6-11-2020

Neptune: Marine Robot ROV Control System

Alex Achramowicz

Chris Layco

Cooper Zediker

SANTA CLARA UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Date: June 11, 2020

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Alex Achramowicz
Chris Layco
Cooper Zediker

ENTITLED

Neptune: Marine Robot ROV Control System

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

Christopher A Kitts
Christopher A Kitts (Jun 12, 2020 07:15 PDT)

Thesis Advisor

Nam Ling
Nam Ling (Jun 12, 2020 07:45 PDT)

Department Chair

Neptune: Marine Robot ROV Control System

by

Alex Achramowicz
Chris Layco
Cooper Zediker

Submitted in partial fulfillment of the requirements for the
degree of
Bachelor of Science in Computer Science and Engineering
School of Engineering
Santa Clara University

Santa Clara, California
June 11, 2020

Neptune: Marine Robot ROV Control System

Alex Achramowicz
Chris Layco
Cooper Zediker

Department of Computer Science and Engineering
Santa Clara University
June 11, 2020

ABSTRACT

The mysteries of the aquatic world and the dangers human activity poses to its environmental health are important considerations. It is pivotal we find efficient methods to study and monitor the subsea environments in order to maintain and improve their health. This document provides insight into the planning, design, implementation, and testing in developing an enhanced control system for a marine remotely operated vehicle (ROV) to effectively control various mounted hardware components. Our goal for this design is to greatly assist scientists and environmentalists observe and collect subsea data to improve the subsea environment.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Over 70% of Earth's surface is covered in water. However, more than 80% of the world's oceans have yet to be explored and mapped for scientific purposes [1]. While all bodies of water are significant to life on earth, it can be argued that those greatest in danger are the shores and shallow bodies of water such as lakes and beaches. Because of their close proximity to human developments, these bodies of water are heavily relied upon. However, this also means that the environment and ecosystems they house are prone to degradation. These ecosystems are becoming polluted due to human proximity and land use. Sources of water pollution include litter, sewage, construction, fertilizer, oil, and radioactive substances[2].

Researchers track, monitor, and enforce pollution levels in order to prevent damage to ecosystems. The process of monitoring water pollution is currently difficult, time consuming, and dangerous. Conventionally, divers collect water samples manually, but this has the limitation of being resource intensive and limits the depth that the researchers can sample. There is a recent trend of using remotely controlled underwater vehicles (ROVs) to replace human divers for marine experiments and missions. These ROVs range in size, but are typically the size of a suitcase and are connected to a terminal on a boat via a tether line. Once functionality is confirmed on the boat, the ROV is thrown overboard and is controlled with the terminal by researchers. These ROVs have increased capabilities to sample water directly, view subsea environments, and deploy sensors to track various data.

1.2 Industrial Application

For the past few years, Santa Clara University (SCU) has been assisting the United States Geological Survey (USGS), the California Department of Wildlife, and private research facilities in building and deploying simple ROVs to help monitor different shallow water properties. Recently, there has been a demand for ROVs with more comprehensive instrumentation to better collect and understand underwater data. Our project is to design the software and control systems for a new ROV iteration which meets these demands. We will be working with a team of Mechanical Engineers

who will be responsible for its mechanical design. Our team will design and integrate control systems, instrument communication, and error handling into the mechanical design. We will include several modules and functions that will further assist researchers and organizations, such as the USGS, in their efforts to research, monitor, and control shallow subsea environments. Among these modules are a live video feed and deployment of new sensors. We will also include extra control lines and feeds for future sensors that researchers may want or build which will increase modularity and reusability of the ROV for the future.

1.3 Customers and Stakeholders

Dr. Geoff Wheat, a research professor at the University of Alaska Fairbanks, has been a recurring sponsor of projects from SCU's Robotics Systems Lab (RSL) for almost 20 years. He is the main customer for our project and has provided insight, requirements, and needs for the ROVs development in order to best fit his research purposes. In addition, general stakeholders of the project include the USGS, the Monterey Bay Aquarium Research Institute (MBARI), which Dr. Wheat is affiliated with, and Dr. Christopher Kitts—director of the RSL and our advisor who guided project development.

1.4 Similar Systems

1.4.1 Triton

In 1999 the RSL deployed the Triton ROV (Figure 1.1) to conduct visual-based underwater missions as a joint project with Oregon State University and the NASA Ames Research Center [6]. More recently it has been operated on by SCU students to conduct research missions in local bodies of water such as Monterey Bay and Lake Tahoe. This system carries a heavy load (approximately 250 lbs.), requires multiple people to deploy, and runs on a high voltage. It does have data capabilities with proven reliability on research missions[7].

Figure 1.1: Triton ROV[5].

1.4.2 Tessie

The RSL and a team of graduate students have recently completed the build of the Tessie ROV in 2016 (Figure 1.2). The engineers spent around $7,400 out of a $10,000 budget—closer to the mid to high end of the spectrum in terms of cost. This ROV was built with robustness in mind along with a live camera feed, sensor capabilities, and depth capabilities up to 500 feet. All of these capabilities were to facilitate easy maneuvering, observations, and collection of data in aquatic bodies such as Monterey Bay and Lake Tahoe [3].

Figure 1.2: Tessie ROV[3].

1.4.3 BlueROV2

Blue Robotics BlueROV2 is an industry standard ROV (Figure 1.3). Ranging in budget from $2,784 to $4,584 (depending on additional hardware purchases) the BlueROV2 is at the lower end of the spectrum in terms of cost and provides open-source software and electronics to control. It provides a live camera feed but requires the user to purchase and build additional sensors for data collection. The BlueROV2 can delve to depths up to 330 feet, 170 feet less than Tessie [4]. This system is also difficult to modify to better fit customer demands, considering the internal wiring and control systems are proprietary.

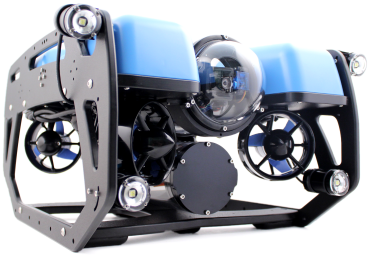


Figure 1.3: Blue Robotics BlueROV2[4].

1.5 Objectives

The goal of the project was to develop a low-cost ROV with improved control, software modularity, and safety features compared to current features on the ROVs owned and operated by the RSL. As mentioned in Section 1.2, due to the large scope of the project we teamed up with an undergraduate mechanical engineering team who were primarily responsible for retrofitting the existing robot frame with new components using mounts and a ballasting system, among many other objectives. Our main area of work included developing and implementing a control system to communicate with the retrofit components in order to effectively maneuver the robot via thrusters and collect data via sensors, video, and sampling. In detail this work included: ensuring the electronic functionality of the components, developing a graphical user interface (GUI) to allow for user input and display data and a video feed, and implementing a software protocol to allow two-way communication from the user to the components through the GUI and chosen microcontroller.

We will further detail these objectives in the next chapter. Throughout the paper we will show the extent to which these objectives were met. As a summary, we successfully tested the functionality of all components using our microprocessor and topside system, implemented a GUI for user benefit, and, using the Robot Operating System (ROS), implemented a node to enforce communication between our main system components.

Given our results this project has established a new system architecture for a low-cost, safety-enhanced, and modern data capture capabilities ROV. Neptune is the first of its kind in the RSL with implementation of on-board batteries for safety and ROS for increased software modularity. In the future our customers and stakeholders can use our ROV for research, subsea data capturing, and future enhancements.

Chapter 2

System Overview

This chapter will provide an overview of our design, starting with the customer needs that were the starting point for our design. The customer needs section is a list of various needs that were compiled from communication with our customer and stakeholders. From those needs, our team determined tangible requirements that we could measure, observe, and design for. After creating a list of requirements, our team was able to design a system that could fulfill these requirements. We designed baseline diagrams shown throughout the chapter to reference throughout the project. These diagrams include: concept of operations, use cases, mechanical, component, and basic and advanced software. At the end of the chapter, we provide list the technologies we used for our design with the rationale behind each of our decisions.

2.1 Customer Needs

This Customer Needs section will detail the information given to our team from stakeholders and customers that defined the scope of our project.

As mentioned in Chapter 1, the main customer for this system is Dr. Geoff Wheat, from the University of Alaska Fairbanks, and general stakeholders include Dr. Kitts and MBARI. Dr. Wheat spoke directly with our Mechanical Engineering team to provide a list of needs, as they had conducted multiple interviews with him online. Input from Dr. Kitts was given directly to our team, as well as to the Mechanical Engineering team during weekly meetings. His continual feedback gave us a more refined idea of our product, as he met with our team frequently, and guided our development process.

Both gave crucial guidance to our development, and based on their feedback we compiled a list of customer needs—shown in Appendix A. Of this list, it was clear to us that our system had to be safe, reliable, accurate, and have high-performance. Both Dr. Wheat and Dr. Kitts specified that our product must be able to maneuver well underwater, and be piloted with little latency. Due to this, we emphasized a lightweight and fast communication protocol to ensure minimal latency between user input and vehicle movement.

Another key attribute that came from our customer needs was safety. This was mostly in regard to the safety of the electronics that are underwater, more specifically the decision to implement subsea, on-board power. Although a lot of the responsibility for hardware decisions and waterproofing was directed towards the Mechanical Engineering Team, we had work in accordance with their constraints when picking control system components for the ROV.

2.2 Requirements

From the Customer Needs in the previous chapter, we determined tangible requirements that we could measure, observe, and design for given the constraints. Our requirements are divided into two categories: Functional and Non-functional. If a requirement is functional, it can be measured using observation, and can be verified in testing. These requirements have been assessed and the results are shown in Chapters 3, 4, 5, and 6. A non-functional requirement is an attribute of our system that is necessary in order for our product to be of high-quality. It cannot be physically measured or observed, but can be apparent through usage and experience with our system. The completion of these requirements were subject to constraints that bounded the scope of our work.

In each of the categories, requirements are divided into further groupings: critical and recommended. A critical requirement is one that is integral to the system's success. A recommended requirement is one that system is not dependent upon, but would increase the quality of our product substantially. We aimed to complete as many recommended requirements in the given time, but prioritized finishing our critical requirements first.

Critical
Convert user input (input/output from joystick or commands)
Measure depth plus/minus 5 feet accuracy
Measure temperature plus/minus 5 degrees Fahrenheit accuracy
Output real-time data to monitor
Heading plus/minus 5 degree accuracy
Under 0.5 second response time
Control water sampling device to take in 100-200mL per sample
Operable up to a depth of 500 feet
Recommended
Autopilot depth lock
Autopilot heading lock
Tele-operations

Table 2.1: Functional requirements.

Table 2.1 contains the critical functional requirements necessary for a completed system to function and the recommended requirements that were stretch goals. In the following chapters the paper covers the extent to which these requirements were met. Section 8.3 covers the recommended requirements in-depth. We wanted all these requirements to be measurable or observable for better testing and results.

Critical
Performance
Usability
Recommended
Availability
Data Integrity

Table 2.2: Non-functional requirements.

The critical non-functional requirements define system attributes necessary for the quality of the system. The system must perform well by having a fast response time. It must have an intuitive user interface so any user can operate the ROV. Availability is a recommended requirement because the system must always be open to the user. Data loss is an important consideration considering the nature of data collection, but the ROV is not in danger of imminent security threats and data loss is often unavoidable due to constraints.

Budget
Hardware performance
Memory limitations
Communication distance

Table 2.3: Constraints.

Table 2.3 covers the constraints that limited design, development, implementation, and testing. Budget is a constraint on all projects, and despite receiving funding it still limited the hardware we could purchase. The performance of hardware introduced varying throughput and accuracy for measurements such as response time and data reads. With our chosen on-board microcontroller we were limited to 256kB of flash memory that limited code storage. Lastly, the operational requirement for a communication distance of 500 feet filtered the types of communication methods for the system and increased reponse time.

2.3 Concept of Operations

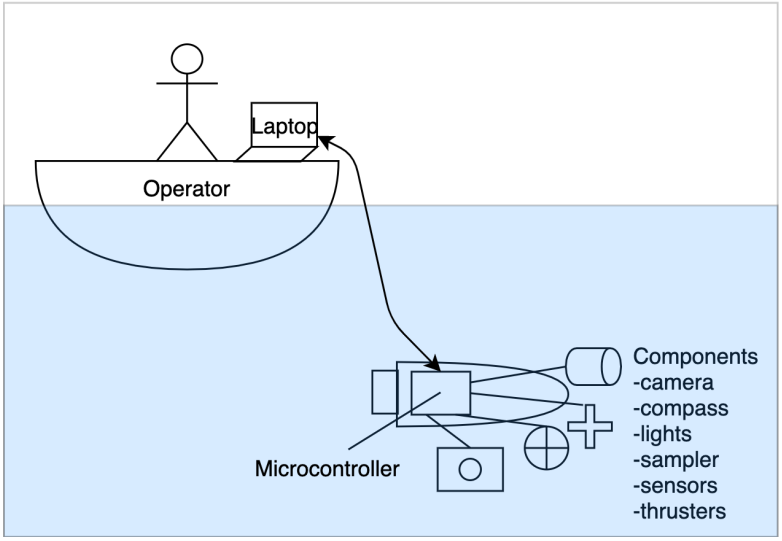


Figure 2.1: Concept of operations diagram.

The above figure provides an overview of the basic characteristics of operations for the system. Topside there will be a operator who will control the ROV via a laptop. The commands will be sent through a tether to the microcontroller that is mounted on the ROV. The microcontroller will parse the commands and direct the command to the correct component. If the component returns data it will be sent back to the topside laptop through the tether.

2.4 Use Cases

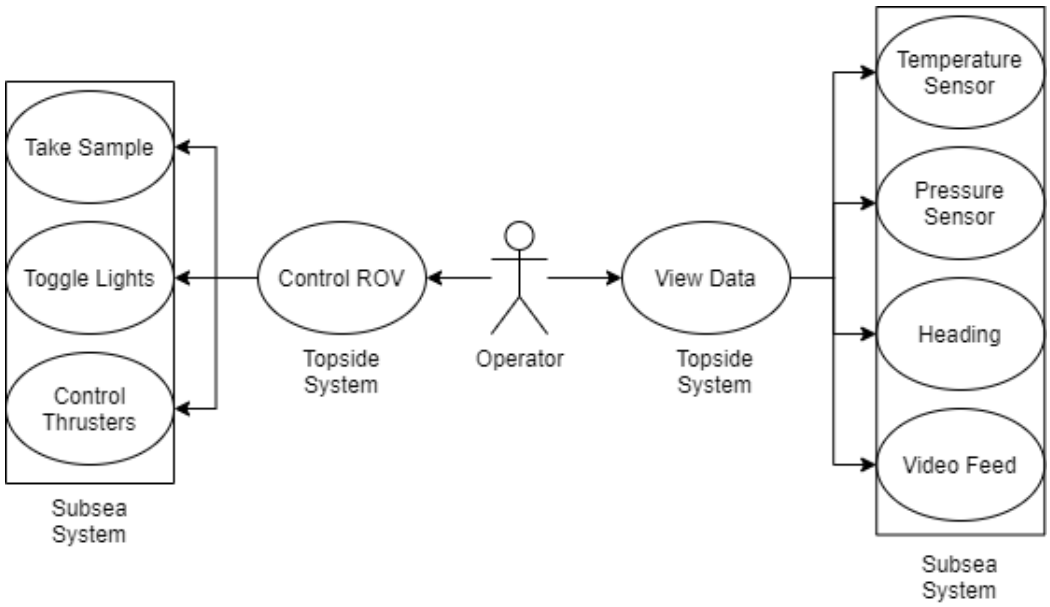


Figure 2.2: Use cases diagram.

The operator is the only user and has direct access to the topside system. From the topside machine the user can control the ROV subsea components and send commands to control thrusters, toggle lights, or take a water sample. Alternatively on the topside system the user can view data communicated from the subsea system. This data is either a video feed, heading read, temperature read, or pressure read.

2.5 Mechanical Diagram

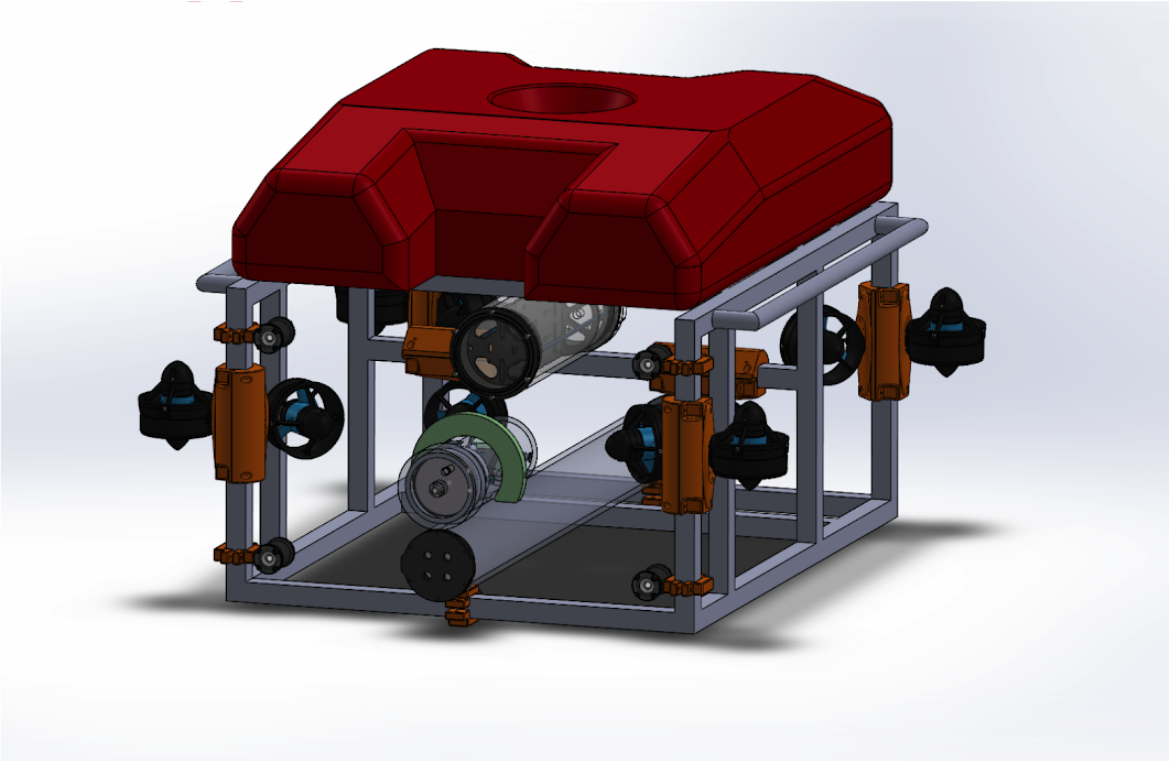


Figure 2.3: Mechanical diagram drawn with CAD Software.

Figure 2.3 is a mechanical diagram of our project. There are ten thrusters mounted with varying angles on the frame with 3D printed mounts (orange). The clear cylindrical tubes mounted in the center of the frame are waterproof and contain the various hardware components from the concept of operations diagram. The tube vertically above the green tube is the control bottle which holds our on-board microcontroller, electronic speed controllers (ESCs), and compass. The green tube is the battery bottle that holds the lithium polymer (LiPo) batteries used for power during missions. The long tube under the battery bottle is used for collecting water samples using a water sampler designed by the mechanical engineering team.

2.6 Basic Component Diagram

Figure 2.4: Basic component diagram.

Figure 2.4 is a basic component diagram that shows a high-level overview of how the main components of the system are connected. The system has a graphical user interface (GUI) connected over a communications system to the on-board control system. The GUI is also connected to the video camera on a separate electrical branch.

2.7 Basic Software Diagram

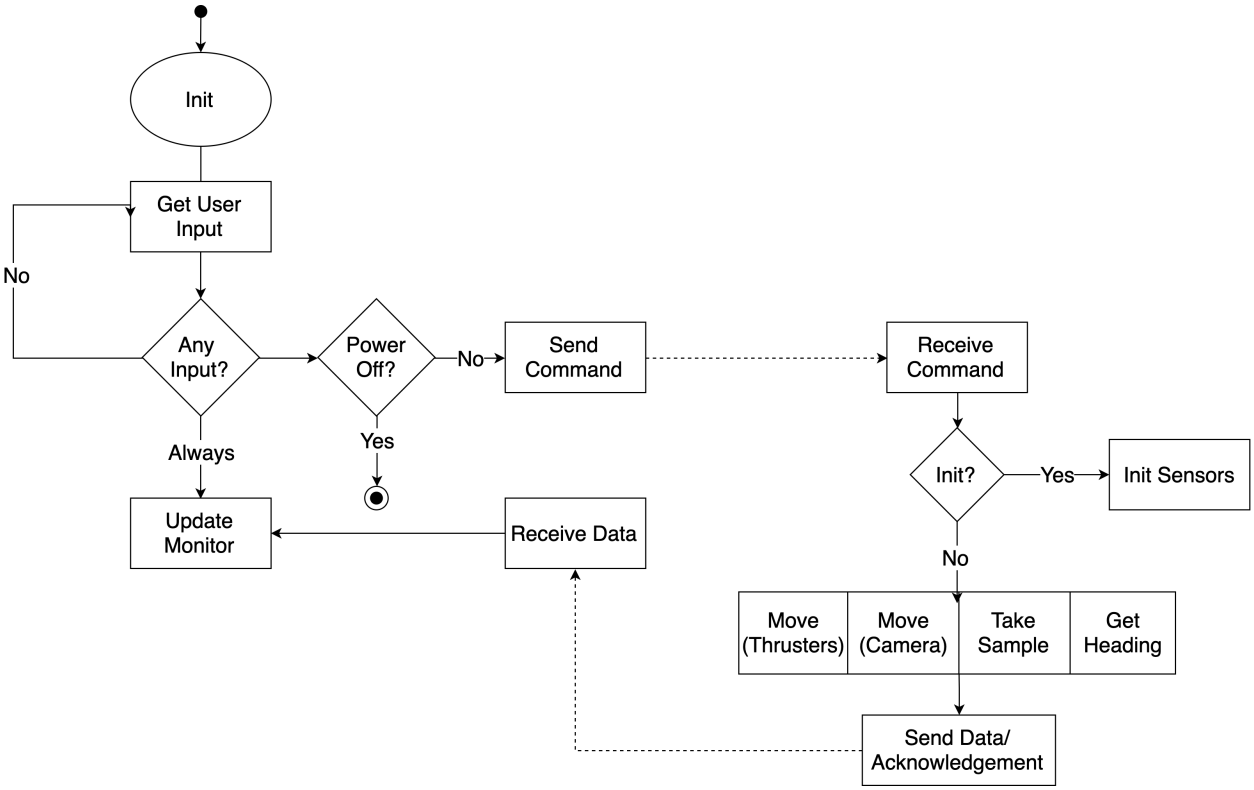


Figure 2.5: Basic software diagram.

Our software system consists of two main sections as can be seen in the Figure 2.5. The first section (left) consists of the software flow on our topside laptop. Our main loop consists of waiting for user input. Once user input is received,

the system checks if the input is to shut down the ROV. If so, the system sends a message to the ROV to power off both sections. Any other command is automatically sent to the other half of the system, the subsea on-board system (right). This system will wait for commands to be received through the tether and will parse the commands to complete the requested action (i.e. move camera, move thrusters, take sample, etc.). Once the action is completed a message is sent back to the topside system with information and an acknowledgement of receiving the initial message. The topside receives the message and updates the display with the information.

2.8 Advanced Software Diagram

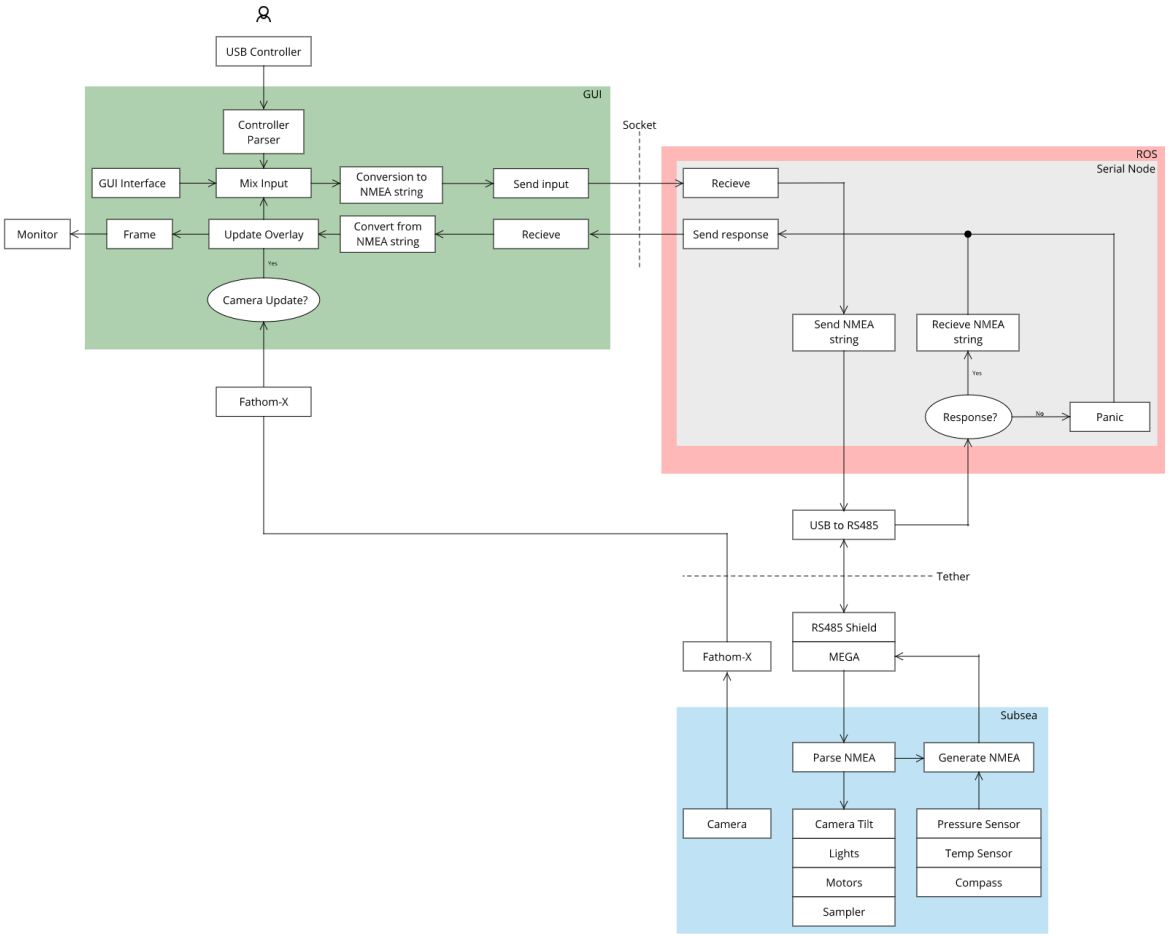


Figure 2.6: Advanced Software Diagram.

Figure 2.6 presents a more architectural, advanced version of the basic software diagram shown in Figure 2.5. Color coded in green is the GUI system responsible for parsing the user controller input to control the thrusters (see blue subsea system). It mixes the thruster commands with the interface inputs (to control lights, camera tilt, and/or sampler) to produce a control string to send subsea. The ROS system (red), is a node responsible for receiving the control string generated from GUI. It sends the string over RS485, through the tether, to the Arduino MEGA 2560 Rev3

microcontroller using a RS485 shield. The subsea system (blue) parses the string and directs the command to the correct component—camera tilt, lights, thrusters, or sampler.

Using a master-slave architecture, which will be covered in-depth in Chapter 4, the subsea system will generate a NMEA string containing pressure, temperature, and heading data that is sent back to the ROS node over the tether through RS485. The ROS node checks for valid data and panics if absent, otherwise it sends the data string back to the GUI over the socket. The GUI converts the string to update the display overlay with the given data. In a separate software branch the GUI receives the camera feed from the subsea system over the tether using two Fathom-X boards to amplify the signal. The GUI updates the overlay with this video feed as well as any additional frames produced on the monitor.

The next four chapters will go in-depth into four main subsystems: GUI, Communications, On-board Control, and Video. This diagram will often be used as a reference in these chapters.

2.9 Technologies Used

This section contains a list of key technologies chose in our design. These technologies played a huge factor in how we implemented our code and built the ROV. Each technology is followed by a brief description for background.

- **Arduino Mega 2560 Rev3**: A small micro-controller which has 50 digital inputs and 15 analog inputs. This hosts our system to receive commands from the tether and parse them into the proper components.
- **Robot Operating System (ROS)**: A Linux based system which has built in libraries for robotics systems. This sends and receive commands and data over the tether.
- **Tkinter**: A de-facto standard Python GUI package. It is a thin object-oriented layer on top of Tcl/Tk.
- **RS485**: A communication protocol which specializes in long distance, reliable communications.
- **I2C**: A communication protocol that focuses on buses. Most of our components support I2C communication so, we used this for ease of communications.
- **LiPo Batteries**: A high power density power battery that is susceptible to over and under charging and discharging. Can be volatile if handled incorrectly, so was placed separate from our other components.

2.10 Design Rationale

This section justifies the decision to use the previous listed technologies for our project. Some of the key attributes that our design needed to uphold were speed, reliability, and safety, which stems from the Customer Needs Section

(Section 2.1). Each of these technologies were chosen because they play some role in bolstering one or more of these attributes.

Arduino Mega 2560: We chose this microcontroller over any other microprocessor or computer because of the cost and number of input pins. The Mega has excellent input support compared to most other microcontrollers. But, the decision for the Mega comes at the cost of lower processing power and memory when compared to a computer or microprocessor such as a Raspberry Pi. Given the low processing nature of our system we concluded this trade-off was appropriate. One side benefit of using the Mega is that it also has low power draw which will assist in extending mission times.

Tkinter: Tkinter was the best choice for building a simple GUI with basic functionality. Since it was a built-in Python package with an intuitive toolkit, we believed that it would be easy to understand and use for our implementation. The only trade-off would be aesthetics, as Tkinter is known for not having many pleasing visual elements. However, we this trade-off was worth it because we are more concerned with functionality.

Robot Operating System (ROS): We chose to use ROS on our topside system because it has been proven successful in other designs in the RSL. There were RSL members with expertise that we wanted to leverage in our implementation. In addition, ROS has a wide range of built in functionality for robotic systems. It contains multiple packages which allowed us to easily handle user input and send and receives messages and commands to control the ROV.

RS485: We needed to consider data loss and reliability over the long distance due to the tether length. We chose to use RS485 because it is designed to be reliable over long distances, which is necessary for consistent to communication with our ROV. Deciding to use this protocol means that we needed an Arduino shield to support the communications, which was a trade-off we were willing to accept.

I2C: Our choice to use I2C is simple based on the fact that all of our components support this communication protocol. Therefore, implementing this protocol into our system was relatively simple and easy compared to making our own communication system.

LiPo Batteries: These batteries have a high power density and are therefore a good choice for extending our possible mission time, despite its volatility.

Chapter 3

Graphical User Interface

This chapter covers the first of the four main subsystems: the GUI. The GUI is primarily responsible for taking input from a pilot and displaying necessary data from the ROV to the pilot. Another key component to the GUI is its ability to interface with the communications subsystem to ensure proper data transmission to the subsea system.

3.1 Software Diagram: GUI

Below is the cropped GUI section of the advanced software diagram from Figure 2.6.

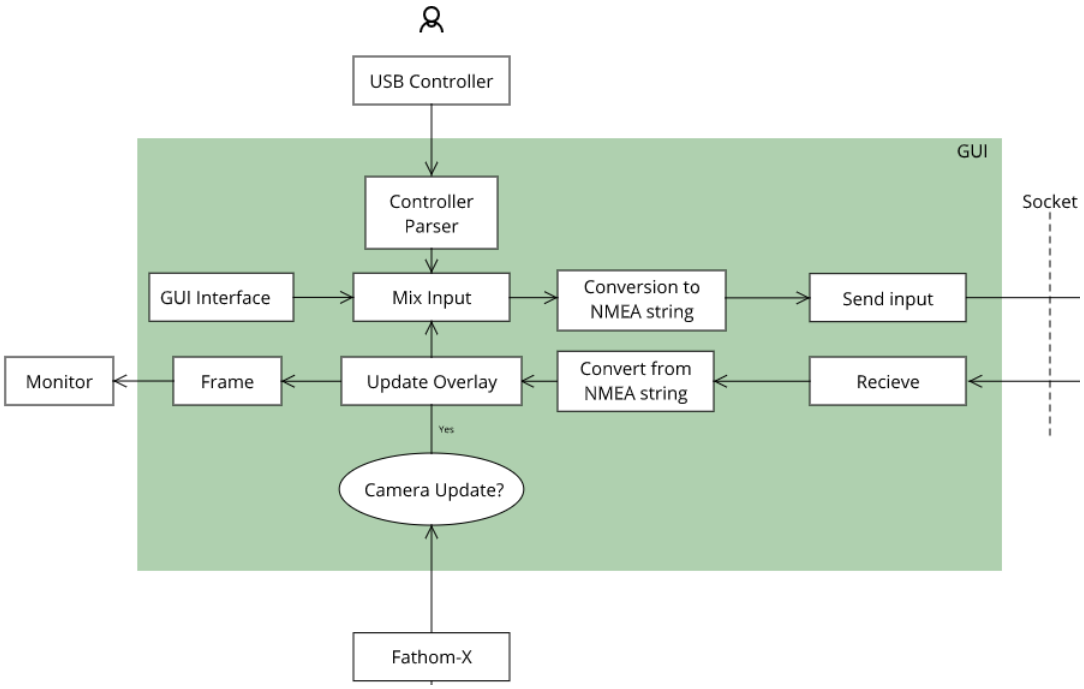


Figure 3.1: Advanced GUI Diagram.

The GUI software has three main inputs: user input, the sensor data string, and video feed. From these inputs, the GUI outputs two things: a control string and an updated display. The following section will explain in detail the flow

of our software and how the outputs are generated given the inputs.

3.2 GUI Code Flow

This section will go in-depth into the GUI software processes. The main source file for the GUI is shown in Appendix B.1. The GUI is primarily concerned with gathering user input. This input can be received in two ways: from the buttons on the GUI itself, or external peripherals such as a gamepad. In our implementation, we created a parser that can interpret input from a Playstation 4 gaming controller, and formats that data for use in a generation process for our outputs. This data is in the form of a simple class containing different values for each of the tokens in our control string. The control string and its tokens are covered in detail in Section 4.4. This class is sent to the NMEA string generator, which wraps all of the class values into a string, defined by the protocol in Section 4.4. After wrapping the values into a single control string, the GUI system sends it to the communications subsystem over a socket—which is explained more in depth in Section 4.8.

The second input that the GUI can receive is the sensor data string. This string is initiated and sent from the subsea subsystem and is parsed using the protocol described in Section 4.4. This data is used to update the GUI display to show the current readings on the ROV sensors.

Lastly, the camera feed input is fed directly into the GUI, and is displayed using OpenCV, a Python package which can display images.

3.3 Implementation

We built our GUI using Python's de-facto standard package for making GUIs: Tkinter. Tkinter provides a thin, object oriented layer used to create different frames and widgets to make up a GUI. Tkinter defines frames as a rectangular region on the screen, and uses them to group one or more widgets together. This way, widgets are more easily organized, moved, and managed. Widgets are the different tools and elements that Tkinter can use to receive input from a user or display data. For example, a standard input widget is a button, and an output widget would be a text label. In general, we used buttons for our input widgets to control different components on the ROV, such as the thrusters, lights, and camera tilt mechanisms, and used graphs as our output widgets that would show all of our sensor data read from the subsea system.

We used these input and output widgets and grouped them by functionality within separate frames to build a well-designed, minimal, and easy-to-use Graphical User Interface for our users. We explain our organization and layout in detail in the following section.

3.4 Layout

A crucial characteristic of a well-designed GUI is its ability to be operated by a user with minimal experience. Because of this, we designed the layout of our GUI to be simple and minimal, contrary to a noisy and complex interface.

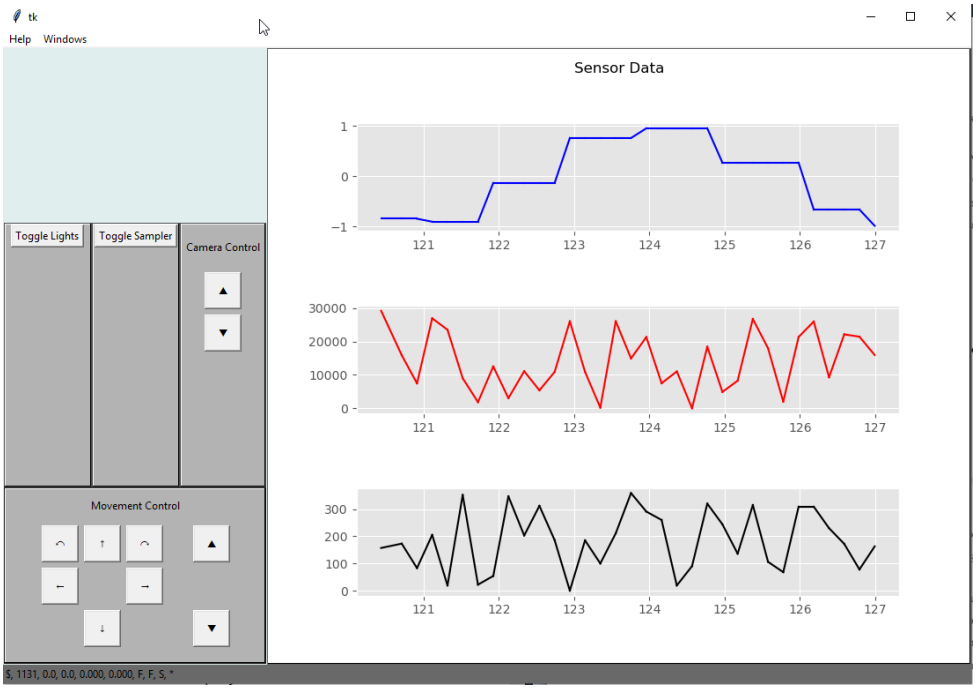


Figure 3.2: Neptune Graphical User Interface (GUI).

The following GUI subsections will cover the various parts, called frames, of the GUI and what each are used for. Each subsection will use Figure 3.1 as a reference.

3.4.1 Movement Control

The movement control frame handles all of the translational, rotational, and vertical movement of the ROV. Each of the buttons activate corresponding thrusters on the ROV that will move it in the way that the buttons denote. On the left side of the frame, the straight arrows in the north, east, south, and west directions will engage the thrusters to move the ROV forward, right, backwards, and left, respectively. The remaining curved arrows will rotate the ROV in the direction specified on the button.

The buttons on the right side of the frame, denoted with upward and downward pointing triangles affect the depth of the ROV by engaging its vertical facing thrusters.

3.4.2 Light Control

This frame simply contains a button to toggle the subsea lights on and off.

3.4.3 Sampler Control

This frame is also another simple frame containing a button to toggle the SMA water sampler* .

3.4.4 Camera Control

This frame provides buttons to control the camera servo. The up and down buttons will tilt the servo up or down, respectively. This allows for a larger view to be seen in the video feed, as well as the perspective to be changed without having the maneuver the ROV.

3.4.5 Sensor Reading Graphs

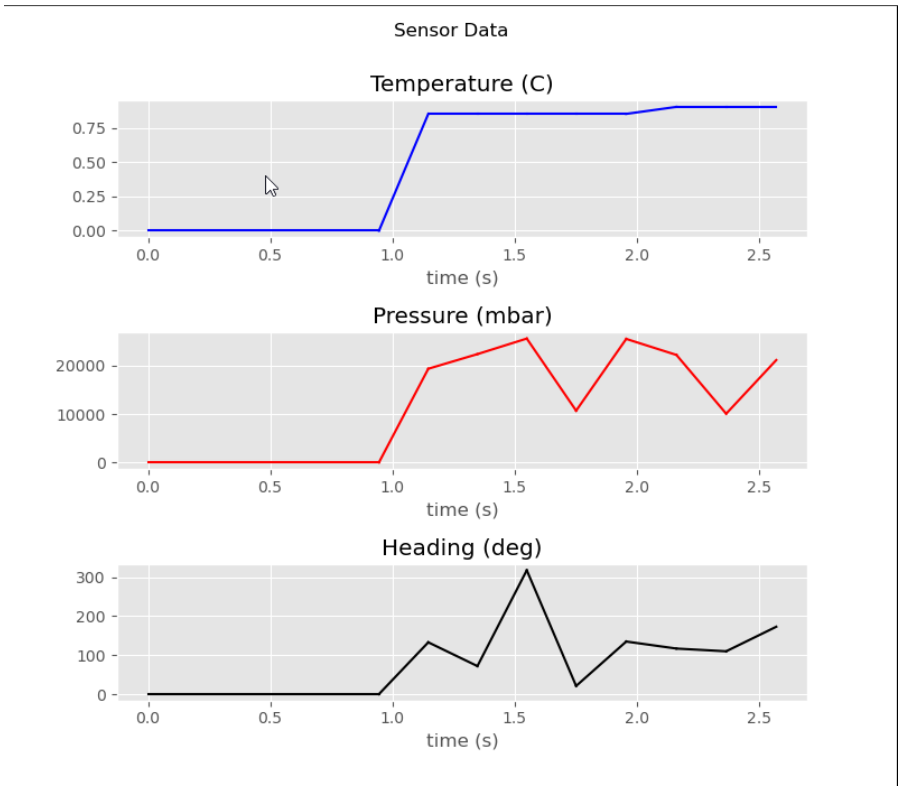


Figure 3.3: GUI sensor graphs.

This frame makes up the largest portion of the GUI. It displays the sensor readings that are reported back from the ROV in a graphical view. It displays the temperature (C), pressure (mbar), and heading (deg).

3.4.6 Viewing Video Feed

The video feed is not immediately shown in the main menu of the GUI. Rather, the option is in the top menu bar to open a new window that shows the video feed. This option can be selected multiple times to spawn multiple feeds.

We chose this implementation of the video feed being in a separate window rather than completely integrating it into the GUI for multiple reasons. The first reason being that completely integrating it into the GUI would tightly couple the video feed into the GUI. This would hinder the GUI from being upgraded, or even replaced. Also, since the video feed window is separate from the GUI, the video feed window can be physically moved on the monitor, or onto another connected display. This would allow a spectator to observe the video feed separately, while the pilot can operate the ROV from the GUI on a different screen. The ability to spawn multiple camera feeds also solves the problem of allowing the pilot to see the camera feed during operation on one display, with observers watching another window of the camera feed on a different display.

3.4.7 Status Bar

The status bar is located on the bottom of the GUI. Although we have not implemented any status update messages yet, the status bar is still available to use for displaying any message. Currently, we are simply displaying the control string that is generated from the user input. This process will be explained in Section 3.6.1, Control String Generation.

3.5 Gamepad Input

Through testing, we realized that clicking buttons would not be the best fit for hands-on operation of the ROV. We decided to implement a way to use a Playstation 4 (PS4) controller for more streamlined operation.

We implemented this by using the PyGame library alongside a Python interpreter program, which bound different ROV controls to buttons on the PS4 controller. This system is also entirely modular by using a controller keybinding file. By changing the keybindings, a user is able to modify which buttons correspond to what instructions to send to the ROV.

3.6 Communications Interface

Once the GUI works as a whole, it is important that the processes done in the GUI space send and receive data to and from the other subsystems correctly. To do this, data must be formatted consistently so that the receiving end of the communication can interpret it properly. This formatting process is covered in detail in Section 4.4, but the following subsections will provide a general overview of the process within the GUI system.

Both the GUI and Arduino systems must be capable of generating and parsing control strings, while the communications system is responsible for passing these strings between the two systems.

3.6.1 Control String Generation

String generation in the GUI system takes in seven main signals: direction, magnitude, vertical, rotation, light toggle, sampler toggle, and camera tilt. These values are packed into a string with the protocol described in Section 4.4.

3.6.2 Sensor String Parsing

The response from the subsea Arduino subsystem contains all of the sensor values read on-board the ROV. The sensor values are for the temperature (C), pressure (mbar), and heading (deg) and are then displayed on the corresponding graph in GUI.

Chapter 4

Communications

4.1 Software Diagram: Communications

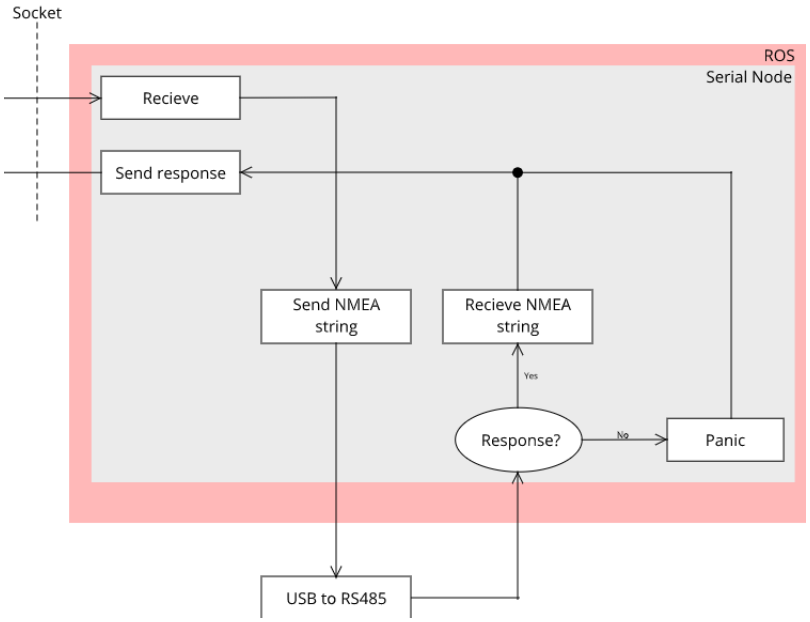


Figure 4.1: Advanced ROS Diagram.

Our ROS implementation revolves around two set of inputs and two set of outputs. As can be seen in Figure 4.1, we have an input from a socket which connects ROS to our GUI. This input is the control string which is generated by the GUI using the user's input. That input is then sent down to the first output which is our tether connection to the Arduino. Our second input is a data string that is received from the Arduino over the same connection and it is processed for accuracy and validity. Once this is done, it is sent to our second output, the socket to our GUI where the data string is then outputted to the user. This is our control loop for the ROV and will continue to do this until either a catastrophic error occurs or the ROV is shut down.

4.2 Protocol

Since we are transmitting over a five-hundred foot long tether, regular communication protocols, such as Ethernet or USB, were insufficient in terms of reliability and efficiency. We consulted with more experienced members of the RSL on what protocols would hold up to our reliability requirements. At some point we were told about a protocol called RS485. RS485 is a standard protocol used in serial communications used commonly in control systems and industrial applications due to its reliability. RS485 is particularly good at noise prevention and as such is highly reliable over long distances, which we took great interest in. This protocol also works over only one twisted pair, leaving three twisted pairs to spare for other systems and future planned expansions.

Normal computers and the Arduino we used for our system do not come with built in ways of utilizing RS485 so in order to implement this, we required a converter. For our top-side system a DSD-TECH SH-U10 was used to convert from USB to RS485. For our bottom-side system a DFROBOT RS485 Shield was purchased and installed onto our Arduino. Connecting the A and B wires on the SH-U10 to the RS485 Shield results in proper communications between the two systems and has been tested to be nearly perfect in terms of reliability on our system.

4.3 Master-Slave Comms

Since RS485 works over serial communications, special care must be taken to prevent bit collision issues. This happens when both sides of the RS485 network transmit at the same time and thus the signals interfere and jumble the signals. This issue is especially prevalent in debugging any sub-sea communications issues, where the main method of debugging is printing to the serial output of the Arduino and will be discussed later.

To resolve these collision issues, we decided on using a master-slave communication process where our top-side system is the master and the sub-sea system is the slave. To start, our master first starts off by sending a NMEA string down to the slave, while at the start the slave is constantly listening for a string to be sent. Once the slave receives the string it immediately interprets it, issues component commands dictated by the received string, and collects sensor data. Once that has been completed, a new NMEA string is generated using the sensor data and is then sent back up to the master. While this is taking place, the master has switched into a listening mode, constantly waiting for the slave's response. Once the master receives the response from the slave containing the sensor data, it then interprets and sends the data off to the GUI. This process is then repeated by having the master issue a new command while the slave listens. This loop happens constantly, unless an error occurs, in which case the side which receives the erroneous message starts its error handling procedure.

4.4 NMEA String Protocol

The National Marine Electronics Association, or NMEA String Protocol is a standard in data transmission, commonly used in GPS devices. In our implementation, we modified the protocol to better fit our needs. The string has two unique formats for the different directions to send the string: from topside to subsea and from subsea to topside. Each are formatted differently because of their contents, as the topside to subsea string contains instructions to control the ROV, while the subsea to topside string contains the values read by the ROV sensors. The following subsections will go into detail on how each of these strings are formatted.

4.4.1 Topside to Subsea String

Below is an example of a NMEA string generated by the GUI from user input, as well as a table that explains each of the values within the string.

$, 1991, 0.432, 32.021, -0.496, 0.830, F, T, U, *

token number	0	1	2	3	4	5	6	7	8	9
data	$	1991	0.432	32.021	-0.496	0.830	F	T	U	*
meaning	start of message	id number	joystick magnitude	joystick angle	vertical	rotation	light toggle	sampler toggle	camera tilt	end of message
possible values	$	[0, 4294967295]	[0, 1]	[0, 360]	[-1, 1]	[-1, 1]	T/F	T/F	U/D/S	*

Figure 4.2: Topside to Subsea String Example and Value Table.

Values in the string are comma-separated. Each value within the string is called a token, with the zeroth token being the start of message token. Tokens are numbered 0-9 because our implementation relies on a sequential parsing technique, where values are read in order starting from the zeroth token. The receiving end trusts that the transmitting end will pack values in the correct order, so that when they are read, the corresponding values are in the correct position. This takes away the need for headers for each of the tokens, since the receiving end will already know the meaning of each value by its token number, which in turn, reduces the amount of overhead for each string transmitted. The following subsections will explain each token in detail and how they are used on the receiving end.

Token 0: Start of Message

The start of message token contains the start message character. In our implementation, we chose to use the '$' character, similar to the official NMEA standard. This start of message token allows the receiving end to know where the transmission data begins. Without this character, if there is any data loss, the receiving end might mistake any of

the other tokens as the zeroth token, and begin reading data in the wrong sequence. This is important, as our control string values rely on reading the data sequentially, and must always start from the start of message character.

Token 1: ID number

The ID number is used to define how many messages have been sent during the operation period. It is also used in the Master-Slave architecture explained in Section 4.3. Since the master first sends a control string to the slave and then listens for the response, there must be a way to ensure that the slave's response is generated from the master's initial string. In our implementation, we used the ID number to do this check. The ID number that the master issues to the initial control string is the same one used that the slave uses to respond. If the ID numbers do not match, an error has occurred and is reported.

Token 2: Joystick Magnitude

The joystick magnitude is used alongside the joystick angle to calculate a vector that describes the lateral movement of the ROV. More specifically, it describes a combination of forward, backward, left, and right movement along the X and Y axis. This magnitude is a float value with 3 digits of precision, in the range [0, 1]. This value is determined by how far away the joystick is from its resting position. The calculation to translate these values is covered in Section **??**, Thruster Mapping.

Token 3: Joystick Angle

The joystick angle is used alongside the joystick magnitude to calculate a vector that describes the lateral movement of the ROV. More specifically, it describes a combination of forward, backward, left, and right movement along the X and Y axis. This angle is a float value with 3 digits of precision, in the range [0, 360]. This value is determined by the angle that the joystick makes with the east direction on the joystick, similar to a unit circle. The calculation to translate these values is covered in Section **??**, Thruster Mapping.

Token 4: Vertical

The vertical token describes how to engage the vertical thrusters, with a positive value meaning to move upwards, and negative to move down along the Z axis. This value is a float value with 3 digits of precision, in the range [-1, 1].

Token 5: Rotation

The rotation token describes how to engage the manuevering thrusters to result in a rotation about the Z axis, without affecting any translational movement along the X and Y axes. This value is a float value with 3 digits of precision, in the range [-1, 1], with the positive values meaning to rotate clockwise, and negative to go counter-clockwise.

Token 6: Light Toggle

The light toggle token is used to signal the ROV's lights to turn on and off. The state of the lights is not passed between the GUI and Arduino systems, rather it is assumed to stay synchronized. There are only two values that this token can have, 'T' for True, and 'F' for False. Upon receiving any True signal, the receiving side will toggle the lights, whether that will turn it on or off. A False signal will cause nothing to happen.

Token 7: Sampler Toggle

The sampler toggle token is used to signal the ROV to use a sampler charge. The state or count of the sampler is not passed between the GUI and Arduino systems, rather they are assumed to stay synchronized. There are only two values that this token can have, 'T' for True, and 'F' for False. Upon receiving any True signal, the receiving side will toggle the sampler. A False signal will cause nothing to happen.

Token 8: Camera Tilt

The camera tilt token is used to control a servo that the onboard camera is mounted on. The servo position is not passed between the GUI and Arduino system, rather that position is calculated onboard the Arduino based on the direction defined in this token. The values 'U', 'D', and 'S' stand for up, down, and stay, respectively.

Token 9: End of Message

The start of message token contains the end message character. In our implementation, we chose to use the '*' character. This end of message token serves a similar purpose to the start of message token by allowint the receiving end to know where the transmission data ends. This allows the receiver to end the parsing process and begin generating the response.

4.4.2 Subsea to Topside String

Below is an example of a NMEA string generated by the Arduino from the sensor readings, as well as a table that explains each of the values within the string.

$, 1991, 23.76, 1264.95, 184.03, *

token number	0	1	2	3	4	5
data	$	1991	23.76	1264.95	184.03	*
meaning	start of message	id number	temperature (C)	pressure (mbar)	heading (deg)	end of message
possible values	$	[0, 4294967295]	-40 < t < 125	0 < p < 30000	0 < x < 360	*

Figure 4.3: Subsea to Topside String Example and Value Table.

Similar to the Topside to Subsea string, the values are comma-separated. Since the Arduino only needs to pack the data read from the sensors, there are only 6 total tokens, with only 3 coming from the sensors. The following subsections will explain each token in detail and how they are used on the receiving end.

Token 0: Start of Message

The start of message character is the same as the Topside to Subsea string, '$' and serves the same purpose: to define the beginning of the message for easier parsing.

Token 1: ID Number

The ID number that the Arduino side generates should be the same as the ID number that the Topside to Subsea string had. This allows us to check whether the response is specifically from the initial command, as well as if it came back on time.

Token 2: Temperature Reading

The temperature reading is a float value with 2 digits of precision that is returned by the Blue Robotics Celsius Temperature Sensor. As it is named, the value is measured in Celsius. The range that this value can be is defined by the operating conditions of the sensor, [-40, 125].

Token 3: Pressure Reading

The pressure reading is a float value with 2 digits of precision that is returned by the Blue Robotics Bar30 Pressure Sensor. The measurement of the sensor is in millibar, or mbar. The range that this value can be is defined by the operating conditions of the sensor, [0, 30000].

Token 4: Heading

The heading is a float value with 2 digits of precision that is returned by the Adafruit LSM303 Mag Sensor. The measurement of the sensor is in degrees, which range from [0, 360]. A 0 or 360 degree value corresponds to magnetic north.

Token 5: End of Message

The end of message character serves the same purpose as the end of message character in the Topside to Subsea string. It allows for easier parsing of the control string, should any data loss occur.

4.5 ROS Implementation

When we were told about NMEA Strings and their usefulness to our project, we looked into seeing if there were any preexisting ROS nodes implementing NMEA Strings. Luckily, we came across a node which seemed to fit our requirements called nmea-comms. This package is certainly difficult to understand and some of the design choices made in it are questionable, but it can be broken down as follows. Two nodes are spawned upon launching the node, one for receiving commands over serial which it then passes over a socket to the other node which interprets and responds to the string.

Out of the box, this node did not work with our setup for various reasons. Several changes needed to be altered in order for this node to transmit and receive messages such as modifying the polling and reading settings and integrating a socket to receive messages from the GUI. In order to prevent reading garbage and prevent segmentation faults, which would randomly happen with the original system, a decision was made to limit the response string to somewhere near 20 bytes which drastically improved performance of our system. But perhaps the biggest change was changing this system to a one node setup. We constantly ran into trouble getting the two node setup to function and start-up reliably so we decided to depart from the original framework and create a new node which encompasses both of the original nodes as well as new aspects which better suited our design. This new node handles everything; it is responsible for sending and receiving over the RS485 connection, for interpreting the strings received, error handling, socket connections, and maintaining the serial connection.

Our new system is nearly perfectly reliable and any error which may come up is resolved with new error handling. In total, the process of sending and receiving a message takes approximately 60ms which is significantly better than our functional requirement. It also shorter than our camera FPS (approximately 10FPS, which translates to 100ms per frame) which means the user will not feel any delay from their input since the camera will not be updated by the time the message is sent and an acknowledgement is received. Theoretically, this 60ms loop is significantly longer than what it should be. Since we are using 9600 as our baud rate, it should only take approximately 16ms to send and 16ms

to receive the messages, totalling to 32ms for the whole loop. Somewhere in our process we are introducing a 28ms overhead which we were unsuccessful in pinning down. We believe that this may be a hardware overhead due to the conversions needed for obtaining RS485 connection capabilities, but cannot substantiate this claim. Nevertheless, we were not worried about this overhead as even at 60ms, the user will not be able to discern a difference in input lag due to the camera feed limitations.

4.6 Communications Code Flow

The ROS system we have implemented is significantly complex and as such we will describe how it works here. Our ROS node implementation is written in C++ and starts off as any normal C++ program would, with **main()**.

4.6.1 main()

main() does not particularly do a whole lot. It is mainly responsible for initiating values and data associated with the ROS node. After all of that is initialized it calls **manage_connection()**.

4.6.2 manage_connection()

manage_connection(). is the main function of this entire node. Ideally this function will run forever, given that no catastrophic errors occur. This function first starts with creating the socket to communicate with the GUI. Once this socket has been created and stored, we then go into creating the file descriptor which we will use to communicate with our RS485 communication link. Once that file descriptor has been created and properly set, we enter the main loop of this whole node. In essence the following loop represent the master-slave design we discussed earlier. The loop first starts with obtaining a message via **getNewSentence()**.

4.6.3 getNewSentence()

getNewSentence() is responsible for using the previously established socket to obtain a control sentence from the user's inputs in the GUI. Once this has been completed this function returns back to the loop in **manage_connection()**. Since we now have a sentence to transmit, we begin the master-slave design with transmitting the sentence over our RS485 connection to the Arduino using **tx_func()**.

4.6.4 tx_func()

tx_func() takes in a sentence and transmits it over the RS485 connection using the previously mentioned file descriptor. There are several nuanced things that happen here before anything is actually transmitted which mainly revolve around polling the serial connection to block the file descriptor so that nothing can transmit while we are transmitting our message. Once that has been done, we transmit the message for the Arduino to receive. Once we know that it has been successfully transmitted, we return to loop in **manage_connection()**. Now that the master has sent its control sentence to the slave, we expect that it will execute the control sentence and in turn send a response sentence back. This response will be captured by **rx_func()**.

4.6.5 rx_func()

tx_func() continually loops until a return sentence is found. Like **tx_func()** there are several nuanced things that happen before we read from the connection which again mainly revolve around polling the connection. We poll for a total of 60ms which takes up the bulk of the communication response time. As stated previously it is not entirely known why this is, but any time less than 60ms has significant reliability issues. After the connection has been polled we then read from it and store that into a buffer. The buffer is then checked for errors by making sure that the sentence starts and ends with the designated tokens. If an error is detected we transmit back down to the Arduino asking to repeat the sentence and **tx_func()** starts back over. If it is detected that the response sentence does not contain errors, we send it back using **sendGUISentence()**.

4.6.6 sendGUISentence()

sendGUISentence() simply takes in the response string and sends it to the GUI over the previously established socket. This function marks the end of the master-slave loop. As such the loop will then restart by going to **getNewSentence()** and will continue forever given that no errors occur. For actual code of the above functions please see appendix B.2.

4.7 Baud Rate

Since we are communicating over serial, we must choose a baud rate in accordance with our design requirements. Since reliability was our most important factor, we prioritized that, but we also considered speed as well. In the end, we chose a baud rate of 9600bps, which is a good compromise between reliability and speed. As can be seen in Figure 4.1 in the 16.0MHz column (clock speed of the Arduino Mega), provides excellent reliability on top of decent speed.

Baud Rate (bps)	f_{osc} = 16.0000 MHz				f_{osc} = 18.4320 MHz				f_{osc} = 20.0000 MHz			
	U2Xn = 0		U2Xn = 1		U2Xn = 0		U2Xn = 1		U2Xn = 0		U2Xn = 1	
	UBRRn	Error	UBRRn	Error	UBRRn	Error	UBRRn	Error	UBRRn	Error	UBRRn	Error
2400	416	-0.1%	832	0.0%	479	0.0%	959	0.0%	520	0.0%	1041	0.0%
4800	207	0.2%	416	-0.1%	239	0.0%	479	0.0%	259	0.2%	520	0.0%
9600	103	0.2%	207	0.2%	119	0.0%	239	0.0%	129	0.2%	259	0.2%
14.4k	68	0.6%	138	-0.1%	79	0.0%	159	0.0%	86	-0.2%	173	-0.2%
19.2k	51	0.2%	103	0.2%	59	0.0%	119	0.0%	64	0.2%	129	0.2%
28.8k	34	-0.8%	68	0.6%	39	0.0%	79	0.0%	42	0.9%	86	-0.2%
38.4k	25	0.2%	51	0.2%	29	0.0%	59	0.0%	32	-1.4%	64	0.2%
57.6k	16	2.1%	34	-0.8%	19	0.0%	39	0.0%	21	-1.4%	42	0.9%
76.8k	12	0.2%	25	0.2%	14	0.0%	29	0.0%	15	1.7%	32	-1.4%
115.2k	8	-3.5%	16	2.1%	9	0.0%	19	0.0%	10	-1.4%	21	-1.4%
230.4k	3	8.5%	8	-3.5%	4	0.0%	9	0.0%	4	8.5%	10	-1.4%
250k	3	0.0%	7	0.0%	4	-7.8%	8	2.4%	4	0.0%	9	0.0%
0.5M	1	0.0%	3	0.0%	–	–	4	-7.8%	–	–	4	0.0%
1M	0	0.0%	1	0.0%	–	–	–	–	–	–	–	–
Max. (1)	1 Mbps		2 Mbps		1.152 Mbps		2.304 Mbps		1.25 Mbps		2.5 Mbps	

1. UBRRn = 0, Error = 0.0%

Figure 4.4: Baud Rates and Their Associated Errors [15].

4.8 GUI Implementation

The GUI communicates with ROS over a socket connection (see Figure 2.6) to send and receive NMEA strings. The socket is implemented over TCP with the ROS Serial Node acting as the server and the GUI as the client. Although a major system requirement was response time and UDP provides a faster connection, TCP uses a handshake protocol making the system reliably operable. Upon successful connection the GUI sends and receives NMEA strings with ROS over this socket.

4.9 Arduino Implementation

Luckily for us, Arduino is natively more compatible with serial than a standard Linux distribution so creating a way to read NMEA strings here is significantly easier. Our basic system revolves around a buffer which bytes from the RS485 link are read into. Once the final character is received, designated by ”*”, the string is complete and is then sent off for processing, which will be discussed in the next chapter. This buffer is especially useful for debugging, as once the NMEA string is complete or the buffer is filled, it can be sent back to the serial for inspection on the Arduino Serial Monitor for further debugging. This is in contrast to immediately outputting the received character which may first come to mind, but will constantly result in bit collisions.

Chapter 5

On-board Control

This chapter covers the third of the four main subsystems: the on-board control. As previously mentioned in our technologies used and design rationale, we used a Arduino Mega 2560 Rev3 microcontroller to control the subsea hardware.

5.1 Software Diagram: Arduino

Below is the subsea section of the Advanced Software Diagram cropped from Figure 2.6. It contains the camera and the Arduino used for on-board control of the hardware components.

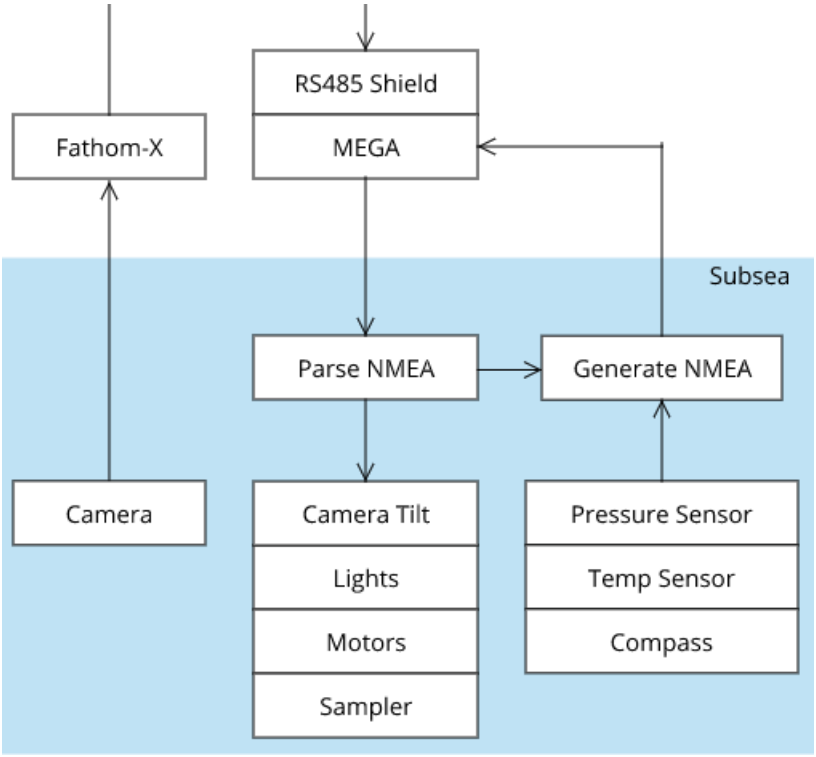


Figure 5.1: Advanced Arduino Diagram.

5.2 Arduino Code Flow

As the above diagram indicates, our Arduino Mega system uses one input and has two separate outputs. To encompass all the various functionality the Arduino provides, we developed an Arduino master file that includes all sensor libraries used to receive sensor data. The input to the file is the control string sent from the ROS node (detailed in Section 4.4.1). Because this string is a list of comma separated tokens that represent values, it was easy to parse to determine the value of a input NMEA string. The parser determines these values which correspond to the component commands. The arduino directs these individual commands to the correct component (lights, sampler, thruster, etc.). The thruster commands will be described in detail in Section 5.4.3. As detailed in Chapter 4, the Arduino is the slave in the communication protocl and thus must respond to the input string with a response NMEA string (detailed in Section 4.4.2) that contains the updated sensor data. After the component commands have been issued, the master file updates data from the pressure, temperature, and compass sensor using library read functions and wraps them into a single response NMEA string. This generated NMEA string is sent back over the tether to the ROS node, which is sent over a socket to the GUI to update the display. Separate from all of this, the camera is constantly outputting over the tether to the GUI in a process which will be described in Chapter 6.

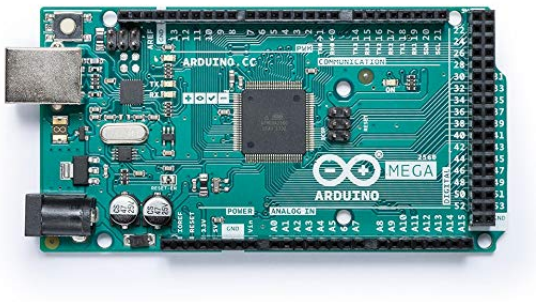


Figure 5.2: Arduino Mega 2560 Rev3[16].

5.3 Wiring Diagram

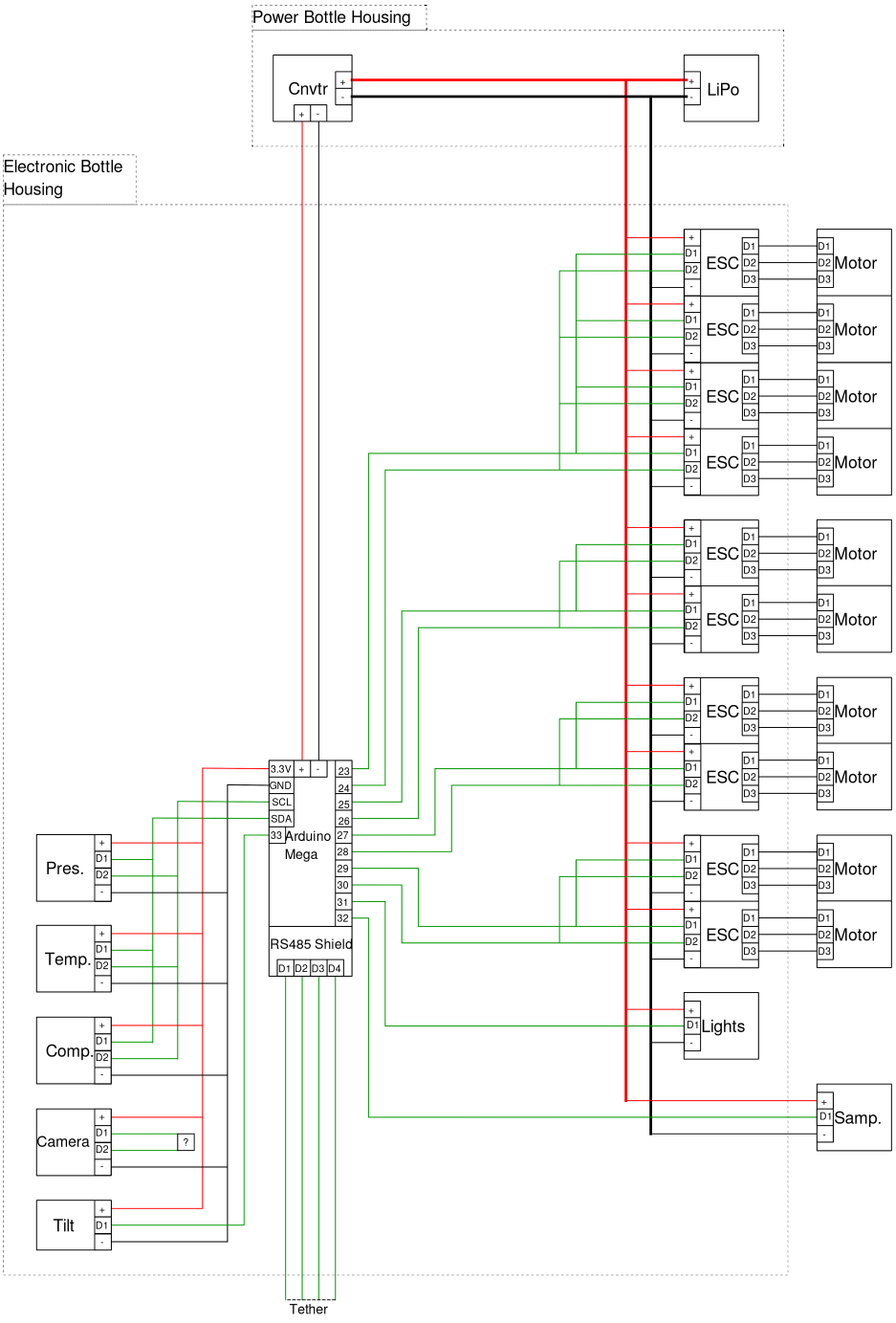


Figure 5.3: Wiring Diagram.

Our ROV has two sealed bottles which house our electrical components. The first bottle contains all of our power supply components, including all of our LiPo batteries. This is done to isolate our system components in case of LiPo battery malfunction and subsequent fire. Included with our batteries we have a power control system which will take

the batteries power and step it down to voltages which our components can handle. We have two sets of voltages: a high voltage rail which powers our lights and thrusters, since they are high voltage components, and a low voltage rail which powers our control components. In the second sealed bottle we have all of the control and measurement systems. Our On-board Control system is centered around an Arduino Mega 2560 which will be connected to all of our retrofit components. Most of our components connect to the Arduino via I2C protocol which most of our components support. The camera is not connected to the Arduino and ties directly into the tether. Our topside system has a dedicated monitor which this feed will directly connect to. The Arduino takes the input from the instruments and process them for messaging back to topside through our tether. We connect to the tether via the RS485 protocol which is utilized via an Arduino shield. For our thrusters, they are wired together into stacks which all contribute to a specific movement. We plan on having four stacks of thrusters, each connected to an electronic speed controller which is connected to our Arduino and will provide an interface to control the thrusters.

5.4 Components

This section outlines the implementation of the various retrofit components for the ROV. This includes sensor implementation, motor functionality, and thruster mapping.

5.4.1 Sensor Implementation

(a) Blue Robotics Bar30 pressure sensor[11]

(b) Blue Robotics Celsius temperature sensor[12]

Figure 5.4: Blue Robotics Sensors.

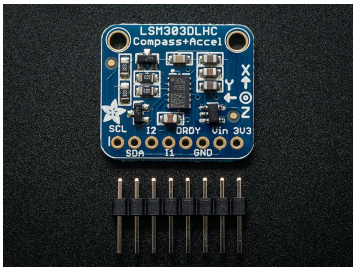
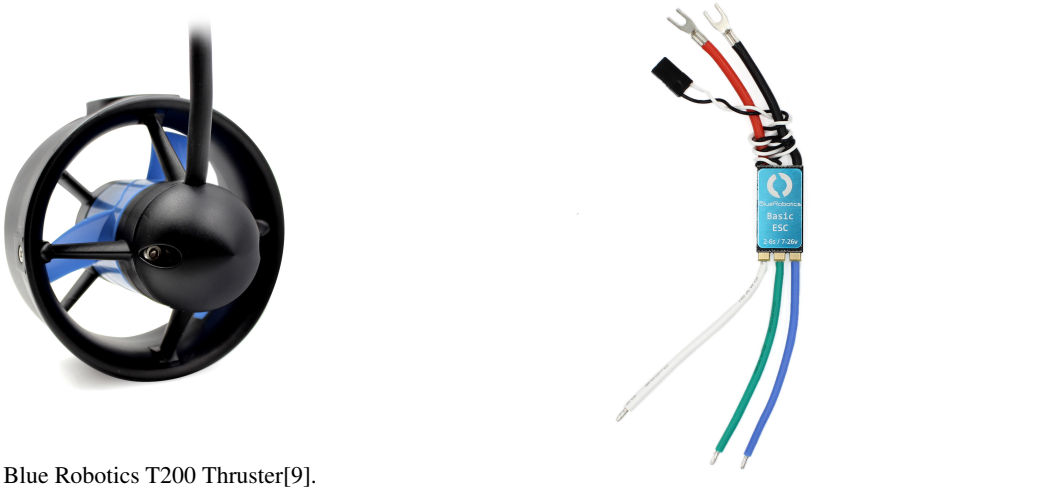


Figure 5.5: Adafruit LSM303 IMU sensor[13].

Pictured above are the three subsea sensors that produce data used in the NMEA string that is sent to the topside user. The Bar30 sensor (Figure 5.3a) provides a millibar pressure read, the Celsius sensor (Figure 5.3b) provides a Celsius temperature read, and the IMU sensor (Figure 5.4) provides a degree (heading) read. These sensors all use I2C protocol to connect to the Arduino. Luckily, as mentionted, all three sensors have existing libraries with Arduino that allowed for easy implementation and functionality testing.

5.4.2 Motor Controlling



(a) Blue Robotics T200 Thruster[9].

(b) Blue Robotics Electronic Speed Controller (ESC)[10]

Figure 5.6: Blue Robotics Thruster and ESC

The mechanical engineering team has retrofit the ROV with ten thrusters (Figure 5.5a) mounted in vertical, horizontal, and angled positions. The thrusters are hooked up to electronic speed controllers (ESCs)(Figure 5.5b) to control the individual PWM value for each thruster—more information on this in the next section. We used LiPo batteries to power the thrusters and a built in Arduino Servo library for functionality testing.

5.4.3 Thruster Mapping

In order to correctly control the ROV's movements with the thrusters, a certain calculation must be done. This calculation takes in the desired movement, and outputs instructions to corresponding thrusters that will fulfill the movement specified. This process is referred to as Thruster Mapping.

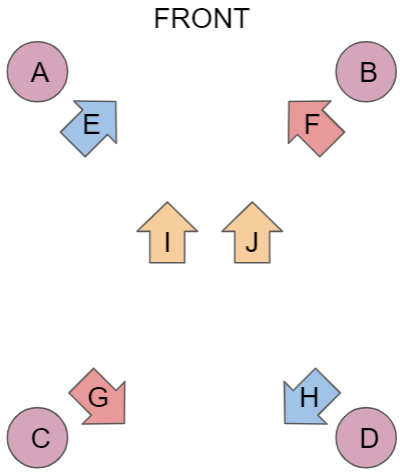


Figure 5.7: Thruster Layout.

Above is a diagram with a clear view of the placement of each thruster onboard the ROV. Each thruster is given a letter A-J as a name. The angle and position of each thruster will determine its role in generating movement. Our movements are categorized by four unique types: moving along the X axis, moving along the Y axis, moving along the Z axis, and rotating on the Z axis. These four types of movements describe how the ROV will be able to move underwater, with the Z axis movement corresponding to depth. With these four types of movements, we categorize our thrusters in only three types: vertical, maneuvering, and forward thrusters. This is done to support a minimal thruster control design. By grouping twelve thrusters into groups, we allow the possibility of controlling multiple thrusters in the same group with a single controller. This will be later explained in the upcoming sections. Also, although the number of types of movements and thruster types are unmatched, the reasoning becomes clear in our implementation of how we divided the responsibilities. The responsibilities in relation to the four movement types are described in the following sections.

Furthermore, each section will provide a description of the mathematical process for each type of thruster. As described in the NMEA string protocol in Section 4.4, the Arduino receives four different values that describe the desired movement specified by the pilot. These values are:

- θ - joystick magnitude (token 2)
- m - joystick angle (token 3)
- v - vertical (token 4)
- r - rotation (token 5)

These variables will be used in the coming equations to return a percentage that will describe what percent of a thrusters power it should use. These values will be put into the following matrix:

$$T = \begin{bmatrix} T_A \\ T_B \\ T_C \\ T_D \\ T_E \\ T_F \\ T_G \\ T_H \\ T_I \\ T_J \end{bmatrix}$$

Figure 5.8: Thruster Matrix.

This matrix will be referred to by the Arduino to retrieve the result of the thruster mapping calculation. For example, the T_a value will be referred to when controlling the A thruster.

Vertical Thrusters

The four vertical thrusters, A, B, C, and D, colored pink, take the sole responsibility of controlling the vertical movement of the ROV. Specifically, it controls the movement along the Z axis. The vertical thrusters do not play any part in the other types of movement, which is clear by its placement in Figure 5.7. The vertical thrusters are placed in the four corners of the ROV, in parallel with the Z axis. In order to get a purely Z-axis parallel movement, we can assume that all four thrusters will need to output the same amount of thrust simultaneously– this is one of the major benefits of this design. By grouping these four thrusters together under a single speed controller, we are able to conserve output pins on our microcontroller. This is in contrast to assigning a speed controller to each of the four thrusters, and individually sending the same value to all four– essentially wasting the output pins of our microcontroller.

The equation for the vertical value is very straightforward. Since the bounds of the vertical *v* instruction (token 4) is already [-1, 1], it can directly be used as a percentage to describe the thrust needed for the vertical thrusters. This vertical matrix is described in 5.9.

$$V = \begin{bmatrix} v \\ v \\ v \\ v \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Figure 5.9: Vertical Matrix.

The *v* value is directly taken from the NMEA string and assigned to the four vertical thrusters A-D. Thrusters E-J do not receive any value during the vertical calculation.

Maneuvering Thrusters

The four maneuvering thrusters, named E, F, G and H, and colored red or blue are placed in 45 degree angles along the inside corners of the ROV frame. The forward direction of these thrusters are denoted by the direction the arrows are pointing in. The forward direction of these thrusters have an extremely important purpose, explained in following paragraphs. However, it is important to note that the symmetry of the thruster directions will play in important role in generating even thrust. These maneuvering thrusters actually take the responsibility of two movement types: moving along the X axis, as well as rotating along the Z axis. Both may not be clear at first, so we will first explain how this layout supports movement along the X axis. Firstly, within the maneuvering thrusters are thruster pairs. Thrusters are paired by which side they are on. Namely, thrusters E and G are paired, and thruster F and H are paired. By pairing these thrusters, we use each pair simultaneously to generate thrust left or right. For example, equally engaging the left thruster pair, E and G forward will result in purely X-axis parallel movement towards the right. Then, the other pair of thrusters– thrusters F and H, are engaged backwards to generate thrust towards the right as well. Even though the thrusters are angled, since they are symmetric, the resultant Y vectors should cancel each other out between thruster pairs. This effect is shown in Figure 5.10 below. The left side shows all of the X and Y component vectors for each thruster, while the right side is the simplified version, where any opposite vectors are cancelled out and removed.

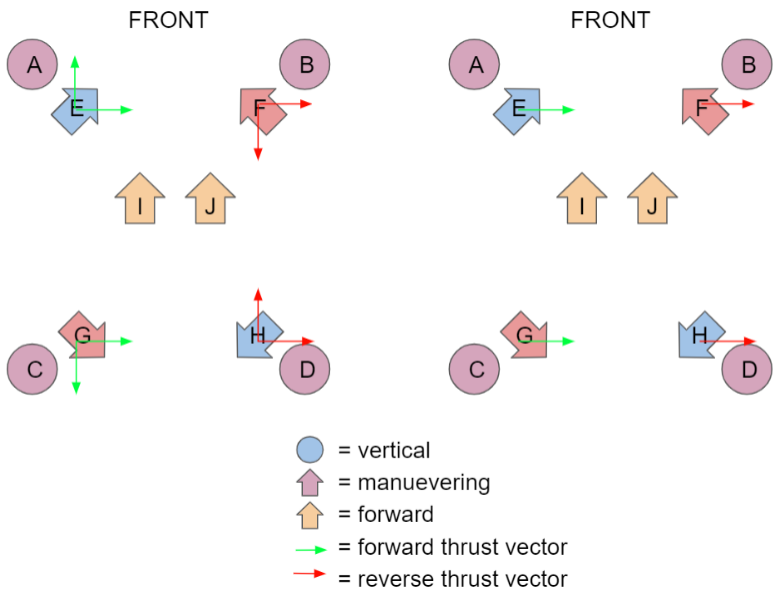


Figure 5.10: Thruster vectors when moving right, alongside vector cancellations.

This effect relies on the assumption that the Y-axis parallel forces cancel out exactly, which will result in purely X-axis movement. For now, we assume that a vector generated from forward thrust, will cancel out with another forward facing thrust, and not a backwards thrust. In Figure 5.10, it is clear to see that between each pair, the Y-component vector is of the same type, and should therefore cancel out. However, this assumption cannot be completely valid without any sort of fine tuning or testing, which we were unable to do. This, and any further features that we were unable to complete are explained in Section 8.4.

It is also worth noting that this effect is also dependent on the direction that the thrusters are facing. The reasoning is that the thrust that the T200 Thrusters output is not symmetrical in the forward and backwards direction. This is shown in the following figure:

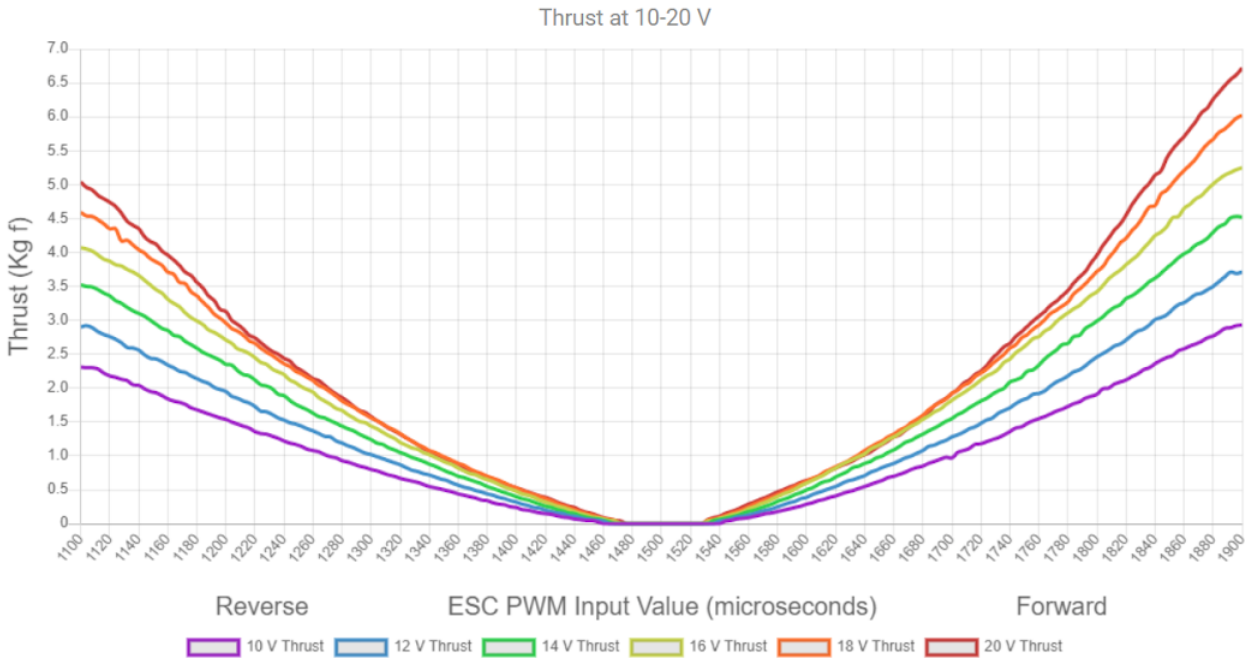


Figure 5.11: Graph of thrust (Kg f) at 10-20V given PWM value (microseconds)[9].

In the middle of the graph is the resting state of the thruster, where no thrust is outputted. Towards the right are the thrust values in the forward direction, and left is the thrust in the backward direction. Upon looking at the graph, it is obvious that the thrusts in the forward and backwards direction is not equal.

The calculation done for the X direction of movement relies on a simple equation that takes in the joystick magnitude m, and the joystick angle θ as inputs. The equation returns the percentage of a thrusters capacity that it should output based on the inputs. The equation is shown below:

$$M_x = \frac{\sqrt{2}}{2} m \cos\theta$$

Figure 5.12: X directed movement equation

This equation was derived through an understanding of the thruster layout. The constant, $\sqrt{2}$ comes from the 45-degree angle that the maneuvering thrusters are placed in. The division by two is because there are four thrusters that will help in the X-directed motion, while there are only two thrusters for the Y direction. Halving the output in the X-direction will make sure movement is equal in all directions. The terms $m\cos\theta$ uses the inputs to calculate the actual percentage based on the inputs from the NMEA string. It is also important to note that the maximum value for M_x is $\frac{\sqrt{2}}{2}$, as this value will come into play later during the rotational value calculation. After M_x is calculated, it is placed in the maneuvering thruster matrix, shown below.

$$M = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ M_x \\ -M_x \\ M_x \\ -M_x \\ M_y \\ M_y \end{bmatrix}$$

Figure 5.13: Maneuvering Matrix

Notice that for thrusters F and H, M_x is negated. This reasoning comes from the thruster pairing explained earlier. For whatever direction one pair is engaged, the other pair must be engaged in the opposite direction. The calculation for M_y is covered in the Forward Thrusters section.

The other responsibility that the maneuvering thrusters have is rotation about the Z axis. Again, the layout clearly supports this possibility because of the angle of the thrusters. Similar to the way we paired thrusters for lateral movement, we also paired thrusters for rotational movement. Rotational pairs are paired along opposite corners of the ROV. Namely, thrusters H and F are paired, and E and G are paired. The thruster pairing is more intuitive when described by an example: if clockwise movement is desired, the E and G paired are engaged in the forward direction, and the F and G pair are engaged backwards. This, and the thrust vectors are shown in Figure 5.14

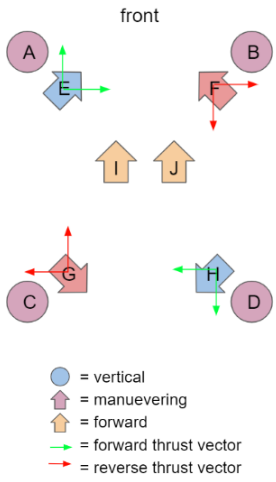


Figure 5.14: Thrust vectors generated when rotating clockwise.

Although the *r* value received in the NMEA string is in the bounds [-1, 1], and therefore can be used as a percentage, there must still be a calculation done to it in order to ensure a working system. The reasoning is that, since the maneuvering thrusters E-H take the responsibility of both X-directed movement as well as rotational, we must be careful not to go beyond the bounds of the thrusters operating regions. Therefore, we must know the maximum percentage that can come from the X-direction calculation, and then allocate the rest for the rotational movement. The calculation for the rotation value is given by the following equation:

$$R_\theta = (1 - \frac{\sqrt{2}}{2})r$$

Figure 5.15: Rotational movement equation

This equation takes into consideration of the upper bound of the X-direction, which is $\frac{\sqrt{2}}{2}$. The term $(1 - \frac{\sqrt{2}}{2})$ takes the rest of the thruster capacity, unused by the X-axis motion, and uses it for rotational motion. This term is multiplied with *r*, and returns a percentage of that allocated capacity to use for rotational motion. Then, similar to the maneuvering matrix, there is also a rotational matrix, where the rotational values, R_θ are placed after calculation. Also similarly to the maneuvering matrix, the rotational paired thrusters share the same sign. This matrix is displayed below.

$$R = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -R_\theta \\ R_\theta \\ R_\theta \\ -R_\theta \\ 0 \\ 0 \end{bmatrix}$$

Figure 5.16: Rotational Matrix

Forward Thrusters

Lastly, the forward thrusters, I and J are directly responsible for movement along the Y-axis. There is no need to angle these as all of the other movement types have already been assigned to the other thruster types. The forward thrusters are placed adjacent to each other in the center of the ROV frame. The forward thrusters are the only thrusters

that should generate any movement along the Y-axis, as any Y-axis movement generated by the maneuvering thrusters should cancel itself out.

The calculation for the forward thrusters shares a similar structure to the X-direction calculation. However, a major difference is that, since the forward thrusters are not angled, there is no need to account for it. The equation is shown below:

$$M_y = m \sin \theta$$

Figure 5.17: Forward movement equation

This equation is straightforward due to the placement of the forward thrusters. Like the vertical thrusters, the forward thrusters only plays a single role– movement in the Y-direction. Therefore, the process for the calculation is much simpler as opposed to the maneuvering thrusters. The result for this calculation however, still belongs in the maneuvering matrix, since the maneuvering matrix is a central location for X and Y directed movement. Referring back to Figure 5.13, it is clear to see that M_y is correctly assigned to the bottom two spaces, where thrusters I and J will receive values from.

Combining

The final phase is to combine all three matrices together and assign each value to the corresponding thruster. A collection of all the previously described matrices are shown below, along with the process of combining them.

$$T = \begin{bmatrix} T_A \\ T_B \\ T_C \\ T_D \\ T_E \\ T_F \\ T_G \\ T_H \\ T_I \\ T_J \end{bmatrix} \quad V = \begin{bmatrix} v \\ v \\ v \\ v \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad M = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ M_x \\ -M_x \\ M_x \\ -M_x \\ M_y \\ M_y \end{bmatrix} \quad R = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -R_\theta \\ -R_\theta \\ R_\theta \\ -R_\theta \\ 0 \\ 0 \end{bmatrix}$$

$$T = V + M + R$$

Figure 5.18: Thruster Matrices Combination Process

Combining the matrices is simple matrix addition, and it is clear to see how the individual thruster values receive the correct percentages. At the end of this process, the thruster matrix, T, will have a collection of percentages to assign each thruster. The Arduino system then reads this matrix, converts the percentage into a PWM value, and assigns it to the corresponding thruster.

5.5 Communications Interface

It is important that the processes done on the Arduino send and receive data to and from the other subsystems correctly. To do this, data must be formatted consistently so that the receiving end of the communication can interpret it properly. This formatting process is covered in detail in Section 4.4, but the following subsections will provide a general overview of the process within the Arduino

Both the GUI and Arduino systems must be capable of generating and parsing control strings, while the communications system is responsible for passing these strings between the two systems.

5.5.1 Control String Parsing

The control string contains seven main signals: direction, magnitude, vertical, rotation, light toggle, sampler toggle, and camera tilt. Each of these values are converted into an instruction to be sent to the corresponding component on the ROV. The direction, magnitude, vertical, and rotation signals are used in a calculation to determine which thrusters to engage for the movement that the signals describe. The light and sampler toggle simply send a signal to the lights and sampler respectively to toggle them. Lastly, the camera tilt signal tells the servo on the ROV to tilt up, down, or stay still.

5.5.2 Sensor String Generation

The sensor string contains 3 values corresponding to the temperature (C), pressure (mbar), and heading (deg) read by the sensors on the ROV. These values are packed into a string according to the protocol described in section 4.4.

Chapter 6

Video

6.1 Overlay

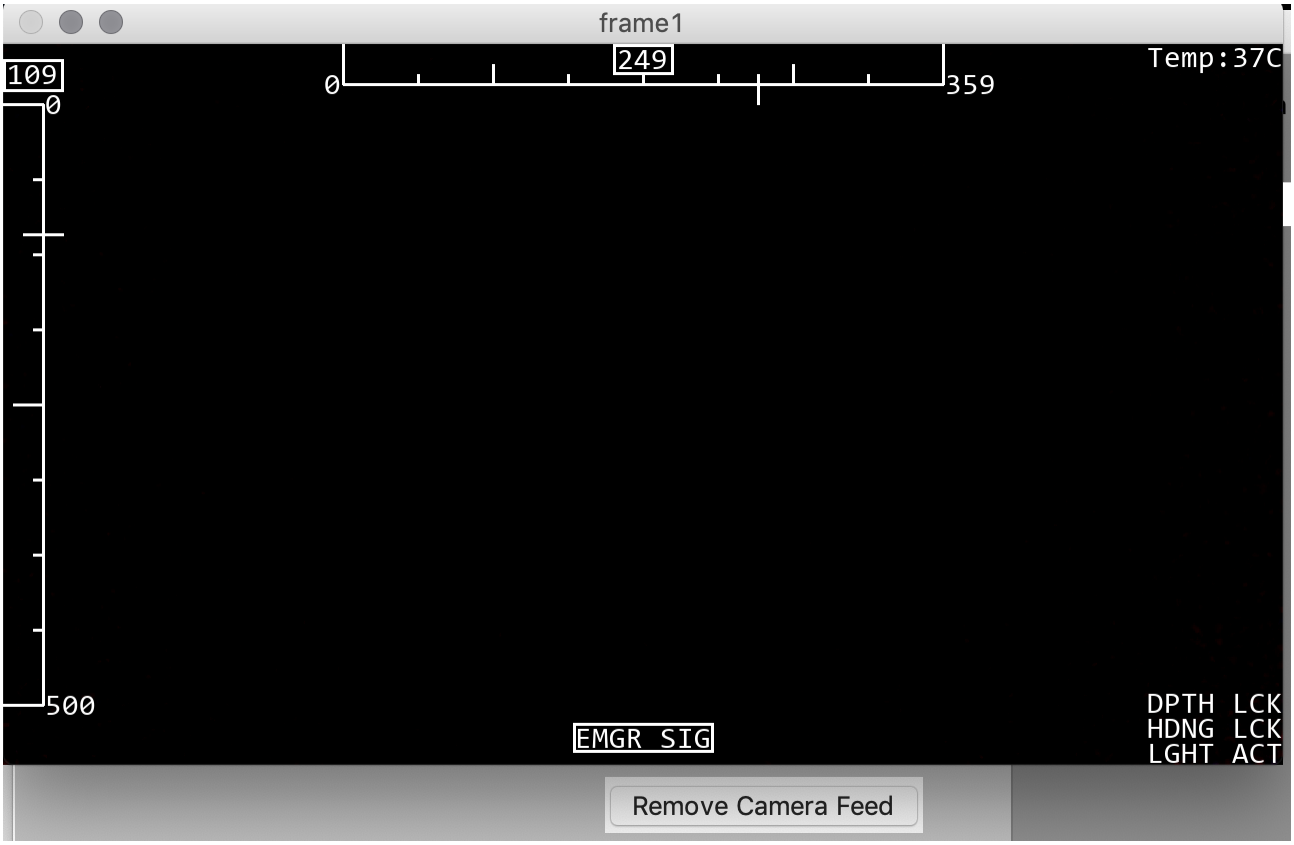


Figure 6.1: Demonstration of our Video Overlay

The overlay for the camera feed has multiple features shown in Figure 6.1. On the left side the vertical bar denotes the depth read ranging from 0 to 500 feet. At the top of the frame is a horizontal bar denoting the heading read ranging from 0 to 359 degrees. At the top right there is a temperature read in Celsius.The data displayed on the overlay is directly fed from the socket connecting the GUI to our ROS node. This data is updated every time the camera supplies

a image to display, linking the camera FPS to the overlay refresh rate.We chose to have the overlay be white as a compromise between visibility and performance. Since this camera will primarily be underwater, the obvious choice for the overlay color is a red to contrast the mostly blue underwater environment. Due to how the overlay is added into the camera feed, we found that we could achieve less performance overhead by sticking strictly with white as our overlay color.

6.2 Integration in GUI

In conjunction with the GUI the video feed has additional functionality. From the GUI the user can spawn up to five separate windowed video feeds if needed to spread across multiple monitors. In addition each of these frames can be resized to the observers content. As mentioned in section 4.4, the video feed has acceptable latency and does not look choppy despite being capped at 10 frames per second.

6.3 Data Signal Amplification

When testing the video capabilities over the tether we ran into signal issues. The video data signal was not strong enough to make it from end to end across the tether, so we purchased a Blue Robotics Fathom-X Interface Set interface set to amplify the camera signal. The set comes with two boards that each connect to the electrical components over Ethernet and include terminal blocks (green) used to connect the tether wires.

6.4 Camera Block diagram

Figure 6.3 shows a block diagram of the camera set up. The Blue Robotics low-light camera is attached to one amplifier board via Ethernet. The amplifier board connects to the Blue Robotics Fathom tether wires over a twisted pair in the terminal block. The other end of the tether connects to the terminal block on the second board, which connects to the topside laptop via Ethernet to provide a video feed to the GUI overlay.

Figure 6.2: Camera Block Diagram [8][14]

Chapter 7

Societal Issues

7.1 Health and Safety

It is the job of the engineer to ensure that the products they develop are safe. We have taken design steps to minimize risk and maintain the health and safety of our users. The implementation of an on-board LiPo battery on the ROV prevents the need for a high-voltage power source topside near the user. The mechanical engineering team has waterproofed tubes to store the subsea components to prevent the underwater impact of pressure and leakage during research missions. They have combined all the batteries needed into a single acrylic tube in order to reduce the number of underwater connections required for safe operation. Despite these steps human error is always a factor in any engineered product. Although the LiPo battery has its own dangers due to its high power density and volatility, there is small risk when in the hands of trained users. It is the job of both the engineer and the user to be LiPo trained prior to handling the ROV.

7.2 Environmental Impact

With this project, we have several stakeholders relating to environmental sustainability. We specifically have kept those organizations in mind while developing the features of this ROV. Features like ease of operation, data displays, and response times were built to ensure that the environmental organizations that plan to use this ROV can do so effectively and reliably. With this, we hope that they are able to obtain more accurate, reliable, and consistent data from the ROV which they can use to determine the state of the environment they seek to monitor. While our efforts do not directly impact the environment, we sought to indirectly make a positive impact on environmental sustainability through listening to our customers needs and providing corresponding features and capabilities with our ROV. After testing what we could given our circumstances we believe we have successfully engineered a system which will give environmental researchers access to an enhanced ROV software suite which will assist in environmental sustainability research.

7.3 Usability

As developers and designers, the long period of time that we have spent with this project has given us a lot of knowledge on how our product works. However, the end user does not have the same expertise that we have. Therefore, it is important that we design a product that can be easily controlled, understood, and used efficiently, without any prior knowledge on how the system works.

A lot of this usability relies on the structure of the Graphical User Interface, or GUI, since it is what the user will mainly interact with when piloting the ROV. Not only does it play a key role in controlling the ROV, but all of the data readouts will be displayed in the GUI as well. It is then important that this data is easily readable and clear. Our design strives for maximum readability by implementing a well-organized structure for the GUI, and as many graphical representations of the data as possible.

Another usability feature that our design has is its modularity. The sensors, thrusters, and other components can be upgraded and switched out with relative ease. Our thorough documentation and organized code will assist any user that wishes to replace any parts on our original design.

Chapter 8

Conclusion

8.1 Summary

In conclusion, our team has developed an enhanced marine ROV control system called Neptune. The control system has a GUI with a minimalistic display that allows for user input, a communications system to process and communicate commands, a subsea system with a microcontroller that powers and commands the retrofit components, and a live-video feed. The implementation of a modernized ROV with an enhanced control system will, in combination with the great work our mechanical engineering counterpart has done, leave the RSL with a firm foundation to further our work in order to make an impact on marine environmental research.

8.2 Obstacles

The most impactful obstacle that hindered our progress was the imposition of the shelter-in-place order that forced the university to close. Because of this, neither group could meet in person to fully complete the system. Another obstacle was our unfamiliarity with ROS. Although there were resources in the RSL to assist our understanding, it was still difficult as beginners to get up to speed and figure out how the new operating system functioned. As with many projects, a final obstacle for us was inter-team communication (between the COEN and MECH teams). Communicating goals, tasks, ideas, etc. across teams produced some associated overhead that slowed progress.

8.3 Lessons Learned

Planning, designing, and developing this control system taught us valuable lessons in project development. First, it cannot be overstated the importance of communication across all parties on a project. When both teams delegated individuals as a direct points of contact for their team, our inter-team communications greatly improved along with development progress. There are often too many voices in a room to communicate goals and needs, so we learned that one must always communicate with clarity and transparency. Communication was especially important when working remotely with team members during the shelter-in-place.

Another lesson learned was the value of version control in a software project. Using a subversion (SVN) repository we could merge our changes and update our files to make sure we were working with the most current working repository. This was helpful when we wanted to revert back to an old copy or track changes team members were making to the project.

Scheduling was another valuable lesson that our advisor Dr. Kitts was very helpful with. During our weekly meetings he provided deadlines that guided our progress and forced our team to create sub-deadlines for each other. Schedules help deal with the scope of the project and incite action that is crucial for progress. Given the circumstances our schedule, in combination with communication, is what kept the team on track.

8.4 Future Work

As mentioned in Section 8.2 the shelter in place order hindered progress, so some of the things we wanted to accomplish, but couldn't because of that barrier were: assembly, field testing and fine tuning. Because we could not physically meet in the RSL the ROV components were shipped to various team members, so we could not assemble the final ROV. Because we couldn't assemble the final product, we could not perform any field testing (we had planned on testing the ROV functionality in Lake Tahoe or Monterey Bay). Finally, because we could not field test or assemble, we could not make any final tweaks to the system to polish its functionality.

In addition, there are some extensions of the system we hope can be implemented in the future: autopilot control, acoustic sensor, and teleoperations. Implementing an autopilot system with a depth and heading lock would allow the ROV to maintain a certain path in operation. Another improvement feature could be acoustic sensor that could track the ROV's position under water (similar to GPS), which would provide important data during a mission. A final feature could be the implementation of teleoperations so that a user could remotely control the ROV or receive a stream of data from land. Because of our implementation of ROS, this could potentially be done using an existing ROS library or package. Most of these features would have required assembly, field testing, or fine tuning to be accomplished.

References

[1] US Department of Commerce and National Oceanic and Atmospheric Administration. 2009. How much of the ocean have we explored? (January 2009). Retrieved December 12, 2019 from https://oceanservice.noaa.gov/facts/exploration.html

[2] Shuqing Zhao et al. 2004. The 7-Decade Degradation of a Large Freshwater Lake in Central Yangtze River, China. ACS Publications (December 2004). DOI:http://dx.doi.org/10.1021/es0490875

[3] Killian Poore. 2016. *Tessie: Expandable Mini Underwater Remotely Operated Vehicle*. SCU Robotics Systems Lab Technical Report. Santa Clara University, Santa Clara, Ca.

[4] Anon. BlueROV2 - Affordable and Capable Underwater ROV. Retrieved December 12, 2019 from https://bluerobotics.com/store/rov/bluerov2/

[5] Oli Francis, Graeme Coakley, and Christopher Kitts. 2002. A Digital Control System for the Triton Undersea Robot. IFAC Proceedings Volumes 35, 2 (2002), 633–638. DOI:http://dx.doi.org/10.1016/s1474-6670(17)34010-7

[6] Jeffrey Abercrombie, Kevin Brashem, Johnny Buccola, and Matt Pavlik. 2010. Triton Autonomous Navigation and Integrated Control. (July 2010).

[7] Killian Poore. 2016. Tessie: Expandable Mini Underwater Remotely Operated Vehicle. (2016).

[8] Blue Robotics Low-Light HD USB Camera. Picture. Retrieved from https://bluerobotics.com/store/sensors-sonars-cameras/cameras/cam-usb-low-light-r1/

[9] Blue Robotics T200 Thruster. Picture. Retrieved from https://bluerobotics.com/store/thrusters/t100-t200-thrusters/t200-thruster/

[10] Blue Robotics Basic ESC (Electronic Speed Controller). Picture. Retrieved from https://bluerobotics.com/store/thrusters/speed-controllers/besc30-r3/

[11] Blue Robotics Bar30 High-Resolution 300m Depth/Pressure Sensor. Picture. Retrieved from https://bluerobotics.com/store/sensors-sonars-cameras/sensors/bar30-sensor-r1/

[12] Blue Robotics Celsius Fast-Response, Temperature Sensor (I2C). Picture. Retrieved from https://bluerobotics.com/store/sensors-sonars-cameras/sensors/celsius-sensor-r1/

[13] Adafruit Triple-axis Accelerometer+Magnetometer (Compass) Board - LSM303. Picture. Retrieved from https://www.adafruit.com/product/1120

[14] Blue Robotics Fathom-X Tether Interface Board Set. Picture. Retrieved from https://bluerobotics.com/store/comm-control-power/tether-interface/fathom-x-r1/

[15] We Are YKK. UART and Buad Rate. Picture. Retrieved from https://sites.google.com/site/weareykk/more-details/uart/uart-and-buad-rate

[16] Arduino Mega 2560 Rev3. Picture. Retrieved from https://store.arduino.cc/usa/mega-2560-r3

Appendix A

Customer Needs

Robot has an accurate control system that can manage average currents
Robot can navigate the ocean floor
Robot has a camera that views the location that is being sampled
Robot has lights to make visibility easier
Robot can accurately measure its depth and temperature
Robot includes an on-board compass to orient the driver
The robot is safe to be used by lightly trained students
The robot is powered by on-board batteries that are 50V or less (5)
The robot allows for the batteries to be easily charged and replaced
The robot is reliable and can be trusted to perform at the required depth and temperature
The robot relies on a simple design

Appendix B

Source Code

B.1 GUI

```
# file: main.py
# owners: Chris Layco, Alex Achramowicz, Cooper Zediker
# description: main file for GUI

from imports import *

nmea_string = None
data = None
map_dict = {}
gui = None
gamepad = None


# function: startup()
# description: called first to initialize components
def startup():

    #prepare necessary resources for gamepad
    if config.gamepad_flag:
        global gamepad
        print("gamepad initializing...")
        #gamepad initialization
        generate_dictionaries()
        gamepad = Gamepad()
        config.gamepad_flag = gamepad.init()

    # socket initialization
    if config.socket_flag:
        s = socket.socket()
        port = 29502
        s.connect(("localhost", port))

    # camera initialization
    if config.cam_flag:
        cam_init()
```

```
# function: processes()
# description: called to send control strings over socket
# and update GUI overlay video and data
def processes():
    global nmea_string, gamepad

    #listen for gamepad
    if config.gamepad_flag:
        gamepad.listen()
        interpret(gamepad)
        pass
    else:
        pass

    nmea_string = generate(config.top_data)

    if config.socket_flag:
        #send control string over socket s
        socket_send(nmea_string, s)
        pass

    # update camera
    if config.cam_flag:
        cam_update()
        pass

    tmpr = math.sin(datetime.now().second)
    pres = random.randrange(0, 30000)
    head = random.randrange(0, 360)

    gui.status_display(nmea_string)
    gui.sensor_display("TMPR", tmpr)
    gui.sensor_display("PRES", pres)
    gui.sensor_display("HEAD", head)
    gui.after(config.PROCESS_RATE, processes)


# function: on_closing()
# description: called on exit to kill camera and GUI processes
def on_closing():
    if messagebox.askokcancel("Quit", "Do you want to quit?"):
        if config.gamepad_flag:
            pass

        if config.socket_flag:
            pass

        if config.cam_flag:
            cam_kill()
            pass

        gui.destroy()
```

```
# main module to execute functions
if __name__ == "__main__":

    startup()

    #build GUI
    gui = neptuneGUI()
    ani = gui.sensor_animate(config.PROCESS_RATE * 2)

    gui.after(50, processes)

    gui.protocol("WM_DELETE_WINDOW", on_closing)
    gui.mainloop()
```

B.2 Communications

```
//Basically this only initializes values. Its main purpose is to launch manage_connection.
int main(int argc, char **argv)
{
  ros::init(argc, argv, "nmea_serial_node");
  ros::NodeHandle n_local("~");
  ros::NodeHandle n;

  std::string port;
  int32_t baud;
  n_local.param<std::string>("port", port, "/dev/ttyUSB0");
  //n_local.param<std::string>("port", port, "/dev/ttyACM0");
  //n_local.param("baud", baud, 115200);
  n_local.param("baud", baud, 9600);

  std::string frame_id;
  n_local.param<std::string>("frame_id", frame_id, "navsat");
  //ros::Timer timer = n.createTimer(ros::Duration(1.0),
                                   //boost::bind(manage_connection, _1, n, port, baud, frame_id));
  manage_connection(n, port, baud, frame_id);
  ros::spin();

  rx_stop_all();

  return 0;
}

/*
 *  This function, as per its name, manages the connection over our RS485 serial connection.
 *  It also contains the main work loop of this whole node.
 *  This function is meant to run forever.
 */
void manage_connection(ros::NodeHandle& n, std::string port, int32_t baud, std::string frame_id)
{
  if (rx_prune_threads() > 0)
  {
    // Serial thread already active. Nothing to do here.
```

```
  return;
}

//Create all the socket stuff for communicating with the GUI
int GUISocket = createSocket();

// Only output error messages when things were previously peachy.
static int previous_success = 1;


while (ros::ok())
{
  ROS_DEBUG_STREAM("Opening serial port: " << port);
  int serial_fd = open(port.c_str(), O_RDWR | O_NOCTTY | O_NDELAY | O_NONBLOCK);

  //The following 2 if statements are from the original serial node. They are pretty archaic and they
  //any reasonable programming techniques.
  if (serial_fd == -1)
  {
    if (previous_success)
    {
      ROS_ERROR_STREAM("Could not open " << port << ".");
    }
    goto retry_connection;
  }

  if (!isatty(serial_fd))
  {
    if (previous_success)
    {
      ROS_ERROR_STREAM("File " << port << " is not a tty.");
    }
    goto close_serial;
  }

  struct termios options;
  tcgetattr(serial_fd, &options);
  options.c_cflag = 0;
  options.c_cflag |= CS8;

  options.c_cflag |= (CLOCAL | CREAD);

  ROS_DEBUG_STREAM("Setting baud rate: " << baud);
  switch (baud)
  {
  case 9600:
    cfsetispeed(&options, B9600);
    cfsetospeed(&options, B9600);
    break;
  case 19200:
    cfsetispeed(&options, B19200);
    cfsetospeed(&options, B19200);
    break;
  case 38400:
```

```
      cfsetispeed(&options, B38400);
      cfsetospeed(&options, B38400);
      break;
    case 57600:
      cfsetispeed(&options, B57600);
      cfsetospeed(&options, B57600);
      break;
    case 115200:
      cfsetispeed(&options, B115200);
      cfsetospeed(&options, B115200);
      break;
    default:
      ROS_FATAL_STREAM("Unsupported baud rate: " << baud);
      ros::shutdown();
    }
    options.c_iflag = 0;
    options.c_oflag = 0;
    options.c_lflag = 0;
    options.c_cc[VMIN]  = 0;
    options.c_cc[VTIME] = 1;
    tcsetattr(serial_fd, TCSAFLUSH, &options);

    // Successful connection setup; kick off servicing thread.
    ROS_INFO_STREAM("Successfully connected on " << port << ".");
    previous_success = 1;

    //This is the most important part of this file. In essence the following lines represent the master-
    //Only once we have received a response do we send any further commands to the ROV.
    char* GUISentence;
    while(1){
      getNewSentence(GUISocket, GUISentence);
      tx_func(GUISentence, serial_fd);
      rx_func(n, serial_fd, frame_id,GUISocket,0);
    }

    ROS_INFO("OUT OF MASTER SLAVE LOOP, SHOULDN'T HAPPEN!");
    break;

close_serial:
    close(serial_fd);

retry_connection:
    if (previous_success)
    {
      ROS_ERROR("Retrying connection every 1 second.");
      previous_success = 0;
    }

    ros::Duration(1.0).sleep();
  }
}

/*
 *  Get a new control sentence from the GUI socket, store into GUISentence
```

```
 */
void getNewSentence(int GUISocket, char *sent)
{
  while(1){
    // receive string from GUI,
    while (read(GUISocket, sent, 40)) {
      ROS_INFO("New Sentence from GUI");
    }
    //close (GUIsocket);
    break;
  }

}

/*
 *  This function will transmit the parameter msg over our serial RS485 connection.
 *  It will continue to try to transmit the message until an error occurs or the message is successfully
 */
void tx_func(char *msg, int fd)
{
  ROS_DEBUG("tx_func entered");

  static int consecutive_errors = 0;

  char buffer[256];
  strcpy(buffer,msg);
  int buffer_length = strlen(msg);
  bool keepgoing=true;
  // No guarantee that write() will write everything, so we use poll() to block
  // on the availability of the fd for writing until the whole message has been
  // written out.
  const char* buffer_write = buffer;
  struct pollfd pollfds[] = { { fd, POLLOUT, 0 } };
  while (keepgoing)
  {
    int retval = poll(pollfds, 1, 1000);

    if (pollfds[0].revents & POLLHUP)
    {
      ROS_INFO("Device hangup occurred on attempted write.");
      return;
    }

    if (pollfds[0].revents & POLLERR)
    {
      ROS_FATAL("Killing node due to device error.");
      ros::shutdown();
    }

    //Once we poll, we can then transmit the message.
    ROS_INFO("TX STARTING");
    retval = write(fd, buffer_write, buffer_length - (buffer_write - buffer));
    if (retval > 0)
    {
```

```
      ROS_INFO_STREAM("TX SUCCESS: "<< msg);
      buffer_write += retval;
      keepgoing = false;
    }
    else
    {
      ROS_DEBUG("TX Bad");
      if (++consecutive_errors >= 10)
      {
        ROS_FATAL("Killing node due to %d consecutive write errors.", consecutive_errors);
        ros::shutdown();
      }
      break;
    }
    if (buffer_write - buffer >= buffer_length)
    {
      consecutive_errors = 0;
      break;
    }
  }
}

/*
 *  This function will recieve a single NMEA string from the serial RS485 connection.
 *  This function will continually attempt to search for a string until it finds a successful transimsi
 *  Upon receiving a malformed NMEA string, it will transmit a REPEAT string which sub-sea will use to
 *  the last transmission. This function is complicated.
 */
static void rx_func(ros::NodeHandle& n, int fd, std::string frame_id, int GUISock, uint32_t byte_time_n
{
  ROS_INFO("RX WAITING...");
  ROS_DEBUG("New connection handler thread beginning.");

  struct pollfd pollfds[] = { { fd, POLLRDNORM, 0 } };

  //Buffer to read into
  char buffer[2048];
  char* buffer_read = buffer;
  char* buffer_end = &buffer[sizeof(buffer)];
  while (threads_active)
  {
    //The next 4 lines all revolve around polling the RS485 connection.
    //poll() is a pretty complex and low level call, please read the linux man page
    //to get a better understanding of it. In short the following are required to
    //read from the connection. The 60ms timeout is the minimum possible timeout for
    //reliable comms, it is unknown why this is the limit since theoritical response time should be ~15
    errno = 0;
    int retval = poll(pollfds, 1, 60);
    ros::spinOnce();  //Unsure if this is necessary... Seems to add reliability
    ROS_DEBUG("Poll retval=%d, errno=%d, revents=%d", retval, errno, pollfds[0].revents);

    if (retval == 0)
    {
      // No event, just 1 sec timeout.
```

```
  continue;
}
else if (retval < 0)
{
  ROS_FATAL("Error polling device. Terminating node.");
  ros::shutdown();
}
else if (pollfds[0].revents & (POLLHUP | POLLERR | POLLNVAL))
{
  ROS_INFO("Device error/hangup.");
  ROS_DEBUG("Shutting down publisher and subscriber.");
  ROS_DEBUG("Closing file descriptor.");
  close(fd);
  ROS_DEBUG("Exiting handler thread.");
  return;
}

// We can save some CPU by sleeping if the number waiting bytes is really small
if (byte_time_ns > 0)
{
  int waiting_bytes;
  errno = ioctl(fd, FIONREAD, &waiting_bytes);
  if (errno == 0)
  {
    int wait_for = 0;
    int buffer_plus_waiting = (buffer_read - buffer) + waiting_bytes;
    if (buffer_plus_waiting < RX_INITIAL_LENGTH)
    {
      wait_for = RX_INITIAL_LENGTH - buffer_plus_waiting;
    }
    else if (waiting_bytes < RX_SUCCESSIVE_LENGTH)
    {
      wait_for = RX_SUCCESSIVE_LENGTH - waiting_bytes;
    }
    if (wait_for > 0)
    {
      struct timespec req = { 0, wait_for * byte_time_ns }, rem;
      ROS_DEBUG_STREAM("Sleeping for " << wait_for << " bytes (" << byte_time_ns << " ns)");
      nanosleep(&req, &rem);
    }
  }
}

// Read in contents of connection to buffer
ros::Time now = ros::Time::now();
errno = 0;
retval = read(fd, buffer_read, 20);
ROS_DEBUG("Read retval=%d, errno=%d", retval, errno);
ROS_DEBUG_COND(retval < 0, "Read error: %s", strerror(errno));
if (retval > 0)
{
  if (strnlen(buffer_read, retval) != retval)
  {
    ROS_WARN("Null byte received from serial port, flushing buffer.");
```

```
        buffer_read = buffer;
        continue;
      }
      //buffer_read += retval;
    }
    else if (retval == 0)
    {
      ROS_INFO("Device stream ended.");
      ROS_DEBUG("Shutting down publisher and subscriber.");
      //pub.shutdown();
      //sub.shutdown();
      ROS_DEBUG("Closing file descriptor.");
      close(fd);
      ROS_DEBUG("Exiting handler thread.");
      return;
    }
    else
    {
      // retval < 0, indicating an error of some kind.
      if (errno == EAGAIN)
      {
        // Can't read from the device, try again.
        continue;
      }
      else
      {
        ROS_FATAL("Error reading from device. retval=%d, errno=%d, revents=%d", retval, errno, pollfds[0
        ros::shutdown();
      }
    }

    //Handle the message received
    ROS_DEBUG("Buffer size after reading from fd: %d", sizeof(buffer_read));
    ROS_DEBUG_STREAM("Starting to check Sencence..." << buffer_read);
    char* sentence = strchr(buffer_read, '$');
    if (sentence == NULL){
      repeat(fd);
      continue;
    }
    ROS_DEBUG("Sencence start good");

    char* sentence_end = strchr(sentence, '*');
    if (sentence_end == NULL){
      repeat(fd);
      continue;
    }
    ROS_DEBUG("Sencence end good");
    *sentence_end = '\0';
    _handle_sentence(now, sentence, frame_id.c_str());
    sendGUISentence(sentence, GUISock);
    buffer_read = sentence_end + 1;
    break;  //If we have gotten this far, the message is good, so we exit.
  }
}
```

B.3 Arduino