.444664647] [master_node]: Counting: 1137
[INFO] [1711178128.858812979] [master_node]: Current service name to request: task1
[INFO] [1711178128.969339733] [master_node]: Current responding task node name: task1
[INFO] [1711178128.970036806] [master_node]: [*] The task node is not yet completed.
[INFO] [1711178128.970761296] [master_node]: False, Buoyancy: Down, Thruster: Left,Duration (Time): 0, Angle: 0
[INFO] [1711178128.971316329] [master_node]: Counting: 1138
[INFO] [1711178129.403025743] [master_node]: Current service name to request: task1
[WARN] [1711178129.900753588] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 0]
[WARN] [1711178130.002233721] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 1]
[WARN] [1711178130.104199181] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 2]
[WARN] [1711178130.206651008] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 3]
[WARN] [1711178130.311242264] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 4]
[WARN] [1711178130.749372127] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 0]
[WARN] [1711178130.851462085] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 1]
[WARN] [1711178130.958155435] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 2]
[WARN] [1711178131.062459945] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 3]
[WARN] [1711178131.167662525] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 4]
[WARN] [1711178131.627738825] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 0]
[WARN] [1711178131.729599411] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 1]
[WARN] [1711178131.835307567] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 2]
[WARN] [1711178131.937627271] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 3]
[WARN] [1711178132.039140195] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 4]
[WARN] [1711178132.487310165] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 0]
[WARN] [1711178132.588839215] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 1]
[WARN] [1711178132.690726008] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 2]
[WARN] [1711178132.793404041] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 3]
[WARN] [1711178132.895949701] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 4]
[INFO] [1711178133.129008287] [master_node]: Current responding task node name: task1
[INFO] [1711178133.129472613] [master_node]: [*] The task node is not yet completed.
[INFO] [1711178133.129881524] [master_node]: False, Buoyancy: Down, Thruster: Left,Duration (Time): 0, Angle: 0
[INFO] [1711178133.130271185] [master_node]: Counting: 1139
[INFO] [1711178133.548221879] [master_node]: Current service name to request: task1
[INFO] [1711178133.601059166] [master_node]: Current responding task node name: task1
[INFO] [1711178133.601717739] [master_node]: [*] The task node is not yet completed.
[INFO] [1711178133.602262857] [master_node]: False, Buoyancy: Down, Thruster: Left,Duration (Time): 0, Angle: 0
```

Figure 82: Freewheeling in Master Node



```
[INFO] [1711178132.475195964] [control_node]: [O] Control Node starts to request the Master Node
Here1
[INFO] [1711178133.003396582] [control_node]: [O]Control Node receives responses from master node. Task name: ***RESETTI
NG RESPONSE***, Buoyancy: Still, Thrusters: None
[INFO] [1711178133.007405774] [control_node]: [Count]: 1138
[thruster_system] Hardware Control: Thruster; Direction: None
[INFO] [1711178133.012746990] [control_node]: [Complete Implement] - Thruster System: None
[INFO] [1711178133.016738974] [control_node]: [buoyancy_system] Hardware Control: Buoyancy; Direction: Still
[INFO] [1711178133.020668543] [control_node]: [B] Current buoyancy count: 4; Buoyancy Request is Skipped
[INFO] [1711178133.024780693] [control_node]: [Complete Implement] - Buoyancy System
[INFO] [1711178133.327991110] [control_node]: [O] Control Node starts to request the Master Node
Here1
[INFO] [1711178133.566332953] [control_node]: [O]Control Node receives responses from master node. Task name: task1, Buo
yancy: Down, Thrusters: Left
[INFO] [1711178133.568640587] [control_node]: [Count]: 1139
[thruster_system] Hardware Control: Thruster; Direction: Left
Debug: thruster_move() has been called. Thruster: right, Period: 20000000.0, d_cycle: 6, t: 0 seconds
Debug: call_bash() has been called.
call_bash() is finished
thruster_move() is finished
[INFO] [1711178133.686273489] [control_node]: [Complete Implement] - Thruster System: Left
[INFO] [1711178133.688713538] [control_node]: [buoyancy_system] Hardware Control: Buoyancy; Direction: Down
[INFO] [1711178133.691541497] [control_node]: ******************
[INFO] [1711178133.694223042] [control_node]: [O] New Depth: -13.6
```

Figure 83: Communication Recovery between Control Node and Master Node



```
Nodes/Topics (all)   /   /
Group: 2   Namespaces [✓]   Actions [✓]   tf [✓]   Images [✓]   Highlight [✓]   Fit [✓]
Hide: Dead sinks [✓]   Leaf topics [✓]   Debug [✓]   tf [ ]   Unreachable [✓]   Params [✓]
```

Figure 84: rqt_graph showing all running nodes

4.1.2 Demonstration of Flow Control in Master Node

In this section, flow control in the Master Node is demonstrated. The flow control allows the Master Node to be directed to the correct destination for requesting control commands. For example, when the first task node (task1_node) is completed, the Master Node can determine to switch to request the second incomplete task node (task2_node) and forward control commands to the Control Node. The following diagram illustrates the whole steps of one implementation cycle for the AUV system in this demonstration.

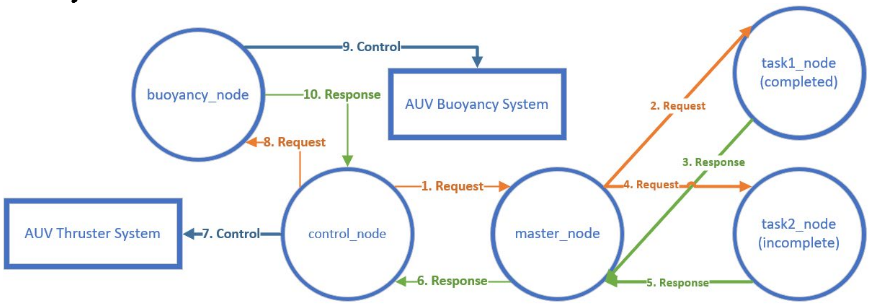
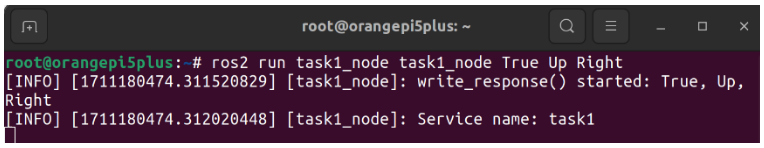


Figure 85: Demonstration of Flow Control in Master Node

Firstly, a simulated task node task1_node was started by running:

ros2 run task1_node task1_node Ture Up Right



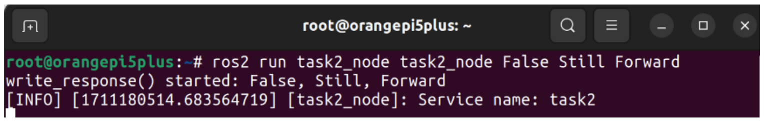
```
root@orangepi5plus:~# ros2 run task1_node task1_node True Up Right
[INFO] [1711180474.311520829] [task1_node]: write_response() started: True, Up,
Right
[INFO] [1711180474.312020448] [task1_node]: Service name: task1
```

Figure 86: Launching a Simulated Task 1 Node

This command launched a task1_node, which had already completed the autonomous task (True). However, as the autonomous program was still implementing behind the node, it would continue to respond with meaningless control commands to the Master Node (buoyancy_direction: Up; thruster_direction: Right).

Secondly, a simulated task2_node was started by running:

ros2 run task2_node task2_node False Still Forward

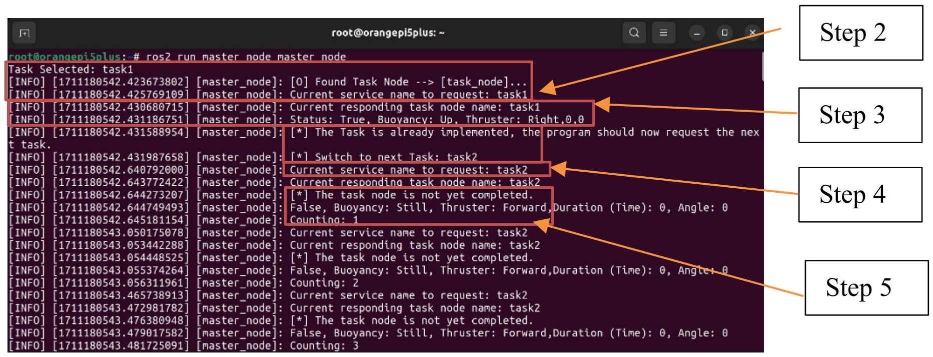


```
root@orangepi5plus:~# ros2 run task2_node task2_node False Still Forward
write_response() started: False, Still, Forward
[INFO] [1711180514.683564719] [task2_node]: Service name: task2
```

Figure 87: Launching a Simulated Task 2 Node

This command launched a task2_node which had not completed the autonomous task (False). The task2_mode would also respond with control commands to the Master Node (buoyancy_direction: Still; thruster_direction: Forward)

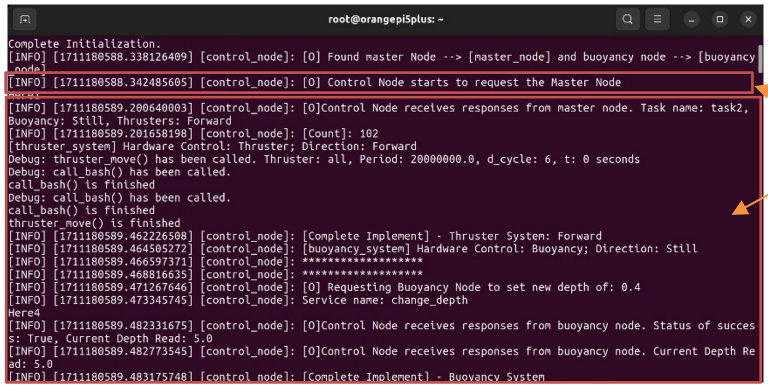
Thirdly, a Master Node (master_node) was launched.



```
root@orangepi5plus:~# ros2 run master_node master_node
Task Selected: task1
[INFO] [1711180542.423673802] [master_node]: [0] Found Task Node --> [task_node]...
[INFO] [1711180542.425769109] [master_node]: Current service name to request: task1
[INFO] [1711180542.430680715] [master_node]: Current responding task node name: task1
[INFO] [1711180542.431186751] [master_node]: Status: True, Buoyancy: Up, Thruster: Right,0,0
[INFO] [1711180542.431588954] [master_node]: [*] The Task is already implemented, the program should now request the nex
t task.
[INFO] [1711180542.431987658] [master_node]: [*] Switch to next Task: task2
[INFO] [1711180542.640792000] [master_node]: Current service name to request: task2
[INFO] [1711180542.643772422] [master_node]: Current responding task node name: task2
[INFO] [1711180542.644273207] [master_node]: [*] The task node is not yet completed.
[INFO] [1711180542.644749493] [master_node]: False, Buoyancy: Still, Thruster: Forward,Duration (Time): 0, Angle: 0
[INFO] [1711180542.645181154] [master_node]: Counting: 1
[INFO] [1711180543.050175078] [master_node]: Current service name to request: task2
[INFO] [1711180543.053442288] [master_node]: Current responding task node name: task2
[INFO] [1711180543.054448525] [master_node]: [*] The task node is not yet completed.
[INFO] [1711180543.055374264] [master_node]: False, Buoyancy: Still, Thruster: Forward,Duration (Time): 0, Angle: 0
[INFO] [1711180543.056311961] [master_node]: Counting: 2
[INFO] [1711180543.465738913] [master_node]: Current service name to request: task2
[INFO] [1711180543.472981782] [master_node]: Current responding task node name: task2
[INFO] [1711180543.476380948] [master_node]: [*] The task node is not yet completed.
[INFO] [1711180543.479017582] [master_node]: False, Buoyancy: Still, Thruster: Forward,Duration (Time): 0, Angle: 0
[INFO] [1711180543.481725091] [master_node]: Counting: 3
```

Figure 88: Output Result in Master Node at Startup

From the **Figure 88**, the master_node first requested the task1_node (**Step 2**). Then, the Master Node received a response from the task1_node (**Step 3**). The Master Node could acknowledge that the task1_node was completed and determined to request the next task node (task2_node) (**Step 4**). Hence, the master_node received responses from the task2_node (**Step 5**).



```
Complete Initialization.
[INFO] [1711180588.338126409] [control_node]: [0] Found master Node --> [master_node] and buoyancy node --> [buoyancy
_node]...
[INFO] [1711180588.342485605] [control_node]: [0] Control Node starts to request the Master Node
[INFO] [1711180589.200640003] [control_node]: [0]Control Node receives responses from master node. Task name: task2,
Buoyancy: Still, Thrusters: Forward
[INFO] [1711180589.201658198] [control_node]: [Count]: 102
[thruster_system] Hardware Control: Thruster; Direction: Forward
Debug: thruster_move() has been called. Thruster: all, Period: 20000000.0, d_cycle: 6, t: 0 seconds
Debug: call_bash() has been called.
call_bash() is finished
Debug: call_bash() has been called.
call_bash() is finished
thruster_move() is finished
[INFO] [1711180589.462226508] [control_node]: [Complete Implement] - Thruster System: Forward
[INFO] [1711180589.464505272] [control_node]: [buoyancy_system] Hardware Control: Buoyancy; Direction: Still
[INFO] [1711180589.466597371] [control_node]: *******************
[INFO] [1711180589.468816635] [control_node]: *******************
[INFO] [1711180589.471267646] [control_node]: [0] Requesting Buoyancy Node to set new depth of: 0.4
[INFO] [1711180589.473345745] [control_node]: Service name: change_depth
Here4
[INFO] [1711180589.482331675] [control_node]: [0]Control Node receives responses from buoyancy node. Status of succes
s: True, Current Depth Read: 5.0
[INFO] [1711180589.482773545] [control_node]: [0]Control Node receives responses from buoyancy node. Current Depth Re
ad: 5.0
[INFO] [1711180589.483175748] [control_node]: [Complete Implement] - Buoyancy System
```

Figure 89: Output Result in Control Node at Startup

In the aspect of the control_node, it was receiving the control commands from the task2_node (**Step 6**) after requesting (**Step 1**), indicating that the master_node was able to establish the flow control and was forwarding correct control commands to the control_node.

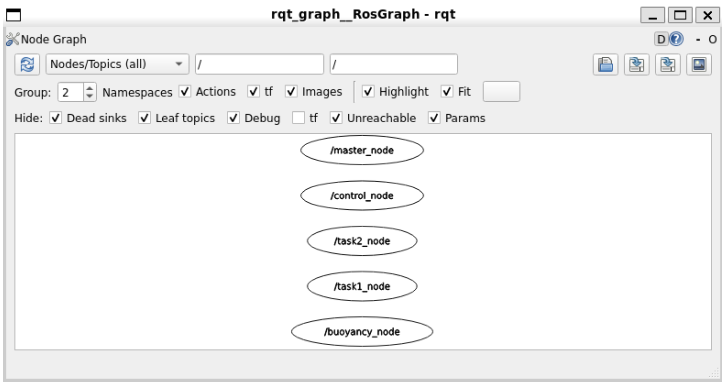


Figure 90: rqt_graph showing all running nodes

4.1.3 Demonstration of Master Node with ROS2 Parameters

As previously discussed in the methodology section, the ROS2 Parameters API offers developers a convenient mechanism for modifying and updating node configurations. This section showcases the application of this API in switching to request task nodes within the Master Node. The following diagram illustrates the whole steps of one implementation cycle for the AUV system in this demonstration.

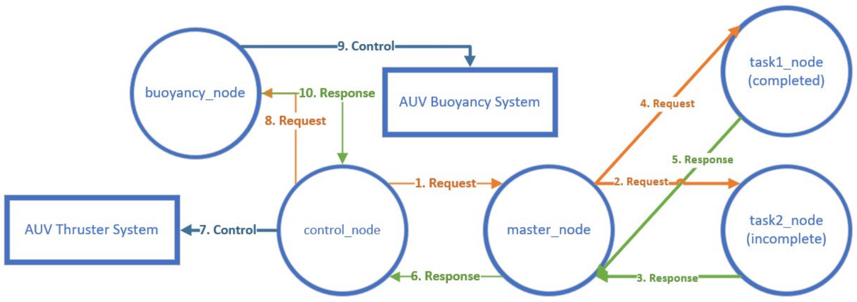
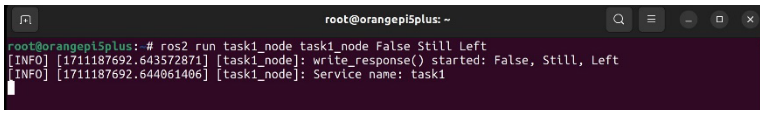


Figure 91: Demonstration of Master Node with ROS2 Parameters

Firstly, a new simulated task1_node was started by running:

ros2 run task1_node task1_node False Still Left

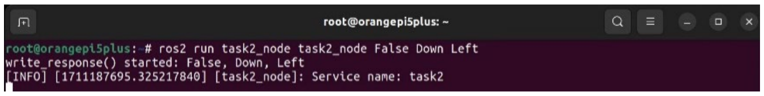


```
root@orangepi5plus:~# ros2 run task1_node task1_node False Still Left
[INFO] [1711187692.643572871] [task1_node]: write_response() started: False, Still, Left
[INFO] [1711187692.644061406] [task1_node]: Service name: task1
```

Figure 92: Launching a Simulated Task 1 Node

Then, a new simulated task2_node was started by running:

ros2 run task2_node task2_node False Down Left

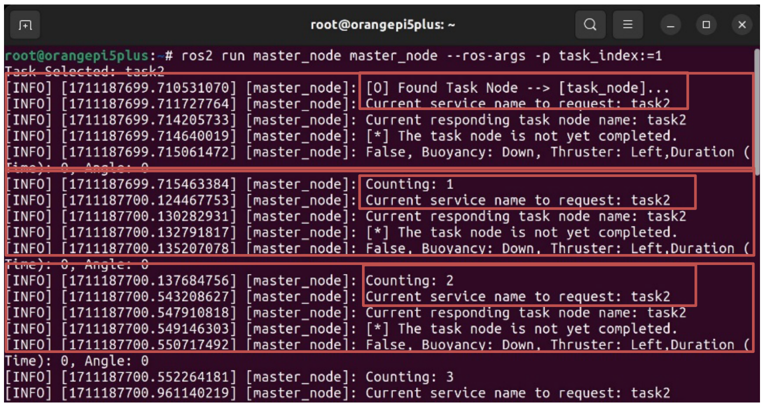


```
root@orangepi5plus:~# ros2 run task2_node task2_node False Down Left
write_response() started: False, Down, Left
[INFO] [1711187695.325217840] [task2_node]: Service name: task2
```

Figure 93: Launching a Simulated Task 1 Node

A master_node was started by running:

ros2 run master_node master_node --ros-args -p task_index:=1

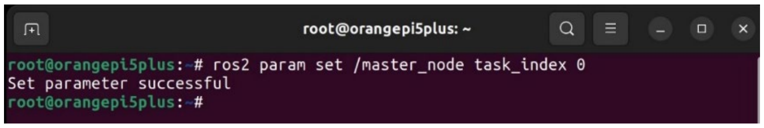


```
root@orangepi5plus:~# ros2 run master_node master_node --ros-args -p task_index:=1
Task Selected: task2
[INFO] [1711187699.710531070] [master_node]: [0] Found Task Node --> [task_node]...
[INFO] [1711187699.711727764] [master_node]: Current service name to request: task2
[INFO] [1711187699.714205733] [master_node]: Current responding task node name: task2
[INFO] [1711187699.714640019] [master_node]: [*] The task node is not yet completed.
[INFO] [1711187699.715061472] [master_node]: False, Buoyancy: Down, Thruster: Left,Duration (
Time): 0, Angle: 0
[INFO] [1711187699.715463384] [master_node]: Counting: 1
[INFO] [1711187700.124467753] [master_node]: Current service name to request: task2
[INFO] [1711187700.130282931] [master_node]: Current responding task node name: task2
[INFO] [1711187700.132791817] [master_node]: [*] The task node is not yet completed.
[INFO] [1711187700.135207078] [master_node]: False, Buoyancy: Down, Thruster: Left,Duration (
Time): 0, Angle: 0
[INFO] [1711187700.137684756] [master_node]: Counting: 2
[INFO] [1711187700.543208627] [master_node]: Current service name to request: task2
[INFO] [1711187700.547910818] [master_node]: Current responding task node name: task2
[INFO] [1711187700.549146303] [master_node]: [*] The task node is not yet completed.
[INFO] [1711187700.550717492] [master_node]: False, Buoyancy: Down, Thruster: Left,Duration (
Time): 0, Angle: 0
[INFO] [1711187700.552264181] [master_node]: Counting: 3
[INFO] [1711187700.961140219] [master_node]: Current service name to request: task2
```

Figure 94: Output Result in Master Node after Launching

From the above figures (**Figure 92-94**), it is observed that the Master Node requested task2_node instead of task1_node at startup. Therefore, this demonstration proves that the ROS2 Parameters API took effects on assigning the value (1) of the **task_index** to the current_node_index so that the Master Node looked up to the second service name in the task_nodes list (**task_nodes[1]= "task2"**) to implement requests. If users require the Master Node to request to the task1_node during the AUV operation, they can run the following command to establish the manipulation:

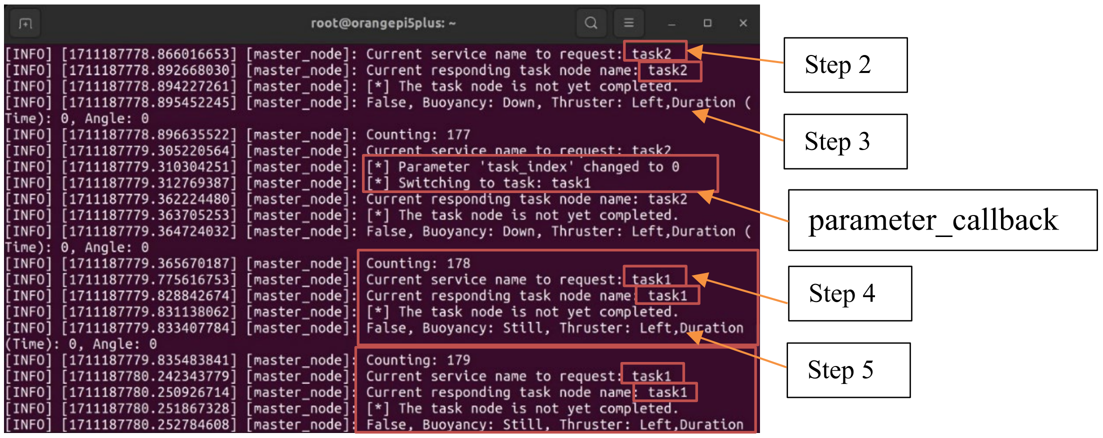
ros2 param set /master_node task_index 0



```
root@orangepi5plus:~# ros2 param set /master_node task_index 0
Set parameter successful
root@orangepi5plus:~#
```

Figure 95: ROS2 Parameters Set Command

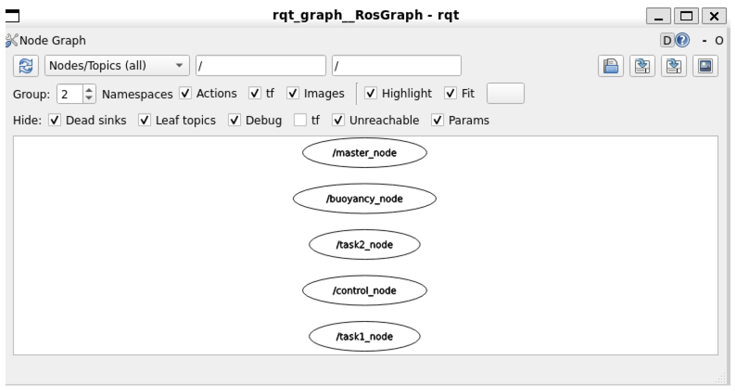
In Figure 96, the Master Node was hot-switched to request the task1_node at runtime since we used the ROS2 Parameters setting command to override the value of the **task_index** to 0. When the task_index was modified, the parameter_callback method was also triggered. This method reassigned the current_node_index by the task_index. Therefore, the Master Node was switched to request task2_node (**task_nodes[0]= "task1"**).



```
[INFO] [1711187778.866016653] [master_node]: Current service name to request: task2
[INFO] [1711187778.892668030] [master_node]: Current responding task node name: task2
[INFO] [1711187778.894227261] [master_node]: [*] The task node is not yet completed.
[INFO] [1711187778.895452245] [master_node]: False, Buoyancy: Down, Thruster: Left,Duration (
Time): 0, Angle: 0
[INFO] [1711187778.896635522] [master_node]: Counting: 177
[INFO] [1711187779.305220564] [master_node]: Current service name to request: task2
[INFO] [1711187779.310304251] [master_node]: [*] Parameter 'task_index' changed to 0
[INFO] [1711187779.312769387] [master_node]: [*] Switching to task: task1
[INFO] [1711187779.362224480] [master_node]: Current responding task node name: task2
[INFO] [1711187779.363705253] [master_node]: [*] The task node is not yet completed.
[INFO] [1711187779.364724032] [master_node]: False, Buoyancy: Down, Thruster: Left,Duration (
Time): 0, Angle: 0
[INFO] [1711187779.365670187] [master_node]: Counting: 178
[INFO] [1711187779.775616753] [master_node]: Current service name to request: task1
[INFO] [1711187779.828842674] [master_node]: Current responding task node name: task1
[INFO] [1711187779.831138062] [master_node]: [*] The task node is not yet completed.
[INFO] [1711187779.833407784] [master_node]: False, Buoyancy: Still, Thruster: Left,Duration
(Time): 0, Angle: 0
[INFO] [1711187779.835483841] [master_node]: Counting: 179
[INFO] [1711187780.242343779] [master_node]: Current service name to request: task1
[INFO] [1711187780.250926714] [master_node]: Current responding task node name: task1
[INFO] [1711187780.251867328] [master_node]: [*] The task node is not yet completed.
[INFO] [1711187780.252784608] [master_node]: False, Buoyancy: Still, Thruster: Left,Duration
```

Figure 96: Output Result in Master Node after ROS2 Parameters Reconfiguration

With this approach, the AUV can be switched manually to implement different autonomous tasks repeatedly or reattempt failed autonomous tasks. Therefore, the ROS2 Parameters API brings considerable advantages in controlling the AUV.

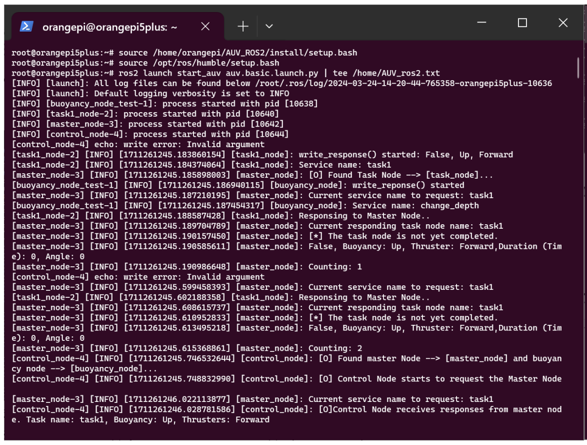


rqt_graph__RosGraph - rqt
Node Graph
Nodes/Topics (all)
Group: 2 Namespaces Actions tf Images Highlight Fit
Hide: Dead sinks Leaf topics Debug tf Unreachable Params
/master_node
/buoyancy_node
/task2_node
/control_node
/task1_node

Figure 97: Figure 84: rqt_graph showing all running nodes

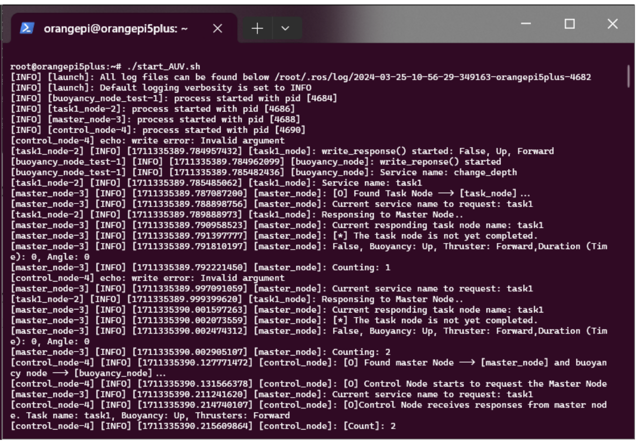
4.1.4 Demonstration of Launching AUV by ROS2 Launch

Once the ROS2-based control system for the AUV is proven to work, ROS2 Launch helps start all necessary nodes in one terminal to simplify the ROS2 deployment in the AUV. Taking the ROS2 Launch configuration provided in the Methodology section as an example, it is observed that the terminal displayed all loggings of the operating AUV nodes.



```
root@orangepi5plus:~# source /home/orangepi/AUV_ROS2/install/setup.bash
root@orangepi5plus:~# source /opt/ros/humble/setup.bash
root@orangepi5plus:~# ros2 launch start_auv auv.basic.launch.py | tee /home/AUV_ros2.txt
[INFO] [launch]: All log files can be found below /root/.ros/log/2024-03-24-14-20-44-765358-orangepi5plus-10636
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [buoyancy_node_test-1]: process started with pid [10638]
[INFO] [task1_node-2]: process started with pid [10640]
[INFO] [master_node-3]: process started with pid [10642]
[INFO] [control_node-4]: process started with pid [10644]
[control_node-4] echo: write error: Invalid argument
[task1_node-2] [INFO] [1711261245.183860154] [task1_node]: write_response() started: False, Up, Forward
[task1_node-2] [INFO] [1711261245.184374064] [task1_node]: Service name: task1
[master_node-3] [INFO] [1711261245.185898003] [master_node]: [O] Found Task Node --> [task_node]...
[buoyancy_node_test-1] [INFO] [1711261245.186940115] [buoyancy_node]: write_reponse() started
[master_node-3] [INFO] [1711261245.187210195] [master_node]: Current service name to request: task1
[buoyancy_node_test-1] [INFO] [1711261245.187454317] [buoyancy_node]: Service name: change_depth
[task1_node-2] [INFO] [1711261245.188587428] [task1_node]: Responsing to Master Node..
[master_node-3] [INFO] [1711261245.189704789] [master_node]: Current responding task node name: task1
[master_node-3] [INFO] [1711261245.190157450] [master_node]: [*] The task node is not yet completed.
[master_node-3] [INFO] [1711261245.190585611] [master_node]: False, Buoyancy: Up, Thruster: Forward,Duration (Tim
e): 0, Angle: 0
[master_node-3] [INFO] [1711261245.190986648] [master_node]: Counting: 1
[control_node-4] echo: write error: Invalid argument
[master_node-3] [INFO] [1711261245.599458393] [master_node]: Current service name to request: task1
[task1_node-2] [INFO] [1711261245.602188358] [task1_node]: Responsing to Master Node..
[master_node-3] [INFO] [1711261245.608615737] [master_node]: Current responding task node name: task1
[master_node-3] [INFO] [1711261245.610952833] [master_node]: [*] The task node is not yet completed.
[master_node-3] [INFO] [1711261245.613495218] [master_node]: False, Buoyancy: Up, Thruster: Forward,Duration (Tim
e): 0, Angle: 0
[master_node-3] [INFO] [1711261245.615368861] [master_node]: Counting: 2
[control_node-4] [INFO] [1711261245.746532644] [control_node]: [O] Found master Node --> [master_node] and buoyan
cy node --> [buoyancy_node]...
[control_node-4] [INFO] [1711261245.748832990] [control_node]: [O] Control Node starts to request the Master Node

[master_node-3] [INFO] [1711261246.022113877] [master_node]: Current service name to request: task1
[control_node-4] [INFO] [1711261246.028781586] [control_node]: [O]Control Node receives responses from master nod
e. Task name: task1, Buoyancy: Up, Thrusters: Forward
```



```
root@orangepi5plus:~# ./start_AUV.sh
[INFO] [launch]: All log files can be found below /root/.ros/log/2024-03-25-10-56-29-349163-orangepi5plus-4682
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [buoyancy_node_test-1]: process started with pid [4684]
[INFO] [task1_node-2]: process started with pid [4686]
[INFO] [master_node-3]: process started with pid [4688]
[INFO] [control_node-4]: process started with pid [4690]
[control_node-4] echo: write error: Invalid argument
[task1_node-2] [INFO] [1711335389.784957432] [task1_node]: write_response() started: False, Up, Forward
[buoyancy_node_test-1] [INFO] [1711335389.784962099] [buoyancy_node]: write_reponse() started
[buoyancy_node_test-1] [INFO] [1711335389.785482436] [buoyancy_node]: Service name: change_depth
[task1_node-2] [INFO] [1711335389.785485062] [task1_node]: Service name: task1
[master_node-3] [INFO] [1711335389.787087200] [master_node]: [O] Found Task Node ⟶ [task_node]...
[master_node-3] [INFO] [1711335389.788898756] [master_node]: Current service name to request: task1
[task1_node-2] [INFO] [1711335389.789888973] [task1_node]: Responsing to Master Node..
[master_node-3] [INFO] [1711335389.790958523] [master_node]: Current responding task node name: task1
[master_node-3] [INFO] [1711335389.791397777] [master_node]: [*] The task node is not yet completed.
[master_node-3] [INFO] [1711335389.791810197] [master_node]: False, Buoyancy: Up, Thruster: Forward,Duration (Tim
e): 0, Angle: 0
[master_node-3] [INFO] [1711335389.792221450] [master_node]: Counting: 1
[control_node-4] echo: write error: Invalid argument
[master_node-3] [INFO] [1711335389.997091059] [master_node]: Current service name to request: task1
[task1_node-2] [INFO] [1711335389.999399620] [task1_node]: Responsing to Master Node..
[master_node-3] [INFO] [1711335390.001597263] [master_node]: Current responding task node name: task1
[master_node-3] [INFO] [1711335390.002073559] [master_node]: [*] The task node is not yet completed.
[master_node-3] [INFO] [1711335390.002474312] [master_node]: False, Buoyancy: Up, Thruster: Forward,Duration (Tim
e): 0, Angle: 0
[master_node-3] [INFO] [1711335390.002905107] [master_node]: Counting: 2
[control_node-4] [INFO] [1711335390.127771472] [control_node]: [O] Found master Node ⟶ [master_node] and buoyan
cy node ⟶ [buoyancy_node]...
[control_node-4] [INFO] [1711335390.131566378] [control_node]: [O] Control Node starts to request the Master Node
[master_node-3] [INFO] [1711335390.211241620] [master_node]: Current service name to request: task1
[control_node-4] [INFO] [1711335390.214740107] [control_node]: [O]Control Node receives responses from master nod
e. Task name: task1, Buoyancy: Up, Thrusters: Forward
[control_node-4] [INFO] [1711335390.215609864] [control_node]: [Count]: 2
```

Figure 98: Using ROS2 Launch to start up nodes and their configurations
Figure 99: Using ROS2 Launch with a shell script

Therefore, utilizing ROS2 Launch is beneficial for debugging all nodes within the AUV subsystems and offers a convenient way to initiate the entire AUV system in practical application scenarios.

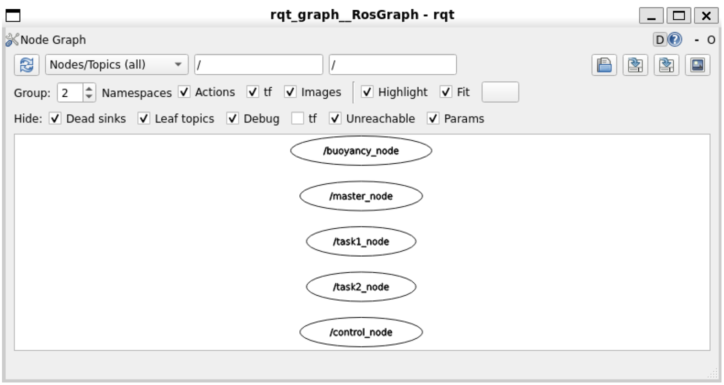

rqt_graph__RosGraph - rqt
Node Graph
Nodes/Topics (all)
Group: 2
Namespaces ✓ Actions ✓ tf ✓ Images ✓ Highlight ✓ Fit
Hide: ✓ Dead sinks ✓ Leaf topics ✓ Debug tf ✓ Unreachable ✓ Params
/buoyancy_node
/master_node
/task1_node
/task2_node
/control_node

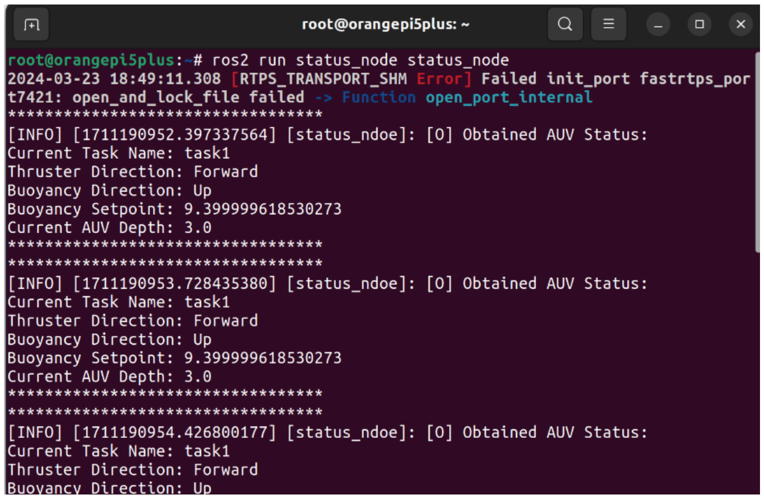
Figure 100: rqt_graph showing all running nodes

4.1.5 Demonstration of Status Node

After all nodes are launched by ROS2 Launch, it is also appropriate to monitor the AUV's status in a more friendly manner. As mentioned previously in the Methodology section, the Control Node also consists of a topics-based interface to publish information such as the received control commands and the AUV's current depth. Therefore, the Status Node is developed to subscribe to the information to allow users to monitor the behaviors of the AUV. The Status Node only consists of a topics-based interface to subscribe to a topic with the topics name "**AUV_status**".

To launch the Status Node, simply running:

ros2 run status_node status_node



```
root@orangepi5plus: ~
root@orangepi5plus:~# ros2 run status_node status_node
2024-03-23 18:49:11.308 [RTPS_TRANSPORT_SHM Error] Failed init_port fastrtps_por
t7421: open_and_lock_file failed -> Function open_port_internal
*********************************
[INFO] [1711190952.397337564] [status_ndoe]: [0] Obtained AUV Status:
Current Task Name: task1
Thruster Direction: Forward
Buoyancy Direction: Up
Buoyancy Setpoint: 9.399999618530273
Current AUV Depth: 3.0
*********************************
*********************************
[INFO] [1711190953.728435380] [status_ndoe]: [0] Obtained AUV Status:
Current Task Name: task1
Thruster Direction: Forward
Buoyancy Direction: Up
Buoyancy Setpoint: 9.399999618530273
Current AUV Depth: 3.0
*********************************
*********************************
[INFO] [1711190954.426800177] [status_ndoe]: [0] Obtained AUV Status:
Current Task Name: task1
Thruster Direction: Forward
Buoyancy Direction: Up
```

Figure 101: Output Result of Master Node at Startup

From the above figure (**Figure 101**), the Status Node displayed necessary information about the AUV system, such as the current control commands required by the task node, the buoyancy depth to be set, and the current depth value of the AUV. Given the simplicity and clarity of the terminal output, displaying this node on the AUV's internal screen can significantly enhance the efficiency of testing and commissioning for the AUV, ensuring that the AUV is functioning as intended and allowing operators to intervene if necessary.

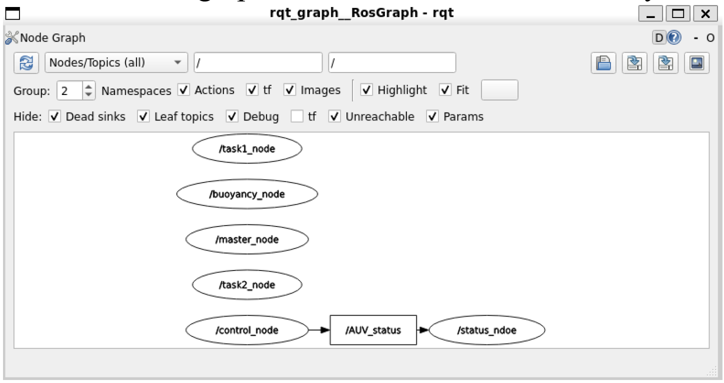
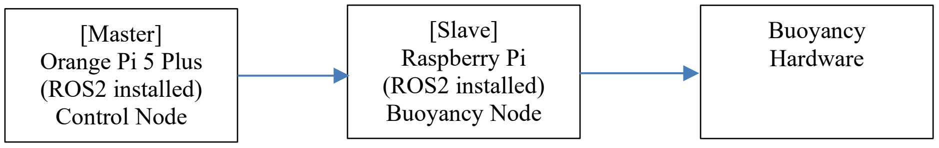


Figure 102: rqt_graph showing all running nodes and an arrow indicating ROS2 Topics interface

4.1.6 ROS2 Network between Orange Pi 5 Plus and Raspberry Pi

The ROS2 framework enables seamless communication between different ROS2 nodes across multiple computing devices on the same network. In the context of this project, the OP5P acts as the master device, responsible for running the Control Node, the Master Node, and several task nodes. Simultaneously, the RPi is assigned to operate the buoyancy node. This configuration provides substantial benefits, with the OP5P offering substantial computing power to execute autonomous programs and manage the AUV system, while the RPi leverages its GPIO pins and mature control libraries to control the buoyancy subsystem.



With this approach, team members utilizing the RPi for developing the buoyancy system are not required to port their code to the OP5P platform. Instead, the RPi can communicate directly with the OP5P over the ROS2 network, allowing for seamless control of the AUV. This setup exemplifies a master-slave relationship, where the OP5P and RPi can interact harmoniously within the same IP segment, with the OP5P acting as a **bridge** to assign an IP address to the RPi. Therefore, the ROS2 network enhances the AUV system's flexibility and scalability, enabling efficient distribution of tasks across diverse hardware platforms.

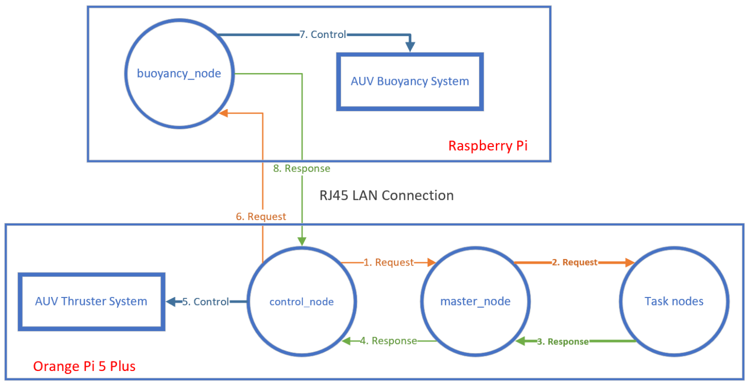
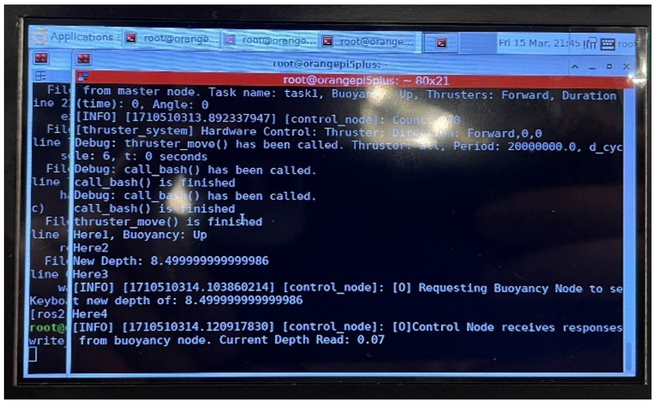
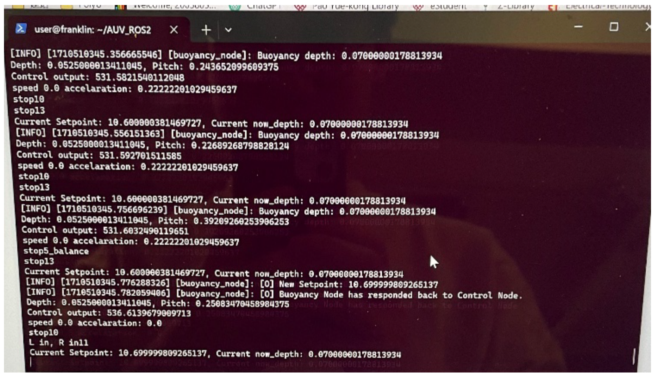


Figure 103: Ethernet Connection between Raspberry Pi and Orange Pi 5 Plus



```
root@orangepi5plus: ~ 80x21
 from master node. Task name: task1, Buoya    Up, Thrusters: Forward, Duration
(time): 0, Angle: 0
[INFO] [1710510313.892337947] [control_node]: Coun   70
[thruster_system] Hardware Control: Thruster: Dir    n: Forward,0,0
Debug: thruster_move() has been called. Thruster:    , Period: 20000000.0, d_cyc
le: 6, t: 0 seconds
Debug: call_bash() has been called.
call_bash() is finished
Debug: call_ba  () has been called.
call_bash() is finished
thruster_move() is finished
Here1, Buoyancy: Up
Here2
New Depth: 8.499999999999986
Here3
[INFO] [1710510314.103860214] [control_node]: [0] Requesting Buoyancy Node to se
t new depth of: 8.499999999999986
Here4
[INFO] [1710510314.120917830] [control_node]: [0]Control Node receives responses
 from buoyancy node. Current Depth Read: 0.07
```

Figure 104: Control System Nodes Running on Orange Pi 5 Plus



```
[INFO] [1710510345.356665546] [buoyancy_node]: Buoyancy depth: 0.07000000178813934
Depth: 0.0525000013411045, Pitch: 0.243652099609375
Control output: 531.5821540112048
speed 0.0 accelaration: 0.2222201029459637
stop10
stop13
Current Setpoint: 10.600000381469727, Current now_depth: 0.07000000178813934
[INFO] [1710510345.556151363] [buoyancy_node]: Buoyancy depth: 0.07000000178813934
Depth: 0.0525000013411045, Pitch: 0.22689268798828124
Control output: 531.592701511585
speed 0.0 accelaration: 0.2222201029459637
stop10
stop13
Current Setpoint: 10.600000381469727, Current now_depth: 0.07000000178813934
[INFO] [1710510345.756696239] [buoyancy_node]: Buoyancy depth: 0.07000000178813934
Depth: 0.0525000013411045, Pitch: 0.39209260253906253
Control output: 531.6032490119651
speed 0.0 accelaration: 0.2222201029459637
stop5_balance
stop13
Current Setpoint: 10.600000381469727, Current now_depth: 0.07000000178813934
[INFO] [1710510345.776288326] [buoyancy_node]: [O] New Setpoint: 10.699999809265137
[INFO] [1710510345.782059406] [buoyancy_node]: [O] Buoyancy Node has responded back to Control Node.
Depth: 0.0525000013411045, Pitch: 0.25083470458984375
Control output: 536.6139679009713
speed 0.0 accelaration: 0.0
stop10
L in, R in11
Current Setpoint: 10.699999809265137, Current now_depth: 0.07000000178813934
```

Figure 105: Buoyancy Node Running on Raspberry Pi

From the above figures (**Figure 104-105**), the first display represents the OP5P running task nodes (autonomous programs), the Master Node, and the Control Node, while the second terminal represents the Raspberry Pi running the Buoyancy Node and executing functions to control its GPIO as well as the buoyancy system. This demonstration showed that ROS2 nodes can be run on separate devices and communicate with each other under the same network.

4.2 Demonstration of IR Remote Control for Controlling AUV's Thrusters

In the previous section, we discussed the importance of developing test methods to counteract the rotating speed of the AUV's thrusters to prevent the AUV from deviating from its intended path due to inconsistent thruster speeds. This section demonstrates the use of an infrared (IR) remote control to control the speed of the AUV's thrusters.

In this program, the process is as follows:

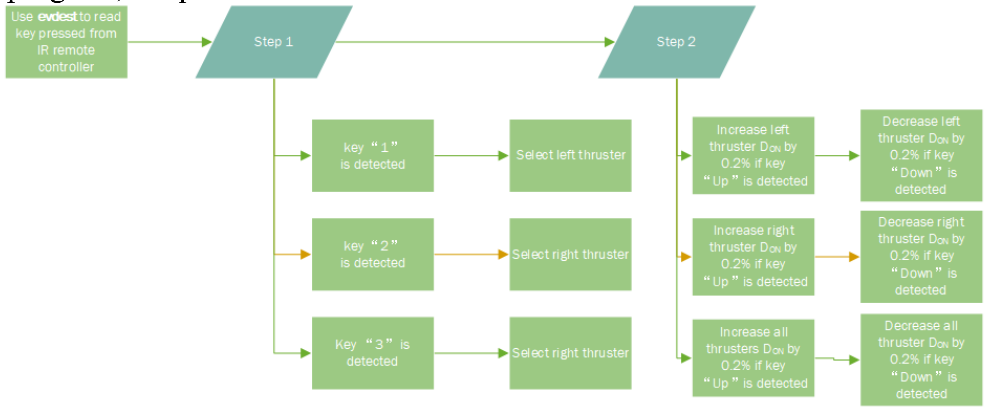
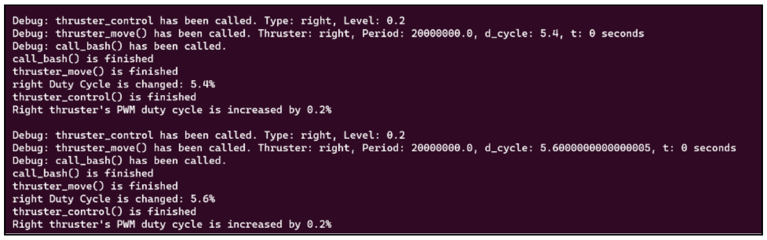

Use evdest to read key pressed from IR remote controller
Step 1
Step 2
key "1" is detected
Select left thruster
key "2" is detected
Select right thruster
Key "3" is detected
Select right thruster
Increase left thruster D_ON by 0.2% if key "Up" is detected
Decrease left thruster D_ON by 0.2% if key "Down" is detected
Increase right thruster D_ON by 0.2% if key "Up" is detected
Decrease right thruster D_ON by 0.2% if key "Down" is detected
Increase all thrusters D_ON by 0.2% if key "Up" is detected
Decrease all thruster D_ON by 0.2% if key "Down" is detected

Figure 106: Control Process for IR Remote Control

First, a key on the IR remote control was pressed to select the specific AUV thruster to control. For instance, pressing the "2" key would select the AUV right thruster's PWM channel. Once the right thruster was selected, the users pressed the "up" or "down" key to adjust the pulse-width modulation (PWM) duty cycle of the OP5P by increments of 0.2%. This corresponded to a change in the PWM duty cycle the ESC received as well as the thruster speed.



```
Debug: thruster_control has been called. Type: right, Level: 0.2
Debug: thruster_move() has been called. Thruster: right, Period: 20000000.0, d_cycle: 5.4, t: 0 seconds
Debug: call_bash() has been called.
call_bash() is finished
thruster_move() is finished
right Duty Cycle is changed: 5.4%
thruster_control() is finished
Right thruster's PWM duty cycle is increased by 0.2%

Debug: thruster_control has been called. Type: right, Level: 0.2
Debug: thruster_move() has been called. Thruster: right, Period: 20000000.0, d_cycle: 5.6000000000000005, t: 0 seconds
Debug: call_bash() has been called.
call_bash() is finished
thruster_move() is finished
right Duty Cycle is changed: 5.6%
thruster_control() is finished
Right thruster's PWM duty cycle is increased by 0.2%
```

Figure 107: IR Remote Control on increasing the rotating speed of the AUV right thruster.

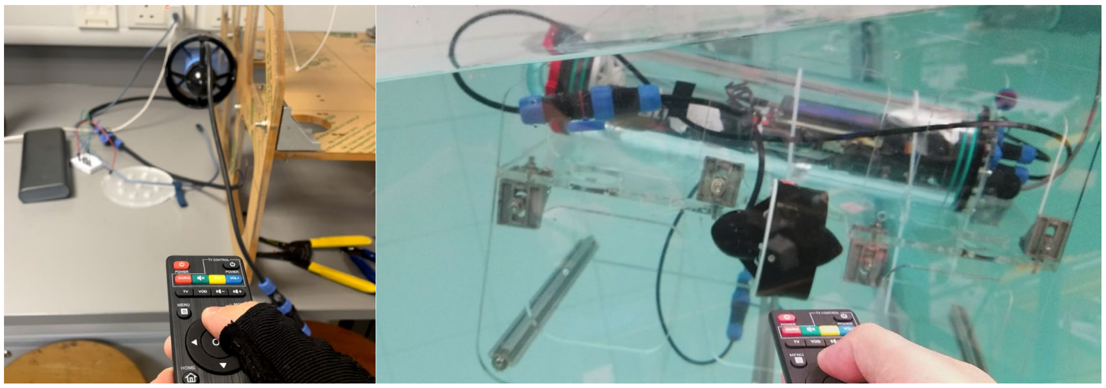


Figure 108: Adjusting the AUV thruster's speed by pressing the "UP" and "Down" button on IR remote control

Using this method, the operator can finely offset the speed of each thruster independently, ensuring that the AUV maintains its trajectory and does not experience drift due to imbalanced thrust. This precise control is crucial for navigating the AUV through complex underwater environments and executing complicated tasks. A detailed demonstration will be provided during the project demonstration.

4.3 Demonstration of 6-bit Telecommunication

In the 6-bit telecommunication control, an Arduino was connected to a keypad to allow users to select input numbers, and the Arduino would continuously "press" buttons on a 27MHz controller by turning on a relay to short-circuit the buttons so that constant signals were transmitted. This approach ensures that the buttons are pressed continuously, and therefore, the 27MHz receiver would not receive unstable or misleading signals.

When the 27MHz receiver received the signals, another Arduino proceeded to convert the signals to a byte or an integer. Then, the Arduino would display the value with an SSD1306 display and transmit the value as a string to computers such as the OP5P through USB UART serial communication.

Take the below as an illustration. When the number "1" was pressed on the keypad, the Arduino on the transmitter side received the command and then continuously turned on the first relay. Therefore, one of the buttons in the 27MHz transmitter was short-circuited and it triggered a "press" action to activate the transmitter to send signals continuously. When the 27MHz receiver received the signals, one of the six outputs became high-level. When the Arduino on the receiver side read the six outputs in the GPIO, it counted the high-level and converted the signals to a byte. Eventually, the byte value of 59 was displayed on an SSD1306 OLED display.

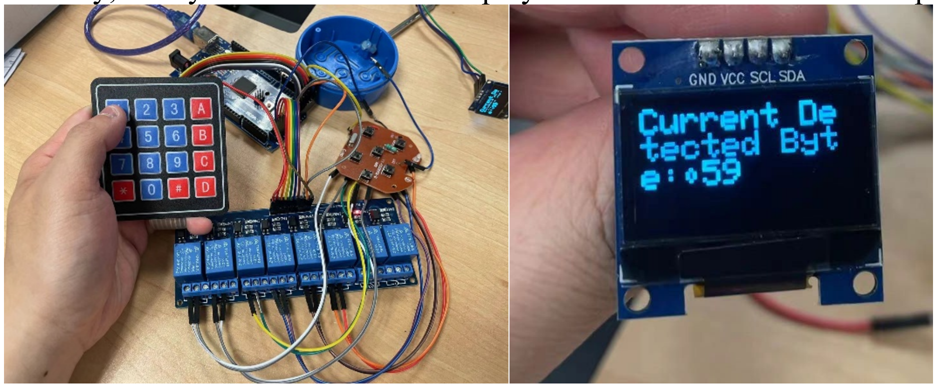


Figure 109: Pressing "1" on Keypad and Receiving End Displays "59"

When the button "2" was pressed on the keypad, the Arduino on the transmitter side turned off the first relay and turned on the second relay, resulting in another button being pressed continuously on the transmitter. Therefore, the receiver received another constant signal, and the converted value (byte) was 58 on the receiver end.

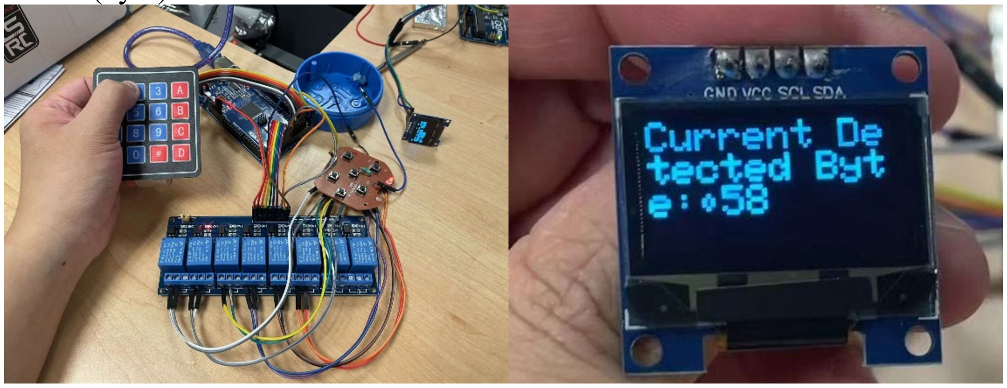


Figure 110: Pressing "2" on Keypad and Receiving End Displays "58"

Similarly, when the button "3" was pressed on the keypad, the receiver received another constant signal and the converted value was 51.

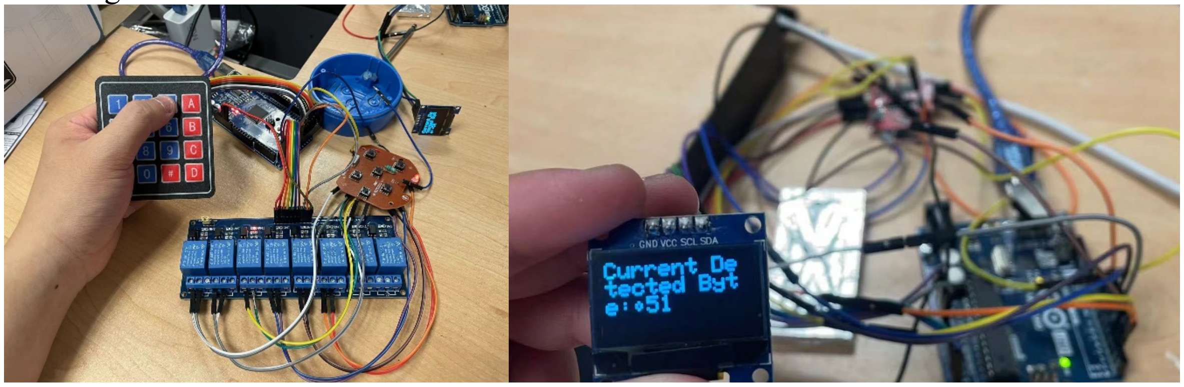


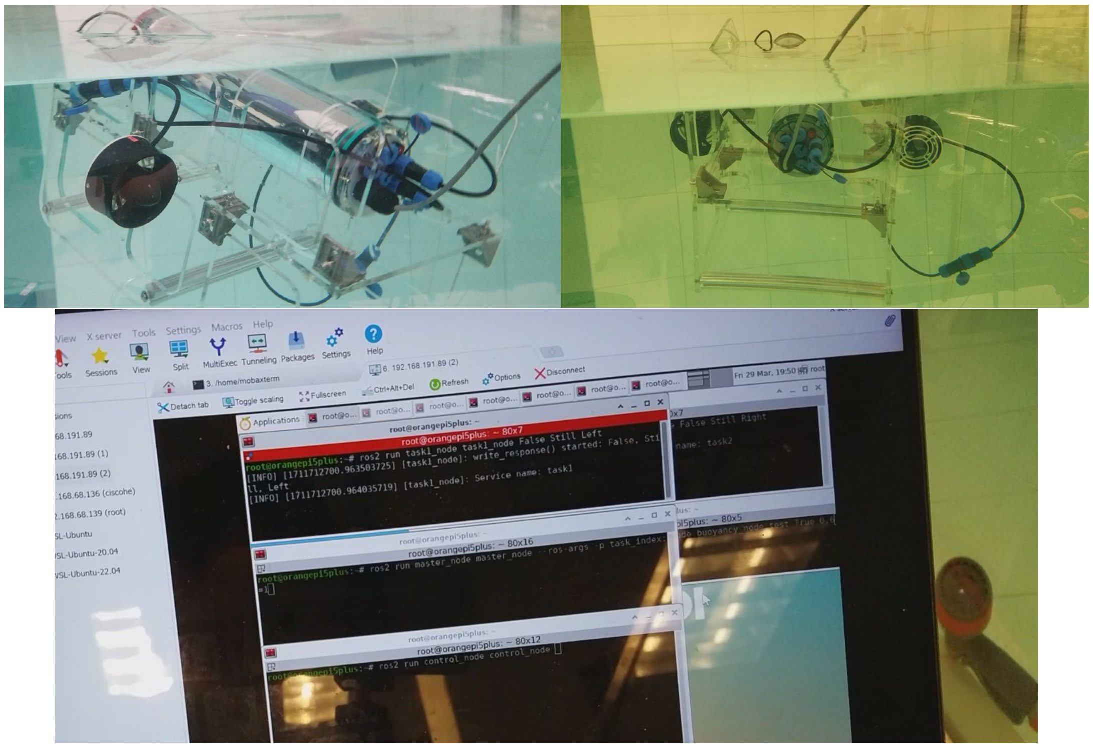
Figure 111: Pressing "3" on Keypad and Receiving End Displays "51"

With this approach, the buttons 1-6 on the keypad would turn on 6 relays respectively, resulting in 6 constant signals being transmitted to the receiver and 6 respective values converted. Therefore, the OP5P or the AUV control system can utilize the 6 values to implement various functions, such as switching the requesting order between the Master Node and the task nodes. A detailed demonstration will be provided during the project demonstration.

4.4 Demonstration of AUV Control System in Real AUV

Since the buoyancy subsystem developed by teammates was considerably difficult to commission individually and some electronic components for the buoyancy subsystem, such as the depth sensor and the MPU6050 gyroscope were malfunctioning and unstable after an accident of water leakage during the testing, the integrations between the AUV control system and the buoyancy subsystem was not able to be demonstrated eventually. However, the ROS2-based AUV control system successfully demonstrated executing control commands from simulated task nodes to control the thruster subsystem in the AUV.

For example, during the real test, the Control Node was able to execute the control commands from the simulated task nodes (autonomous programs) to move forward and turn left and right accordingly. The Master Node was also able to select the correct task nodes to forward the control commands to the Control Node. Furthermore, the real task nodes (autonomous subsystem) could also successfully transmit control commands to control the AUV hardware through the control nodes, although the autonomous programs could not reveal their functionality owing to space constraints in the laboratory. However, the real AUV underwater test could prove that the program demonstrations for the ROS2-based developed AUV control system in the above sections (Section 4.1.1-4.1.3) are effective and confident in handling autonomous program control commands to control the hardware system of the AUV.



```
root@orangepi5plus: ~ 80x7
root@orangepi5plus:~# ros2 run task1_node task1_node False Still Left
[INFO] [1711712700.963503725] [task1_node]: write_response() started: False, Sti
ll, Left
[INFO] [1711712700.964035719] [task1_node]: Service name: task1
```

```
False Still Right
name: task2
```

```
root@orangepi5plus: ~ 80x16
root@orangepi5plus:~# ros2 run master_node master_node --ros-args -p task_index:
=1
```

```
root@orangepi5plus: ~ 80x5
de buoyancy_node test True 0.
```

```
root@orangepi5plus: ~ 80x12
root@orangepi5plus:~# ros2 run control_node control_node
```

Figure 102: Demonstration of AUV Underwater with ROS2 Control System

A detailed demonstration will be provided during the project demonstration.

5. Discussion

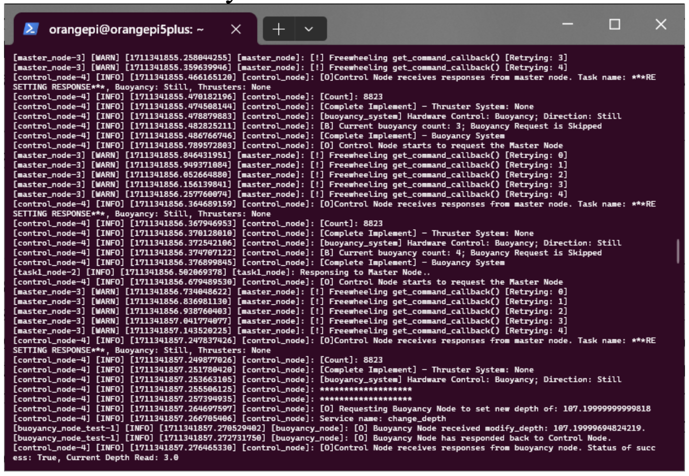
5.1 Analysis of Results

Upon completion of this project, a robust AUV control system was successfully developed, enabling the AUV's subsystems to effectively process commands from autonomous programs and manage the vehicle's hardware and behaviors accordingly. For instance, both the thruster subsystem and the buoyancy subsystem correctly received control commands from the autonomous subsystem. Within the AUV control system, the master node functioned as a "reverse proxy," appropriately requesting task nodes and forwarding control commands to the relevant control nodes. Moreover, the AUV control system facilitated a manual override feature, allowing operators to switch between autonomous tasks as needed. The project also implemented various ROS2 launch configurations, ensuring a seamless one-click startup for the entire AUV system. In terms of the AUV's electrical architecture and essential underwater mechanisms, the selected battery met the power requirements of the AUV, while the DC-DC converter provided stable voltage regulation. The designed frame, 3D-printed mounting rack, and battery box contributed to the AUV's stable operation underwater.

Overall, the project successfully achieved its initial objectives and project goal, which was the development of an AUV hardware and control system utilizing ROS2. The outcomes of this project lay a solid foundation for further advancements in autonomous underwater vehicle technology.

5.2 Limitations and Weaknesses

Despite the successful development of the AUV control system, several limitations and weaknesses were identified during the project. One significant concern is the potential for lost responses between the Master Node and the autonomous task nodes within the ROS2 control system. Although the project incorporated compensation methods to recover communication between the control system nodes, the resulting latency led to time consumption and unnecessary expenditure of the AUV's power and computational resources. This highlights the need for further research into how to prevent response loss, which could significantly improve the efficiency and performance of the control system.



```
[master_node-3] [WARN] [1711341855.258044255] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 3]
[master_node-3] [WARN] [1711341855.359639946] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 4]
[control_node-4] [INFO] [1711341855.466165120] [control_node]: [O]Control Node receives responses from master node. Task name: ***RE
SETTING RESPONSE***, Buoyancy: Still, Thrusters: None
[control_node-4] [INFO] [1711341855.470182196] [control_node]: [Count]: 8823
[control_node-4] [INFO] [1711341855.474508144] [control_node]: [Complete Implement] - Thruster System: None
[control_node-4] [INFO] [1711341855.478879883] [control_node]: [buoyancy_system] Hardware Control: Buoyancy; Direction: Still
[control_node-4] [INFO] [1711341855.482825211] [control_node]: [B] Current buoyancy count: 3; Buoyancy Request is Skipped
[control_node-4] [INFO] [1711341855.486766746] [control_node]: [Complete Implement] - Buoyancy System
[control_node-4] [INFO] [1711341855.789572803] [control_node]: [O] Control Node starts to request the Master Node
[master_node-3] [WARN] [1711341855.846431951] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 0]
[master_node-3] [WARN] [1711341855.949371084] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 1]
[master_node-3] [WARN] [1711341856.052664880] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 2]
[master_node-3] [WARN] [1711341856.156139841] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 3]
[master_node-3] [WARN] [1711341856.257760074] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 4]
[control_node-4] [INFO] [1711341856.364689159] [control_node]: [O]Control Node receives responses from master node. Task name: ***RE
SETTING RESPONSE***, Buoyancy: Still, Thrusters: None
[control_node-4] [INFO] [1711341856.367946953] [control_node]: [Count]: 8823
[control_node-4] [INFO] [1711341856.370128010] [control_node]: [Complete Implement] - Thruster System: None
[control_node-4] [INFO] [1711341856.372542106] [control_node]: [buoyancy_system] Hardware Control: Buoyancy; Direction: Still
[control_node-4] [INFO] [1711341856.374707122] [control_node]: [B] Current buoyancy count: 4; Buoyancy Request is Skipped
[control_node-4] [INFO] [1711341856.376899845] [control_node]: [Complete Implement] - Buoyancy System
[task1_node-2] [INFO] [1711341856.502069378] [task1_node]: Responsing to Master Node..
[control_node-4] [INFO] [1711341856.679489530] [control_node]: [O] Control Node starts to request the Master Node
[master_node-3] [WARN] [1711341856.734048622] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 0]
[master_node-3] [WARN] [1711341856.836981130] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 1]
[master_node-3] [WARN] [1711341856.938760403] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 2]
[master_node-3] [WARN] [1711341857.041774077] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 3]
[master_node-3] [WARN] [1711341857.143520225] [master_node]: [!] Freewheeling get_command_callback() [Retrying: 4]
[control_node-4] [INFO] [1711341857.247837426] [control_node]: [O]Control Node receives responses from master node. Task name: ***RE
SETTING RESPONSE***, Buoyancy: Still, Thrusters: None
[control_node-4] [INFO] [1711341857.249877026] [control_node]: [Count]: 8823
[control_node-4] [INFO] [1711341857.251780420] [control_node]: [Complete Implement] - Thruster System: None
[control_node-4] [INFO] [1711341857.253663105] [control_node]: [buoyancy_system] Hardware Control: Buoyancy; Direction: Still
[control_node-4] [INFO] [1711341857.255506125] [control_node]: *******************
[control_node-4] [INFO] [1711341857.257394935] [control_node]: *******************
[control_node-4] [INFO] [1711341857.264697597] [control_node]: [O] Requesting Buoyancy Node to set new depth of: 107.19999999999818
[control_node-4] [INFO] [1711341857.266705406] [control_node]: Service name: change_depth
[buoyancy_node_test-1] [INFO] [1711341857.270529402] [buoyancy_node]: [O] Buoyancy Node received modify_depth: 107.19999694824219.
[buoyancy_node_test-1] [INFO] [1711341857.272731750] [buoyancy_node]: [O] Buoyancy Node has responded back to Control Node.
[control_node-4] [INFO] [1711341857.276465330] [control_node]: [O]Control Node receives responses from buoyancy node. Status of succ
ess: True, Current Depth Read: 3.0
```

Figure 103: Potential lost responses in the AUV Control System

Another limitation pertains to the interface used for communication between the buoyancy node and the control node. The current ROS2 services-based interface, which employs frequency control to manage the buoyancy subsystem, is suboptimal. This approach requires developers to manually adjust the buoyancy_count to match the required timing for buoyancy adjustments.

Therefore, a more effectively improved solution would involve an **actions-based interface**. With actions, the buoyancy node could provide real-time feedback on the AUV's current depth to the Control Node, allowing the Control Node to automatically wait until the AUV reaches the desired depth before commencing the next implementation cycle. This would enhance the responsiveness and precision of the AUV's buoyancy control, which is crucial for the AUV's overall performance and safety.

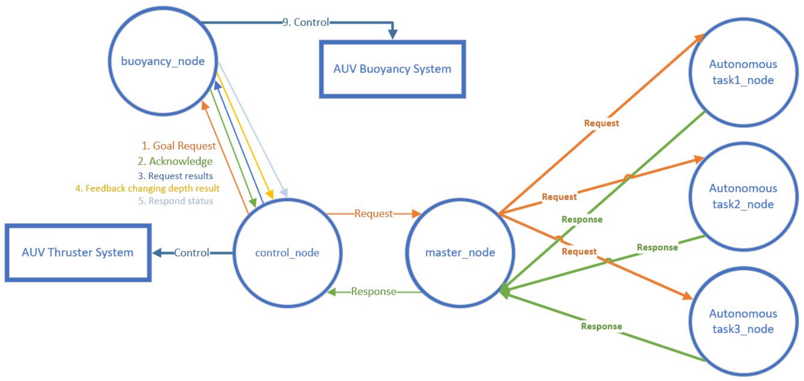


Figure 104: Utilizing ROS2 Actions-based interface between Control Node and Buoyancy Node

5.3 Suggestions for Improvement:

Apart from the AUV control system, the proposed 27MHz telecommunication method can also be embedded in the ROS2 system to allow extra functionality. For example, the operators can send requests to the AUV ROS2 nodes to establish certain behaviours, such as switching autonomous tasks and changing configurations for task nodes, etc., through telecommunication. This method can be developed by allowing the OP5P to read received control bytes from the 27MHz receiver directly (GPIO) or through the RPi Pico's UART communication. Hence, the OP5P implements different request actions for different nodes based on the control bytes.

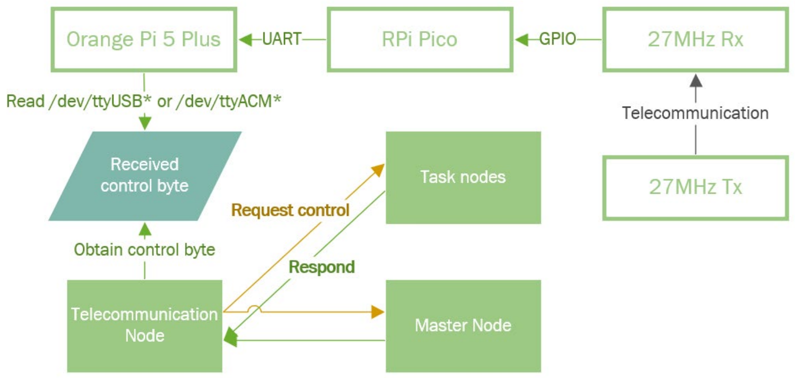


Figure 105: Embedding 27MHz Telecommunication in ROS2 Control System

6. Conclusion

6.1 Project Achievements

This project has successfully developed a hardware system for an autonomous underwater vehicle (AUV), which includes the electrical architecture and essential mechanisms for underwater functionality. The AUV control system, built on ROS2, effectively utilizes ROS2 client library APIs to establish a robust communication layer for the AUV's subsystems, such as the autonomous, buoyancy, and thruster subsystems, although the autonomous subsystem and the buoyancy subsystem were not able to demonstrate eventually. Furthermore, the project has introduced practical telecommunication solutions, including IR and 27MHz remote control, enabling efficient remote control, configuration, testing, and commissioning of the AUV system.

6.2 Learning Outcomes

This project has yielded several significant learning outcomes:

1. Acquired the structure design of an AUV, such as the AUV electrical system and the mechanical system.
2. Gaining experience in controlling Linux operating systems.
3. Learning GPIO control, PWM control, UART, and I2C communications in embedded systems.
4. Acquiring Python object-oriented programming skills.
5. Utilizing ROS2 software libraries effectively.
6. Building ROS2 nodes according to personal design plans.
7. Developing effective communication skills for team collaboration on the AUV project.

7. References

[1] Multidisciplinary engineering education through marine engineering projects, https://ieeexplore.ieee.org/document/8084572 (accessed Mar. 19, 2024).

[2] Autonomous Underwater Vehicle (AUV) for mapping marine biodiversity in coastal and shelf waters: Implications for marine management, https://www.researchgate.net/publication/224183107_Autonomous_Underwater_Vehicle_AUV_for_mapping_marine_biodiversity_in_coastal_and_shelf_waters_Implications_for_marine_management (accessed Mar. 19, 2024).

[3] Russell B. Wynn a et al., "Autonomous underwater vehicles (auvs): Their past, present and future contributions to the Advancement of Marine Geoscience," Marine Geology, https://www.sciencedirect.com/science/article/pii/S0025322714000747 (accessed Mar. 19, 2024).

[4] "AUVs in marine science," Picsea, https://picsea.co.uk/services/marine-science/ (accessed Mar. 19, 2024).

[5] V. A. M. Jorge et al., "A survey on unmanned surface vehicles for Disaster Robotics: Main challenges and directions," Sensors (Basel, Switzerland), https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6387351/ (accessed Mar. 19, 2024).

[6] Design of a micro-AUV for autonomy development and multi-vehicle ..., https://ieeexplore.ieee.org/document/8084807 (accessed Mar. 19, 2024).

[7] The Singapore AUV Challenge 2024 Rulebook, https://sauvc.org/rulebook/ (accessed Mar. 19, 2024).

[8] R. and Markets, "Global AUV & ROV Market Report 2023: Industry Analysis 2019-2022 and Forecasts 2023-2030 - Rise in Demand in Offshore Oil and Gas Sector and Under the Ice in the Arctic," GlobeNewswire News Room, Apr. 24, 2023. https://www.globenewswire.com/en/news-release/2023/04/24/2653038/28124/en/Global-AUV-ROV-Market-Report-2023-Industry-Analysis-2019-2022-and-Forecasts-2023-2030-Rise-in-Demand-in-Offshore-Oil-and-Gas-Sector-and-Under-the-Ice-in-the-Arctic.html (accessed Mar. 19, 2024).

[9] R. A. Fernandez, E. A. Parra R., Z. Milosevic, S. Dominguez, and C. Rossi, "Design, modeling and control of a spherical autonomous underwater vehicle for Mine Exploration," 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Oct. 2018. doi:10.1109/iros.2018.8594016

[10] A.-G. Ștefan, L. Ștefăniță Grigore, S. Marzavan, I. Priescu, and I. Oncioiu, "Theoretical and experimental aspects regarding the forced mounting of a cylinder containing the electronics of a mini submarine," MDPI, https://www.mdpi.com/2077-1312/9/8/855 (accessed Mar. 21, 2024).

[11] Robot operating system 2: Design, architecture, and uses in the wild,

https://www.science.org/doi/10.1126/scirobotics.abm6074 (accessed Mar. 21, 2024).

[12] M. Quigley et al., "ROS: an open-source Robot Operating System." Available: http://robotics.stanford.edu/~ang/papers/icraoss09-ROS.pdf.

[13] "Understanding topics," Understanding topics - ROS 2 Documentation: Humble documentation, https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Topics/Understanding-ROS2-Topics.html (accessed Mar. 21, 2024).

[14] Understanding services - ROS 2 Documentation: Humble documentation, https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html (accessed Mar. 21, 2024).

[15] R. Patil, "HTTP request, HTTP response, context and headers : Part III.," Medium, https://medium.com/@rohitpatil97/http-request-http-response-context-and-headers-part-iii-5c37bd4cb06b (accessed Mar. 21, 2024).

[16] Understanding actions - ROS 2 Documentation: Humble documentation, https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html (accessed Mar. 21, 2024).

[17] Interfaces - ROS 2 Documentation: Humble documentation, https://docs.ros.org/en/humble/Concepts/Basic/About-Interfaces.html (accessed Mar. 21, 2024).

[18] methylDragon, GitHub, https://github.com/methylDragon/ros-tutorials/blob/master/ROS/03%20ROS%20-%20Messages%2C%20Services%2C%20and%20Actions.md (accessed Mar. 24, 2024).

[19] msg/Image Documentation, https://docs.ros2.org/latest/api/sensor_msgs/msg/Image.html (accessed Mar. 24, 2024).

[20] Quality of Service settings - ROS 2 Documentation: Humble documentation, https://docs.ros.org/en/humble/Concepts/Intermediate/About-Quality-of-Service-Settings.html (accessed Mar. 24, 2024).

[21] Topics vs Services vs Actions - ROS 2 Documentation: Foxy documentation, https://docs.ros.org/en/foxy/How-To-Guides/Topics-Services-Actions.html (accessed Mar. 24, 2024).

[22] B. Farriz Mohd, "Design and Development of an Autonomous Underwater Vehicle (AUV-FKEUTeM)," Available: https://core.ac.uk/reader/235651210

[23] "How to control BLDC motor using PWM?," BLDC Motor, https://www.bldcmotor.org/how-to-control-bldc-motor-using-pwm.html (accessed Mar. 24, 2024).

[24] "Orange Pi 5 Plus - Wiki-Orange Pi," Orangepi.org, 2023.
http://www.orangepi.org/orangepiwiki/index.php/Orange_Pi_5_Plus (accessed Mar. 24, 2024).

[25] "What is a reverse proxy server," NGINX,
https://www.nginx.com/resources/glossary/reverse-proxy-server/ (accessed Mar. 27, 2024).

[26] What is a reverse proxy, https://www.cloudflare.com/learning/cdn/glossary/reverse-proxy/ (accessed Mar. 27, 2024).

[27] 1. Introduction - The Linux Kernel documentation, https://docs.kernel.org/input/input.html (accessed Mar. 27, 2024).

[28] BrickExperimentChannel et al., "RC submarine 4.0 – radio (7/10)," Brick Experiment Channel, https://brickexperimentchannel.wordpress.com/2022/07/13/rc-submarine-4-0-radio-7-10/ (accessed Mar. 27, 2024).

8. Appendix

Appendix 8.1: Control Method for AUV's thrusters

- Arduino: https://connectpolyu-my.sharepoint.com/:u:/g/personal/20058054d_connect_polyu_hk/EWDswrgtXzpNmFh9DNdYvgMBUgdoqgWoFS_yk7yhfVHNWw?e=hPucKh

Appendix 8.2: 27MHz receiver and UART Communication between OP5P and Pico

- Pico/ESP32/Arduino: https://connectpolyu-my.sharepoint.com/:u:/g/personal/20058054d_connect_polyu_hk/EX7030HHlt1OjlJwtx4WA2oBpcfNrrQjb0MdSqkAiWWjjA?e=uAwUJo
- Orange Pi 5 Plus: https://connectpolyu-my.sharepoint.com/:u:/g/personal/20058054d_connect_polyu_hk/EVzv8gdGtFtNoSTfHKc71FwBq3LDU0Jfg2EvrsquBQIbAw?e=NKSxXf

Appendix 8.3: 27MHz transmitter with keypad and 8-module relay

- Arduino: https://connectpolyu-my.sharepoint.com/:u:/g/personal/20058054d_connect_polyu_hk/EQzyvJNuMoBIpX6H9_5HRMYBbjAcPqaIty3T3erYR-F0yg?e=5KDp0k

Appendix 8.4: Reading Key values from IR remote control and evdest

- Orange Pi 5 Plus: https://connectpolyu-my.sharepoint.com/:u:/g/personal/20058054d_connect_polyu_hk/EcIs5F1kxnhDqWCfDilIKmsBjAKqaY36D4M0E5jr-IXuaA?e=AiQnY4

Appendix 8.5: IR_control() with thruster_move()

- Orange Pi 5 Plus: https://connectpolyu-my.sharepoint.com/:u:/g/personal/20058054d_connect_polyu_hk/EQUeumkzE5lMj-wRbxieti4BwUm6sbFH1u75Ji02or9Wxw?e=71pLuv

Appendix 8.6: Source Code for ROS2-based AUV Control System

https://github.com/zenkernelsam/PolyU_EEE_AUV2024_FYP/tree/master/src

- **Control Node:** https://github.com/zenkernelsam/PolyU_EEE_AUV2024_FYP/blob/master/src/control_node/control_node/control_node.py
- **Master Node:** https://github.com/zenkernelsam/PolyU_EEE_AUV2024_FYP/blob/master/src/master_node/master_node/master_node.py
- **Simulated Task 1 Node:** https://github.com/zenkernelsam/PolyU_EEE_AUV2024_FYP/blob/master/src/master_node/master_node/master_node.py

- **Simulated Task 2 Node:** https://github.com/zenkernelsam/PolyU_EEE_AUV2024_FYP/blob/master/src/master_node/master_node/master_node.py
- **Simulated Buoyancy Node:** https://github.com/zenkernelsam/PolyU_EEE_AUV2024_FYP/blob/master/src/buoyancy_node/buoyancy_node/buoyancy_node_test.py
- **Buoyancy Node for Raspberry Pi:** https://github.com/zenkernelsam/PolyU_EEE_AUV2024_FYP/blob/master/src/buoyancy_node/buoyancy_node/buoyancy_node.py
- **Status Node:** https://github.com/zenkernelsam/PolyU_EEE_AUV2024_FYP/blob/master/src/status_node/status_node/status_node.py
- **Control Interfaces (Services - srv):** https://github.com/zenkernelsam/PolyU_EEE_AUV2024_FYP/blob/master/src/control_interfaces/srv/BuoyancyControl.srv https://github.com/zenkernelsam/PolyU_EEE_AUV2024_FYP/blob/master/src/control_interfaces/srv/GetCommand.srv https://github.com/zenkernelsam/PolyU_EEE_AUV2024_FYP/blob/master/src/control_interfaces/srv/GetTask.srv
- **Control Interfaces (Topics - msg):** https://github.com/zenkernelsam/PolyU_EEE_AUV2024_FYP/blob/master/src/control_interfaces/msg/AUVStatus.msg
- ROS2 Launch Configuration: https://github.com/zenkernelsam/PolyU_EEE_AUV2024_FYP/blob/master/src/start_auv/launch/auv.launch.py

Appendix 8.7: Progress Logging

In 2023/24 semester 1, this project has achieved the following objectives (sort by time):

1. Studying and designing AUV's electrical system (September-December 2023)
2. Full control of T200 thrusters by PWM signals from Orange Pi 5 Plus (October 2023)
3. **[Deprecated in AUV Project]** Data extraction from pressure sensor (October-November 2023)
4. Circuit soldering and wiring (November-December 2023)
5. 3D printing for electrical rack (December 2023)
6. IR remote control for T200 thrusters and water pump in the buoyancy system (December 2023)
7. **[Deprecated in AUV Project]** Migration of the buoyancy system's control logic from teammate to Orange Pi 5 Plus (December 2023)
8. AUV assembling and operating tests (December 2023-January 2024)
9. Video Submission for SAUVC (January 2024)

In 2023/24 semester 2, this project has achieved the following objectives (sort by time):

1. Developed 27MHz remote telecommunication methods (January 2024)
2. Studying ROS2 Client Library APIs (Python) (January-February 2024)
3. Studying Python Object-oriented programming (January-February 2024)
4. Designing and developing ROS2 nodes for the AUV control system framework (February-March 2024)
5. Testing and optimizing ROS2 nodes for the AUV control system (March 2024)
6. Consolidating nodes' interfaces from other teammates' AUV subsystems to interact with the AUV control system nodes (March 2024)
7. Testing the AUV with ROS2 platform in underwater (March 2024)