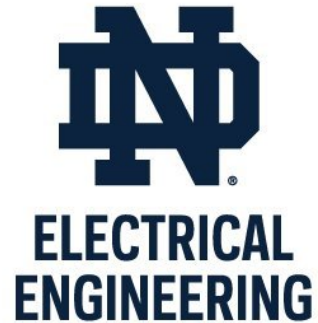


Design Build Fly Senior Design Project: Autonomous Glider

Final Report



Ricardo Ortiz, Daniel Noronha, Matthew Zagrocki

University of Notre Dame
Department of Electrical Engineering
Senior Design II

Table of Contents

1. Introduction.....	5
1.1. Problem Overview.....	6
Figure 1. The Glider's Bonus Points Landing Box.....	7
1.2. Solution Overview.....	7
Figure 2. Assembled Glider Isometric Projection in Autodesk Fusion 360 CAD...	9
1.3. Summary of Results.....	9
Figure 3. Final Assembly of the Glider.....	10
2. Detailed System Requirements.....	11
Table 1. System Requirements.....	11
2.1. Subsystem Hardware Requirements.....	13
2.1.1. Microcontroller.....	13
2.1.2. Sensors.....	14
2.1.3. Power.....	14
2.1.4. Flight Control Actuation.....	14
2.1.5. Release Detection and Strobe Lights.....	14
2.1.6. Flight Data Recorder.....	15
2.2. Software Requirements.....	15
3. Detailed Project Description.....	16
3.1. System Hardware Block Diagram.....	16
Figure 4. System Hardware Block Diagram.....	16
3.2. Power Components.....	17
Figure 5. Power System Block Diagram (excludes 3.3V-5V boost converter for strobe LEDs).....	17
Figure 6. Power System Schematic (1).....	17
Figure 7. Power System Schematic (2).....	18
3.3. Sensor Selection.....	20
3.4. Programming and Processing Capabilities.....	22
Figure 8. ESP32 Microcontroller and Programming Components Schematic.....	22
3.5. Flight Control Surface Actuation Hardware.....	24
3.6. Release Detection and Strobe Lights.....	25
Figure 9. Strobe lights remained on after a test deployment and landing.....	26
3.7. PCB and RF Design.....	27
Figure 10. Final PCB Layout.....	27
Figure 11. RF Design: GND stitching vias and impedance-matched antenna trace..	

27	
Figure 12. JLC0416H-7628 4-layer impedance-controlled PCB stackup description (1.6 mm thickness with 1 oz. outer copper weight and 0.5 oz. inner copper weight).....	27
Figure 13. Final 3D PCB Design CAD.....	29
3.8. System Software Block Diagram.....	29
Figure 14. ESP32 FreeRTOS Control Code Flowchart.....	30
3.9. Autopilot.....	31
3.10 Structural Design and Assembly.....	33
Figure 15. Glider Bottom (left) and top (right) assembly.....	34
3.11 Release Mechanism.....	35
Figure 16. RC Aircraft Mounted Release Mechanism.....	36
4. Testing.....	37
4.1. Flight Data Acquisition.....	37
Figure 17. Example of impartial flight test data collection.....	37
4.2. Release Detection and Strobe Lights.....	37
4.3. Servo Actuation (Ground Test).....	38
4.4 Flight Testing.....	38
5. Instruction Manual.....	40
6. Potential Design Improvements.....	42
6.1. Aerodynamic Design and Flight Stability.....	42
6.2. Release Mechanism.....	42
6.3. Homing (Proportional Integral Derivative Controller) Algorithm.....	42
7. Conclusion.....	44
8. Appendix.....	45
8.1. Electrical Schematic.....	46
Figure 18. Electrical Schematic Page 1 of 6.....	46
Figure 19. Electrical Schematic Page 2 of 6.....	47
Figure 20. Electrical Schematic Page 3 of 6.....	48
Figure 21. Electrical Schematic Page 4 of 6.....	49
Figure 22. Electrical Schematic Page 5 of 6.....	50
Figure 23. Electrical Schematic Page 6 of 6.....	51
8.2. PCB Layout.....	51
Figure 24. Circuit Board 3D View (1).....	52
Figure 25. Circuit Board 3D View (2).....	52

Figure 26. Circuit Board Layout View.....	52
Figure 27. Circuit Board CAD Drawing (Multiple Views).....	53
8.3. Source Code Listing.....	54

1. Introduction

This year, the AIAA Design Build Fly Competition (held in Tucson, AZ) outlined our primary goal: to build an autonomous glider that separates from the main aircraft mid-flight, flashes strobe lights, and lands in a designated landing zone. Teams are scored based on several criteria, including the number of laps the main aircraft flies, the weight of the glider, and how close the glider lands to the landing zone. Our Senior Design team, partnering with the Design Build Fly (DBF) Club of Notre Dame, has taken on the challenge of creating the glider for this project. The DBF Club is responsible for the structure and aerodynamics of the glider. Our senior design group will handle the electronics for the project. These include a printed circuit board equipped with an ESP32 microcontroller that processes data input from several sensors and actuates the aircraft's control surfaces accordingly. This final report provides a comprehensive overview of system requirements, design, testing, and results.

1.1. Problem Overview

The AIAA Design Build Fly Competition has a list of requirements and constraints on the glider aircraft that must be met in order to be eligible for the competition.

Firstly, the glider can have a maximum weight of 0.55 pounds (250 grams). Teams are allowed to determine means of flight control and navigation. However, no radio controlled receivers are allowed to be integrated onto the glider. The glider must fit between the two external fuel tanks on the airplane and be secured to the airplane for all stages of flight, except for the mission during which it is launched. There is a minimum gap of 0.25 inches between any part of the airplane fuselage and the wings of the glider. The glider must have strobe lights that turn on after it is released from the airplane. No points will be received if the lights turn on before launch, or fail to turn on after launch.

The glider must be launched from the Design Build Fly club's main RC airplane at an altitude of 200-400 feet above the ground. To achieve bonus points, the glider must release itself from the airplane and execute a 180 degree turn. Then, using a descending or gliding pattern of choice, the

glider will land on the ground. If the glider comes to rest within one of the landing zones as shown in Figure 1, bonus points will be awarded. The scoring calculation is shown below in Equation 1.

$$Score = 2 + \# \text{ of laps flown} + \frac{\text{Bonus Box Score}}{\text{Glider Weight}} \quad (1)$$

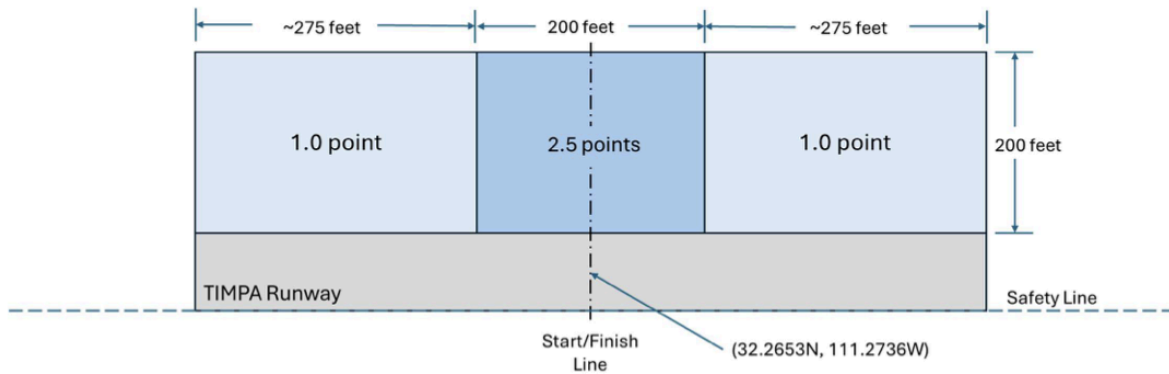


Figure 1. The Glider's Bonus Points Landing Box

The goal for our team is to allow the glider to receive as many bonus points as possible. Therefore, it is critical the glider lands in the highest-scoring landing zone, makes a successful 180 degree turn, and has a working set of strobe lights that both turn on at the correct time and are visible by the judges, while being as lightweight as possible.

1.2. Solution Overview

Our team was primarily responsible for the electronics inside the glider. To complete this project successfully and achieve the highest possible competition score, we implemented several devices that work together effectively and efficiently. At the core of our circuit board is the ESP32 microcontroller chip, which handles data collection, processing, and output. Several sensors

interface with the microcontroller, including an inertial measurement unit (IMU), a differential pressure sensor with a pitot tube (for airspeed), and a GPS module with ceramic chip antenna. These sensors provide critical data for the glider's navigation and autonomous control. All flight data is recorded to an on-board microSD card and the entire system is powered by a 7.4V 2S LiPo battery (300mAh) with appropriate level-shifting circuitry.

A proportional control algorithm processes the incoming sensor data to determine precise adjustments necessary for stable flight and optimal control surface actuation to perform the desired maneuvers (180 degree turn and landing in the box). This control strategy ensures the glider accurately targets the landing zone, maximizing potential bonus points from the judges.

For control surface actuation, the ESP32 controls two servo motors (via PWM) connected to the glider's pitcherons. This allows the entire wing surface to rotate, ensuring adequate pitch and roll authority. The design eliminates the need for an actuated rudder to reduce weight but includes a fixed vertical stabilizer for aerodynamic stability.

Additionally, our team also had input on several of the structural and aerodynamic features of the glider. To ensure the main electronics and fuselage of the glider survived impact upon landing, carbon fiber plates were used with cutouts for the release mechanism and for weight reduction and a wooden bar was used to extend the tail section for improved stability. The tail section is also inverted so that the glider fits underneath the main aircraft's fuselage and the overall design is a high wing design as shown in the Figure 2 3D CAD rendering.

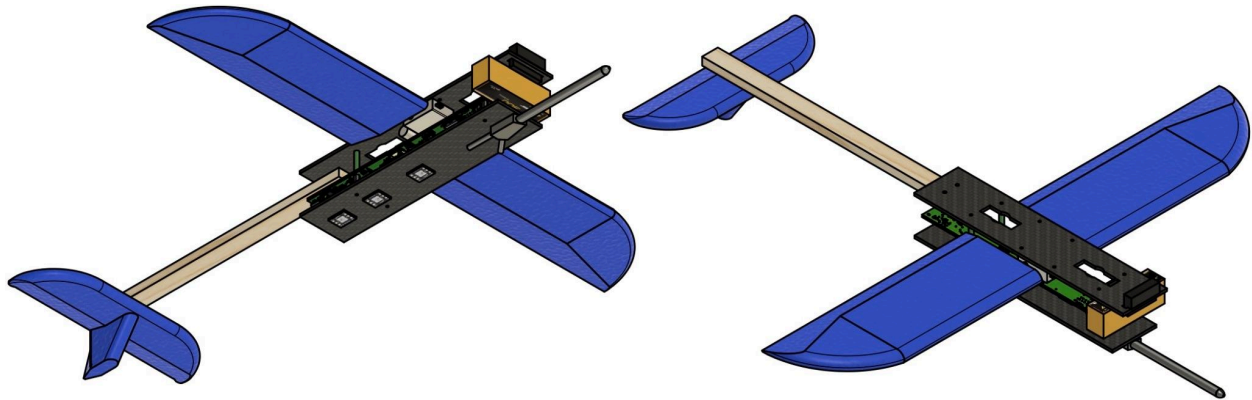


Figure 2. Assembled Glider Isometric Projection in Autodesk Fusion 360 CAD

1.3. Summary of Results

The glider and its release mechanism can be demonstrated successfully on the ground. As soon as it is released, the glider immediately activates its control surfaces to perform a 180 degree right hand turn and begins flashing the strobe lights. As the right roll angle is increased manually, the control surfaces respond by returning to their neutral position in a proportional manner. If done outdoors (for GPS fix), once the 180 degree turn has been completed, the glider will actuate its control surfaces to aim the glider in the direction that points towards its pre-programmed GPS target coordinates and continuously updates this direction (and hence the control surface commands) as the glider moves. Once the glider is approximately wings level and pointing in the right direction, the pitcheron control commands focus on maintaining the airspeed within a certain range by adjusting pitch to prevent stalling or overspeeding. Flight envelope protections can also be demonstrated as if roll and/or pitch are outside the predefined flight envelope, the control surfaces actuate maximally in the opposite direction to restore stable flight. All flight data is also successfully logged to our on-board microSD card and can be retrieved and examined later. An image of the final construction of the glider can be found below in Figure 3.

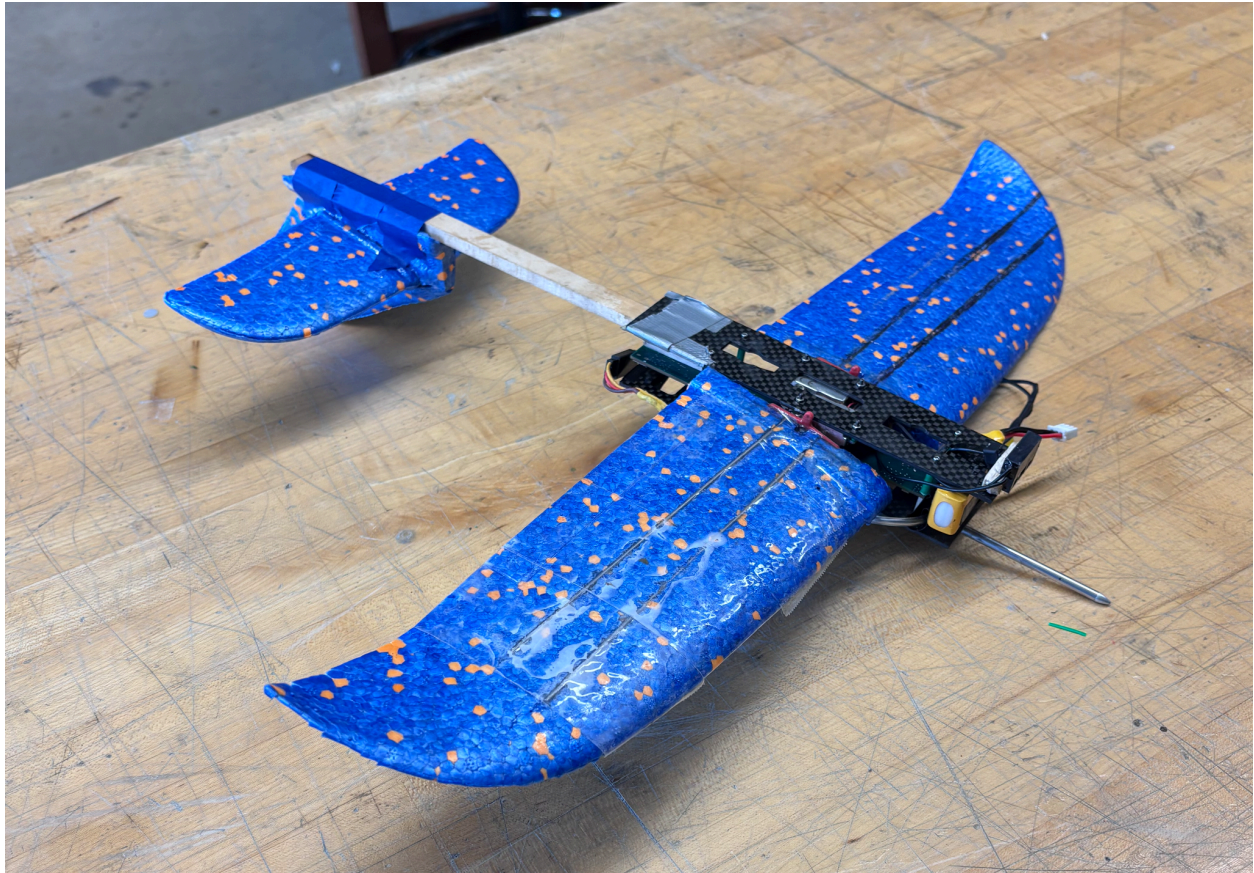


Figure 3. Final Assembly of the Glider

However, during the competition itself, there was a structural issue caused by the attachment of the wings, which interfered with the release mechanism and prevented the glider from successfully detaching from the aircraft. This issue was later addressed in flight testing.

2. Detailed System Requirements

Table 1 summarizes the system requirements based on the competition rules and derived requirements from the DBF team or the EE Senior Design Course.

Table 1. System Requirements

System Requirement	Category
The X-1 test vehicle is a glider capable of autonomous flight.	Rules
The X-1 test vehicle will be launched from the airplane at an altitude of 200-400 feet above ground level.	Rules
The X-1 test vehicle must transition to stable flight after release from the airplane	Rules
The X-1 test vehicle must execute a 180-degree turn after launch	Rules
The X-1 test vehicle must fly a descending pattern or orbit of the teams choosing until landing on the ground	Rules
The X-1 test vehicle must land in one of the bonus boxes shown in Figure 1 or else no bonus points will be awarded.	Rules
The X-1 test vehicle shall have flashing lights or strobes that come on after release from the airplane. If the lights come on before launch or the lights fail to come on after launch, no bonus points will be awarded.	Rules
After the X-1 test vehicle comes to rest in a bonus box, the lights/strobes must still be working (flashing) to achieve bonus points.	Rules
The X-1 test vehicle flight control and navigation may include an	Rules

autopilot/flight control with or without GPS	
No RadioControlled receivers may be integrated into the X-1 test vehicle.	Rules
The X-1 test vehicle shall have a maximum weight of 0.55 lbs.	Rules
The X-1 test vehicle must be carried underneath the airplane fuselage	Rules
There must be a minimum gap of 0.25 inches between any part of the airplane fuselage, wings, or outer surface and X-1 test vehicle wings.	Rules
The X-1 test vehicle must be secured to the airplane for all phases of flight – take-off, flight, and landing – other than intentional launch in Mission 3.	Rules
The X-1 test vehicle must be capable of a commanded release from the airplane via the pilot's transmitter.	Rules
After completing the first lap or any subsequent lap, the X-1 test vehicle will be released after crossing the start/finish line and prior to executing the upwind turn. Each team will determine the number of laps flown prior to launching the X-1 test vehicle. The X-1 test vehicle must be launched to achieve a successful mission score.	Rules
The X-1 test vehicle must come to rest on the ground within the 5-minute flight window for any applicable bonus points to count.	Rules
The X-1 test vehicle must use an ESP32 microcontroller	Derived from EE SD
The X-1 test vehicle must be powered by a LiPo battery	Derived from DBF Team

The X-1 test vehicle board must use servos that interface with pitcherons for pitch and roll control	Derived from DBF Team
The X-1 test vehicle must be capable of logging data	Derived from DBF Team
The X-1 test vehicle board must have mounting holes to interface with the rest of the vehicle	Derived from DBF team

It should be noted that not all competition requirements were fulfilled by the senior design team. Some were directly fulfilled by the DBF team. However, coordination amongst both groups was required to fulfill all requirements listed in Table 1.

2.1. Subsystem Hardware Requirements

The main hardware requirements for the project are having a sensor suite capable of providing accurate flight data to a microcontroller that then actuates servos according to a control algorithm to adjust the glider's roll, pitch, and yaw. Reliable release detection and strobe lights are additional requirements based on the competition rules. A robust power delivery system is also critical to ensure that the system performs optimally. Although not strictly required, a flight data recorder that stores all flight data is very useful for post-flight analysis, debugging, and control algorithm tuning.

2.1.1. Microcontroller

The microcontroller chosen needs to have sufficient GPIO pins to receive inputs from all sensors and for motor control. Additionally, it needs to have hardware PWM, I²C, and SPI peripheral support for sensors and a microSD card. It also needs to have a low power consumption as the system must be powered by a battery. At the same time, it also needs to have high performance for real-time data processing with fast program loop cycling (>10Hz). Lastly, as a course requirement, it must be from the ESP32 family of microcontrollers.

2.1.2. Sensors

The sensor suite must be capable of reliably and accurately sensing the glider's orientation, speed, and position. This information must then be sent to the microcontroller for processing. All sensors should use a common data transmission protocol (e.g. I²C) for ease of use with the microcontroller. Additionally, the sensors must have a sufficiently high sampling rate to prevent bottlenecks and delayed control surface actuation.

2.1.3. Power

The system must be powered by a portable power source since it is an autonomous glider. The power source should be lightweight while holding sufficient charge to power the system from the time it is attached to the main aircraft until it lands. Additionally, appropriate level shifting is necessary to ensure that the microcontroller's circuitry only receives 3.3V. Powering the strobe lights and servos will also require appropriate level-shifting circuitry and power multiplexing.

2.1.4. Flight Control Actuation

To minimize weight, only the glider's wings will be actuated (as pitcherons) with the vertical stabilizer section being fixed. This requires two servo motors that are small and lightweight enough to fit on the glider while not drawing too much current when they actuate.

2.1.5. Release Detection and Strobe Lights

The glider's release detection subsystem must be reliable enough to prevent false release detection as that would cause strobe light flashing before actual release, resulting in disqualification. At the same time, it should also not interfere with the glider's physical release mechanism and should decouple very easily as soon as the glider has been physically released.

The only requirements for the strobe lights are that they need to be controlled by the microcontroller in some way (to flash in a strobe pattern and not before release) and that they need to be bright enough to be visible from 200-400 feet in the sky in daylight conditions.

2.1.6. Flight Data Recorder

This subsystem only needs to contain a non-volatile digital storage medium that is large enough to store data from at least one flight. The interface should also not act as a bottleneck that slows down the rest of the system. Therefore, a microSD Card was used to log all flight data from each sensor to a .csv file for post-flight analysis. In addition to relevant sensor data and time, the autopilot finite state machine modes, target roll, pitch, and bearing, as well as pitcheron angle/direction commands are logged. This allows the SD card to act like a ‘black box’ or flight data recorder for the glider.

2.2. Software Requirements

Because multiple sensors need to be read in parallel to avoid bottlenecks, a real-time operating system (e.g. FreeRTOS) needs to be used to perform multiple tasks concurrently. This also allows us to take advantage of the ESP32’s dual core processor so that one core is dedicated only to sensor reading while the other core performs other functions (datalogging, autopilot, strobe light control).

Although a proportional-integral-derivative (PID) control algorithm would be ideal for flight control surface actuation, given time constraints and the amount of flight testing required to fine-tune PID constants taking aerodynamics into consideration, a proportional control algorithm is the minimum requirement for reasonably accurate flight control surface actuation. To maximize performance, all software was written in C/C++ with the Arduino framework and FreeRTOS functions on PlatformIO.

3. Detailed Project Description

Based on the above requirements, the autonomous glider had the following design characteristics.

3.1. System Hardware Block Diagram

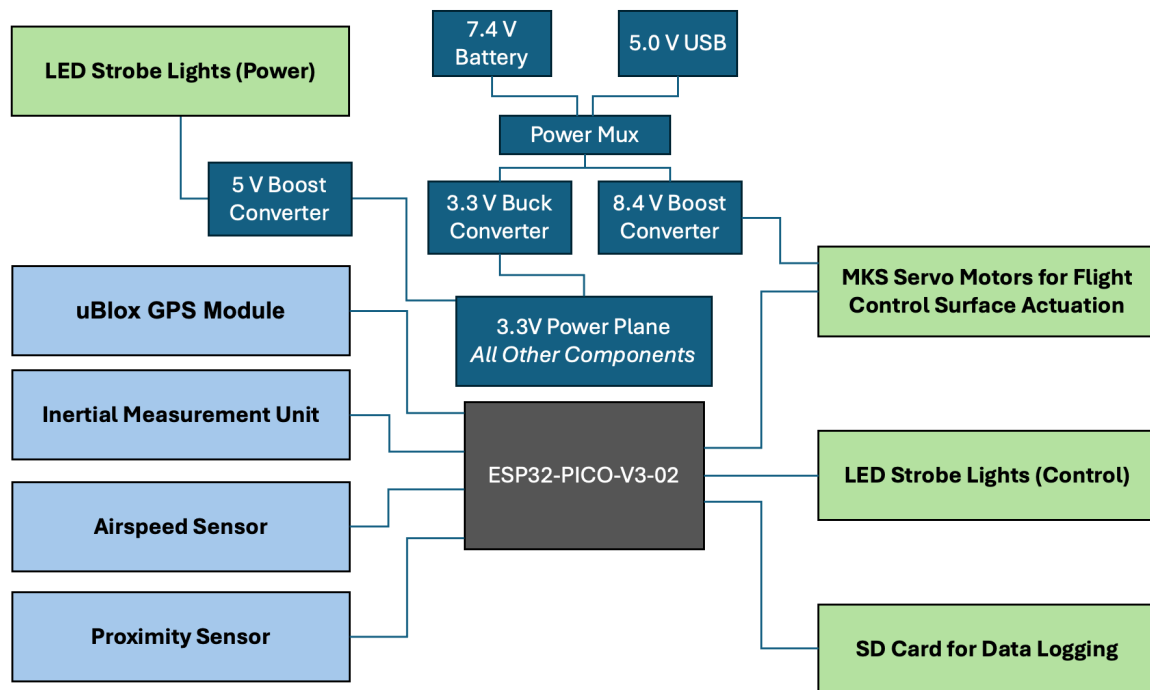


Figure 4. System Hardware Block Diagram

3.2. Power Components

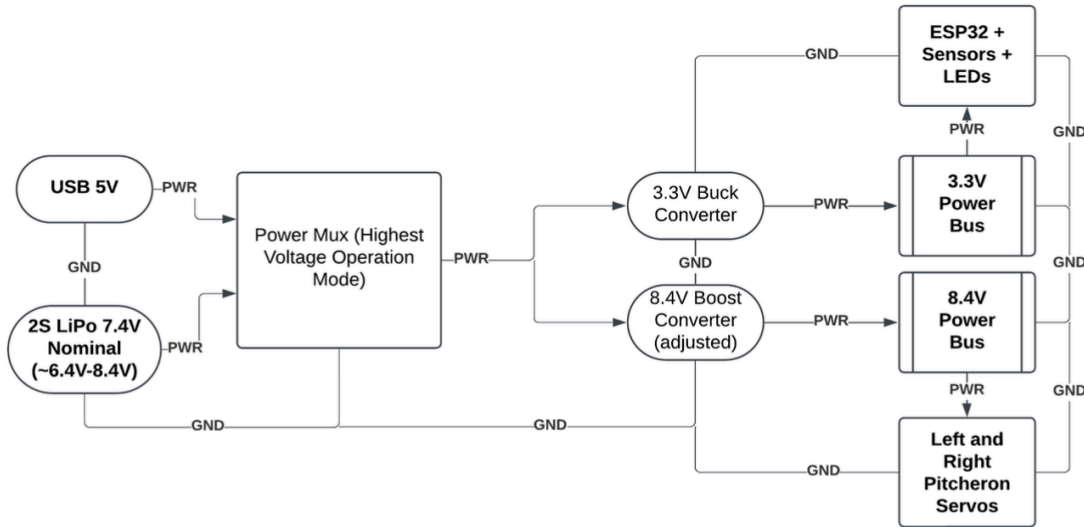


Figure 5. Power System Block Diagram (excludes 3.3V-5V boost converter for strobe LEDs)

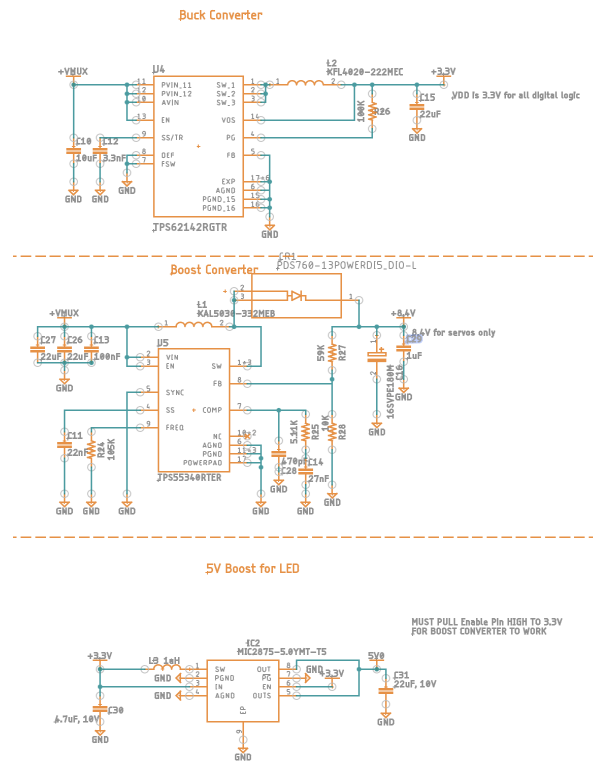


Figure 6. Power System Schematic (1)

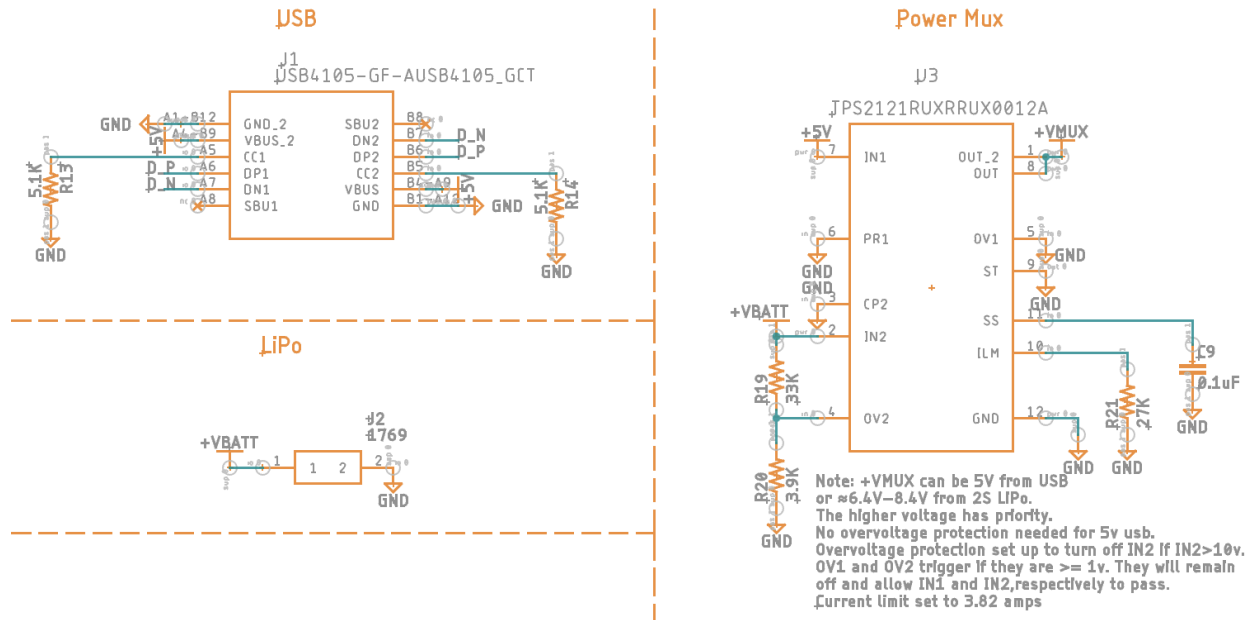


Figure 7. Power System Schematic (2)

- **Source 1:** USB (5V)
- **Source 2:** 300mAh Rechargeable LiPo Battery (~6.4V-8.4V)
- **Power Mux:** Texas Instruments TPS2121RUXR (Highest Voltage Operation, VMUX)
- **DC-DC Buck Convertor:** Texas Instruments TPS62142RGTR (VMUX=>3.3V Out)
- **DC-DC Boost Convertor 1:** Texas Instruments TPS55340RTER (VMUX=>8.4V Out)
- **DC-DC Boost Convertor 2:** Microchip Technology MIC 2875 (3.3V=>5V)

The system can be powered by either 5V USB or a 2-cell LiPo battery or both. If two power sources are used, the power mux will choose the one with the highest voltage.

This allows the servos to run properly on a battery (laptop USB ports limit current causing brownouts on servo actuation) while allowing serial port communication via USB for testing and debugging (e.g. when trimming the servos). When the system is

battery-powered, all sensors and the ESP32 operate on 3.3V. Supplying this 3.3V requires a (fixed) buck converter since the 2S LiPo is at ~7.4V nominally. Operating the servos at maximum torque requires 8.4V, so a boost converter (adjusted to 8.4V) is included with a large 180uF capacitor. This has the added benefit of regulating the servo voltage to minimize jitter and drops in voltage due to current spikes. To supply the 5V required for the NeoPixel LEDs, we could have used a 5V buck converter from the power mux output (~7.4V=>5V) but instead decided to boost the 3.3V buck output to 5V with a boost converter. This is because it minimizes routing complexity given that we designed a 4-layer PCB stackup (Signal-GND-+3.3V-Signal) that allowed very easy access to the 3.3V power plane since that was the most commonly used supply voltage. Although this leads to some efficiency loss, it is worth the tradeoff given that the alternative would be to use a much longer/more convoluted trace that would have similar efficiency losses. All system components share a common ground.

Other power-related components included on the PCB are 4 LEDs: a (3.3V) power LED, a GPIO 'built-in' LED, and TX/RX LEDs used when programming the ESP32 or communicating via UART serial. The NEO-M9N GPS module recommends using a backup battery for hot start, so a 1.5V coin cell is included solely for that purpose. No breakout pins or test points were included for space/weight reasons with the exception of requesting untented vias from the PCB fabrication company.

3.3. Sensor Selection

The three sensor types used to accurately determine orientation, speed, and position are an Inertial Measurement Unit (IMU), differential pressure sensor with pitot tube, and a GPS module with ceramic chip antenna.

- BNO085 Inertial Measurement Unit (IMU) to collect roll, pitch, and yaw data (Euler angles normalized to 0-360 degrees) to determine the orientation of the glider in 3D at all times. This is used to perform the 180-degree turn on release from the main aircraft and for airspeed management via pitch control during descent (after successful GPS homing). Additionally, it is used to determine whether the glider has exceeded its flight envelope (roll and/or pitch) and take corrective action as required.
- ABP2DRRT001PD2A3XX Differential Pressure Sensor connected to a pitot tube that is used to determine the glider's airspeed using the difference between static and dynamic pressure. This differential pressure sensor has a maximum rating of 1 psi, the equivalent of about 200 ft/s, which is much higher than any airspeed it will encounter prior to release from the main aircraft. This is the only through-hole mounted component (excluding screw terminal block). Airspeed information is important to ensure that the glider does not stall or overspeed, which would trigger emergency corrective measures. Outside the envelope protection cases, airspeed data is also used for proportional control of pitch to maintain the airspeed within an ideal predetermined range. After landing, the airspeed reading (of 0) is used to trigger the "Landed" state which returns the pitcheron servos to wings level. Below ~5m/s, the airspeed readings are unreliable and are hence corrected to 0. Above 5m/s, Equation 2 is used to calculate airspeed from known air density (based on local air pressure), ρ , and differential pressure, ΔP .

$$v = \sqrt{\frac{2\Delta P}{\rho}} \quad (2)$$

- NEO-M9N-00B GNSS Receiver Module used to precisely determine the current location of the glider using GPS satellites. This information is used for two purposes. Firstly, knowledge of the current GPS coordinates together with the target GPS coordinates is used to calculate the target bearing required to get from the current location to the target location. Target GPS coordinates have been provided for the TIMPA runway (Arizona) in the AIAA DBF Competition rules (32.2653N, 111.2736W). For testing purposes, the test runway coordinates (in South Bend) will be used instead. Equation 3 is used to calculate the (initial) bearing required for a straight line between the two points: the current GPS coordinates and the target GPS coordinates.

$$\theta = \text{atan2}(\sin(\Delta\lambda) \cdot \cos(\varphi_2), \cos(\varphi_1) \cdot \sin(\varphi_2) - \sin(\varphi_1) \cdot \cos(\varphi_2) \cdot \cos(\Delta\lambda)) \quad (3)$$

Electrical schematics for all sensors (and the entire system) are included in the appendix.

3.4. Programming and Processing Capabilities

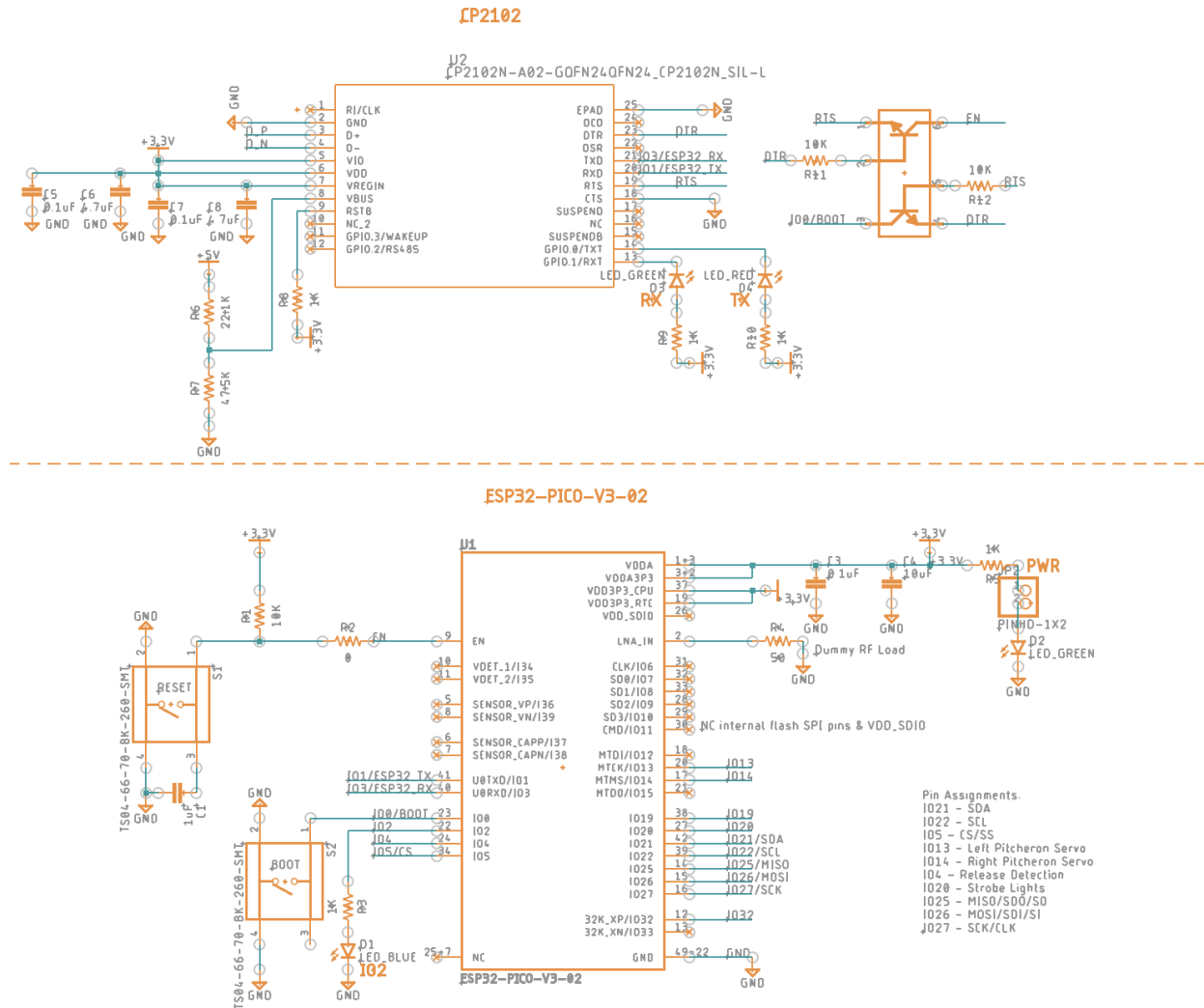


Figure 8. ESP32 Microcontroller and Programming Components Schematic

This subsystem consists of the following components:

- The ESP32-PICO-V3-02 microcontroller was chosen specifically because it is a system-in-package (SIP), which contains an oscillator and flash memory embedded into a single IC package which is extremely light and has a very small footprint

(7mmx7mmx1.11mm). This is important because the team's competition score for the glider payload mission is inversely proportional to the glider's weight, so minimizing weight as much as possible is a priority for the team. Moreover, because of the physical dimensions of the glider, the board dimensions must be limited to ~5.64" long but only 0.5" wide. Therefore, we decided not to use a module like the ESP32-WROOM. Only minimal additional circuitry was required (only a few extra 0603 passives like capacitors). A 50Ω dummy load was connected in place of a WiFi/Bluetooth antenna since those capabilities are unnecessary for the mission. Unused GPIO pins also did not need to be broken out. The majority of the code consists of FreeRTOS task functions and functions called by those functions given the complexity of the system and the need for concurrency/parallelism using both processor cores for maximum performance. After initializing all subsystems, the ESP32 goes into light sleep and only wakes up on release detection ('interrupt' on GPIO pin 4), after which all tasks are created (sensor reading, data logging, autopilot, and strobe light) allowing the FreeRTOS scheduler to run.

- The CP2102N USB-UART Bridge was used with BJTs for DTR/RTS to allow programming of the ESP32 through a USB-C port. Differential pair routing had to be used for the USB signals for signal integrity.
- Boot and Reset buttons are included to allow the microcontroller to enter download mode and to reset it respectively. Not including the 0.1uF capacitor on the boot button fixed a minor issue where the reset button had to be pressed to make the microcontroller operate correctly whenever it is first energized.

The ESP32-PICO-V3-02 requires a good 3.3V power supply with at least 0.25mA current-handling capability to operate well while the CP2102 requires both 3.3V for VIO/VDD and 5V from the USB Bus for VBUS.

Communication with the CP2102N USB-UART bridge is done using UART serial (TX/RX). The USB-UART bridge then communicates with the programming device (laptop) via USB differential data signals. Communication with the flight data acquisition subsystem is done using the SPI interface only for the microSD card. All sensors communicate with the ESP32 using the I²C bus on default SDA and SCL pins. Release detection functions like an interrupt (GPIO binary transition detection). All actuators and lights operate through GPIO control via hardware PWM signals.

3.5. Flight Control Surface Actuation Hardware

This subsystem consists of two KST X08H Plus servo motors. These servos actuate the connected pitcherons according to the autopilot task software commands. The manufacturer recommended PWM frequency of 333Hz is used for each servo with a HIGH duration of 900us to 2100us corresponding to -60° to +60° of actuation. The pitcherons combine the traditional elevator and aileron control surfaces into a single surface. Although this reduces size and weight, it limits control authority to only roll or pitch but not both simultaneously without losing effectiveness.

The servo rated voltage is 7.4V but they are capable of operating from 3.8V to 8.4V (albeit less effectively below 7.4V). To maximize servo effectiveness, using an 8.4V power supply for the servos would be ideal. This also happens to be the nominal voltage of a fully charged 2-cell LiPo

battery. Electrical current requirements would be up to 1.5A for both servos combined to allow for a reasonable amount of air resistance acting against the servos in flight.

The servos were also selected due to their small package and lightweight, coming in at only 9.5g each. This helps the team achieve a higher score since mission score is inversely proportional to glider weight. Despite being small and lightweight, they provide 5.3Kgf.cm, which is more than enough torque to handle the loads the servos would experience during flight prior to deployment. This is because the servo gears are made with hardened steel, much stronger than your typical nylon gears found in 9g arduino servos. The main aircraft flew at an airspeed of about 100 ft/s, so making sure the servos could withstand the wingloading at those higher speeds was critical, as any structural failure prior to deployment of the glider would mean an automatic disqualification from the mission.

3.6. Release Detection and Strobe Lights

A magnetic release detection sensor is connected to GPIO pin 4 which is capable of waking up the ESP32 microcontroller from light sleep when a transition from LOW to HIGH is detected. This normally only happens after the glider is released and is required according to the competition rules for strobe light activation. The sensor is connected via a 2-pin screw terminal and release detection functions like an interrupt (GPIO binary transition detection). A debounce delay of 250ms is added to reduce the chance of failure.

The strobe lights chosen are Adafruit NeoPixel RGB LEDs which can be chained together into a strip. They are required to blink after the glider is released and to continue blinking after landing according to AIAA competition rules. The LEDs require quite low current (at most ~60mA per

addressable LED), but require 5V to operate the data line. Since the ESP32 GPIO voltage is 3.3V, logic level shifting is required to correctly supply the LED data pin. This is done using a n-channel MOSFET and 1 kOhm resistors. The 5V supply comes from the output of a 5V DC-DC boost converter that uses the 3.3V plane as its power input.

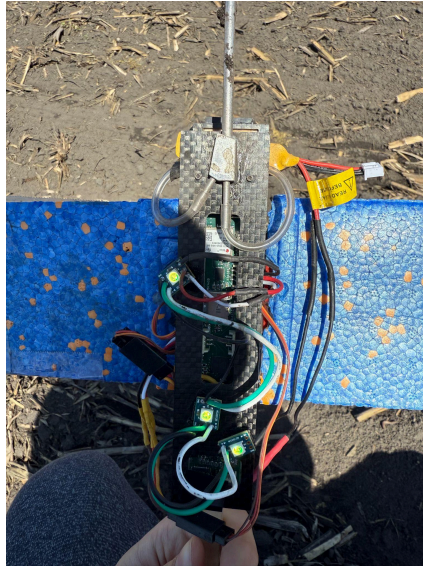


Figure 9. Strobe lights remained on after a test deployment and landing

3.7. PCB and RF Design

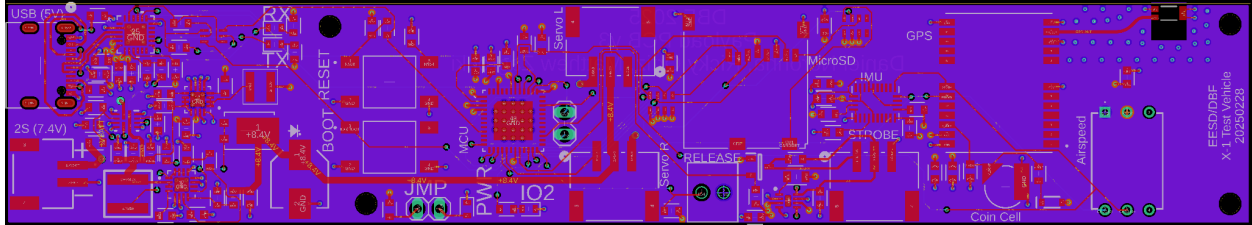


Figure 10. Final PCB Layout

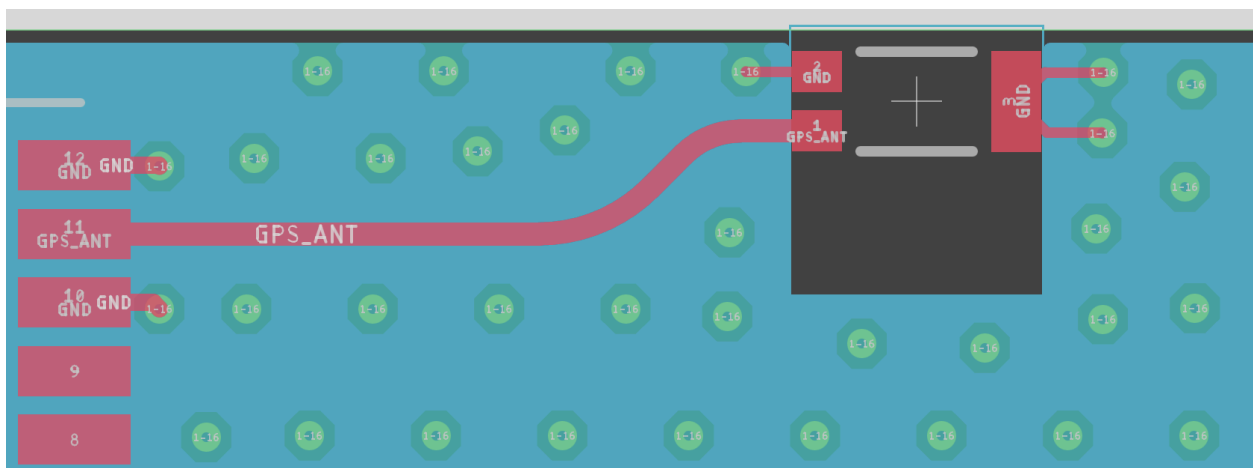


Figure 11. RF Design: GND stitching vias and impedance-matched antenna trace

Layer	Material Type	Thickness	
Top Layer	Copper	0.035mm	
Prepreg	7628*1	0.21040mm	
Inner Layer L2	Copper	0.0152mm	1.1mm H/HOZ with copper
Core	Core	1.065mm	
Inner Layer L3	Copper	0.0152mm	
Prepreg	7628*1	0.21040mm	
Bottom Layer	Copper	0.035mm	

Figure 12. JLC0416H-7628 4-layer impedance-controlled PCB stackup description (1.6 mm thickness with 1 oz. outer copper weight and 0.5 oz. inner copper weight)

A GPS antenna RF circuit was custom-designed for the W3010 passive ceramic chip antenna based on the manufacturer's PCB layout recommendations. The antenna was placed at the very edge of the board with a copper keepout zone maintained on all four layers in the footprint. Additionally, JLCPCB's controlled impedance calculator was used to determine the 50Ω trace width for the JLC04161H-7628 stackup to be 13.75 mil to ensure a good impedance match and minimum signal loss. Ground vias for stitching were also placed around the chip antenna and its feedline according to manufacturer recommendations and the antenna was kept as far away from other components as possible to minimize noise due to unwanted electromagnetic interference.

The PCB schematic and layout were completed in Autodesk Fusion 360 as it provided better mechanical integration than KiCAD. The 3D board model could directly be exported and included in DBF's complete system CAD model. Moreover, Fusion 360 offers better collaboration features since it is cloud-based. Before sending the board out for fabrication, ERC and DRC checks were completed. Additionally, a DFM check was completed using JLCPCB's online tool to avoid unexpected fabrication issues. Overall PCB dimensions were 5.64 inches by 0.5 inches and the 4-layer stackup used consisted of Signal-GND-+3.3V-Signal with untented vias. Figure 13 shows the PCB layout in 3D with component models included. The detailed PCB layout and electrical schematics may be found in the Appendix. With very few exceptions, all passive components were 0603 surface mount devices and board assembly was done in the EIH.

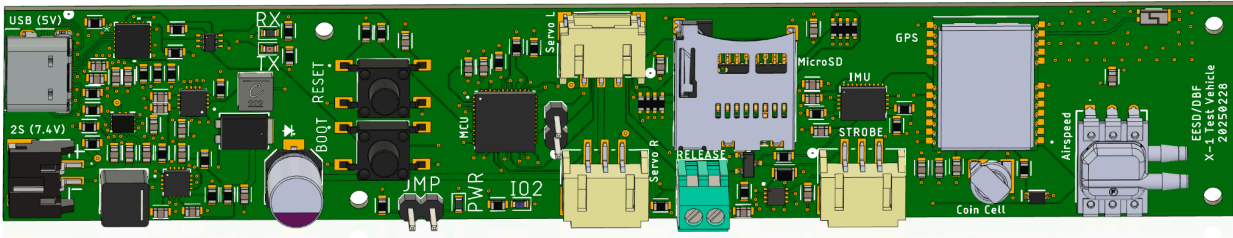


Figure 13. Final 3D PCB Design CAD

3.8. System Software Block Diagram

The flight control software running on the ESP32 microcontroller uses FreeRTOS tasks to maximize system performance by leveraging parallelism across both cores and is outlined in Figure 14. In addition to the flowchart code, an additional mode of operation is included in the code (that can be enabled/disabled with a `#define` flag) to only allow servo trimming based on commands/prompting sent via serial. Additional flags allow serial logging, SD logging, and/or pitcheron servo actuation to be disabled as necessary.

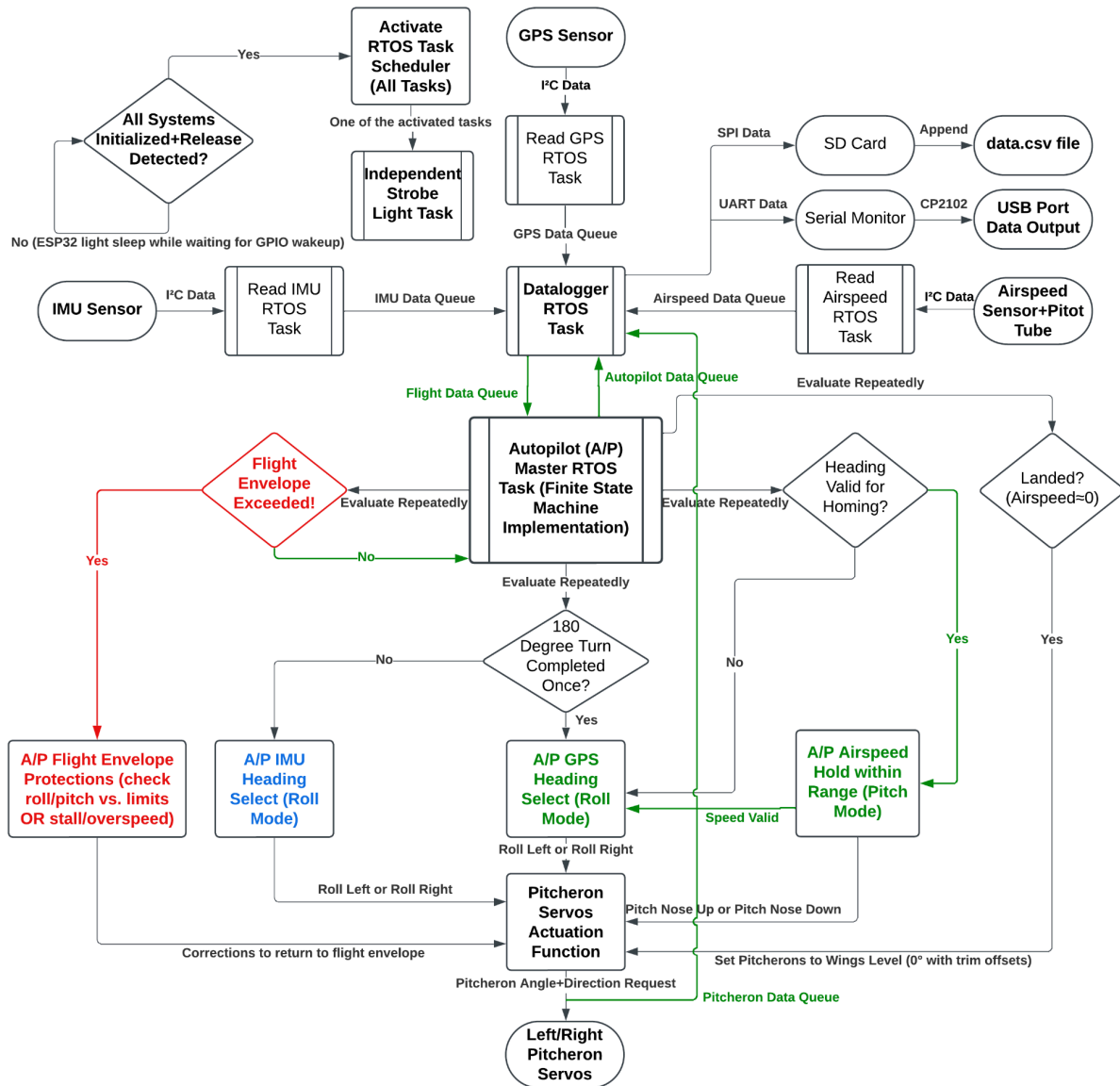


Figure 14. ESP32 FreeRTOS Control Code Flowchart

The system first performs an initialization routine that involves verifying that each of the ESP32's peripherals are functioning correctly. This includes the serial I/O, I2C bus for each sensor (GPS, IMU, and airspeed), SD Card (SPI bus) data logging, strobe LED (initialized off), and both wing actuation servos. Servo actuation tests confirm the servo range of travel and allow

for verification of the servo trim settings necessary to maintain a neutral (level wings) position. After initialization is complete, the microcontroller enters light sleep and suspends both CPUs until a high to low transition is detected on GPIO pin 4 to which the magnetic release detection sensor is attached.

Once release is detected, the following tasks are created and run: GPS reading, IMU reading, airspeed reading, datalogger, autopilot, and strobe light. The strobe light task operates independently and continues indefinitely after release is detected to ensure that the strobe LED continues flashing periodically after the glider has landed. Using FreeRTOS queues, sensor data is transferred to a central datalogger task that manages all data logging to the SD card (in a .csv file) and/or the serial monitor. It also routes all relevant data to the autopilot task. All sensors share an I2C mutex that only allows one sensor to access the I2C bus at a time to prevent conflicts. Additionally, after running once, each task is suspended until all other tasks have also run to prevent bottlenecks. After successful datalogging, all tasks are resumed before the next execution loop (with an optional delay).

3.9. Autopilot

The autopilot master task is implemented as a finite state machine that uses the flight data during each execution loop to determine the current state of the aircraft. These states can be either "performing 180 degree turn," "GPS homing," "speed-pitch control," or "landed." Each state is associated with one of the following autopilot modes: IMU heading select, GPS heading select, or speed descent. In addition, all flight states (except for "landed") are associated with emergency flight envelope protection autopilot modes for roll, pitch, and airspeed. These have the highest priority and always activate whenever a flight envelope exceedance condition is met

(regardless of state). They ensure that corrective action is taken to avoid exceeding a bank angle limit, pitch up/down limits, and stall and overspeed protections that use pitch adjustments to increase or decrease airspeed as required. The autopilot also sends data back to the datalogger task regarding the autopilot/flight modes, current wing angles, and target roll/pitch/bearing. Time since release detection in seconds and a line number are also logged for easier reference during post-flight data analysis.

The IMU heading select and GPS heading select modes are roll control modes that use the current bearing and other flight data to determine the roll angle required to achieve the target bearing. The IMU heading select mode uses the initial yaw angle detected on release and subtracts 180 degrees from it to determine the target yaw angle. The target roll angle is proportionally related to the difference between the current and target yaw angles. The wing servo actuation angles are also proportionally related to the difference between the current and target roll angles by adjusting the duty cycle of the PWM pulses. Once the 180-degree turn is complete and roll angle is near 0, the autopilot's GPS heading select mode is activated. This mode operates very similarly to the IMU heading select mode but uses GPS heading instead of IMU yaw angle. It continuously recalculates the target bearing, θ , with the current and target GPS coordinates using Equation 3. Equation 3 is based on the spherical law of cosines, where ϕ and λ represent latitude and longitude respectively, so that (ϕ_2, λ_2) and (ϕ_1, λ_1) are the current and target GPS coordinates respectively. The arctan function variant used, atan2 , outputs a result that lies in the first two quadrants only (i.e. in the -180 degree to +180-degree range). The target bearing is normalized to the 0-to-360-degree range to match the GPS heading conventions used to determine the current heading. The only edge case is when the current and target coordinates are exactly equal, in which case the target bearing is made equal to the current bearing.

The only regular pitch control autopilot mode is the speed descent mode, which only activates after the glider is on the correct trajectory towards the landing zone and within roll angle limits. It is always overridden by GPS heading select whenever the glider goes off-course, or by flight envelope protections, and has the lowest priority. This mode actuates both wings in the same direction with an angle proportional to the pitch error, which is itself proportional to the airspeed error. For instance, if the target airspeed is higher than the current airspeed, the target pitch will be negative (nose down) and the wings will tilt up proportionally to achieve this pitch angle. The proportionality constants and flight envelope limits are fine-tuned based on flight testing results. Lastly the autopilot's landed state is activated when an airspeed of 0 ft/s is detected and simply returns both wings to their neutral angles (wings level), disabling the autopilot to prevent unwanted actuations from draining the battery after landing. Strobe light blinking and data collection continue until the system is powered off or reset. For testing purposes, stall and overspeed protections (and landed state) are disabled via compiler directives until a reasonable airspeed envelope can be determined through further flight testing.

The complete code listing is included at the end of the appendix (section 8).

3.10 Structural Design and Assembly

The glider fuselage is constructed using two carbon fiber plates, the top being 5.9 in x 1.4 in x 0.079 in and the bottom being 6.3 in x 1.4 in x 0.079 in. The plates are vertically connected and separated by four 0.4375 in threaded hex stand-offs, which are secured with 0.625 in head screws. Carbon fiber was selected as the construction material of the vehicle because of its superior durability and minimal weight, ensuring the vehicle is able to safely reach the designated landing zone and withstand landing impact. The top plate features two water jet-cut

incisions for the release mechanism as well as a 1 in x 0.13 in horizontal incision to allow for the attachment of a magnetic proximity sensor for release confirmation. It also includes the screw holes for both the stands and the servo mounting screws.

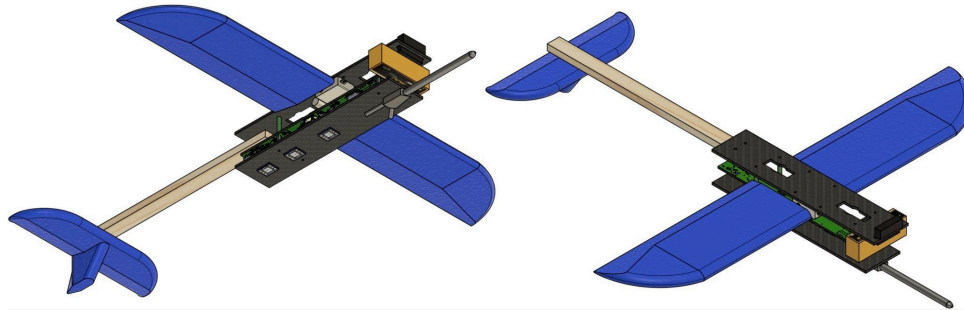


Figure 15. Glider Bottom (left) and top (right) assembly

The pitch and roll of the test vehicle are controlled using two foam wings. The foam wings were taken from a commercial off the shelf glider that served as a donor. A low wing was initially chosen to ensure a quarter inch clearance between the glider and main aircraft wings, but after testing, the wings were found to break less frequently upon landing with a high wing design. In order to compensate for the clearance needed between the wings, the release mechanism was elongated, further explained in section 3.11.

These wings feature a semi-rectangular planform with lifted wingtips, actuated at the quarter cord by the two X08H KST Servos. The wings move symmetrically to control the pitch and asymmetrically to control the roll. The vehicle also includes a foam static horizontal and vertical stabilizer to improve flight stability. The tail was mounted to a quarter by half inch piece of basswood empennage that was epoxied onto the bottom of the top plate of the glider. The length of the empennage was made to mimic the exact length of the distance between the wings and tail

of the commercial foam glider that was used as a donor for the glider's wings and tail. The only major difference is that the vertical stabilizer was installed upside down so that it would not hit the bottom of the fuselage of the main aircraft. This does not affect the overall stability of the plane as the vertical stabilizer stabilizes the glider against forces perpendicular to it.

The pitot tube was glued on the underside of the glider while the LEDs were held on by velcro on the underside of the glider, as this allowed the team to easily change the placement of the LEDs for visibility. The battery was placed at the front of the glider and secured using a zip tie. This was done to ensure that the CG of the glider would remain in front of the quarter chord of the wings, as this is the ideal location for a stable aircraft.

3.11 Release Mechanism

The glider is attached to the main aircraft internally by two 3D-printed ABS plastic servo horns, screwed into two servos attached to the underside of a flat piece of wood. This entire platform and servo release mechanism system, shown in Figure 16, is removable, in compliance with AIAA rules. Each servo horn consists of a 0.4 in diameter, 1.287 in long cylinder with four 0.268 in x 0.3 in x 0.15 in tabs that clamp around the top plate of the glider to secure it in place. When the glider needs to be released, the servos rotate the horns 90 degrees and the cutout on the top plate of the glider aligns with the shape of the servo horn, releasing the vehicle. When the release mechanism is inserted into the plane, the servo horns slot into cutouts on a battery platform placed 1.45 in above the floor of the plane's fuselage, and protrude through holes on the floor of the fuselage, allowing the glider to sit external to the plane. This release mechanism design was selected because the servo horns are lightweight, minimally impede the airflow underneath the main aircraft, and facilitate a secure attachment of the test vehicle to the main aircraft. The

release mechanism was strategically placed near the quarter chord of the main aircraft wing to ensure that once the glider was dropped, the CG of the main aircraft would be minimally affected.

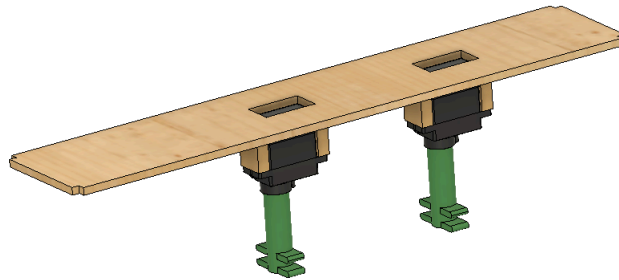


Figure 16. RC Aircraft Mounted Release Mechanism

4. Testing

Testing was an essential part of this project and was designed to validate the glider's systems under live conditions and ensure the competition requirements were met. This section details the performance of the flight data acquisition system, release detection and strobe lights, servo actuation, and overall flight testing across multiple environments. The outcomes of these tests provided insight into the strengths of the design and identified areas for further refinement.

4.1. Flight Data Acquisition

From the successful test flights, the glider's onboard sensors recorded a full set of flight data including GPS coordinates, airspeed, IMU attitude (pitch, roll, yaw), control surface commands, and system state transitions. An example of this data sheet can be seen below in Figure 17.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
1	Line_Num	ESP32_Time_ID0_IMU	UnAcc_x	UnAcc_y	UnAcc_z	Pitch	Roll	Yaw	Gyro_x	Gyro_y	Gyro_z	Magnet_uT_x	Magnet_uT_y	Magnet_uT_z	Grav_x	Grav_y	Grav_z	Quat_re	Quat_i	Quat_j	Quat_k
2	1	0.035612	0	1.794	-0.782	-8.933	-1.307	-2.05	250.689	14.772	-456.688	-76.096	-31.875	5.125	22.375	-3.435	-0.231	-22.556	0.008	0.816	-0.578
3	2	0.104043	0	-2.576	2.666	64.117	-22.057	2.998	245.131	43.308	-197.961	-78.994	-31.875	5.125	22.375	-5.28	0	-22.223	0.125	-0.824	0.532
4	3	0.165547	0	2.576	10.906	-30.015	-21.294	24.084	243.908	271.707	91.427	-15.107	-31.875	5.125	22.375	-12.534	-2.23	-18.942	0.27	-0.795	0.541
5	4	0.231234	0	-3.358	4.626	-74.024	-14.01	24.436	248.045	-21.15	387.53	42.188	-31.875	5.125	22.375	-9.586	-9.356	-18.25	0.241	-0.79	0.564
6	5	0.308849	0	-9.676	6.549	-53.801	6.811	20.128	261.836	84.713	444.602	153.199	-31.875	5.125	22.375	1.64	-7.561	-21.531	0.094	-0.75	0.636
7	6	0.388073	0	-16.532	14.02	12.662	20.061	36.314	281.094	152.08	71.62	78.67	-31.875	5.125	22.375	5.818	-13.021	-18.634	0.067	-0.636	0.688
8	7	0.516703	0	-5.306	6.767	-9.83	16.74	22.144	281.886	-85.384	-130.258	-37.488	-31.875	5.125	22.375	2.153	-3.922	-23.94	0.009	-0.633	0.736
9	8	0.632889	0	3.934	4.242	-12.726	1.356	10.794	275.963	111.906	-98.925	8.729	-31.875	5.125	22.375	-1.948	-2.666	-24.914	0.054	-0.667	0.739
10	9	0.712922	0	-0.051	12.444	-10.855	-0.37	16.409	276.757	348.363	29.319	-11.191	-31.875	5.125	22.375	-1.692	-13.149	-22.376	0.097	-0.657	0.74
11	10	0.921784	0	-0.115	20.941	-6.267	1.494	55.062	277.213	365.932	5.483	36.034	-31.875	5.125	22.375	-2.025	-21.966	-14.636	0.297	-0.591	0.661
12	11	0.99596	0	8.728	24.106	-46.021	-1.499	79.213	280.173	168.754	300.915	181.623	-31.875	5.125	22.375	-12.252	-23.94	3.153	0.417	-0.488	0.596
13	12	1.076674	0	16.827	19.8	-37.012	-40.853	105.587	323.146	29.543	225.938	269.581	-31.875	5.125	22.375	-19.839	-18.173	-2.153	0.436	0.085	0.625
14	13	1.131818	0	21.962	13.995	-31.565	-65.054	92.074	345.91	239.031	138.428	289.053	-31.875	5.125	22.375	-24.376	-10.099	-4.844	0.445	0.312	0.628
15	14	1.228677	0	24.119	9.714	-32.45	-83.706	33.866	66.747	377.123	115.039	258.614	-31.875	5.125	22.375	-25.375	-5.947	-5.588	0.414	0.554	0.488
16	15	1.306703	0	22.761	3.178	-32.475	-76.817	-27.885	148.487	503.129	134.623	269.022	-31.875	5.125	22.375	-24.888	-1.179	-8.535	0.341	0.694	0.345
17	16	1.370508	0	14.443	-1.986	-40.985	-50.997	-16.963	197.541	491.378	164.054	307.629	-31.875	5.125	22.375	-22.043	-0.256	-13.662	0.067	0.892	-0.073
18	17	1.435793	0	10.778	2.576	-25.926	-34.1	3.733	228.276	486.119	208.145	316.917	-31.875	5.125	22.375	-19.685	-1.897	-16.532	0.148	-0.868	0.399
19	18	1.626789	0	7.113	4.665	-14.687	-26.319	16.423	248.619	350.041	219.671	335.27	-31.875	5.125	22.375	-14.994	-5.152	-20.121	0.242	-0.778	0.57
20	19	1.719766	0	1.025	-5.37	-8.535	-28.293	7.249	344.073	158.011	172.111	385.18	-31.875	5.125	22.375	-10.304	4.255	-23.094	0.25	-0.119	0.961
21	20	1.769533	0	1.692	-5.87	-6.344	-29.621	-9.805	23.04	98.813	-56.512	252.683	-31.875	5.125	22.375	-8.484	7.869	-22.992	0.266	0.171	0.948
22	21	1.839697	0	3.063	-3.588	-7.536	-30.164	-9.861	33.626	230.078	-91.539	145.925	-31.875	5.125	22.375	-8.689	5.229	-23.76	0.272	0.257	0.927
23	22	1.913894	0	4.255	-0.09	-9.573	-34.801	0.986	46.094	317.029	-115.823	96.239	-31.875	5.125	22.375	-10.535	0.461	-23.684	0.276	0.374	0.878
24	23	2.072701	0	7.869	8.587	-11.944	-40.859	13.748	50.802	366.156	-104.296	73.298	-31.875	5.125	22.375	-14.533	-8.971	-19.685	0.263	0.437	0.823
25	24	2.129506	0	11.227	16.673	-33.552	-49.271	40.507	53.81	292.634	63.339	84.601	-31.875	5.125	22.375	-18.762	-17.301	-5.998	0.206	0.515	0.695
26	25	2.190497	0	14.328	17.442	-32.962	-58.546	76.479	72.719	336.725	155.325	95.232	-31.875	5.125	22.375	-19.326	-17.276	-3.665	0.011	-0.65	-0.372
27	26	2.307696	0	17.122	16.737	-47.944	-61.871	74.395	95.993	194.157	134.623	147.604	-31.875	5.125	22.375	-20.89	-15.481	-2.538	0.111	-0.716	-0.226
28	27	2.394841	0	21.761	9.637	-54.736	-65.743	69.929	110.253	331.689	86.727	236.233	-31.875	5.125	22.375	-24.658	-7.382	-2.409	0.141	-0.743	-0.138
29	28	2.475522	0	23.363	3.255	-64.233	-79.825	18.427	204.23	522.153	60.989	273.05	-31.875	5.125	22.375	-25.247	0.384	-3.717	0.253	-0.719	0.259
30	29	2.552746	0	21.364	-5.088	-70.628	-69.193	-28.313	300.492	608.991	55.953	298.565	-31.875	5.125	22.375	-23.376	4.204	-8.382	0.378	-0.517	0.624
31	30	2.705542	0	9.215	3.319	-42.933	-54.343	-9.68	342.398	821.613	202.214	276.295	-31.875	5.125	22.375	-20.018	-2.691	-14.969	0.438	-0.174	0.87
32	31	2.792784	0	2.781	16.584	-45.432	-41.223	26.678	19.751	558.97	386.187	342.32	-31.875	5.125	22.375	-16.909	-15.456	-12.585	0.3	0.236	0.883
33	32	2.872754	0	4.088	14.431	-44.766	-41.443	60.556	112.218	421.325	332.584	358.434	-31.875	5.125	22.375	-18.788	-14.815	-10.996	0.221	-0.77	-0.302
34	33	3.066683	0	8.817	9.971	-34.244	-53.989	49.701	175.476	197.628	223.028	393.797	-31.875	5.125	22.375	-20.633	-8.971	-14.226	0.358	-0.815	0.159

Figure 17. Example of impartial flight test data collection.

4.2. Release Detection and Strobe Lights

The release detection system and strobe lights consistently operated when the glider was signaled to separate from the plane. Upon successful release, the magnetic sensor immediately detected the event and triggered the onboard ESP32 to initiate the flight sequence, including strobe light

activation. Although mechanical separation was not always successful because of aerodynamic forces and minor interference at the mounting clips, the electronic detection operated independently and correctly regardless of physical release. This was confirmed through both real-time ground observation and post-flight data logs, where timestamps for release detection and strobe activation aligned with the expected moments of glider deployment. The brightness of the strobe lights was not always sufficient for visual confirmation from the ground, but they stayed on after landing, which was essential for competition scoring.

4.3. Servo Actuation (Ground Test)

The team performed several ground tests of the servo actuation that confirmed the usability of the algorithm, and the correct orientation of the pitcherons. The team would perform a manual 180 degree turn of the glider, and then a tilt and turn test to ensure the control surfaces rotated as expected. These ground tests proved the servos would as expected. Additionally, all flight envelope protections were triggered correctly. If the glider rolled too far in any direction, the control surfaces would undergo maximum deflection in the opposite direction to return the glider to stable flight, and the same applies for pitch. Stall and overspeed protections were disabled for obvious reasons during ground testing.

4.4 Flight Testing

Flight testing took place in several locations. These included an airfield in Tucson, Arizona, and a model flying club (AMA) airfield in South Bend. The flight test in Tucson suggested that the glider was being overcontrolled, so the maximum pitcheron angles were reduced from 14 degrees to 9 degrees, with proportionality constants adjusted accordingly. But the glide itself appeared quite reasonable. The second flight test appeared worse because the glider entered a flat spin almost immediately after release from the main aircraft (likely due to turbulent air from the propeller). Without a movable rudder, spin recovery was impossible in that scenario. Given that the glider still performed a 180 degree turn and landed within 100 feet of the target coordinates, meeting the main flight requirements. Moreover, release detection worked perfectly and the

strobe lights were visible immediately upon release and continued flashing after landing, which met the remaining miscellaneous requirements and the glider's final weight was 0.33 lb, which is under the 0.55 lb limit, so all mission requirements were met successfully.

5. Instruction Manual

This section details the required steps to drop the glider during a flight. The autonomous glider is relatively straightforward to use, especially since it does not need to be manually controlled after a drop. First, plug in the JST connector from the 7.4 V LiPo battery into the circuit board (the plug and socket are slotted which prevents reversing the terminals). On the circuit board, a blue LED will turn on, and the glider's servo motors will then run through an initialization test to ensure they are properly functioning. Once this test is over, the blue LED will turn off, and the wings will pitch down at a 15 degree angle. This means the initialization stage is completed, and the glider is ready to be mounted to the aircraft.

Next, the user should ensure the aircraft's release clips are oriented in parallel to the fuselage of the aircraft. This can be done by flipping the release switch on the aircraft's RC controller. Then, slide the glider into the clips, and ensure the black magnet at the front of the glider is connected to the other black magnet on the underside of the aircraft. On the glider, the carbon fiber mounting plate will have two complementary slots which receive the release clips from the main aircraft. You must hold the glider in place while this process is being completed. After these clips are slotted into the glider's mounting plate, flip the release switch once again on the RC controller. Now, release the glider from your grip. At this point, the blue LED should turn on once again for 10 seconds. After these 10 seconds finish, the blue LED will turn off, and the glider is prepared for flight.

You may now take-off using the RC aircraft. Once RC aircraft is up to the desired altitude, you may flip the release switch on the RC controller. This will release the glider from the aircraft. At this point, the glider will recognize its detachment from the plane, turn on the LED lights,

complete its 180 degree turn, and head towards the programmed GPS coordinates. The glider will automatically log all the flight data to the onboard microSD card. To repeat this process, unplug the battery's JST connector from the circuit board, and plug it back in. This will reset the process and allow you to prepare the glider for another flight.

6. Potential Design Improvements

There are several areas of this project that could be improved. These sections are outlined below, along with explanations of where enhancements could be made.

6.1. Aerodynamic Design and Flight Stability

While the glider design saw significant improvements over the course of the semester, there is still room to enhance the aerodynamic performance and overall stability of the glider. Although flight data and visual characteristics provided clear evidence that lift was being generated after release, additional measurements and testing could further optimize the design and improve post-drop stability. Moreover, using an actuated rudder or at least elevators and ailerons instead of pitcherons would have greatly improved flight stability and control because pitcherons can correct roll or pitch, but not both at the same time (without a very advanced algorithm).

6.2. Release Mechanism

Another area for improvement is the reliability of the release mechanism. While the release mechanism often activated correctly, the glider would occasionally fail to separate from the plane. To address this, the team trimmed down the clips that connected the glider to the plane to prevent unintended entanglements. This solution was effective but still leaves room for a redesign in the release system design.

6.3. Homing (Proportional Integral Derivative Controller) Algorithm

The final major area for improvement lies within the proportional integral derivative controller homing algorithm. Review of the flight data showed that the glider struggled to achieve sufficient stability before adjusting its wing angles toward the GPS target. With additional testing, the team would aim to refine the algorithm to achieve more stable flight and more

accurate GPS-based guidance. Due to insufficient testing and data, only a proportional control algorithm was implemented. But with more testing and/or aerodynamics simulations, we could determine integral and derivative constants and fine tune them to achieve more precise control over flight control surfaces that reduces induced oscillations by avoiding overcorrections. An even further improvement would account for flight control surface effects at different airspeeds to optimize actuation and flight response.

7. Conclusion

This project successfully demonstrated the design, implementation, and testing of an autonomous glider system capable of mid-air release, strobe light activation, and guided descent toward a predefined landing zone. The team effectively used an ESP32 microcontroller and several peripherals to manage flight control surfaces, data logging, and flight state transitions within a FreeRTOS software architecture. This resulted in a glider that was capable of flying and operating without continuous user input.

The glider consistently displayed the maneuvers of a 180-degree turn and GPS-based heading adjustments during flight. Ground tests verified servo reliability and control logic. Flight tests confirmed the functionality of the systems including release detection mechanism, LED strobe light activation, and data acquisition. Although we experienced minor issues with the release mechanism, which inhibited the glider's scoring at competition, this issue was subsequently addressed in later tests and resolved for a reliable result.

This project was an exploration into several important areas of research and showed the intersection of embedded systems and autonomous intelligence in an aerospace application. The team concluded that while the honing algorithm, release mechanism, and aerodynamic design still require refinement, the project was an overall success. Overall, the objectives laid out in the problem statement were achieved with the team having developed a lightweight glider capable of autonomous flight.

8. Appendix

This section contains all the appendices of this report. This includes the electrical schematic, the PCB layout, and the source code.

8.1. Electrical Schematic

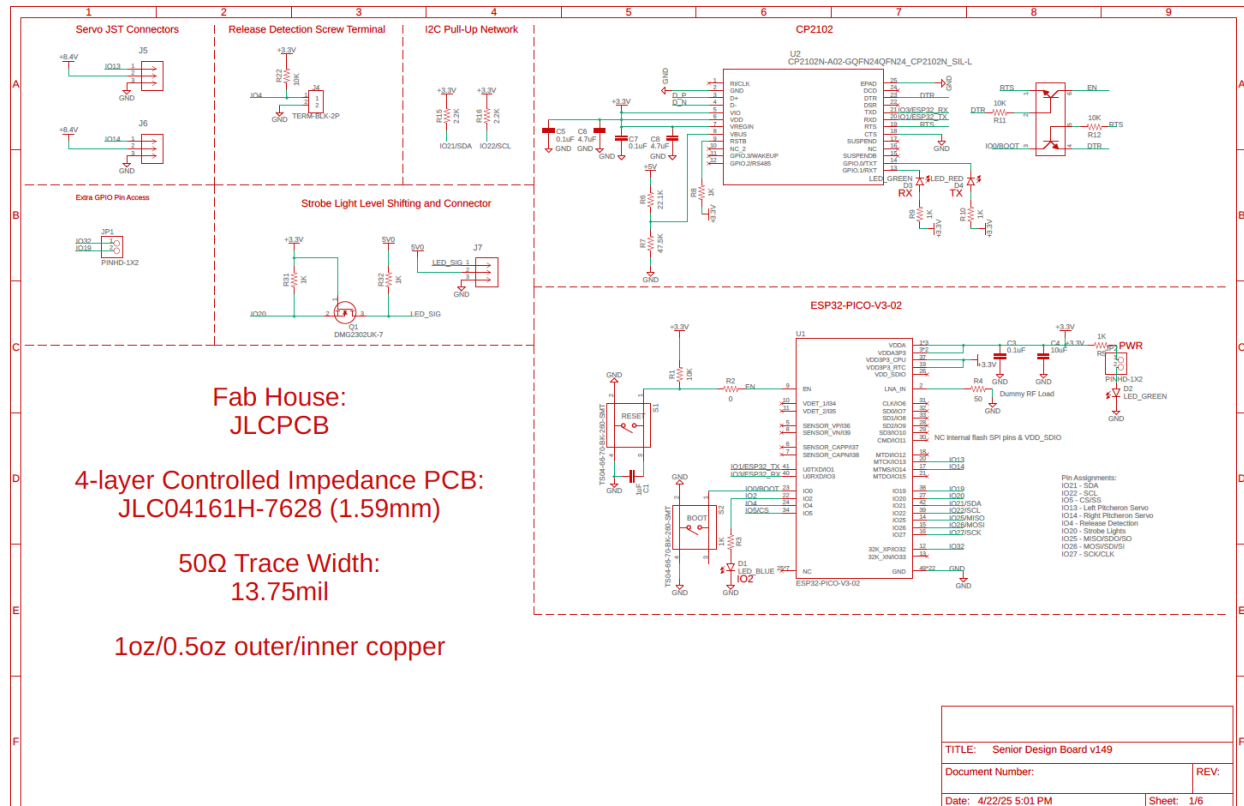


Figure 18. Electrical Schematic Page 1 of 6

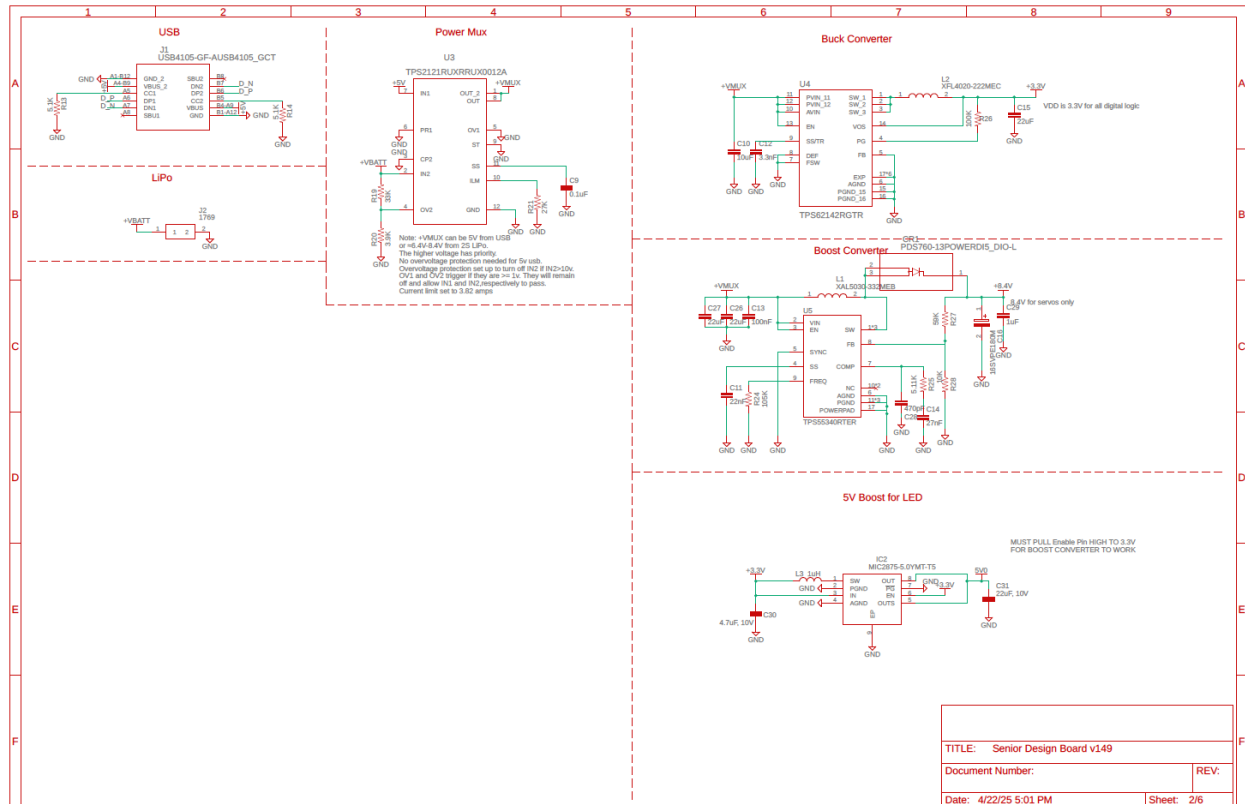


Figure 19. Electrical Schematic Page 2 of 6

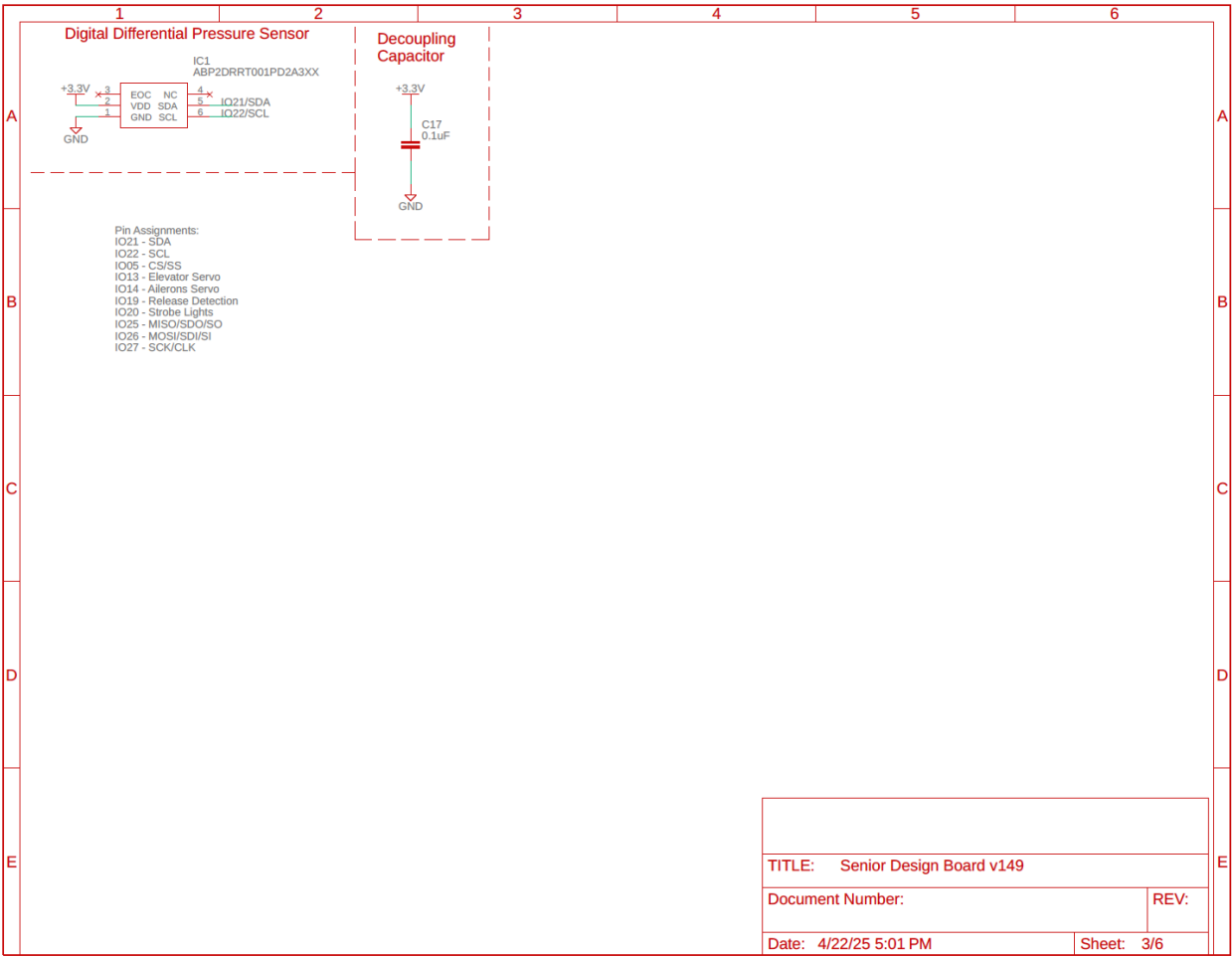


Figure 20. Electrical Schematic Page 3 of 6



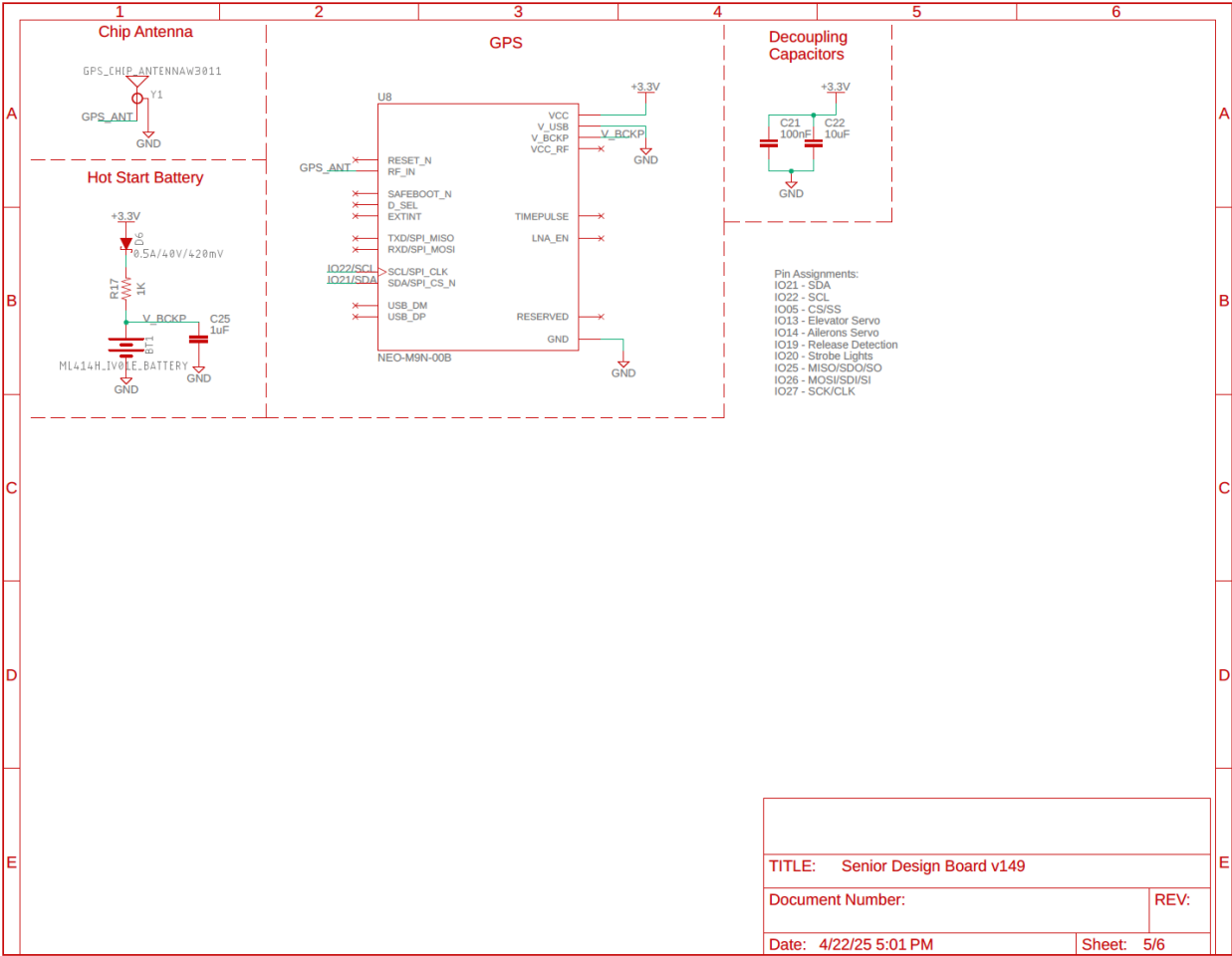


Figure 22. Electrical Schematic Page 5 of 6

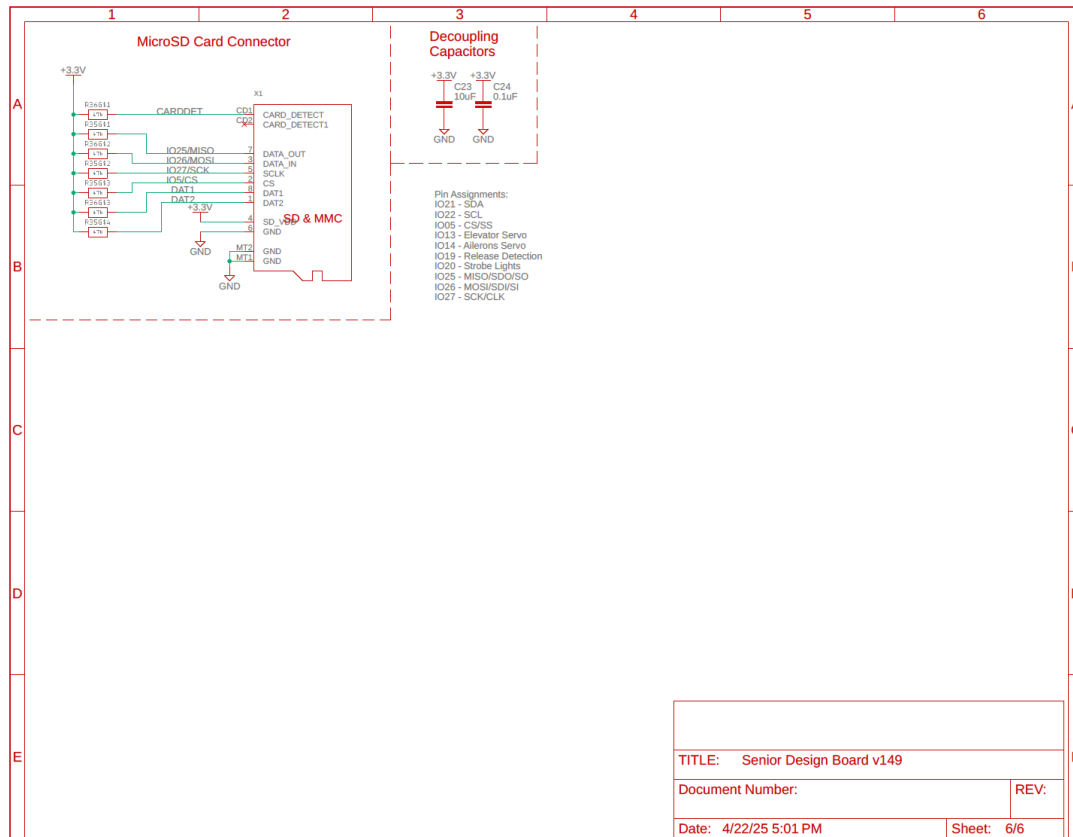


Figure 23. Electrical Schematic Page 6 of 6

8.2. PCB Layout

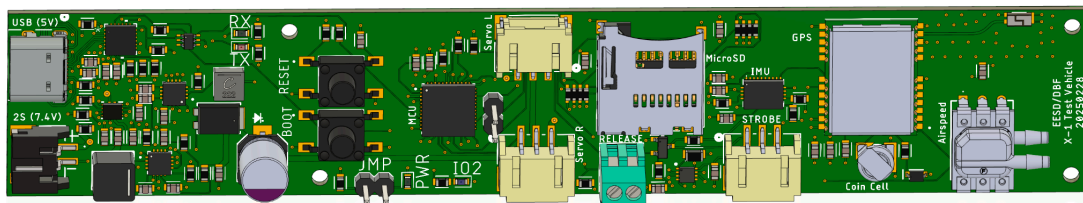


Figure 24. Circuit Board 3D View (1)



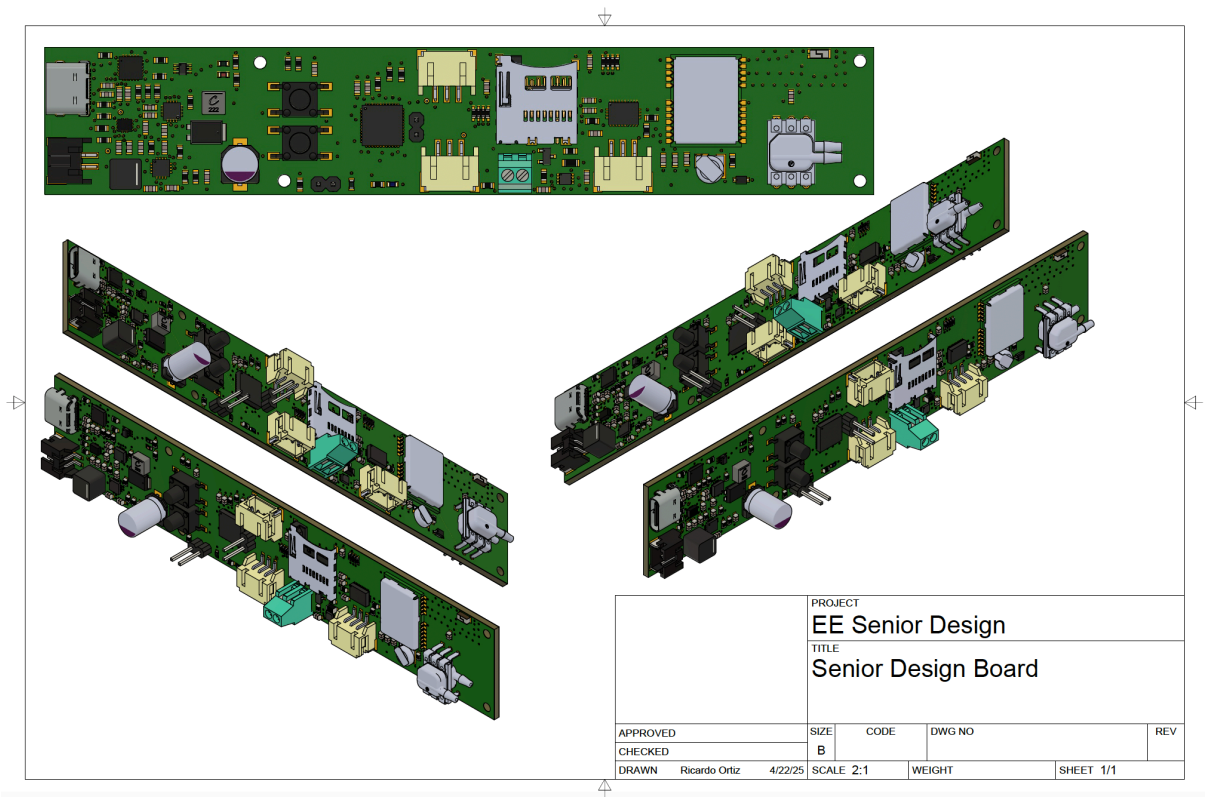


Figure 27. Circuit Board CAD Drawing (Multiple Views)

8.3. Source Code Listing

https://github.com/rortiz4/DBF2425/tree/main/Code/DBF_X1_Glider

```
=====
// File: platformio.ini
=====
```

```
; PlatformIO Project Configuration File
;
; Build options: build flags, source filter
; Upload options: custom upload port, speed and extra flags
; Library options: dependencies, extra library storages
; Advanced options: extra scripting
;
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html
```

```
[env:esp32dev]
platform = espressif32
board = esp32dev
framework = arduino
lib_deps =
adafruit/Adafruit BNO08x@^1.2.5
sparkfun/SparkFun u-blox GNSS Arduino Library@^2.2.27
stevemarple/MicroNMEA@^2.0.6
https://github.com/madhephaestus/ESP32Servo
adafruit/Adafruit NeoPixel@^1.12.4
```

```
=====
// File: autopilot.h
=====
```

```
#ifndef AUTOPILOT_H
#define AUTOPILOT_H
```

```
// This struct will be used by datalogger
```

```
struct Autopilot_Data {
    unsigned int sensor_id; // 4
    const char* flight_phase;
    const char* ap_mode;
    float ap_target_bearing;
    float ap_target_roll;
    float ap_target_pitch;
};

enum AP_Modes {
    AP_OFF, // 0 (Almost never used except on ground)
    AP_HDG_SEL_IMU, // 1
    AP_HDG_SEL_GPS, // 2...
    AP_SPD_TRIM,
    AP_PITCH_FIXED,
    AP_ROLL_FIXED,
    AP_ENVELOPE_PROT, // Flight envelope protection modes below (AP_ENVELOPE_PROT
specifically not logged. See below for logged modes.)
    AP_PROT_ROLL_MIN,
    AP_PROT_ROLL_MAX,
    AP_PROT_PITCH_MIN,
    AP_PROT_PITCH_MAX,
    AP_PROT_STALL,
    AP_PROT_OVERSPEED
};

enum Flight_Phases {
    // Must always alternate between a Roll Mode and a Pitch Mode.
    U_TURN_HDG, // Roll Mode: 180 degree turn after release using IMU only
    GPS_HOMING, // TBD Roll Mode: Continuous update of target bearing + bearing correction
from current GPS coordinates + COG/heading. Target coordinates hard-coded. Same logic as
U_TURN_HDG flight phase after that.
    SPD_DESCENT, // Pitch Mode: Primary mode after U-Turn. Will alternate with
U_Turn_HDG/GPS_HOMING as needed (common to both roll modes).
    LANDED // Used to return pitcherons to neutral position and turn off autopilot after landing.
};
```

```

void Autopilot_MASTER(void* pvParameters); // Calls on each of the below functions to
perform manoeuvres as required. Maintains flight envelope protections too.
bool Autopilot_HDG_SEL_IMU(float roll, float yaw, float bearing_change, Autopilot_Data&
AP_log_data); // Current bearing from IMU "Yaw" Angle. Servo actuation done to turn by
bearing_change (e.g. 180degree turn)
bool Autopilot_SPD_TRIM(float airspeed, float pitch, float target_airspeed, Autopilot_Data&
AP_log_data); // Controls pitch to maintain airspeed within +/-max_dev of target.
bool Autopilot_HDG_SEL_GPS(bool act_true, float roll, float current_heading, float current_lat,
float current_long, float target_lat, float target_long, Autopilot_Data& AP_log_data); // Current
bearing from GPS+current+target coordinates (used to calculate target bearing for turn). Used
after 180 degree turn for homing.
bool Autopilot_FLT_ENVELOPE_PROT(float roll, float pitch, float airspeed, Autopilot_Data&
AP_log_data);

// Unused in AP_MASTER
bool Autopilot_ROLL_FIXED(float roll, float target_roll, Autopilot_Data& AP_log_data); //
Will probably go unused, but included for completeness (direct roll control by angle.
bool Autopilot_PITCH_FIXED(float pitch, float target_pitch, Autopilot_Data& AP_log_data); //
Will probably go unused, but included for completeness (direct pitch control by angle.
Dangerous due to high stall potential! Good for levelling off.)

// Note: max_dev is allowed tolerance in each case for bearing/airspeed/pitch from target

#endif

=====
// File: datalogger.h
=====

#ifndef DATALOGGER_H
#define DATALOGGER_H

extern float current_time;
extern bool serial_log;
extern bool SD_log;

```

```
// This struct will be used in Autopilot
struct Flight_Data {
    float time; // will probably not be used
    float pitch;
    float roll;
    float yaw;
    float latitude;
    float longitude;
    float heading;
    float airspeed;
    float gnd_speed; // could be used for wind corrections later
};

void init_SD(bool serial_log, bool SD_log);
void log_data(void* pvParameters);

#endif
```

```
=====
// File: pin_map.h
=====
```

```
#ifndef PIN_MAP_H
#define PIN_MAP_H
```

```
// For datalogger (SPI pins)
#define SD_CS 5
#define SD_MISO 25
#define SD_MOSI 26
#define SD_SCK 27
```

```
// For all sensors (I2C pins)
#define SDA_PIN 21
#define SCL_PIN 22
```

```

// Outputs (LEDs+Servos)
#define BUILTIN_LED_PIN 2

#define RELEASE_DET_PIN 4
#define STROBE_LED_PIN 20

// Separate left and right servo pin defs
#define SERVO_L_PIN 14
#define SERVO_R_PIN 13

#endif

=====
// File: pitcheron_servos.h
=====

#ifndef PITCHERON_SERVOS_H
#define PITCHERON_SERVOS_H

//
https://cdn.shopify.com/s/files/1/0570/1766/3541/files/X08H\_V6.0\_Technical\_Specification.pdf?v=1700472376
//
https://kstservos.com/collections/glider-wing-servos/products/x08h-plus-horizontal-lug-servo-5-3kg-cm-0-09s-9-5g-8mm

#define MIN_SERVO_ANGLE -60 // deg (unused except for internal angle2us mapping or manual trimming override because dangerous!)
#define MAX_SERVO_ANGLE 60 // deg (unused except for internal angle2us mapping or manual trimming override because dangerous!)
#define SERVO_MIN_ALLOWED -9
#define SERVO_MAX_ALLOWED 9

enum Pitcheron_Actions {
    WINGS_LEVEL, // 0 (could also do WINGS_LEVEL = 0 for custom assignment, and so on.)
    ROLL_LEFT, // 1

```

```

    ROLL_RIGHT, // 2
    PITCH_NOSE_UP, // 3
    PITCH_NOSE_DOWN, // 4
    MAINTAIN_ANGLE // 5
};

struct Pitcheron_Data {
    unsigned int sensor_id;
    unsigned int angle_target; // absolute value only
    const char* action_target; // e.g. WINGS_LEVEL, ROLL_RIGHT, etc.
    int raw_angle_l; // Angle including trim adjustments and CG/CW corrections (raw)
    int raw_angle_r; // Angle including trim adjustments and CG/CW corrections (raw)
    // All angles in degrees.
};

// Functions just for initial testing/trimming (angle specified is directly used for actuation as-is)
void actuate_servo_l(int angle);
void actuate_servo_r(int angle);

void init_servos(bool actuation_test); // Use either init_servos or init_servos_trim in a program,
but NOT BOTH.

void init_servos_trim(void); // Do not use, except for in another program (not main())
exclusively to initialize for trimming individual servos.

void actuate_pitcherons(unsigned int angle, enum Pitcheron_Actions act_type_direction); //
Specify PITCH or ROLL with direction for actuation type + direction (both servos use same
angle but potentially different trim offsets)
#endif

=====
// File: queues.h
=====

#ifndef QUEUES_H
#define QUEUES_H
#include <Arduino.h>

```

```
// This just lets the compiler know that this queue is declared elsewhere so other files see it when
// including this header
extern QueueHandle_t IMU_Queue;
extern QueueHandle_t Airspeed_Queue;
extern QueueHandle_t GPS_Queue;
extern QueueHandle_t Autopilot_Queue;
extern QueueHandle_t Pitcheron_Queue;

extern QueueHandle_t Flight_Data_Queue;

void init_queues();

#endif

=====
// File: semaphores.h
=====
#ifndef SEMAPHORES_H
#define SEMAPHORES_H
#include <Arduino.h>

extern SemaphoreHandle_t I2C_MUTEX;

// extern SemaphoreHandle_t imu_done;
// extern SemaphoreHandle_t airspeed_done;
// extern SemaphoreHandle_t gps_done;

void init_semaphores();

#endif
```

```
=====
// File: sensors.h
=====
```

```
#ifndef SENSORS_H
#define SENSORS_H
```

```
// Defining containers for data
```

```
struct IMU_Data {
    unsigned int sensor_id;
    float lin_accel[3]; // x,y,z
    float euler[3];
    float gyro[3]; // x,y,z
    float magnetic[3]; // x,y,z
    float gravity[3]; // x,y,z
    float rotation[4]; // Quaternion - real, i, j, k
};
```

```
struct Airspeed_Data {
    unsigned int sensor_id;
    float diff_pressure;
    float airspeed[2]; // raw, corrected
    float temperature;
};
```

```
struct GPS_Data {
    unsigned int sensor_id;
    float latitude;
    float longitude;
    float gnd_speed; // knots
    float altitude;
    float heading;
    uint8_t hours;
    uint8_t minutes;
    uint8_t seconds;
    uint8_t hundredths;
```



```
uint8_t satellites;
};

void init_low_level_hw();
bool init_bno085(); // Unused in main
bool init_abp2(); // Unused in main
void init_all_sensors();
void read_bno085(void* pvParameters);
void read_abp2(void* pvParameters);
void read_gps(void* pvParameters);

#endif
```

```
=====
// File: strobe.h
=====
```

```
#ifndef STROBE_H
#define STROBE_H

void init_strobe(void);
void blink_strobe(void* pvParameters);

#endif
```

```
=====
// File: tasks.h
=====
```

```
#ifndef TASKS_H
#define TASKS_H
#include <Arduino.h>

extern TaskHandle_t read_imu_task;
extern TaskHandle_t read_pitot_task;
```

```
extern TaskHandle_t read_gps_task;
extern TaskHandle_t log_data_task; // Currently unused handle
extern TaskHandle_t autopilot_task;
extern TaskHandle_t strobe_task;
```

```
void init_tasks();
```

```
#endif
```

```
=====
// File: trim_servos.h
=====
```

```
#ifndef TRIM_SERVOS_H
#define TRIM_SERVOS_H
#include <Arduino.h>
```

```
void trim_servos(void);
```

```
#endif
```

```
=====
// File: autopilot.cpp
=====
```

```
#include "autopilot.h"
#include "datalogger.h"
#include "queues.h"
#include "pitcheron_servos.h"
```

```
#define AP_ENABLE true // Set true to enable Autopilot, false to disable.
#define LANDED_DISABLED true // Disables LANDED state
#define ROLL_PROT_EN true
#define PITCH_PROT_EN true
#define STALL_PROT_EN false
#define PITCH_SPEED_CONTROL true
```

```
#define OVSPD_PROT_EN false

// Note: Convention used by autopilot: + means right/up, - means left/down. ALL ANGLES IN
// DEGREES AND SPEEDS IN ft/s.
// Flight Envelope Limits
#define MAX_PITCHERON_ANGLE SERVO_MAX_ALLOWED // Defined in
pitcheron_servos.h
#define STALL_SPEED 30
#define OVERSPEED 60
#define ROLL_LIM_MIN -30
#define ROLL_LIM_MAX 30
#define PITCH_LIM_MIN -18
#define PITCH_LIM_MAX 18

// Autopilot Settings and Control Limits
#define U_TURN_BEARING_CHANGE 180 // deg: From DBF 2024 Competition Rules
#define RIGHT_TURN_BIAS true // Takes priority over LEFT_TURN_BIAS if both are true
#define LEFT_TURN_BIAS false
#define TURN_BIAS 20 // Will turn right for corrections up to -160 during U Turn
#define BULLSEYE_LATITUDE 32.2653//32.2653 //32.1201// //41.5194 // degN: From DBF
2024 Competition Rules
#define BULLSEYE_LONGITUDE -111.2736//-111.2736 //-110.7630// //-86.2400 // degW:
From DBF 2024 Competition Rules
#define HDG_TARGET_DEVIATION_LOW -5
#define HDG_TARGET_DEVIATION_HIGH 5
#define ROLL_TARGET_DEVIATION_LOW -5
#define ROLL_TARGET_DEVIATION_HIGH 5
#define SPD_TARGET 45 // ft/s
#define SPD_TARGET_DEVIATION_LOW -15 // -10 means STALL_SPEED = 40 ft/s with 50
ft/s target
#define SPD_TARGET_DEVIATION_HIGH 15
#define PITCH_TARGET_DEVIATION_LOW -3
#define PITCH_TARGET_DEVIATION_HIGH 3

// PID Proportionality Constants (leave margin for min/max to avoid exceeding flight envelope
limits)
```

```

#define Kp_ROLL_BEARING_CORR 1// Roll proportional to amount of turning required (yaw
change)
#define Kp_PITCH_SPD_CORR -1// MUST BE NEGATIVE!!! Pitch proportional to amount of
speed change required (current speed error from target)
#define Kp_SERVO_ANGLE_ROLL 0.3// Servo Angle proportional to error in roll from target
#define Kp_SERVO_ANGLE_PITCH 0.5// Servo Angle proportional to error in pitch from
target

// Utility Functions for bearings
// Wrap angle to range [0, 360)
float wrap_angle(float angle) {
    float wrapped = fmod(angle, 360.0); // Calculates remainder of angle/360 (wraps angle)
    return wrapped < 0 ? wrapped + 360 : wrapped; // For negative angles
}
// To add bearings: wrap_angle(angle_1+angle_2)
// To subtract bearings: wrap_angle(angle_1-angle_2)
// This function gives the shortest necessary correction for bearings
// e.g. current_bearing = 10deg, target_bearing = 350deg, returns -20deg
// Negative angles mean left turn needed. Positive angles mean right turn needed.
float signed_bearing_correction(float current_bearing, float target_bearing) {
    float brg_corr = wrap_angle(target_bearing - current_bearing);
    if (brg_corr > 180.0) {
        brg_corr -= 360.0;
    }
    return brg_corr;
}
// Function to calculate the bearing from current and target GPS coordinates
float calculate_bearing(float current_lat, float current_long, float target_lat, float target_long,
float current_bearing) {
    float target_bearing = current_bearing; // Just setting default condition for variable
initialization (modified later).
    // Edge case: Current latitude/longitude = Target latitude/longitude (prevents undefined target
bearing)
    if (current_lat == target_lat && current_long == target_long) target_bearing =
current_bearing;

```

```
else {
    // Convert latitude and longitude from degrees to radians
    current_lat = current_lat * M_PI / 180.0f;
    current_long = current_long * M_PI / 180.0f;
    target_lat = target_lat * M_PI / 180.0f;
    target_long = target_long * M_PI / 180.0f;

    // Calculate the difference in longitude
    float delta_long = target_long - current_long;

    // Calculate the components of the formula
    float x = sinf(delta_long) * cosf(target_lat);
    float y = cosf(current_lat) * sinf(target_lat) - sinf(current_lat) * cosf(target_lat) *
    cosf(delta_long);

    // Calculate the initial bearing
    float initial_bearing = atan2f(x, y);

    // Convert bearing from radians to degrees and normalize to 0-360
    target_bearing = fmodf((initial_bearing * 180.0f / M_PI) + 360.0f, 360.0f);
}

return target_bearing;
}

// Main Autopilot functions
void Autopilot_MASTER(void* pvParameters) {
    Flight_Data sensor_data;
    Autopilot_Data AP_log_data;
    AP_log_data.sensor_id = 3;
    int ap_flight_phase = U_TURN_HDG;
    AP_log_data.flight_phase = "U_TURN_HDG";
    AP_log_data.ap_mode = "AP_OFF";
    bool started = false;
```

```

bool u_turn_done = false;
AP_log_data.ap_target_bearing = 0; // just for initialization
AP_log_data.ap_target_roll = 0; // just for initialization
AP_log_data.ap_target_pitch = 0; // just for initialization
if (AP_ENABLE) Serial.println("Autopilot ON!");
else Serial.println("Autopilot OFF. Maintaining Pitcheron Neutral Position.");
while(true) {
    //Serial.println("Autopilot Task");
    xQueueReceive(Flight_Data_Queue, &sensor_data, portMAX_DELAY);
    // First use the received data to identify the current phase of flight and AP Mode. But we
    start with U-Turn immediately
    // Landed
    if (!AP_ENABLE) {
        actuate_pitcherons(0, MAINTAIN_ANGLE); // Do nothing if AP has been disabled
        through flag.
    }
    else if (sensor_data.airspeed == 0 && !LANDED_DISABLED) {
        actuate_pitcherons(0, WINGS_LEVEL);
        AP_log_data.flight_phase = "LANDED";
        AP_log_data.ap_mode = "AP_OFF";
    }
    else if (!started) {
        // Always start in HDG_SEL_IMU mode to get correct 180 degree turn bearing
        Autopilot_HDG_SEL_IMU(sensor_data.roll, sensor_data.yaw,
        U_TURN_BEARING_CHANGE, AP_log_data);
        started = true;
    }
    else if ((ap_flight_phase == U_TURN_HDG) && !u_turn_done) {
        AP_log_data.flight_phase = "U_TURN_HDG";
        // Always verify flight envelope first
        if (Autopilot_FLT_ENVELOPE_PROT(sensor_data.roll, sensor_data.pitch,
        sensor_data.airspeed, AP_log_data)) {
            if (Autopilot_HDG_SEL_IMU(sensor_data.roll, sensor_data.yaw,
            U_TURN_BEARING_CHANGE, AP_log_data)) {
                u_turn_done = true;
                ap_flight_phase = GPS_HOMING;
            }
        }
    }
}

```

```

    }
    else ap_flight_phase = U_TURN_HDG;
  }
}
else if (ap_flight_phase == GPS_HOMING) {
  AP_log_data.flight_phase = "GPS_HOMING";
  // Always verify flight envelope first
  if (Autopilot_FLT_ENVELOPE_PROT(sensor_data.roll, sensor_data.pitch,
sensor_data.airspeed, AP_log_data)) {
    if (Autopilot_HDG_SEL_GPS(true, sensor_data.roll, sensor_data.heading,
sensor_data.latitude, sensor_data.longitude, BULLSEYE_LATITUDE,
BULLSEYE_LONGITUDE, AP_log_data) && PITCH_SPEED_CONTROL) {
      ap_flight_phase = SPD_DESCENT;
    }
    else ap_flight_phase = GPS_HOMING;
  }
}
else if (ap_flight_phase == SPD_DESCENT) {
  AP_log_data.flight_phase = "SPD_DESCENT";
  // Always verify flight envelope first
  if (Autopilot_FLT_ENVELOPE_PROT(sensor_data.roll, sensor_data.pitch,
sensor_data.airspeed, AP_log_data)) {
    if (Autopilot_HDG_SEL_GPS(false, sensor_data.roll, sensor_data.heading,
sensor_data.latitude, sensor_data.longitude, BULLSEYE_LATITUDE,
BULLSEYE_LONGITUDE, AP_log_data)) {
      if (Autopilot_SPD_TRIM(sensor_data.airspeed, sensor_data.pitch, SPD_TARGET,
AP_log_data)) {
        ap_flight_phase = GPS_HOMING;
      }
      else ap_flight_phase = SPD_DESCENT;
    }
    else {
      ap_flight_phase = GPS_HOMING;
    }
  }
}
}

```

```

    xQueueSend(Autopilot_Queue, &AP_log_data, portMAX_DELAY);
    vTaskSuspend(NULL);
}
}

bool Autopilot_HDG_SEL_IMU(float roll, float yaw, float bearing_change, Autopilot_Data&
AP_log_data) {
    AP_log_data.ap_mode = "AP_HDG_SEL_IMU";
    unsigned int pitcheron_angle = 0;
    static const float target_bearing = wrap_angle(yaw + bearing_change); // Gets target_bearing
    only from first yaw reading (on release detection).
    float bearing_correction = signed_bearing_correction(yaw, target_bearing); // Continuously
    recalculated from current bearing.
    if ((bearing_correction >= -180) && (bearing_correction <= (-180+TURN_BIAS)) &&
    RIGHT_TURN_BIAS) bearing_correction = -bearing_correction;
    else if ((bearing_correction >= (180-TURN_BIAS)) && (bearing_correction <= 180) &&
    LEFT_TURN_BIAS) bearing_correction = -bearing_correction;
    float target_roll = Kp_ROLL_BEARING_CORR*bearing_correction; // To turn right, target
    roll is right
    if (target_roll < ROLL_LIM_MIN) target_roll = ROLL_LIM_MIN;
    else if (target_roll > ROLL_LIM_MAX) target_roll = ROLL_LIM_MAX;
    AP_log_data.ap_target_bearing = target_bearing;
    AP_log_data.ap_target_roll = target_roll;
    AP_log_data.ap_target_pitch = 0;

    // First case: Flight within desired envelope for bearing
    if ((bearing_correction >= HDG_TARGET_DEVIATION_LOW) && (bearing_correction <=
    HDG_TARGET_DEVIATION_HIGH)) {
        target_roll = 0;
        AP_log_data.ap_target_roll = 0;
        // Verify that roll also within desired envelope for 0 bearing correction
        if ((target_roll-roll >= ROLL_TARGET_DEVIATION_LOW) && (target_roll-roll <=
        ROLL_TARGET_DEVIATION_HIGH)) {
            actuate_pitcherons(0, MAINTAIN_ANGLE);
            return true; // Don't do anything (Handover to AP_SPD_TRIM)
        }
    }
}

```



```

// Second case: Flight within desired envelope for bearing but not for roll
else {
    // Could be greatly simplified to wings level, but doing this initially to match third case
    pitcheron_angle = (unsigned int)round(Kp_SERVO_ANGLE_ROLL*(fabs(target_roll -
roll)));
    if (pitcheron_angle > MAX_PITCHERON_ANGLE) pitcheron_angle =
MAX_PITCHERON_ANGLE;
    if (target_roll - roll > ROLL_TARGET_DEVIATION_HIGH) {
        // Roll is too low (aircraft is banking too much to the right), correct right
        actuate_pitcherons(pitcheron_angle, ROLL_RIGHT);
    }
    else if (target_roll - roll < -ROLL_TARGET_DEVIATION_HIGH) {
        // Roll is too high (aircraft is banking too much to the left), correct left
        actuate_pitcherons(pitcheron_angle, ROLL_LEFT);
    }
    else if (target_roll - roll > -ROLL_TARGET_DEVIATION_LOW) {
        // Roll is slightly low, apply small correction right
        actuate_pitcherons(pitcheron_angle, ROLL_RIGHT);
    }
    else if (target_roll - roll < ROLL_TARGET_DEVIATION_LOW) {
        // Roll is slightly high, apply small correction left
        actuate_pitcherons(pitcheron_angle, ROLL_LEFT);
    }
    else {
        // Roll is within the acceptable range, no correction needed
        actuate_pitcherons(0, MAINTAIN_ANGLE);
    }
    // return false; (by fall through)
}
}

// Third case: Flight outside of envelope for bearing (roll envelope irrelevant)
else {
    // Figure out whether left or right turn is needed
    // Left Turn Needed = Roll Left
    pitcheron_angle = (unsigned int)round(Kp_SERVO_ANGLE_ROLL*(fabs(target_roll -

```

```

roll));
    if (pitcheron_angle > MAX_PITCHERON_ANGLE) pitcheron_angle =
MAX_PITCHERON_ANGLE;
    if (target_roll - roll > ROLL_TARGET_DEVIATION_HIGH) {
        // Roll is too low (aircraft is banking too much to the right), correct right
        actuate_pitcherons(pitcheron_angle, ROLL_RIGHT);
    }
    else if (target_roll - roll < -ROLL_TARGET_DEVIATION_HIGH) {
        // Roll is too high (aircraft is banking too much to the left), correct left
        actuate_pitcherons(pitcheron_angle, ROLL_LEFT);
    }
    else if (target_roll - roll > -ROLL_TARGET_DEVIATION_LOW) {
        // Roll is slightly low, apply small correction right
        actuate_pitcherons(pitcheron_angle, ROLL_RIGHT);
    }
    else if (target_roll - roll < ROLL_TARGET_DEVIATION_LOW) {
        // Roll is slightly high, apply small correction left
        actuate_pitcherons(pitcheron_angle, ROLL_LEFT);
    }
    else {
        // Roll is within the acceptable range, no correction needed
        actuate_pitcherons(0, MAINTAIN_ANGLE);
    }

}
return false;
}

bool Autopilot_HDG_SEL_GPS(bool act_true, float roll, float current_heading, float current_lat,
float current_long, float target_lat, float target_long, Autopilot_Data& AP_log_data) {
    AP_log_data.ap_mode = "AP_HDG_SEL_GPS";
    unsigned int pitcheron_angle = 0;
    float target_bearing = calculate_bearing(current_lat, current_long, target_lat, target_long,
current_heading); // Gets target_bearing from current and target coordinates (continuously
recalculated)
    float bearing_correction = signed_bearing_correction(current_heading, target_bearing); //

```

Continuously recalculated from current bearing.

```

float target_roll = Kp_ROLL_BEARING_CORR*bearing_correction; // To turn right, target
roll is right
if (target_roll < ROLL_LIM_MIN) target_roll = ROLL_LIM_MIN;
else if (target_roll > ROLL_LIM_MAX) target_roll = ROLL_LIM_MAX;
AP_log_data.ap_target_bearing = target_bearing;
AP_log_data.ap_target_roll = target_roll;
AP_log_data.ap_target_pitch = 0;

// First case: Flight within desired envelope for bearing
if ((bearing_correction >= HDG_TARGET_DEVIATION_LOW) && (bearing_correction <=
HDG_TARGET_DEVIATION_HIGH)) {
    target_roll = 0;
    AP_log_data.ap_target_roll = 0;
    // Verify that roll also within desired envelope for 0 bearing correction
    if ((target_roll-roll >= ROLL_TARGET_DEVIATION_LOW) && (target_roll-roll <=
ROLL_TARGET_DEVIATION_HIGH)) {
        if (act_true) actuate_pitcherons(0, MAINTAIN_ANGLE);
        return true; // Don't do anything (Handover to AP_SPD_TRIM)
    }
    // Second case: Flight within desired envelope for bearing but not for roll
    else {
        // Could be greatly simplified to wings level, but doing this initially to match third case
        pitcheron_angle = (unsigned int)round(Kp_SERVO_ANGLE_ROLL*(fabs(target_roll -
roll)));
        if (pitcheron_angle > MAX_PITCHERON_ANGLE) pitcheron_angle =
MAX_PITCHERON_ANGLE;
        if (target_roll - roll > ROLL_TARGET_DEVIATION_HIGH) {
            // Roll is too low (aircraft is banking too much to the right), correct right
            actuate_pitcherons(pitcheron_angle, ROLL_RIGHT);
        }
        else if (target_roll - roll < -ROLL_TARGET_DEVIATION_HIGH) {
            // Roll is too high (aircraft is banking too much to the left), correct left
            actuate_pitcherons(pitcheron_angle, ROLL_LEFT);
        }
        else if (target_roll - roll > -ROLL_TARGET_DEVIATION_LOW) {

```

```

        // Roll is slightly low, apply small correction right
        actuate_pitcherons(pitcheron_angle, ROLL_RIGHT);
    }
    else if (target_roll - roll < ROLL_TARGET_DEVIATION_LOW) {
        // Roll is slightly high, apply small correction left
        actuate_pitcherons(pitcheron_angle, ROLL_LEFT);
    }
    else {
        // Roll is within the acceptable range, no correction needed
        actuate_pitcherons(0, MAINTAIN_ANGLE);
    }
    // return false; (by fall through)
}
}
// Third case: Flight outside of envelope for bearing (roll envelope irrelevant)
else {
    // Figure out whether left or right turn is needed
    // Left Turn Needed = Roll Left
    pitcheron_angle = (unsigned int)round(Kp_SERVO_ANGLE_ROLL*(fabs(target_roll -
roll)));
    if (pitcheron_angle > MAX_PITCHERON_ANGLE) pitcheron_angle =
MAX_PITCHERON_ANGLE;
    if (target_roll - roll > ROLL_TARGET_DEVIATION_HIGH) {
        // Roll is too low (aircraft is banking too much to the right), correct right
        actuate_pitcherons(pitcheron_angle, ROLL_RIGHT);
    }
    else if (target_roll - roll < -ROLL_TARGET_DEVIATION_HIGH) {
        // Roll is too high (aircraft is banking too much to the left), correct left
        actuate_pitcherons(pitcheron_angle, ROLL_LEFT);
    }
    else if (target_roll - roll > -ROLL_TARGET_DEVIATION_LOW) {
        // Roll is slightly low, apply small correction right
        actuate_pitcherons(pitcheron_angle, ROLL_RIGHT);
    }
    else if (target_roll - roll < ROLL_TARGET_DEVIATION_LOW) {

```

```

    // Roll is slightly high, apply small correction left
    actuate_pitcherons(pitcheron_angle, ROLL_LEFT);
}
else {
    // Roll is within the acceptable range, no correction needed
    actuate_pitcherons(0, MAINTAIN_ANGLE);
}

}
return false;
}

bool Autopilot_SPD_TRIM(float airspeed, float pitch, float target_airspeed, Autopilot_Data&
AP_log_data) {
    AP_log_data.ap_mode = "AP_SPD_TRIM";
    unsigned int pitcheron_angle = 0;
    float speed_correction = target_airspeed - airspeed; // Positive speed correction = need to
    speed up (too slow) => Pitch nose down.
    float target_pitch = Kp_PITCH_SPD_CORR*speed_correction; // Kp NEGATIVE! e.g. +10
    speed correction = -10 target_pitch
    // Limit checking for target_pitch
    if (target_pitch < PITCH_LIM_MIN) target_pitch = PITCH_LIM_MIN;
    else if (target_pitch > PITCH_LIM_MAX) target_pitch = PITCH_LIM_MAX;
    AP_log_data.ap_target_pitch = target_pitch;
    AP_log_data.ap_target_roll = 0;
    // First case: Flight within desired speed envelope
    if (((speed_correction >= SPD_TARGET_DEVIATION_LOW)) && (speed_correction <=
    SPD_TARGET_DEVIATION_HIGH)) {
        target_pitch = 0;
        AP_log_data.ap_target_pitch = 0;
        // Verify that pitch also within desired envelope for 0 airspeed correction
        if (((target_pitch-pitch >= PITCH_TARGET_DEVIATION_LOW) && (target_pitch-pitch
        <= PITCH_TARGET_DEVIATION_HIGH)) {
            actuate_pitcherons(0, MAINTAIN_ANGLE);
            return true; // Don't do anything (handover to AP_HDG_SEL_IMU)
        }
    }
}

```

```

    // Second case: Flight within desired envelope for speed but not for pitch
    else {
        pitcheron_angle = (unsigned int)round(Kp_SERVO_ANGLE_PITCH*(fabs(target_pitch
- pitch)));
        if (pitcheron_angle > MAX_PITCHERON_ANGLE) pitcheron_angle =
MAX_PITCHERON_ANGLE;
        if (target_pitch-pitch > PITCH_TARGET_DEVIATION_HIGH)
actuate_pitcherons(pitcheron_angle, PITCH_NOSE_UP);
        else if (target_pitch-pitch < -PITCH_TARGET_DEVIATION_HIGH)
actuate_pitcherons(pitcheron_angle, PITCH_NOSE_DOWN);
        else if (target_pitch-pitch > -PITCH_TARGET_DEVIATION_LOW)
actuate_pitcherons(pitcheron_angle, PITCH_NOSE_UP);
        else if (target_pitch-pitch < PITCH_TARGET_DEVIATION_LOW)
actuate_pitcherons(pitcheron_angle, PITCH_NOSE_DOWN);
        else actuate_pitcherons(0, MAINTAIN_ANGLE);
    }
}
// Third case: Flight outside of envelope for speed (pitch envelope irrelevant)
else {
    // Figure out whether pitch up or pitch down is needed
    // Pitch Up Needed = PITCH_NOSE_UP
    pitcheron_angle = (unsigned int)round(Kp_SERVO_ANGLE_PITCH*(fabs(target_pitch -
pitch)));
    if (pitcheron_angle > MAX_PITCHERON_ANGLE) pitcheron_angle =
MAX_PITCHERON_ANGLE;
    if (target_pitch-pitch > PITCH_TARGET_DEVIATION_HIGH)
actuate_pitcherons(pitcheron_angle, PITCH_NOSE_UP);
    else if (target_pitch-pitch < -PITCH_TARGET_DEVIATION_HIGH)
actuate_pitcherons(pitcheron_angle, PITCH_NOSE_DOWN);
    else if (target_pitch-pitch > -PITCH_TARGET_DEVIATION_LOW)
actuate_pitcherons(pitcheron_angle, PITCH_NOSE_UP);
    else if (target_pitch-pitch < PITCH_TARGET_DEVIATION_LOW)
actuate_pitcherons(pitcheron_angle, PITCH_NOSE_DOWN);
    else actuate_pitcherons(0, MAINTAIN_ANGLE);
}
return false;
}

```

```

bool Autopilot_FLT_ENVELOPE_PROT(float roll, float pitch, float airspeed, Autopilot_Data&
AP_log_data) {
    AP_log_data.ap_mode = "AP_FLT_ENVELOPE_PROT";
    // Priority 1: Roll Protection
    // Priority 2: Pitch Protection
    // Priority 3: Speed Protection
    if ((roll < ROLL_LIM_MIN) && ROLL_PROT_EN) {
        AP_log_data.ap_mode = "AP_PROT_ROLL_MIN";
        float target_roll = ROLL_LIM_MAX;
        AP_log_data.ap_target_roll = target_roll;
        AP_log_data.ap_target_pitch = 0;
        float pitcheron_angle = (unsigned int)round(Kp_SERVO_ANGLE_ROLL*(fabs(target_roll
- roll)));
        if (pitcheron_angle > MAX_PITCHERON_ANGLE) pitcheron_angle =
MAX_PITCHERON_ANGLE;
        actuate_pitcherons(pitcheron_angle, ROLL_RIGHT);
    }
    else if ((roll > ROLL_LIM_MAX) && ROLL_PROT_EN) {
        AP_log_data.ap_mode = "AP_PROT_ROLL_MAX";
        float target_roll = ROLL_LIM_MIN;
        AP_log_data.ap_target_roll = target_roll;
        AP_log_data.ap_target_pitch = 0;
        float pitcheron_angle = (unsigned int)round(Kp_SERVO_ANGLE_ROLL*(fabs(target_roll
- roll)));
        if (pitcheron_angle > MAX_PITCHERON_ANGLE) pitcheron_angle =
MAX_PITCHERON_ANGLE;
        actuate_pitcherons(pitcheron_angle, ROLL_LEFT);
    }
    else if ((pitch < PITCH_LIM_MIN) && PITCH_PROT_EN) {
        AP_log_data.ap_mode = "AP_PROT_PITCH_MIN";
        float target_pitch = PITCH_LIM_MAX;
        AP_log_data.ap_target_pitch = target_pitch;
        AP_log_data.ap_target_roll = 0;
        float pitcheron_angle = (unsigned
int)round(Kp_SERVO_ANGLE_PITCH*(fabs(target_pitch - pitch)));

```

```

    if (pitcheron_angle > MAX_PITCHERON_ANGLE) pitcheron_angle =
MAX_PITCHERON_ANGLE;
    actuate_pitcherons(pitcheron_angle, PITCH_NOSE_UP);
}
else if ((pitch > PITCH_LIM_MAX) && PITCH_PROT_EN) {
    AP_log_data.ap_mode = "AP_PROT_PITCH_MAX";
    float target_pitch = PITCH_LIM_MIN;
    AP_log_data.ap_target_pitch = target_pitch;
    AP_log_data.ap_target_roll = 0;
    float pitcheron_angle = (unsigned
int)round(Kp_SERVO_ANGLE_PITCH*(fabs(target_pitch - pitch)));
    if (pitcheron_angle > MAX_PITCHERON_ANGLE) pitcheron_angle =
MAX_PITCHERON_ANGLE;
    actuate_pitcherons(pitcheron_angle, PITCH_NOSE_DOWN);
}
else if ((airspeed < STALL_SPEED) && STALL_PROT_EN) {
    AP_log_data.ap_mode = "AP_PROT_STALL";
    // float target_airspeed = STALL_SPEED+OVERCORRECTION_SPEED;
    float target_pitch = PITCH_LIM_MIN;
    AP_log_data.ap_target_pitch = target_pitch;
    AP_log_data.ap_target_roll = 0;
    unsigned int pitcheron_angle = (unsigned
int)round(Kp_SERVO_ANGLE_PITCH*(fabs(target_pitch - pitch)));
    if (pitcheron_angle > MAX_PITCHERON_ANGLE) pitcheron_angle =
MAX_PITCHERON_ANGLE;
    actuate_pitcherons(pitcheron_angle, PITCH_NOSE_DOWN);
}
else if ((airspeed > OVERSPEED) && OVSPD_PROT_EN) {
    AP_log_data.ap_mode = "AP_PROT_OVERSPEED";
    // float target_airspeed = OVERSPEED-OVERCORRECTION_SPEED;
    float target_pitch = PITCH_LIM_MAX;
    AP_log_data.ap_target_pitch = target_pitch;
    AP_log_data.ap_target_roll = 0;
    unsigned int pitcheron_angle = (unsigned
int)round(Kp_SERVO_ANGLE_PITCH*(fabs(target_pitch - pitch)));
    if (pitcheron_angle > MAX_PITCHERON_ANGLE) pitcheron_angle =

```



```

MAX_PITCHERON_ANGLE;
    actuate_pitcherons(pitcheron_angle, PITCH_NOSE_UP);
}
else return true;

return false;
}

bool Autopilot_ROLL_FIXED(float roll, float target_roll, Autopilot_Data& AP_log_data) {
    AP_log_data.ap_mode = "AP_ROLL_FIXED";
    if (target_roll < ROLL_LIM_MIN) target_roll = ROLL_LIM_MIN;
    else if (target_roll > ROLL_LIM_MAX) target_roll = ROLL_LIM_MAX;
    AP_log_data.ap_target_roll = target_roll;
    AP_log_data.ap_target_pitch = 0;
    if ((target_roll - roll >= ROLL_TARGET_DEVIATION_LOW) && (target_roll - roll <=
ROLL_TARGET_DEVIATION_HIGH)) {
        actuate_pitcherons(0, MAINTAIN_ANGLE);
        return true; // Don't do anything (Handover to AP_SPD_TRIM)
    }
    // Second case: Flight within desired envelope for bearing but not for roll
    else {
        float pitcheron_angle = (unsigned int)round(Kp_SERVO_ANGLE_ROLL*(fabs(target_roll
- roll)));
        if (pitcheron_angle > MAX_PITCHERON_ANGLE) pitcheron_angle =
MAX_PITCHERON_ANGLE;
        if (target_roll - roll > ROLL_TARGET_DEVIATION_HIGH) {
            // Roll is too low (aircraft is banking too much to the right), correct right
            actuate_pitcherons(pitcheron_angle, ROLL_RIGHT);
        }
        else if (target_roll - roll < -ROLL_TARGET_DEVIATION_HIGH) {
            // Roll is too high (aircraft is banking too much to the left), correct left
            actuate_pitcherons(pitcheron_angle, ROLL_LEFT);
        }
        else if (target_roll - roll > -ROLL_TARGET_DEVIATION_LOW) {
            // Roll is slightly low, apply small correction right

```

```

        actuate_pitcherons(pitcheron_angle, ROLL_RIGHT);
    }
    else if (target_roll - roll < ROLL_TARGET_DEVIATION_LOW) {
        // Roll is slightly high, apply small correction left
        actuate_pitcherons(pitcheron_angle, ROLL_LEFT);
    }
    else {
        // Roll is within the acceptable range, no correction needed
        actuate_pitcherons(0, MAINTAIN_ANGLE);
    }
}
return false;
}

bool Autopilot_PITCH_FIXED(float pitch, float target_pitch, Autopilot_Data& AP_log_data) {
    AP_log_data.ap_mode = "AP_PITCH_FIXED";
    if (target_pitch < PITCH_LIM_MIN) target_pitch = PITCH_LIM_MIN;
    else if (target_pitch > PITCH_LIM_MAX) target_pitch = PITCH_LIM_MAX;
    AP_log_data.ap_target_pitch = target_pitch;
    AP_log_data.ap_target_roll = 0;
    if ((target_pitch-pitch >= PITCH_TARGET_DEVIATION_LOW) && (target_pitch-pitch <=
PITCH_TARGET_DEVIATION_HIGH)) {
        actuate_pitcherons(0, MAINTAIN_ANGLE);
        return true; // Don't do anything (handover to AP_HDG_SEL_IMU)
    }
    else {
        float pitcheron_angle = (unsigned
int)round(Kp_SERVO_ANGLE_PITCH*(fabs(target_pitch - pitch)));
        if (pitcheron_angle > MAX_PITCHERON_ANGLE) pitcheron_angle =
MAX_PITCHERON_ANGLE;
        if (target_pitch-pitch > PITCH_TARGET_DEVIATION_HIGH)
actuate_pitcherons(pitcheron_angle, PITCH_NOSE_UP);
        else if (target_pitch-pitch < -PITCH_TARGET_DEVIATION_HIGH)
actuate_pitcherons(pitcheron_angle, PITCH_NOSE_DOWN);
        else if (target_pitch-pitch > -PITCH_TARGET_DEVIATION_LOW)
actuate_pitcherons(pitcheron_angle, PITCH_NOSE_UP);
    }
}

```

```

        else if (target_pitch-pitch < PITCH_TARGET_DEVIATION_LOW)
        actuate_pitcherons(pitcheron_angle, PITCH_NOSE_DOWN);
        else actuate_pitcherons(0, MAINTAIN_ANGLE);
        // return false; (by fall through)
    }
    return false;
}

```

```

=====
// File: datalogger.cpp
=====

// This file contains all SD card related functions to initialize and write to the SD Card
#include <SPI.h>
#include <SD.h>
#include "pin_map.h"
#include "datalogger.h"
#include "sensors.h"
#include "autopilot.h"
#include "pitcheron_servos.h"
#include "tasks.h"
#include "queues.h"
#include "semaphores.h"

#define FILE_COUNT_START 0
#define INIT_DELAY_SD 100
#define LINE_NUM_START 1
#define DP_DATA 3 // Decimal places to record
#define DP_GPS 6 // Latitude/Longitude decimal places
#define GPS_FIX_DELAY_THRESHOLD 0.250 // If more than 250ms passed since last fix,
take data from other sensors again

bool log_to_serial;
bool log_to_SD;

```

```

File datafile;           // File object to handle file writing
SPIClass mySPI(VSPI);

void init_SD(bool serial_log, bool SD_log) {
    if (serial_log) log_to_serial = true;
    else log_to_serial = false;
    if (SD_log) log_to_SD = true;
    else log_to_SD = false;
    if (!log_to_SD) return;
    unsigned int file_counter = FILE_COUNT_START;    // Start the file counter at 1
    char filename[16];
    Serial.print("Initializing SD Card...");
    pinMode(SD_CS, OUTPUT);
    digitalWrite(SD_CS, HIGH);
    mySPI.begin(SD_SCK, SD_MISO, SD_MOSI, SD_CS);
    // SPI.setFrequency(1000000);
    if (!SD.begin(SD_CS, mySPI, 80000000)) {
        Serial.println("Card failed, or not present.");
        log_to_SD = false;
    }
    Serial.println("DONE!");
    if (log_to_SD == true) {
        Serial.println("Searching for next available filename");
        // Get next filename for filename
        sprintf(filename, "/data%03u.csv", file_counter); // Create the filename with the counter
        while(SD.exists(filename)) {
            file_counter++;
            if (file_counter >= 1000) {
                file_counter = FILE_COUNT_START;
                Serial.println("Overwriting file /data000.csv... 1000+ files in storage. Please DELETE SOME!!!");
                sprintf(filename, "/data%03u.csv", file_counter); // Create the filename with the
counter
                break;
            }
        }
    }
}

```

```

        else sprintf(filename, "/data%03u.csv", file_counter); // Create the filename with the
counter
    }

    datafile = SD.open(filename, FILE_WRITE);
    // Open File
    if (datafile) {
        Serial.printf("Writing to file: /data%03u.csv\n", file_counter);
    }
    else {
        Serial.printf("Failed to Open File: /data%03u.csv for writing!\n", file_counter);
        log_to_SD = false;
        return;
    }
    // Write Header to file
    const char csv_header[] =
"Line_Num,ESP32_Time_s,ID0_IMU,LinAcc_x,LinAcc_y,LinAcc_z,Pitch,Roll,Yaw,Gyro_x,G
yro_y,Gyro_z,Magnet_uT_x,Magnet_uT_y,Magnet_uT_z,"

"Grav_x,Grav_y,Grav_z,Quat_re,Quat_i,Quat_j,Quat_k,ID1_ASPD,RawPress_Pa,temp_C,Raw
Airspeed,CorrAirspeed,"

"ID2_GPS,latitude,longitude,heading,gnd_speed,altitude,hours,mins,secs,hundredths,satellites,"

"ID3_AP,Flight_Phase,AP_Mode,AP_HDG_TGT,AP_ROLL_TGT,AP_PITCH_TGT,"

"ID4_SERVO,servo_L_angle,servo_R_angle,servo_Angle_TGT,servo_Action_TGT\n";

    Serial.println("Writing .csv header:");
    datafile.print(csv_header);
    datafile.flush();
    if (log_to_serial) {
        Serial.print(csv_header);
    }
    Serial.println(".csv Header Writing DONE!");
}

```

```
}

void log_data(void* pvParameters) {
    float current_time = 0;
    IMU_Data imu;
    Airspeed_Data pitot;
    GPS_Data gps;
    Flight_Data ap_input_data;
    Autopilot_Data AP_data;
    Pitcheron_Data pitcherons;
    digitalWrite(BUILTIN_LED_PIN, LOW);
    unsigned long line_num = LINE_NUM_START; // These are only initialized once
    while(true) {
        //Serial.println("Datalogging Task");
        static unsigned long start_time = micros();
        // Serial.println("Taking GPS Semaphore");
        // xSemaphoreTake(gps_done, portMAX_DELAY);
        xQueueReceive(GPS_Queue, &gps, portMAX_DELAY);

        // Serial.println("Taking IMU Semaphore");
        // xSemaphoreTake(imu_done, portMAX_DELAY);
        xQueueReceive(IMU_Queue, &imu, portMAX_DELAY);

        // Serial.println("Taking Airspeed Semaphore");
        // xSemaphoreTake(airspeed_done, portMAX_DELAY);
        xQueueReceive(Airspeed_Queue, &pitot, portMAX_DELAY);

        current_time = (float)((micros() - start_time)/1000000.0);
        // Load received data onto Flight_Data_Queue for Autopilot
        ap_input_data.time = current_time;
        ap_input_data.pitch = imu.euler[0];
        ap_input_data.roll = imu.euler[1];
        ap_input_data.yaw = imu.euler[2];
        ap_input_data.airspeed = pitot.airspeed[1]; // Corrected airspeed
        ap_input_data.gnd_speed = gps.gnd_speed;
```

[illegible]

```

    DP_DATA, imu.rotation[3]);

    // Airspeed/Pitot Tube Data
    datafile.printf("%u,%.*f,%.*f,%.*f,%.*f,", pitot.sensor_id, \
    DP_DATA, pitot.diff_pressure, \
    DP_DATA, pitot.temperature, \
    DP_DATA, pitot.airspeed[0], \
    DP_DATA, pitot.airspeed[1]);

    // GPS Data
    datafile.printf("%u,%.*f,%.*f,%.*f,%.*f,%.*f,%u,%u,%u,%u,", gps.sensor_id, \
    DP_GPS, gps.latitude, \
    DP_GPS, gps.longitude, \
    DP_GPS, gps.heading, \
    DP_DATA, gps.gnd_speed, \
    DP_GPS, gps.altitude, \
    gps.hours, gps.minutes, gps.seconds, gps.hundredths, gps.satellites);

    // Autopilot Data
    datafile.printf("%u,%s,%s,%.*f,%.*f,%.*f,", AP_data.sensor_id, AP_data.flight_phase, \
    AP_data.ap_mode, DP_DATA, AP_data.ap_target_bearing, DP_DATA,
    AP_data.ap_target_roll, DP_DATA, AP_data.ap_target_pitch);

    // Servo Data
    datafile.printf("%u,%d,%d,%u,%s\n", pitcherons.sensor_id, \
    pitcherons.raw_angle_l, \
    pitcherons.raw_angle_r, \
    pitcherons.angle_target, \
    pitcherons.action_target); // Do not use comma in action target String!

    datafile.flush();
}

if (log_to_serial) {
    // Line Num + ESP Time + IMU Data

```


[illegible]

```

    DP_GPS, gps.latitude, \
    DP_GPS, gps.longitude, \
    DP_DATA, gps.heading, \
    DP_GPS, gps.gnd_speed, \
    DP_DATA, gps.altitude, \
    gps.hours, gps.minutes, gps.seconds, gps.hundredths, gps.satellites);

// Autopilot Data
Serial.printf("%u,%s,%s,%.*f,%.*f,%.*f,", AP_data.sensor_id, AP_data.flight_phase, \
    AP_data.ap_mode, DP_DATA, AP_data.ap_target_bearing, DP_DATA,
AP_data.ap_target_roll, DP_DATA, AP_data.ap_target_pitch);

// Servo Data
Serial.printf("%u,%d,%d,%u,%s\n", pitcherons.sensor_id, \
    pitcherons.raw_angle_l, \
    pitcherons.raw_angle_r, \
    pitcherons.angle_target, \
    pitcherons.action_target); // Do not use comma in action target String!

// Serial.flush();
}

line_num++;
// Resume suspended reading tasks after logging data to SD card and loop again.
if((digitalRead(BUILTIN_LED_PIN) == LOW) && (gps.satellites != 0)) {
    digitalWrite(BUILTIN_LED_PIN, HIGH);
}
else {
    digitalWrite(BUILTIN_LED_PIN, LOW);
}

//vTaskDelay(pdMS_TO_TICKS(10)); // Extra Delay (in milliseconds) to make serial
monitor data human-readable. Too fast otherwise!
vTaskResume(read_gps_task);
vTaskResume(read_imu_task);

```

```

    vTaskResume(read_pitot_task);
    vTaskResume(autopilot_task);
}
}

```

```

=====
// File: main.cpp
=====

```

```

#include <Arduino.h>
#include "soc/soc.h"
#include "soc/rtc_cntl_reg.h"
#include "esp_sleep.h"
#include "tasks.h"
#include "queues.h"
#include "semaphores.h"
#include "sensors.h"
#include "datalogger.h"
#include "pitcheron_servos.h"
#include "trim_servos.h"
#include "strobe.h"
#include "autopilot.h"
#include "pin_map.h"

#define SERIAL_LOG true // Log Data to Serial
#define SD_LOG true // Log Data to SD Card file (failsafe for SD card popping out if true is
included. If that happens, true constant is ignored.)
#define TRIM_SERVOS false // Choose whether to run this program in regular or servo
trimming mode
#define SERVO_ACTUATION_TESTS true // Perform pitcheron servo tests during
initialization? (ignored if TRIM_SERVOS=true)
#define RELEASE_INIT false // Wait for release before running main code
#define BOOTUP_DELAY 2000 //ms

```

```

#define INSTALL_DEBOUNCE_DELAY 250 //ms
#define INSTALL_DELAY 10000
#define RELEASE_DELAY 250 //ms

void setup() {
    delay(BOOTUP_DELAY);
    //WRITE_PERI_REG(RTC_CNTL_BROWN_OUT_REG, 0);
    pinMode(RELEASE_DET_PIN, INPUT);
    esp_sleep_enable_ext0_wakeup((gpio_num_t)RELEASE_DET_PIN, HIGH);
    init_low_level_hw();
    init_strobe();
    init_queues();
    init_semaphores();
    init_all_sensors();

    if (TRIM_SERVOS == false) init_servos(SERVO_ACTUATION_TESTS);
    else trim_servos(); // Note: this instruction is blocking. No further lines of code in this file will
    execute and RTOS Scheduler never starts.

    init_SD(SERIAL_LOG, SD_LOG); // FORMAT SD CARD TO FAT32 BEFORE FIRST USE

    if (RELEASE_INIT) {
        Serial.println("All Systems Initialized. Waiting for GPIO 19 release detection
        (HIGH=>LOW=>HIGH)...");
        digitalWrite(BUILTIN_LED_PIN, LOW);
        while (true) {
            while (true) {
                if (digitalRead(RELEASE_DET_PIN) == HIGH)
                    delay(INSTALL_DEBOUNCE_DELAY);
                else {
                    delay(INSTALL_DEBOUNCE_DELAY);
                    if (digitalRead(RELEASE_DET_PIN) == LOW) break;
                }
            }
        }
    }
}

```

```

    digitalWrite(BUILTIN_LED_PIN, HIGH);
    delay(INSTALL_DELAY);
    if (digitalRead(RELEASE_DET_PIN) == HIGH) {
        digitalWrite(BUILTIN_LED_PIN, LOW);
        delay(250);
        digitalWrite(BUILTIN_LED_PIN, HIGH);
        continue;
    }
    else break;
}
digitalWrite(BUILTIN_LED_PIN, LOW);

while (digitalRead(RELEASE_DET_PIN) == LOW) {
    esp_light_sleep_start();
    delay(RELEASE_DELAY); // for debouncing release detection magnet
}
}
init_tasks();
Serial.println("All Systems ONLINE! All Tasks Started Successfully! RTOS Task Scheduler
RUNNING!\n");

}

void loop() {
    ;//vTaskStartScheduler();
}

=====
// File: pitcheron_servos.cpp
=====
#include <Arduino.h>
#include <ESP32Servo.h>

```

```
#include "pin_map.h"
#include "pitcheron_servos.h"
#include "queues.h"

#define DISABLE_SERVO_L false
#define DISABLE_SERVO_R false

// Basic Assumption: Pitcheron Angle = Servo Angle
//
https://cdn.shopify.com/s/files/1/0570/1766/3541/files/X08H\_V6.0\_Technical\_Specification.pdf?v=1700472376
//
https://kstservos.com/collections/gliders-wing-servos/products/x08h-plus-horizontal-lug-servo-5-3kg-cm-0-09s-9-5g-8mm

// Servos l and r are connected to left and right pitcherons as viewed from behind the glider.
// CW_CONVENTION: Clockwise = Positive Angle, Counterclockwise = Negative Angle. Set
// CONVENTION to -1 if opposite (still assumes 2 identical servos)
#define CW_CONVENTION -1 // 1=Clockwise Positive, -1=Counterclockwise Positive

// CG_CONVENTION: Determine whether fully assembled glider CG is behind or in front of the
// pitcherons (wingtip test!)
#define CG_CONVENTION 1 // 1 = Pitcherons point up, nose goes down (pitcherons behind
// CG, like elevators). -1 = Pitcherons point up, nose goes up (pitcherons in front of CG).

// Note: center would be 0 degrees left and right in code, but may not be the case in real life.
// Trim offset added to center
#define RAW_TRIM_L 9 // deg. Set this to whatever angle must be requested in independent
// servo tests (actuate_servo_l) to center left servo when testing (regardless of CONVENTION).
#define RAW_TRIM_R 7 // deg. Set this to whatever angle must be requested in independent
// servo tests (actuate_servo_r) to center right servo when testing (regardless of CONVENTION).
// Define Servo Physical Limits
#define MIN_SERVO_us 1000 // us
#define MAX_SERVO_us 2000 // us
#define PWM_FREQUENCY 333 // Hz
#define INITIAL_ANGLE 14
```

```

#define ACTUATION_DELAY_ms 1000 // How long to wait after each actuation test

Servo left_servo;
Servo right_servo;

void init_servos_trim(void) {
    Serial.println("Initializing Servos...");
    if (DISABLE_SERVO_L) Serial.println("Left Servo Disabled. Code will run as if it is
enabled, but without physically initializing/actuating.");
    if (DISABLE_SERVO_R) Serial.println("Right Servo Disabled. Code will run as if it is
enabled, but without physically initializing/actuating.");
    ..... if (!DISABLE_SERVO_L) ESP32PWM::allocateTimer(0); // Allocate timer for both servos
    ..... if (!DISABLE_SERVO_R) ESP32PWM::allocateTimer(1); // Allocate timer for both servos
    if (!DISABLE_SERVO_L) left_servo.setPeriodHertz(PWM_FREQUENCY); // 333 Hz
servo
    if (!DISABLE_SERVO_R) right_servo.setPeriodHertz(PWM_FREQUENCY); // 333 Hz
servo
    if (!DISABLE_SERVO_L) left_servo.attach(SERVO_L_PIN, 900, 2100); // Attach left servo
to pin
    if (!DISABLE_SERVO_R) right_servo.attach(SERVO_R_PIN, 900, 2100); // Attach right
servo to pin
    Serial.println("Servo Initialization Complete. Start Trimming.");
}

// This function maps angles to microseconds (PWM width) based on servo datasheet
int angle2us(int angle, int angle_min, int angle_max, int us_min, int us_max) {
    if (angle < angle_min) angle = angle_min; // Handle limit
    else if (angle > angle_max) angle = angle_max; // Handle limit
    return us_min + ((angle - angle_min) * (us_max - us_min)) / (angle_max - angle_min);
}

void actuate_servo_l(int raw_angle_servo_l) {
    if (!DISABLE_SERVO_L) left_servo.writeMicroseconds(angle2us(raw_angle_servo_l,
MIN_SERVO_ANGLE, MAX_SERVO_ANGLE, MIN_SERVO_us, MAX_SERVO_us));
}

```

```

void actuate_servo_r(int raw_angle_servo_r) {
    if (!DISABLE_SERVO_R) right_servo.writeMicroseconds(angle2us(raw_angle_servo_r,
MIN_SERVO_ANGLE, MAX_SERVO_ANGLE, MIN_SERVO_us, MAX_SERVO_us));
}

// This function initializes the servos, checks range of travel, and then centers servos
void init_servos(bool actuation_test = true) {
    Serial.println("Initializing Servos...");
    if (DISABLE_SERVO_L) Serial.println("Left Servo Disabled. Code will run as if it is
enabled, but without physically initializing/actuating.");
    if (DISABLE_SERVO_R) Serial.println("Right Servo Disabled. Code will run as if it is
enabled, but without physically initializing/actuating.");
    ..... if (!DISABLE_SERVO_L) ESP32PWM::allocateTimer(0); // Allocate timer for both servos
    ..... if (!DISABLE_SERVO_R) ESP32PWM::allocateTimer(1); // Allocate timer for both servos
    if (!DISABLE_SERVO_L) left_servo.setPeriodHertz(PWM_FREQUENCY); // 333 Hz
servo
    if (!DISABLE_SERVO_R) right_servo.setPeriodHertz(PWM_FREQUENCY); // 333 Hz
servo
    if (!DISABLE_SERVO_L) left_servo.attach(SERVO_L_PIN, 900, 2100); // Attach left servo
to pin
    if (!DISABLE_SERVO_R) right_servo.attach(SERVO_R_PIN, 900, 2100); // Attach right
servo to pin

    // Set servos to 0 position
    Serial.printf("Centering LEFT servo centered with trim @%dddeg.\n", RAW_TRIM_L);
    actuate_servo_l(RAW_TRIM_L+0);
    Serial.printf("Centering RIGHT servo centered with trim @%dddeg.\n", RAW_TRIM_R);
    actuate_servo_r(RAW_TRIM_R+0);
    Serial.println("Servo Initialization Complete! Verify both pitcherons are now correctly
trimmed/centered.");
    if (CW_CONVENTION == -1) Serial.println("Note: Using alternate convention for angles (+
=> counterclockwise, - => clockwise).");
    else if (CW_CONVENTION != 1) Serial.println("INVALID CW_CONVENTION
SPECIFIED! STOP! Roll control will be scaled incorrectly!");
    delay(ACTUATION_DELAY_ms);
    if (actuation_test == true) {
        Serial.println("Testing Servos... Observe movement carefully to verify correct

```



```

actuation/response.");
    // If both servos turn counterclockwise or clockwise, Pitcherons = Ailerons
    // If one turns clockwise and the other counterclockwise, Pitcherons = Elevator

    // Turn both servos in same direction (counterclockwise: counter-rotating pitcherons). Left
    // pitcheron points down, right points up (roll left)
    actuate_servo_l(RAW_TRIM_L+(SERVO_MIN_ALLOWED*CW_CONVENTION));
    actuate_servo_r(RAW_TRIM_R+(SERVO_MIN_ALLOWED*CW_CONVENTION));
    Serial.printf("Test #1/6: Confirm that LEFT pitcheron points DOWN and RIGHT pitcheron
    points UP (both@min allowed deflection = %ddeg).\n", SERVO_MIN_ALLOWED);
    delay(ACTUATION_DELAY_ms);

    // Turn both servos in same direction (clockwise: counter-rotating pitcherons). Left
    // pitcheron points up, Right points down (roll right)
    actuate_servo_l(RAW_TRIM_L+(SERVO_MAX_ALLOWED*CW_CONVENTION));
    actuate_servo_r(RAW_TRIM_R+(SERVO_MAX_ALLOWED*CW_CONVENTION));
    Serial.printf("Test #2/6: Confirm that LEFT pitcheron points UP and RIGHT pitcheron
    points DOWN (both@max allowed deflection = %ddeg).\n", SERVO_MAX_ALLOWED);
    delay(ACTUATION_DELAY_ms);

    // Center servos
    actuate_servo_l(RAW_TRIM_L+0);
    actuate_servo_r(RAW_TRIM_R+0);
    Serial.printf("Test #3/6: Confirm that BOTH pitcherons are CENTERED (servo_l
    trim@%ddeg; servo_r trim@%ddeg).\n", RAW_TRIM_L, RAW_TRIM_R);
    delay(ACTUATION_DELAY_ms);

    // Turn servos in opposite directions (L: clockwise, R: counterclockwise). Left pitcheron
    // points up, Right points up (pitch nose down/up)
    actuate_servo_l(RAW_TRIM_L+(SERVO_MAX_ALLOWED*CW_CONVENTION));
    actuate_servo_r(RAW_TRIM_R+(SERVO_MIN_ALLOWED*CW_CONVENTION));
    Serial.println("Test #4/6: Confirm that BOTH pitcherons point UP (max deflection).");
    if (CG_CONVENTION == 1) Serial.println("Pitcherons are BEHIND CG. Confirm this
    action causes NOSE PITCH DOWN.");
    else if (CG_CONVENTION == -1) Serial.println("Pitcherons are IN FRONT OF CG.
    Confirm this action causes NOSE PITCH UP.");

```

```

    else Serial.println("INVALID CG_CONVENTION SPECIFIED! STOP! There will be no
pitch control in-flight (only roll)!");
    delay(ACTUATION_DELAY_ms);

    // Turn servos in opposite directions (L: counterclockwise, R: clockwise). Left pitcheron
points down, Right points down (pitch nose up/down)
    actuate_servo_l(RAW_TRIM_L+(SERVO_MIN_ALLOWED*CW_CONVENTION));
    actuate_servo_r(RAW_TRIM_R+(SERVO_MAX_ALLOWED*CW_CONVENTION));
    Serial.println("Test #5/6: Confirm that BOTH pitcherons point DOWN (max deflection).");
    if (CG_CONVENTION == 1) Serial.println("Pitcherons are BEHIND CG. Confirm this
action causes NOSE PITCH UP.");
    else if (CG_CONVENTION == -1) Serial.println("Pitcherons are IN FRONT OF CG.
Confirm this action causes NOSE PITCH DOWN.");
    else Serial.println("INVALID CG_CONVENTION SPECIFIED! STOP! There will be no
pitch control in-flight (only roll)!");
    delay(ACTUATION_DELAY_ms);

    // Center servos
    actuate_servo_l(RAW_TRIM_L+0);
    actuate_servo_r(RAW_TRIM_R+0);
    Serial.printf("Test #6/6: Confirm that BOTH pitcherons are CENTERED (servo_l
trim@%dddeg; servo_r trim@%dddeg).\n", RAW_TRIM_L, RAW_TRIM_R);
    delay(ACTUATION_DELAY_ms);

    Serial.println("Servo Testing Complete! Verify both pitcherons are now correctly
trimmed/centered.");
}

}

// This function actuates 2 servos using actuate_servo_l and actuate_servo_r. Specify
ROLL_LEFT/ROLL_RIGHT/PITCH_NOSE_UP/PITCH_NOSE_DOWN or WINGS_LEVEL
for act_type_direction.
void actuate_pitcherons(unsigned int angle, enum Pitcheron_Actions act_type_direction) {
    // Never call this function twice in a row unless xQueueReceive(Pitcheron_Queue,
&pitcherons, portMAX_DELAY) has been called in between.

```

```

Pitcheron_Data new_pitcheron_data;
new_pitcheron_data.sensor_id = 4;
new_pitcheron_data.angle_target = angle;
new_pitcheron_data.raw_angle_l = RAW_TRIM_L+0;
new_pitcheron_data.raw_angle_r = RAW_TRIM_R+0;
new_pitcheron_data.action_target = "MAINTAIN_ANGLE";
switch(act_type_direction) {
  case WINGS_LEVEL:
    // Angle ignored (same as doing any action with angle=0)
    actuate_servo_l(RAW_TRIM_L+0);
    actuate_servo_r(RAW_TRIM_R+0);
    new_pitcheron_data.action_target = "WINGS_LEVEL";
    new_pitcheron_data.raw_angle_l = RAW_TRIM_L+0;
    new_pitcheron_data.raw_angle_r = RAW_TRIM_R+0;
    break;
  case ROLL_LEFT:
    // Servo rotation in same direction = Pitcherons actuate in opposite directions
    actuate_servo_l(RAW_TRIM_L-(angle*CW_CONVENTION));
    actuate_servo_r(RAW_TRIM_R-(angle*CW_CONVENTION));
    new_pitcheron_data.action_target = "ROLL_LEFT";
    new_pitcheron_data.raw_angle_l = RAW_TRIM_L-(angle*CW_CONVENTION);
    new_pitcheron_data.raw_angle_r = RAW_TRIM_R-(angle*CW_CONVENTION);
    break;
  case ROLL_RIGHT:
    // Servo rotation in same direction = Pitcherons actuate in opposite directions
    actuate_servo_l(RAW_TRIM_L+(angle*CW_CONVENTION));
    actuate_servo_r(RAW_TRIM_R+(angle*CW_CONVENTION));
    new_pitcheron_data.action_target = "ROLL_RIGHT";
    new_pitcheron_data.raw_angle_l = RAW_TRIM_L+(angle*CW_CONVENTION);
    new_pitcheron_data.raw_angle_r = RAW_TRIM_R+(angle*CW_CONVENTION);
    break;
  case PITCH_NOSE_UP:
    // Servo rotation in opposite direction = Pitcherons actuate in same directions
    actuate_servo_l(RAW_TRIM_L-(angle*CW_CONVENTION*CG_CONVENTION));
    actuate_servo_r(RAW_TRIM_R+(angle*CW_CONVENTION*CG_CONVENTION));

```

```

        new_pitcheron_data.action_target = "PITCH_NOSE_UP";
        new_pitcheron_data.raw_angle_l =
RAW_TRIM_L-(angle*CW_CONVENTION*CG_CONVENTION);
        new_pitcheron_data.raw_angle_r =
RAW_TRIM_R+(angle*CW_CONVENTION*CG_CONVENTION);
        break;
    case PITCH_NOSE_DOWN:
        // Servo rotation in opposite direction = Pitcherons actuate in same directions
        actuate_servo_l(RAW_TRIM_L+(angle*CW_CONVENTION*CG_CONVENTION));
        actuate_servo_r(RAW_TRIM_R-(angle*CW_CONVENTION*CG_CONVENTION));
        new_pitcheron_data.action_target = "PITCH_NOSE_DOWN";
        new_pitcheron_data.raw_angle_l =
RAW_TRIM_L+(angle*CW_CONVENTION*CG_CONVENTION);
        new_pitcheron_data.raw_angle_r =
RAW_TRIM_R-(angle*CW_CONVENTION*CG_CONVENTION);
        break;
    case MAINTAIN_ANGLE:
        new_pitcheron_data.action_target = "MAINTAIN_ANGLE";
        break;
    default:
        // Just center and do nothing else (same as WINGS_LEVEL).
        actuate_servo_l(RAW_TRIM_L+0);
        actuate_servo_r(RAW_TRIM_R+0);
        new_pitcheron_data.action_target = "WINGS_LEVEL";
        new_pitcheron_data.raw_angle_l = RAW_TRIM_L+0;
        new_pitcheron_data.raw_angle_r = RAW_TRIM_R+0;
        break;
    }
    xQueueSend(Pitcheron_Queue, &new_pitcheron_data, portMAX_DELAY);
}

```

```
// File: queues.cpp
```

```
#include "queues.h"
#include "sensors.h"
#include "pitcheron_servos.h"
#include "autopilot.h"
#include "datalogger.h"

// Declare queues (allocate space) for externs in queues.h
QueueHandle_t IMU_Queue = NULL;
QueueHandle_t Airspeed_Queue = NULL;
QueueHandle_t GPS_Queue = NULL;
QueueHandle_t Autopilot_Queue = NULL;
QueueHandle_t Pitcheron_Queue = NULL;

QueueHandle_t Flight_Data_Queue = NULL; // internal (not datalogged)

void init_queues() {
    // 1-element queues containing structs defined in queues.h
    IMU_Queue = xQueueCreate(1, sizeof(IMU_Data));
    Airspeed_Queue = xQueueCreate(1, sizeof(Airspeed_Data));
    GPS_Queue = xQueueCreate(1, sizeof(GPS_Data));
    Autopilot_Queue = xQueueCreate(1, sizeof(Autopilot_Data));
    Pitcheron_Queue = xQueueCreate(1, sizeof(Pitcheron_Data));

    Flight_Data_Queue = xQueueCreate(1, sizeof(Flight_Data));
}

=====
// File: semaphores.cpp
=====

#include "semaphores.h"

SemaphoreHandle_t I2C_MUTEX = NULL;
// SemaphoreHandle_t imu_done = NULL;
// SemaphoreHandle_t airspeed_done = NULL;
```

```
// SemaphoreHandle_t gps_done = NULL;

// Note: Mutex is a type of semaphore but with task ownership
void init_semaphores() {
    // Mutex
    I2C_MUTEX = xSemaphoreCreateMutex();

    // Binary Semaphores
    // imu_done = xSemaphoreCreateBinary();
    // airspeed_done = xSemaphoreCreateBinary();
    // gps_done = xSemaphoreCreateBinary();
}

=====
// File: sensors.cpp
=====
// This file contains all sensor related code functions to initialize and read from sensors.
#include <Arduino.h>
#include <Wire.h>
#include "pin_map.h"
#include "sensors.h"
#include "queues.h"
#include "semaphores.h"
/* Include Sensor Libraries */
#include <Adafruit_BNO08x.h>
#include <SparkFun_u-blox_GNSS_Arduino_Library.h> //Click here to get the library:
http://librarymanager/All#SparkFun\_u-blox\_GNSS
#include <MicroNMEA.h> //http://librarymanager/All#MicroNMEA

#define RHO 1.142363 //1.225 //kg/m^3 - from
https://www.omnicalculator.com/physics/air-density#what-is-the-density-of-air
#define ROLL_INVERTED true // true if PCB is inverted (top) relative to airplane

#define SERIAL_MONITOR_BAUDRATE 250000 // bits/sec
```

```
#define STARTUP_DELAY 2500 // ms x2
#define I2C_BUS_SPEED 400000 // 100kHz Default
#define MIN_AIRSPEED 5*3.28084 // m/s (pitot reads 0 if under 5m/s due to inaccuracy)
#define UTC_TIMEZONE_OFFSET -4 // EST is 4 hours behind UTC
#define GPS_SAMPLE_RATE 25 // Hz (25Hz max)
#define NMEA_BUFFER_SIZE 255
#define INIT_DELAY 100

/* Instantiate sensor classes and types */
// BNO085
Adafruit_BNO08x bno085(-1);
sh2_SensorValue_t bno085_value;
// GPS
SFE_UBLOX_GNSS myGNSS;
// Create buffer variables for NMEA Sentence Parsing
char nmeaBuffer[NMEA_BUFFER_SIZE];
MicroNMEA nmea(nmeaBuffer, sizeof(nmeaBuffer));

// Initialize Serial and I2C hardware
void init_low_level_hw() {
    // Startup Delay is blocking but that's ok.
    Serial.begin(SERIAL_MONITOR_BAUDRATE);
    delay(STARTUP_DELAY);
    Serial.println("\nESP32 DBF 2025 Payload X1 Glider RTOS Data Collection Software -
v3.4");
    Serial.println("By Daniel Noronha, Ricky Ortiz, and Matthew Zagrocki");
    Serial.println("Last Software Update: April 03, 2025");
    Serial.println("Wish Me Luck!!!\n");

    delay(STARTUP_DELAY);
    Wire.begin(SDA_PIN, SCL_PIN);
    Wire.setClock(I2C_BUS_SPEED);
    Serial.println("Serial IO & I2C Initialized Successfully!");
    pinMode(BUILTIN_LED_PIN, OUTPUT);
    digitalWrite(BUILTIN_LED_PIN, HIGH);
}
```

```

}

/* Sensor Initialization Functions */
// IMU
bool init_bno085() {
    Serial.print("Initializing BNO085 IMU...");
    // Reports Available: SH2_ACCELEROMETER, SH2_GYROSCOPE_CALIBRATED,
    SH2_MAGNETIC_FIELD_CALIBRATED,
    // SH2_LINEAR_ACCELERATION, SH2_GRAVITY, SH2_ROTATION_VECTOR,
    SH2_GEO_MAGNETIC_ROTATION_VECTOR,
    // SH2_GAME_ROTATION_VECTOR, SH2_STEP_COUNTER,
    SH2_STABILITY_CLASSIFIER, SH2_RAW_ACCELEROMETER,
    // SH2_RAW_GYROSCOPE, SH2_RAW_MAGNETOMETER, SH2_SHAKE_DETECTOR,
    SH2_PERSONAL_ACTIVITY_CLASSIFIER
    if (!bno085.begin_I2C()) {
        Serial.println("\nFailed to find BNO08x chip!");
        return false;
    }
    if (!bno085.enableReport(SH2_ROTATION_VECTOR)) {
        Serial.println("\nCould not enable rotation vector");
        return false;
    }
    if (!bno085.enableReport(SH2_LINEAR_ACCELERATION)) {
        Serial.println("\nCould not enable accelerometer (linear acceleration)");
        return false;
    }
    if (!bno085.enableReport(SH2_GRAVITY)) {
        Serial.println("\nCould not enable gravity vector output");
        return false;
    }
    if (!bno085.enableReport(SH2_GYROSCOPE_CALIBRATED)) {
        Serial.println("\nCould not enable gyroscope");
        return false;
    }
    if (!bno085.enableReport(SH2_MAGNETIC_FIELD_CALIBRATED)) {
        Serial.println("\nCould not enable magnetic field calibrated");
    }
}

```



```
        return false;
    }

    Serial.println("DONE!");
    return true;
}

// Differential Pressure Sensor (Pitot Tube Airspeed)
bool init_abp2() {
    Serial.print("Initializing ABP2 Differential Pressure/Airspeed Sensor...");
    Serial.println("DONE!");
    return true;
}

bool init_gps() {
    Serial.print("Initializing GPS...");
    // Starting communication with GPS (assume default I2C Address)
    if (!myGNSS.begin()) {
        Serial.println("\nError communicating with sensor!");
        return false;
    }

    myGNSS.setI2COutput(COM_TYPE_UBX | COM_TYPE_NMEA); //Set the I2C port to
    output both NMEA and UBX messages
    myGNSS.saveConfigSelective(VAL_CFG_SUBSEC_IOPORT); //Save (only) the
    communications port settings to flash and BBR
    myGNSS.setProcessNMEAMask(SFE_UBLOX_FILTER_NMEA_GGA |
    SFE_UBLOX_FILTER_NMEA_RMC); // We only want GGA and RMC NMEA Messages,
    ignore others
    myGNSS.setNavigationFrequency(GPS_SAMPLE_RATE); // 5 Hz originally

    Serial.println("DONE!");
    return true;
}

void init_all_sensors() {
```

```

while (!init_bno085()) {
    delay(INIT_DELAY);
    Serial.println("BNO085 IMU INITIALIZATION FAILED. RETRYING...");
}

while (!init_abp2()) {
    delay(INIT_DELAY);
    Serial.println("MS4525DO DIFFERENTIAL PRESSURE SENSOR INITIALIZATION
FAILED. RETRYING...");
}
while(!init_gps()) {
    delay(INIT_DELAY);
    Serial.println("GPS INITIALIZATION FAILED. RETRYING...");
}
Serial.println("All Sensors Initialized Successfully!");
}

void quat2eul (float re, float i, float j, float k, float* euler_angles, bool degrees=true) {
    float sqre = sq(re);
    float sqi = sq(i);
    float sqj = sq(j);
    float sqk = sq(k);
    // Note: re/real part = w; i,j,k are x,y,z in w+xi+yj+zk quaternion components
    if (degrees) {
        euler_angles[0] = RAD_TO_DEG * (asin(-2.0 * (i * k - j * re) / (sqi + sqj + sqk + sqre))); //
Pitch
        euler_angles[1] = RAD_TO_DEG * (atan2(2.0 * (j * k + i * re), (-sqi - sqj + sqk + sqre)));
// Roll
        euler_angles[2] = RAD_TO_DEG * (atan2(2.0 * (i * j + k * re), (sqi - sqj - sqk + sqre))); //
Yaw
    }
    else {
        euler_angles[0] = asin(-2.0 * (i * k - j * re) / (sqi + sqj + sqk + sqre)); // Pitch
        euler_angles[1] = atan2(2.0 * (j * k + i * re), (-sqi - sqj + sqk + sqre)); // Roll
        euler_angles[2] = atan2(2.0 * (i * j + k * re), (sqi - sqj - sqk + sqre)); // Yaw
    }
}

```

```

}

/* Sensor Reading Functions */
// IMU
void read_bno085(void* pvParameters) {
    // Initialize IMU Data struct
    IMU_Data new_imu_data;
    new_imu_data.sensor_id = 0;
    while(true) {
        //Serial.println("BNO Reading Task");
        bool rot_read = false;
        bool acc_read = false;
        bool grav_read = false;
        bool gyro_read = false;
        bool mag_read = false;
        int read_count = 0;

        while(read_count < 5) {
            xSemaphoreTake(I2C_MUTEX, portMAX_DELAY); // This is blocking.
xSemaphoreGive() is not
            // Try to get sensor data
            if (!bno085.getSensorEvent(&bno085_value)) {
                xSemaphoreGive(I2C_MUTEX);
                continue;
            }
            // Once data is obtained, find out which sensor it belongs to
            switch(bno085_value.sensorId) {
                case SH2_ROTATION_VECTOR:
                    // Only read data from a particular sensor once in the while loop
                    if(!rot_read) {
                        new_imu_data.rotation[0] = bno085_value.un.rotationVector.real; // w
                        new_imu_data.rotation[1] = bno085_value.un.rotationVector.i; // x
                        new_imu_data.rotation[2] = bno085_value.un.rotationVector.j; // y
                        new_imu_data.rotation[3] = bno085_value.un.rotationVector.k; // z
                    }
                }
            }
        }
    }
}

```

```

xSemaphoreGive(I2C_MUTEX);

float euler_vector[3] = {0.0,0.0,0.0};

quat2eul(new_imu_data.rotation[0],new_imu_data.rotation[1],new_imu_data.rotation[2],new_imu_data.rotation[3],euler_vector,true);
new_imu_data.euler[0] = -euler_vector[0]; // Pitch
if (ROLL_INVERTED) {
    if (euler_vector[1] > 0) euler_vector[1] -= 180;
    else euler_vector[1] += 180;
}
new_imu_data.euler[1] = euler_vector[1]; // Roll
new_imu_data.euler[2] = -euler_vector[2]+180; // Yaw (normalized from 0 to
360)
if (new_imu_data.euler[2] < 0) new_imu_data.euler[2] = 0;
else if (new_imu_data.euler[2] > 360) new_imu_data.euler[2] = 360; // This
matches GPS heading range.

    read_count++;
    rot_read = true;
}
else xSemaphoreGive(I2C_MUTEX);
break;
case SH2_LINEAR_ACCELERATION:
    // Only read data from a particular sensor once in the while loop
    if(!acc_read) {
        new_imu_data.lin_accel[0] = bno085_value.un.linearAcceleration.x * 3.28084; //
ft/s^2
        new_imu_data.lin_accel[1] = bno085_value.un.linearAcceleration.y * 3.28084; //
ft/s^2
        new_imu_data.lin_accel[2] = bno085_value.un.linearAcceleration.z * 3.28084; //
ft/s^2
        xSemaphoreGive(I2C_MUTEX);
        read_count++;
        acc_read = true;
    }

```

```

        else xSemaphoreGive(I2C_MUTEX);
        break;
    case SH2_GRAVITY:
        // Only read data from a particular sensor once in the while loop
        if(!grav_read) {
            new_imu_data.gravity[0] = bno085_value.un.gravity.x * 3.28084; // ft/s^2
            new_imu_data.gravity[1] = bno085_value.un.gravity.y * 3.28084; // ft/s^2
            new_imu_data.gravity[2] = bno085_value.un.gravity.z * 3.28084; // ft/s^2
            xSemaphoreGive(I2C_MUTEX);
            read_count++;
            grav_read = true;
        }
        else xSemaphoreGive(I2C_MUTEX);
        break;
    case SH2_GYROSCOPE_CALIBRATED:
        // Only read data from a particular sensor once in the while loop
        if(!gyro_read) {
            new_imu_data.gyro[0] = RAD_TO_DEG * bno085_value.un.gyroscope.x; // rad/s
            new_imu_data.gyro[1] = RAD_TO_DEG * bno085_value.un.gyroscope.y; // rad/s
            new_imu_data.gyro[2] = RAD_TO_DEG * bno085_value.un.gyroscope.z; // rad/s
            xSemaphoreGive(I2C_MUTEX);
            read_count++;
            gyro_read = true;
        }
        else xSemaphoreGive(I2C_MUTEX);
        break;
    case SH2_MAGNETIC_FIELD_CALIBRATED:
        // Only read data from a particular sensor once in the while loop
        if(!mag_read) {
            new_imu_data.magnetic[0] = bno085_value.un.magneticField.x; // uT
            new_imu_data.magnetic[1] = bno085_value.un.magneticField.y; // uT
            new_imu_data.magnetic[2] = bno085_value.un.magneticField.z; // uT

```

-> deg/s

-> deg/s

=> deg/s

```

        xSemaphoreGive(I2C_MUTEX);
        read_count++;
        mag_read = true;
    }
    else xSemaphoreGive(I2C_MUTEX);
    break;
default:
    xSemaphoreGive(I2C_MUTEX);
    break;
}
}
xQueueSend(IMU_Queue, &new_imu_data, portMAX_DELAY);
// xSemaphoreGive(imu_done);
vTaskSuspend(NULL); // Data logging task will resume this as soon as all data has been
logged.
// If queue sending fails, it will try again without suspending.
}
}

```

```
// ABP2DRRT001PD2A3XX
```

```

void read_abp2(void* pvParameters) {
    // Initialize Airspeed_Data struct
    Airspeed_Data new_airspeed_data;
    uint8_t id = 0x28; // i2c address
    uint8_t data[7]; // holds output data
    uint8_t cmd[3] = {0xAA, 0x00, 0x00}; // command to be sent

    // float outside_temp = 32; // in fahrenheit
    // float airpressure = 100000; // in pascals
    // float dewpoint = 28; // in fahrenheit
    // float relative_humidity = 0;

    float outputmax = 15099494; // output at maximum pressure [counts]
    float outputmin = 1677722; // output at minimum pressure [counts]
    float pmax = 1; // maximum value of pressure range in psi
}

```

```

float pmin = -1; // minimum value of pressure range in psi
float PSI_to_pascal = 6894.7572931783;

float percentage = 0; // holds percentage of full scale data

new_airspeed_data.sensor_id = 1;
while(true) {
    //Serial.println("Pitot Reading Task");
    xSemaphoreTake(I2C_MUTEX, portMAX_DELAY);

    Wire.beginTransmission(id);

    int stat = Wire.write (cmd, 3); // write command to the sensor
    stat |= Wire.endTransmission();
    vTaskDelay(pdMS_TO_TICKS(10));
    Wire.requestFrom(id, (uint8_t)7); // read back Sensor data 7 bytes
    int i = 0;
    for (i = 0; i < 7; i++) {
        data [i] = Wire.read();
    }

    xSemaphoreGive(I2C_MUTEX);

    float press_counts = data[3] + data[2] * 256 + data[1] * 65536; // calculate digital pressure
counts
    float temp_counts = data[6] + data[5] * 256 + data[4] * 65536; // calculate digital
temperature counts
    float temp_C = (temp_counts * 200 / 16777215) - 50; // calculate temperature in deg c

    //calculation of pressure value according to equation 2 of datasheet
    float pressure_PSI = (((press_counts - outputmin) * (pmax - pmin)) / (outputmax -
outputmin)) + pmin;
    float raw_diff_pressure = -pressure_PSI * PSI_to_pascal;
    float raw_airspeed = (sqrt(fabs(2 * raw_diff_pressure / RHO)))*3.28084;
    if (raw_diff_pressure < 0) raw_airspeed *= -1;

```

```

    float corr_airspeed = (raw_airspeed < MIN_AIRSPEED) ? 0.0:raw_airspeed; // Based on
    Calibration (airspeed inaccurate below ~5m/s)
    new_airspeed_data.diff_pressure = raw_diff_pressure;
    new_airspeed_data.airspeed[0] = raw_airspeed;
    new_airspeed_data.airspeed[1] = corr_airspeed;
    new_airspeed_data.temperature = temp_C;

    xQueueSend(Airspeed_Queue, &new_airspeed_data, portMAX_DELAY);
    // xSemaphoreGive(airspeed_done);
    vTaskSuspend(NULL); // Data logging task will resume this as soon as all data has been
    logged.
}
}

```

```

void read_gps(void* pvParameters) {
    // Initialize GPS_Data struct
    GPS_Data new_gps_data;
    bool first_fix = false;
    new_gps_data.sensor_id = 2;
    while(true) {
        //Serial.println("GPS Reading Task");
        xSemaphoreTake(I2C_MUTEX, portMAX_DELAY);
        myGNSS.checkUblox();
        xSemaphoreGive(I2C_MUTEX);
        // Fetch GPS data character by character
        if(!nmea.isValid()) {
            if (!first_fix) {
                new_gps_data.latitude = 0;
                new_gps_data.longitude = 0;
                new_gps_data.heading = 0;
                new_gps_data.gnd_speed = 0;
                new_gps_data.altitude = 0;
                new_gps_data.hours = 0;
                new_gps_data.minutes = 0;
            }
        }
    }
}

```



```

        new_gps_data.seconds = 0;
        new_gps_data.hundredths = 0;
        new_gps_data.satellites = 0;
        xQueueSend(GPS_Queue, &new_gps_data, portMAX_DELAY);
        // xSemaphoreGive(gps_done);
        vTaskSuspend(NULL);
    }
    //xQueueSend(GPS_Queue, &new_gps_data, portMAX_DELAY);
    //xSemaphoreGive(gps_done);
    //vTaskSuspend(NULL);
    continue;
}
if (!first_fix) {
    first_fix = true;
    Serial.printf("First GPS Fix Acquired! (in %f seconds)\n", ((float)micros())/1000000.0);
}
// Store NMEA parsed data (with consistent type-casting)
uint8_t num_sats = nmea.getNumSatellites(); // Can be int but makes queue implementation
much easier
if (num_sats < 1) continue; // Even if NMEA is valid, we do not want to send data with no
satellites (inaccurate)
    long alt_long;
    long heading_long;
    float latitude_mdeg = (float)nmea.getLatitude();
    float longitude_mdeg = (float)nmea.getLongitude();
    float heading = (float)nmea.getCourse();
    float gnd_speed = (float)nmea.getSpeed();
    bool altitude = nmea.getAltitude(alt_long);
    float alt = (float)alt_long;
    uint8_t hours = ((nmea.getHour() + UTC_TIMEZONE_OFFSET + 24) % 24); // EST is 4
hours behind UTC
    uint8_t minutes = nmea.getMinute();
    uint8_t seconds = nmea.getSecond();
    uint8_t hundredths = nmea.getHundredths();
    // Clear nmea buffer

```

```
nmea.clear(); // We already stored the data in variables above.
// Adjusting Entries!
latitude_mdeg = latitude_mdeg / 1000000;
longitude_mdeg = longitude_mdeg / 1000000;
gnd_speed = gnd_speed * (1.68781 / 1000); // Knots to ft/s
alt = (alt / 1000)*3.28084; // m to ft
heading = heading / 1000;

// Store Data in struct, then send to queue
new_gps_data.latitude = latitude_mdeg;
new_gps_data.longitude = longitude_mdeg;
new_gps_data.heading = heading;
new_gps_data.gnd_speed = gnd_speed;
new_gps_data.altitude = alt;
new_gps_data.hours = hours;
new_gps_data.minutes = minutes;
new_gps_data.seconds = seconds;
new_gps_data.hundredths = hundredths;
new_gps_data.satellites = num_sats;

xQueueSend(GPS_Queue, &new_gps_data, portMAX_DELAY);
// xSemaphoreGive(gps_done);
vTaskSuspend(NULL);
}

}

//This function gets called from the SparkFun u-blox Arduino Library
//As each NMEA character comes in you can specify what to do with it
//Useful for passing to other libraries like tinyGPS, MicroNMEA, or even
//a buffer, radio, etc.
void SFE_UBLOX_GNSS::processNMEA(char incoming)
{
    //Take the incoming char from the u-blox I2C port and pass it on to the MicroNMEA lib
    //for sentence cracking
```

```

    nmea.process(incoming);
}

```

```

=====
// File: strobe.cpp
=====

```

```

#include <Arduino.h>
#include "Adafruit_NeoPixel.h"
#include "strobe.h"
#include "pin_map.h"

```

```

#define NOT_NEOPIXEL false
#define BLINK_ON_TIME_ms 200
#define BLINK_OFF_TIME_ms 200
#define BLINK_RESET_TIME_ms 0

```

```

#define TEST_DELAY_ms 1000
#define LED_BRIGHTNESS_PERCENT 100 // %
#define NUM_LEDS 3

```

```

Adafruit_NeoPixel leds = Adafruit_NeoPixel(NUM_LEDS, STROBE_LED_PIN, NEO_GRB +
NEO_KHZ800);

```

```

void init_strobe() {
    if (NOT_NEOPIXEL) {
        // For Regular LEDs
        pinMode(STROBE_LED_PIN, OUTPUT);
        digitalWrite(STROBE_LED_PIN, HIGH);
        Serial.println("Testing LED. Confirm it lights up!");
        delay(TEST_DELAY_ms);
        //while(true);
        digitalWrite(STROBE_LED_PIN, LOW);
    }
}

```

```

    Serial.println("Strobe LED Initialized (ensure OFF).");
}
else {
    // For Addressable LEDs
    leds.begin();
    leds.setBrightness((uint8_t)((LED_BRIGHTNESS_PERCENT/100.0)*255));
    for (unsigned int i = 0; i < NUM_LEDS; i++) leds.setPixelColor(i, leds.Color(0, 0, 0));
    leds.show();
    if (NUM_LEDS == 1) Serial.println("Testing LED. Confirm it lights up!");
    else Serial.printf("Testing LEDs. Confirm all %d light up!\n", NUM_LEDS);
    delay(TEST_DELAY_ms);
    //while(true);
    for (unsigned int i = 0; i < NUM_LEDS; i++) leds.setPixelColor(i, leds.Color(179, 255, 0));
    leds.show();
    delay(TEST_DELAY_ms);
    for (unsigned int i = 0; i < NUM_LEDS; i++) leds.setPixelColor(i, leds.Color(0, 0, 0));
    leds.show();
    Serial.println("Strobe LEDs Initialized.");
}
}

void blink_strobe(void* pvParameters) {
    while(true) {
        // Serial.println("Strobe Task");
        // Pattern: BlinkBlink.....BlinkBlink.....
        if (NOT_NEOPIXEL) {
            // For Regular LEDs
            digitalWrite(STROBE_LED_PIN, HIGH);
            vTaskDelay(pdMS_TO_TICKS(BLINK_ON_TIME_ms));
            digitalWrite(STROBE_LED_PIN, LOW);
            vTaskDelay(pdMS_TO_TICKS(BLINK_OFF_TIME_ms));

            digitalWrite(STROBE_LED_PIN, HIGH);
            vTaskDelay(pdMS_TO_TICKS(BLINK_ON_TIME_ms));
            digitalWrite(STROBE_LED_PIN, LOW);
        }
    }
}

```

```

        vTaskDelay(pdMS_TO_TICKS(BLINK_OFF_TIME_ms));

        vTaskDelay(pdMS_TO_TICKS(BLINK_RESET_TIME_ms));
    }
    else {
        // For Addressable LEDs
        for (unsigned int i = 0; i < NUM_LEDS; i++) leds.setPixelColor(i, leds.Color(179, 255,
0));
        leds.show();
        vTaskDelay(pdMS_TO_TICKS(BLINK_ON_TIME_ms));
        for (unsigned int i = 0; i < NUM_LEDS; i++) leds.setPixelColor(i, leds.Color(0, 0, 0));
        leds.show();
        vTaskDelay(pdMS_TO_TICKS(BLINK_OFF_TIME_ms));
        for (unsigned int i = 0; i < NUM_LEDS; i++) leds.setPixelColor(i, leds.Color(179, 255,
0));
        leds.show();
        vTaskDelay(pdMS_TO_TICKS(BLINK_ON_TIME_ms));
        for (unsigned int i = 0; i < NUM_LEDS; i++) leds.setPixelColor(i, leds.Color(0, 0, 0));
        leds.show();
        vTaskDelay(pdMS_TO_TICKS(BLINK_OFF_TIME_ms));

        vTaskDelay(pdMS_TO_TICKS(BLINK_RESET_TIME_ms));
    }
}
}
}

```

```

=====
// File: tasks.cpp
=====

```

```

#include <Arduino.h>
#include "tasks.h"
#include "sensors.h"
#include "datalogger.h"
#include "autopilot.h"

```

```
#include "strobe.h"

#define COMMON_STACK_SIZE 4096 // bytes. All 4 tasks work with 3072, but not 2048, so
4096 chosen to give enough margin.
#define CPU0 0
#define CPU1 1

TaskHandle_t read_imu_task = NULL;
TaskHandle_t read_pitot_task = NULL;
TaskHandle_t read_gps_task = NULL;
TaskHandle_t log_data_task = NULL; // Currently Unused handle
TaskHandle_t autopilot_task = NULL;
TaskHandle_t strobe_task = NULL;

void init_tasks() {
    // 4 Tasks in total: read each of the three sensors and log the data
    xTaskCreatePinnedToCore(
        log_data,
        "Task to log Data to SD Card",
        COMMON_STACK_SIZE,
        NULL,
        2,
        &log_data_task,
        CPU1 // CPU 1 - Logging can happen independently of data collection to speed things up
        (separate processor)
    );

    Serial.println("SD Logging Task Started");

    xTaskCreatePinnedToCore(
        read_gps,
        "Task to read GPS Data",
        COMMON_STACK_SIZE,
        NULL,
        1,
```

```
    &read_gps_task,  
    CPU0 // CPU 0 (All sensors on same core since MUTEX needed for I2C bus anyway, also  
same priority.)  
);  
  
Serial.println("GPS Data Logging Task Started");  
  
xTaskCreatePinnedToCore(  
    read_bno085,  
    "Task to read IMU Data",  
    COMMON_STACK_SIZE,  
    NULL,  
    1,  
    &read_imu_task,  
    CPU0  
);  
  
Serial.println("IMU Data Logging Task Started");  
  
xTaskCreatePinnedToCore(  
    read_abp2,  
    "Task to read Pitot Tube (Airspeed) Data",  
    COMMON_STACK_SIZE,  
    NULL,  
    1,  
    &read_pitot_task,  
    CPU0  
);  
  
Serial.println("Pitot Tube Reading Task Started");  
  
xTaskCreatePinnedToCore(  
    Autopilot_MASTER,  
    "Full Autopilot (HDG, SPD, ROLL, PITCH) + ENV_PROT",  
    32768,
```

```

    NULL,
    2,
    &autopilot_task,
    CPU1
);

Serial.println("Autopilot Task Started");

xTaskCreatePinnedToCore(
    blink_strobe,
    "Strobe Light Blinking Task",
    COMMON_STACK_SIZE,
    NULL,
    1,
    &strobe_task,
    CPU1
);

Serial.println("Strobe Light Blinking Task Started");
}

=====
// File: trim_servos.cpp
=====

#include "trim_servos.h"
#include "pitcheron_servos.h"

// Mini-program for trimming servos if requested by setting TRIM_SERVOS flag
void trim_servos() {
    init_servos_trim();
    int trim_l = 0;
    int trim_r = 0;
    Serial.println("Servo Trim Program Running. Note down RAW_TRIM_L, RAW_TRIM_R,
and CW_CONVENTION needed to CENTER PITCHERONS.");

```



```

Serial.println("Reset/Reprogram with TRIM_SERVOS set to false to disable trim mode.
Then update RAW_TRIM flags in pitcheron_servos.cpp.");
Serial.println("Send all commands over Serial when prompted as Text (ASCII) with \\n (LF)
LINE TERMINATOR.");
Serial.println("Always use (int)DEGREES for any angles (+/-).");
actuate_servo_l(0);
actuate_servo_r(0);
Serial.printf("Current trim settings: trim_l = %ddeg, trim_r = %ddeg\\n", trim_l, trim_r);
while(true) {
    Serial.printf("Select Servo to Trim [L/R]: ");
    while (Serial.available() <= 0); // Wait for user input
    String user_input = Serial.readStringUntil('\\n');
    user_input.trim();

    if ((user_input == "L") || (user_input == "R")) {
        while(true) {
            Serial.printf("Enter angle in degrees for LEFT servo (range: %ddeg to %ddeg) or
press Enter to stop trimming: ", MIN_SERVO_ANGLE, MAX_SERVO_ANGLE);
            while (Serial.available() <= 0); // Wait for user input
            user_input = Serial.readStringUntil('\\n');
            user_input.trim();
            if (user_input == "") break;
            int user_trim = user_input.toInt();
            // Check limits
            if (user_trim < MIN_SERVO_ANGLE) {
                user_trim = MIN_SERVO_ANGLE;
                Serial.printf("WARNING: Trim value OUT OF RANGE (too LOW). Setting
trim_l to Minimum (%ddeg)\\n", MIN_SERVO_ANGLE);
            }
            else if (user_trim > MAX_SERVO_ANGLE) {
                user_trim = MAX_SERVO_ANGLE;
                Serial.printf("WARNING: Trim value OUT OF RANGE (too HIGH). Setting
trim_l to Maximum (%ddeg)\\n", MAX_SERVO_ANGLE);
            }
            trim_l = user_trim;
            actuate_servo_l(trim_l);

```

```

        Serial.printf("Current trim settings: trim_l = %ddeg, trim_r = %ddeg\n", trim_l,
trim_r);
    }
}

else if ((user_input == "R") || (user_input == "r")) {
    while(true) {
        Serial.printf("Enter angle in degrees for RIGHT servo (range: %ddeg to %ddeg) or
press Enter to stop trimming: ", MIN_SERVO_ANGLE, MAX_SERVO_ANGLE);
        while (Serial.available() <= 0); // Wait for user input
        user_input = Serial.readStringUntil('\n');
        user_input.trim();
        if (user_input == "") break;
        int user_trim = user_input.toInt();
        // Check limits
        if (user_trim < MIN_SERVO_ANGLE) {
            user_trim = MIN_SERVO_ANGLE;
            Serial.printf("WARNING: Trim value OUT OF RANGE (too LOW). Setting
trim_r to Minimum (%ddeg)\n", MIN_SERVO_ANGLE);
        }
        else if (user_trim > MAX_SERVO_ANGLE) {
            user_trim = MAX_SERVO_ANGLE;
            Serial.printf("WARNING: Trim value OUT OF RANGE (too HIGH). Setting
trim_r to Maximum (%ddeg)\n", MAX_SERVO_ANGLE);
        }
        trim_r = user_trim;
        actuate_servo_r(trim_r);
        Serial.printf("Current trim settings: trim_l = %ddeg, trim_r = %ddeg\n", trim_l,
trim_r);
    }
}

else {
    Serial.println("Invalid input provided! Try again and enter only 'L' or 'R'
(case-insensitive).");
}

```

Spring 2025

Ortiz, Noronha, Zagrocki

}
}