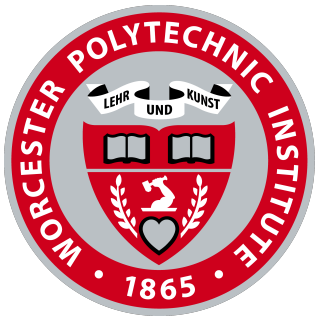


WPI

TSS MQP Report

NOVEL TETHERED SUBMERSIBLE FOR MARINE SURVEY APPLICATIONS

WORCESTER POLYTECHNIC INSTITUTE

DEPTS. OF: ROBOTICS ENGINEERING, MECHANICAL ENGINEERING,
ELECTRICAL & COMPUTER ENGINEERING, COMPUTER SCIENCE

MAJOR QUALIFYING PROJECT

Authors:
CONKLIN, JASON
HUANG, JEFFREY
KIRSCHNER, RAVI
WARD, BENJAMIN

Advisor:
PROF. MICHALSON, WILLIAM R.

Additional Contributions:
MAUDE, EVELYN
ROSENBAUM, JONATHAN

A Major Qualifying Project submitted to the faculty of Worcester Polytechnic Institute in partial fulfillment of the requirements of the Degree of Bachelor of Science.

May 6th, 2021

Contents

1 Abstract

Current underwater inspection vehicles are either extremely slow or high priced despite their plethora of use cases. The Tethered Submersible Surveyor is an underwater, semi-autonomous tethered vehicle designed to glide across the ocean floor for fast close inspection work. The tether provides the craft's primary propulsion as well as communications and power supplied from a boat on the surface. The TSS aims to enable applications such as topology scanning, utility mapping, and other surveying applications.

2 Introduction

The Tethered Submersible Surveyor (TSS) is an underwater, semi-autonomously controlled vehicle akin to a UAV, or "drone." Primary propulsion is provided by means of a tether, connecting the vehicle to a boat on the surface. Topside infrastructure also provides an operators console, telemetry read-out, navigational elements and sensing, and primary power via a battery bank and/or generator. The primary purpose of the TSS is for underwater exploration and surveying tasks.

2.1 Background

There are many existing robotics vehicles and solutions to problems involving underwater exploration and manipulation. When brainstorming our Tethered Submersible Surveyor (TSS) we looked at some of the existing solutions and the gaps or shortcomings they have and how they relate to our project.

2.2 Prior Art

There are many existing examples of drones and/or sensors designed for subsurface data collection. Some examples we found inspiration include both military and civilian craft. These craft range from hobbyist tethered ROVs, that require direct control from a surface operator, to completely autonomous unmanned underwater Vehicles (UUV) used by the United States Military for water mine countermeasures operations. Below are some examples of vehicles and sensors that provided context and/or inspiration during our research:

2.2.1 Hydroid Remus MK18 Family of Drones

The United States Navy uses this family of drones for shallow water mine countermeasures, seafloor mapping, and hydrographic reconnaissance [14]. These drones function autonomously and surface for retrieval after completing a mission. These drones tend to be extremely long (> 3 meters) and require specialized technicians to operate.

Figure 1: Hydroid MK 18 Mod 1 Swordfish UUV [13]

2.2.2 Boat-towed Metal Detectors

There are many commercial options for high sensitivity boat-towed metal detectors. These sensors are usually single purpose, however, and a generally extremely expensive (approx. $10,000 USD (2020)). Despite this, they are a common tool of treasure hunters for their wide detection area [12].

Figure 2: JWFisher Pulse 12 Boat Towed Metal Detector [12]

2.2.3 Robotic Operated Underwater Vehicles (ROVs)

Used for both commercial and research purposes, ROVs are designed for direct control by a remote operator. ROVs also tend to have manipulation / actuation capabilities for collecting samples or performing repairs [6].

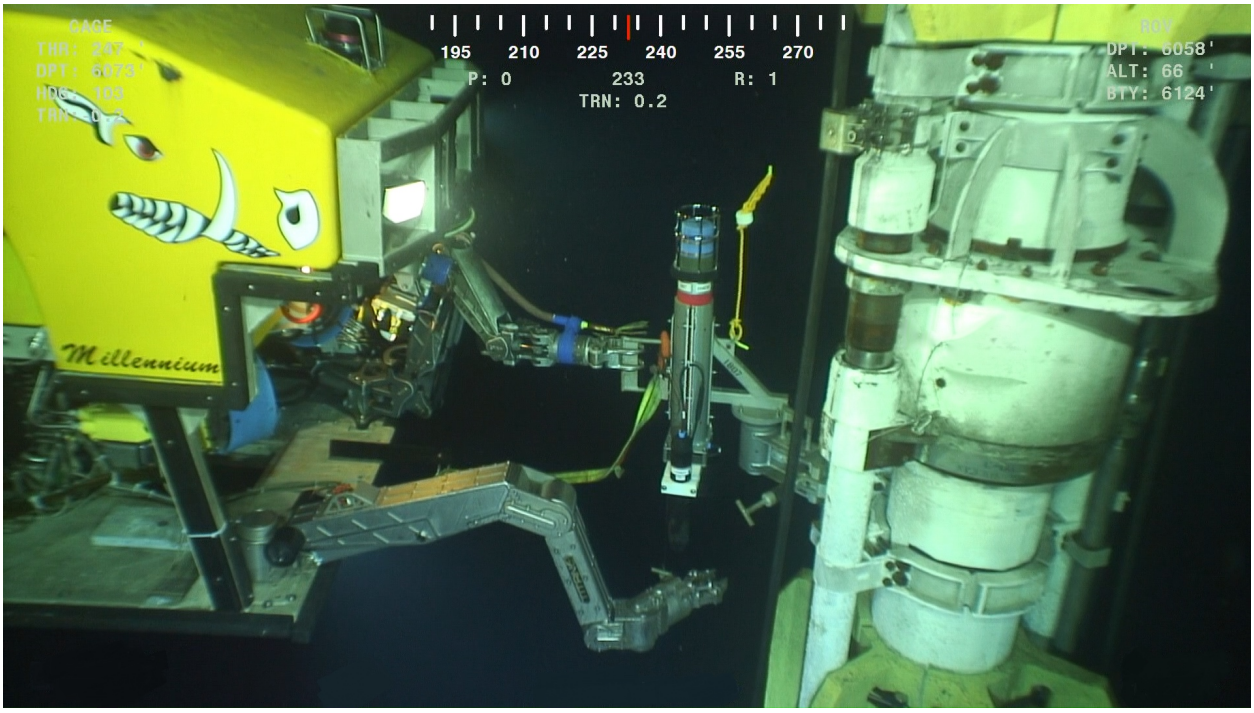


Figure 3: ROV performing subsurface work on an offshore drilling platform [6]

2.3 Project Scope and Goals

We aimed to construct the TSS underwater vehicle and topside infrastructure sufficient for basic operation - correct buoyancy, hydrodynamics, waterproofing, and control systems - as well as a full sensor suite and an operators' console with telemetry display. The main goal of this project is to construct a working system and to enable further development of sophisticated control and autonomy systems including obstacle avoidance, payload integration, and mapping tasks.

2.3.1 Product Requirements

A minimum viable product will have the following elements and capabilities:

- Can control and maintain depth setpoint down to 20 meters depth
- Is controllable and operable underwater.
- Waterproofing is sufficient to ensure no uncontrolled leakage at 20 meters depth

- Environmental sensing capability including depth and inertial measurements
- Basic control software including actuation of control surfaces
- Depth modulation using a ballast tank

These requirements are representative of a baseline operational environment, including a large factor of safety for initial pool testing. With these capabilities, the craft should be able to attain basic maneuvering as well as providing enough sensor capabilities suitable for simple operation.

Additional stretch goals include:

- Modular or semi-modular sensor bay availability for payload
- Environmental sensing capability including water speed, water pressure, front+side rangefinding, and inertial measurements
- Control software to enable autonomous bottom-following and arbitrary actuation tasks

More sophisticated sensors and control software will allow us to meet our operational goals and maneuver along the bottom of the pool or in a lake or ocean environment free of most obstacles.

Additionally, the project will emphasize the following design goals:

- 3D printed structural components where possible to enable rapid design iteration
- Continuous testing and integration of software and hardware to validate continued functionality
- Minimized cost, with the understanding that one-off or low-volume hardware is expensive
- Minimized size of underwater vessel in order to decrease manufacturing time, maximize assembly speed, and facilitate transport

These design goals allow for the most rapid iteration and reduced machine tool requirements. This should allow us to reduce the time spent on-campus, as well as simplifying procedures.

2.4 Legal Implications

Depending on the locale, a variety of laws and treaties limit the rights of underwater surveyors in the activities of treasure hunting, underwater archaeology, and salvage. Relevant summaries are presented in the subsections below.

2.4.1 UNESCO Convention on the Protection of the Underwater Cultural Heritage

A 2001 UNESCO Convention saw the signing of several articles which limit or prohibit the act of treasure hunting by increasing the scope of the definition of 'underwater heritage' [1]. While this component of international law limits treasure hunting, underwater archaeology, and salvage activities in the signatory countries, many major countries (including USA) are not signatories to this agreement. This means that treasure hunting and salvage operations in the United States are unaffected by the Convention [11].

2.4.2 United States Abandoned Shipwreck Law

Depending on the state, the laws surrounding the exploration and extraction of abandoned shipwrecks can vary. Some states (such as Massachusetts) have highly regulated and organized boards overseeing the underwater resources within their territory, while others have little to no regulation. More research is required regarding specific states' laws and how they compare with the Federal laws. [3].

3 System Overview

We separated our system into three sections - the mechanical section, the electrical section, and the software section. The mechanical section focuses on creating a physical body that is hydrodynamic when moving through the water as well as designing the electromechanical systems for control of the craft. The electronics provide the power systems that run the craft and handles the input and output of data and control signals to allow the craft perform its duties underwater. The software

section interprets the data and control signals and moves the mechanical sections to the appropriate positions. It also sends data to the topside, which allows for control by the user through an application that monitors and controls the craft's subsystems.

3.1 Mechanical Overview

The key characteristics relevant to mechanical design in our project include the need to allow for a craft to be tethered and dragged by a boat through water at a maximum depth of 20m and at a speed of approximately 5-15 knots. Although the end goal of this project is to integrate enable real-time scanning of the sea bed for treasure and other inspection related uses, the focus of the first stage of our project this year was to create small-scale, proof of design concept prototype that could potentially be scaled up to meet the ultimate objectives.

As the first iteration prototype for this project, we placed heavy emphasis on the creation of a baseline for a hydrodynamic craft that could be iterated on in future work. 4 shows a labelled system diagram of the major components for our craft, which includes the fuselage, wings, tail, and internal subsystems.

Once the main body of the craft was designed, we could determine additional requirements for interior design of the craft and subsystems needed for control of the craft. The main design goal for the mechanical portion of this project was to design and assemble a hydrodynamic craft that could submerse and emerge from the water independently through remote control as well as having basic control functions to maintain stability while in motion when dragged with a tether through the water. Key mechanical design requirements for the craft include water tightness for all mechanical subsystems and as well as modularity and compactness for internal subsystems.

Although much of the design of hydrodynamic components for our craft, such as the wings and the fuselage, was based on the design and documentation for airplanes as both crafts are essentially large streamlined objects designed to move through fluids as efficiently as possible, it was important to review update as necessary the calculations and formulas presented in referenced material to correctly align with our distinct objectives and environment. Two of the biggest areas of concern include

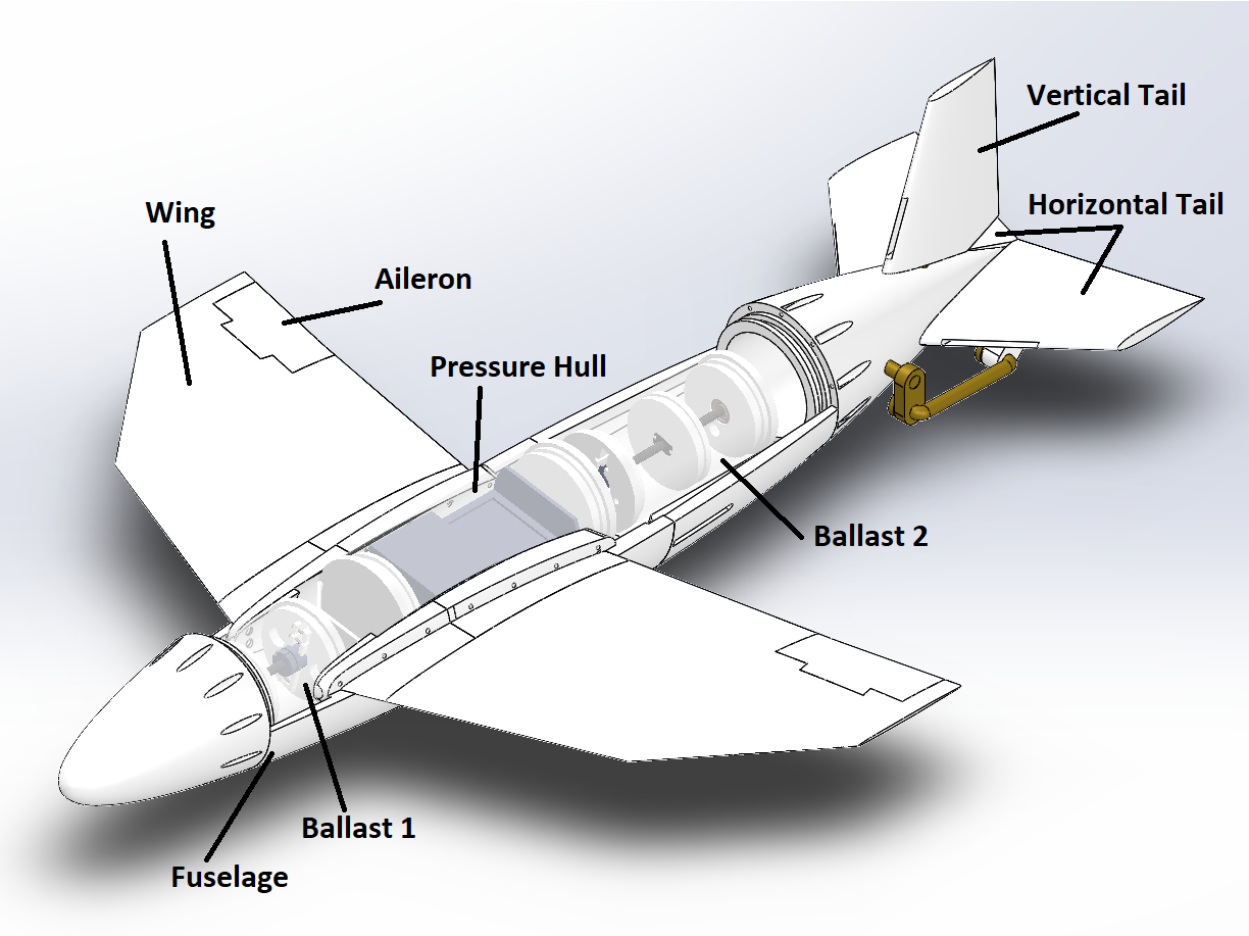


Figure 4: System diagram for mechanical components

the differences in operating medium, as the denser water over air necessitates a differentiation and correction in the uses of in-water weight over weight due to gravity, and the lack of self-propulsion of the craft since it would be towed by an external source, which affects free-body diagrams and necessitate that analysis of the motion of the craft be about the tether point rather than the center of mass.

Much of the craft was prototyped by 3D-printing parts using ABS at 50-60% infill since we were able to break-up the craft into modular components that were printable and allowed for relative ease in making given the limited resources that we had. External parts including the hull and the wing were coated in epoxy as a method of ensuring a water-tight seal for parts individually as well as reinforcement; however, despite much time spent on the design these components, we encountered many difficulties and obstacles during the assembly and integration phase with electrical subsystems, namely the waterproofing at the interfaces between components and unaccounted tolerances due to post-processing of 3D-printed parts.

3.2 Electrical Overview

The electrical system has two main goals: Ensure power is able to flow to all the components, and commands are able to be sent to and from the craft. A diagram can be seen in figure 5 showing the overall connectivity of our electrical system.

Motor specifications were determined based on power and torque requirements derived from mechanical design. Other considered factors included overall size and operational voltage. The motors were the main driver of our power requirements, though the robustness of our power system was less than desired due to severe pressure hull space constraints.

We chose motors operating at 12V with sufficient output torque to drive the ballasts without a gearbox. This allowed us to place the motors on-axis with the ballast and reduce our overall packaging envelope, since a gearbox would have taken significant space. The 12V rating allowed us to use a standard automotive 24V->12V buck voltage regulator, meaning we could run power over the tether at a higher voltage to lessen resistance losses at higher current draw. For our aileron and tail

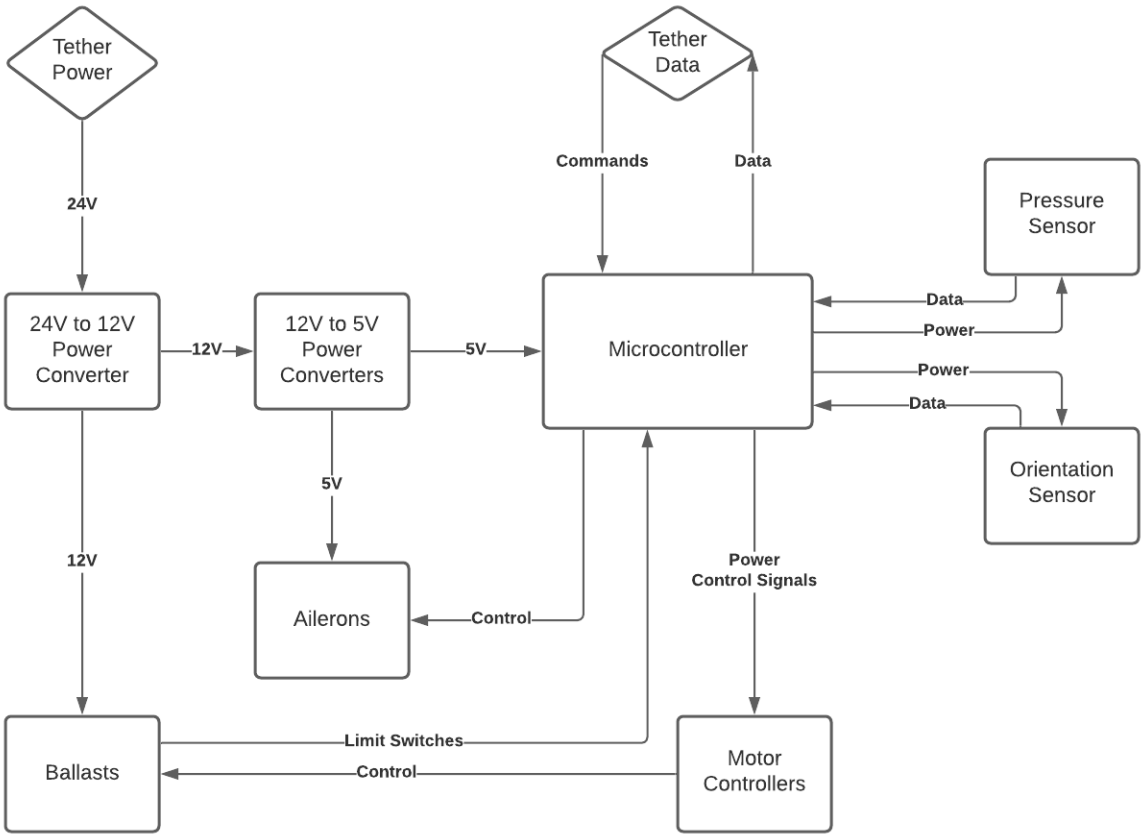


Figure 5: Electrical System Block Diagram

motors, we selected Gobilda 2000-0025-0002 servos for their small size and suitable output torque. Since those operated from 4.8V-7.4V, we selected standard 12V->5V voltage regulators.

The components are vertically stacked - the 24 to 12V power converter and two motor controllers sit on top of a 3D printed divider, which is on top of the microcontroller, and below that are the 12V to 5V power converters. There is almost no vertical or horizontal room left, which compresses the wires. The tether provides data and power through Ethernet, and that Ethernet cable further reduces our space available. Our initial electronics bay used solid core wiring, which led to many instances of the wires breaking. In our final iteration, we used stranded wire, which was far more flexible and easy to work with.

3.3 Software Overview

The Software for this project is divided into two main parts: The Operator Console and the Robot Control System.

Within these two parts are four distinct, yet connected components: The first is the Craft Control component, which maintains the functionality of the craft during operation, and interprets instructions into actions. The second is the Communications component, which ensures smooth and consistent information exchange between the craft's on-board software and a remote server. The third is the Data Management component, which aggregates and visualizes key data about the system for a human operator. The fourth is the Command Interface, which provides an intuitive and accurate method for a human operator to control the craft.

Much of the software for this project is based around providing the ability for a human operator to remotely control the craft. This goal presents numerous challenges, including those of networking, human-computer interaction, data aggregation and visualization, and feedback-based control. To address the various challenges associated with kind of software application, we used several different technologies in tandem.

The software element of the project takes advantage of many well-established technologies. In the design and implementation of each software component, we attempted to use the simplest and

most stable tools to accomplish the necessary requirements. This design also provides the foundation for future improvement of the software as a whole. With the end goal of this project being location of specific elements in and around the seafloor, we selected certain technologies for their capacity for future improvement, and designed the software system to be able to support the addition of new sensors, networking methods, data visualizations, and operator controls as the requirements and goals of the project evolve.

4 System Design and Configuration

The following sections describe overall architecture decisions as well as research and development in specific project hardware or software areas.

4.1 Topside Infrastructure

On the surface of the water, supporting infrastructure for operation of the TSS is housed on a boat, including a tether, power delivery hardware, batteries, and computing hardware for data processing and operator control.

4.1.1 Tether

To connect our robot and the boat, we needed a tether. The tether provides both a mechanical and data link between the two craft - the data link will be discussed in section 4.3.2. To start, we needed to choose a type of rope for the tether. There are many types of rope, and Home Depot provides a comprehensive list [24]. They suggested that polypropylene-based rope would be our best bet. Polypropylene is buoyant, and hollow-braided polypropylene allows us to run the cables through the rope, providing protection and making sure that the cable does not hold any of the towing tension. Furthermore, in our research of other ROVs, we found that SVSeeker, another underwater ROV, used hollow braided polypropylene rope for their tether [25], which further backed up our choice.

4.1.2 Power Delivery

The tether provides power to our vehicle, which requires a large on-boat battery to provide enough power to last more than two hours. This battery sits at a high voltage - and is transferred down to the vehicle through the tether. We decided to use stranded, burial-rated Ethernet cabling, which is water-resistant and allows for repeated bending without breakage. By running a high-voltage, low-current power line, we are able to minimize the power loss due to resistance from small wire diameter in the tether, leading to greater efficiency. Though the tether does heat under high loads, heat is dissipated into the surrounding water.

We chose our battery voltage to be 24V - We could not fit a larger 48V converter in the current iteration of the craft. On board the craft we have 24V->12V and 12V->5V that step down to operational voltage for our motors and electronics.

4.1.3 Infrastructure

The topside infrastructure communicates with the robot via Ethernet as part of the tether assembly. The two systems communicate using a standard UDP client and server configuration. The direct physical connection and elementary networking protocol allows for speedy transmission of data in both directions. Raw sensor data from the robot are transmitted and processed by the data management software component of the topside infrastructure, and control commands from the topside operator console are transmitted to the robot.

4.1.4 Computing Hardware

Given the mostly-software oriented nature of the Topside Infrastructure component of the project, the required hardware is limited. The primary component needed is a laptop to allow for visual information display and user input. This laptop can be outfitted with rugged outdoor / saltwater protection for the environment of a seagoing vessel.

4.2 Mechanical Design

4.2.1 Body Design Parameters

The design of the body of this craft was relatively simple in comparison to the rest of the craft since we knew approximately what we wanted it to look and perform like from the outset. We knew we wanted a long torpedo-like shape for housing all of our payload and ballast, and we knew that it needed to have very low drag, since drag causes more horizontal force on the tether, which results in a longer tether being necessary to achieve the same depth.

Most importantly, it needed to be large enough to contain all of our electronics and ballast. Our first design used a revolved profile of two ellipses to create the shape in Figure 6 for initial approximate simulations.

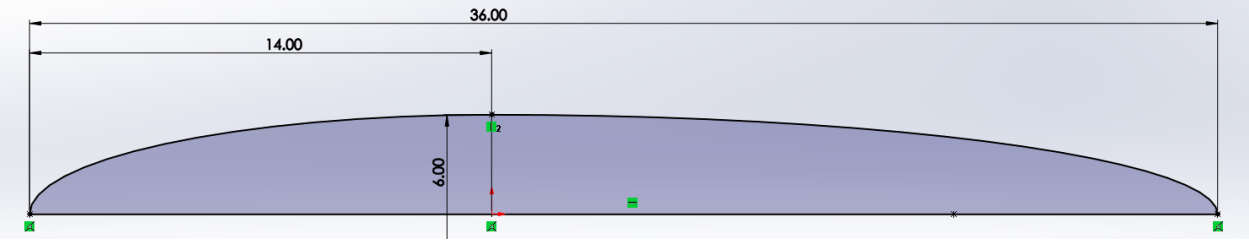


Figure 6: Unoptimized body shape used in initial simulations.

Although this has relatively low drag, it is far from optimized. We decided to turn to conventional wisdom on this aspect and use an extremely low drag NACA 16-006 airfoil, which has a drag coefficient that ranges between 0.007 and 0.1 depending on angle of attack.

In our application, the craft will typically be moving straight forwards through the water at a near zero degree angle of attack, and when ascending and descending will only have a few degrees, so the drag coefficient will be limited to approximately 0.02 or 0.03, plenty low enough to minimize force on our tether. After revolving that profile around its chord, we get the shape shown in Figure 7.

The slightly more pointed nose of the profile reduces drag as it pierces the water, and the pointed tail eliminates flow separation and vortices, a large source of drag in the first design. Additionally, the relatively steep angles on the front and back make it easier to add longitudinal screws, which helps

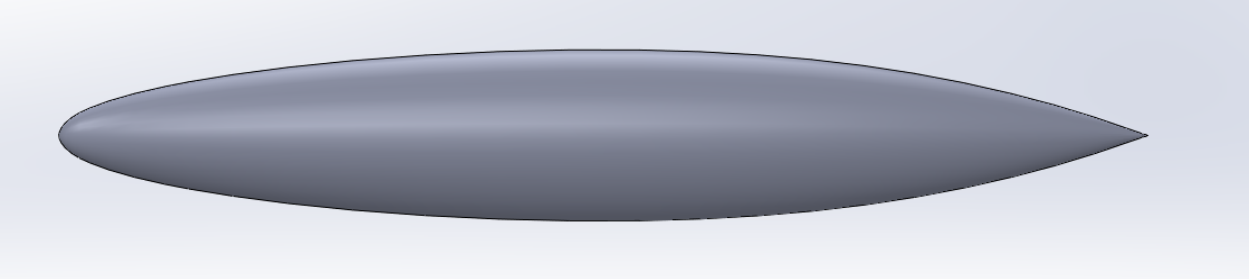


Figure 7: Revolved solid of a NACA 16-006 airfoil profile.

immensely with assembly, as will be shown in section 4.2.8.

4.2.2 Static Analysis

Tether Attachment The tether attachment mechanism to the TSS specifically needs to be able to accomplish the following:

- Quick and automatic detachment when the tether or craft becomes caught/stuck and thus immediate cease of operations is necessary, and
- Allow for water tight bypass of electrical cables from the tether into the hull.

For the initial prototype at this first stage, we did not design a complex tether mounting mechanism for the hull of the craft, and instead focus on the attachment point of the tether to the hull as well as design for bypass of cables into the craft. Placement of the tether mounting point directly affects stability in the towing of the craft when operated, and thus should meet the following design points [23]:

- The mount point should be directly above the center of weight of the craft when submerged in water
- The mount point should be in front of the aerodynamic center of the craft
- The center of mass of the craft should be in between the tow point and the aerodynamic center and closer to the tow point

- The center of buoyancy should be behind the center of weight when submerged in the water in order for the craft to pitch downwards at low velocities and thus generate negative lift to help the craft stay submerged.

The specific placement of the tether mount point is discussed in Section 4.2.7 in context of the design of the control surfaces for the tail.

4.2.3 Wing Design

As far as priorities go, our wing design process went somewhat similarly to our body design process. Again, we primarily focused on drag reduction in order to minimize tether force. In this case, however, we also needed to make sure that the wings would produce sufficient lift force for the craft to be maneuverable in the water and able to dive and surface quickly in order to accomplish basic turns and obstacle avoidance during operation.

Since we knew the hull was going to be a torpedo-shape, we could narrow down our reasonable wing options very quickly. Essentially, our choices were a single long narrow wing down the whole side of the craft or a more airplane style wide winged setup

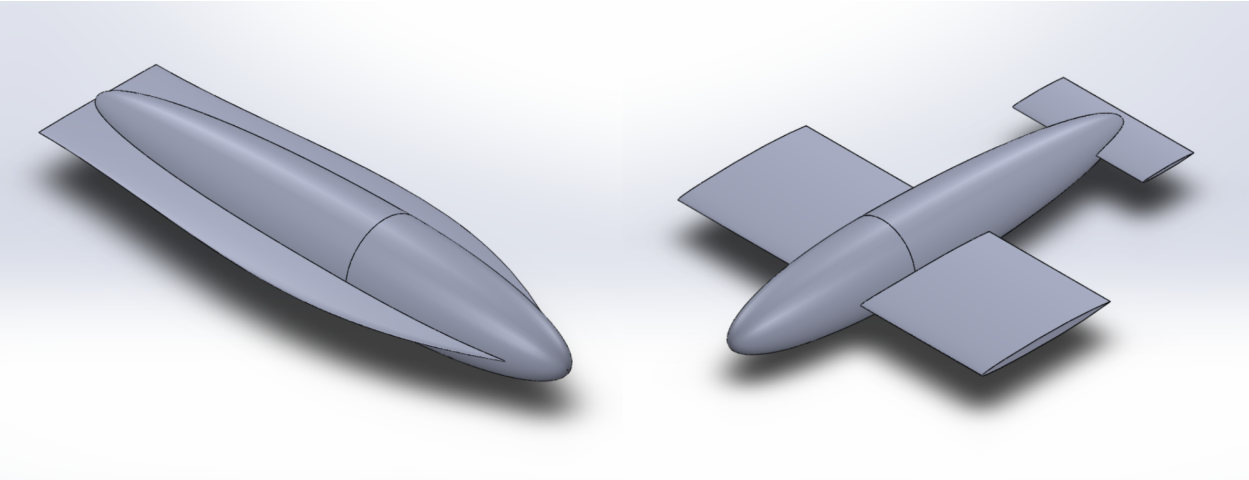


Figure 8: Our two wing design options for simulation.

with a separate tail. These two options are shown in Figure 8. The main advantage of the first, narrower, idea was less drag, but upon simulating both in Ansys Fluent, we found that the second

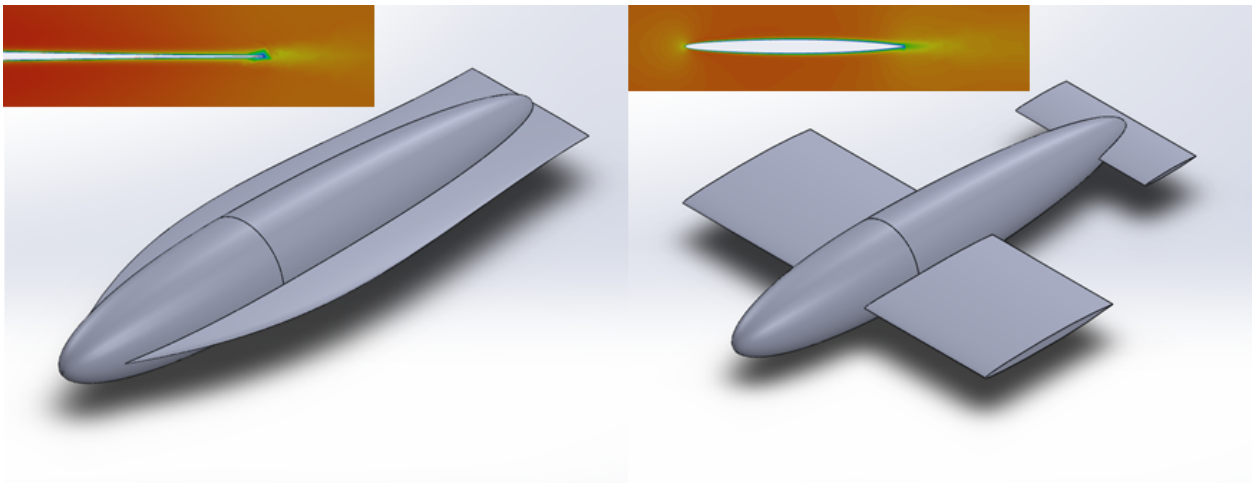


Figure 9: Flow simulation results from a basic Ansys simulation for drag coefficients between two wing design options

actually had a lower drag coefficient, effectively eliminating any perceived advantages of the first. The results and comparison of the flow simulation from Ansys for the two wing designs are shown in Figure 9 and supports the conclusion that the second design is more suitable based on drag forces induced by the wings.

Although we did discuss using an asymmetrical airfoil to create downward lift at a zero degree angle of attack to offset the upward force of the tether, we decided to offset the tether force by aggressive drag reduction and generate upward and downward force purely by changing our angle of attack. Once again, we used the extremely low drag NACA 16-006 airfoil due to its low drag and decent lift at moderate angles of attack. Our initial simulations, which are shown in Figure 10, indicated that at an angle of attack of 5°, the lift force was enough that we were nearly sure it would tear the wings off and would generate massive vortices at the wingtips, so we shortened the wings substantially as well as raking the tips to reduce induced drag [4]. After these changes and adding temporary rudder tailpieces, we arrived at the design shown in Figure 11.

This design has a lift coefficient of 0.1602 at an angle of attack of 5°, which, given a surface area of 0.64 m^2, means that it will generate 319 N (72 lbf) at 5 knots (2.5 m/s) and 3031 N (681 lbf) at 15 knots (7.7 m/s). The high amount of lift force indicates the need to carefully regulate our angle of attack at high speeds lest we apply 600 pounds of force to our wings and tear them off. The drag

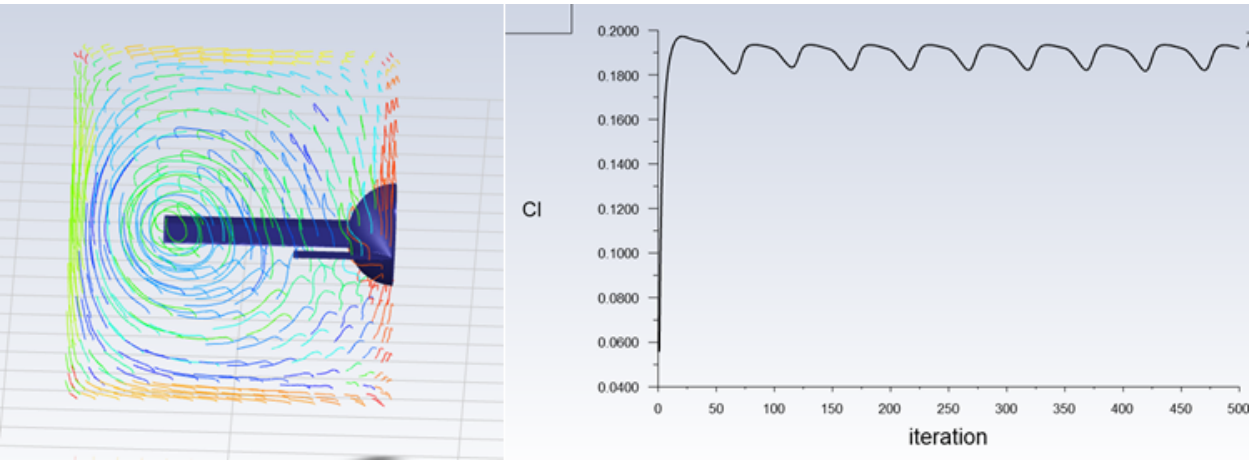


Figure 10: CFD for chosen wing design and simulated lift coefficient

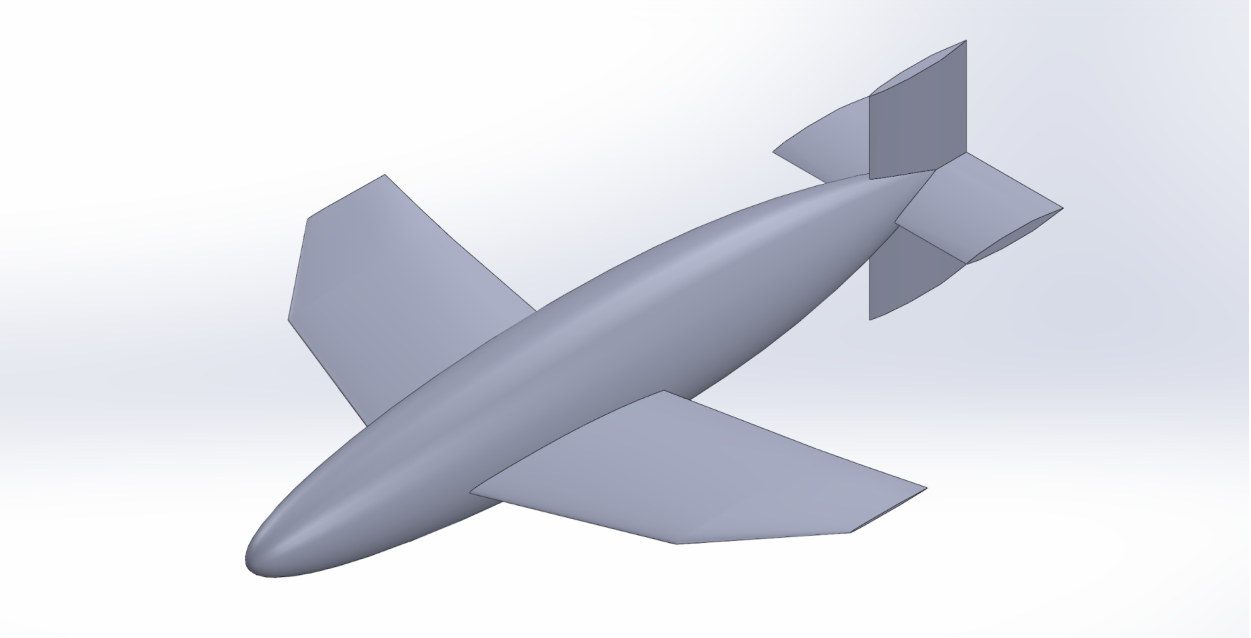


Figure 11: Final body and wing design with temporary rear control surfaces and raked wingtips.

coefficient is 0.0425 at the same angle of attack, which results in 84 N (19 lbf) of drag force at 5 knots and 804 N (181 lbf) at 15 knots. Once again, this shows that at high speeds we need to have a very small angle of attack to prevent that force ever actually being applied. Overall, we can confidently say that this design will produce plenty of lift even at low speeds while having relatively low drag forces.

4.2.4 Pressure Hull/Electronics Enclosure Design

The main purpose of the pressure hull for the TSS is to contain all of the electronics and non-water resistant internal components for the craft in a centralized place. The final design for the pressure hull is shown in Figure 12.

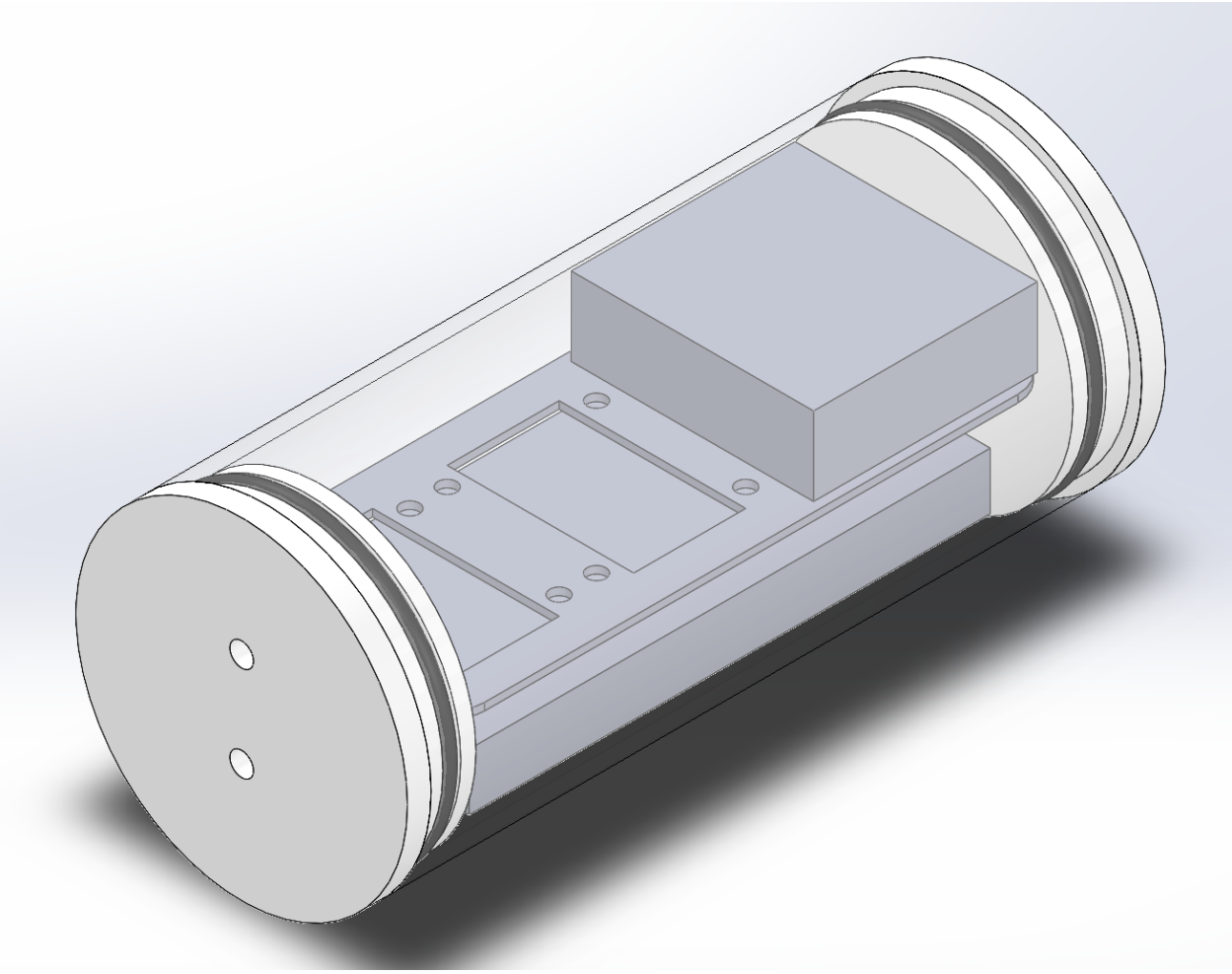


Figure 12: Final pressure hull design

The main components for the pressure hull include the pressure hull tank, two end caps with O-

rings and bypass connectors, and an internal holder platform to organize and separate electronics. Due to the requirement for the pressure hull to be fully watertight in order to protect the internal components contained within the pressure hull, an overarching design goal was to test and ensure that no water could enter the hull in the worst case scenario, which is to be fully submerged. Additional design requirements considered for the pressure hull was that the container needed to be big enough to contain an assortment of internal components for the TSS and not just the basic electronics for controlling the craft, e.g., sensors and additional modular/customizable payload components. Due to the nature of prototyping, the pressure hull also needed to be designed to be easily assembled/disassembled and assessed for both any damage to the hull itself as well as the condition of the components contained within, e.g., status LEDs from the MCU, disconnections from insertion/assembly, leaks, etc. Finally, the pressure hull needs to be able to withstand the worst case environment, which is full submersion in ocean water at extreme depths in the case that the TSS becomes disconnected and the external hull floods/collapses due to water pressures.

Pressure Hull Design The main factors that affect the design choices for the pressure hull container include ease of assembly and assessment, resilience to worst case environments, and size constraints. To meet the first requirement, the pressure hull was desired to be made of a transparent material since it allows for easy debugging and examination of components contained as discussed in the aforementioned examples. This eliminates opaque materials that are traditionally used, such as PVC and metal tubes. We used this criteria alone to narrow down our material choices to three materials: glass, polycarbonate, and acrylic. Glass was removed from the list despite having an ideal density (being the heaviest material out of the three), since glass is be inherently difficult to machine and work with, and sourcing glass tubes of a large length and diameter to meet dimension requirements would be more difficult compared to the other two.

To meet the requirement of resilience to extreme environments, both polycarbonate and acrylic function well under harsh conditions. Both are chemically resistant to saline environments and relatively easy to obtain for a wide range of sizes, and both are able to withstand the water pressure

at extreme depths; for our prototype in the first stage of the project, the deepest depth possible for testing conditions is approximately 50-60ft, which translates to a pressure of 2 atm at most. With the yield strength of acrylic on average 75MPa and polycarbonate being slightly lower at 65MPa, both are able to handle water depths of approximately 7km (pressure increases by an atm for every 10m descended), well exceeding even our overall product requirements. Polycarbonate was ultimately chosen between the two due to costs, since acrylic tubes were about three times more expensive than acrylic ones for the same size. In order to accommodate for the size of the components needed to be contained within the hull, consultation with ECE yielded dimensions of 8" in length and OD/ID of 3.5"/3.25" to be sufficient for containing all of the necessary electronics needed for the TSS for at the very least the elements required for control and basic sensors for the craft. The size of the hull was also ensured to fit within the external hull of the craft.

End Caps The main two design constraints for the end caps of the pressure hull include being easy for assembly and maintaining water tightness under all circumstances. To create the endcaps, we decided to 3D print them for rapid prototyping and ease of accessibility to the tools needed under COVID restrictions. To obtain the level of waterproofing necessary for the pressure hull, we relied on three techniques: selection of filament material, printing method, and post processing.

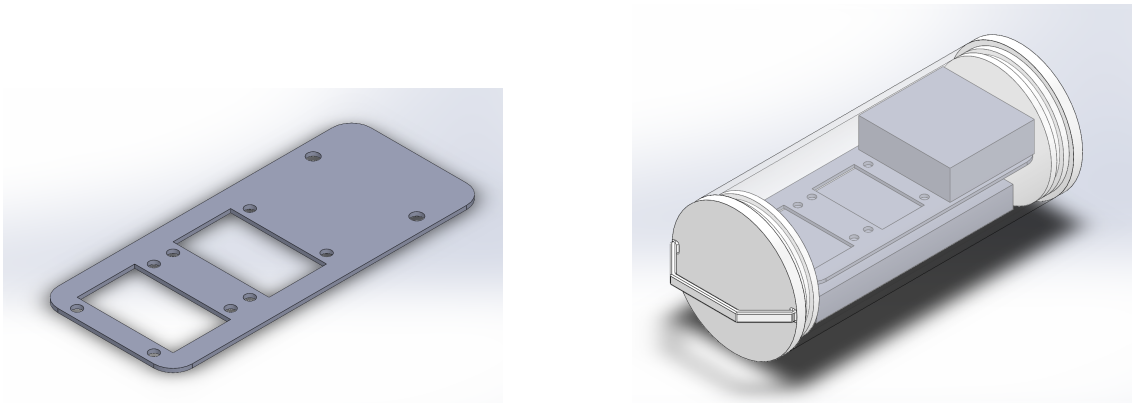
Based on online research of material commonly used for watertight applications, the four different filaments were weighed against each other, which is summarized in the following table. Mechanical properties were taken from the Matweb database [10, 9, 8, 7].

From the list of materials in Table 1, both PP filaments were immediately eliminated, since PP is less dense than water while PP GF30 is too expensive and had properties that exceeded our requirements by more than necessary. Between ABS and PETG, while PETG overall has more desirable properties in terms of density, water absorption, and ease of use, ABS was selected initially due to its availability and largely due to its ability known compatibility with post processing for waterproofing.

Specifically, post processing techniques were needed in order to prevent leakage through the 3D printed material, since the layers of a 3D printed part inherently contain microscopic gaps that

Table 1: Characteristic comparison between 3D printing material

Material	Properties
Polypropylene (PP)	• Commonly used for food packaging/bottles/containers • Hydrophobic • One of the most chemically resistant 3D printing materials • Flexible • Density: 0.915g/cc • Water absorption: 0.0109% • Tensile strength: 32.6 MPa • Cost: $50/500g
PP GF30	• 30% glass fiber reinforced PP • Extremely resistant to highly aggressive environments such as high heat, UV, and chemicals • Glass fiber reinforcement adds rigidity/strength • Density: 1.14g/cc • Water absorption: 0.0258% • Tensile strength: 77.6 MPa • Cost: $100+/700g
PETG	• Easiest to print • Impact resistant • Slightly flexible • Poor processing applications • Density: 1.26 g/cc • Water absorption: 0.146% • Tensile strength: 48.2 MPa • Cost: $20/1000g
ABS	• Difficult to print • Long history in use for 3D printing • Known compatibility with water proofing post processing techniques • Density: 1.07 g/cc • Water absorption: 0.410 % • Tensile strength: 44.8 MPa • Cost: $20/1000g



(a) Holder design (b) Holder within pressure hull

Figure 13: Holder design for electronics in pressure hull

would allow for water to leak through. To counter this effect, we selected the commonly used post processing epoxy XTC-3D as it was well documented to work with ABS and known to ensure water tightness in the part.

Additionally, we needed a reliable and non-permanent sealing method to prevent water from entering the hull through the endcaps. To achieve such sealing, we used a custom-made O-ring using O-ring cord stock bought from McMaster-Carr. The O-ring width and corresponding gland was spec'd according to the SAE AS568 standard listed in the Parker O-ring handbook; since the standard requires backup rings for only when pressures exceed 1.5 ksi, only a single O-ring was integrated in the design for the end caps [5]. The O-ring material was selected to be EPDM, as it was cheaper than silicone cord stock and is known to be resistant in salt water applications.

Internal Electronics Holder In order to allow for organization and prevention of short circuiting of the electronics placed inside the pressure hull, we designed an internal holder for the various electronics and other components shown in Figure 13. The holder fastens the two motor controllers needed for the ballasts as well as the power converter to step down the voltage from 24V delivered over Ethernet to the 12V used to drive the motors.

4.2.5 Ballast Design

The ballast of the TSS plays a crucial role in its operation, since it allows for general control over the attitude of the craft and thus gives it the ability to ascend and descend in the water. The ballasts in the TSS mainly control the attitude of the craft when it is at rest, i.e. when it is not being towed by the boat; during towing operation, the operator can actuate the control surfaces in order to adjust the position of the craft in the water, and therefore the ballasts are used to ensure that the craft can be neutrally buoyant during the operation phase and resurface on its own for retrieval.

Ballast Operating Principles The main principle in concern for the ballast is the amount and distribution of buoyancy that the craft carries. Due to the fact that the inside of the craft will be contained with mostly air, the craft will be naturally buoyant in water since the weight of the water displaced by the craft will be greater than that of the craft. The buoyant force for the craft is calculated simply via the following equation:

$$\Delta B = \frac{F_B - F_G}{g} = M - \nabla\rho \tag{1}$$

where ΔB is the net buoyancy, F_B is the buoyant force, F_G is the weight of the TSS, g is acceleration due to gravity, M is the total mass of the TSS, ∇ is the displacement volume of the TSS, and ρ is the density of the displaced medium, in our case salt water, which has a density of approximately 1025 $\mathrm{kg/m^3}$, or freshwater, which has a density of 1000 $\mathrm{kg/m^3}$.

In order to control the buoyancy of an object in water, two parameters can be changed, which are the mass of the object or the volume of water that it displaces. Since in our case the volume of the main body of the craft must be held constant and can't be varied easily, the focus of the ballast is to consequently adjust the amount of mass that is on board the TSS. Additionally, dense weights such as lead shots can be used to offset the mass of the TSS, thus aiding the ballast by reducing the capacity of water needed for successful control of the attitude of the TSS.

It should be noted here that while the ballast is used to control attitude for the TSS, its main

purpose is simply to allow for a general method for the TSS to ascend and descend easily and independently, not to maintain an attitude. Since the ballast is able to adjust the amount of mass on-board the TSS and thus its buoyancy, the TSS can be adjusted to be either positively, neutrally, or negatively buoyant, where in each state the TSS will ascend, maintain attitude, or descend respectively. In the neutrally buoyant state, the craft will experience zero buoyant force; however, due to the external forces of drag and tension from the tether acting on the TSS, independent control surfaces such as in the form of ailerons and elevators are necessary for counteracting these forces in order to maintain a constant depth.

Ballast Driving Mechanism As stated before, the two primary parameters that can be controlled by the ballast to adjust buoyancy for the TSS is its mass and/or its volume. Based on prior ballast designs for ROVs and AUVs, three different types of ballasts have been known to work for such vehicles, which are as follows [2]:

1. Enclosed: a bidirectional pump in conjunction with valves and a plumbing system allows for water to be let in and out of a ballast tank.

2. Piston: a motor uses linear actuation to drive a piston head back and forth in a ballast tank, where when the piston head is driven backwards, water is sucked into the tank and when the piston head is driven forwards, water is expelled

3. Flexible: water is pumped in and out of a balloon or other flexible apparatus and is located outside of the TSS rather than being internally contained within the craft

As stated prior, the flexible option was immediately eliminated due to the concerns about reliability and effects on the operation of the TSS, since such a ballast has a possibility of bursting and would negatively affect the drag of the craft when being towed. In order to narrow down between a piston or pump design, we used the decision flow chart provided by Tiwari and Sharma [22] and is shown in the following figure:

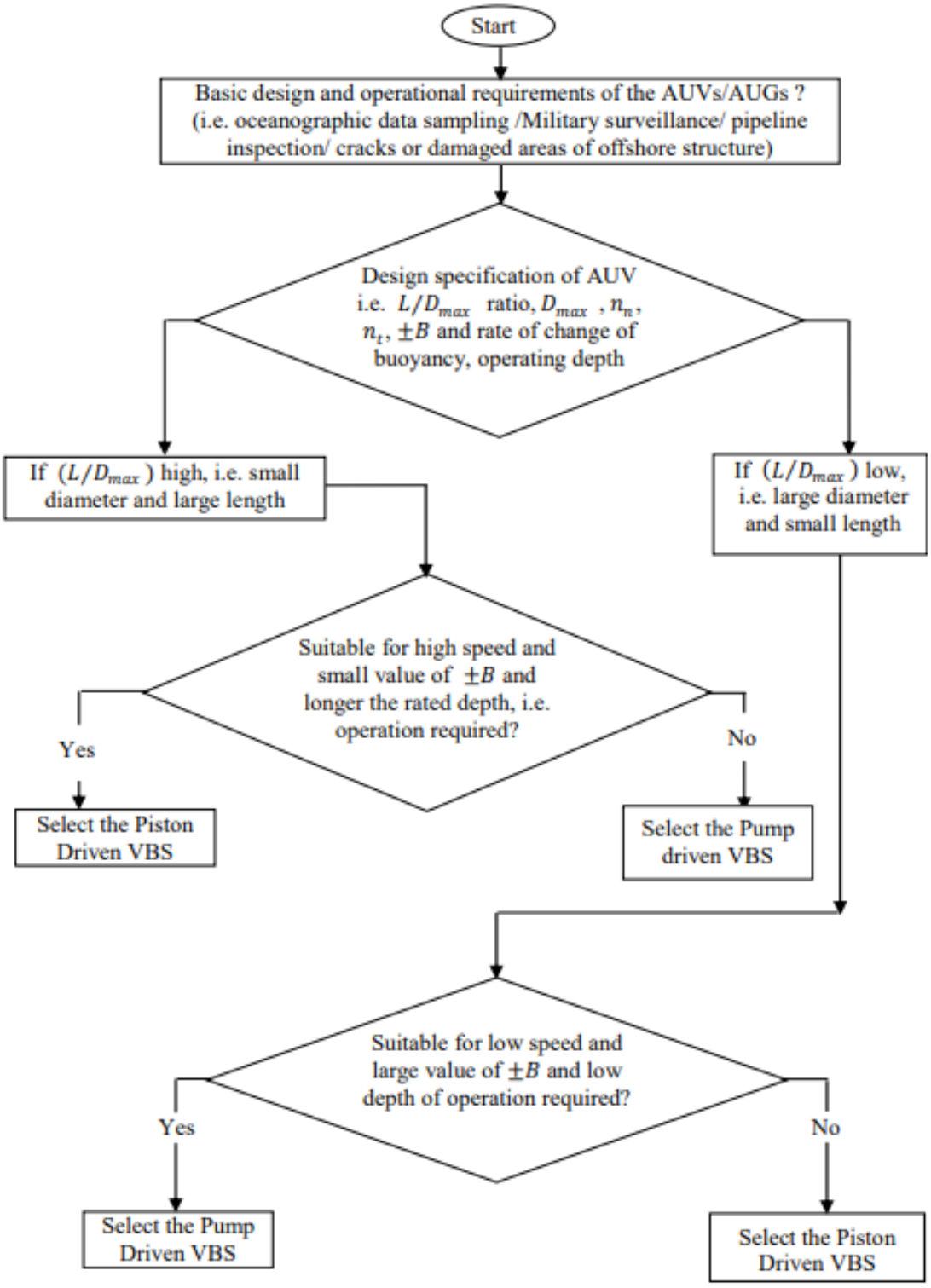


Figure 14: Decision flow chart for ballast mechanism selection

Since the general shape of our TSS has a long and thin geometry and thus a high L/D ratio, and is operated for long periods and at fast speeds, we determined that a piston driven variable ballast system (VBS) would be most suitable for our needs. The main components for a piston driven VBS is a cylindrical ballast tank, the piston head and complementary linear actuator, as well as the intake line for the water to enter and exit the ballast tank. Furthermore, in order to aid in pitch stability, we used two ballast tanks to help distribute the load from the ballast to be between the head and tail of the body.

The size of the front and rear ballast tanks are 2.75” in diameter and 5” in length each, which gives a total ballasting capacity of approximately 1.75lb. The final design is shown in Figure 15.

We decided to use two ballasts instead of a single one for two reasons: first, by having two ballasts at the front and rear, we could independently shift the weight distribution of the craft about its center of mass, which helps to improve intrinsic pitch stability; second, by having ballasts fitted at the ends of the craft, we could centralize the electronics into a single pressure hull at the center of the craft rather than broken up into separate containers distributed throughout the craft, therefore minimizing places for potential failure and resources required. The isometric and side profile views of the ballasts in the craft are shown in Figure 16.

Due to space constraints from size of the hull unique to our first stage prototype, we had to find an alternative method to actuate the piston head, since the traditional method with the lead screw would require space outside of the ballast tank where the lead screw could fully extend when the head is retracted to draw in water. To reduce the size footprint of the ballast system, we considered three different designs: a smooth rod driven by angled bearings, a telescoping/compressive design, and a bypass lead screw design. Examples of the angled bearings and telescoping system are shown in Figure 17.

Since the piston driving mechanism needs to be able to handle the pressure from the water in the tank to maintain constant ballast capacity at all times, the smooth rod and bearing system was eliminated. The telescoping mechanism was eliminated due to its space efficiencies and complexity in the mechanism, since a large portion of the tank needs to be sacrificed even in the compressed

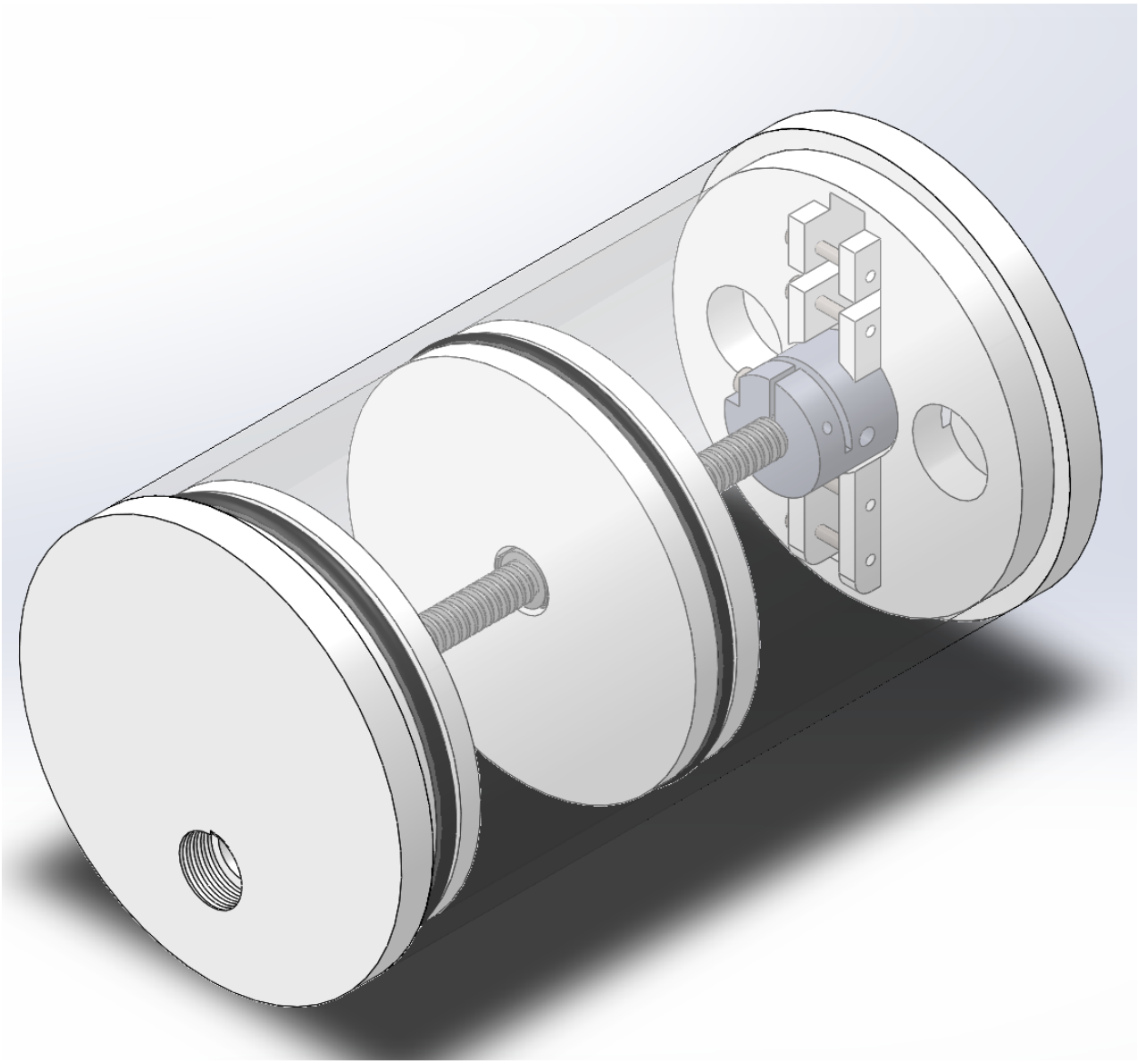
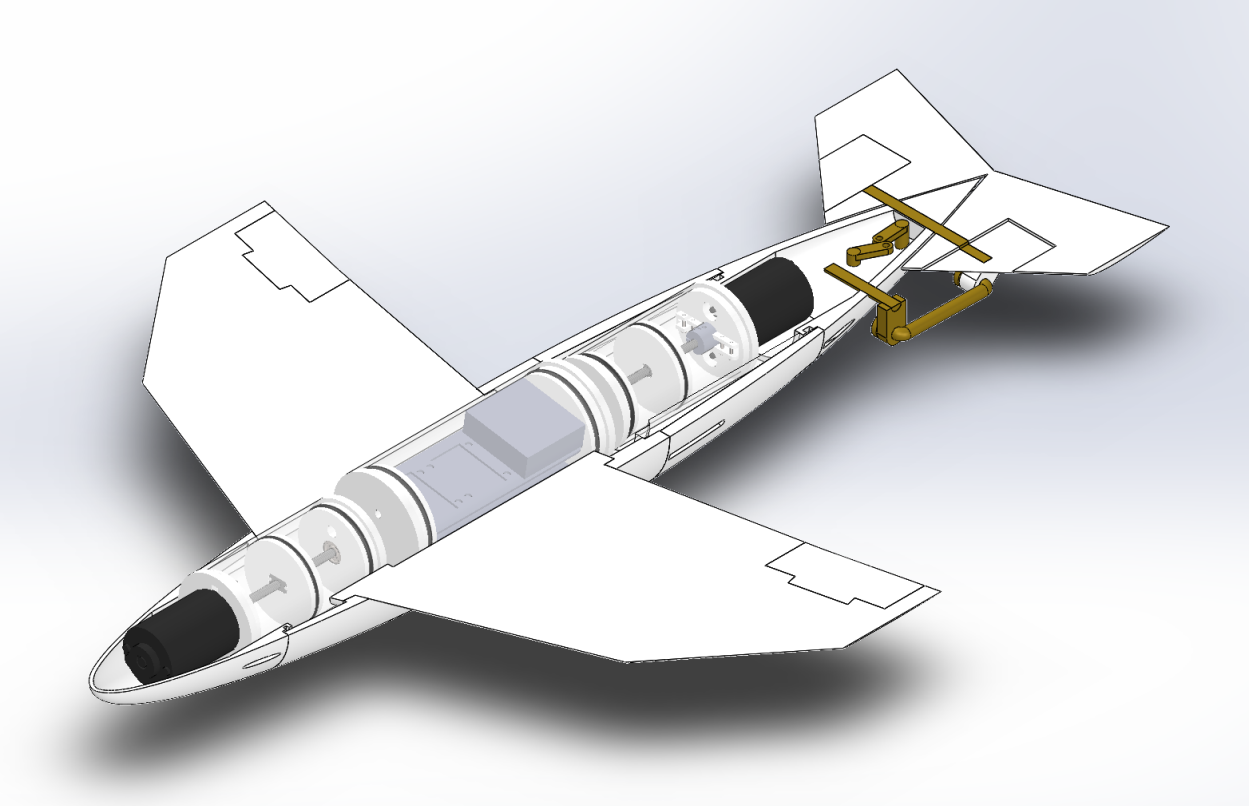
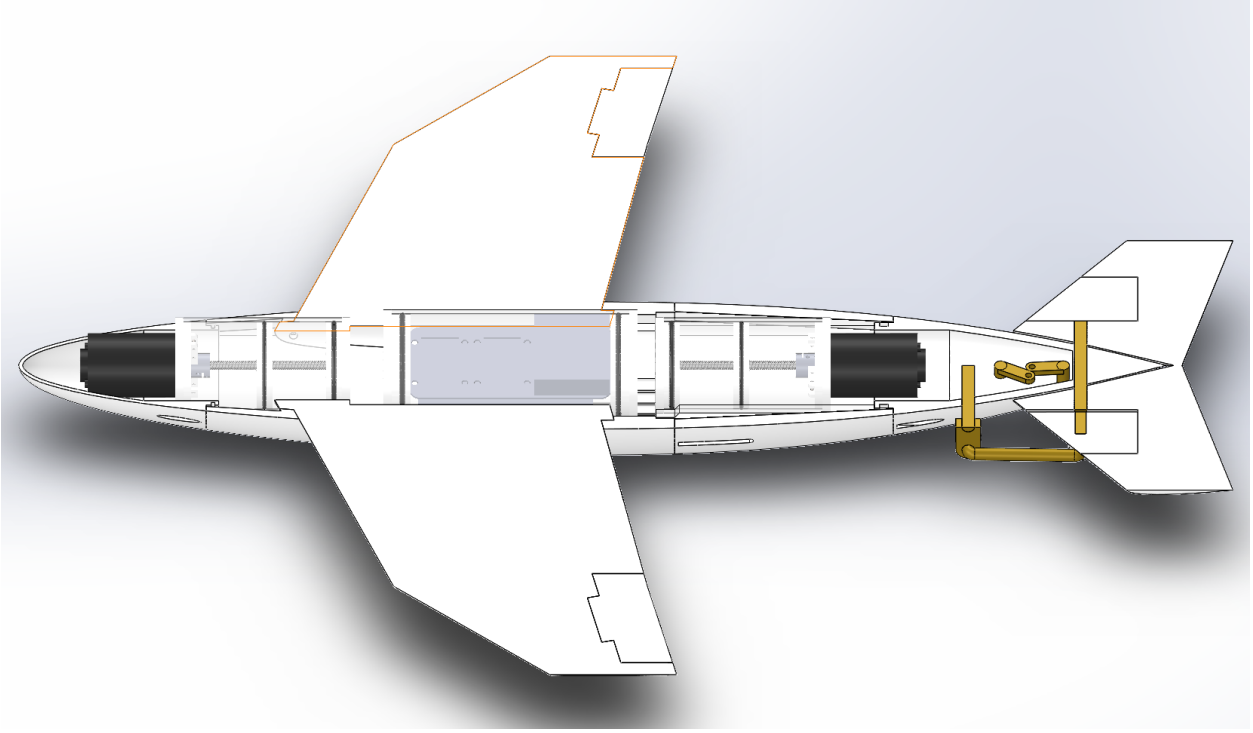


Figure 15: Final ballast design

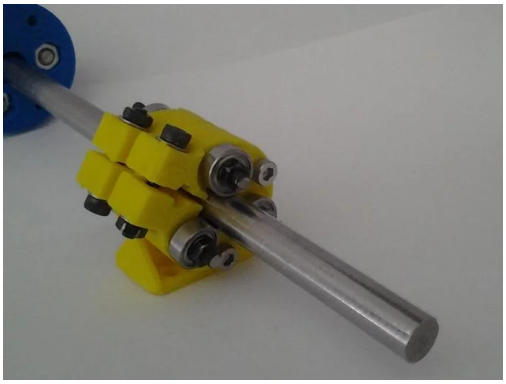


(a) Isometric view of ballasts in craft

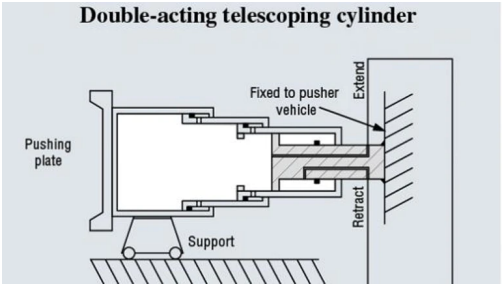


(b) Side profile view of ballasts in craft

Figure 16: Dual ballast system embedded in craft



(a) smooth rod and bearing [21]



(b) telescoping mechanism [19]

Figure 17: Alternative piston driving mechanisms

form, and the increased amount of parts in the system entails additional complications in creating a reliable system. Thus, a bypass lead screw design was chosen.

The main concern for the bypass lead screw design lies in maintaining a watertight seal at the interface between the piston head and the lead screw that passes through. In order to accomplish a watertight seal, we relied on two methods, which were 1) using an extremely tight tolerance leadscrew and flange to form a labyrinth seal, and 2) using lubrication to fill in the remaining space between the flange and the leadscrew. The use of lubrication was inspired from the original design use of a labyrinth seal to retain lubrication in machines, and works well in our case since the use of the same marine grease used to lubricate the O-rings was viscous enough to prevent water from leaking through the path along the leadscrew. To test the sealing interface, we filled a water bottle with water and had the leadscrew-flange attached to the cap so that water would only pass through if there was a leak through the leadscrew as shown in Figure 18.

Although our initial test without the lubrication resulted in continuous leakage, after lubrication was run along the entire leadscrew, the flange was completely effective in sealing the interface after a 12 hour test with the bottle sitting cap side down. Once the actuation for the ballast was complete, we moved forward with prototyping our design. The final assembled prototype is shown in Figure 19.

The ballast tank was selected to be made of a PETG cylinder with ID 2.75” and OD 3”, since the mechanical properties of PETG are more desirable compared to acrylic in regards to its higher density

Figure 18: Test for leakage through leadscrew flange combination

and yield strength. The leadscrew and flange nut were their ultra-precision parts with diameter of 7/32”-20.8 thread size and 4 starts. The leadscrew and flange nut are made of 303 stainless steel and acetal plastic respectively in order to withstand the aquatic environments.

To control the ballasting, two limit switches were placed on the end cap in the holes seen on the top and bottom of the right-hand side end cap in Figure 15. One limit switch is placed with the switch facing inwards to limit the full ballast position, as the piston head will press the switch once retracted all the way. The other limit switch is controlled by a wire attached to the piston head in order to limit the piston head and signal the empty state. The wire is connected to the limit switch facing outwards away from the piston head and measured so that once the piston head reaches the front end of the ballast, the wire pulls on the switch and thus signals the empty state.

To determine the actuation force required to move the piston head, we first calculated the thrust required to move the piston head at the maximum depth, which is where pressure on the piston head would be the highest. The thrust force required is given by the following equation [22]:

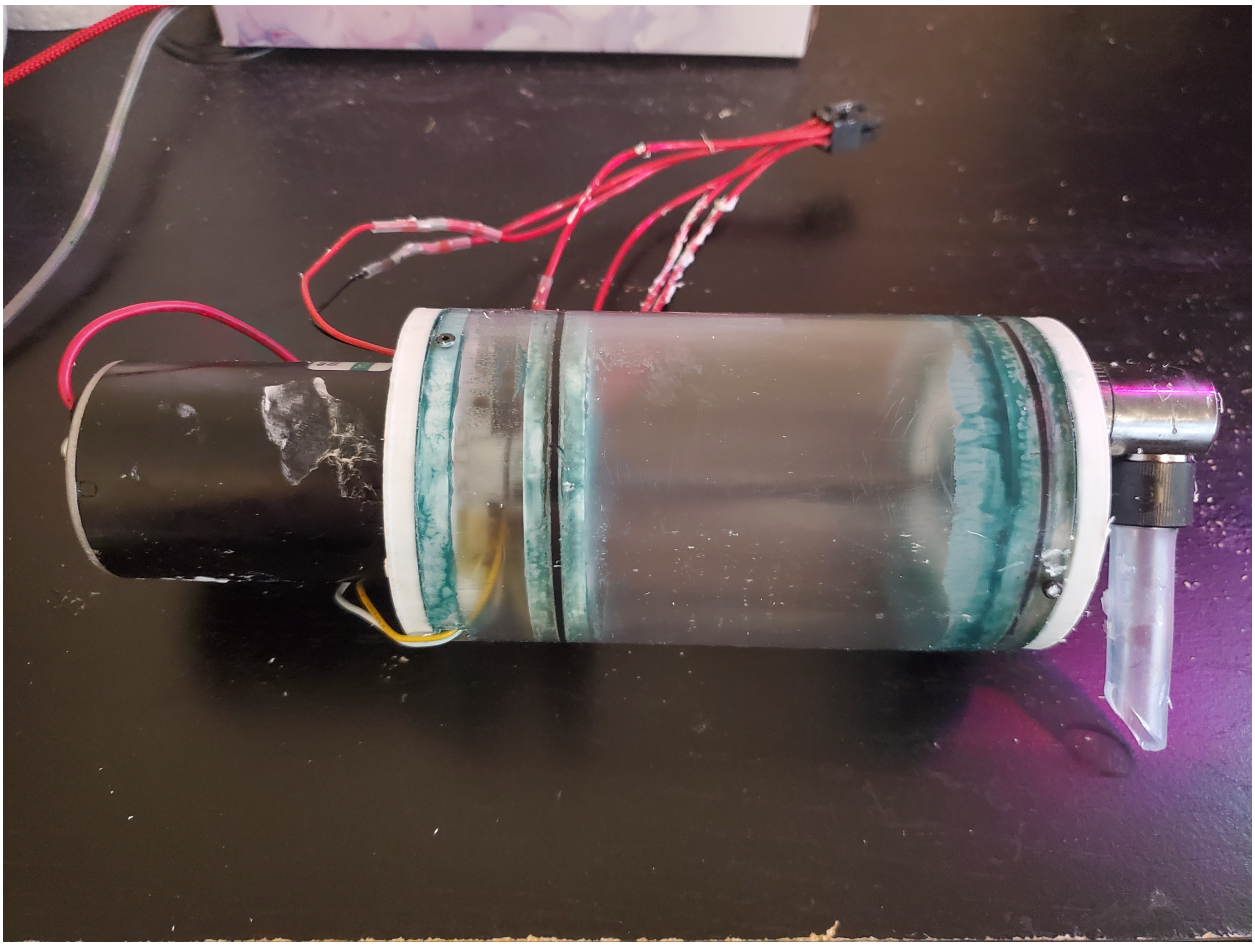


Figure 19: Prototype of ballast

$$T = A_{cs}\Delta P = (\pi/4)D_p^2\Delta P = (\pi/4)D_p^2((\rho g h + P_{atm}) - P_{in}) \quad (2)$$

where P_{in} is pressure inside the ballast tank, P_{atm} is atmospheric pressure, h is submersion depth, and g is acceleration due to gravity. With the first semester prototype operating at a maximum depth of approximately 50ft, an internal pressure of atmospheric pressure, and a piston head diameter of 2.75in, we can expect approximately 130lbf of thrust required to actuate the head.

Then, using the axial force equation provided by SDP/SI's guide on leadscrew calculation, we can calculate the required torque for the motor to drive the piston head [18].

$$\text{Force} = \frac{2\pi}{16} T p \eta \quad (3)$$

where Force is the load on the flange nut, T is the torque exerted on the lead screw by the motor, p is the threads per inch of the lead screw, and η is the lead screw efficiency. Using 130lbf for the force, 20.8 for the tpi, and 0.65 for the lead screw efficiency based on the median for acme rods with plastic nuts [20], we arrive at a torque required of 70oz-in. To allow for a safety margin of operation of 2 to account for greater leadscrew inefficiencies, friction at the piston head, coupler inefficiency between the lead screw and the motor, and other losses or factors not accounted for in our calculations, we double the torque requirement for the motor to be approximately 150 oz-in.

Additionally, for the motor requirements of the ballast, we need to determine the rpm that it can run at. Since the piston head travels 4.25" from front to back in the ballast and we desire for the ballast to be able to go from full to empty in about 5 seconds, we can calculate the rpm as follows:

$$\omega_{rpm} = \frac{Lp}{nt} * 60 \quad (4)$$

where L is the length traveled, p is tpi, n is the number of starts, and t is travel time. With a length of 4.25", 20.8 tpi, 4 starts, and 5 seconds of travel time, we determine a rpm requirment of approximately 270rpm for the motor.

4.2.6 Aileron Design

To allow control over the movement of the craft in cases such as basic roll stabilization or object avoidance, we needed to design ailerons for the wings along their tips in addition to the rudder and elevator at the rear. The final design of the aileron and the linkage needed to control it are shown in Figure 20.

Since the goal of the design of the aileron was to establish a rough baseline for future iterations to fine tune through testing, dimensions and parameters of the aileron are based on average specifications commonly found for ailerons from [16], which includes a aileron chord to wing chord ratio of 0.25 and an inner and outer span ratio of 0.65 and 1.00 respectively. The aileron is designed to be actuated for a maximum angle of attack α of $-30°$ to 30°. The material of the aileron is consistent with the rest of the craft and is 3D-printed with ABS with 100% infill, while the links are all machined out of brass. The key parameters of the aileron are listed in Table 2.

Table 2: Parameters for aileron design

Aileron parameter	Value
$S_a/2$	41.12 cm^2
$b_a/2$	10.04 cm
$b_{ai}/2$	24.96 cm
$\overline{C_a}$	4.10 cm
$\delta_{a_{\max}}$	+/-30°

To actuate the ailerons themselves, we use a 4-bar linkage mechanism, where a shaft that runs throughout the wings is directly connected to and rotated by a servo embedded in the wings closer to the body of the TSS. At the end of the motor shaft is a coupler that connects the motor shaft to the aileron link, which is then connected to arm under the aileron. The hinge point where the intermediate aileron link attaches to the aileron is directly underneath the hinge for the aileron itself to allow for

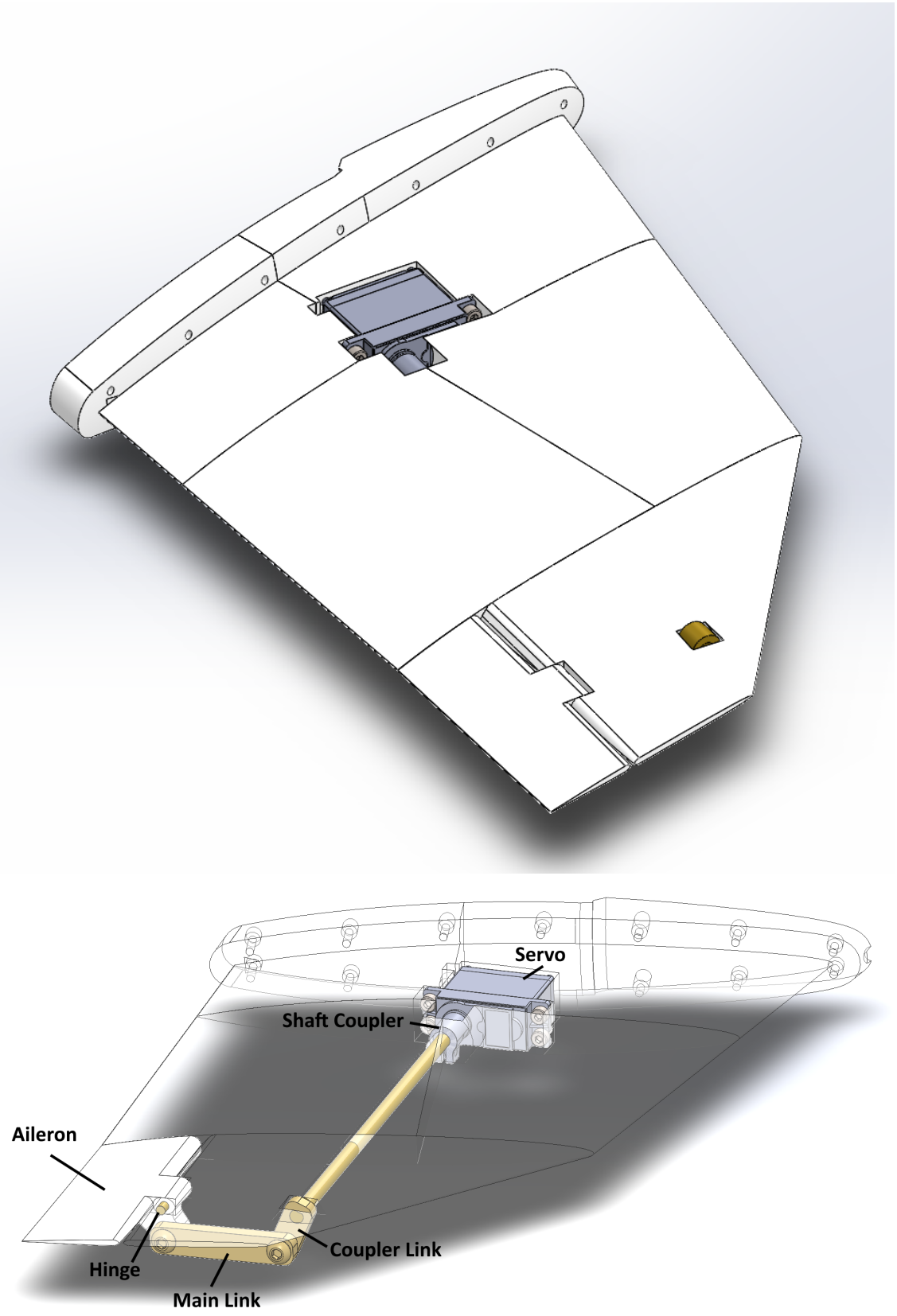


Figure 20: Aileron Subsystem Diagram

equal changes in rotation between the servo and the aileron. This design was the simplest design that was feasible for our wings due to the increasing thinness moving out from the body.

To determine the amount of torque required for the servo to actuate the aileron to the maximum deflection angle of +/-30°, we use calculated the hinge moment of the aileron using the equations provided by Sadraey [16], which is given as follows:

$$H = \frac{1}{2}\rho U^2 S_c C_c C_h \tag{5}$$

$$C_h = C_{h_o} + C_{h_\alpha}\alpha_{LS} + C_{h_{\delta c}}\delta_c \tag{6}$$

where H is the hinge moment, ρ is density, U is the velocity, S_c and C_c are the planform area and mean aerodynamic chord (MAC) of the aileron respectively, C_h is the hinge moment coefficient, C_{h_o} is the hinge moment coefficient due to the pitching moment, C_{h_α} and α_{LS} are the hinge moment coefficient and angle of attack of the lifting wing, and similarly $C_{h_{\delta c}}$ and δ_c are the hinge moment coefficient and angle of attack of the aileron. Since α_{LS} of the wing is 0 and the wing is symmetric, and using the typical value -0.3 rad^{-1} for $C_{h_{\delta c}}$ and a maximum deflection angle of 30°, we calculated an approximate hinge moment of 0.33 Nm for the aileron. We then double the moment required as a safety factor to ensure that the servo motor actuating the aileron can produce enough torque for maximum deflection, yielding a final torque requirement of 0.66 Nm for the aileron servo motor.

Since this torque requirement is only 30% of the rated torque of the selected servo motors, the aileron links were chosen to be about a 1:1 overall ratio, providing a large factor of safety and minimum footprint.

4.2.7 Elevator and Rudder Design

In addition to the ailerons to control roll, we needed additional control surfaces in order to stabilize and control the pitch and yaw of the craft, which can be accomplished by actuating an elevator and rudder at the rear of the craft. The majority of the design process and calculations for these control surfaces is again based on Sadraey's textbook on aircraft design with necessary adjustments to

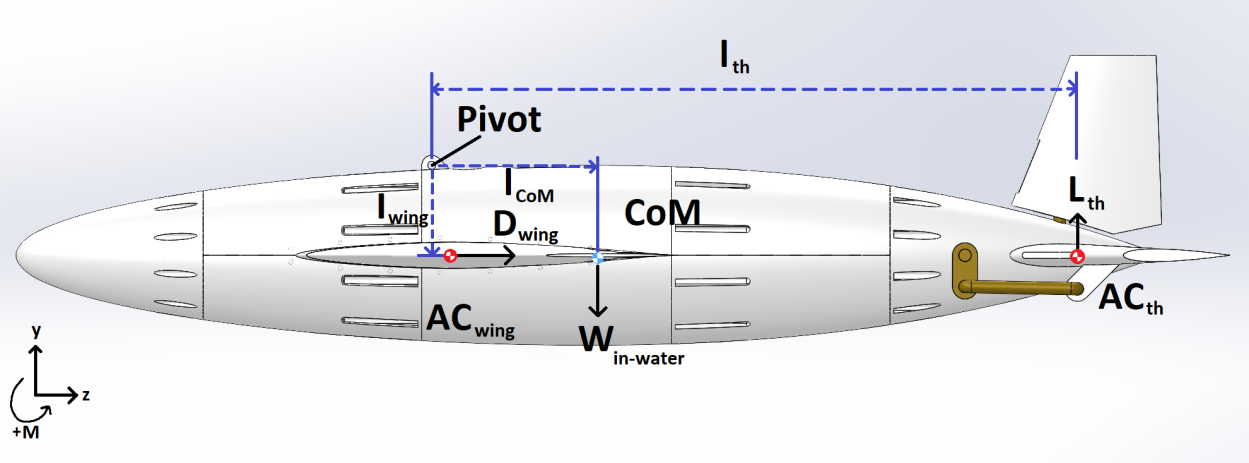


Figure 21: Free body diagram of the craft for design of the horizontal tail

reflect the account for the physics of a towed craft underwater [17].

Horizontal Tail In order to control and stabilize the pitch of the craft as it was being towed, we designed the horizontal tail to act as both a tail and a rudder. The parameters of the tail are chosen and calculated based on steady-state operation when the craft has been submerged to the desired depth and is simply moving forward at a constant depth. Figure 21 shows the free body diagram (FBD) with the key forces needed to determine the location and size of the tail.

In the FBD, pivot is the pivot point where the tether attaches and tows the craft, l is the respective distance between points of interest, AC is the aerodynamic center, D is the drag on the wing, W is the weight of the craft when it is submerged, L_{th} is the lift generated by the horizontal tail, and CoM is the center of mass of the entire craft. Since the wing has a symmetric airfoil and is fixed at an angle of attack of 0°, we omit the pitching moment at the wing's aerodynamic center as well as the pitching moment for the horizontal tail since it is relatively small compared to the other forces acting on the craft.

The location of the horizontal wing was determined by selecting a ratio of 0.55 between the distance from the wing aerodynamic center to the horizontal tail aerodynamic center and the aircraft overall length from Table 3, since the configuration of our craft most similarly resembles the conditions of number 2 where the tether force corresponds to the thrust force generated by the engine

Table 3: Typical values for l/L for various aircraft configurations from Sadraey [17]

No.	Aircraft configuration/type	*l/L*
1	An aircraft whose engine is installed at the nose and has an aft tail	0.6
2	An aircraft whose engine(s) are installed above the wing and has an aft tail	0.55
3	An aircraft whose engine is installed at the aft fuselage and has an aft tail	0.45
4	An aircraft whose engine is installed under the wing and has an aft tail	0.5
5	Glider (with an aft tail)	0.65
6	Canard aircraft	0.4
7	An aircraft whose engine is inside the fuselage (e.g., fighter) and has an aft tail	0.3

above the wing and our a craft possesses an aft tail.

Since we conduct our calculations assuming steady state operation, then the moments about the pivot point must be balanced. The summation of these moments is derived using the FBD and is illustrated in the following equation.

$$\Sigma M_{\text{pivot}} = D_{\text{wing}} l_{\text{wing}} - W_{\text{in water}} l_{\text{CoM}} + L_{\text{th}} l_{\text{th}} = 0 \tag{7}$$

The lift and drag of the wing and horizontal tail can be calculated using the standard lift equation

$$L = \frac{1}{2} \rho V^2 S C_L$$

where ρ is the density of the medium, V is the velocity of the craft, S is the planform area of the aerodynamic body, and C_L is the lift or drag coefficient. We assumed a velocity of approximately 4.5 m/s, as the first, scaled down prototype would unlikely reach the full speed of 10 m/s in our original objective. However, since both the lift and the size of the horizontal tail is both unknown early on in preliminary design, we had to select a typical volume coefficient from Table 4. Since our application most closely resembles the motor glider listed in the first row of Table 4, we selected a volume coefficient $\overline{V}_H$ of 0.6.

Table 4: Typical values for horizontal and vertical tail volume coefficients from Sadraey [17]

No.	Aircraft	Horizontal tail volume coefficient ($\overline{V}_H$)	Vertical tail volume coefficient ($\overline{V}_v$)
1	Glider and motor glider	0.6	0.03
2	Home-built	0.5	0.04
3	GA single prop-driven engine	0.7	0.04
4	GA twin prop-driven engine	0.8	0.07
5	GA with canard	0.6	0.05
6	Agricultural	0.5	0.04
7	Twin turboprop	0.9	0.08
8	Jet trainer	0.7	0.06
9	Fighter aircraft	0.4	0.07
10	Fighter (with canard)	0.1	0.06
11	Bomber/military transport	1	0.08
12	Jet transport	1.1	0.09

Using the volume coefficient of 0.6, we can calculate the planform area of the wing with the following equation:

$$S_{th} = \frac{\overline{V}_H \overline{C} S_w}{l} \tag{8}$$

where $\overline{C}$ is the mean aerodynamic chord (MAC) length of the wing, S_w and S_{th} are the planform areas of the wing and horizontal tail respectively, and l is the distance from the wing aerodynamic center to the horizontal tail aerodynamic center. Once the planform area of the horizontal tail was determined, the lift coefficient required from the horizontal tail as well as the remaining parameters could be calculated. Additional parameters related to the size of the wing include the aspect ratio, which was taken to be simply 2/3 of the wing's aspect ratio, the taper ratio, which was arbitrarily selected to be 0.5 to fit in the range of typical values for transport aircrafts, and the sweep angle, which was chosen to be the same as the sweep angle as the wing. Finally, based on the recommendation of Sadraey, we simply selected the NACA 0012 airfoil, as the specific airfoil at this stage in design is not critical and can be iterated based on results from testing later on. Table 5 lists all of the significant parameters that make up the horizontal tail. Additional details on the calculations and formulas used can be found in Chapter 6 of Sadraey's textbook [17].

Vertical Tail The procedure for designing the vertical tail was similar to the design of the horizontal tail, with most of the parameters derived from either recommendations by Sadraey [17] or from the existing parameters for the horizontal tail. Parameters that were kept the same between the horizontal vertical tail include the distance between the wing aerodynamic center and the vertical tail aerodynamic center, the NACA airfoil profile, the taper ratio, and the sweep angle. The volume coefficient was selected from the higher end of typical values for vertical tail volume coefficients in Table 4 since the vertical tail would function doubly as the rudder required to essentially steer the craft and control the craft along the yaw axis. Sadraey recommends that the aspect ratio is a value between 1 and 2, and so was simply chosen to be the midpoint at 1.5, which was reflected based on the average aspect ration from the provided examples in the textbook. All other values for the vertical tail can be

Table 5: Parameters for horizontal tail design

Horizontal tail parameter	Value
l	50.52cm
$\overline{V_H}$	0.6
S_{th}	373.41 cm^2
C_{Lh}	0.002635
AR_{th}	2.13
λ_{th}	0.5
Λ_{th}	16.13°
$\overline{C_{th}}$	13.23 cm
$C_{h_{\text{root}}}$	17.01 cm
$C_{h_{\text{tip}}}$	8.505 cm

derived in the same fashion as for the horizontal tail, which are listed in Table 6.

As hinted to before, the need for the vertical tail in regards to maintaining yaw stability is minimal, since under steady state conditions there would not be an asymmetric force distribution about the pivot of the craft, thus theoretically generating minimal to no moment about the pivot. However, the horizontal tail is still required as it plays a large role in the control of the craft in the yaw axis, such as in the cases where the boat makes a turn; the craft would be required to properly rotate in response to the turning of the boat so that it can maintain the same direction. Thus, while a preliminary vertical tail was designed with reference to existing documentation for vertical tail design for aircrafts, future testing must ensure and adjust accordingly the parameters of the tail so that it can meet the demands of various scenarios for control of the orientation of the craft at all times.

Table 6: Parameters for vertical tail design

Vertical tail parameter	Value
l	50.52cm
$\overline{V_V}$	0.08
S_{tv}	171.36 cm^2
AR_{tv}	1.5
λ_{tv}	0.5
Λ_{tv}	16.13°
$\overline{C_{tv}}$	10.69 cm
$C_{v_{\text{root}}}$	13.74 cm
$C_{v_{\text{tip}}}$	6.87 cm

Motor Control Requirements Torque requirements for the horizontal and vertical tail were calculated by treating them as rudders used for boats, in which case readily available formulas exist for calculating the torque needed to rotate the shaft that the rudder is attached to. We referenced the DNV GL "Rules for classification: Ships" on how to calculate the torque needed to drive the tail surfaces [15]. The formulas to solve for the torque are as follows:

$$C_R = 132 K_1 K_2 K_3 A V^2 \tag{9}$$

$$K_1 = (AR + 2)/3 \tag{10}$$

$$Q_R = C_R c (0.33 - k) \tag{11}$$

where C_R is the rudder force, A is the planform area of the rudder, V is velocity, K_1 is a coefficient depending on aspect ratio AR, K_2 is the rudder profile coefficient and is 1.1 for a NACA-00 series

profile, K_3 is a rudder interference coefficient and is taken to be 1 since there are no propellers or nozzle, Q_R is the rudder torque, c is the MAC of the rudder, and k is the ratio of the planform area in front of the center line of the pivot of the rudder to the total planform area of the rudder.

Given these equations, we determined that the vertical tail required a torque input of 1.23Nm at the pivot shaft while the horizontal tail would require an input of 6.32Nm. Both tail surfaces are also designed to deflect +/- 15°.

Linkage Specifications To determine the exact linkage specifications, an iterative approach was taken based on the method of instantaneous centers. The linkage was simply modeled in Solidworks and link lengths were chosen to minimize the effective gear ratio n. The effective gear ratio for a 4-bar linkage anchored at two points can be derived geometrically as:

$$n = \frac{AO_2 * BI_{13}}{AI_{13} * BO_4} \tag{12}$$

where AO_2 is the length of link A, BO_4 is the length of link B, and AI_{13} and BI_{13} are the distances between link A and the instantaneous center, and link B and the instantaneous center, respectively.

Output torque (about O_4) within the +/- 15° actuation range must fulfill the 6.32Nm requirement at minimum. Beginning with a base distance (between O_2 and O_4) of 120mm, based on motor location constraints, link lengths were varied to maximize effective gear ratio. Based on our servo motor selection, which had the highest output torque available in that form factor (2.2Nm), we needed to maintain an effective gear ratio of less than or equal to $2.2/6.32 = 0.335$. Link lengths of 20 and 61.5 respectively accomplished this, with the highest ratio near the neutral position of the elevator, aka the position of lowest applied torque.

4.2.8 Multi-part Assembly

One of the major design challenges that we faced when it came to the external hull of the craft was how to design it in such a way that would make it easy to create, assemble, seal, and disassemble multiple times. The final division of the hull into pieces that could fit together and be 3D-printed is

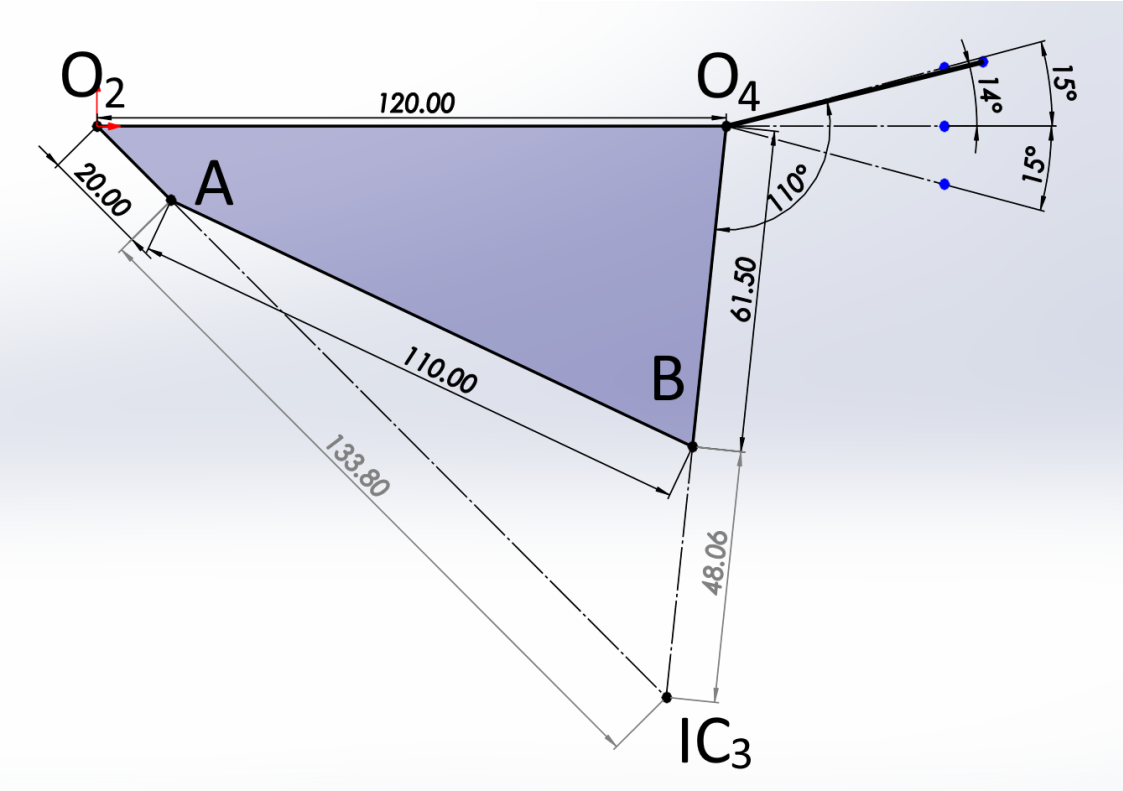


Figure 22: Method of instantaneous centers for elevator linkage

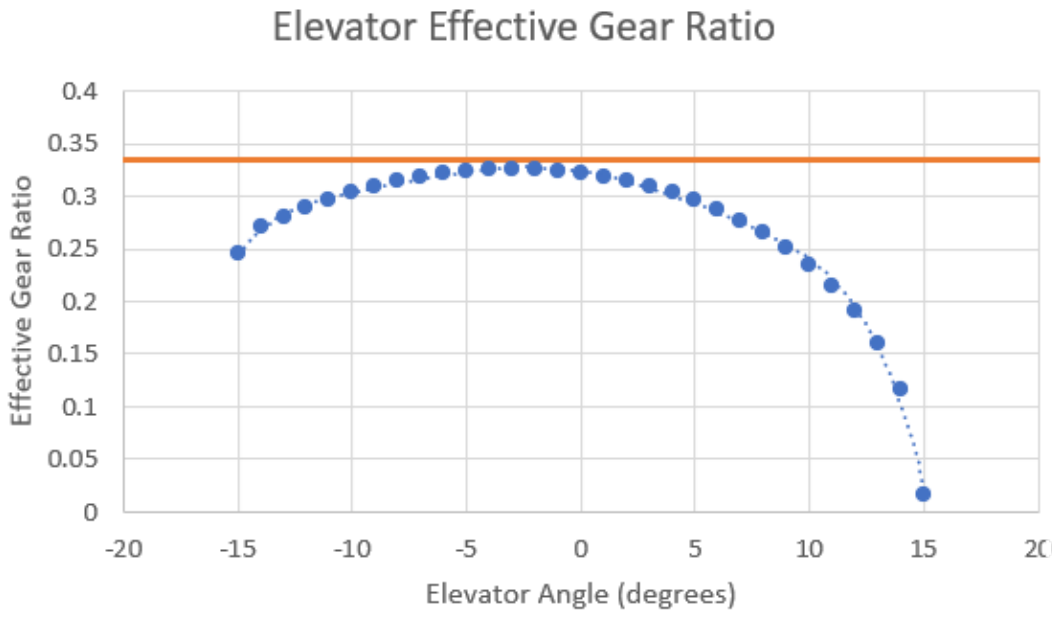


Figure 23: Effective Gear Ratio from Elevator Linkage

shown in an exploded view in Figure 24.

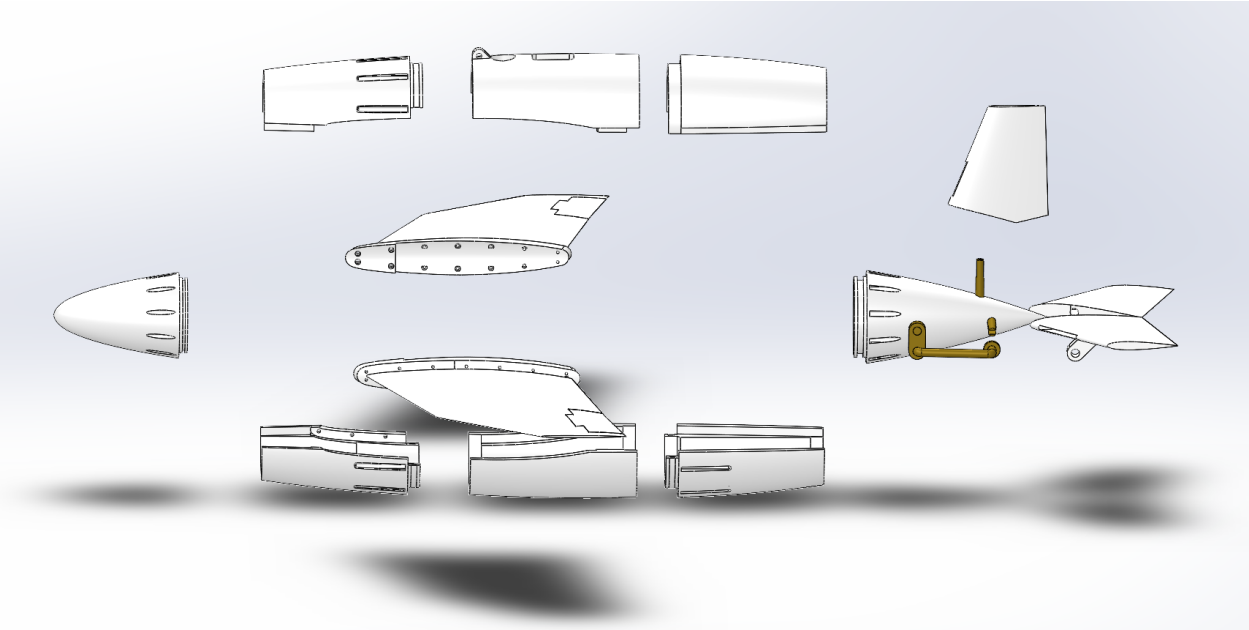


Figure 24: Division of hull for multi-part creation and assembly

Parametric modeling Due to build-volume limitations of 3D printers, we needed to ensure that each piece of the hull could fit within the space constraints. To begin, we used Solidworks derived parts and the Intersect feature to parametrically model separate pieces of the hull.

In Figure 25, each plane denotes a surface on which the model was cut. After splitting the main body model using the planes, the component parts are each processed in their own files to add details such as screw holes for connecting to each other, interfaces between parts to secure their positions relative to each other, cavity spaces to fit internal components such as the pressure hull and ballasts, and tolerances for 3D-printing and post-processing.

Similarly, the wings were split up in the same manner as shown in Figure 26.

As a means of simplifying the reassembly and disassembly process, the center pieces of the hull and all of the pieces of the wing were epoxied together to permanently secure them as a single piece. Figure 27 shows the exploded view of each of the divided pieces of the hull and wings grouped together into what would form the independent pieces of the craft when fully created and can be easily assembled and disassembled.

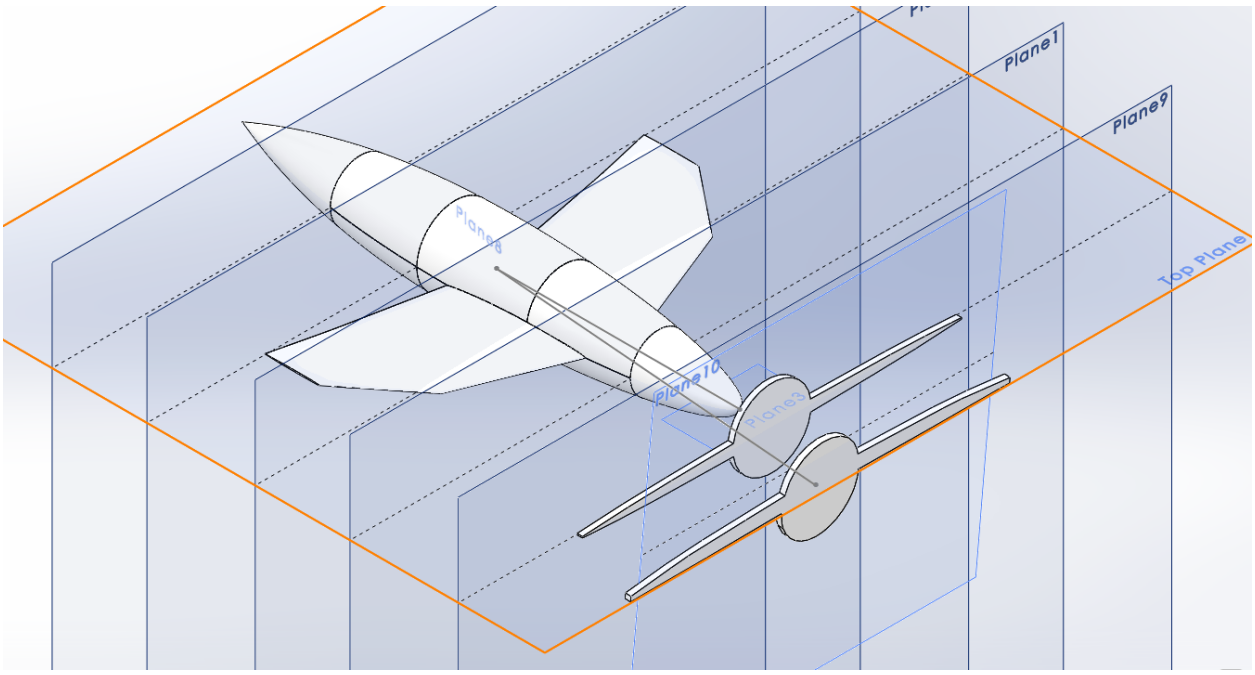


Figure 25: Use of planes to cut the model into printable sections.

4.3 Electronics Design

Based on our research and criteria for our craft, we produced a list of requirements for our electrical system:

- Sensors
- Motors and motor controllers
- Power from the tether
- Onboard voltage transformers
- Powerful microcontroller
- Data connection through the tether

4.3.1 Required Systems

The vehicle has subsystems that provide it information and capabilities. Our list for the systems includes:

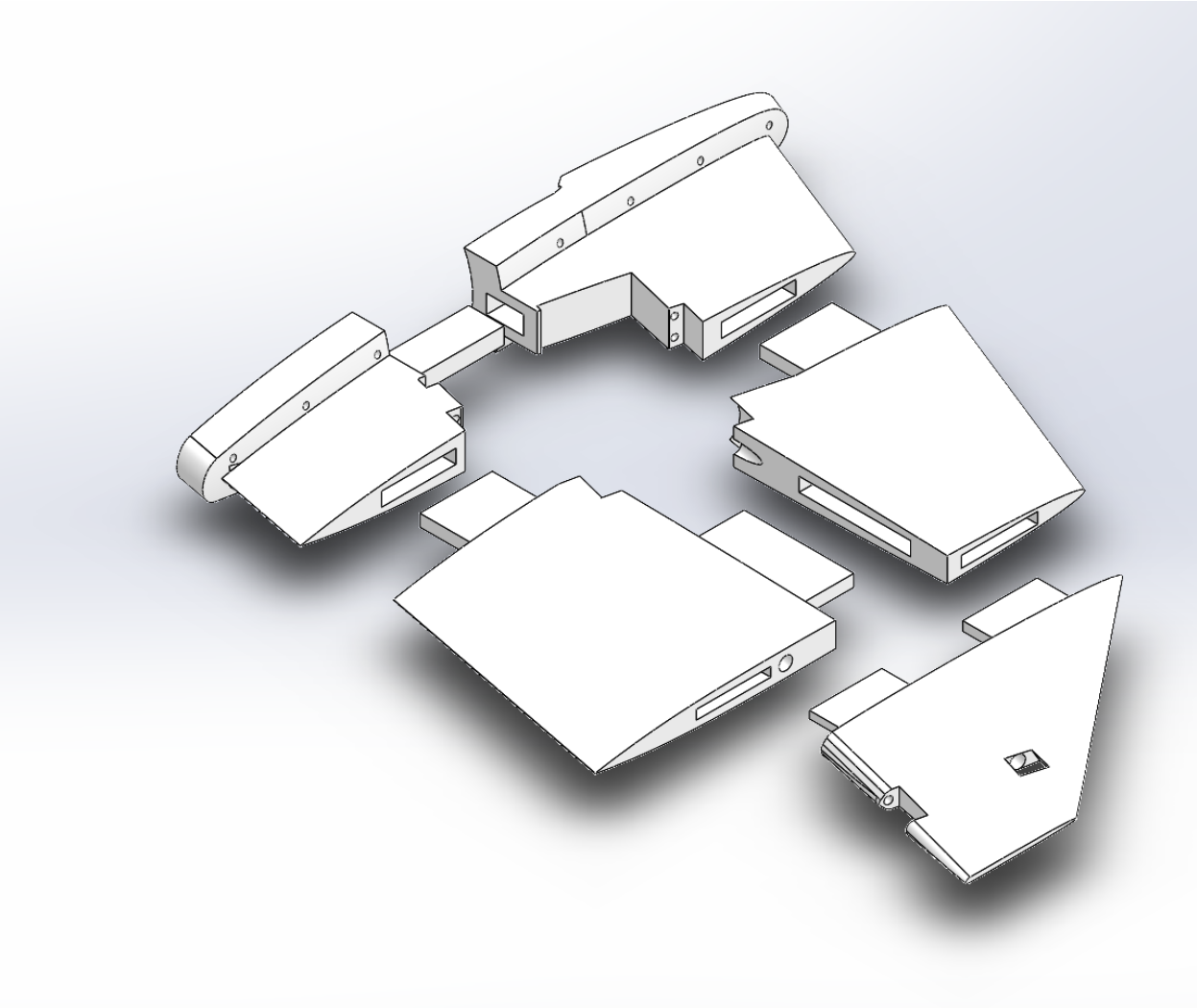


Figure 26: Multi-part assembly of the wing

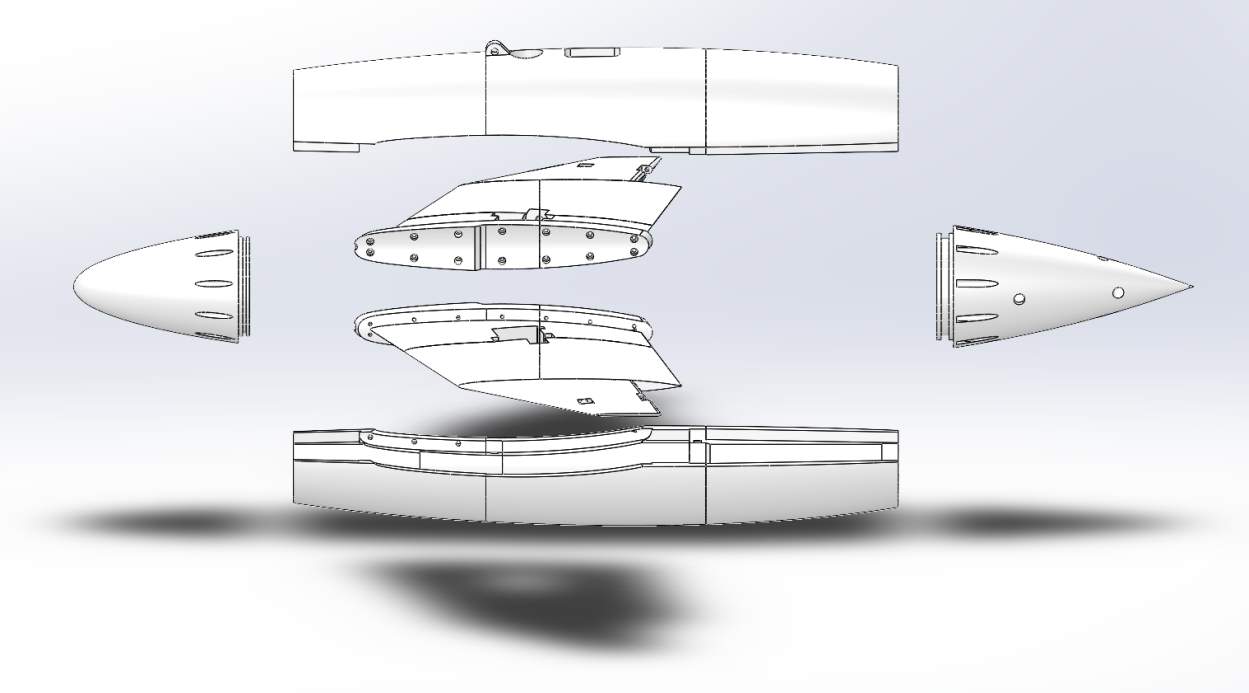


Figure 27: Combined parts of the hull and wing into separable groups when made

- Power System
- Orientation Sensor
- Pressure Sensor
- Motors
- Motor Controllers
- Servos

Power System From the topside, the tether provided us a 24V signal. None of our components can run on 24V, so we needed a way to step it down to a more usable voltage. Our motors ran on 12V, so we decided on using a 24->12V converter. Our motors required 2.5A nominal, 3A max for operation at 12V, while servos would require 2A stall current at 5V, so we would want at least 10A output. We eventually settled on a HOMELYLIFE Waterproof 120W converter, which was small enough to fit in our pressure hull but also had enough current output to power our systems.

The microcontroller, however, runs on 5V. In our first iteration, we decided to use a portable phone

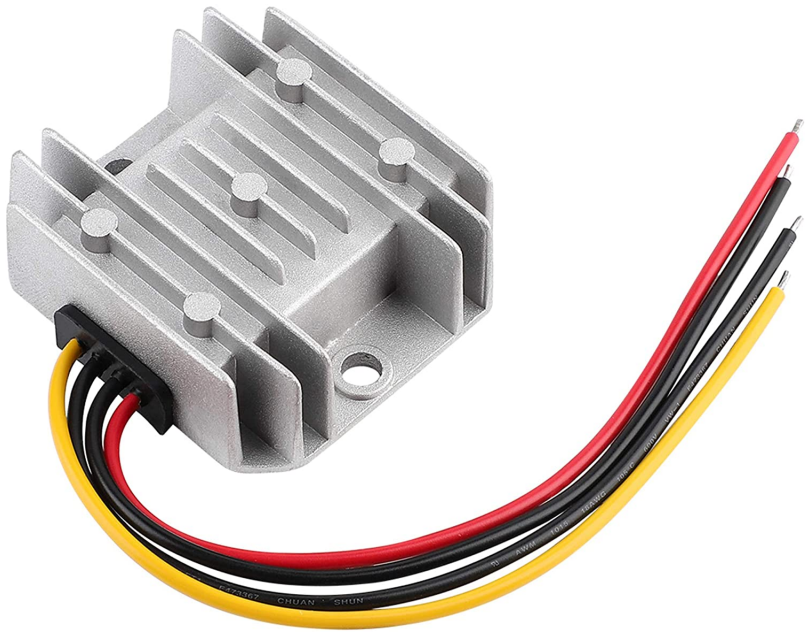


Figure 28: The HOMELYLIFE Waterproof 120W 24V to 12V Power Converter

charger to power our microcontroller. The main issue with this approach was that the microcontroller would always be on and drawing power when we sealed the craft, which complicated the testing procedure and will be discussed in the Results section.

In our second iteration, we decided on a suite of 12V to 5V converters. While we only used one, as the ailerons were never connected, there is space for up to five converters; one that powers the microcontroller, two that power the ailerons, and two that would power the rudder and elevator. This removes the need for the wakeup system described in section 6.1, but also provides power for the entire craft through only one avenue, which makes the system far simpler.

With our quick disconnect system, found in section 4.3.3, we connect the Ethernet and power delivery system to a series of 24AWG connectors as the internal ethernet connections are 24AWG. However, this creates issues with power draw - while the ballasts are drawing power, the pressure sensor doesn't have enough power to run. This worked in our first iteration, as we had 20AWG wires connecting from the Ethernet wires to the power converter. In a future iteration, this could be aleviated by upgrading the ballast connector to 8 pins rather than 6 pins, and using the extra two to connect the power part of the PoE system.

Orientation Sensor A form of orientation sensing is required to keep the craft upright and level. To implement this, we needed a 9DoF (Degrees of Freedom) sensor, which includes a 3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer. We began our implementation with a value analysis - from prior experience, we knew that the magnetometers found in all-in-one 9DoF sensors were poor, so our chosen implementation was a 6DoF sensor consisting of an accelerometer and gyroscope, and then a separate magnetometer. The value analysis can be seen in Figure 7 and 8.

Table 7: Value analysis for 6 DoF Sensors

Sensor	Cost	Accel Range	Accel Noise	Angular Range	Angular Noise	Supply Voltage	Interface	Current Consumption	Others
lsm6dsm	2.92	+-2/4/8/16 g	75-130 ug/sqrt(Hz)	±125/±250/±500/±100 dps	3.5 mdps/sqrt(Hz)	1.7-3.6	I2C - Up to 400kHz	450-650uA	Aux SPI, Master I2C
icm42688p	6.66	+-2/4/8/16 g	70 ug/sqrt(Hz)	±15.625, ±31.25, ±62.5, ±125, ±250, ±500, ±1000, and ±2000 degrees/sec	2.8 mdps/sqrt(Hz)	1.7-3.6	I2C - Up to 1MHz	880uA	DMP
icm20649	8.69	+-4/8/16/30 g	285 ug/sqrt(Hz)	±500 dps, ±1000 dps, ±2000 dps, and ±4000 dps	17.5 mdps/sqrt(Hz)	1.7-3.6	I2C - Up to 400kHz	3.21mA	Auxiliary I2C, DMP
bmi270	6.01	+-2/4/8/16 g	160 ug/sqrt(Hz)	±125/±250/±500/±100 dps	7 mdps/sqrt(Hz)	1.7-3.6	I2C - Up to 1MHz	685uA @ 1.8V	Auxiliary Sensor Interface

Table 8: Value Analysis for Magnetometer

Sensor	Cost	Field Range	Field Noise	Bit data output/sensitivity	Supply Voltage	Interface	Current Consumption
mmc5883ma	4.93	+-800uT	0.4-1.2mG RMS	16 bit, 25uT/LSB	2.16-3.6	I2C - Up to 400kHz	120uA
ak09940	7.12	+-1200uT	40-70 nT RMS	18 bit, 10nT/LSB	1.7-2	I2C - Up to 400kHz	100-200uA
LIS2MDL	1.77	+-5000uT	3mG RMS	16 bit, 150uT/LSB	1.7-3.6	I2C - Up to 400kHz	200uA

From these, we chose to use the LSM6DSM and LIS2MDL together, as they provided fine performance along with being the cheapest of the combinations. From research, we noted that others tended to use this combination as well. We designed the schematic, and decided to connect the LIS2MDL to the LSM6DSM's auxilary I2C connector. This was done so that we could communicate with both at the same time and not worry about the data being out of sync.

The schematic and PCB were both designed using KiCad. While both sensors run on 3.3V, we decided that the input should be 5V and to use a 5V to 3.3V LDO to minimize noise and ensure power stability. When we implement our power supply system, it will be 48V and converted to lower voltages on board, which we expect to be 12V and 5V. Converting to 3.3V and running those power lines would've been an extra hassle. Furthermore, the LDO has a high PSRR, which means that there should be very low levels of noise, as these sensors are sensitive to power supply noise. We are using I2C to communicate with the sensor as it needs fewer wires (easier to route inside the craft) as well as that it handles longer distances better than SPI. The schematic and PCB can be found in Figure 29 and 30 respectively.

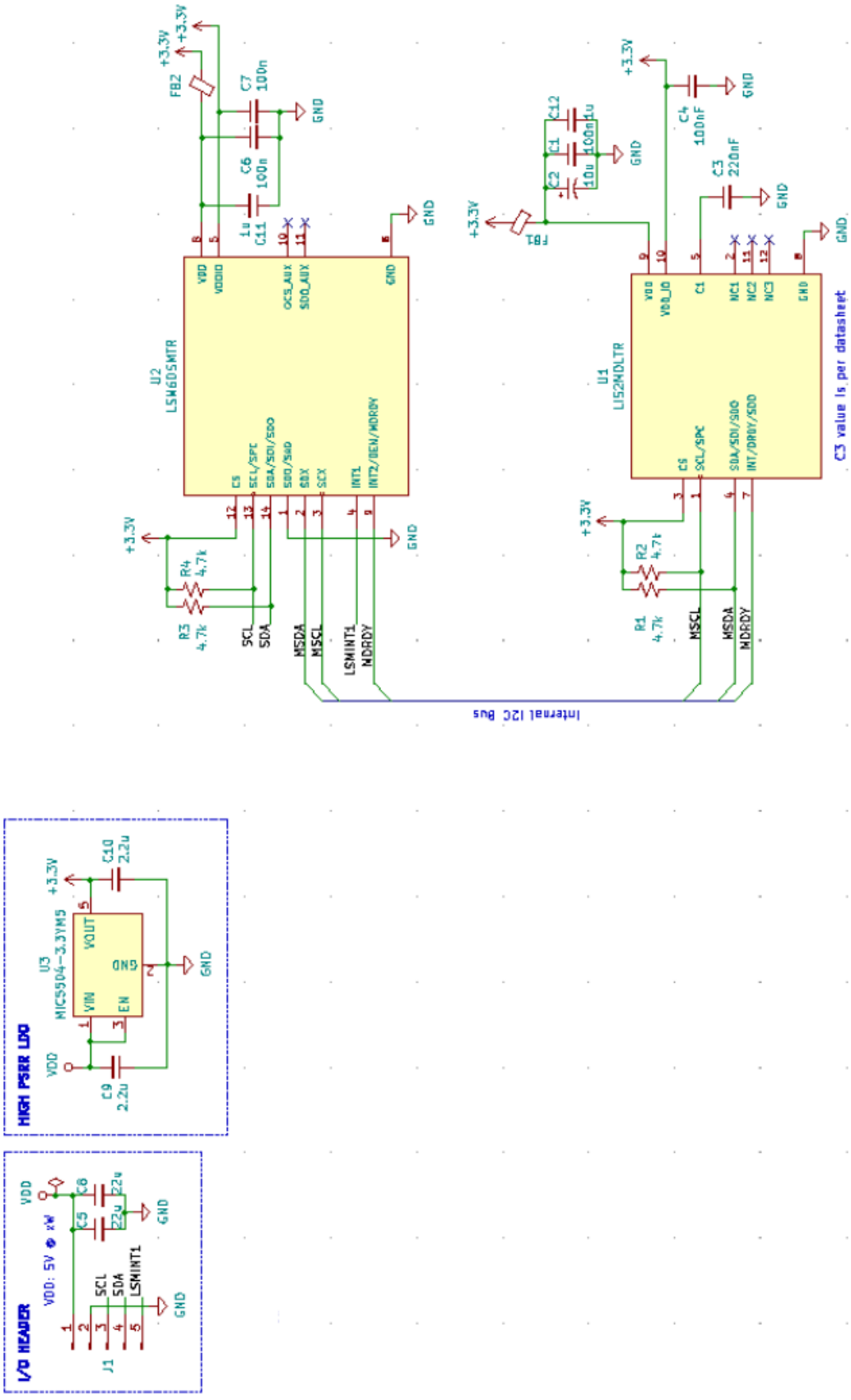


Figure 29: Inertial Measurement Unit (IMU) schematic

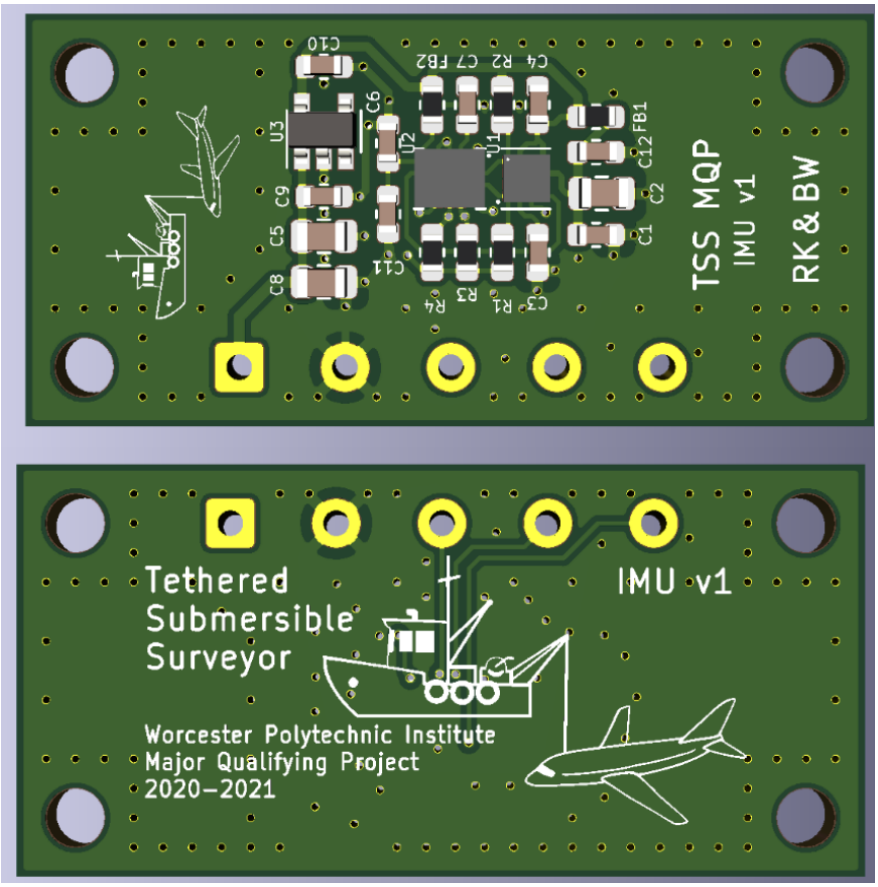


Figure 30: IMU PCB

Pressure Sensor To determine the absolute depth of water that the craft is submerged in, we decided to use a pressure sensor. We researched OpenROV's implementation, and they used the MS5803 Pressure Sensor. We decided to use the same sensor, as it provides readings of up to 14bar or 150m. OpenROV claims that they were able to get 1cm of sensitivity, which is perfect for our needs, as we want to be as close to the ocean floor as possible, and this will allow us to know whether we are keeping a consistent depth or not.

OpenROV put their IMU and pressure sensor on the same board, however we decided to separate them. This was due to concerns about the IMU possibly being exposed to water, as well as making it easier to place on the craft. The MS5803 is placed next to the Ethernet port on the top of the hull, and potted to avoid water getting on anything but the sensor.

We used I2C to communicate with the pressure sensor, as that is what we were already using with

the IMU. We also put a 5V to 3.3V LDO on the board for the same reasons as we did with the IMU. The schematic and PCB can be found in Figure 31 and 32 respectively.

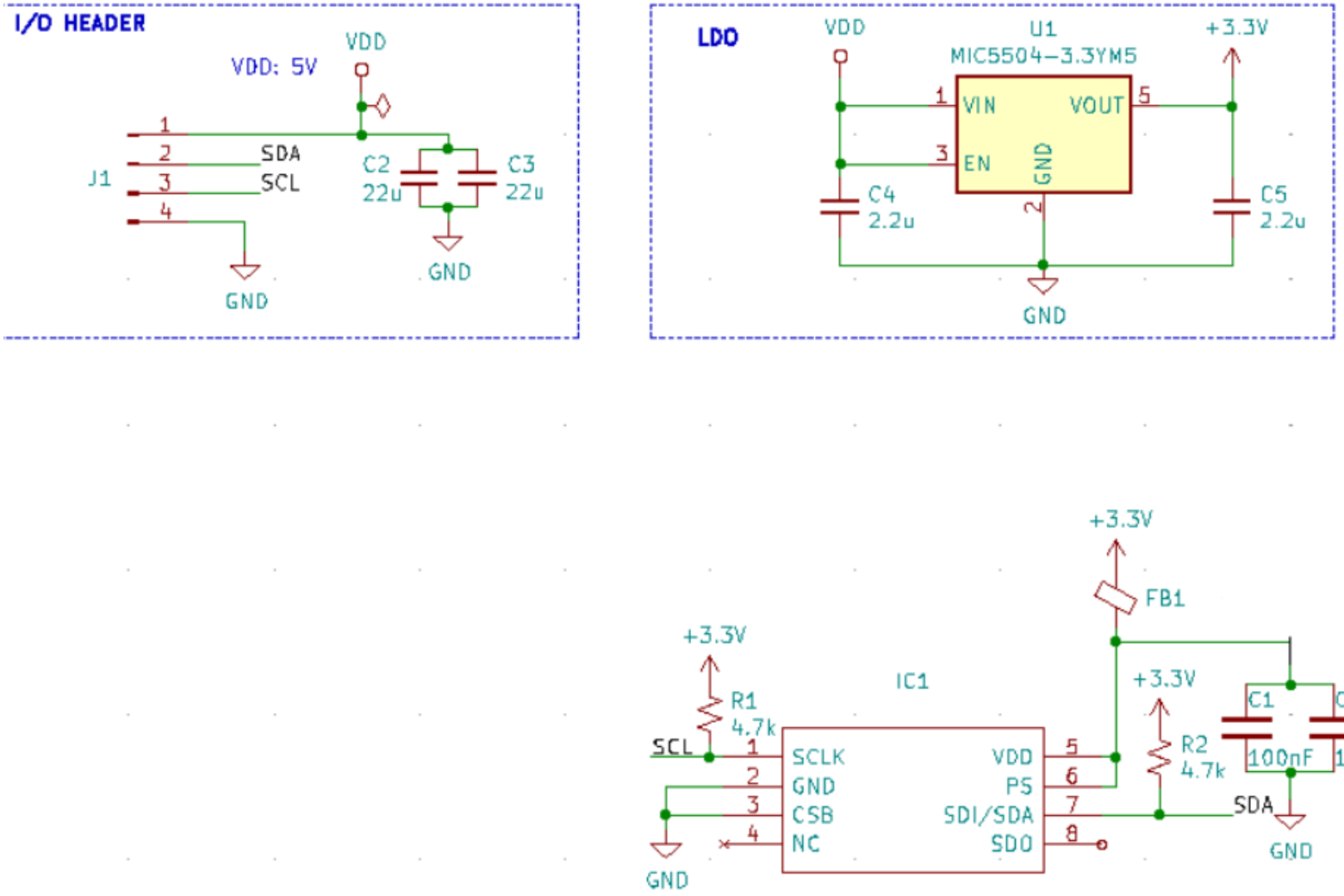


Figure 31: Pressure Sensor schematic

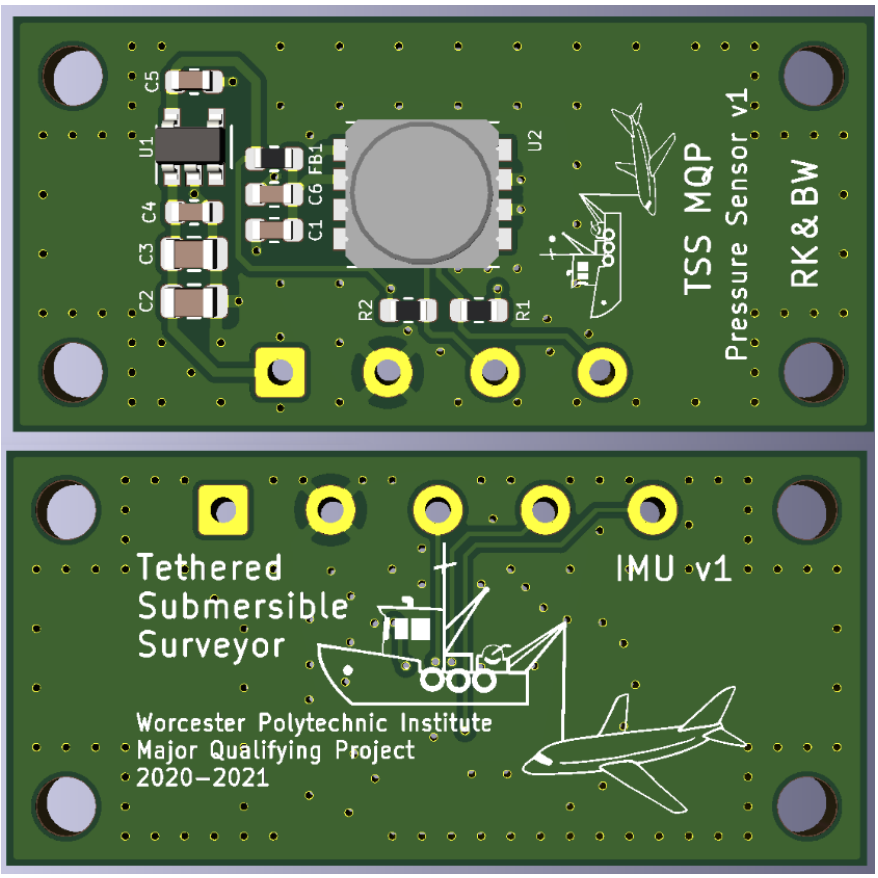


Figure 32: Pressure Sensor PCB

We wrote our own I2C driver for the MS5803. Once the I2C bus is initialized, we need to read the coefficients from the PROM. There are 6 coefficients in total, and they are calibration coefficients necessary to compute the temperature and pressure. To get the temperature and pressure values, we read from the ADC. To start, we send a code comprised of the command (read ADC), the type of command (temperature or pressure), and the precision. Once that is written, we wait a few milliseconds, and then tell the MS5803 to move into read mode. From there, we read 24 bits out.

To get the true temperature and pressure values, we need to read both, as the calculations require on each other. You can find the code in Appendix B.

Motors Ballast motors were selected to fulfill the calculated requirements of 150 oz-in at 270 rpm. While there are a very wide range of motors available for purchase, it is difficult to identify specific SKUs since there are few central storefronts. We identified a suitable motor available from Amazon

with a rated torque of 15 oz-in at 3000rpm, which fits these specifications when paired with an 10:1 gearbox. The motor actually far exceeds these specifications and was able to drive the full load of the ballast when directly driving the leadscrew (no gearbox) at a 50% duty cycle. This motor operates at 12V, matching our system voltage, with a rated current of 3A under full load. While we never observed this current consumption, we ensured that our voltage regulator would be able to provide that much power.

Motor Controllers We needed motor controllers to control the ballast motors, as direct-driving through the microcontroller would be impossible. We ordered the BTS7960 PWM H-Bridge Motor Controllers, which are small, simple, and rated for a high enough current. Many off the shelf motor controllers are available, but most small, cheap controllers are rated for lower voltage and/or lower power operation. Since the stall current of the ballast motors is approximately 3A, we needed a rating of at least 36W continuous. With their large heat sink, the BTS7960 motor controllers are theoretically rated for 43A, and are suitable to driving brushed DC motors between 6V and 27V, so they fit our application perfectly. They also cost very little, at about $15 for two controllers. In practice, these controllers will likely not tolerate rated specifications, so we ensured that we would run them with a large factor of safety.

They work through pulse-width modulation, or PWM. This is when instead of passing a DC signal to your motor, you run a square wave at varying duty cycles. The higher the duty cycle, the faster the motor.

On one side, we have the microcontroller connections. This include a 5V and GND connection for the controller to function, a current sensing output, a left and right enable signal, and a left and right PWM signal. The other side's connections are a screw terminal, where there are two battery inputs, and two motor inputs.

For the system to work, both enable signals must be turned on - that turns on the two BTS7960B Drivers and allows them to function. The Current Sense outputs are unnecessary to the function of the device but allow us to see whether there is an overcurrent event.

Figure 33: The BTS7960 Motor Controller

The two PWM signals work as opposites. If you want to turn the motor counterclockwise, the right PWM signal needs to be on and the left needs to be off. If you want to turn the motor clockwise, the left PWM signal needs to be on and the right needs to be off.

We used Timer 2, which provides 3 PWM channels. We used channels 1 and 3, with a prescaler of 15 - which divides our clock by 16. Then, we had our counter at 299 - which gave us a timer running at 20kHz. Both PWM signals are initially turned off, but both the duty cycle and the direction can be set through signals sent from the boat.

Our code has four functions - one that shuts off the motors by disabling their enable signals, one that turns on the motors by enabling the enable signals, ones that stop the motor by turning off their PWM signal, and one that rotates the motor in a specific direction and by a specific duty cycle. The code can be found in Appendix C.

Servos The aileron motor was selected to fulfill the 0.66Nm calculated torque requirement and to fit within small volume constraints. The most advantageous motor packaging is inside the otherwise-unused space within the wings. To fit within this space, a standard-form-factor servo motor was selected. The highest-torque option available common off the shelf was the Gobilda 2000-0025-0002 ”Dual-Mode Torque” servo with a torque rating of 2.12Nm at 5V. This far exceeds the aileron torque requirement, and allows us to reuse the same small servo motors, with linkage, for the tail elevator

and rudder. Additionally, the integrated absolute position feedback allows us to reduce overall part count and software complexity by utilizing the servo's hardware control loop.

Servos are simpler to control in comparison to the motor controllers. For power, they can handle anywhere from 4.8V to 7.4V. We selected an operating voltage of 5V so we could reuse the same 12V->5V voltage regulators used to power the other system electronics.

Its third connection is a PWM signal that tells it what position to go to. This PWM signal works off of signal timing rather than duty cycle, so a 500μs pulse puts the servo at its minimum position, while a 2500μs pulse puts the servo at its maximum position, which is around 300 degrees.

To implement this, we provided a 50Hz PWM signal, which was created by using a prescaler of 959 and a counter period of 2000. To tell the servo to go to its minimum position, we give it a PWM pulse of 50 - which corresponds to a 500μs pulse. Its maximum position is at 250, which corresponds to a 2500μs pulse. This maps 300 degrees of travel to 200 values, which is enough precision for our application.

4.3.2 Connectivity

To connect our vehicle to the boat, we need both a physical link and a data link. The physical link came in the form of a tether, as discussed in section 4.1.1. For the data link, we considered two options - we could use the UART (Serial), or Ethernet convention. UART is easy to use but has a low data rate, while Ethernet is much more complicated to implement but is high data rate and ubiquitous. We decided to use Ethernet, as its high throughput provides enough overhead for us to put as many sensors on the craft as possible and still be able to send all the data to our operators console. It also would allow the end-user to plug in the robot and set it up with little to no issue, as most computers have an ethernet port.

From here, we had to decide how we wanted to wire up the tether. We very quickly decided on a standard ethernet cable, as they are cheap, ubiquitous, and would already fit into most peoples computers. This would also allow us to use Power over Ethernet - a power delivery method where we send power down the unused wire pairs in the ethernet cable for use on the robot.

To connect the tether internally and externally, we relied on Power Over Ethernet (PoE) splitters. These are components designed to either combine or split Ethernet and power. On the topside, we used a splitter to connect our batteries and laptop, and on the craft side we used a splitter to connect the Ethernet to the microcontroller and the power to the converter without having to manually break out the wires into power and ethernet.

Power over Ethernet Power over Ethernet comes in two types: active or passive. Active PoE determines necessary voltage that the ethernet cable needs to supply, and requests it. Passive PoE means that the voltage down the cable is constant, even if the device connected to it needs a different amount than the supply. Active is safer, as it ensures that there will be no over-voltage event, but requires a lot more space compared to passive. We decided to use passive PoE on our robot, as we are designing the hardware ourselves. Active PoE requires additional hardware and design space and increases overall system complexity. With passive, we are free to design our own power delivery system that takes in the provided voltage and steps down to the voltages that we require, with no software negotiation required.

4.3.3 Quick Disconnect System

After our first test, we found out that passing wires through the pressure hull endcaps without permanent sealing results in leakage, no matter how well-sealed it looks. If we were to permanently seal the wires, however, we could never remove the electronics from the pressure hull. To work around this issue, we designed a quick-disconnect system (QDS) for the electronics bay. The block diagram can be seen in Figure 34.

We used two connectors for the QDS. The ballast connector used 18-AWG wires, so we used 6-pin Mini-Fit Jr 5557 connectors, which are better known for being the connectors used in 6-pin ATX connectors in computers. We chose them as Jeffrey had experience crimping them, they could handle 18AWG wires, and they had locking mechanisms. The second connectors chosen were for the PoE system, pressure sensor, and ailerons, and were meant to handle 24AWG wires. We chose

JST-PA 6-pin connectors, as they could handle the wire gauge we were using, as well as lock into place. However, we were unable to crimp them properly, and ended up buying XINGYHENG Female-Male Connection Plugs, which were pre-crimped 24AWG connectors, and while they did not lock like the Mini-Fit connectors, they were sturdy enough for our purposes.

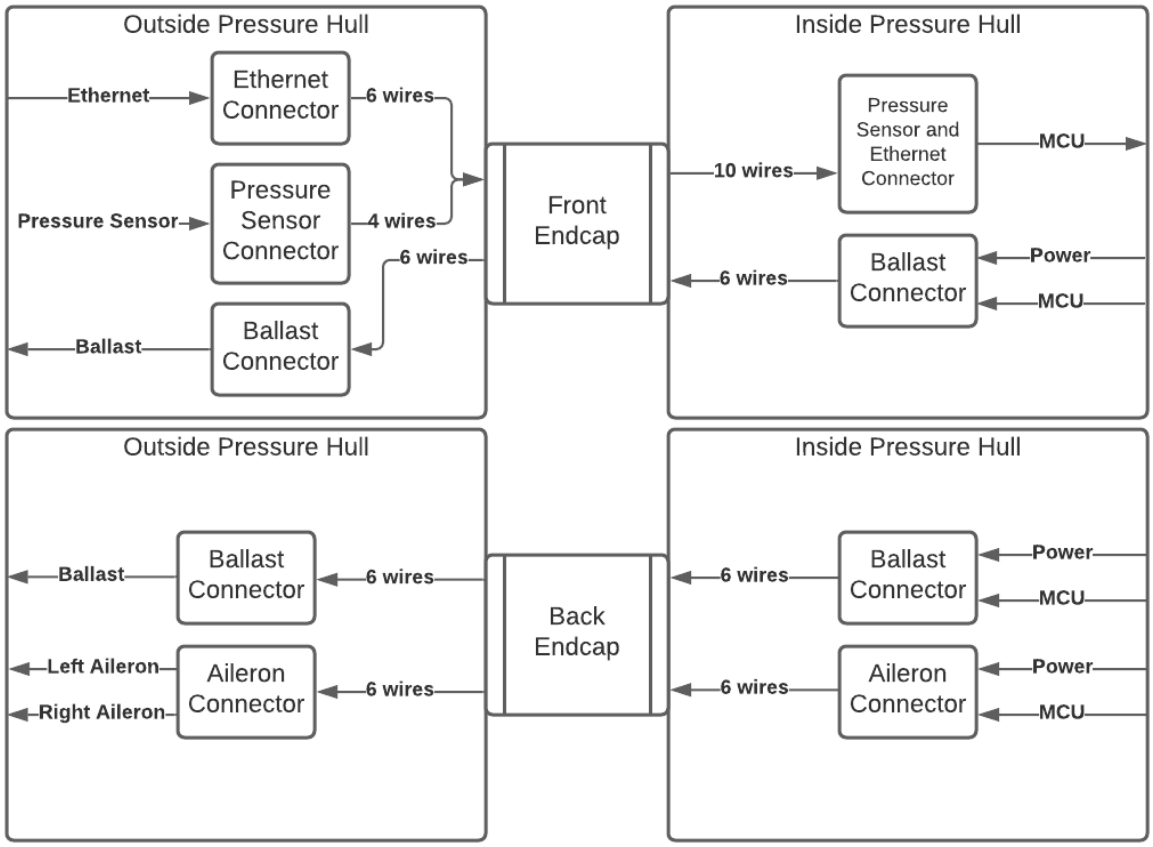


Figure 34: Illustrated quick disconnect system

Figure 35 shows an initial test of the quick-disconnect system. On one end is the ballast, and on the other end is the motor controller, limit switches, and enable switches. The motor worked through this connection, which meant that we proceeded to implement the QDS into both endcaps.

This solved multiple issues. In our first iteration, we designed the system to be fully modular. However, due to everything being soldered to the microcontroller, it was one bulky unit, which made it hard to do maintenance on any one part. Furthermore, we are able to seal the wires of the QDS into the endcaps using epoxy, which meant that there was no possibility for leakage. We could still

Figure 35: Initial test of quick-disconnect system.

remove the endcaps, as we can just disconnect the system on either side to separate the parts. ‘ There were still issues with the implementation, however. The wires that of the ballast connectors inside of the endcap were stranded, but also 18AWG, so they were tough to maneuver. Furthermore, the connectors themselves were also quite rigid, so it was tough to squeeze them in. We were able to put the connectors to the sides of the motor controllers and the 24AWG connectors for the Ethernet, pressure sensor, and ailerons were able to fit under the board.

Figure 36 shows the final implementation of the quick-disconnect system. On the left is the rear endcap, which holds a ballast connector and two aileron connectors. On the right, the front endcap can be seen with numbered connectors for the Ethernet and Pressure Sensor. This is because the XINGYHENG connectors came in sets of three, and since we needed 10 wires for the front endcap, we had to use four connectors which all looked identical, so we needed a way to distinguish between them. Connector 1 was the pressure sensor SDA, SCL, and GND. Connector 2 was pressure sensor VDD. Connector 3 was the Ethernet’s red, black, and white connectors. Finally, connector 4 was the

Ethernet's green connector, as well as the Power over Ethernet VDD and GND.

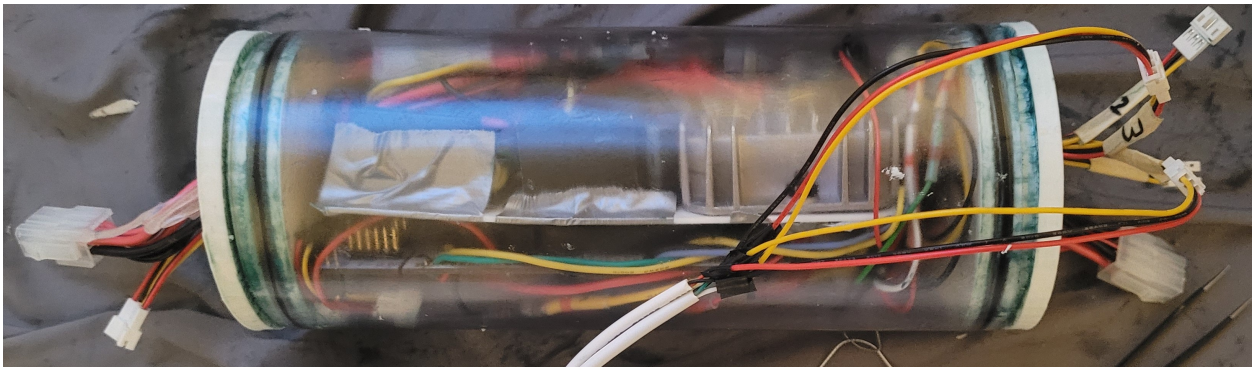


Figure 36: Quick disconnect system installed into the pressure hull

4.3.4 Computing Hardware

Our robot needs to be able to make decisions very quickly, as well as at a predictable speed. To do this, we needed a microcontroller with enough processing power to handle our sensors and to be able to run a real-time operating system (RTOS), which is a highly deterministic operating system capable of running very tight timing restraints for systems that need to be predictable, like ours. To choose a microcontroller, we came up with a list of what we valued in a microcontroller.

- Cost
- Speed
- Onboard RAM
- IO pins
- Communication methods (Ethernet is a must)

From these, we looked at what kind of microcontrollers others had used in the past, and created a comparison chart between our choices. Many of the options we looked at are STM32 based, as there are a wide variety of STM32 microcontrollers that would suit our needs.

Table 9: Value analysis of different microcontrollers

Microcontroller	Processor	Cost	RAM	Flash	IO Pins	Comm Protocols
STM32F746ZGT6	ARM Cortex M7	14.25	320K x8	1M x8	114	CANbus, EBI/EMI, Ethernet, I²C, IrDA, LINbus, SAI, SD, SPDIF-Rx, SPI, UART/USART, USB
STM32F767ZIT6	ARM Cortex M7	16.69	512K x8	2M x8	114	CANbus, EBI/EMI, Ethernet, I²C, IrDA, LINbus, MMC/SD/SDIO, QSPI, SAI, SPDIF, SPI, UART/USART, USB OTG
TM4C	ARM Cortex M4F	16.25	256K x8	1M x8	90	CANbus, EBI/EMI, Ethernet, I²C, IrDA, QEI, SPI, SSI, UART/USART, USB OTG
R7FA	ARM Cortex M4	11.79	384K x8	1M x8	109	CANbus, EBI/EMI, Ethernet, I²C, IrDA, MMC/SD, SPI, UART/USART, USB
STM32F207ZGT6	ARM Cortex M3	13.22	132K x8	1M x8	114	CANbus, Ethernet, I²C, IrDA, LINbus, Memory Card, SPI, UART/USART, USB OTG
STM32F217VGT6TR	ARM Cortex M3	12.45	132K x8	1M x8	82	CANbus, Ethernet, I²C, IrDA, LINbus, Memory Card, SPI, UART/USART, USB OTG
STM32F407VGT6	ARM Cortex M4	11.79	192K x8	1M x8	82	CANbus, DCMI, EBI/EMI, Ethernet, I²C, IrDA, LINbus, SPI, UART/USART, USB OTG

From the list in Table 9, we narrowed our choices down to the Renesas R7FA, and the STM32F746. Eventually, we decided that the STM32F746 was our best fit - while the R7FA was far cheaper and provided more RAM, the F746 had a far better processor for a slightly higher price that would not amount to much if we produced boards in a small enough quantity. For our initial prototypes, we decided to use a development board as we felt our time, at this stage of the project, would be better spent working with a development board than attempting to design and produce a custom PCB. The most well-known brand of STM32 development boards come from Nucleo, and only cost around $25 no matter what microcontroller we ended up choosing. Due to this, we decided to go with the STM32F767, which had more RAM and flash than the F746. This would provide more processing headroom with only a small cost increase.

While working on the project, we learned that the STM32F767 drivers had many issues, and while the processor is great on paper, has many issues during implementation. We do not recommend continuing work with this processor, but instead moving to a different one. The F746 is a good option, and does not seem to have the same driver related problems.

4.4 Software Architecture

The software for this project is divided into two main parts: The Operator Console and the Robot Control System. Within each of the parts are specific components, which address a specific challenge associated with that part of the system.

The Operator Console is the software application which coordinates the functionality of the entire system. This application runs on a computer located upon the vessel towing the tethered submersible. The application is responsible for taking in raw data from the robot, the environment, and the internet, and crafting helpful visualizations to allow a human operator to easily understand the environment around the robot, and provide instructions to the robot.

The Data Management component of the Operator Console is responsible for aggregation, visualization, and general management of all data sent between the craft and the operator console. Included in the process of Data Management is the creation of logs.

The Command Interface component of the Operator Console is responsible for providing the user of the Operator Console to control the actions of the craft with an intuitive interface.

The Robot Control System is the software that coordinates the different functional elements of the robot in operation. The Robot Control System oversees the robot's responses to stimuli, receives and processes commands from the Operator Console, and packages recorded data for transmission. The control system abstracts the low-level commands to actuators so that the human operator does not have to pay attention to the details of the robot's physical operation over time.

The Craft Control component of the Robot Control System is responsible for translating commands sent to craft into actions, including actuation of motors.

The Communications component of the Software interacts with both the Robot Control System and the Operator console, and is responsible for transmitting and receiving messages for both systems. This software component provides the foundation for the correct implementation of all of the other Software components.

4.4.1 Components of the Operator Console

The Operator Console software application includes three of the four primary components of the software system of the project.

- Communications
- Data Management
- Command Interface

In addition to these components, the Operator Console includes the Graphical User Interface which supports multiple components.

Communications The communications component consists of a UDP Client/ Server setup between the Operator Console and the Craft, implemented with the simple and ubiquitous network socket

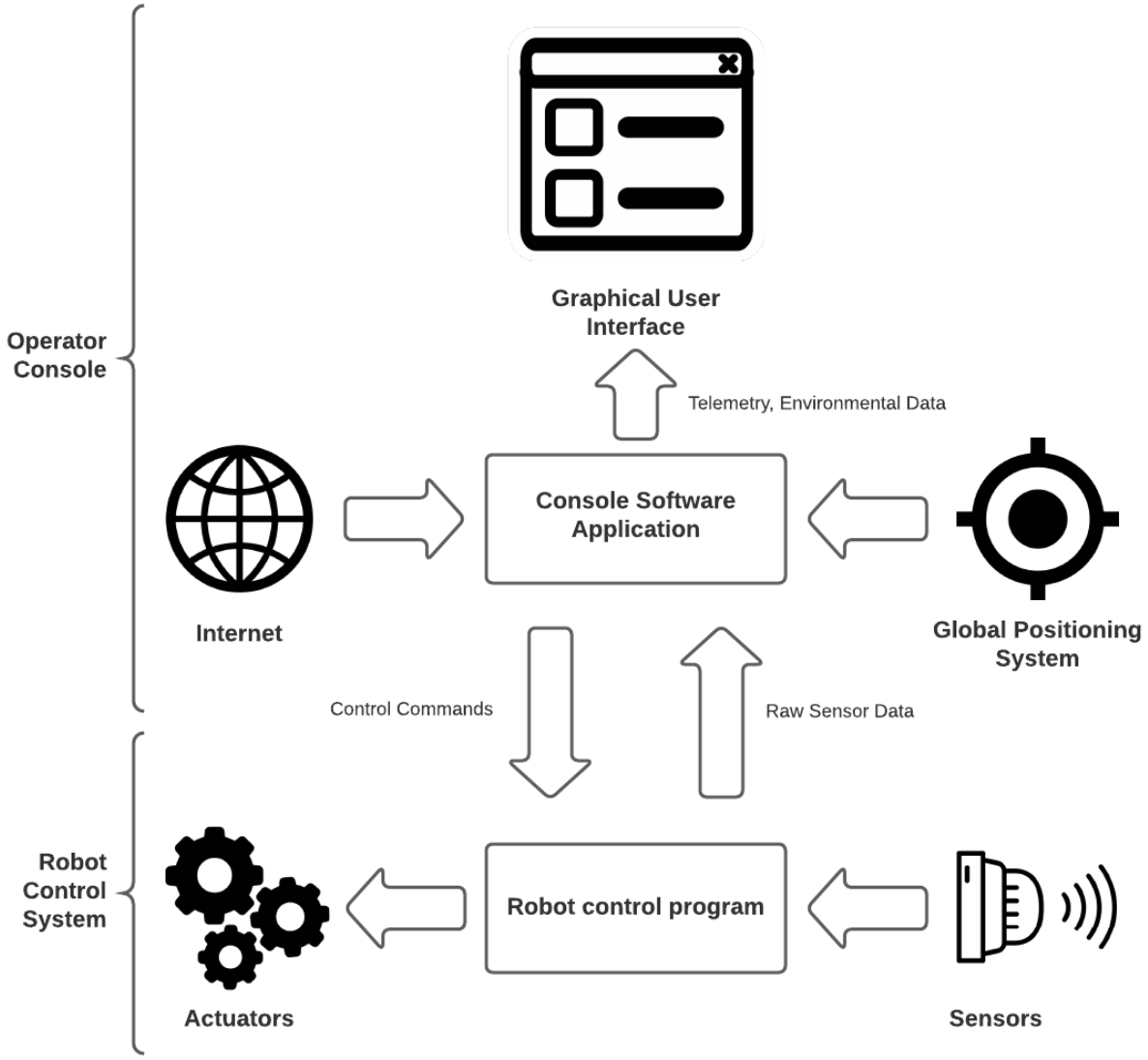


Figure 37: High Level Software Diagram

architecture. The server and client are configured with Static IP addresses and connected in a peer-to-peer network. The Operator Console application acts as the server, while the craft's control software acts as the client. The craft streams data to the Operator console, while the Operator console sends control instructions to the craft.

Graphical User Interface The Graphical User Interface (GUI) is the main part of the application with which the human operator directly interacts. The GUI consists of various text and images which allows the operator to visualize the robot's telemetry and sensor data. The GUI also includes input to allow the operator to send commands to the submersible via the tether. The GUI interacts heavily with Data Management and Command Interface components.

Data Management The data management part of the application consists of both Data aggregation and Visualization pieces. Data aggregation is performed by identifying and converting sensor data streamed from the submersible into usable information for display. This more useful data is then used for visualizing the data for the operator.

Command Interface The Command Interface is the component of the software which processes a user's commands and ensures that the corresponding message is handed to the Communications component.

4.4.2 Components of the Robot Control System

The Robot Control System includes two parts:

- Craft Control
- Communications

Craft Control The Craft Control component of the software interprets commands coming from the Communications component, and translates them into real-world actions, such as movement of the

control surfaces or entering sleep mode.

4.4.3 Process

Establishing Connectivity To start development, we first addressed the need for basic Craft Control software, in order to prevent the development and testing of electronic and mechanical aspects of the project from having to wait on software development. Initially, we attempted to use a Real-Time Operating System (RTOS) as the supporting software for the robotic craft's control software. There are many different kinds of RTOS, but we chose FreeRTOS for its popularity, and therefore support, and open-source nature. We also were not sure of all the specific requirements of the software as the craft was actively being developed, and therefore opted for a general-purpose, mainstream RTOS for ease of development.

From there, we needed an Ethernet driver for FreeRTOS. We narrowed our choices down to the Lightweight Internet Protocol (LwIP) and FreeRTOS's proprietary TCP/IP stack. FreeRTOS documentation provided a step-by-step setup for their proprietary stack, so we decided to use that. Unfortunately FreeRTOS was not compatible with the hardware platform we used, due to a conflict of drivers, so we used LwIP instead without FreeRTOS.

Successfully using LwIP to connect the craft's microcontroller to the server proved difficult. After much troubleshooting, we found that we needed to use a static IP configuration on a peer-to-peer (P2P) network in order for the two devices to recognize each other.

Once we established that the two devices could network together, we set up a simple UDP server and client configuration to pass messages back-and-forth across the network. This was accomplished using the software resources provided by the manufacturer of our microcontroller, and a basic Python sockets application. With UDP configuration and P2P networking configured, we were finally able to send and receive messages across the network. We used this capability in a simple python application to create a real time graph, that updated as data was sent to the computer from the craft's microcontroller.

Handling Multiple Simultaneous Tasks Once reliable connectivity between the client and server was established, we moved toward implementing basic information exchange between the two devices. The major challenge of this goal was to process incoming data for each device, take the appropriate action, and handle other tasks simultaneously. For the server, we had to design the capability to receive many messages in a very short period of time, due to rapid data output from the microcontroller. Based on the contents of the message, different actions would be performed, such as visualization, calculation, or logging. The server also had to run the Graphical User Interface and handle user input.

The way we handled this was through multithreading. Each logical task performed by the Operator's Console is separated into its own distinct thread, which runs independently of the others. This allows us to maximize the concurrency of the system and ultimately accomplish more tasks in a smaller amount of time. The threads exchange information through Python's queue module, which allows for thread-safe storage and retrieval of arbitrary information. Each software component of the Operator's Console has its own dedicated thread. These are the UDP Server (Communications) thread, Logging (Data Management) thread, and Command thread. The Graphical User Interface is run inside of the main process and therefore does not have its own separate thread.

The onboard software for the craft uses a slightly different approach to handling many tasks. Instead of using multithreading, the craft software uses a polling system. The software reads sensors and checks for incoming messages many times a second. If data is detected, it is immediately processed. For sensor data, this means it is sent to the server. For an incoming message, this means that the message content is immediately read and an action determined.This method prevents data from becoming stale by immediately acting on data as it received. Since each operation is so quick to complete, there is no noticeable disadvantage to not doing multiple concurrent threads in this case.

5 System Integration

During the second half of our project, we gradually transitioned our efforts to integration of mechanical and electrical components to achieve a fully functional prototype that could be tested in water. In preparation of our first major test, where we aimed to seal and submerge the entire craft with the minimum electronics needed to communicate and control the ballasts, we broke up our process into a series of assembly, integration, and testing tasks, which is shown in the flowchart in Figure 38.

During the integration process, we encountered several difficulties that we failed to address prior in the design of each subsystem. The major problems that we needed to resolve during the integration phase prior to our first full test are summarized as follows.

5.1 Bypasses and Modularity

Although one of the initial design goals for our craft was to maximize modularity by designing subsystems to be independent and separable from each other, which we had to a degree succeeded in, we failed to account for the ability for each subsystem to be actually disconnected from each other in our design even though the physical subsystems were separate from each other. For example, although the ballasts and the pressure hull were independent physical entities, the wiring that connected the motor and the limit switches were originally planned to simply bypass through the endcaps of the pressure hull. With this design, we essentially nullified the benefits of the modularity, since they could not be detached from each other and thus the internal components of the pressure hull had to be inserted permanently and as a result made the integration and testing process awkward to work with as a whole.

5.2 Insufficient Tolerances for Assembly

Another major setback during the integration process was the poor incorporation of proper tolerances in the hull, which was made worse when compounded by the amount of warping and imperfections

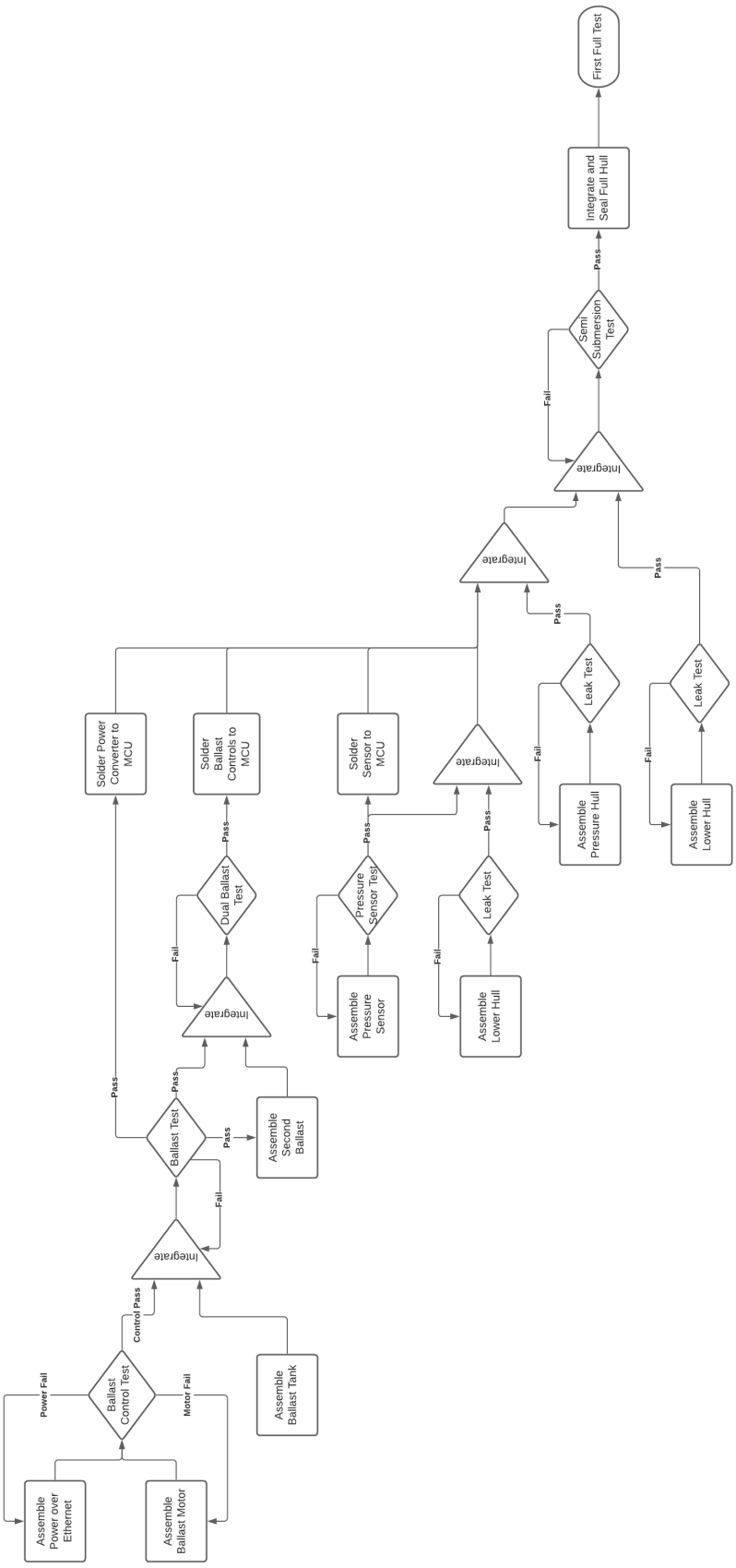


Figure 38: Flow chart highlighting key stages in the integration and testing process

that arose when 3D-printing the hull components. Although the initial test assembly of the hull components by themselves showed no apparent problem, we found that once the XTC-3D epoxy was coated onto the parts, the additional layer as well as increased stiffness in the parts made it extremely difficult for the upper and lower halves of the hull to fit together; this was due to the tight fit in the design of the interface between the upper and lower halves of the hull, and as a result the epoxy after coating acted as a filler and therefore prevented the parts of the hull from fitting together properly.

5.3 Space Constraints in Hull and Pressure Hull

The issue with unaccounted tolerances was worsened by unexpected space constraints internal to the craft, which were only made apparent after we had embedded the electrical components such as the pressure sensor and Ethernet jack and tried to assemble the hull with the all of the subsystem components placed inside. Specifically, the internal cavity of the hull left for the pressure hull was designed to be exactly the same diameter as the pressure hull at 3.5”. This design decision was made in the hopes of decreasing the amount of free space internal to the craft as much as possible so that we could reduce the amount of additional weights needed to calibrate the buoyancy of the craft. However, by doing so, we failed to account the need to route wires alongside the pressure hull, since the wires for the pressure sensor and Ethernet intruded the hull directly above the pressure hull, thus needing to route them so that they could enter the pressure hull through one of the endcaps. This problem, along with the aforementioned lack of tolerances, resulted in the laborious manual removal of hull material so that the hull could actually be assembled for our first full test, and eventually a reprint of the upper half of the hull with these issues accounted for.

Additionally, we were also met with another major obstacle involving space constraints and integration, which was the lack of space that was required for the control linkages of the rudder and elevator. Because motor selection for the ballast and design of the control surfaces came later on in the project after design of the fuselage was complete, we had found out too late that the size of the motors and their respective linkages needed to meet the requirements of the ballast and rear control

surfaces were larger than what we could afford in the space leftover after the pressure hull and ballasts. This issue was temporarily resolved through the use of a hull extension, and we expect that this problem would naturally resolve itself once our prototype could be scaled up to the size that we originally had in mind, as the space allocated to the electronics would increase at a lesser magnitude compared to the hull.

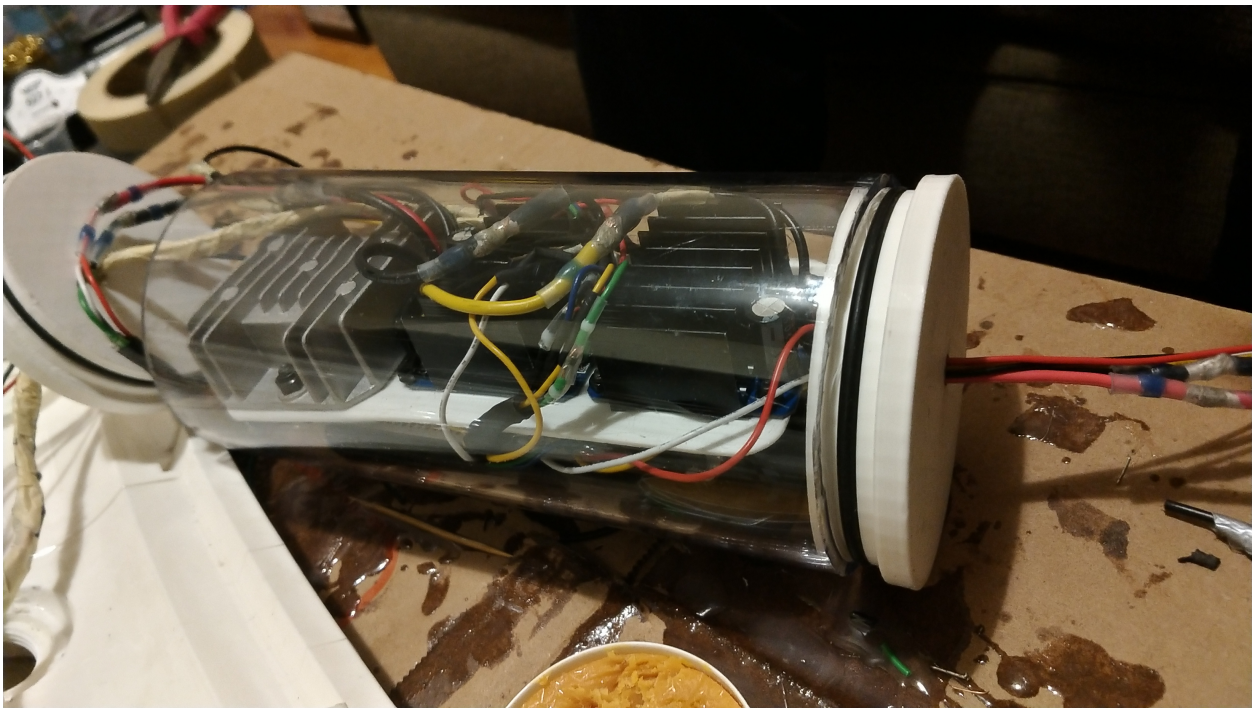


Figure 39: First iteration Pressure Hull with electronics inside

6 Results

6.1 Subsystem and Integration Testing

In order to build up to a full system test where we could deploy the entire craft in its sealed state underwater, we conducted a series of rigorous tests throughout the integration and assembly of the craft in order to ensure that each subsystem would work with each other and that we could build up to the full test in stages rather than all at once. By doing so, we were able to detect, troubleshoot,

and resolves issues more thoroughly as well as ensure reliability of subsystems through the repeated tests administered each time we moved forward with a stage of integration.

Pressure Hull Waterproof Testing In order to test the pressure hull, the entire assembly was constructed and submerged in a bathtub of water with a depth of approximately 9” for close to 24 hours. The assembled pressure hull and testing environment are shown in Figure 40.

Figure 40: Pressure hull submersion test

After the 24 hour period of submersion, no leaks were observed with the inside of the tube maintaining complete dryness. However, there are a couple notes to point out about the test: first, while the pressure hull was able to successfully achieve a sealed state in this test, being submerged in only 9” of water means that the results of this test can’t ensure that the same watertightness can be achieved when submerged in several feet of water due to the greatly increased amount of water pressure on either end caps; and second, the printed end caps were not coated with the XTC-3D epoxy for this round of testing, which indicates that the concern for water leakage through a printed part was potentially overestimated in design. However, this can only be verified after future testing at greater

depths.

Wakeup System In our first iteration of the craft, we used a portable phone charger to power the microcontroller and sensors. This meant that, when we sealed the craft, the system would be on until we unsealed and unplugged the microcontroller, or the battery ran out of power. To work around this, we came up with the Wakeup System.

The STM32F767 comes with many low-power modes, one of which is called Standby. In standby mode, all peripherals are turned off and the SRAM and registers are wiped. This has the benefit of extremely low power consumption, where at absolute worst conditions, it would draw around 68 µA of current. Our battery was rated at 5000 mA h, which meant it would take years for it to drain in standby. The downside was we would have to wake up the craft every time we plugged it in, and the only way to do that was using the Real-Time Clock and giving it a date & time to provide an interrupt, or giving a signal on one of 5 dedicated wakeup pins.

We chose to do the wakeup pin route, and connected a voltage divider to pin PC13, where one end was connected to the 12V line and the other was connected to ground. When we connected the 24V on topside, the divider would create a 4V signal to wake up the craft. The voltage divider was constructed using a 1 MΩ and 510 kΩ resistor. We ran into issues where it would only intermittently work, and discovered that the Nucleo has built-in 220 kΩ pull-down resistors. When we changed to a 300 kΩ and 510 kΩ resistor setup instead, it worked. If we told it to go into standby while power was still connected, however, it would wake up immediately. If we put it into standby again, it would be unable to wake up. However, if we disconnected power before putting it into standby, it would be able to wake up and sleep as many times as necessary.

In our second iteration, we replaced the phone battery with 12V to 5V power converters, so the wakeup system was discarded.

Semi-Submersion Testing Prior to our full system submersion test after the complete assembly and sealing of the craft, we conducted a semi-submersion test by submerging half of the assembled

craft in a bathtub and testing to make sure that the internal electromechanical subsystems of the craft, including the ballasts, pressure sensor, and internal pressure hull electronics, would all work as expected in simulated conditions where the craft would encounter water. Specifically, we made sure that both of the ballasts would be able to draw water externally through the tubing passed through the bottom of the hull, wakeup and power delivery over the ethernet was working correctly, and data could be successfully read and sent back to the operator's console from the pressure sensor after wakeup. This stage in the integration process was vital to the success of our first full system test of the craft as it served as the final checkpoint before sealing the entire craft, which in essence meant that no further adjustments could be made internally until after the full system test as doing so would break the seal. Figure 41 captures a moment during the semi-submersion test.

Figure 41: Bathtub testing of the craft for the semi-submersion test

6.2 Software Results

Onboard Controls The onboard control software allows us to have fine-tuned control over the ballast and hydrodynamic control surfaces of the craft. This is important for controlling the motion and maintaining stability during operation. The operator interacts with GUI and Command Interface to send instructions to the robot, which processes the instruction messages to perform a real world action, such as changing the fill level of one or both ballast tanks, or changing the angle of one of the hydrodynamic control surfaces.

Operator Console The Operator's console allows the user to view the incoming messages from the craft, and includes the Control Interface which allows the user to send commands to the craft.

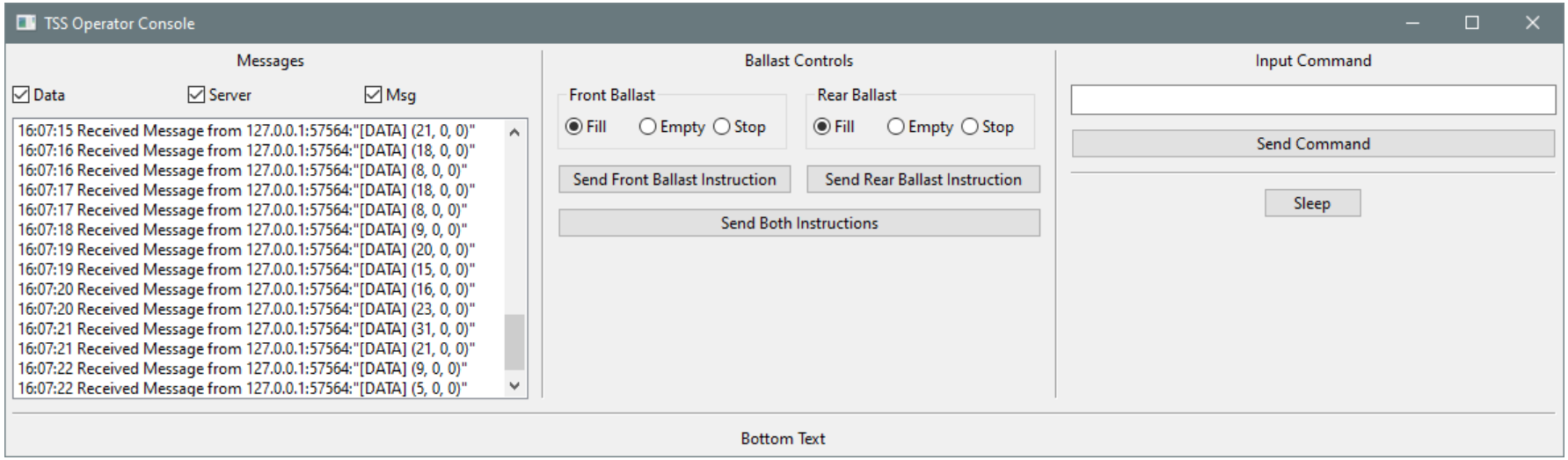


Figure 42: Operator's Console Graphical User Interface

6.3 Full System First Dive

On April 8th, we conducted what we consider to be the very first full system integrated test, where all integration of electrical and mechanical components up until this point was deemed ready to be fully sealed in the hull and deployed in waters beyond the bathtub.

6.3.1 Testing Procedure

We began preparations on April 7th. Our first step was to seal the craft, where we started by sealing the pressure hull, and because this was our first iteration, we put the craft into standby mode. We used hot glue to seal the wires that were entering and leaving the endcaps in case the craft leaked.

Figure 43: The craft just before final checks and sealing

Then, we sealed the craft itself. We put silicone in the connections between the hull and the craft, and then attempted to put the two sections together. This proved to be quite difficult, as the two sections were unable to fit perfectly with the ballasts, leaving gaps in between the two halves as well as in the nosecone and tailcone. We applied a liberal amount of hot glue and silicone to remedy this. Because of the gaps in the craft, we were unable to screw the end-cones in, so we had to duct tape them on to make sure they wouldn't be torn off.

We also put together a checklist of materials to bring with us to the pool. This checklist included:

- Tether with Ethernet Cable
- Topside PoE splitter

Figure 44: Letting the silicone rest for 24 hours before the first test

Figure 45: The craft just before we went to the pool for the first time

- 2 12V Batteries
- 3 sets of alligator clips
- Phone Camera
- The craft itself
- Laptop with control software
- Emergency Duct Tape
- Weights
- Cardboard to put everything on top of

To make sure the tether was both waterproof and unable to come out of the jack, we used hot glue and duct tape. On our way to the pool, the tether popped out and keeping it in was a challenge.

When we got to the pool, we set up the craft and woke it up. After setting up all of the topside infrastructure, namely power from the batteries and communication with the laptop, we did a final check to ensure that the internal electronics were still responsive and working as expected. Once the condition of the craft was ascertained, we lowered the craft into the pool and floated as expected. To begin, we tested the ballasts and made sure that the craft could take in and expel water from both ballasts.

Once we determined that the ballasts were functional, we moved onto conducting a preliminary attempt to calibrate the ballasts, where we attached one of the weights to the craft using a plastic bag and rope. The importance of calibrating the ballasts is due to the default positive buoyancy of the craft since we did not fill in the empty space inside the craft; in order for the ballasts to function properly, we require the craft to be neutrally buoyant when the ballasts are filled to half capacity. At full capacity, the additional water taken in by the ballast would cause the craft to be negatively buoyant and thus sinking the craft, and vice versa, when the ballasts are emptied, the craft should be positively buoyant again and allow it to surface on its own. Thus, we needed to determine the amount additional weight we would need in order for the craft to be neutrally buoyant when the ballasts were filled to half capacity.

During testing, we encountered multiple issues that we were unable to properly resolve in the span of the allotted testing time. First, we were unable to secure the tether such that the Ethernet would remain plugged in and watertight, which resulted in the Ethernet port being filled with water. To remedy this, we tried to cover the port by using three layers of duct tape on it, yet we suspect that this was not enough to fully prevent water from entering. Second, when we sunk the craft, we found that at full ballast, 9.2lb of weights were enough to sink it. However, we were unable to test with any more granular than the single weight of 9.2lbs.

6.3.2 Testing Results

Upon returning, we attempted to connect the craft to the laptop to empty the ballasts and put it in standby mode. We were unable to get an Ethernet connection, which was a big red flag.

When we removed the external silicone and the end-cones, water flowed from the craft, showing that we were unable to seal it properly, but this was not unexpected. When we went to pry it open, however, the top half of the craft broke. It broke in half at the Ethernet area.

Opening the pressure hull, we noticed it was slightly damp and the power brick was off. When we plugged it in, the brick was completely discharged, which was not good considering it was fully charged the day before. When we plugged in the microcontroller to our laptop, we noted that the

Figure 46: The TSS floating in the water for the first time

Figure 47: Upon opening the craft, the top broke in two.

Ethernet port was only lighting up intermittently and that the laptop did not recognize the board. We determined that water had leaked into the pressure hull through the hot-glue sealant as well as possibly leaked into the Ethernet port during the submersion test. We tested the power converter, which seemed to still work. We did not test the motor controllers, but decided to use new ones anyway.

Planning for the next iteration began almost immediately. We decided to reprint the craft, thinning it out such that it would be able to fit everything inside. We also got to work redesigning the electrical system.

In the new electrical system, we used stranded wire rather than solid core. Our first iteration used only solid core, which led to many wires breaking due to the strain that they were under, whereas stranded is far more malleable and will not break when we bend it.

We also decided to design the system with a series of quick-disconnects - on the outside of the craft, we can connect the ballasts, Ethernet, pressure sensors, and ailerons to connectors, and do the same on the interior of the craft. That way, we are able to remove the ballasts, endcaps, and electronics without having to keep everything in close proximity. This is explained in more detail in section 4.3.3.

6.4 Full System Second Dive

On May 6th, we conducted our second and final full system integrated test. After the first full system test, we determined that it was not essential to deploy the craft in the pool as it would fit in the bathtub without the wings.

6.4.1 Testing Procedure

The preparations for this test were similar to the preparations we made in the previous full dive test. Our goals for this test were to demonstrate buoyancy control of the craft through filling and emptying of the ballast. The tolerances and fit between components of the hull were improved in this iteration

of the craft, meaning that we did not have any large gaps in the interfaces between parts that needed to be filled with hot glue; we sealed the craft fully with silicone and screwed the nosecones in rather than using duct tape to fasten together the hull. The craft post-sealing can be found in Figure 48.

There were multiple improvements beyond the QDS (discussed in Section 4.3.3) - we redesigned the tether attachment point to go into the hull itself and sealed the outside of the tether, as well as powered the MCU through the tether rather than a portable charger. We also redesigned the top part of the hull to have more internal space for the ballasts and pressure hull, and we improved the tolerances between the top and bottom halves of the hull to reduce gaps.

Figure 48: The craft post-sealing

We decided to test in the bathtub as we were only focused on testing the craft's ability to independently control its buoyancy and calibrating the in-water weight of the craft to be neutrally buoyant when the ballasts are at half capacity. Thus, the bathtub provided a deep and wide enough area for the test to take place since we did not have to deal with the full wingspan of the craft. We also did not know how long our tests would take, and we would have had a 45 minute time limit in the pool, undercutting the full hour that we needed to complete our test.

We lowered the craft into the bathtub and made sure that the ballasts worked, then sank it manually to test leakage. Right away, we noticed bubbles in the unused grooves of the nosecone allotted for the screws. Since we deemed it only necessary to screw in half of the fastening screws between the nosecone and the middle parts of the hull, the remaining holes were left empty. Therefore, although the area between the nosecone and the hull was sealed, the leftover holes were inadequately sealed and thus allowed for minor flooding of the craft during submersion.

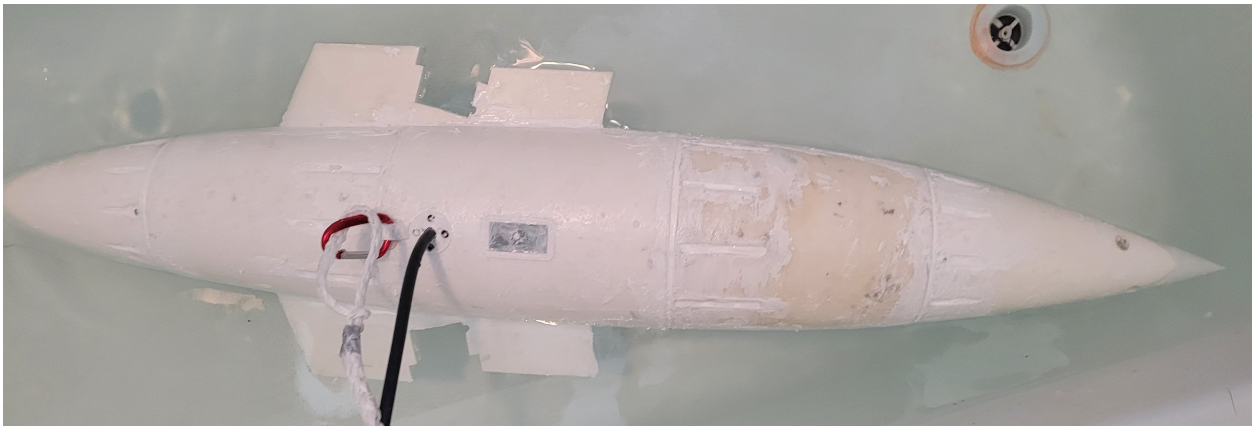


Figure 49: The TSS floating in the bathtub

We also found that the front ballast failed to fill fully, the reason for which we were unable to diagnose during testing. The back ballast also had problems during emptying - it could fill, but it would stop after a second during emptying, and a second command had to be issued for it to continue emptying.

To weigh down the craft, we put together a rig that could hold two bags of ball bearings, which can be seen in Figure 50. This held two bags of ball bearings that we could adjust the offset weight needed to calibrate the buoyancy of the craft to be neutrally buoyant and allow the ballasts to change the craft to be either positive and negative buoyancy.

6.4.2 Testing Results

We were able to get the craft to sink a few inches underwater on its own power, using approximately 8.2lb of ball bearings divided equally between the two sides of the rig. We could not sink it further due to the flooding in the hull as well as failure of the intake cap for the rear ballast. Due to the failed ballasts, the weight required to sink the TSS has a margin of error of around +/-1.7lb, which is the total weight of water that the ballasts can intake.

While the leak in the nosecone was the most obvious, we also noticed some leaks at the tether attachment point as well as the tail-cone. Upon unscrewing all 3 of them, water began to flow out, leading us to believe that there were sealing issues in the interfaces different parts of the hull.

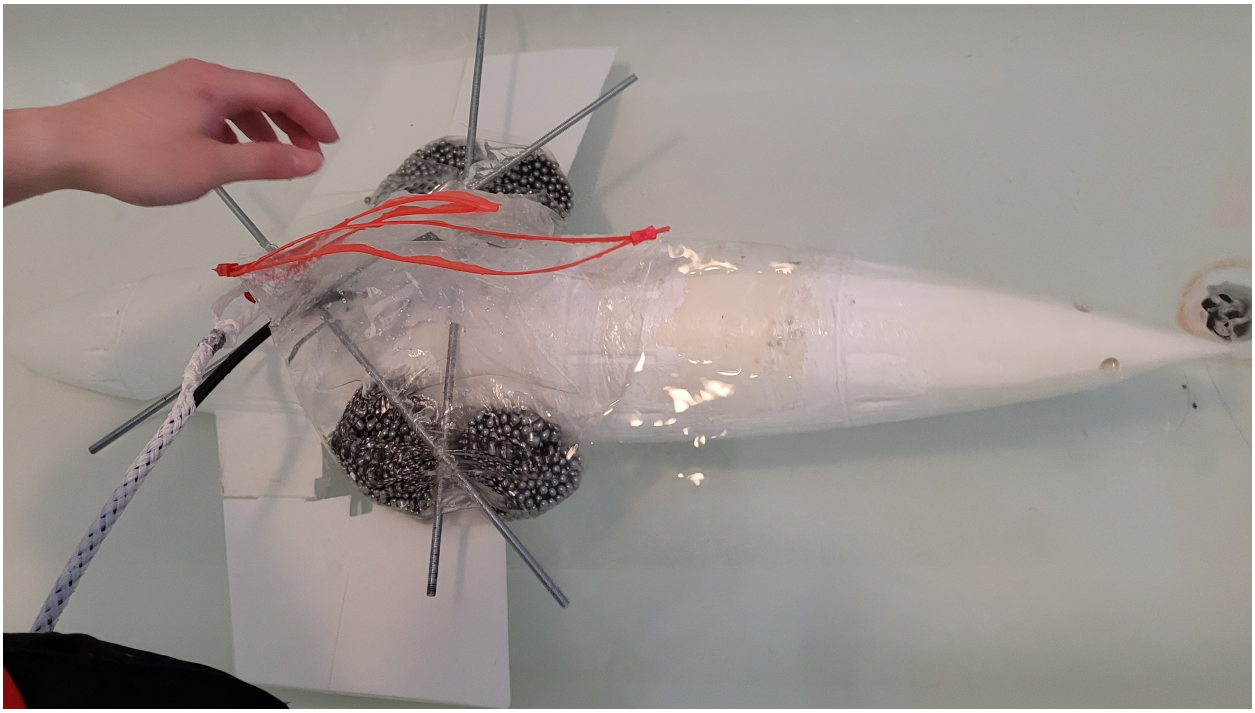


Figure 50: The TSS with the ball-bearing rig

Upon disassembling the craft, the hull once again fractured at the lower-mid aft section, as seen in Figure 51. To fix this problem, the craft needs to be redesigned with internal waterproofing such that it doesn't need internal silicone, thus making it easier to pull apart. Additionally, the hull components could be further reinforced, as the XTC-3D epoxy provided only a surface finish and the hull components by themselves inherently are weakened by 3D-printing faults and the 50% infill they were printed at.

We also found that water had leaked into the pressure hull. We had done a test where we filled up the pressure hull with water, sealed it, and let it sit for a few hours, which did not exhibit signs of leakage. However, we did not do a full-submersion test as we had done in our first iteration of the pressure hull. Luckily, the entry of water into the pressure hull did not noticeably damage the electronics, as the pressure hull stayed upright throughout the test and all the water congregated at the bottom away from the rest of the electronics. We suspect that the leakage was due to inadequate sealing at the endcaps for the QDS, and in the future, a full-submersion test of the pressure hull needs to be performed and the waterproofing needs to be fixed accordingly.

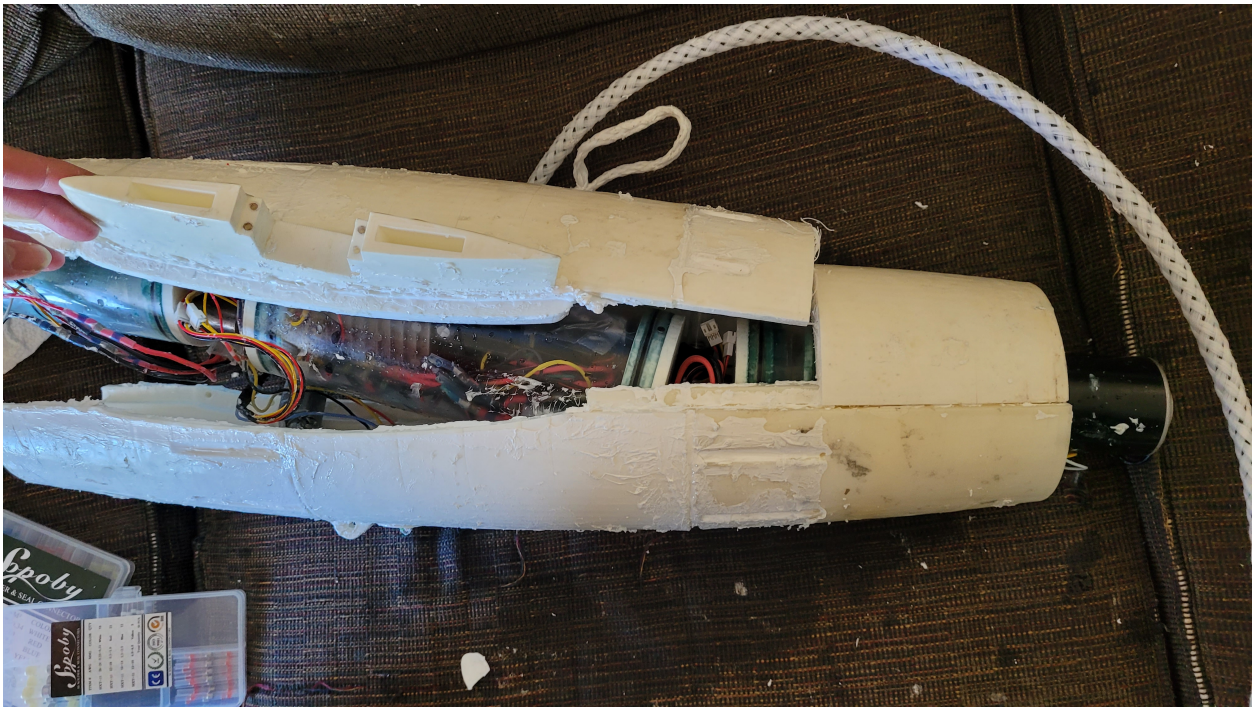


Figure 51: The TSS after its second test

Figure 52: A few millimeters of water leaked into the pressure hull.

Upon inspection of the rear ballast, we found that the fitting on the front end cap had actually separated the top layer of the part, resulting a hole that water could escape from as shown in Figure 53. This is what caused the rear ballast to fail us at the end of the test. To fix this issue, the endcap should be reinforced with epoxy and printed at a higher infill percentage beyond the 20%, ideally greater than 50%, to make sure it is structurally sound.

Figure 53: The broken front endcap on the rear ballast

7 Conclusions

7.1 Summary of Goals

- Can control and maintain depth setpoint down to 20 meters depth
- Is controllable and operable underwater.
- Waterproofing is sufficient to ensure no uncontrolled leakage at 2 meters depth

- Environmental sensing capability including depth and inertial measurements
- Basic control software including actuation of control surfaces
- Depth modulation using a ballast tank

We partially met two of the critical mission goals: depth modulation using ballast tanks and operable underwater. We could somewhat control depth, but due to leakage and insufficient calibration, we could not control the depth fully. We were almost able to waterproof the entire craft, but due to small oversights with the nosecone and tail-cone, we were unable to fully waterproof the craft. We did not include environmental sensing capabilities, and our control software was limited to the ballasts as the control surfaces were not finished in time for the end of the project.

7.2 Future Work

Hull Design A major improvement to the project would be a more rigorous airfoil design and expanded hull volume to accommodate a larger pressure hull and more electronics. More expansive hydrodynamic simulation would be useful for increasing performance. A continuation project could first directly scale up the existing design.

Control Systems While control surfaces and system dynamics were partially modeled, a full system model was never developed. Additional control code is necessary to operate and maneuver the system effectively.

Custom Electronics Off the shelf electronics accelerated development time but also consumed much of our constrained pressure hull space. Custom hardware for voltage regulation and perhaps a simple motor controller would be useful to reduce electronics volume and increase performance.

Sensors More sensing capabilities should be added to increase overall capabilities. To accomplish the navigation tasks, we propose a scanning sonar sensor, additional IMU capabilities, internal sens-

ing to monitor pressure hull moisture and temperature, and better position sensing for the ballast plungers.

Live Location Mapping A live location display over a map of the current surrounding area would allow an operator not only to have awareness and visualization of the subsurface conditions surrounding the craft, but also for the surface surrounding the towing vessel. This interface would work similarly to mobile Map software, and could use geofencing concepts to inform the operator about important weather conditions, national borders, restricted areas, and sites of scientific or historic interest. One software suite we identified that could help with this goal is called OpenCPN, and is used by many mariners to help with navigation.

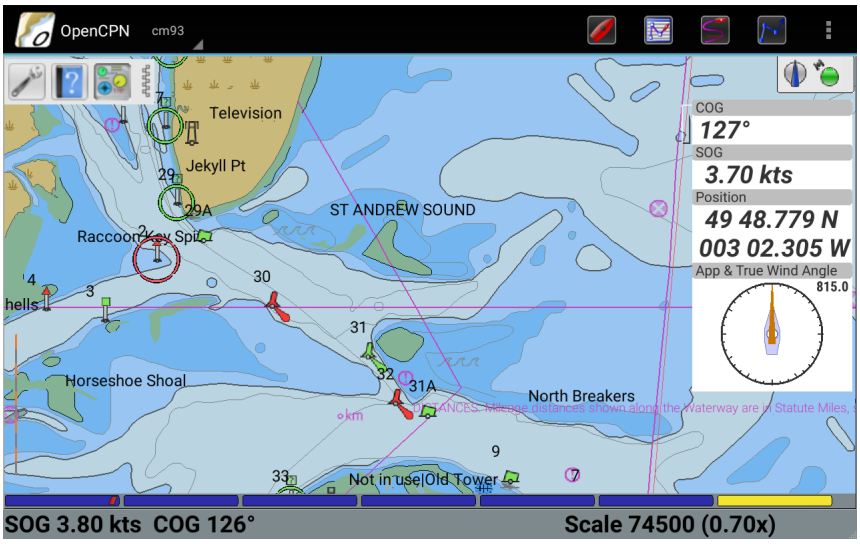


Figure 54: Screen Capture from OpenCPN's Mobile App

Object Detection with Sonar Incorporating a sonar transducer would allow our robot to detect objects its vicinity. Combining sonar with specialized software incorporated into the Operator's Console, we could include a basic object detection and seafloor mapping functionality. This would give more insight to the operator about the local subsurface physical conditions surrounding the craft, and would prove useful in the search for objects or sites of value.

Flooding the Craft An idea that came up very early in the design of the craft was allowing the hull itself to flood, and making sure that the internal components were perfectly waterproof. We decided to focus on external waterproofing, which turned out to be far harder than anticipated. If a future team decides to flood the hull, the ballast needs to be reworked such that it is fully watertight on both ends, and the motor needs to be properly waterproofed.

Redesign of Operators Console During the integration process, we determined that the Operator Console performance was inadequate for a user's needs. The console struggles to handle incoming data at high rates. When the TSS Control Software provided rapid data output, it caused freezing and crashing in the Operator Console, leading to a frustrating testing process. A future extension of this project could focus on improving the performance and user experience of the Operator Console in order to ensure a smooth experience and effective command capability.

An additional improvement for the Operator Console is to increase the readability of the incoming data. The current operator console only displays the messages from the TSS Control Software in raw format. This is useful for debugging purposes, but unhelpful for functional operation. A better data visualization component that uses charts, graphs, or other standard data visualization methods would benefit the Operator Console in terms of readability, ease-of-use, and overall aesthetics. This could potentially be achieved using a standard python data visualization library, such as Matplotlib. Furthermore, implementing such data visualization of sensor data in real time would doubly enhance the utility that the Operator Console provides a user.

8 References

[1] *About the Convention on the Protection of the Underwater Cultural Heritage*. UNESCO, 2001. URL: http://www.unesco.org/new/en/culture/themes/underwater-cultural-heritage/2001-convention/. (accessed: 2020-10-19).

[2] M. Aras et al. "Auto depth control for underwater remotely operated vehicles using a flexible ballast tank system". In: 7.1 (June 2015). (accessed: 2020-10-19).

[3] L. Crafton. *Under the Sea: Maritime Law and Treasure Hunting*. PBS, July 2014. URL: https://www.pbs.org/wgbh/roadshow/stories/articles/2014/7/7/maritime-law-and-treasure-hunting-resources. (accessed: 2020-10-19).

[4] Saal Das. "Reduction of Induced Drag by Single Slotted Raked Wingtip". In: *International Journal of Innovative Research in Science, Engineering and Technology* 04 (Mar. 2015), pp. 1008–1017. DOI: 10.15680/IJIRSET.2015.0403041.

[5] Parker Hannifin Corporation O-Ring Division. "Static O-Ring Sealing". In: *Parker O-Ring Handbook ORD 5700*. Parker Hannifin Corporation, 2018, pp. 4-2–4-8.

[6] *Keep Calm And Carry On: This Software Code Can Protect Subsea Rigs From Hurricanes*. General Electric. URL: https://www.ge.com/news/reports/keep-calm-and-carry-on-this-software-code-can-protect-subsea-rigs-from-hurricanes. (accessed: 2020-10-19).

[7] MatWeb. *Overview of materials for Acrylonitrile Butadiene Styrene (ABS), Extruded*. URL: http://www.matweb.com/search/DataSheet.aspx?MatGUID=3a8afcddac864d4b8f58d40570d2e5aa.

[8] MatWeb. *Overview of materials for PETG Copolyester*. URL: http://www.matweb.com/search/datasheet.aspx?matguid=4de1c85bb946406a86c52b688e3810d0.

[9] MatWeb. *Overview of materials for Polypropylene with 30% Glass Fiber Filler*. URL: http://www.matweb.com/search/DataSheet.aspx?MatGUID=e2a28c79390342e4904502d582e69c62.

[10] MatWeb. *Overview of materials for Polypropylene, Extrusion Grade*. URL: http://www.matweb.com/search/DataSheet.aspx?MatGUID=a882a1c603374e278d062f106dfda95b.

[11] C. Newman. *Finders Keepers? Not Always in Treasure Hunting*. National Geographic, Mar. 2001. URL: https://www.nationalgeographic.com/news/2013/3/130306-finders-keepers-treasure-hunting-law-uk-us/. (accessed: 2020-10-19).

[12] *Pulse 12 Metal Detector*. JW Fishers. URL: http://jwfishers.com/products/p12.html. (accessed: 2020-10-19).

[13] *Remote Environmental Monitoring Unit System (REMUS)*. navaldrones.com, 2016. URL: http://www.navaldrones.com/Remus.html. (accessed: 2020-10-17).

[14] *Remote Environmental Monitoring Unit System (REMUS)*. NavalDrones. URL: http://www.navaldrones.com/Remus.html. (accessed: 2020-10-19).

[15] *Rudders and steering*. Oct. 2015.

[16] Mohammad Sadraey. "Design of Control Surfaces". In: *Aircraft Design: A Systems Engineering Approach*. Wiley Publications, Sept. 2012. Chap. 12.

[17] Mohammad Sadraey. "Tail Design". In: *Aircraft Design: A Systems Engineering Approach*. Wiley Publications, Sept. 2012. Chap. 6.

[18] SDP/SI. *BALL & ACME LEAD SCREW TECHNICAL INFORMATION*. URL: https://www.sdp-si.com/D810/PDFS/Ball%5C%20And%5C%20Acme%5C%20Lead%5C%20Screw%5C%20Technical%5C%20Info.pdf.

[19] Bob Sheaf. *Double-acting telescope cylinder*. Endeavor Business Media LLC., May 2013. URL: https://www.hydraulicspneumatics.com/technologies/maintenance/article/21883356/troubleshooting-challenge-telescoping-doubleacting-cylinder-failure.

[20] *System Calculations - Leadscrew Drives*. Computmotor. URL: http://www.compumotor.com/catalog/catalogA/A60-A62.pdf. (accessed: 2021).

[21] *Threadless Ball Screw*. Makerbot. URL: https://www.thingiverse.com/thing:112718.

[22] B. Tiwari and R. Sharma. "Design and Analysis of a Variable Buoyancy System for Efficient Hovering Control of Underwater Vehicles with State Feedback Controller". In: *Journal of Marine Science and Engineering* 8.4 (Apr. 2020).

[23] M. Triantafyllou. *Maneuvering and Control of Surface and Underwater Vehicles (13.49)*. MIT OpenCourseWare, 2004. URL: https://ocw.mit.edu/courses/mechanical-engineering/2-154-maneuvering-and-control-of-surface-and-underwater-vehicles-13-49-fall-2004/. (accessed: 2020-10-19).

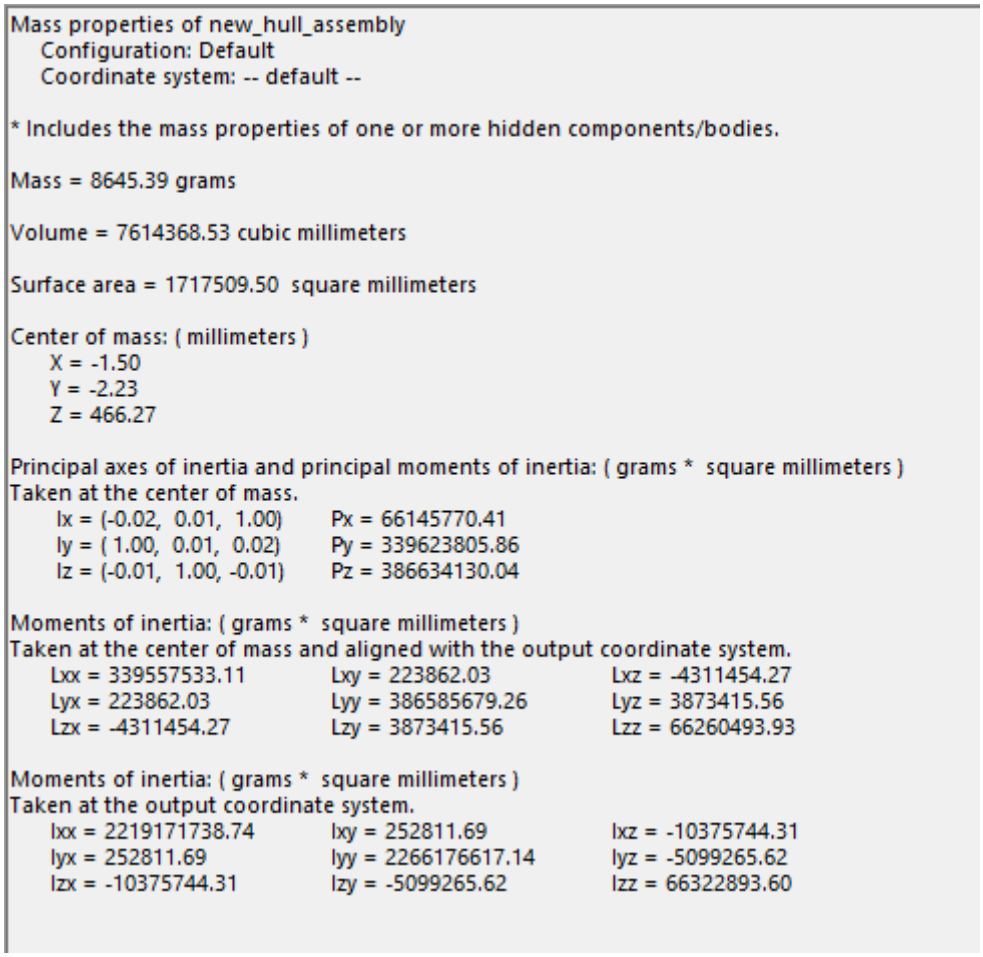
[24] *Types of Rope*. The Home Depot. URL: https://www.homedepot.com/c/ab/types-of-rope/9ba683603be9fa5395fab9020598ae9. (accessed: 2020-10-19).

[25] *Winch & Cable*. SV Seeker. URL: https://www.svseeker.com/wp/sv-seeker-2/underwater-rovs/open-source-towed-sonar-rov/winch-cable/. (accessed: 2020-10-19).

Appendix

A Craft Mechanical Properties

The following image shows the mechanical properties of the craft assembly generated using Solidwork's Mass Properties tool, and provides information regarding the approximately mass, volume, and center of mass coordinates for the craft as a whole, including the pressure hull, ballasts, and control surfaces.



Mass properties of new_hull_assembly
Configuration: Default
Coordinate system: -- default --

* Includes the mass properties of one or more hidden components/bodies.

Mass = 8645.39 grams

Volume = 7614368.53 cubic millimeters

Surface area = 1717509.50 square millimeters

Center of mass: (millimeters)
X = -1.50
Y = -2.23
Z = 466.27

Principal axes of inertia and principal moments of inertia: (grams * square millimeters)
Taken at the center of mass.
Ix = (-0.02, 0.01, 1.00) Px = 66145770.41
Iy = (1.00, 0.01, 0.02) Py = 339623805.86
Iz = (-0.01, 1.00, -0.01) Pz = 386634130.04

Moments of inertia: (grams * square millimeters)
Taken at the center of mass and aligned with the output coordinate system.
Lxx = 339557533.11 Lxy = 223862.03 Lxz = -4311454.27
Lyx = 223862.03 Lyy = 386585679.26 Lyz = 3873415.56
Lzx = -4311454.27 Lzy = 3873415.56 Lzz = 66260493.93

Moments of inertia: (grams * square millimeters)
Taken at the output coordinate system.
Ixx = 2219171738.74 Ixy = 252811.69 Ixz = -10375744.31
Iyx = 252811.69 Iyy = 2266176617.14 Iyz = -5099265.62
Izx = -10375744.31 Izy = -5099265.62 Izz = 66322893.60

Figure 55: Mass properties of entire craft assembly generated by Solidworks

B Pressure Sensor Code

```c
/*
 * ms5803.c
 *
 *  Created on: Feb 22, 2021
 *      Author: Ravi Kirschner
 */

#include "ms5803.h"

/**
 * @brief Reads from MS5803
 * @param handle The I2C handle being used
 * @param bufp The buffer to be read into
 * @param len The length of the buffer in 8-bit increments
 * @retval HAL Status
 */
HAL_StatusTypeDef MS5803_read(void *handle, uint8_t *bufp, uint16_t len) {
  return HAL_I2C_Master_Receive(handle, MS5803_ADDR, bufp, len, 1000);
}

/**
 * @brief Writes to MS5803
 * @param handle The I2C handle being used
 * @param bufp The buffer to read from
 * @param len The length of the buffer in 8-bit increments
 * @retval HAL Status
 */
HAL_StatusTypeDef MS5803_write(void *handle, uint8_t *bufp, uint16_t len) {
  return HAL_I2C_Master_Transmit(handle, MS5803_ADDR, bufp, len, 1000);
}
```

```
32 /**
33  * @brief Resets the MS5803
34  * @param handle The I2C Handle being used
35  * @retval HAL Status
36  */
37 HAL_StatusTypeDef MS5803_reset(void *handle) {
38   uint8_t buf[12];
39   buf[0] = MS5803_RESET;
40   HAL_StatusTypeDef ret = HAL_I2C_Master_Transmit(handle, MS5803_ADDR, buf,
         1, 1000);
41   HAL_Delay(3);
42   return ret;
43 }

45 /**
46  * @brief Gets the 6 Coefficients from the MS5803 and reads them into the
       MS5803_coefficient array.
47  * @param handle The I2C Handle being used
48  * @param coeff The pointer to the coefficient being read in to
49  * @param value The coefficient number
50  * @return HAL Status
51  */
52 HAL_StatusTypeDef MS5803_coeff(void *handle, uint16_t *coeff, uint8_t value)
       {
53   uint8_t buf[12];
54   buf[0] = MS5803_PROM + (value << 1); //coefficient to read
55   HAL_StatusTypeDef x = MS5803_write(handle, buf, 1); //tell MS5803 that we
       want it
56   HAL_Delay(2); //delay until it is ready
57   uint8_t c[2];
58   x = MS5803_read(handle, c, 2); //read the coefficient
```

```
59    *coeff = (c[0] << 8) + c[1]; //turn the two 8-bit values into one coherent
          value.
60    return x;
61  }

63  /**
64   * @brief Reads the MS5803 ADC
65   * @param handle The I2C Handle being used
66   * @param type The measurement type, chosen from measurement enum
67   * @param prec The precision to use, chosen from precision enum
68   * @retval Raw 24-bit data from the ADC
69   */
70  uint32_t MS5803_ADC(void *handle, measurement type, precision prec) {
71    uint32_t result;
72    uint8_t buf[12];
73    buf[0] = MS5803_ADC_CONV + type + prec; //tell the ADC to convert along
          with the precision and type
74    HAL_StatusTypeDef x = MS5803_write(handle, buf, 1);
75    HAL_Delay(1);
76    switch(prec) {
77      case ADC_256: HAL_Delay(1);
78      case ADC_512: HAL_Delay(3);
79      case ADC_1024: HAL_Delay(4);
80      case ADC_2048: HAL_Delay(6);
81      case ADC_4096: HAL_Delay(10); //Delay longer if higher precision, as
            conversion takes longer.
82    }
83    buf[0] = MS5803_ADC_READ; //Tell the MS5803 that we want to read the ADC
84    MS5803_write(handle, buf, 1);
85    HAL_Delay(2);
86    uint8_t c[3];
87    MS5803_read(handle, c, 3); //Read out the ADC
```

```
88    result = (c[0] << 16) + (c[1] << 8) + c[2]; //Convert the three 8-bit
          values into one value.
89    return result;
90  }

92  /**
93   * @brief Gets temperature and pressure values from the MS5803
94   * @param handle The I2C Handle being used
95   * @param prec The precision to be used
96   * @param temperature The pointer to the temperature variable being read in
         to.
97   * @param pressure The pointer to the pressure variable being read in to.
98   */
99  void MS5803_get_values(void *handle, precision prec, float *temperature,
       float *pressure) {
100   uint32_t temperature_raw = MS5803_ADC(handle, TEMPERATURE, prec);
101   uint32_t pressure_raw = MS5803_ADC(handle, PRESSURE, prec); //get
          temperature and pressure raw values

103   int32_t sub = MS5803_coefficient[4] * 256;
104   int32_t dT = temperature_raw - sub;
105   int32_t temp = 2000 + (dT*MS5803_coefficient[5])/(8388608);
106   *temperature = temp/100.f; //determine temperature according to datasheet

108   int64_t OFF = ((int64_t)MS5803_coefficient[1])*65536+
109       ((int64_t)MS5803_coefficient[3])*((int64_t)(temperature_raw - sub))
              /128;
110   int64_t SENS = MS5803_coefficient[0] * (32768) + (MS5803_coefficient[2]*dT
         )/(256);
111   int32_t pres = (pressure_raw*SENS/2097152-OFF)/32768;
112   *pressure = pres/10.0f; //determine pressure according to datasheet
113 }
```

C Motor Controller Code

```
/*
 * BTS7960.c
 *
 *  Created on: Mar 17, 2021
 *      Author: Ravi
 */

#include "main.h"
#include "BTS7960.h"

void ballast_rotate(TIM_HandleTypeDef *PWM, ballast number, uint8_t dir,
    uint16_t PWMVAL) {
  if(number == BALLAST1) {
    if(dir == CCW) {
      PWM->Instance->CCR1 = PWMVAL;
      PWM->Instance->CCR3 = 0;
    }
    else if(dir == CW) {
      PWM->Instance->CCR1 = 0;
      PWM->Instance->CCR3 = PWMVAL;
    }
    else {
       ballast_stop(PWM, number);
    }
  }
  else if(number == BALLAST2) {
    if(dir == CCW) {
      PWM->Instance->CCR1 = PWMVAL;
      PWM->Instance->CCR2 = 0;
    }
```

```
30      else if(dir == CW) {
31        PWM->Instance->CCR1 = 0;
32        PWM->Instance->CCR2 = PWMVAL;
33      }
34      else {
35        ballast_stop(PWM, number);
36      }
37    }
38  }

40  void ballast_stop(TIM_HandleTypeDef *PWM, ballast number) {
41    if(number == BALLAST1) {
42      PWM->Instance->CCR1 = 0;
43      PWM->Instance->CCR3 = 0;
44    }
45    else {
46      PWM->Instance->CCR1 = 0;
47      PWM->Instance->CCR2 = 0;
48    }
49  }
50  void ballast_shutoff(ballast number) {
51    if(number == BALLAST1) {
52      HAL_GPIO_WritePin(R_EN1_GPIO_Port, R_EN1_Pin, GPIO_PIN_RESET);
53      HAL_GPIO_WritePin(L_EN1_GPIO_Port, L_EN1_Pin, GPIO_PIN_RESET);
54    }
55    else {
56      HAL_GPIO_WritePin(R_EN2_GPIO_Port, R_EN2_Pin, GPIO_PIN_RESET);
57      HAL_GPIO_WritePin(L_EN2_GPIO_Port, L_EN2_Pin, GPIO_PIN_RESET);
58    }
59  }

61  void ballast_on(TIM_HandleTypeDef *PWM, ballast number) {
62    ballast_stop(PWM, number);
```

```
63    if(number == BALLAST1) {
64      HAL_GPIO_WritePin(R_EN1_GPIO_Port, R_EN1_Pin, GPIO_PIN_SET);
65      HAL_GPIO_WritePin(L_EN1_GPIO_Port, L_EN1_Pin, GPIO_PIN_SET);
66    }
67    else {
68      HAL_GPIO_WritePin(R_EN2_GPIO_Port, R_EN2_Pin, GPIO_PIN_SET);
69      HAL_GPIO_WritePin(L_EN2_GPIO_Port, L_EN2_Pin, GPIO_PIN_SET);
70    }
71  }
```

D Main Control Code

```
1   /* USER CODE BEGIN Header */
2   /**
3     ******************************************************************************

4     * @file           : main.c
5     * @brief          : Main program body
6     ******************************************************************************

7     * @attention
8     *
9     * <h2><center>&copy; Copyright (c) 2021 STMicroelectronics.
10    * All rights reserved.</center></h2>
11    *
12    * This software component is licensed by ST under BSD 3-Clause license,
13    * the "License"; You may not use this file except in compliance with the
14    * License. You may obtain a copy of the License at:
15    *                        opensource.org/licenses/BSD-3-Clause
16    *
```

```
17    ******************************************************************************

18    */
19  /* USER CODE END Header */
20  /* Includes
        ------------------------------------------------------------------*/
21  #include "main.h"
22  #include "lwip.h"

24  /* Private includes
        ----------------------------------------------------------*/
25  /* USER CODE BEGIN Includes */
26  #include "udp_client.h"
27  #include "BTS7960.h"
28  #include "ms5803.h"
29  #include <string.h>
30  /* USER CODE END Includes */

32  /* Private typedef
        -----------------------------------------------------------*/
33  /* USER CODE BEGIN PTD */

35  /* USER CODE END PTD */

37  /* Private define
        ------------------------------------------------------------*/
38  /* USER CODE BEGIN PD */
39  /* USER CODE END PD */

41  /* Private macro
        -------------------------------------------------------------*/
42  /* USER CODE BEGIN PM */
```

```
44 /* USER CODE END PM */
46 /* Private variables
       ---------------------------------------------------------*/
48 I2C_HandleTypeDef hi2c2;
50 TIM_HandleTypeDef htim2;
51 TIM_HandleTypeDef htim3;
52 TIM_HandleTypeDef htim4;
54 UART_HandleTypeDef huart3;
56 PCD_HandleTypeDef hpcd_USB_OTG_FS;
58 /* USER CODE BEGIN PV */
60 /* USER CODE END PV */
62 /* Private function prototypes
       -----------------------------------------------*/
63 void SystemClock_Config(void);
64 static void MX_GPIO_Init(void);
65 static void MX_USART3_UART_Init(void);
66 static void MX_USB_OTG_FS_PCD_Init(void);
67 static void MX_TIM2_Init(void);
68 static void MX_TIM3_Init(void);
69 static void MX_I2C2_Init(void);
70 static void MX_TIM4_Init(void);
71 /* USER CODE BEGIN PFP */
73 /* USER CODE END PFP */
```

```
75 /* Private user code
       ---------------------------------------------------------*/
76 /* USER CODE BEGIN 0 */

78 /* USER CODE END 0 */

80 /**
81   * @brief  The application entry point.
82   * @retval int
83   */
84 int main(void)
85 {
86   /* USER CODE BEGIN 1 */

88   /* USER CODE END 1 */

90   /* MCU Configuration
         ---------------------------------------------------------*/

92   /* Reset of all peripherals, Initializes the Flash interface and the
        Systick. */
93   HAL_Init();

95   /* USER CODE BEGIN Init */

97   /* USER CODE END Init */

99   /* Configure the system clock */
100  SystemClock_Config();

102  /* USER CODE BEGIN SysInit */

104  /* USER CODE END SysInit */
```

```
106   /* Initialize all configured peripherals */
107   MX_GPIO_Init();
108   MX_USART3_UART_Init();
109   MX_USB_OTG_FS_PCD_Init();
110   MX_TIM2_Init();
111   MX_LWIP_Init();
112   MX_TIM3_Init();
113   MX_I2C2_Init();
114   MX_TIM4_Init();
115   /* USER CODE BEGIN 2 */
116   command_buf = (char *)malloc(sizeof(char) * 63);
117   char* value = (char*)malloc(41*sizeof(char));
118   *command_buf = NULL;
119   ip4_addr_t ipaddr;
120   IP4_ADDR(&ipaddr, 192, 168, 0, 2);
121   udp_echoclient_connect(&ipaddr, 8888);
122   ballast_on(&htim2, BALLAST1);
123   ballast_on(&htim4, BALLAST2);
124   HAL_TIM_Base_Start_IT(&htim3);
125   HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN4);



128   MS5803_reset(&hi2c2); //reset
129   HAL_Delay(2);
130   for(int i = 1; i <= 6; i++) {
131     MS5803_coeff(&hi2c2, &MS5803_coefficient[i-1], i); //get coefficients
132   }
133   uint8_t wantToSleep = 0;
134   uint8_t slept = 1;
135     if(__HAL_PWR_GET_FLAG(PWR_FLAG_SB)) {
136       __HAL_PWR_CLEAR_FLAG(PWR_FLAG_SB);
137       PWR->CR2|=PWR_CR2_CWUPF4;
```

```
138         wantToSleep = 0;
139         slept = 1;
140         MS5803_reset(&hi2c2);
141       }
142       //BALLAST 1
143       uint8_t dir_val1 = 0;
144       uint16_t pwm_val1 = 0;
145       uint8_t newdir_val1 = 0;
146       uint16_t newpwm_val1 = 0;
147       //BALLAST2
148       uint8_t dir_val2 = 0;
149       uint16_t pwm_val2 = 0;
150       uint8_t newdir_val2 = 0;
151       uint16_t newpwm_val2 = 0;

153     /* USER CODE END 2 */

155     /* Infinite loop */
156     /* USER CODE BEGIN WHILE */
157     while (1)
158     {
159       /* USER CODE END WHILE */

161       /* USER CODE BEGIN 3 */
162       //BALLAST1
163       uint8_t RightLimit1 = !HAL_GPIO_ReadPin(LIMITR_1_GPIO_Port, LIMITR_1_Pin
              );
164       uint8_t LeftLimit1 = !HAL_GPIO_ReadPin(LIMITL_1_GPIO_Port, LIMITL_1_Pin)
              ;
165       //BALLAST2
166       uint8_t RightLimit2 = !HAL_GPIO_ReadPin(LIMITR_2_GPIO_Port, LIMITR_2_Pin
              );
```

```
167     uint8_t LeftLimit2 = !HAL_GPIO_ReadPin(LIMITL_2_GPIO_Port, LIMITL_2_Pin)
            ;
168     MX_LWIP_Process();
169     if(*command_buf){
170       if(strstr(command_buf, "FillBoth") != NULL) {
171         newdir_val1 = CCW;
172         newpwm_val1 = 125;
173         newdir_val2 = CCW;
174         newpwm_val2 = 125;
175       }
176       else if(strstr(command_buf, "EmptyBoth") != NULL) {
177         newdir_val1 = CW;
178         newpwm_val1 = 125;
179         newdir_val2 = CW;
180         newpwm_val2 = 125;
181       }
182       else if(strstr(command_buf, "StopBoth") != NULL) {
183         newpwm_val1 = 0;
184         newpwm_val2 = 0;
185       }
186       else if(strstr(command_buf, "FillFront") != NULL) {
187         newdir_val1 = CCW;
188         newpwm_val1 = 125;
189       }
190       else if(strstr(command_buf, "FillBack") != NULL) {
191         newdir_val2 = CCW;
192         newpwm_val2 = 125;
193       }
194       else if(strstr(command_buf, "EmptyFront") != NULL) {
195         newdir_val1 = CW;
196         newpwm_val1 = 125;
197       }
198       else if(strstr(command_buf, "EmptyBack") != NULL) {
```

```
199         newdir_val2 = CW;
200         newpwm_val2 = 125;
201       }
202       else if(strstr(command_buf, "StopFront") != NULL) {
203         newpwm_val1 = 0;
204       }
205       else if(strstr(command_buf, "StopBack") != NULL) {
206         newpwm_val2 = 0;
207       }
208       else if(strstr(command_buf, "Sleep") != NULL) {
209         wantToSleep = 1;
210       }
211       else if(strstr(command_buf, "ResetPressure") != NULL) {
212         HAL_StatusTypeDef test = MS5803_reset(&hi2c2);
213         HAL_Delay(2);
214         for(int i = 1; i <= 6; i++) {
215           MS5803_coeff(&hi2c2, &MS5803_coefficient[i-1], i); //get
                coefficients
216         }
217       }
218       *command_buf = NULL;
219     }
220     if(slept) {
221       udp_echoclient_send("MSG: AWAKE");
222       slept = 0;
223     }
224     if(READMS5803) {
225       READMS5803 = 0;
226       MS5803_get_values(&hi2c2, ADC_512, &temperature, &pressure); //get
            values.
227       sprintf(value, "DATA: temp: %.2f, pres: %.2f; ", temperature, pressure
            );
228       udp_echoclient_send(value);
```

```
229     }
230
231     //BALLAST FRONT
232     if(RightLimit1 && (dir_val1 == CCW && newdir_val1 == CCW)) {
233       ballast_stop(&htim2, BALLAST1);
234       pwm_val1 = 0;
235       newpwm_val1 = 0;
236       udp_echoclient_send("MSG: LIMIT FILLED, BALLAST FRONT");
237     }
238     else if (LeftLimit1 && (dir_val1 == CW && newdir_val1 == CW)) {
239       ballast_stop(&htim2, BALLAST1);
240       pwm_val1 = 0;
241       newpwm_val1 = 0;
242       udp_echoclient_send("MSG: LIMIT EMPTY, BALLAST FRONT");
243     }
244     //BALLAST BACK
245     if(RightLimit2 && (dir_val2 == CCW && newdir_val2 == CCW)) {
246       ballast_stop(&htim4, BALLAST2);
247       pwm_val2 = 0;
248       newpwm_val2 = 0;
249       udp_echoclient_send("MSG: LIMIT FILLED, BALLAST BACK");
250     }
251     else if (LeftLimit2 && (dir_val2 == CW && newdir_val2 == CW)) {
252       ballast_stop(&htim4, BALLAST2);
253       pwm_val2 = 0;
254       newpwm_val2 = 0;
255       udp_echoclient_send("MSG: LIMIT EMPTY, BALLAST BACK");
256     }
257     while(1) {
258       ballast_rotate(&htim2, BALLAST1, dir_val1, pwm_val1);
259       ballast_rotate(&htim4, BALLAST2, dir_val2, pwm_val2);
260       if((dir_val1 == newdir_val1 && pwm_val1 == newpwm_val1) &&
261           (dir_val2 == newdir_val2 && pwm_val2 == newpwm_val2)) break;
```

```
262       if(dir_val1 != newdir_val1 && pwm_val1 > 0) {
263         pwm_val1--;
264       }
265       else if(dir_val1 != newdir_val1 && pwm_val1 == 0) {
266         dir_val1 = newdir_val1;
267       }
268       else if(dir_val1 == newdir_val1 && newpwm_val1 > pwm_val1) {
269         pwm_val1++;
270       }
271       else if(dir_val1 == newdir_val1 && pwm_val1 > newpwm_val1) {
272         pwm_val1--;
273       }
274       if(dir_val2 != newdir_val2 && pwm_val2 > 0) {
275         pwm_val2--;
276       }
277       else if(dir_val2 != newdir_val2 && pwm_val2 == 0) {
278         dir_val2 = newdir_val2;
279       }
280       else if(dir_val2 == newdir_val2 && newpwm_val2 > pwm_val2) {
281         pwm_val2++;
282       }
283       else if(dir_val2 == newdir_val2 && pwm_val2 > newpwm_val2) {
284         pwm_val2--;
285       }
286       HAL_Delay(2);
287     }
288     if(wantToSleep) {
289       udp_echoclient_send("MSG: SLEEPING");
290       RCC->APB1ENR |= (RCC_APB1ENR_PWREN);
291       HAL_PWR_EnterSTANDBYMode();
292     }
293   }
294   /* USER CODE END 3 */
```

```
295 }
297 /**
298   * @brief System Clock Configuration
299   * @retval None
300   */
301 void SystemClock_Config(void)
302 {
303   RCC_OscInitTypeDef RCC_OscInitStruct = {0};
304   RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
305   RCC_PeriphCLKInitTypeDef PeriphClkInitStruct = {0};
307   /** Configure LSE Drive Capability
308   */
309   HAL_PWR_EnableBkUpAccess();
310   /** Configure the main internal regulator output voltage
311   */
312   __HAL_RCC_PWR_CLK_ENABLE();
313   __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE3);
314   /** Initializes the RCC Oscillators according to the specified parameters
315   * in the RCC_OscInitTypeDef structure.
316   */
317   RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
318   RCC_OscInitStruct.HSEState = RCC_HSE_ON;
319   RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
320   RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
321   RCC_OscInitStruct.PLL.PLLM = 4;
322   RCC_OscInitStruct.PLL.PLLN = 96;
323   RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
324   RCC_OscInitStruct.PLL.PLLQ = 4;
325   if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
326   {
327     Error_Handler();
```

```
328   }
329   /** Activate the Over-Drive mode
330   */
331   if (HAL_PWREx_EnableOverDrive() != HAL_OK)
332   {
333     Error_Handler();
334   }
335   /** Initializes the CPU, AHB and APB buses clocks
336   */
337   RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
338                                 |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
339   RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
340   RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
341   RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
342   RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

344   if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_3) != HAL_OK)
345   {
346     Error_Handler();
347   }
348   PeriphClkInitStruct.PeriphClockSelection = RCC_PERIPHCLK_USART3|
        RCC_PERIPHCLK_I2C2
349                                 |RCC_PERIPHCLK_CLK48;
350   PeriphClkInitStruct.Usart3ClockSelection = RCC_USART3CLKSOURCE_PCLK1;
351   PeriphClkInitStruct.I2c2ClockSelection = RCC_I2C2CLKSOURCE_PCLK1;
352   PeriphClkInitStruct.Clk48ClockSelection = RCC_CLK48SOURCE_PLL;
353   if (HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct) != HAL_OK)
354   {
355     Error_Handler();
356   }
357 }

359 /**
```

```
360   * @brief I2C2 Initialization Function
361   * @param None
362   * @retval None
363   */
364 static void MX_I2C2_Init(void)
365 {
366
367   /* USER CODE BEGIN I2C2_Init 0 */
368
369   /* USER CODE END I2C2_Init 0 */
370
371   /* USER CODE BEGIN I2C2_Init 1 */
372
373   /* USER CODE END I2C2_Init 1 */
374   hi2c2.Instance = I2C2;
375   hi2c2.Init.Timing = 0x20303E5D;
376   hi2c2.Init.OwnAddress1 = 0;
377   hi2c2.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
378   hi2c2.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
379   hi2c2.Init.OwnAddress2 = 0;
380   hi2c2.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
381   hi2c2.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
382   hi2c2.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
383   if (HAL_I2C_Init(&hi2c2) != HAL_OK)
384   {
385     Error_Handler();
386   }
387   /** Configure Analogue filter
388   */
389   if (HAL_I2CEx_ConfigAnalogFilter(&hi2c2, I2C_ANALOGFILTER_ENABLE) !=
        HAL_OK)
390   {
391     Error_Handler();
```

```
392   }
393   /** Configure Digital filter
394   */
395   if (HAL_I2CEx_ConfigDigitalFilter(&hi2c2, 0) != HAL_OK)
396   {
397     Error_Handler();
398   }
399   /* USER CODE BEGIN I2C2_Init 2 */

401   /* USER CODE END I2C2_Init 2 */

403 }

405 /**
406   * @brief TIM2 Initialization Function
407   * @param None
408   * @retval None
409   */
410 static void MX_TIM2_Init(void)
411 {

413   /* USER CODE BEGIN TIM2_Init 0 */

415   /* USER CODE END TIM2_Init 0 */

417   TIM_ClockConfigTypeDef sClockSourceConfig = {0};
418   TIM_MasterConfigTypeDef sMasterConfig = {0};
419   TIM_OC_InitTypeDef sConfigOC = {0};

421   /* USER CODE BEGIN TIM2_Init 1 */

423   /* USER CODE END TIM2_Init 1 */
424   htim2.Instance = TIM2;
```

```
425   htim2.Init.Prescaler = 15;
426   htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
427   htim2.Init.Period = 299;
428   htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
429   htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
430   if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
431   {
432     Error_Handler();
433   }
434   sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
435   if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
436   {
437     Error_Handler();
438   }
439   if (HAL_TIM_PWM_Init(&htim2) != HAL_OK)
440   {
441     Error_Handler();
442   }
443   sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
444   sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
445   if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) !=
        HAL_OK)
446   {
447     Error_Handler();
448   }
449   sConfigOC.OCMode = TIM_OCMODE_PWM1;
450   sConfigOC.Pulse = 0;
451   sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
452   sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
453   if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1) != HAL_OK
        )
454   {
455     Error_Handler();
```

```
456   }
457   if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_3) != HAL_OK
          )
458   {
459     Error_Handler();
460   }
461   /* USER CODE BEGIN TIM2_Init 2 */
462   HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
463   HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_3);
464   /* USER CODE END TIM2_Init 2 */
465   HAL_TIM_MspPostInit(&htim2);
466
467 }
468
469 /**
470   * @brief TIM3 Initialization Function
471   * @param None
472   * @retval None
473   */
474 static void MX_TIM3_Init(void)
475 {
476
477   /* USER CODE BEGIN TIM3_Init 0 */
478
479   /* USER CODE END TIM3_Init 0 */
480
481   TIM_ClockConfigTypeDef sClockSourceConfig = {0};
482   TIM_MasterConfigTypeDef sMasterConfig = {0};
483
484   /* USER CODE BEGIN TIM3_Init 1 */
485
486   /* USER CODE END TIM3_Init 1 */
487   htim3.Instance = TIM3;
```

```
488   htim3.Init.Prescaler = 65535;
489   htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
490   htim3.Init.Period = 30;
491   htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
492   htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
493   if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
494   {
495     Error_Handler();
496   }
497   sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
498   if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
499   {
500     Error_Handler();
501   }
502   sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
503   sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
504   if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) !=
        HAL_OK)
505   {
506     Error_Handler();
507   }
508   /* USER CODE BEGIN TIM3_Init 2 */
509   HAL_TIM_PWM_Start(&htim4, TIM_CHANNEL_1);
510   HAL_TIM_PWM_Start(&htim4, TIM_CHANNEL_2);
511   /* USER CODE END TIM3_Init 2 */

513 }

515 /**
516   * @brief TIM4 Initialization Function
517   * @param None
518   * @retval None
519   */
```

```
520 static void MX_TIM4_Init(void)
521 {
523   /* USER CODE BEGIN TIM4_Init 0 */
525   /* USER CODE END TIM4_Init 0 */
527   TIM_MasterConfigTypeDef sMasterConfig = {0};
528   TIM_OC_InitTypeDef sConfigOC = {0};
530   /* USER CODE BEGIN TIM4_Init 1 */
532   /* USER CODE END TIM4_Init 1 */
533   htim4.Instance = TIM4;
534   htim4.Init.Prescaler = 15;
535   htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
536   htim4.Init.Period = 299;
537   htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
538   htim4.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_ENABLE;
539   if (HAL_TIM_PWM_Init(&htim4) != HAL_OK)
540   {
541     Error_Handler();
542   }
543   sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
544   sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
545   if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig) !=
        HAL_OK)
546   {
547     Error_Handler();
548   }
549   sConfigOC.OCMode = TIM_OCMODE_PWM1;
550   sConfigOC.Pulse = 0;
551   sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
```

```
552   sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
553   if (HAL_TIM_PWM_ConfigChannel(&htim4, &sConfigOC, TIM_CHANNEL_1) != HAL_OK
          )
554   {
555     Error_Handler();
556   }
557   if (HAL_TIM_PWM_ConfigChannel(&htim4, &sConfigOC, TIM_CHANNEL_2) != HAL_OK
          )
558   {
559     Error_Handler();
560   }
561   /* USER CODE BEGIN TIM4_Init 2 */
562   HAL_TIM_PWM_Start(&htim4, TIM_CHANNEL_1);
563   HAL_TIM_PWM_Start(&htim4, TIM_CHANNEL_2);
564   /* USER CODE END TIM4_Init 2 */
565   HAL_TIM_MspPostInit(&htim4);

567 }

569 /**
570   * @brief USART3 Initialization Function
571   * @param None
572   * @retval None
573   */
574 static void MX_USART3_UART_Init(void)
575 {

577   /* USER CODE BEGIN USART3_Init 0 */

579   /* USER CODE END USART3_Init 0 */

581   /* USER CODE BEGIN USART3_Init 1 */
```

```
583   /* USER CODE END USART3_Init 1 */
584   huart3.Instance = USART3;
585   huart3.Init.BaudRate = 115200;
586   huart3.Init.WordLength = UART_WORDLENGTH_8B;
587   huart3.Init.StopBits = UART_STOPBITS_1;
588   huart3.Init.Parity = UART_PARITY_NONE;
589   huart3.Init.Mode = UART_MODE_TX_RX;
590   huart3.Init.HwFlowCtl = UART_HWCONTROL_NONE;
591   huart3.Init.OverSampling = UART_OVERSAMPLING_16;
592   huart3.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
593   huart3.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
594   if (HAL_UART_Init(&huart3) != HAL_OK)
595   {
596     Error_Handler();
597   }
598   /* USER CODE BEGIN USART3_Init 2 */

600   /* USER CODE END USART3_Init 2 */

602 }

604 /**
605   * @brief USB_OTG_FS Initialization Function
606   * @param None
607   * @retval None
608   */
609 static void MX_USB_OTG_FS_PCD_Init(void)
610 {

612   /* USER CODE BEGIN USB_OTG_FS_Init 0 */

614   /* USER CODE END USB_OTG_FS_Init 0 */
```

```
616   /* USER CODE BEGIN USB_OTG_FS_Init 1 */

618   /* USER CODE END USB_OTG_FS_Init 1 */
619   hpcd_USB_OTG_FS.Instance = USB_OTG_FS;
620   hpcd_USB_OTG_FS.Init.dev_endpoints = 6;
621   hpcd_USB_OTG_FS.Init.speed = PCD_SPEED_FULL;
622   hpcd_USB_OTG_FS.Init.dma_enable = DISABLE;
623   hpcd_USB_OTG_FS.Init.phy_itface = PCD_PHY_EMBEDDED;
624   hpcd_USB_OTG_FS.Init.Sof_enable = ENABLE;
625   hpcd_USB_OTG_FS.Init.low_power_enable = DISABLE;
626   hpcd_USB_OTG_FS.Init.lpm_enable = DISABLE;
627   hpcd_USB_OTG_FS.Init.vbus_sensing_enable = ENABLE;
628   hpcd_USB_OTG_FS.Init.use_dedicated_ep1 = DISABLE;
629   if (HAL_PCD_Init(&hpcd_USB_OTG_FS) != HAL_OK)
630   {
631     Error_Handler();
632   }
633   /* USER CODE BEGIN USB_OTG_FS_Init 2 */

635   /* USER CODE END USB_OTG_FS_Init 2 */

637 }

639 /**
640   * @brief GPIO Initialization Function
641   * @param None
642   * @retval None
643   */
644 static void MX_GPIO_Init(void)
645 {
646   GPIO_InitTypeDef GPIO_InitStruct = {0};

648   /* GPIO Ports Clock Enable */
```

```
649   __HAL_RCC_GPIOC_CLK_ENABLE();
650   __HAL_RCC_GPIOF_CLK_ENABLE();
651   __HAL_RCC_GPIOH_CLK_ENABLE();
652   __HAL_RCC_GPIOA_CLK_ENABLE();
653   __HAL_RCC_GPIOB_CLK_ENABLE();
654   __HAL_RCC_GPIOD_CLK_ENABLE();
655   __HAL_RCC_GPIOG_CLK_ENABLE();

657   /*Configure GPIO pin Output Level */
658   HAL_GPIO_WritePin(GPIOB, LD1_Pin|LD3_Pin|LD2_Pin, GPIO_PIN_RESET);

660   /*Configure GPIO pin Output Level */
661   HAL_GPIO_WritePin(USB_PowerSwitchOn_GPIO_Port, USB_PowerSwitchOn_Pin,
          GPIO_PIN_RESET);

663   /*Configure GPIO pin Output Level */
664   HAL_GPIO_WritePin(GPIOC, R_EN2_Pin|L_EN2_Pin|R_EN1_Pin|L_EN1_Pin,
          GPIO_PIN_RESET);

666   /*Configure GPIO pins : LD1_Pin LD3_Pin LD2_Pin */
667   GPIO_InitStruct.Pin = LD1_Pin|LD3_Pin|LD2_Pin;
668   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
669   GPIO_InitStruct.Pull = GPIO_NOPULL;
670   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
671   HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

673   /*Configure GPIO pin : USB_PowerSwitchOn_Pin */
674   GPIO_InitStruct.Pin = USB_PowerSwitchOn_Pin;
675   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
676   GPIO_InitStruct.Pull = GPIO_NOPULL;
677   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
678   HAL_GPIO_Init(USB_PowerSwitchOn_GPIO_Port, &GPIO_InitStruct);
```

```
680   /*Configure GPIO pin : USB_OverCurrent_Pin */
681   GPIO_InitStruct.Pin = USB_OverCurrent_Pin;
682   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
683   GPIO_InitStruct.Pull = GPIO_NOPULL;
684   HAL_GPIO_Init(USB_OverCurrent_GPIO_Port, &GPIO_InitStruct);

686   /*Configure GPIO pins : LIMITR_2_Pin LIMITR_1_Pin */
687   GPIO_InitStruct.Pin = LIMITR_2_Pin|LIMITR_1_Pin;
688   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
689   GPIO_InitStruct.Pull = GPIO_PULLUP;
690   HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

692   /*Configure GPIO pins : R_EN2_Pin L_EN2_Pin R_EN1_Pin L_EN1_Pin */
693   GPIO_InitStruct.Pin = R_EN2_Pin|L_EN2_Pin|R_EN1_Pin|L_EN1_Pin;
694   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
695   GPIO_InitStruct.Pull = GPIO_NOPULL;
696   GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
697   HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

699   /*Configure GPIO pin : LIMITL_1_Pin */
700   GPIO_InitStruct.Pin = LIMITL_1_Pin;
701   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
702   GPIO_InitStruct.Pull = GPIO_PULLUP;
703   HAL_GPIO_Init(LIMITL_1_GPIO_Port, &GPIO_InitStruct);

705   /*Configure GPIO pin : LIMITL_2_Pin */
706   GPIO_InitStruct.Pin = LIMITL_2_Pin;
707   GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
708   GPIO_InitStruct.Pull = GPIO_PULLUP;
709   HAL_GPIO_Init(LIMITL_2_GPIO_Port, &GPIO_InitStruct);

711 }
```

```
713 /* USER CODE BEGIN 4 */
714 void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
715 {
716   // Check which version of the timer triggered this callback and toggle LED
717   if (htim == &htim3)
718   {
719     READMS5803 = 1;
720   }
721 }
722 /* USER CODE END 4 */

724 /**
725   * @brief  This function is executed in case of error occurrence.
726   * @retval None
727   */
728 void Error_Handler(void)
729 {
730   /* USER CODE BEGIN Error_Handler_Debug */
731   /* User can add his own implementation to report the HAL error return
        state */

733   /* USER CODE END Error_Handler_Debug */
734 }

736 #ifdef  USE_FULL_ASSERT
737 /**
738   * @brief  Reports the name of the source file and the source line number
739   *         where the assert_param error has occurred.
740   * @param  file: pointer to the source file name
741   * @param  line: assert_param error line source number
742   * @retval None
743   */
744 void assert_failed(uint8_t *file, uint32_t line)
```

```
745 {
746   /* USER CODE BEGIN 6 */
747   /* User can add his own implementation to report the file name and line
        number,
748      tex: printf("Wrong parameters value: file %s on line %d\r\n", file,
           line) */
749   /* USER CODE END 6 */
750 }
751 #endif /* USE_FULL_ASSERT */

753 /************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE
      ****/
```

E Operator Console Code

```
1  from time import sleep, asctime
2  import wx
3  import gui
4  from threading import Thread, Lock
5  from queue import Queue
6  from udp_server import UDPServerThread
7  from udp_client import UDPClientThread
8  from dummy_data import DummyIMUThread
9  from logger import LoggingThread

11 commands_queue = Queue()
12 commands_queue_lock = Lock()
13 server_messages_queue = Queue()
14 server_messages_queue_lock = Lock()
15 dummy_data_queue = Queue()
16 dummy_data_queue_lock = Lock()
```

```
client_messages_queue = Queue()
client_messages_queue_lock = Lock()
logging_messages_queue = Queue()
logging_messages_queue_lock = Lock()

udp_server_thread = UDPServerThread(1, "UDP Server Thread", (
    server_messages_queue, server_messages_queue_lock),
                                    (commands_queue, commands_queue_lock))
udp_client_thread = UDPClientThread(2, "UDP Client Thread", (
    client_messages_queue, client_messages_queue_lock),
                                    (dummy_data_queue, dummy_data_queue_lock
                                        ))
dummy_imu_thread = DummyIMUThread(3, "Dummy IMU Thread", (dummy_data_queue,
    dummy_data_queue_lock))

f = open("logs/log_%s.txt" % asctime().replace(" ", "_").replace(":", "_"),
    "w")

logging_thread = LoggingThread(4, "Logging Thread", f, (
    logging_messages_queue, logging_messages_queue_lock))

logging_thread.start()
udp_server_thread.start()
sleep(1)
# udp_client_thread.start()
# dummy_imu_thread.start()

app = wx.App()
frame = gui.frameMain(None, (server_messages_queue,
    server_messages_queue_lock), (commands_queue, commands_queue_lock), (
    logging_messages_queue, logging_messages_queue_lock))
frame.timer.Start(1000)
frame.Show()
```

```
app.MainLoop()
udp_server_thread.join()
# udp_client_thread.join()
# dummy_imu_thread.join()
logging_thread.join()
f.close()
```

```
import socket
from threading import Thread
import queue
from time import time, gmtime, strftime

HOST = ''
PORT = 8888



class UDPServerThread(Thread):

    def __init__(self, thread_id, name, message_queue, outgoing_queue):
        Thread.__init__(self)
        self.thread_id = thread_id
        self.name = name
        self.messages, self.messages_lock = message_queue
        self.outgoing, self.outgoing_lock = outgoing_queue

    def message(self, msg):
        str_time = strftime("%H:%M:%S", gmtime(time()))
        self.messages_lock.acquire()
        self.messages.put("%s %s" % (str_time, msg))
        self.messages_lock.release()

    def run(self):
        bound = False
```

```
27          addr = None
28          while not bound:
29              # attempt socket creation
30              try:
31                  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
32                  self.message('[Server] Socket created')
33              except socket.error as msg:
34                  self.message('[Server] Failed to create socket. Error Code :
                        ' + str(msg[0]) + ' Message ' + msg[1])
35              # attempt socket bind
36              try:
37                  s.bind((HOST, PORT))
38              except socket.error as msg:
39                  self.message('[Server] Bind failed. Error Code : ' + str(msg
                        ))

41              self.message('[Server] Socket bind complete')
42              bound = True
43          s.settimeout(1)
44          while True:
45              # only try and send if we have a destination and a message
46              if addr is not None and not self.outgoing.empty():
47                  self.outgoing_lock.acquire()
48                  msg = self.outgoing.get()
49                  self.outgoing_lock.release()
50                  s.sendto(str.encode(msg), addr)
51                  self.message('[Server] Sent message to %s:%s:"%s"' % (addr
                        [0], str(addr[1]), msg.strip()))
52              # check for incoming message
53              try:
54                  recv = s.recvfrom(1024)
55                  data = recv[0].decode()
56                  addr = recv[1]
```

```
                self.message('Received Message from %s:%s:"%s"' %(addr[0],
                    str(addr[1]), data.strip()))
            except socket.timeout as msg:
                pass
        s.close()
```

```
from threading import Thread
import queue

class LoggingThread(Thread):

    def __init__(self, thread_id, name, file, log_queue):
        Thread.__init__(self)
        self.thread_id = thread_id
        self.name = name
        self.log_queue, self.log_lock = log_queue
        self.file = file

    def run(self):

        while True:
            if not self.log_queue.empty():
                self.log_lock.acquire()
                line = self.log_queue.get()
                self.log_lock.release()
                self.file.write("%s\n" % line)
                self.file.flush()
```