Project Caelus is a non-profit rocketry organization I helped start in high school. The team aims to be the first high school rocketry team to develop a liquid-fueled rocket engine and launch it to the edge of space (100km above the Earth’s surface). Over my three-year tenure with the team, I helped develop the engine test stand, a testing platform that allowed us to validate our engine design, flight software, and electrical systems.

In particular, as the head of the Avionics sub-team, I led a team of four to develop an avionics system that could support the processing needs of our flight software; make accurate pressure, temperature, and fuel measurements; and automatically toggle various solenoid valves to control the fuel/oxidizer outputs to the engine. The system also needed to be able to allow human operators to manually override the flight software and abort from a safe distance.

**Design Overview**

At the heart of the system is a two-layer copper PCB that hosts a Teensy 3.6 flight computer running C++ flight software. The system can simultaneously:

* Take measurements from:
  + 12 Pressure Transducers
  + 5 Thermocouples (temperature sensors)
  + 3 Load Cells (force/weight sensors)
* Actuate up to 12 solenoid valves
* Transmit all sensor measurements/valve states/additional telemetry to ground station
* Provide hardware override for ground station operators in need of abort

With the onboard batteries, the system could be powered continuously for up to three hours, which was necessary for long-duration tests. All parts were conformal coated, protecting them from dust, moisture, and temperature extremes.

**Design Process**

After receiving the design requirements and constraints from the Propulsion team, the Avionics team got to work designing a schematic for the system in KiCAD’s Schematic Designer.

Pressure Transducers (PTs)

Each PT provides three wires: Red (5V+), Black (GND), and Green (Signal). The sensor outputs a 5V signal that needs to be attenuated to a 3.3V signal for our Teensy 3.6 microcontroller. To achieve this, every signal input is routed through a voltage divider that cuts the signal by a third. Due to high power losses across the resistors, future iterations will utilize Inverting Op-Amps with a fractional gain to achieve the attenuated signal.

Thermocouples

Each K-type, ungrounded thermocouple requires an amplifier to make the millivolt output readable by the Teensy 3.6 flight computer. For this, we chose the Adafruit MAX31856 amplifier due to its compatibility, reliability, and affordability. The MAX31856 communicates with the FC via the SPI serial protocol.

Load Cells

Load cells are essentially electronic strain gauges, so they can be used to measure the weight of our tanks and the thrust generated by our engine. Similar to the thermocouples, each load cell outputs a millivolt signal which needs to be amplified for the FC. For this, we use the SparkFun HX711 amplifier which communicates with the FC via serial communication.

Solenoid Actuation

Solenoid valves are relatively simple devices – internally, they contain a coil of wire attached to a plunger that opens and closes the valve when a current is applied. The Teensy 3.6 cannot supply enough current to actuate these valves; thus, the FC controlled Arduino Relays, which supplied power from our 6S Lipos to open/close the valves. In future iteratios, the Arduino Relays will be replaced by power transistors which consume less power during operation.

However, to allow for groundstation override capabilities via the Launchbox (explained later), each relay needs to be connected to both the FC and the groundstation. To achieve this, every relay signal line was attached to a DG419 IC (essentially an electronic switch) that allows the relay to be controlled by both input sources. When the Launchbox is enabled, the enable pin for each DG419 is toggled HIGH, thereby connecting each relay to the groundstation and allowing for manual override.

Data Transmission

All sensor measurements, valve states, and flight software variables are transmitted to the ground station via an Xbee radio. Data packets are sent 10 times a second, allowing for ground station operators to get a comprehensive view of the state of the entire system at any moment.

Power Distribution Board (PDB)

As mentioned in the Solenoid Actuation section, every solenoid valve needs to be powered via an external battery source (in our case, a 6S lipo). The Power Distribution Board (PDB) provides multiple 24V and 12V terminals to power the various solenoids on the test stand. The board is powered via a 6S Lipo (24V battery) that runs through an external step-down converter to power the 12V terminals. Future iterations of the board will include built-in XT-60 connectors instead of solder pads to allow for better wiring organization and easier connections.

After developing the schematic, laying out the footprints in KiCAD’s Layout Editor, drawing the copper traces, and generating the gerber files, we sent our plans out to be manufactured by JLCPCB.

**Testing**

As part of my high school Senior Research project, some friends and I wanted to build a SpaceX-like rocket that could autonomously hover and land. Though our formal reason for undertaking this project, codenamed Ascent, was to learn about control algorithms and mechanical design (which was true), the real reason was because we thought this was a dang-cool project 😊.

**Design Overview**

The goal of this project was to create an electrically powered rocket that could hover to 5m and autonomously land. The design consisted of two counter-rotating propellers that produced a max thrust of roughly 2kg. Thus, to achieve a 2:1 thrust-to-weight ratio, the mass of the craft was limited to at most 1kg.

To control the craft in the pitch and yaw axes, we decided to use a gimballed nozzle. Though we explored using servo-actuated thrust vanes (fins placed in the airstream that deflect thrust), we found that the thrust losses from these actuators would be too high to sustain for our small craft. Additionally, throughout our preliminary design review, we did not find any hobby or research projects that took on the challenge of a gimballed nozzle; thus, we decided to try it for ourselves.

**Mechanical Design**

The main hardware components of this project were the gimballed nozzle and the frame. Each was first designed in Fusion 360 CAD then manufactured primarily through 3D-printing.

Gimbal

Our gimbal design was primarily inspired by the solid-motor gimbal designed by Joe Barnard at BPS.Space. The design consists of two concentric rings that allow the converging nozzle to be actuated in the craft’s yaw and pitch axes. Two 9g servos actuate the nozzle in both axes using pushrods. This allows for a maximum gimbal rotation of &#177; 15 degrees in each axis. Four small skateboard bearings were included to allow for smooth actuation of each ring. The total weight of the gimballed nozzle, including the servos, screws, bearings, and nozzle, was roughly 180g, 20 grams lower than our mass budget had initially allotted for.

Frame

The frame had to be designed to be durable but also easily manufactured in the inevitable case of a crash landing (or as SpaceX fans call it, a Rapid Unscheduled Disassembly – RUD for short). Thus, we built the frame out of strong, lightweight, cheap 5mm carbon fiber tubes. These were cut on a bandsaw to the right dimensions then attached together in a rectangular prism through easy-to-make, 3D-printed joints. We added styrofoam to the ends of the landing legs to absorb impact upon landing.

**Control Systems**

Simulink is an industry-standard software used to design and analyze all sorts of control systems. The high-level control system can be divided into three main components: PID Feedback, Environment Modelling, and Craft Modelling.

PID Feedback Block

Our main control system was built-upon three PID Feedback loops, one for each of the pitch, yaw, and roll axes. The onboard IMU provides position data in all 9 degrees of freedom; thus, the error term for each feedback loop is calculated by subtracting the actual rotational position from the expected position. This error is fed into the PID, which outputs the necessary torque needed in each axis to correct the rotation. These torques are fed into a physics block that applies kinematic equations to find the required nozzle angles needed to cancel all rotations.

Environment Block

Our custom environment block utilizes built-in atmospheric models in Simulink to predict how wind would affect the performance of our craft. Using the COESA Atmosphere block, which outputs measurements based on a standard atmospheric model, as well as a Wind Shear simulator for each axis, we were able to compute the net forces and moments on the craft caused by external factors.

Craft Modelling (Plant)

The craft was modelled using a 6DOF physics block, which is built into Simulink. The block takes in a matrix of the net forces and moments on the craft and outputs the resulting linear and angular positions of the craft. Of particular interest to us was the Euler angle outputs, which tell us the rotation of the craft in each of the three axes. These are fed back into the PID Feedback Block to compute the error terms and the cycle restarts.

**Kalman Filter**

One of my personal goals for this project was to understand how Kalman Filters work, a standard algorithm used to remove noise from sensor readings. To the right is a Python testing platform we created to test our filter. The code simulates data from our IMU, adding in Gaussian noise, and runs these through our Kalman Filter. The graphical and statistical outputs let us easily see the effectiveness of our algorithm.

**C++ Flight Software**

We chose to write our flight software in C++ as it is a compiled language that can easily be uploaded to our Arduino Teensy flight computer. Our flight software was based on a Read-Control-Actuate architecture, which employs “tasks” to execute different functions:

* Read Tasks – Gather and store all sensor information and groundstation commands
* Control Tasks – Utilize data to make decisions and update variables
* Actuate Tasks – Execute decisions made by Control Tasks

These tasks are run sequentially in a single thread by the MainControlLoop, which essentially just calls the <code> .execute </code> of each task. All sensor data and decision variables are stored inside the StateFieldRegistry, which contains specific fields that can only be edited by specific tasks.

This project is still in the works. Stay tuned to see it fly!

**Custom Drone – Mark II**

If you want to see my very first drone build, check it out here

After creating my first drone using off-the-shelf parts and open-source software, I knew there was still a lot to learn. I still didn’t understand PID Feedback or the architecture of drone flight software. Therefore, I decided to build a completely custom drone with a Raspberry Pi 3 (later changed to Arduino Teensy) running my own custom flight software.

**Mechanical Design**

The drone’s frame was designed in Fusion 360 and 3D-printed using PLA plastic. I used four D2212 motors outfitted with generic 5-inch propellers for a maximum thrust of roughly 2.5kg. I decided to use cheap PWM Electronic Speed Controllers to control the motors, as I was running tight on my budget, and they could easily interface with the PWM outputs on my Raspberry Pi 3 flight computer. The onboard IMU (Adafruit BNO005) provided real-time angular positions that could be read by the flight computer. All of this was powered by a 3S, 2200mAh LiPo battery that had a theoretical battery life of 10-12 minutes.