Magnetically-Driven Submersible

Rodrigo Blanco Arce, Kyle Ito, Thomas Fabian Kragerud, Nicholas Chard

Final Project Report – Group 29

MECENG 102B: Mechatronics Design

Department of Mechanical Engineering, University of California at Berkeley

Professor Hannah Stuart

December 14th, 2023.

Opportunity:

One of the biggest challenges of designing a mechatronic device that interacts with water is that it requires extensive waterproofing and the implementation of specialized parts to avoid damaging the electronics. The current market for watertight electric motors, bearings, and electrical connectors tends to be very expensive, and of limited availability. This created an opportunity for us to make a device that can interact with marine environments without the need for specialized electronic equipment, allowing enthusiasts to create projects that interact with water without breaking the bank.

High-level strategy:

We have designed a submersible that does not require any specialized watertight electronics or connections, and only implements consumer-grade electronics. This was achieved by creating a water-tight chamber that houses all of the electronics (microcontroller, battery, motor, etc.) and has no physical connections to the outside environment. Inside the water-tight chamber, a DC motor drives a non-contact magnetic shaft coupler that transmits the shaft power to an external mechanical transmission that drives the submersible. The water-tight chamber was machined out of a single piece of Delrin plastic, as a cylinder with only one open end that needed to be made waterproof. The waterproofing was achieved by using an off-the-shelf oring in between a polycarbonate end-plate and the main Delrin hull. We wanted our vehicle to be watertight at 0.5m of depth for at least 30 minutes and achieve a maximum speed of 0.35 m/s. The vehicle experienced no leaks when submerged at 0.5m for 45 minutes and achieved a maximum speed of 0.30 m/s as determined through video footage analysis.

Integrated device:

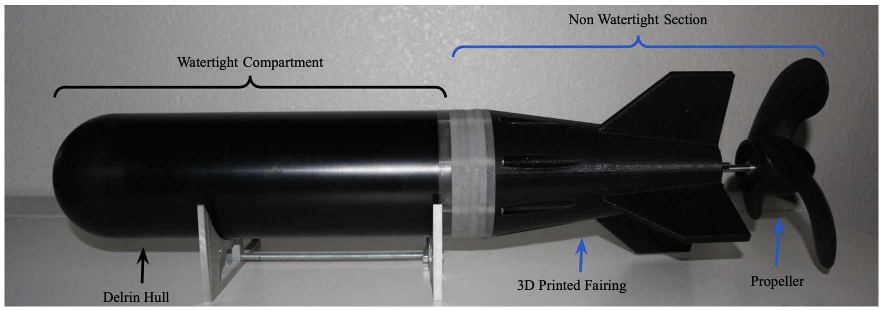


Figure 1: Full assembly.

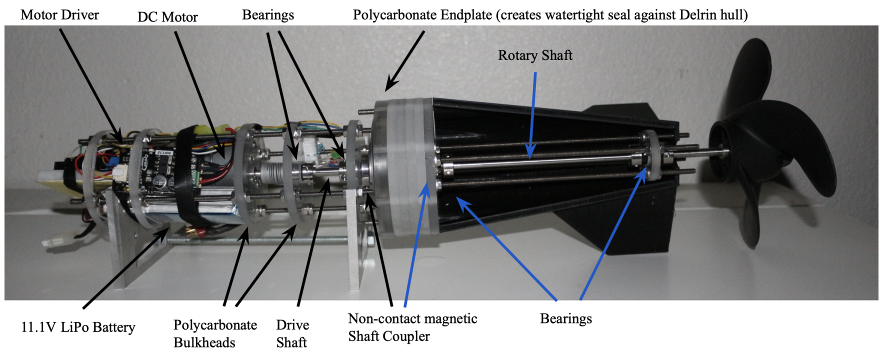

Motor Driver
DC Motor
Bearings
Polycarbonate Endplate (creates watertight seal against Delrin hull)
Rotary Shaft
11.1V LiPo Battery
Polycarbonate Bulkheads
Drive Shaft
Non-contact magnetic Shaft Coupler
Bearings

Figure 2: Assembly without Delrin hull and open fairings for visualization.

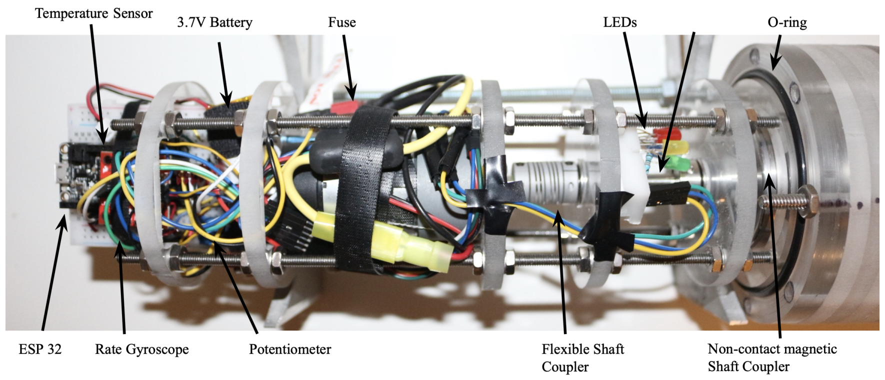


Figure 3: Sub-assembly – inside watertight chamber.

Function-critical decisions:

One of the biggest design challenges of this project was packaging all of the elements inside the main hull and external faring of the submersible while retaining the ability to influence the fore/aft weight bias to adjust the pitch. To address this, we made a modular design based on a series of water-jet cut polycarbonate bulkheads that housed the main elements of both mechanical transmissions and the electronics. The bulkheads sit on longitudinally-spanning threaded rods, and the spacing between them can be modified by sliding them along these rods. All bearings used to support both transmissions, inside and outside of the watertight compartment, were press-fitted into these bulkheads, and the DC motor was directly mounted on one of them. The bearings were preloaded using disk springs and shaft collars.

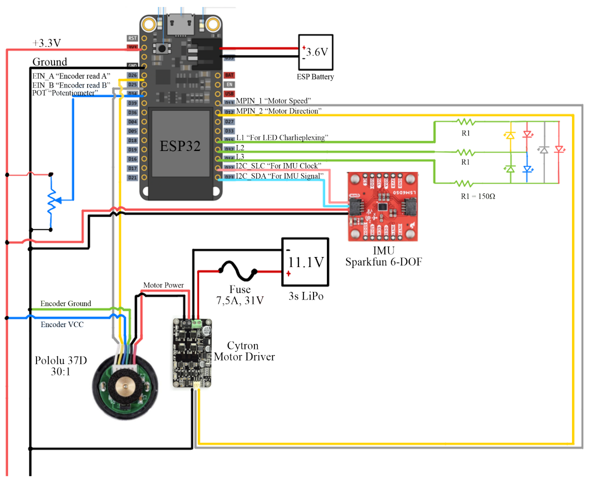
For the selection of the motor and magnetic shaft coupler, we performed the following series of calculations based on our propeller, hull shape, and ultimate desired vehicle speed. Since the propeller has a $Pitch = 0.145$ [m/rev], assuming a $Slippage = 50\%$, the true pitch can be given as $Pitch_{true} = Pitch \times Slippage = 0.0725$ [m/rev]. By setting the desired velocity for our vehicle to be $V_{des} = 0.35$ [m/s], we can solve for the required shaft angular velocity as $\Omega_{Shaft_{req}} = V_{des}/Pitch_{true} \approx 280$ [rpm]. This is the speed at which the propeller would need to rotate for the vehicle to achieve its target speed.

Next, we can find the torque required to spin the propeller through the water to aid in our selection of DC motor and magnetic shaft coupler; ensuring that we can both generate and transmit enough torque to move the submarine. To do so, we approximate the drag force of the vehicle as $F_d = \frac{1}{2}\rho C_d A V^2 = 0.2313$ [N], where A is the cross sectional area of the hull, ρ is water density, and $C_d = 0.5$ is the estimated drag coefficient of the hull. Then we can calculate the power required to move at the desired speed through water as $P = F_d V = 0.0782$ [W]. Lastly, we can calculate the required shaft torque due to hydrodynamic forces on the propeller as $\tau_{req} = \frac{P}{\Omega} = 0.2753$ [kg·mm]. The maximum torque that both the DC motor and the magnetic shaft coupler could achieve needed to be higher than this.

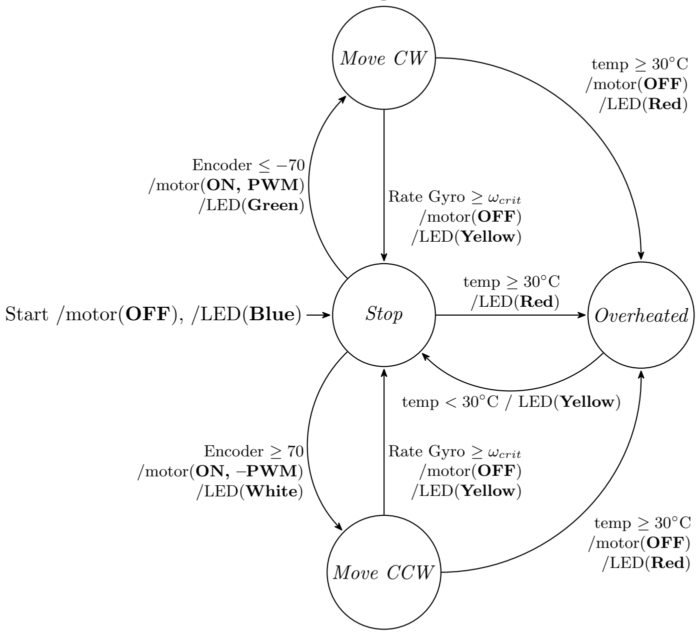
Given these constraints, we've selected the magnetic shaft coupler to have a $\tau_{max_{magn}} = 69.13$ kg·mm, which is well above the minimum required value to ensure that it wouldn't slip under normal operating conditions. In order to maximize the speed attainable by our submersible, we selected the Pololu 37D motor, with a 30:1 ratio gearbox. This motor has a maximum speed of 330 rpm and can produce $\tau_{max_{motor}} = 140$ kg·mm. This satisfies the angular velocity required for the propeller, taking into account a safety margin to increase motor longevity, while having the ability to generate a torque larger than the maximum torque that the magnetic shaft coupler can transmit. The decision of having a motor that can produce a torque that's considerably larger that what the magnetic shaft coupler can transmit was made so that if the propeller gets caught in an obstacle, the magnetic shaft coupler would act as a clutch allowing for slip without stalling the motor.

To adjust the speed of the submarine, we included a potentiometer (analog input) that sets the maximum PWM signal for the DC motor using an open loop controller. To start the motor, the encoder (digital input) was used to detect movement of the shaft. The user can then rotate the propeller in the desired direction, and the motor will start rotating in that same direction. This is possible since the maximum torque of the magnetic shaft coupler exceeds that needed to manually rotate the DC motor. A rate gyroscope (IMU) and a temperature sensor were used to stop the motor by detecting an irregular angular velocity of the submarine as well as potential problems like overheating. These desired behaviors were achieved through a state machine using event driven programming in the Arduino IDE on an ESP32 microcontroller.

Circuit Diagram:



State Diagram:


Move CW
Stop
Overheated
Move CCW
Start /motor(**OFF**), /LED(**Blue**)
Encoder ≤ -70 /motor(**ON, PWM**) /LED(**Green**)
Rate Gyro $\geq \omega_{crit}$ /motor(**OFF**) /LED(**Yellow**)
temp $\geq 30°$C /motor(**OFF**) /LED(**Red**)
temp $\geq 30°$C /LED(**Red**)
temp $< 30°$C / LED(**Yellow**)
Encoder ≥ 70 /motor(**ON, –PWM**) /LED(**White**)
Rate Gyro $\geq \omega_{crit}$ /motor(**OFF**) /LED(**Yellow**)
temp $\geq 30°$C /motor(**OFF**) /LED(**Red**)

Discussion:

It took us numerous design iterations to get to the final product presented in this report. Some of the previous iterations aimed for a completely different functionality. Through discussing with Professor Stuart and the teaching staff, we realized that they were too ambitious and time constraints wouldn't have allowed us to achieve a prototype as refined as the one we have. After seeing our results, we are convinced that making a device that elegantly performs an arguably simple task is much better than trying to make a design with too many functionalities that will not be performed as elegantly. We are overall extremely happy with the outcome of this project. Its success was made possible in no small part by the support of the Machine Shop staff members at both Etcheverry Hall and Hesse Hall. They proved to be an extremely valuable resource for both discussing design ideas as well as manufacturing and obtaining required parts. We would strongly encourage students to seek support from them as early in their design process as possible.

Appendix A: Complete bill of materials

Part ID	Part Name	Quantity	Cost	Link
1	Half-Sized Breadboard	1	$0.00**	https://www.adafruit.com/product/64
2	Adafruit ESP32	1	$0.00**	https://www.adafruit.com/product/3405
3	Low-Strength Steel Threaded Rod	4	$2.88	https://www.mcmaster.com/98790A052/
4	6”x6”x1/2” Clear Polycarbonate	1	$12.20	https://www.mcmaster.com/8574K321/
5	Buna-N O'Ring 3/32”	1	$12.05	https://www.mcmaster.com/9452K173/
6	Non-Contact Magnetic Shaft Coupling	2	$203.34	https://www.mcmaster.com/9199T2/
7	Stainless Steel Ball Bearing	4	$25.68	https://www.mcmaster.com/57155K305/
8	3”x1/4” Stainless Steel Rotary Shaft	1	$5.75	https://www.mcmaster.com/1257K113/
9	9”x1/4” Stainless Steel Rotary Shaft	1	$10.73	https://www.mcmaster.com/1257K118/

10	Precision Flexible Shaft Coupling	1	$66.31	https://www.mcmaster.com/6208K389/
11	12V Gearmotor with encoder and 10:1 gearbox	1	$0.00**	https://www.pololu.com/product/4758
12	12”x24”x1/4” Polycarbonate Sheet	1	$30.25	https://www.mcmaster.com/8574K43/
13	Super Corrosion Resistant Stainless Steel Threaded Rod	4	$16.68	https://www.mcmaster.com/90575A174/
14	3D Printed ABS Fairing	1	$0.00**	https://jacobsinstitute.berkeley.edu/jacobs-self-service-printing/
15	Clamping Shaft Collar	6	$27.66	https://www.mcmaster.com/6436K32/
16	Nylon 7.5” Propeller	1	$16.99	https://www.amazon.com/DEDC-Strength-Propeller-Performance-Outboard/dp/B07VXNDY1Q/ref=sr_1_1_sspa?crid=3UNMKNNBWCKM6&keywords=model+boat+propellers&qid=1697933004&sprefix=model+boat+propellars%2Caps%2C130&sr=8-1-spons&sp_csd=d2lkZ2V0TmFtZT1zcF9hdGY&psc=1

17	Stainless Steel Hex Nut	60	$8.65	https://www.mcmaster.com/90257A009/
18	Stainless Steel Button Head Hex Drive Screw	8	$12.02	https://www.mcmaster.com/98164A663/
19	Stainless Steel Ring Shim	8	$8.63	https://www.mcmaster.com/97022A372/
20	Belleville Disc Springs	4	$3.94	https://www.mcmaster.com/94065K26/
21	Cytron MD13S Motor Driver	1	$0.00**	https://www.amazon.com/Cytron-13A-Motor-Driver-MD13S/dp/B07CW3GRL6
22	11.1V 2200 mAh LiPo Battery	1	$0.00**	https://www.amazon.com/Gens-ace-Battery-2200mAh-Airplane/dp/B00WJN4LG0
23	Stainless Steel Socket Head Screws	6	$11.00	https://www.mcmaster.com/90751A113/
24	Washer	100	$4.11	https://www.mcmaster.com/90107A010/
25	12”x 4” OD Delrin stock	1	$0.00**	https://www.mcmaster.com/8572K36/

26	Assorted Color LEDs	4	$0.00**	https://www.sparkfun.com/products/12062
27	10 kOhm Potentiometer	1	$0.00**	https://www.sparkfun.com/products/9806
28	Eaton 7.5A Fuze	5	$4.48	https://www.waytekwire.com/product/eaton-s-bussmann-series-bk-atm-7-1-2
29	6DOF Accelerometer	1	$0.00**	https://www.sparkfun.com/products/18020gtVayWCFBlsRoCOncQAvD_BwE
30	3.7V LiPo Battery	1	$0.00**	https://www.adafruit.com/product/3898?gad_source=1&gclid=CjwKCAiA1MCrBhAoEiwAC2d64b_xo26qezCgeR6iEDBIXlyK62VXOzY1HPG9y6h0F8yXpY8nS0zwjRoC6ooQAvD_BwE

** The items marked with two asterisks have been provided by the Etcheverry and Hesse Hall's Lab Technicians or Jacobs 3D printing.

Total:	**$483.35**

Appendix B: CAD images of mechanical transmission elements

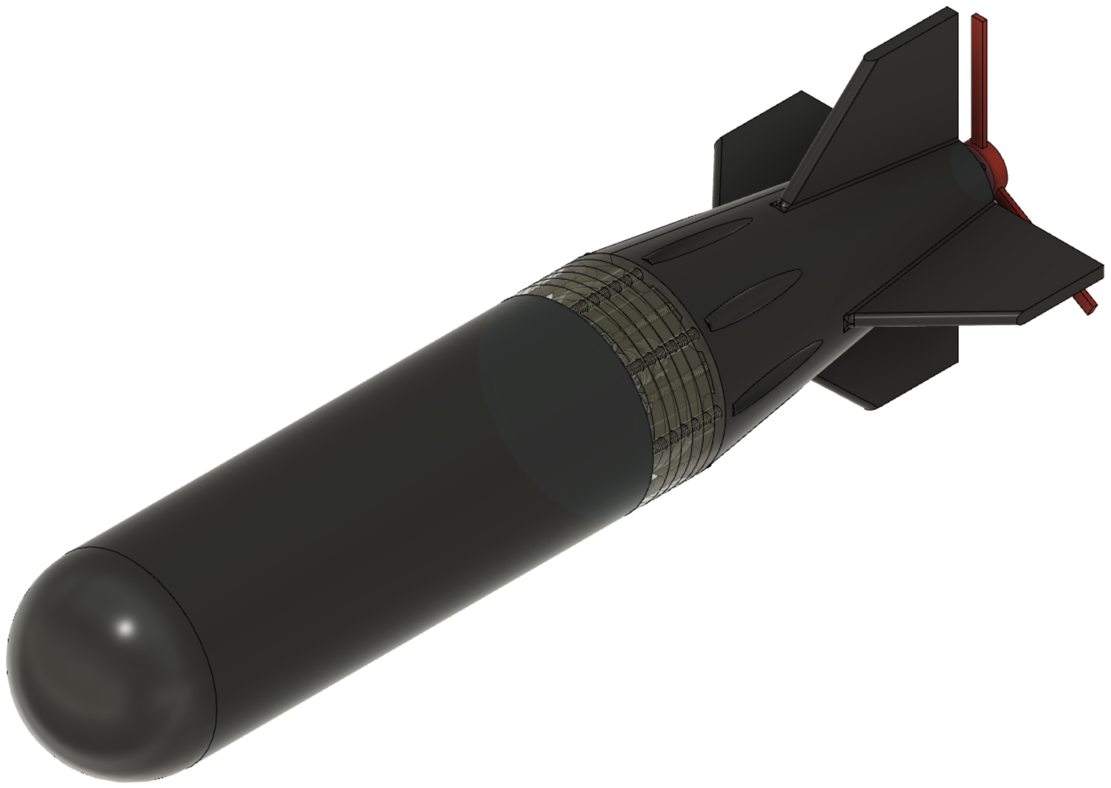


Figure 4: CAD - Isometric View of Full Assembly.

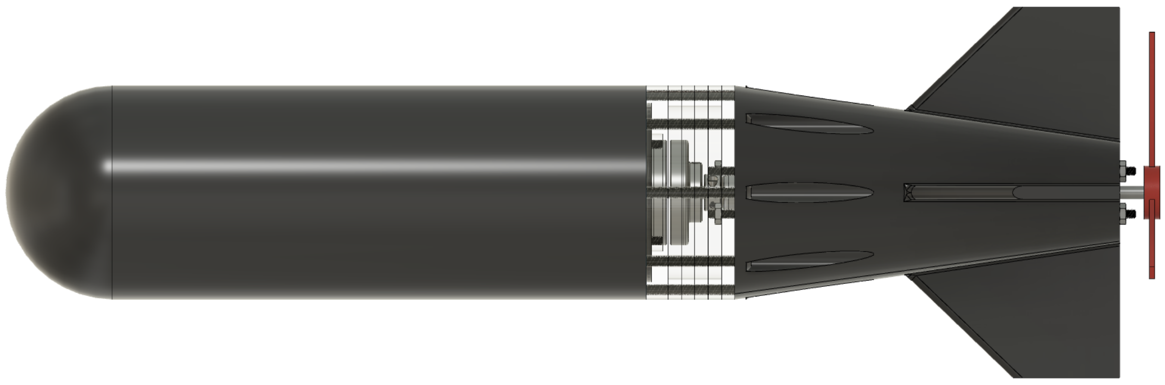


Figure 5: CAD - Side View of Full Assembly.

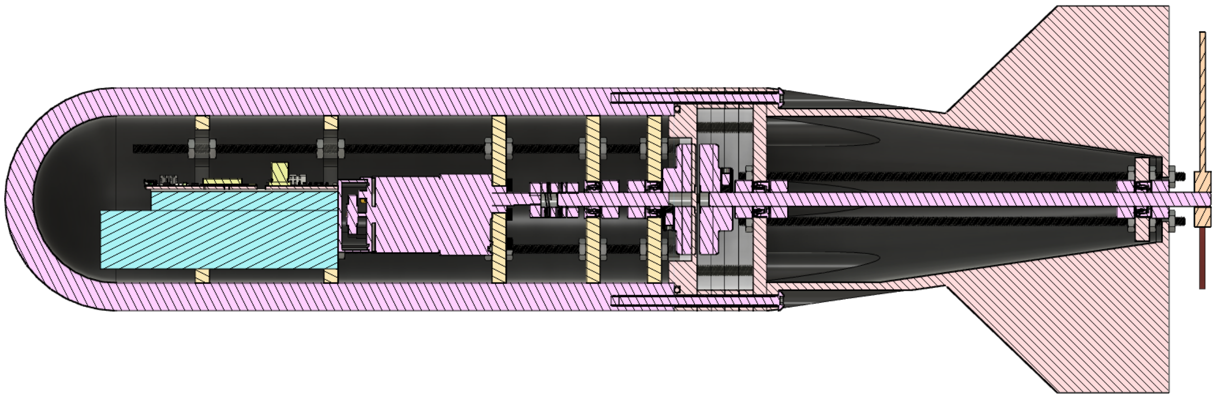


Figure 6: CAD - Cross-sectional View of Full Assembly.

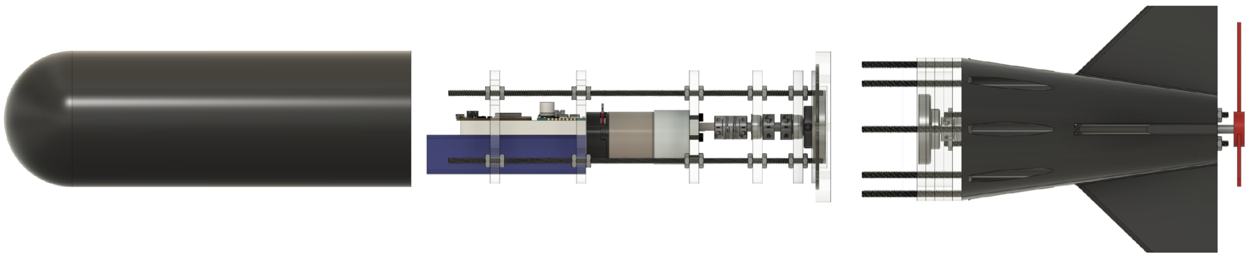


Figure 7: CAD - Exploded View of Full Assembly.

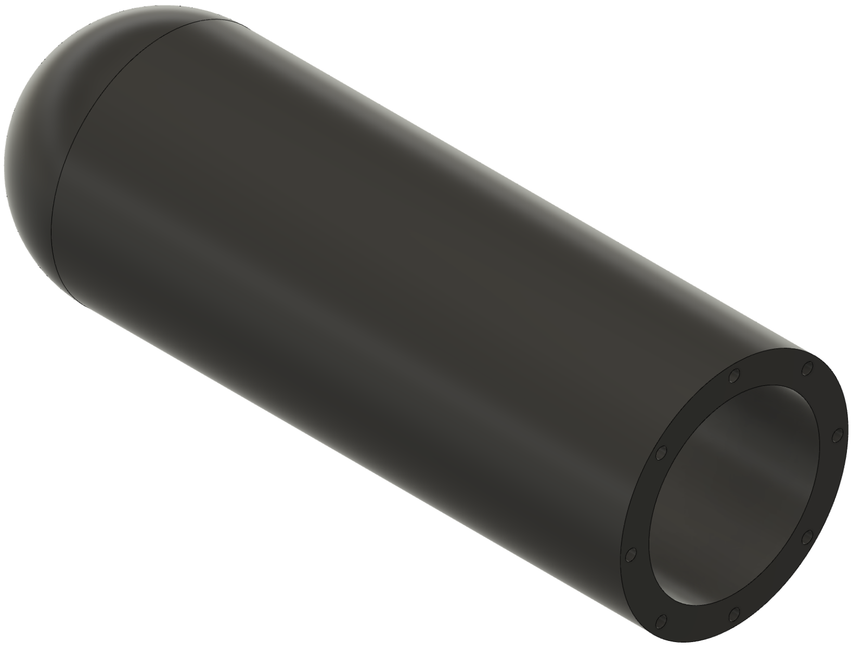


Figure 8: CAD - Isometric View of Delrin Hull (Watertight Compartment's Outer Shell).

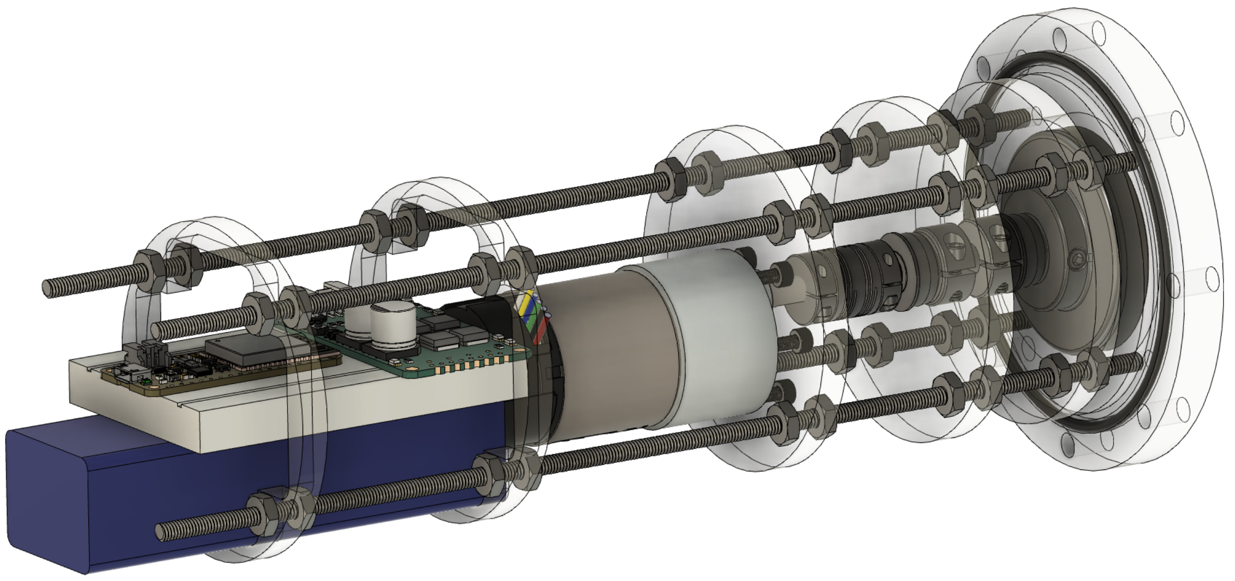


Figure 9: CAD - Isometric View of Internal Sub-Assembly.

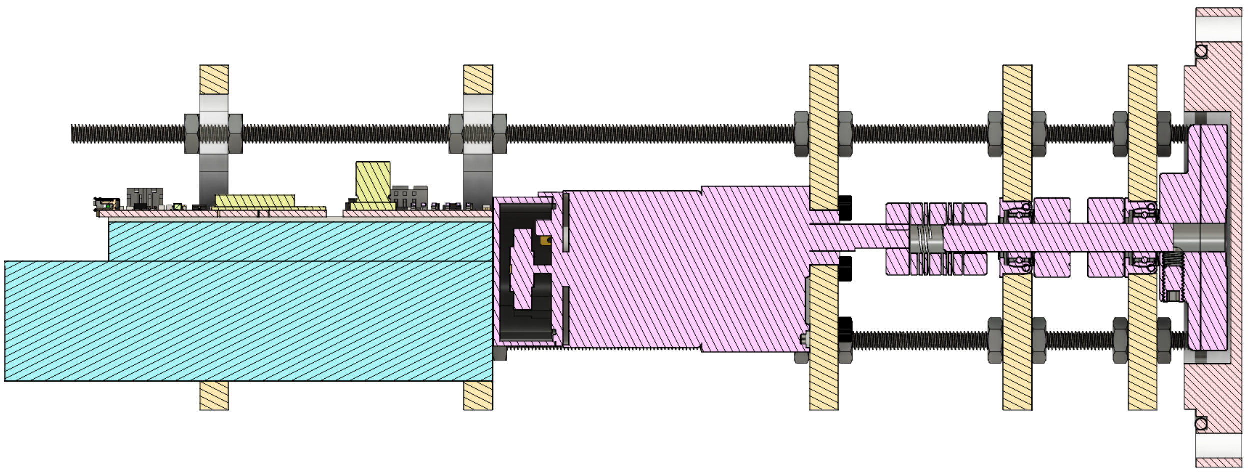


Figure 10: CAD - Cross-sectional View of Internal Sub-Assembly.

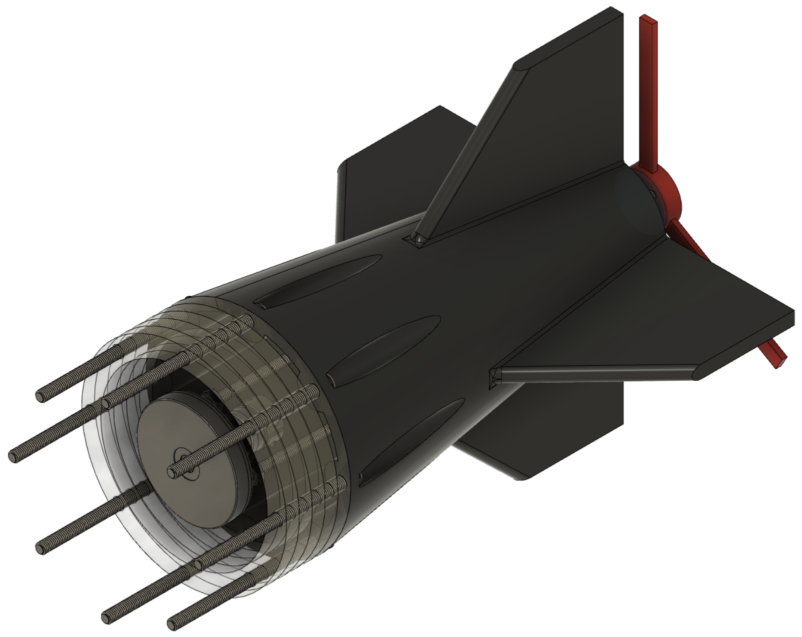


Figure 11: CAD - Isometric View of External Sub-Assembly.

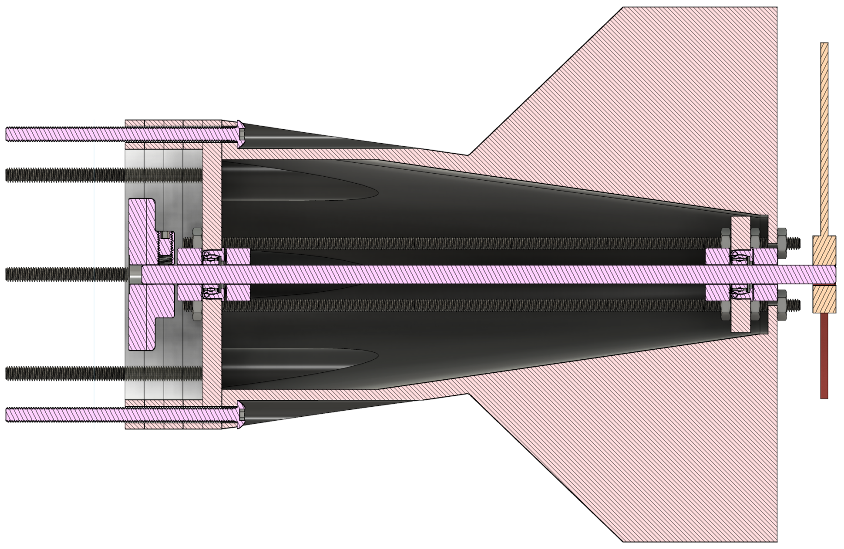


Figure 12: CAD - Cross-sectional View of External Sub-Assembly.

Appendix C: Full listing of code

```
#include "CytronMotorDriver.h"  // Library for the Cytron motor driver card
#include "SparkFunLSM6DSO.h"    // Library for the LSM6DS0 IMU
#include <Wire.h>               // Wire library for I2C communicaition with the IMU
#include <Ticker.h>             // Ticker library for non-blocking countdown timer

// Pin Definitions
#define EIN_A 26 // YELLOW Encoder A pin
#define EIN_B 25 // WHITE Encoder B pin

#define MPIN_1 13 // WHITE Motor Controll pin 1
#define MPIN_2 12 // YELLOW Motor Controll pin 2

#define I2C_SDA 23 // I2C SDA for IMU
#define I2C_SCL 22 // I2C SCL for IMU

// LED Charlieplexing Control Pins
#define L1 15
#define L2 32
#define L3 14

// Potentiometer Pin for Motor Speed Control
#define POT 34 // YELLOW potentiometer

// Constants Definitions
#define GEAR_RATIO 30               // Gear ratio of the motor
#define COUNTS_PER_REVOLUTION 16    // Encoder counts per revolution

 float ENC_TO_ROT = 1.0 / (GEAR_RATIO * COUNTS_PER_REVOLUTION); // Encoder count to rotation conversion
 float CUT_OFF_TEMPERATURE = 30.0; // Temperature threshold in degrees Celsius
 float CUT_OFF_GYRO = 300;         // Gyro threshold for movement detection

 // Sensor and Control Variables
 float temperature = 0.0;
 float GyroX = 0.0, GyroY = 0.0, GyroZ = 0.0;
 int POT_MAX = 4095; // Maximum value for 12-bit ADC
 int potValue = 0;
 int pwmValue = 0;
 int pos = 0;         // Non-critical section position

 // IMU Setup
 LSM6DSO myIMU; // IMU object
 int imu_data;  // Variable to store IMU data

 // Encoder Setup
 volatile int position = 0;
 volatile int velocity = 0;
 volatile int prev_position = 0;
 float n_rotations = 0;
 float rpm = 0;
 bool enableUpdatePosition = true;

 CytronMD motor(PWM_DIR, MPIN_1, MPIN_2); // Motor driver object

// State Machine Enums
enum states { STOP, MOVE_CW, MOVE_CCW, OVERHEATED };
enum states state = STOP;

enum LEDS { YELLOW, RED1, GREEN, BLUE, WHITE, RED2 };

// Interrupt Management Variables
 volatile bool encoderUpdated = false;
 hw_timer_t *timer0 = NULL;
 portMUX_TYPE encoderMux = portMUX_INITIALIZER_UNLOCKED;
 Ticker resetPositionTimer; // ticker object for reseting the position
```

```
// Interrupt Service Routine for Timer
// Every second we update the velocity
void IRAM_ATTR onTime0(){
  portENTER_CRITICAL_ISR(&encoderMux);
  velocity = prev_position - position;
  prev_position = position;
  portEXIT_CRITICAL_ISR(&encoderMux);
}

// Encoder Reading Function
// Works by observing changes to the magnetic field created by a magnet attached to the motor shaft
// If A is high before B, we are moving clockwise
// If B is high before A, we are moving counter-clockwise
// The function is called by an edge-triggered interrupt on A (when A is high), then by checking B
// we can determine the direction
void readEncoder() {
  portENTER_CRITICAL_ISR(&encoderMux); // Entering and exiting two times the ISR allows for less time
  // spent in them, and execute more complex ISR
  encoderUpdated = true;
  int b = digitalRead(EIN_B);
  portEXIT_CRITICAL_ISR(&encoderMux);

  if (enableUpdatePosition) {
    portENTER_CRITICAL_ISR(&encoderMux);
    if (b == HIGH) {
      position++;  // If B is HIGH when A rises, increment position (clockwise rotation)
    } else {
      position--;  // If B is LOW when A rises, decrement position (counterclockwise rotation)
    }
    portEXIT_CRITICAL_ISR(&encoderMux);
  }
}


// Setup Function
void setup(){
  Serial.begin(115200);
  set_LED(BLUE); // Set LED to blue, indicating that we are booting up correctly

  // Initialize Encoder Pins and Interrupt
  pinMode(EIN_A, INPUT_PULLUP);
  pinMode(EIN_B, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(EIN_A), readEncoder, RISING);

  // Timer Setup for Velocity Calculation
  // Set the 0th HW timer to count upwards
  timer0 = timerBegin(0, 80, true);                // Divide by prescaler 80 to get 1MHz tick frequency
  timerAttachInterrupt(timer0, &onTime0, true);    // Attach onTime0 function to timer
  timerAlarmWrite(timer0, 1000000, true);          // Alarm triggers once every second, resets after being
  // triggered
  timerAlarmEnable(timer0);                        // Enable alarm

  // Setup for IMU
  Wire.begin(I2C_SDA, I2C_SCL);
  if (myIMU.begin()){
    Serial.println("IMU Ready");
  } else {
    Serial.println("Could not connect to IMU.");
  }
  if (myIMU.initialize(SOFT_INT_SETTINGS)) {
    Serial.println("Loaded Settings.");
  }
}

// *********************** Main Loop ***********************
void loop() {
  delay(10); // Preventing rapid state changes
```

```
if (encoderUpdated == true) {
  updateSpeedAndPos();
}

switch (state) {
  case STOP:
    if (CheckForCriticalTemperature() == true) {                          // Event Checker
      motor_off();                                                        // Service Function
      Serial.println("Maximum temperature exceeded. Switching to state = OVERHEATED");
      // Print for debugging
      set_LED(RED1);                                                    // Service Function
      state = OVERHEATED;
    }
    if (CheckForEncoderRotationCW() == true) {                            // Event Checker
      motor_on_CW();                                                      // Service Function
      Serial.println("CW Manual start detected. Switching to state = MOVE_CW");
      // Print for debugging
      set_LED(GREEN);                                                   // Service Function
      state = MOVE_CW;
    }
    if (CheckForEncoderRotationCCW() == true) {                           // Event Checker
      motor_on_CCW();                                                     // Service Function
      Serial.println("CCW Manual start detected. Switching to state = MOVE_CCW");
      // Print for debugging
      set_LED(WHITE);                                                   // Service Function
      state = MOVE_CCW;
    }
    break;

  case MOVE_CW:
    if (CheckForCriticalTemperature() == true) {                          // Event Checker
      motor_off();                                                        // Service Function
      Serial.println("Maximum temperature exceeded. Switching to state = OVERHEATED");
      //Print for debugging
      set_LED(RED1);                                                    // Service Function
      state = OVERHEATED;
    }
    if (CheckForCriticalGyro() == true) {                                 // Event Checker
      motor_off();                                                        // Service Function
      Serial.println("Maximum gyro exceeded. Switching to state = STOP"); // Print for debugging
      set_LED(YELLOW);                                                  // Service Function
      state = STOP;
    }
    break;

  case MOVE_CCW:
    if (CheckForCriticalTemperature() == true) {                              // Event Checker
      motor_off();                                                            // Service Function
      Serial.println("Maximum temperature exceeded. Switching to state = OVERHEATED");
      //Print for debugging
      set_LED(RED1);                                                    // Service Function
      state = OVERHEATED;
    }
    if (CheckForCriticalGyro() == true) {                                 // Event Checker
      motor_off();                                                        // Service Function
      Serial.println("Maximum gyro exceeded. Switching to state = STOP"); // Print for debugging
      set_LED(YELLOW);                                                  // Service Function
      state = STOP;
    }
    break;

  case OVERHEATED:
    if (CheckForCriticalTemperature() == false) {                             // Event Checker
      Serial.println("Temperature back to normal. Switching to state = STOP"); //Print for debugging
      set_LED(YELLOW);                                                  // Service Function
      reset_position();
      // (since we didn't turn motor off here as it is already off, we need to explicitly reset the
      // position of the encoder)
```

```
        state = STOP;
      }
      break;
    }
  }

// ************************ Event Checkers (Functions) ************************
bool CheckForEncoderRotationCW() { // Check for clockwise rotation
  return position < -70;
}

bool CheckForEncoderRotationCCW() { // Check for clockwise rotation
  return position > 70;
}

bool CheckForCriticalTemperature() {
  temperature = myIMU.readTempC();
  return temperature >= CUT_OFF_TEMPERATURE;
}

bool CheckForCriticalGyro() {
  GyroX = myIMU.readFloatGyroX();
  GyroY = myIMU.readFloatGyroY();
  GyroZ = myIMU.readFloatGyroZ();
  return (abs(GyroX) >= CUT_OFF_GYRO) || (abs(GyroY) >= CUT_OFF_GYRO) || (abs(GyroZ) >= CUT_OFF_GYRO);
}

// ************************ Service Functions ************************
void motor_on_CW() {
  potValue = analogRead(POT);
  pwmValue = map(potValue, 0, POT_MAX, 50, 128);
  motor.setSpeed(pwmValue);
  Serial.print("Motor ON (CW). Desired Motor PWM: ");
  Serial.println(pwmValue);
}

void motor_on_CCW() {
  potValue = analogRead(POT);
  pwmValue = map(potValue, 0, POT_MAX, 50, 128);
  motor.setSpeed(-pwmValue);
  Serial.print("Motor ON (CCW). Desired Motor PWM: ");
  Serial.println(-pwmValue);
}

void motor_off() {
  motor.setSpeed(0);
  position = 0;
  enableUpdatePosition = false;
  resetPositionTimer.once(1, reset_position); // calls the reset position function after 1 sec
  Serial.println("Motor OFF");
}

void reset_position() {
  position = 0;
  prev_position = 0;
  velocity = 0;
  rpm = 0;
  enableUpdatePosition = true;
  //resetPositionTimer.detach();
}


// Update rpm, pos and rotation count
void updateSpeedAndPos() {
  int localPosition;
  int localVelocity;

  portENTER_CRITICAL(&encoderMux);
  encoderUpdated = false;
  localPosition = position;
  localVelocity = velocity;
  portEXIT_CRITICAL(&encoderMux);

  pos = localPosition;
  n_rotations = pos * ENC_TO_ROT;
  rpm = localVelocity * ENC_TO_ROT * 60;
}


// *********************** Functions for LEDs (control charlieplexing) ***********************
// Implementing charlieplexing, a technique uest to control multiple LEDs with fewer I/O pins
// exploiting the tristate capabilities of the microcontroller, this means that pins can be both
// on off and "disconnected"

// Set LED pin to high (sourcing current)
void set_H(int pin) {
  pinMode(pin, OUTPUT);
  digitalWrite(pin, HIGH);
}

// Set LED pin to low (sinking current)
void set_L(int pin) {
  pinMode(pin, OUTPUT);
  digitalWrite(pin, LOW);
}

// Set LED pin to Z (high-impedans)
void set_Z(int pin) {
  pinMode(pin, INPUT);     // Put pin in high-impedans state, neither souces nor sinks current
  digitalWrite(pin, LOW); // Enable internal pull-down resistor, ensure pin pulled to low voltage
}

void set_LED(LEDS color) {
  switch (color)
  {
    case YELLOW:
      set_L(L1); set_H(L2); set_Z(L3);
      break;

    case RED1:
      set_H(L1); set_L(L2); set_Z(L3);
      break;

    case GREEN:
      set_Z(L1); set_L(L2); set_H(L3);
      break;

    case BLUE:
      set_Z(L1); set_H(L2); set_L(L3);
      break;

    case WHITE:
      set_L(L1); set_Z(L2); set_H(L3);
      break;
    case RED2:
      set_H(L1); set_Z(L2); set_L(L3);
      break;
  }
}
```