Automatic Flight Control and Landing of a Model Glider

Shea Charkowsky, Damond Li, Arielle Sampson, Daniel Xu

ME 507: Mechanical Control System Design

Professor John Ridgely

Department of Mechanical Engineering, Cal Poly, San Luis Obispo

Fall 2022

Full Documentation:

https://damondli.github.io/airheads/index.html

Source Code Repository:

https://github.com/damondli/airheads

I. Index

a. List of Figures

II. Introduction

One of the main causes of accidents in small general aviation airplanes is pilot error on takeoff and landing. While taking off and landing, it is easy for the pilot to stall the airplane because of slow speeds and high angles of attack. With stall situations, it is also common for general aviation airplanes to experience spin, which is a yaw-aggravated stall that results in rotation about the spin axis. Without significant practice, stalls and spins can lead to deadly accidents.

For this project, we focused on controlling the landing of a model glider. The model glider represents a small, general aviation airplane that has lost engine power and must perform a power-off landing. Power-off landings are particularly dangerous because the pilot has no option to abort the landing or perform a touch-and-go landing, which is a form of aborted landing after a touchdown. So, the control system described in this report is designed to automatically control the unpowered landing of a model glider. This project represents a proof-of-concept for an automated landing system that will theoretically take over from a pilot if they encounter a dangerous situation during landing. The control system is designed to maintain steady flight through descent and then commands a landing configuration with a slight positive pitch to safely land the aircraft.

This control system is designed for use especially by inexperienced pilots who may encounter dangerous stall or spin situations without the expertise to quickly correct the problem. While it would be most useful for new pilots, it could also be implemented into all general aviation airplanes as an added safety measure. Especially on takeoff and landing, it is critical that the airplane does not stall or spin because there is not enough altitude to recover from the mistake. This control system could prevent and correct pilot errors leading to these dangerous situations. The following report discusses the specifications, development, and results of the hardware and software for this stabilization system.

III. Specifications

This flight control system is designed to prevent stall and spin situations in small airplanes during unpowered landings. The system will be integrated into a model glider to demonstrate its performance. The system has several testable specifications, each of which has a pass or fail criterion. The flight control system must meet the requirements listed in Table 1 below.

Table 1: Specifications for Flight Control System

ID	Requirement	Verification Method	Rationale
1.	The system shall measure the aircraft's distance to the ground with 10 cm accuracy.	Demonstration	The system must recognize when the aircraft is close to the ground so that the landing configuration can be established.
2.	The system shall measure the aircraft's roll, pitch, and yaw angles with 1-degree accuracy.	Demonstration	The system must identify the attitude of the aircraft to prevent stalls and spins during the flight.
3.	The system shall actuate a rudder control surface between -35 and 35 degrees.	Demonstration	The system must be able to respond to dangerous yaw angles to prevent spins during flight.
4.	The system shall actuate an elevator control surface between -30 and 17 degrees.	Demonstration	The system must be able to respond to dangerous pitch angles to prevent stalls during flight.
5.	The system shall communicate the aircraft's vertical position and attitude via an HTML webpage.	Demonstration	The system must communicate with a remote user so the user can verify the status of the aircraft.
6.	The system shall fit inside the fuselage of a Flite Test Simple Soarer glider (2" x 2" x 12") [1].	Inspection	The system will be demonstrated using a model glider body for proof-of-concept.
7.	The system shall weigh less than 100 grams.	Inspection	The aircraft used to test the system will be a model glider that can have a payload of up to 100 grams.

Each of these specifications for the automatic flight control system will be validated during the project according to the listed validation method.

IV. Design Development

Initially, we considered designing a flight control system for automated takeoff and landing but realized that there may be issues with flying powered remotely controlled airplanes on Cal Poly's campus. So, we "landed" on a glider design that would be capable of demonstrating an automated landing system. Using a glider without a propulsion system, the flight control system demonstrates the ability to prevent stall or spin situations on landing. During takeoff and landing, the airplane is relatively close to the ground, so there is little time that can be used to recover from a stall or spin. Especially in an engine-out landing scenario, the airplane flies at very low speeds, so it is easy for a pilot to inadvertently stall the aircraft. It is important for the airplane not to pitch up too far on landing, which could stall the airplane. Additionally, if the airplane stalls with a yaw component, a spin may occur. Spins are especially difficult for inexperienced pilots to recover from in general aviation airplanes. So, the flight control system discussed in this report will prevent stall and spin situations during unpowered landings.

The system will be integrated into a model glider to demonstrate the control of the elevator and rudder control surfaces during unpowered flight. The flight control system will prevent the airplane from pitching up and yawing during landing. The system will use an IMU in the fuselage of the airplane to detect orientation and acceleration in the pitch and yaw axes. Using the information from the IMU, the system will automatically move the rudder and elevator of the model glider to prevent high angles of attack with yaw components. The system will also include an ultrasonic sensor that will be used to detect when the glider is approaching the ground. The ultrasonic sensor will detect when the glider has landed, and the system will automatically power off to prevent any damage or additional movement of the control surfaces while on the ground. This is a proof-of-concept system that could be further developed for use in general aviation airplanes.

a. Hardware Design

Hardware Parts

The objectives of the project which involved hardware included (a) detection of plane orientation, (b) movement of rudder and elevator, and (c) launch and landing detection. Orientation was measured with a small IMU on a breakout board purchased from Adafruit. Control surface movement was performed with a DC motor in a servo body mounted to the fuselage. A potentiometer was coupled to the DC motor to detect the orientation of the motor shaft. Launch and landing was determined by the distance below the plane measured by an ultrasonic sensor. These parts were integrated with an ESP32 Feather attached to a custom PCB and powered by a 9V source. A list of major hardware components used is provided in Table 2. Components such as voltage regulators, resistors, and capacitors were also used in the design of this system but are not included in Table 2 for clarity.

Table 2: Major Hardware Components Used

Part	Quantity	Rationale
ESP32 Feather Board	1	Lighter and smaller than normal ESP32
HC-SR04 Ultrasonic Sensor	1	Detect distance from ground
DC Motor in Micro Servo Body	2	Control rudder and elevator
Potentiometer	2	Attach to DC motor for control surface movement
3.7V 450mAh Li-Polymer Battery	1	Small, light battery with enough voltage to power all system components
LSM6DSOX 6-DoF IMU	1	Accelerometer and gyro to tell direction and speed
DRV8871 Motor Driver Chip	2	Motor driver chip to control DC motors

Board Design

The PCB was designed to be as small as feasible to fit into the glider fuselage, approximately 3 inches long and 1 inch wide, with screw terminals extending upward and downward about half an inch each. The circuitry was laid out on an extended shield fit for the pins of the ESP32 Feather.

Our initial schematic and board layout appear in Figure 1 and Figure 2. The board can be divided into five sections:

1. *Power*. This section includes a 2-pin screw terminal block for a 9V power source, a Schottky diode for protection, two capacitors for filtering, and a voltage regulator to shift down to 5V power for some components.
2. *Motors*. This section contains two copies of the same circuit, each of which contains a 2-pin screw terminal, a DRV8871 motor driver chip, a small capacitor for filtering, and a current-limiting resistor. In the diagram the capacitor and resistor are 0.1 μF and 32kΩ respectively; due to availability of parts, these were replaced with 1 μF and 33kΩ components. This circuit receives 9V power.
3. *Potentiometers*. This section contains through-holes for header pins and traces to 3.3V power from the Feather and to microcontroller input pin locations.
4. *IMU*. This section contains a 4-pin screw terminal and two pull-up resistors for I2C data. It is connected to 3.3V power. The 4.7kΩ resistors of the design were replaced with 3.3kΩ resistors due to availability.
5. *Ultrasonic sensor*. This section includes a 4-pin screw terminal and two level-shifting circuits, each containing two resistors and an N-channel MOSFET. Like for the IMU, 4.7kΩ resistors were replaced with 3.3kΩ resistors.

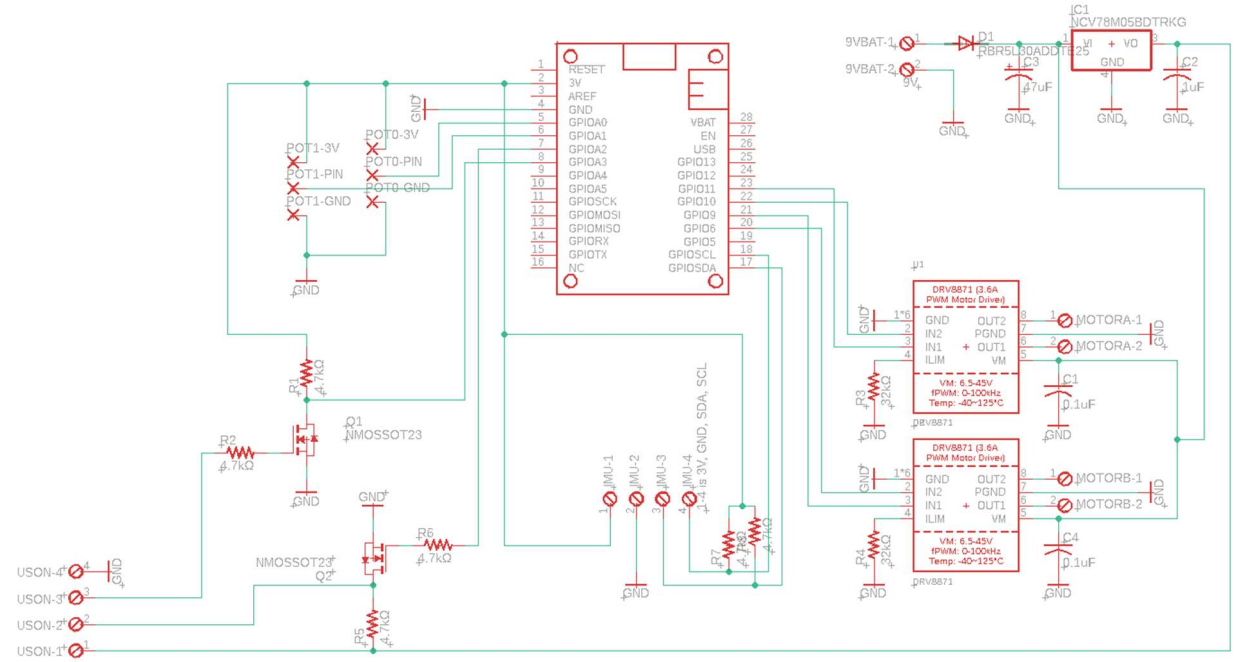


Figure 1: PCB Schematic for Flight Controller

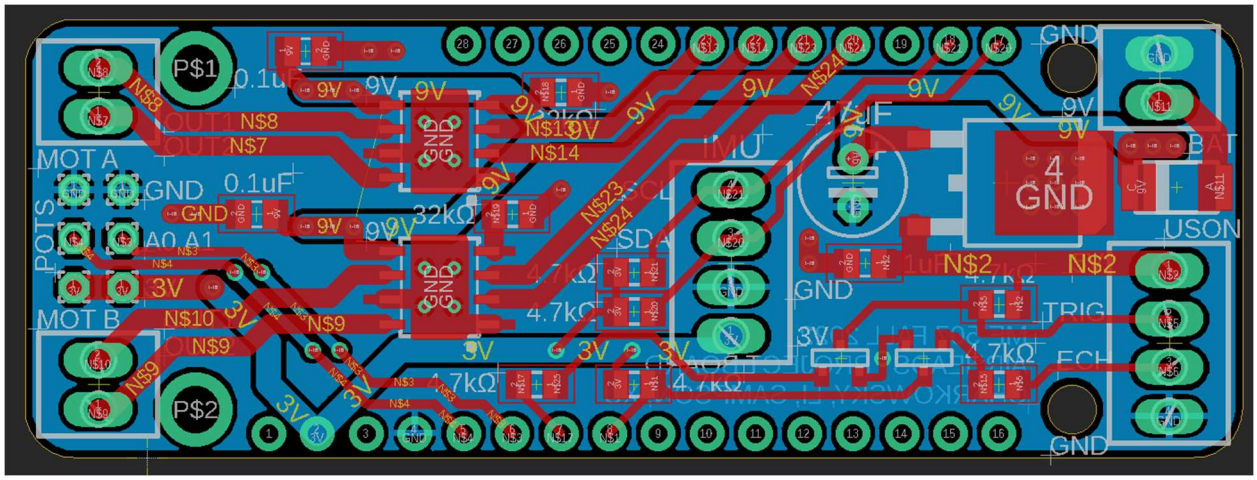


Figure 2: PCB Layout for Flight Controller

Motor and Potentiometer Bracket

Due to the nature of this project, it was determined that using servo motors would be too simple of a solution. To make it more challenging, we decided to use DC motors and implement some form of feedback for positioning the motors. Since the glider was originally designed by Josh

Bixler to use small body servo motors, we went with micro servo body DC motors [1]. To implement some form of feedback, we decided to couple the motor shaft to a potentiometer. The idea for this was inspired by haptic gloves for virtual reality developed by Lucas De Bonet [2].

A mounting bracket was designed to fit a micro servo body and position a potentiometer directly above the motor shaft. The initial mounting bracket concept is shown in Figure 3 below.

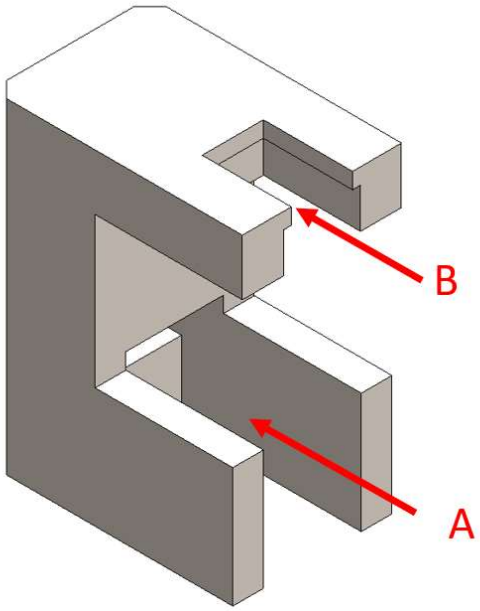


Figure 3: Micro Servo and Potentiometer Mounting Bracket

The region marked as A is the pocket which the micro servo body fits into. B marks the pocket where the potentiometer will fit in. There is a little lip at B which constrains the potentiometer vertically without getting in the way of the terminals. The potentiometer base is also rectangular, so the parallel pocket will prevent it from rotating. With the motor body and the potentiometer in position, an adapter is needed to couple the two together. A cross shaped micro servo horn was chosen for the motor shaft because its geometry would make coupling the two much simpler. Conveniently, the potentiometer has a slot on the knob which we could use to our advantage. An

adapter was designed with the slot and servo cross horn in mind. A three-dimensional drawing of the DC motor-to-potentiometer coupling knob is shown in Figure 4 below.

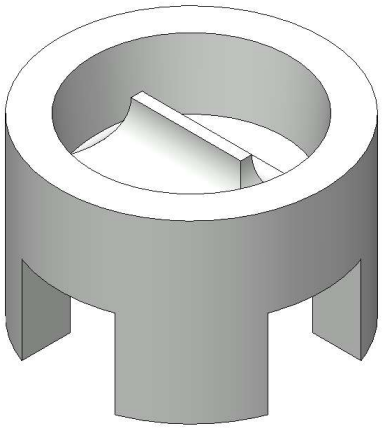


Figure 4: DC Motor-Potentiometer Coupling Adapter

The four legs would fit between the servo cross horn and the top has a slot that fits into the potentiometer. The slot is generously filleted to provide structure. The parts were designed with 3D printing in mind as the geometry is rather complex and several iterations are likely needed to ensure the parts fit nicely together. The complete assembly is shown in Figure 5 below.

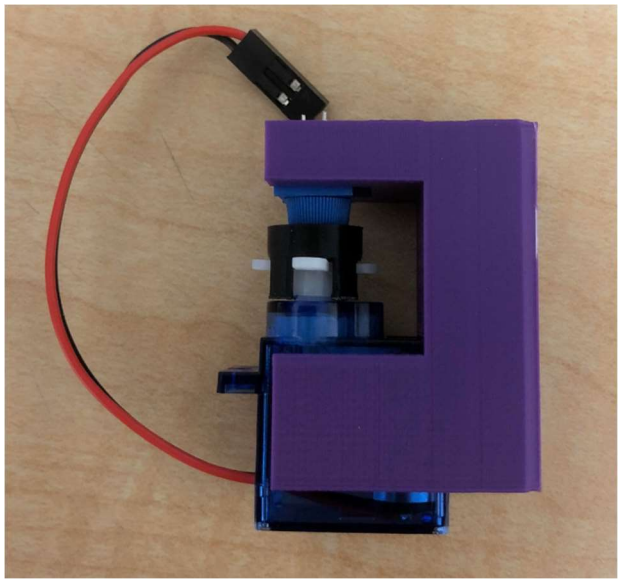


Figure 5: Complete DC Motor and Potentiometer Mounting Assembly

One issue we noticed with this design is that assembly is easy, but disassembly was difficult. The parts are so well constrained that both the motor and potentiometer need to be pulled out at the same time or you risk breaking the thin and fragile tab on the adapter. The simplest solution would be to split the part in half which is what we did. Our final design for the DC motor-to-potentiometer mounting assembly is shown in Figure 6 below.

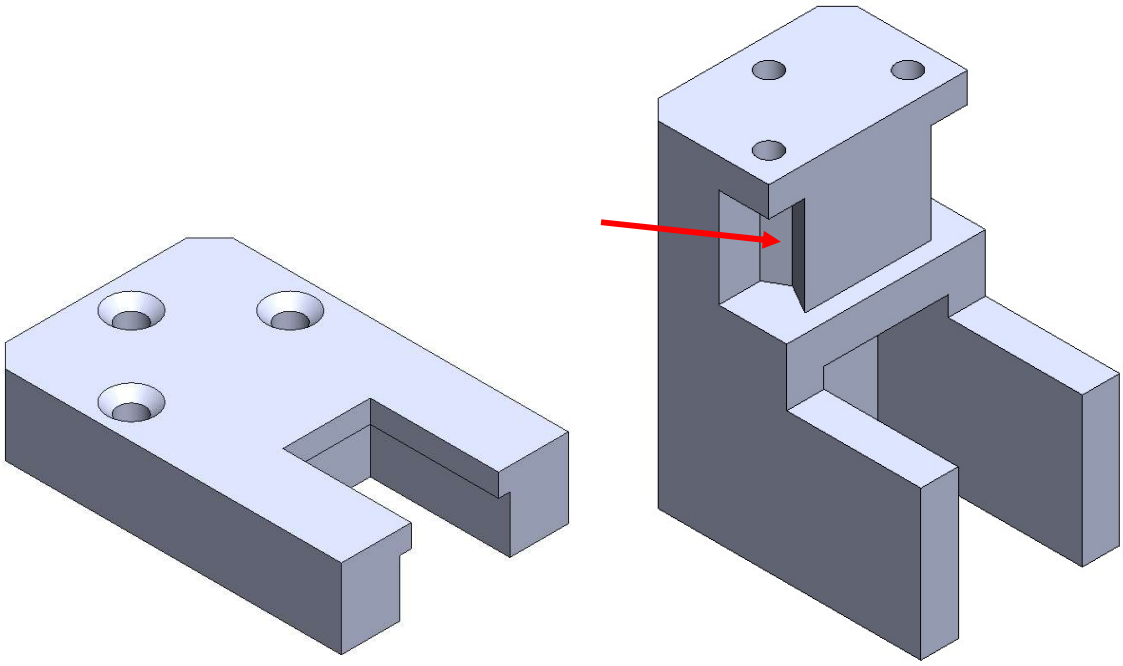


Figure 6: Redesigned DC Motor and Potentiometer Mounting Bracket

The cap and the base are held together with M2 countersunk screws. The pocket indicated by the arrow fits an M2 nut and allows us to tighten the screws without fitting a socket or wrench which would be extremely difficult given the size and geometry of the part. There is also a similar pocket for the screw in the back (not shown). For the complete set of engineering drawings, please refer to Appendix B.

b. Software Design

The software implemented for this flight control system consists of four drivers (motor, inertial measurement unit (IMU), potentiometer, and ultrasonic sensor), a PID controller class, a module for a webpage through the ESP32's Wi-Fi, and a main script to run everything. Other source files such as the baseshare, taskshare, and printstream modules provided by Dr. John Ridgely were also utilized during the development of this project [3][4].

Motor Driver

The motor driver was modified from a script that changes the brightness of an LED which is originally written by Rui and Sara Santos [5]. The modified script was then converted into a class which allows the user to specify the pins and channels they want to use. A snippet of code from the DRV8871 motor driver class constructor is shown in Figure 7 below.

```
/** @brief   Constructor for the DRV8871 motor driver class
 *  @param   pin_A The GPIO pin from the ESP32 (non-zero PWM for a positive
 *           duty cycle)
 *  @param   pin_B The GPIO pin from the ESP32 (non-zero PWM for a negative
 *           duty cycle)
 *  @param   channel_A The timing channel for pin_A
 *  @param   channel_B The timing channel for pin_B
 */
DRV8871::DRV8871(uint8_t pin_A, uint8_t pin_B, uint8_t channel_A, uint8_t
channel_B)
{
    // Establish the output pins
    PIN_A = pin_A;
    PIN_B = pin_B;

    // Establish the channels
    CHANNEL_A = channel_A;
    CHANNEL_B = channel_B;


    // Setup pins with the appropriate resolution and frequency
    ledcSetup(CHANNEL_A, frequency, resolution);
    ledcSetup(CHANNEL_B, frequency, resolution);

    // Attach the pins to the channel
    ledcAttachPin(PIN_A, CHANNEL_A);
    ledcAttachPin(PIN_B, CHANNEL_B);
}
```

Figure 7: DRV8871 Motor Driver Class Constructor

To create a DRV8871 motor object, two pins and two channels need to be specified. The pins correspond to the GPIO pins, and there are a total of 16 difference channels on an ESP32. Because we need each pin output its own independent waveform, we made sure to enter a unique channel for each pin even across multiple instances of the motor class. The frequency refers to the frequency of the pulse width modulated wave. We used a frequency of 20,000 which is just outside the limits of human hearing so there is no annoying audible noise when operating our motors. The resolution refers to the number of bits that will be used to describe the duty cycle. We set our

resolution to 8, meaning that to get a 100% duty cycle we need to set the pin channel to a value of 255. The method that is used to set various duty cycles is shown in Figure 8 below.

```
/** @brief   Outputs the desired PWM signal to the appropriate output pin given
 *           a duty cycle
 *  @param   duty_cycle The duty cycle to run the motors
 */
void DRV8871::set_duty(int16_t duty_cycle)
{
    // Check max and min boundaries for duty cycle inputs
    if (duty_cycle > 100)
    {
        duty = 100;
    }
    else if (duty_cycle < -100)
    {
        duty = -100;
    }
    else
    {
        duty = duty_cycle;
    }

    // Scale the duty cycle according to the resolution of the channel
    duty = duty * max_duty / 100;

    // Check to see which channel to use given the signage of the duty cycle
    if (duty > 0)                       // Use Channel A
    {
        ledcWrite(CHANNEL_A, duty);
        ledcWrite(CHANNEL_B, 0);
    }
    else if (duty < 0)                  // Use Channel B
    {
        ledcWrite(CHANNEL_B, (-1 * duty));
        ledcWrite(CHANNEL_A, 0);
    }
    else                                // Both channels set to zero duty cycle
    {
        ledcWrite(CHANNEL_A, 0);
        ledcWrite(CHANNEL_B, 0);
    }
}
```

Figure 8: Method to Set Various Duty Cycles

The set_duty method first checks if the input argument exceeds the upper or lower bound of 100 or -100, respectively. The input duty cycle is then scaled to the maximum value which is determined by the resolution of the channel (255 for an 8-bit resolution channel). Depending on the sign of the input duty cycle, the channels will be set accordingly. CHANNEL_A will be non-zero and CHANNEL_B will be zero for positive duty cycles and vice versa for negative duty cycles. Note that the ledcWrite function only accepts positive values, so the duty is multiplied by negative one before calling the ledcWrite function for negative duty cycles.

IMU Driver

The IMU driver is used to find the yaw, pitch, and roll and is composed of two different chips. The LSM6DSOX chip is responsible for interfacing with the accelerometer and the gyroscope. The LIS3MDL chip is responsible for interfacing with the magnetometer. Both chips use I2C to communicate with the MCU, of which, we integrated the Adafruit public libraries to do so. The most important feature of our IMU driver class is to combine magnetometer, accelerometer, and gyroscope measurements to get the measurements we want, which we did so through the get_angle method. This is done by first reading the raw data for the different sensors. Using accelerometer measurements, the angles ϕ and ψ are calculated using equations 1 and 2 below.

$$\boldsymbol{\phi = tan^{-1}\left(\frac{Accel_X}{Accel_Y^2+Accel_Z^2}\right)} \qquad \textbf{Equation 1}$$

$$\boldsymbol{\psi = tan^{-1}\left(\frac{Accel_Y}{Accel_X^2+Accel_Z^2}\right)} \qquad \textbf{Equation 2}$$

The code implementation of this equation is shown in Figure 9. It is uneccessary to calculate θ using this method because the accelerometer uses gravitational forces. Since θ uses the downward force, changes in measurements can not be evaluated, therefore yaw can not be determined. The

code implementing the ϕ and ψ angles in the get_angle method is shown in Figure 9 below. The term that is zeroed out is an integration term based on gyroscope movement, which we ignored because it caused unpredictable fluctuations in measured angles.

```
// read data values for gyro and accelerometer
read_data(GyroX, GyroY, GyroZ, AccelX, AccelY, AccelZ);

// calculate phi and psi for pitch and roll
// phi is used to determine pitch
// psi is used to determine roll
phi = atan2(AccelX,(sqrt(AccelY*AccelY + AccelZ*AccelZ)));
psi = atan2(AccelY,(sqrt(AccelX*AccelX + AccelZ*AccelZ)));


// Calculate pitch and roll
// Equations for both integration of gyroscope with accelerometer
// and equations that do not use gyroscope

// equations with gyro integration
// pitch_in = 0.98*(pitch + GyroX*dt) + 0.02*phi - pitch_offset;
// roll_in = 0.98*(roll + GyroY*dt) + 0.02*psi - roll_offset;

// equations without gyro integration
pitch_in = round((0.00*(pitch + GyroX*dt) + 1.00*phi - pitch_offset) *
    180/M_PI) * M_PI/180;
roll_in = round((0.00*(roll + GyroY*dt) + 1.00*psi - roll_offset) *
    180/M_PI) * M_PI/180;
```

Figure 9: Implementation of ϕ and ψ code in the get_angle method.

ϕ and ψ are variables used to calculate pitch and roll angles. Pitch and roll calculations can be done with the implementation gyroscope measurements alongside accelerometer measurements, however, due to gyroscope drift and other inaccuracies relating to the sensor, we decided to use only the accelerometer. Additionally, we used a roll and pitch offset to account for the initial measurement errors for when the IMU is flat.

To calculate yaw, the magnetometer data is needed. The magnetometer measures the strength and direction of the magnetic field. This data can be used in conjunction with the pitch and roll calculations to find yaw. A snippet of the code used for this method is shown in Figure 10.

```
// Calculates new Magnetometer angles with tilt compensations
nMAGX = MAGX*cos(pitch) + MAGZ*sin(pitch);
nMAGY = MAGX*sin(roll) * sin(pitch) + MAGZ*sin(roll)*cos(pitch);

// calculates yaw using new magnetometer readings
yaw_in = atan2(-nMAGY,nMAGX) - yaw_offset;

// sets pitch, roll, and yaw to be stored for future use
pitch = pitch_in;
roll = roll_in;
yaw = yaw_in;
```

Figure 10: Determining Yaw

First, using the pitch and roll calculations, new magnetic field measurements for the X and Y directions can be calculated. This allows for effects of tilt to be accounted for so that X and Y readings can be accurate, even when the IMU is not parallel to the ground. Yaw can then be calculated from the adjusted magnetometer readings. A yaw offset is used so the initial direction that the plane is facing can be set to 0 degrees. In an ideal scenario, the magnetometer would only have to measure the Earth's magnetic field. However, due to the many large metal objects in the vicinity of the lab space that distorted the magnetic field, its measurements were not reliable. While we were not able to find a method to account for these distortions, an actual implementation of the glider flying outside would not encounter these problems.

Potentiometer Driver

The potentiometer driver is rather simple. The voltage will be measured from a specified GPIO pin and converted into an angle value [degrees]. The conversion between voltage to degrees was determined experimentally. A snipped of code from the potentiometer driver class that shows methods for zeroing and getting the position of the potentiometer is shown in Figure 11 below.

```
/** @brief   Measures position of the potentiometer
 *  @returns The position of the potentiometer in units of degrees
 */
float Potentiometer::get_angle(void)
{
    // Default resolution is 12 bits. Outputs 0 - 4096
    adc_value = analogRead(ADC_PIN);

    // Convert the ADC values to a voltage
    voltage = VOLTAGE_SOURCE * adc_value / ADC_RANGE;

    // Offset the voltage reading and convert to degrees
    float angle = (voltage - voltage_offset) * VOLTAGE_TO_DEGREES;

    return angle;
}

/** @brief   Zeros the position of the potentiometer by setting the offset
voltage
 *           to the voltage measured at its current position
 */
void Potentiometer::zero(void)
{
    // Assuming the potentiometer is positioned at the zero
    // Get current voltage reading
    float current_voltage = get_voltage();

    // Set the offset to the current voltage
    voltage_offset = current_voltage;
}
```

Figure 11: Method for Zeroing and Getting Position of Potentiometer

In the get_angle method, the voltage is read from ADC_PIN, and scaled appropriately with VOLTAGE_SOURCE and ADC_RANGE which converts the ADC reading to a value between 0-3.3V. The angle is then calculated by multiplying the difference between the current voltage and voltage_offset by a constant called VOLTAGE_TO_DEGREES. It is important to note that the relationship between voltage and its angular position is not linear. Rotating the potentiometer 90 degrees in one direction may output a value of 90 degrees but rotating 90 degrees in the other direction from center may output a value of about 140 degrees. The potentiometer also tends to saturate well before reaching its rotational limit. In other words, there is only a small region where the output voltage changes with position and anything beyond that will either read as 0V or 3.3V.

Ultrasonic Sensor Driver

The ultrasonic sensor driver is modified from a script written by Arbi Abdul Jabbaar. We modified Jabbaar's script by converting it into a class. The ultrasonic sensor works by intermittently emitting ultrasonic pulses and measuring the time that it takes to bounce off an object and back to the sensor. A snippet of code which shows how the distance is measured is included in Figure 12 below.

```
/** @brief   Measure the distance between the sensor and the object in
 *  front of it
 *  @returns The distance, in centimeters, between the sensor and the object in
front of it
 */
float ultrasonic::get_distance (void)
{
    distance;       // variable to store calculated distance
    duration;       // variable for the duration of sound wave travel

    // Clears the trigPin condition
    digitalWrite(trigPin, LOW);
    delayMicroseconds(2);

    // Sets the trigPin HIGH (ACTIVE) for 10 microseconds
    digitalWrite(trigPin, HIGH);
    delayMicroseconds(10);
    digitalWrite(trigPin, LOW);

    // Reads the echoPin, returns the sound wave travel time in microseconds
    duration = pulseIn(echoPin, HIGH);

    // Calculating the distance
    distance = duration * 0.034 / 2; // Speed of sound wave divided by 2 (go and
back)

    return distance; // return distance measurement in cm
}
```

Figure 12: Distance Calculation from Ultrasonic Sensor

The trigger pin (trigPin) is set to high for ten microseconds which emits the ultrasonic pulses. The pulseIn function is then used to determine the time it takes for the pulse to bounce off the object and back to the sensor. The 0.034 value is from the 340 m/s speed of sound, and the division by two is from the fact that the pulse will have traveled twice the distance between the sensor and the object before it is received.

Network Module

To host our own webpage from our ESP32, we used a module provided by Dr. John Ridgely which is based on an example by A. Sinha. One drastic modification that we have made to this module is by implementing R"rawliteral()rawliteral". The HTML header from the original code is shown in Figure 13 below.

```
a_string += "<!DOCTYPE html> <html>\n";
a_string += "<head><meta name=\"viewport\" content=\"width=device-width,";
a_string += " initial-scale=1.0, user-scalable=no\">\n<title> ";
a_string += page_title;
a_string += "</title>\n";
a_string += "<style>html { font-family: Helvetica; display: inline-block;";
a_string += " margin: 0px auto; text-align: center;}\n";
a_string += "body{margin-top: 50px;} h1 {color: #4444AA;margin: 50px auto
            30px;}\n";
a_string += "p {font-size: 24px;color: #222222;margin-bottom: 10px;}\n";
a_string += "</style>\n</head>\n";
```

Figure 13: Original Code for HTML Network Header

This is essentially trying to squeeze HTML code into one string which makes it difficult to read and debug if needed, especially with "\" and "\n" scattered throughout. By implementing R"rawliteral()rawliteral", the same piece of code is rewritten in Figure 14 below.

```
a_string += R"rawliteral(
        <!DOCTYPE html>
        <html lang="en">
            <head>
                <meta charset="utf-8">
                <meta name="viewport" content="initial-scale=1, width=device-
                        width">
                <title>)rawliteral";
a_string += page_title;
a_string += R"rawliteral(
                </title>
                <style>
                    html { font-family: Helvetica; display: inline-block; margin:
                         0px auto; text-align:center;}
                    body { margin-top: 50px;}
                    h1 { color: #4444AA; margin:50px auto 30px;}
                    p { font-size: 24px; color: #222222; margin-bottom:10px;}
                    input { width:250px;height:100px;font-size:20px;}
                </style>
            </head>
            )rawliteral";
```

Figure 14: Rewritten HTML Header Code

Although it is not as compact as the original code, we are able to retain the HTML format where the beginning and end of each tag are indented and aligned appropriately. This change is more apparent when writing the body portion of our webpage, shown in Figure 15 below.

```
a_str += R"rawliteral(
    <body>
        <main>
            <div id="webpage">
                <h1>Main Page for ME507 Glider Project</h1>
                <h2>Control Panel</h2>
                <table>
                    <tr>
                        <form action="/activate">
                            <input type="submit" value="Activate Flight
                            Control">
                        </form>
                        <form action="/deactivate">
                            <input type="submit" value="Deactivate Flight
                            Control">
                        </form>
                        <form action="/calibrate">
                            <input type="submit" value="Calibrate/Zero">
                        </form>
                    </tr>
                </table>
                <h2>
                    Manual Control
                </h2>
                <form action="/set_rudder">
                    <input type="text" style="width:150px;height:50px;font-
                    size:20px;">
                    <input type="submit" value="Set Rudder (-90, 90)"
                    style="width:250x;height:50px;font-size:20px;">
                </form>
                <br>
                <form action="/set_elevator">
                    <input type="text" style="width:150px;height:50px;font-
                    size:20px;">
                    <input type="submit" value="Set Elevator (-90, 90)"
                    style="width:250x;height:50px;font-size:20px;">
                </form>
                <br>
                <form action="/">
                    <input type="text" style="width:150px;height:50px;font-
                    size:20px;">
                    <input type="submit" value="Set Rudder Gain"
                    style="width:250x;height:50px;font-size:20px;">
                </form>
                <br>
                <form action="/">
                    <input type="text" style="width:150px;height:50px;font-
                    size:20px;">
                    <input type="submit" value="Set Elevator Gain"
                    style="width:250x;height:50px;font-size:20px;">
                </form>
                <br>
                <form action="/">
                    <input type="submit" value="Reset Default Gain"
                    style="width:250x;height:50px;font-size:20px;">
                </form>
            </div>
        </main>
    </body>
</html>
)rawliteral";
```

Figure 15: HTML Webpage Body

Our webpage consists of three main buttons: one to activate the flight control, one to deactivate the flight control, and one to zero the potentiometers on the glider. When the buttons are pressed, the user is redirected to their respective HTML page: “/activate”, “/deactivate”, and “/calibrate”. The code to handle the “/activate” HTML page is shown in Figure 16 below.

```
/** @brief   Switches the state in a FSM when called by the web server.
 *  @details This method alters a shared variable that contains the current state
 *           of a FSM in the controller task located in main.cpp. The state is
 *           switched to 1.
 */
void handle_Activate (void)
{
    tc_state.put(1);

    String toggle_page = "<!DOCTYPE html> <html> <head>\n";
    toggle_page += "<meta http-equiv=\"refresh\" content=\"1; url='/'\" />\n";
    toggle_page += "</head> <body> <p> <a href='/'>Back to main page</a></p>";
    toggle_page += "</body> </html>";

    server.send (200, "text/html", toggle_page);
}
```

Figure 16: Code for “Activate” Button on HTML Webpage.

When the user is sent to “/activate”, the page refreshes and redirects the user back to “/”. The code shown does not follow the R”rawliteral()rawliteral” formatting described earlier simply because little to no change needed to be made as its only purpose is to redirect the user back to “/”.

PID Controller Class

We decided to use proportional, integral, and derivative (PID) controller for our glider. We concluded that for our proof-of-concept demonstration, only proportional control was necessary for this project. However, we did structure our PID controller task so that integral and derivative control can easily be implemented if we knew the gains. The code for the PID controller class is shown in Figure 17 below.

```
/** @brief Initialize PIDController class
 */
PIDController::PIDController(float Kp_in, float Ki_in, float Kd_in, float dt_in)
{
    Kp = Kp_in;
    Ki = Ki_in;
    Kd = Kd_in;
    dt = dt_in;

    errIntegral = 0;      // Reset integral of error
    errPrev = 0;          // Reset error at previous time
}


/** @brief Set gains to user-inputted values
 */
void PIDController::setGains(float Kp_in, float Ki_in, float Kd_in)
{
    Kp = Kp_in;
    Ki = Ki_in;
    Kd = Kd_in;
}


/** @brief Calculate PID control output at current time
 */
float PIDController::getCtrlOutput(float posCurrent, float posDesired)
{
    float err = posDesired - posCurrent;
    errIntegral += err*dt;
    float derr = (err - errPrev) / dt;

    return ( Kp*err
           + Ki*errIntegral
           + Kd*derr );

    errPrev = err;
}
```

Figure 17: PID Controller Class

The PID controller includes methods to set and update the proportional, integral, and derivative gains as well as a method to get the controller output. To run the controller, the current and desired

values are first passed in as arguments to the getCtrlOutput method. The difference between the desired and current value is multiplied by the proportional gain to get the controller output.

c. Operation of Machine (Tasks and States)

The aircraft system operates using the following six simultaneous tasks:

1. *task_ultrasonic*, which detects the distance below the plane to the nearest object.
2. *task_IMU*, which calculates the current attitude of the plane.
3. *task_webserver*, by which the webpage prompts the user to activate, deactivate, or calibrate the system.
4. *task_controller*, which receives the attitude of the plane and the current motor positions and sends appropriate duty cycles.
5. *task_rudder_motor*, which receives a duty cycle and outputs corresponding signals to the rudder motor.
6. *task_elevator_motor*, which receives a duty cycle and outputs corresponding signals to the elevator motor.

These tasks exchange shares and are timed and prioritized according to Figure 18 below.

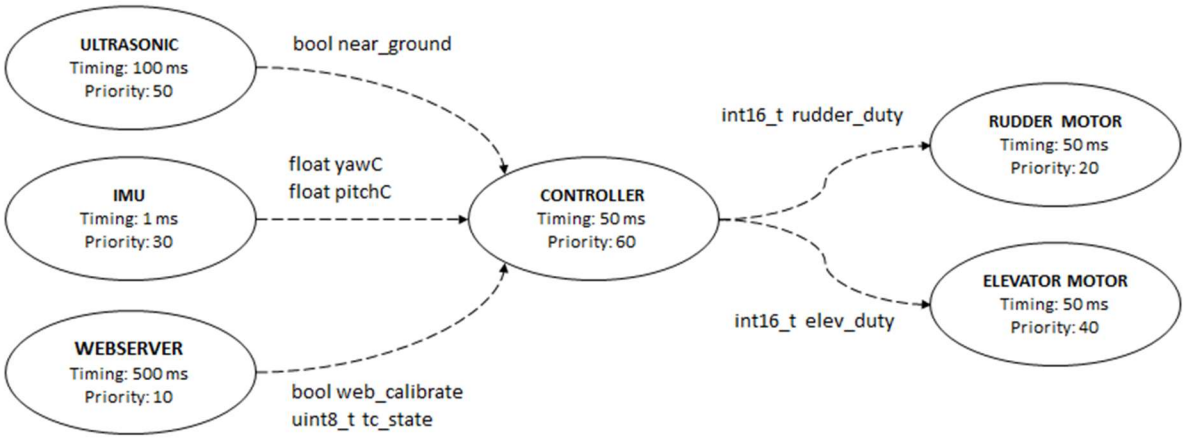


Figure 18: Flight Controller Task Diagram

Sensor Tasks (task_ultrasonic, task_IMU)

The tasks for IMU and Ultrasonic are data collection tasks that follow nearly identical state transition diagrams (Figures 19 and 20). For these tasks, upon start, they immediately initialize their sensors. Then, they move to state 1, where data is collected and stored inside a share.

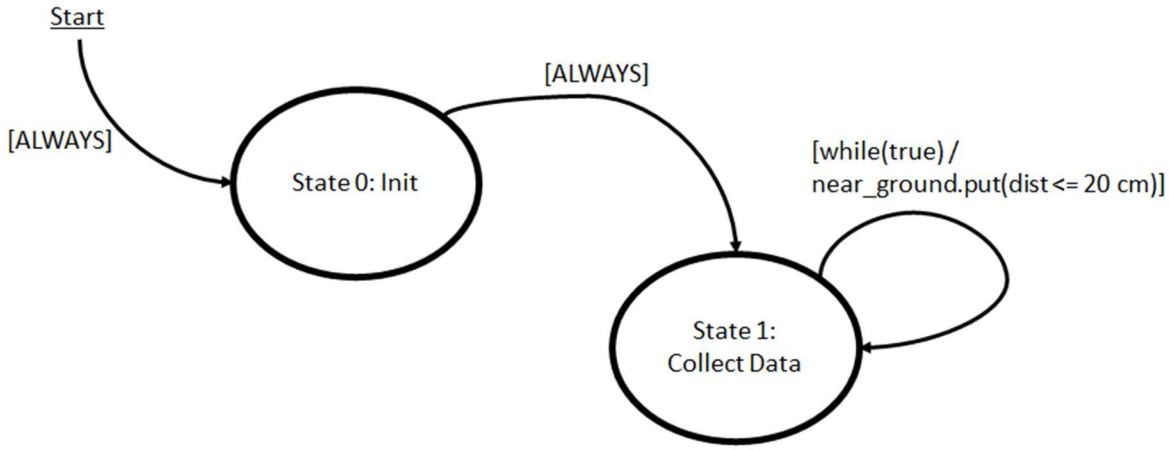


Figure 19: State Transition Diagram for Ultrasonic Sensor Task

The ultrasonic sensor task fetches the detected distance to the nearest object, and then puts to a share called near_ground whether that distance is below a threshold value, which we set to 20 cm. This occurs every 100 ms. The share near_ground affects the operation of the controller task.

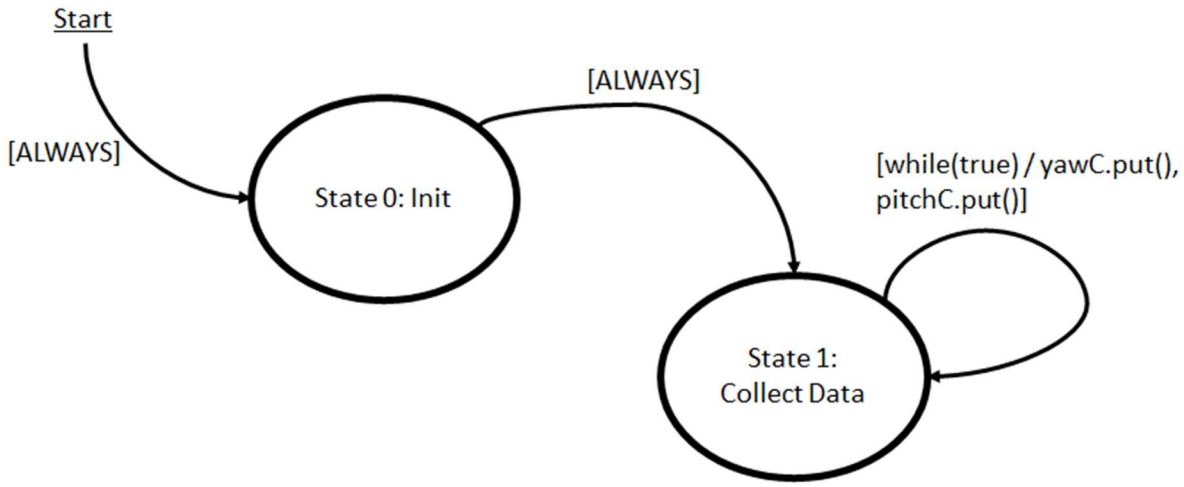


Figure 20: State Transition Diagram for IMU Task

The IMU task fetches the angle of the IMU and puts those values to shares (yawC and pitchC) for use by the controller. The task period is very short to avoid drift in integrated values. Angles were determined based on the direction of gravitational acceleration. For the purposes of the demonstration, roll angle was used in place of yaw since the indoor environment caused issues with magnetometer measurements.

Webserver Task (task_webserver)

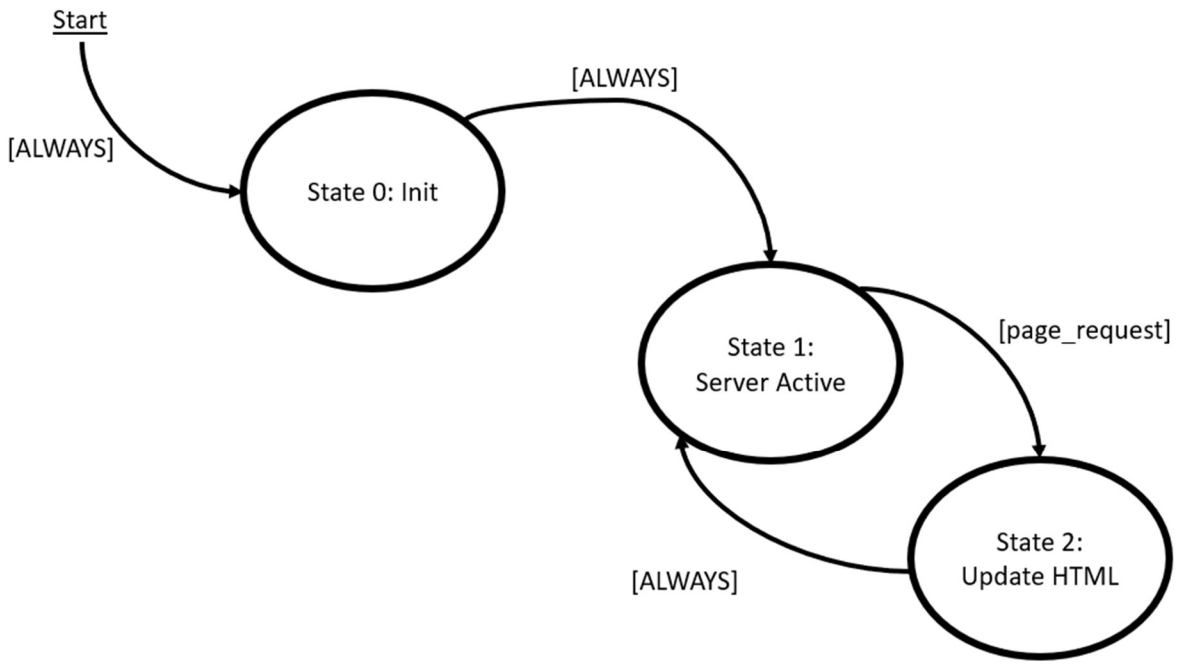


Figure 21: State Transition Diagram for Webserver Task

The state transition diagram for the web server is shown in Figure 21. When it starts, it always initializes the server components necessary for it to function. Then, it always goes to state 1, where the server is active. In state 1, the web server waits for a page update request, then moves onto state 2, where it updates the HTML. After updating the webpage, the server moves back to waiting for another page request.

The webserver task provides the user the ability to activate, deactivate, and calibrate or zero the system. The “Activate” button is necessary to enable the controller. The controller is disabled, and motor duty cycles are set to zero when the “Deactivate” button is pressed. When the user clicks the “Calibrate/zero” button, the controller is disabled, the current heading of the plane is set as the new desired direction, and current position of the control surfaces becomes their new neutral position.

Controller Task (task_controller)

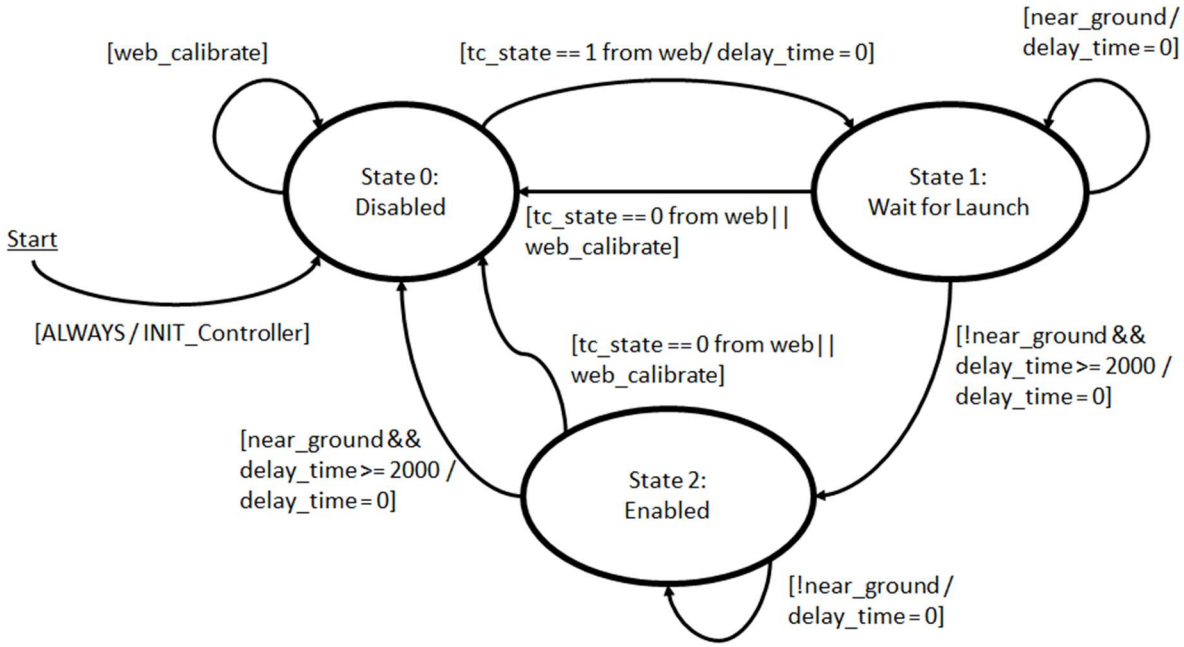


Figure 22: State Transition Diagram for Controller Task

The state transition diagram for the controller task is shown in Figure 22 above. The controller task receives current plane angles from the IMU task (via the shares yawC and pitchC), fetches the current control surface angles from the potentiometers, and outputs motor duty cycles into the shares called rudder_duty and elev_duty, which are received by the motor tasks.

There are three states in the controller task:

0. *Disabled*. Motor duty cycles are held at zero. There is no automatic transition into state 1; this can only be performed by webpage activation through the webserver task.
1. *Wait for launch*. This delay period is established to prevent unpredictable control surface movement while the glider is being thrown. Before launch, the user's hand covers the ultrasonic sensor so that the sensor reads a very small number, i.e., the share near_ground

is set to true by the ultrasonic sensor task. When the user hand-launches the glider, the ultrasonic detects that the plane is no longer being held (near_ground is false); if this is the case for two seconds, the controller will move to state 2 and be enabled.

2. *Enabled*. In this state, the controller will send appropriate duty cycles to the motor tasks. The overall system requires two layers of control, one for the relationship between plane attitude and control surface angle, and one for the relationship between the control surface angle and motor duty cycle. Two layers for each of two systems (rudder and elevator) resulted in four PID controller objects. In each of these, we only used P control, and gains were chosen by empirical tuning. When the plane nears landing (i.e., when near_ground is true), the desired pitch increases to slow the plane. If the plane is near the ground for a continuous length of time, presumably having landed, the controller moves back to state 0, disabled.

Within all states, the controller task checks whether the "Deactivate" or "Calibrate/Zero" commands have been called from the webpage. Both send the controller state to 0. Instead of including copies of this code in every state, these checks are written at the top of the task loop so that they are run during each state.

Motor Tasks (task_rudder_motor and task_elevator_motor)

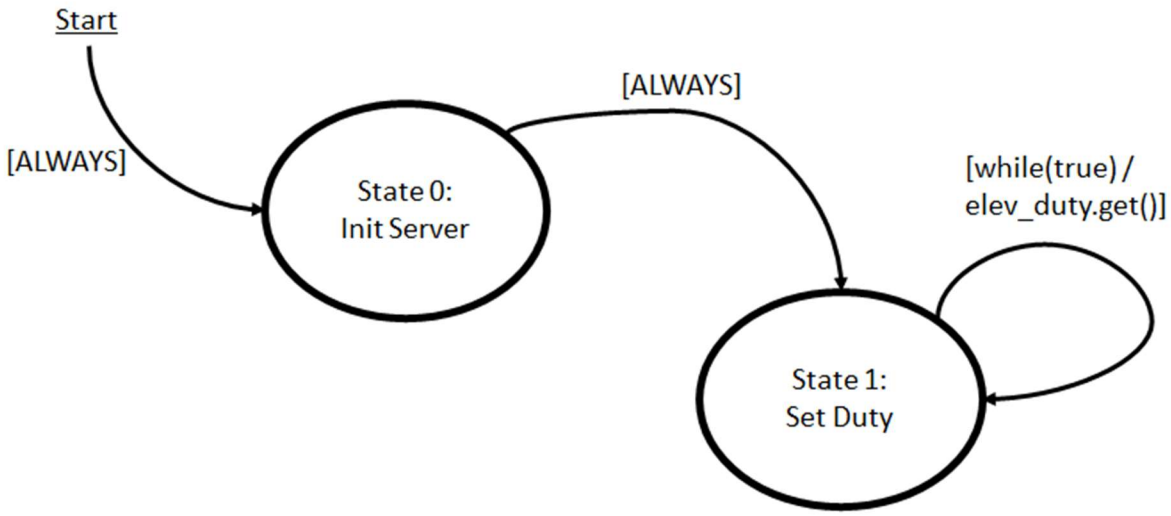


Figure 23: State Transition Diagram for Elevator Motor Task (task_elevator_motor)

The tasks for both the rudder and elevator motor tasks have functionally identical state transition diagrams. Figure 23 shows that of the elevator motor; the rudder motor task uses the share rudder_duty instead of elev_duty. For these tasks, upon start, they immediately initialize the pins necessary for PWM control. Then, they move to state 1, where they wait until a duty cycle instruction is received and then set a duty cycle accordingly.

V. Results

To demonstrate how the flight control system worked, we integrated the system into the model glider. The system was designed to control the rudder and elevator of the model glider, so we attached control rods to the two DC motors in micro servo bodies. The control rods were attached to control horns on the rudder and elevator of the glider. We used a power supply in the lab to power the system, so we could not demonstrate the glider's flight as a wireless system. However, we demonstrated that by moving the IMU attached to a separate breadboard, the rudder and elevator of the glider automatically moved to counteract the movement of the IMU. The flight control system behaved as expected during the lab demonstration. If the IMU was placed in the fuselage of the glider, we could demonstrate that the movement of the glider would directly affect the movement of the control surfaces.

If we were to continue working on this project, it would be ideal to have the system become wireless and demonstrate its ability to safely land the glider automatically. We would also include a landing sequence code that would instruct the control surfaces to push the glider's nose slightly up, in more of a realistic landing configuration. The landing sequence would be activated by the ultrasonic sensor's measured distance to the ground. Due to the time constraint, we were unable to implement integral and derivative control to our PID controller. The added resistance from coupling the potentiometer with the motor increased the stiction in the motor, leading to higher steady state error. As for the ESP32 hosted webpage, we were able to successfully implement three buttons: activate, deactivate, and calibrate. With more time, we would like to set it up so that we can tune each PID gain through the webpage. To do so, we would need to process the user's input, likely through a get or post request. We would also need to make sure that the user's input is valid (i.e., no letters or no special characters). Although this itself is not too challenging, it requires time

which we did not have. Furthermore, with the IMU, there were some additional capabilities that we did not fully investigate. For example, there was a free-fall detection feature that we could have implemented. This would have given us a method to detect stalling and add a free-fall recovery task within the system. Implementing this would have required additional time, calculations, and testing that would not be possible within the span of this class.

Our in-class demonstration did not use all of our PCB components. While we would have liked to test and use all five circuits, we chose to reconstruct some circuits using bread boards and breakout boards to avoid the need to troubleshoot the PCB. The power, potentiometer, and motor circuits worked in testing, but the ultrasonic sensor circuit did not, and the IMU circuit was left untested. Of course, all circuits would need to be operational before a true flight test could be possible, and we would have tested and troubleshot all parts of the PCB if we were in a position in our schedule to do so.

The biggest improvement that we can make to our glider would be directly measuring the orientation of the control surfaces rather than the potentiometers coupled to the motor. One idea we had for this was to couple the potentiometers to the control surfaces instead of the motor. Real airplanes typically have angle-of-attack sensors on their wings to warn pilots of impending stall. This system would benefit from having similar sensors on its wings and control surfaces. Additionally, it would be helpful if our glider had a method to measure air speed. A way to implement this would be to add a pitot tube to the device. By knowing the airspeed, we would have more data to work with to assist in stabilizing the glider. However, due to size constraints on our glider, this would be more applicable for a larger aircraft.

Overall, this project was a great, hands-on learning experience that produced a demonstrable control system with real-world applications.

VI. Conclusion

The mechanical control system discussed in this report has the potential to save lives. While its current form is somewhat simplified as a proof-of-concept system, an automatic stabilization mechanism for general aviation airplanes is a potentially life-saving system. As seen with the Boeing 737 MAX crashes in 2019, a faulty automatic stabilization system also has the ability to end lives. The flight control system developed in this class could positively affect the welfare of pilots and passengers alike, and the safety of those using this system was a continuous concern. In general aviation airplanes, this system is designed to prevent dangerous situations for pilots; for this project, we also considered the safety of those testing the system. For example, we ensured that once the glider had landed, the control surfaces would stop moving to prevent any damage to the model or to people in the area.

The flight control system behaved as expected and demonstrated its ability to automatically move the glider's elevator and rudder in response to movement of the IMU. Overall, this was a successful project for The Airheads and a significant learning experience. This project included an embedded computer, several sensors for measuring operational parameters, a custom circuit board, several actuators, multitasking software to implement controls, digital communication via Wi-Fi, and documentation of the control system development process.

Throughout this project, we each gained experience in designing printed circuit boards and mechanical control systems for real-world applications. Using the skills developed in ME 507, we selected a problem for which mechatronics was a good solution and performed high-level design of a mechatronic system. For this project, we designed complex programs using an organized methodology including task and state diagrams. We designed a printed circuit board for the flight control system and selected appropriate components at reasonable costs. Throughout the project,

we found and utilized design data for system components using resources like data sheets. We documented hardware and software designs through written documentation and code commenting. As a team, we effectively worked to develop a flight control system as a solution to a mechatronics problem.

VII. Appendix

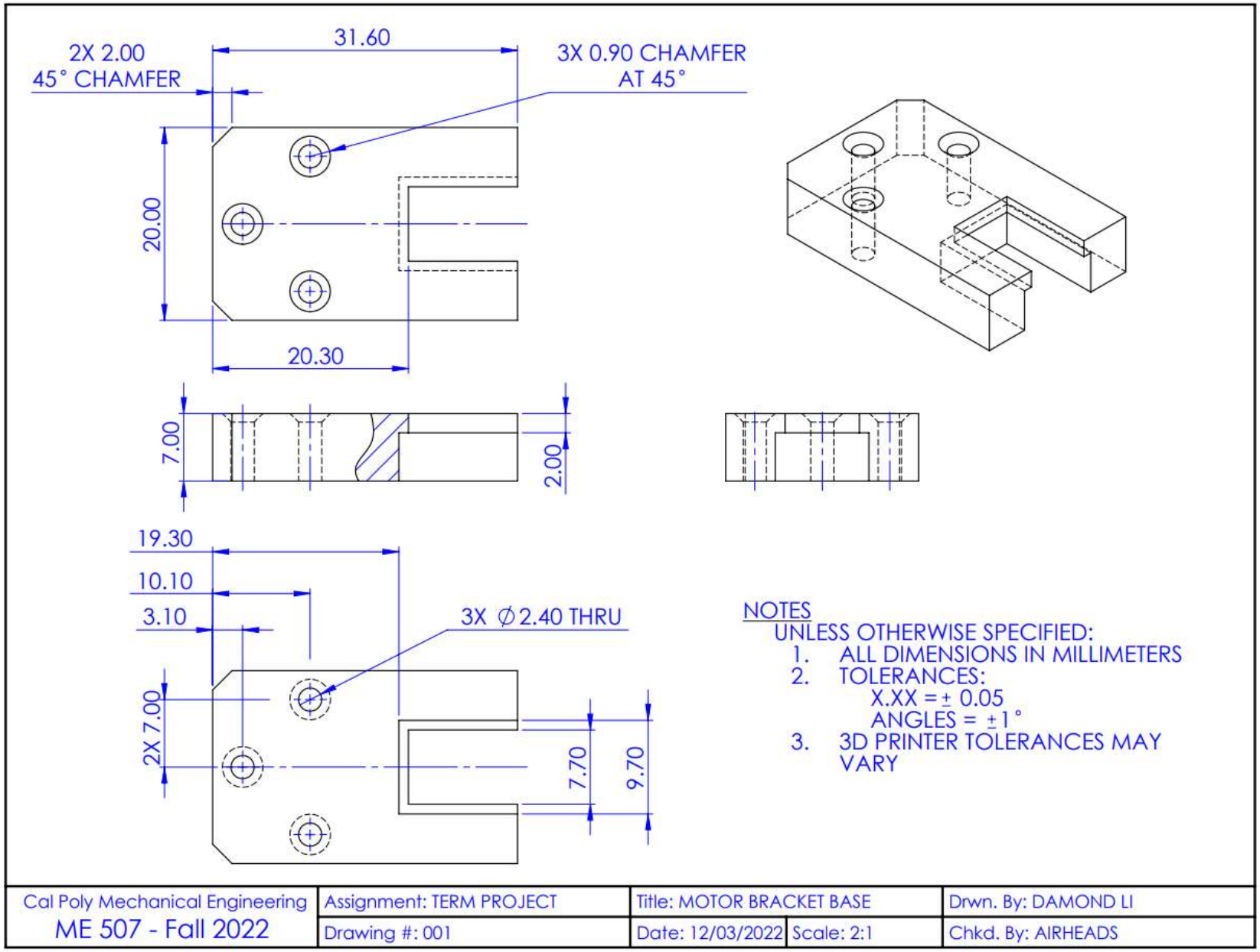
a. Source Code and Documentation

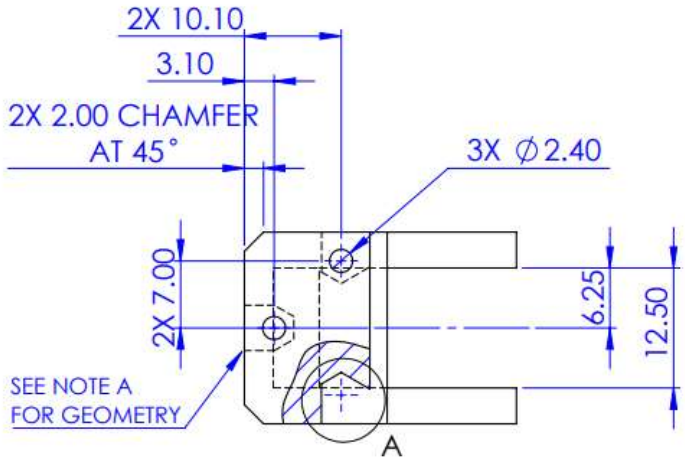
Full Documentation: https://damondli.github.io/airheads/index.html

Source Code Repository: https://github.com/damondli/airheads

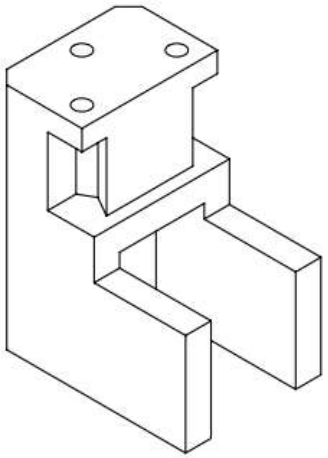
Dr. John Ridgely's ME507 Support Files: https://github.com/spluttflob/ME507-Support

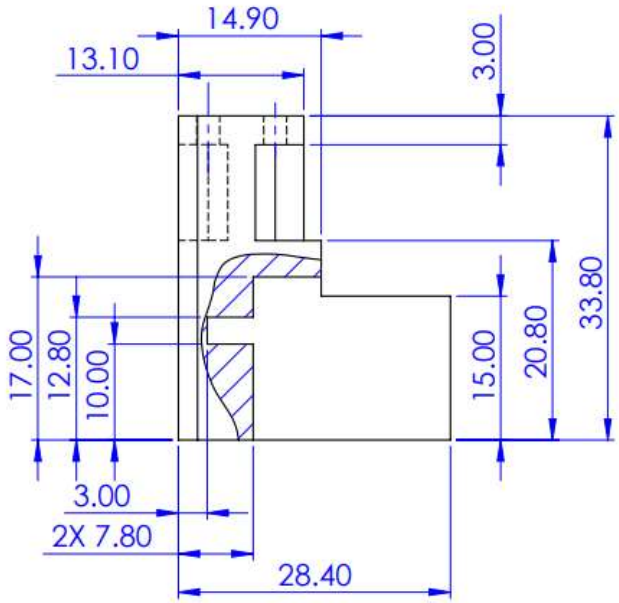
b. Servo Body Mount Design

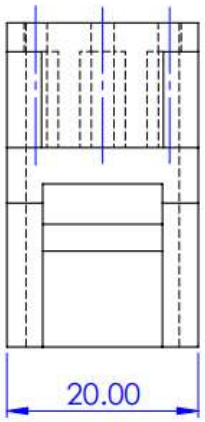


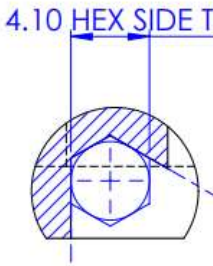


NOTES
UNLESS OTHERWISE SPECIFIED:
1. ALL DIMENSIONS IN MILLIMETERS
2. TOLERANCES:
   X.XX = ± 0.05
   ANGLES = ±1°
3. 3D PRINTER TOLERANCES MAY VARY



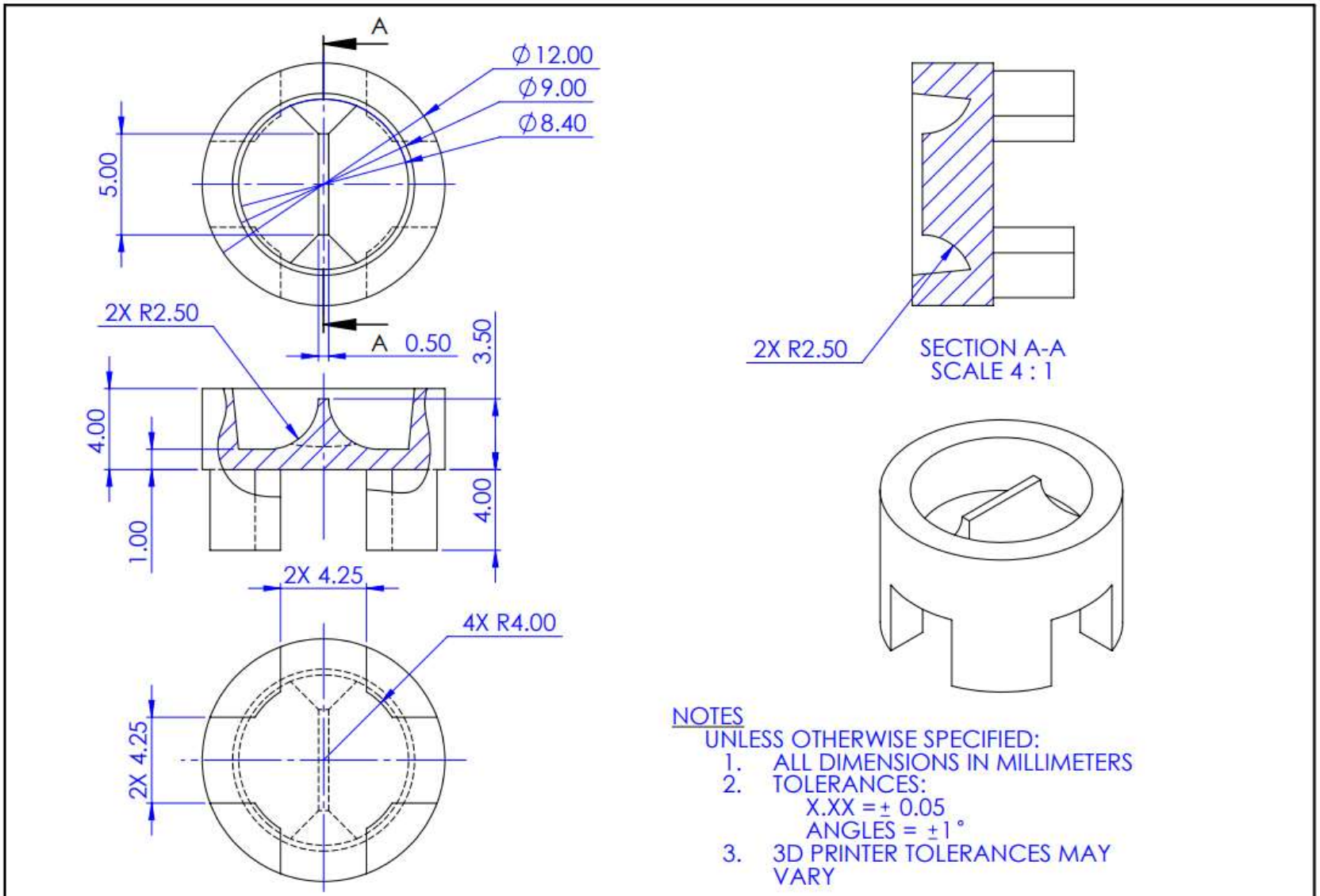




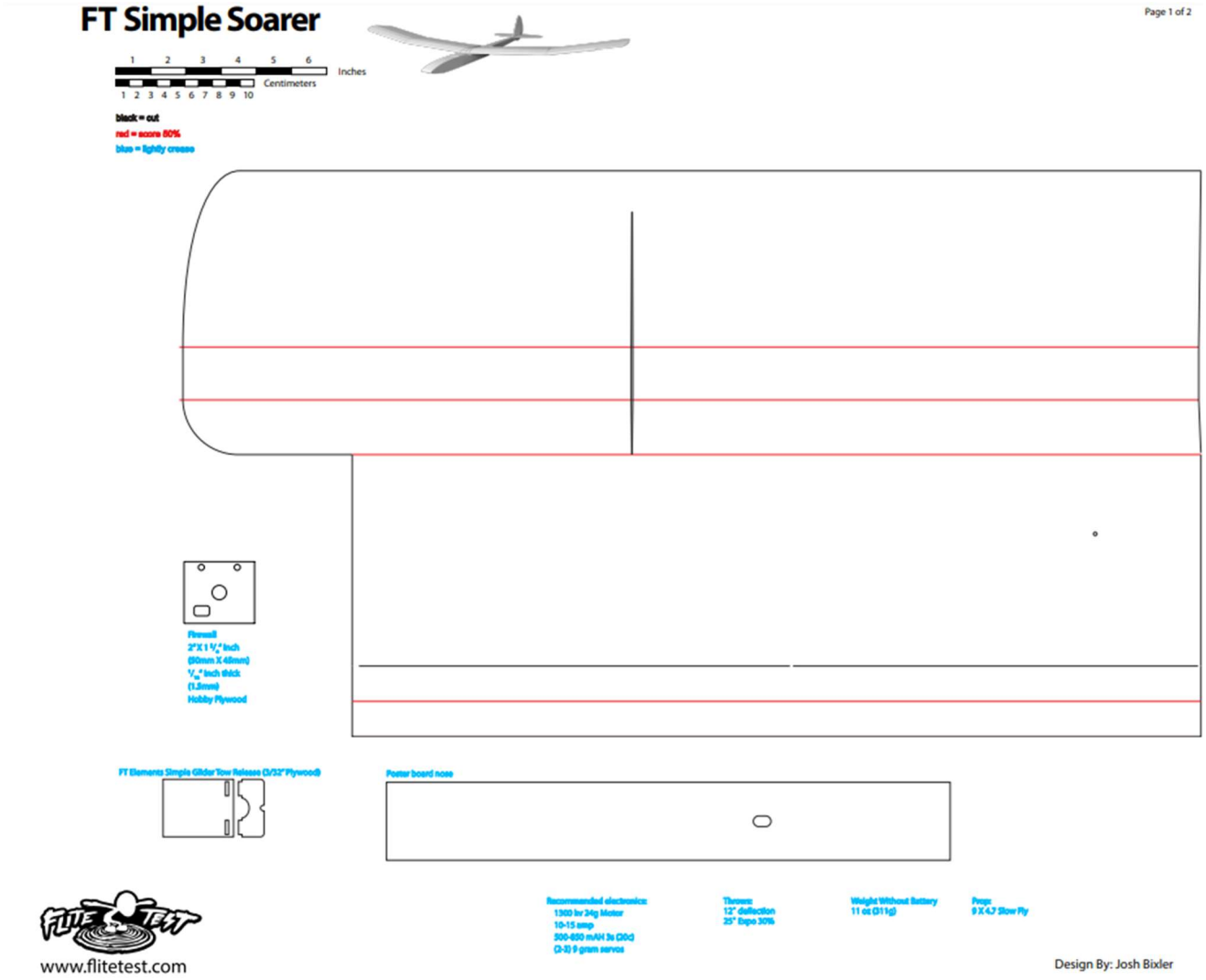


NOTE A:
ARBITRARY HEX ORIENTATION AS LONG AS A NUT CAN SEAT IN POCKET WITH AT LEAST 3 SIDE OF CONTACT - HEX CENTERED WITH HOLE

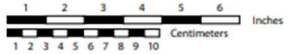
Cal Poly Mechanical Engineering ME 507 - FALL 2022	Assignment: TERM PROJECT Drawing #:002	Title: MOTOR BRACKET BASE Date: 12/03/2022 Scale: 3:2	Drwn. By: DAMOND LI Chkd. By: AIRHEADS


A
Ø12.00
Ø9.00
Ø8.40
5.00
2X R2.50
A 0.50
3.50
4.00
4.00
1.00
2X 4.25
4X R4.00
2X 4.25
2X R2.50
SECTION A-A
SCALE 4 : 1
NOTES
UNLESS OTHERWISE SPECIFIED:
1. ALL DIMENSIONS IN MILLIMETERS
2. TOLERANCES:
X.XX = ± 0.05
ANGLES = ±1°
3. 3D PRINTER TOLERANCES MAY VARY
Cal Poly Mechanical Engineering
ME 507 - Fall 2022
Assignment: TERM PROJECT
Drawing #: 003
Title: ADAPTER
Date: 12/03/2022
Scale: 4:1
Drwn. By: DAMOND LI
Chkd. By: AIRHEADS

c. Model Glider Design by Josh Bixler [1]

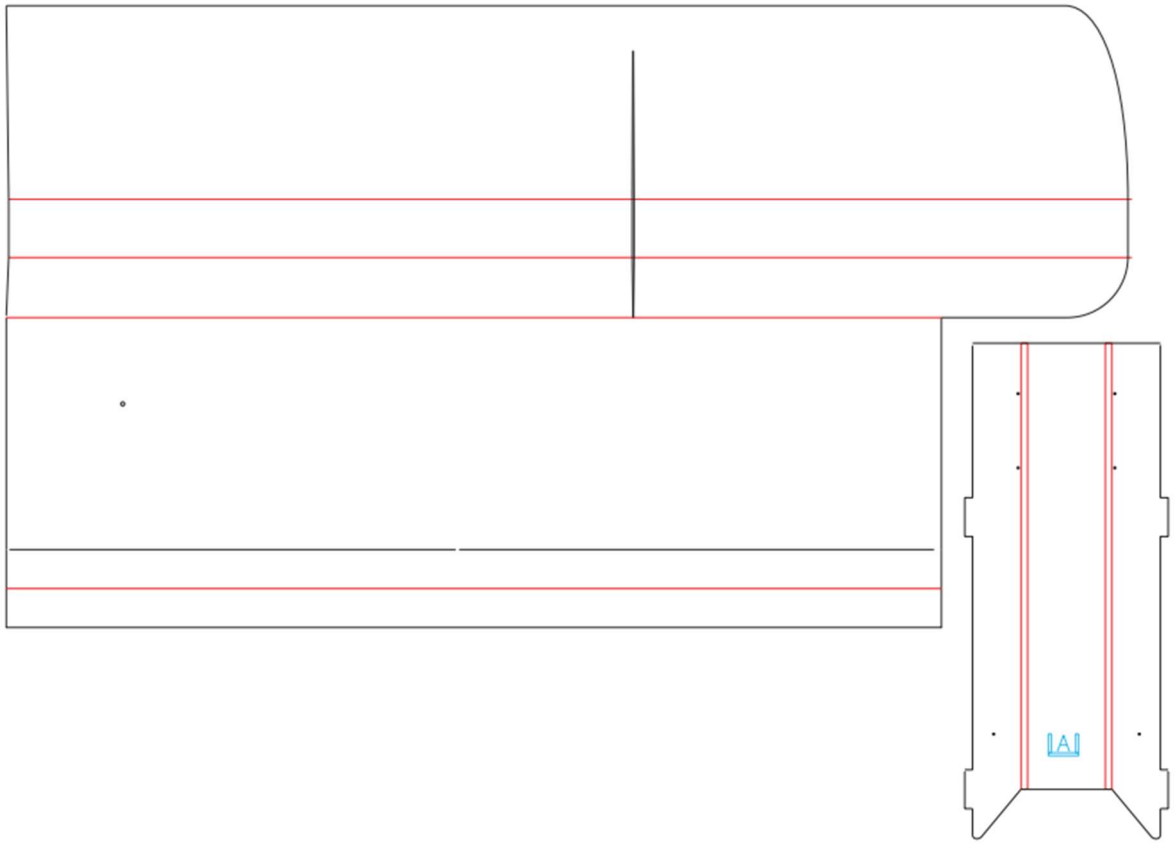


FT Simple Soarer



black = cut
red = score 80%
blue = lightly crease

www.flitetest.com



Design By: Josh Bixler

FT Simple Soarer

www.flitetest.com

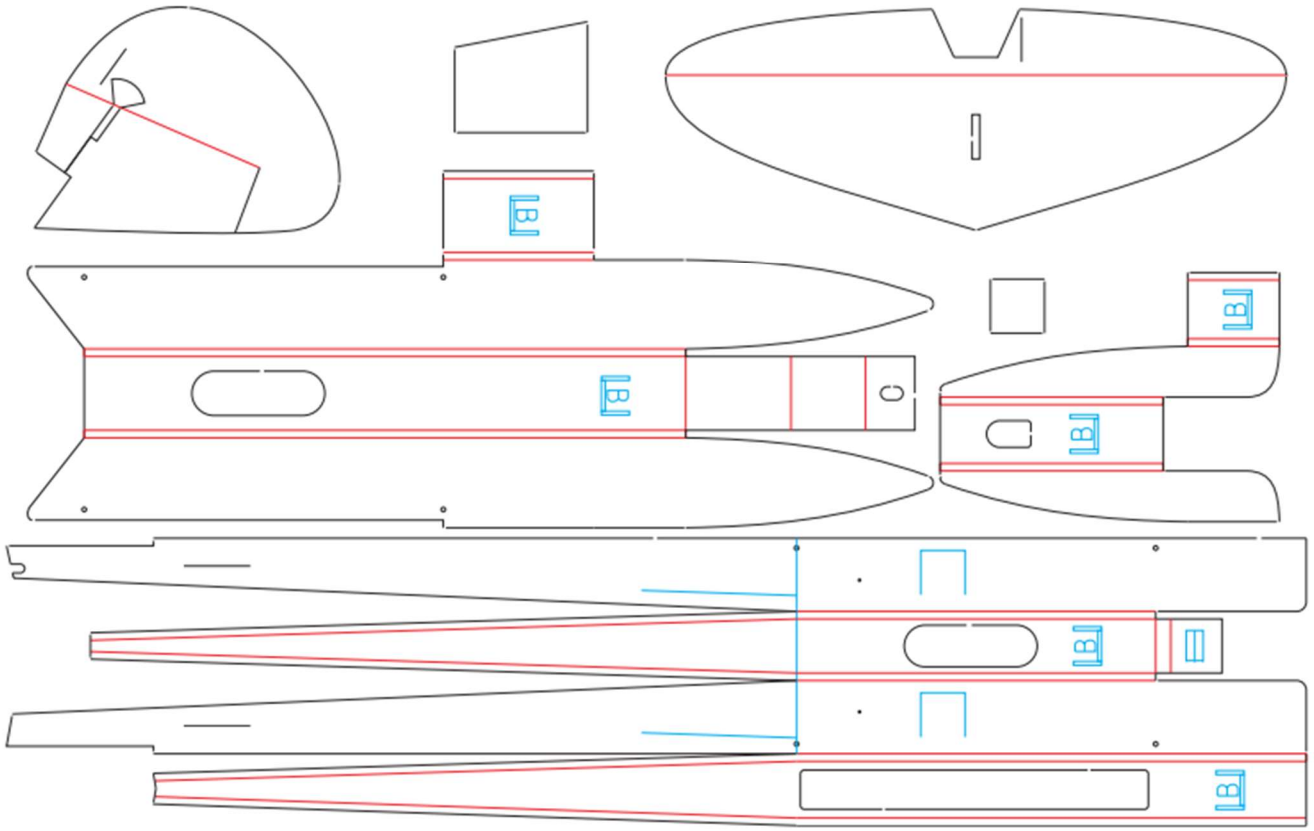
1 2 3 4 5 6 Inches

Centimeters 1 2 3 4 5 6 7 8 9 10

black = cut

red = score 80%

blue = lightly crease



Design By: Josh Bixler

VIII. References

[1] Bixler, John, “FT Simple Soarer”, Flite Test Store. https://store.flitetest.com/ft-simple-soarer-wr-1460mm/.

[2] De Bonet, Lucas, “How to build cheap VR Haptic Gloves to FEEL VR.”, https://www.youtube.com/watch?v=2yF-SJcg3zQ.

[3] Ridgely, John, “ME-507 Canvas Modules”, https://canvas.calpoly.edu/courses/85917.

[4] Ridgely, John, “ME 507 Support Files”, https://github.com/spluttflob/ME507-Support.

[5] Santos, Rui, “ESP32 PWM with Arduino IDE (Analog Output)”, https://randomnerdtutorials.com/esp32-pwm-arduino-ide/.