The Design of a Modular Avionics System for Spaceshot Liquid Rockets

Cody Kaminsky[*], Tymur Tkachenko[†], Jeff Shelton[‡], Sasha Callaway[§], Vineet Kulkarni[¶], and Aras Shirwan[‖]
Georgia Institute of Technology, Atlanta, Georgia, 30332

Avionics systems in high-performance collegiate liquid rocketry are crucial for data acquisition, control, and recovery. However, these systems are often custom-built for each vehicle, requiring repetitive research and development, which extends timelines and leads to decreased testing and reliability. This paper discusses the design of a modular and customizable avionics system developed by the Yellow Jacket Space Program, which includes both flight hardware, ground hardware, and software components. The Yellow Jacket Space Program simultaneously operates two vehicle programs and an engine development program. The goal of streamlining avionics development led to the creation of a modular, easily revisable, and functionally efficient avionics system. The system is divided into a flight system and a ground system; the flight system focuses on controlling the propulsion system, acquiring vehicle sensor data, estimating its position and heading, and providing a recovery signal upon vehicle landing. The ground system allows operators to observe and control vehicle states by receiving live data from sensors and controlling valves. Central to the Yellow Jacket Space Program (YJSP) avionics is the System Avionics Module (SAM) printed circuit board assembly (PCBA), which provides localized valve control and data acquisition of current loop sensors, thermocouples, resistance temperature detectors, and differential sensors, transmitting live data to a custom GUI operated in mission control. The flight system utilizes the SAM, along with five other custom PCBAs, to meet the functional requirements of the vehicles. All components in the system are programmed using Rust to create memory safe code and eliminate unexpected errors. Devices are programmed with custom firmware to enable efficient data acquisition and guarantee expected performance. The avionics system topology discussed in this paper allows for rapid testing and revision of individual components while enabling YJSP to adapt to future vehicle programs regardless of new requirements.

I. Introduction

The Yellow Jacket Space Program (YJSP) is a motivated collegiate team at the Georgia Institute of Technology, with the goal to become the first group to launch a liquid-fueled rocket into space. Our first major achievement was Goldilox, a pressure-fed kerosene and liquid oxygen rocket, launched in January 2023. Goldilox reached an apogee of 1,508 meters and served as a proof-of-concept, providing valuable insights and experience for the development of our next pressure-fed rocket, Vespula. During the development of Goldilox, we also introduced Darcy I, a simpler liquid propulsion system based on Half Cat's nitrous oxide and isopropyl alcohol rocket. Launched in July 2022, Darcy I reached an estimated apogee of 6,096 meters. Building on the successes of Darcy I, Darcy II was designed to be a record-breaking vehicle. It was successfully launched in 2023 claiming the collegiate rocketry altitude record of 9,198 meters. After Goldilox and Darcy II launched, there was an increased desire to design better engines for future higher performing liquid rockets. This resulted in the design of the Helluva Engine Test Stand (HETS) program.

Leveraging the knowledge gained from Goldilox and Darcy II, YJSP aimed to streamline the design process to create faster and more reliable systems. This goal extended to the Avionics sub-team as well. Throughout the design, integration, and flight of Darcy II and Goldilox there were a number of issues identified that the avionics team sought to improve. The most significant challenge was the lack of testing and reliability in the different parts of the system. This challenge is particularly hard for a student team where the team sizes are small and commitment levels can vary.

[*]Vespula Avionics Lead, Yellow Jacket Space Program, AIAA Graduate Student Member
[†]Darcy Avionics Lead, Yellow Jacket Space Program, AIAA Undergraduate Student Member
[‡]Director of Avionics, Yellow Jacket Space Program, AIAA Undergraduate Student Member
[§]Helluva Engine Test Stand Avionics Lead, Yellow Jacket Space Program, AIAA Undergraduate Student Member
[¶]Software Lead, Yellow Jacket Space Program, AIAA Undergraduate Student Member
[‖]Flight Computer Responsible Engineer, Yellow Jacket Space Program, AIAA Undergraduate Student Member

Furthermore, the previous software architecture was hardware-specific, lacked configuration, and placed constraints on the operator's experience regarding viewing data and actuating valves simultaneously. Given these constraints, the decision was made to design a modular avionics system that reused components as much as possible. The goal was for common parts to be used across all three systems, increasing reliability and reducing development time. A secondary problem was over-reliance on a few individuals to complete the system. We aimed for the modularity of our avionics system to address this problem as well. Each board designed had minimal inter dependencies, allowing individual components to be modified without impacting the design of other components. To achieve modularity in the ways mentioned, we distilled the requirements of the systems down to their essential parts.

This paper will discuss the proposed avionics design for the new engine test stand, our second kero-lox vehicle, and the third Darcy vehicle. The focus will be on the advantages of designing our system with modularity and how that affected the three different programs. The system designs shown are proposed as examples for how an avionics system can be developed for liquid spaceshot rockets, such as Darcy Space and the successor to Vespula.

II. Proposed System Overview

Each of YJSP's three programs-HETS, Darcy, and Vespula-requires distinct functionality, leading to variations in system implementation. Our modular system allows us to reuse components across all three programs while adapting them to meet specific mission requirements through configuration and system integration. The role of our avionics systems in each of our programs is outlined below.

A. Helluva Engine Test Stand

The Helluva Engine Test Stand (HETS), is YJSP's transportable engine test stand. HETS is a pressure-fed system built around a trailer which houses the fluids and pneumatics systems connected to a ground-anchored thrust structure upon which the engine is mounted. The primary operational goal of the test stand is the rapid and safe characterization of YJSP's engines. The avionics system supporting HETS requires localized data acquisition of sensors, valve actuation, and control of various other systems such as a fuel pump, igniters, and cameras. The custom component developed for this is the System Avionics Module (SAM), a modular board that supports pressure transducers, thermocouples, load cells, differential pressure sensors, relays, and pneumatic valves. Avionics are remotely monitored and controlled in the Mission Control (MC) trailer located a safe distance away. The MC trailer hosts the ground computer, the control server, and the operator's Graphical User Interface (GUI), which are explained in the Software Implementation section of this paper.

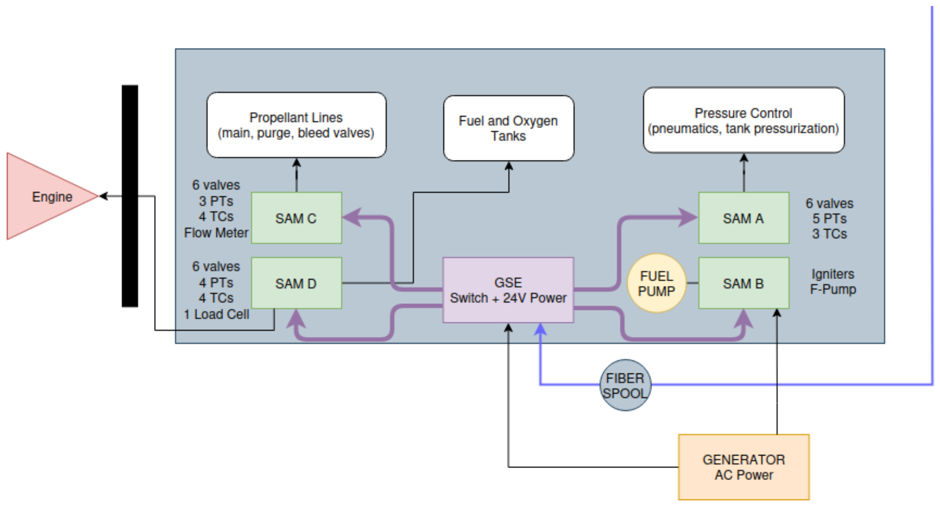


Fig. 1 HETS Avionics System Diagram

For a spatially distributed system like HETS, having SAMs in close proximity to their valves and sensors allows for shorter harness lengths and localized control. With HETS being an outdoor test stand with continued incremental changes and upgrades, having a modular system allows for increased flexibility and ease of replacement in the case that a SAM gets damaged. Replacement or addition of an SAM simply requires mapping the valves and sensors in the GUI to the new SAM and connecting it to power and the network. All SAMs are connected to a Ground Support Equipment (GSE) box via an Ethernet connection and DC power. The GSE box features power distribution and a network switch that connects all SAMs and IP cameras to the MC trailer over a fiber optic connection. The MC trailer has four computers, two dedicated to the GUI operated by engine test stand operators, a computer for IP cameras positioned around the stand, and one computer to run the ground computer, control server, and all connections to the SAMs.

Since the test stand is designed for engine characterization, the avionics system must support a variety of measurement projects. Such projects include heat flux, calorimetry, and different temperature measurements within the engine and injectors. Having a modular system that allows for the quick addition of sensor capabilities is invaluable for better engine understanding and measurement. To give a specific example, the Torch Igniter is an in-progress engine development project that requires 2 additional valves and several sensors. In order to support the testing of this project as well as integration onto HETS, the changes needed from the avionics side simply necessitates adding a singular new SAM and harnessing.

B. Darcy Space: Nitrous Oxide and IPA

The Darcy vehicle program retains the ground testing necessities of HETS with the addition of a flight avionics system. The Darcy architecture does not require any avionics control over propulsive system elements, allowing the avionics to be entirely contained within the avionics bay located in the nosecone. The goal of the system is to enable accurate data collection during flight, consistent deployment of parachutes, and reliable vehicle tracking after landing. The current Darcy vehicle implements the common flight avionics stack, greatly decreasing the risk of an unexpected and untested failure during flight.

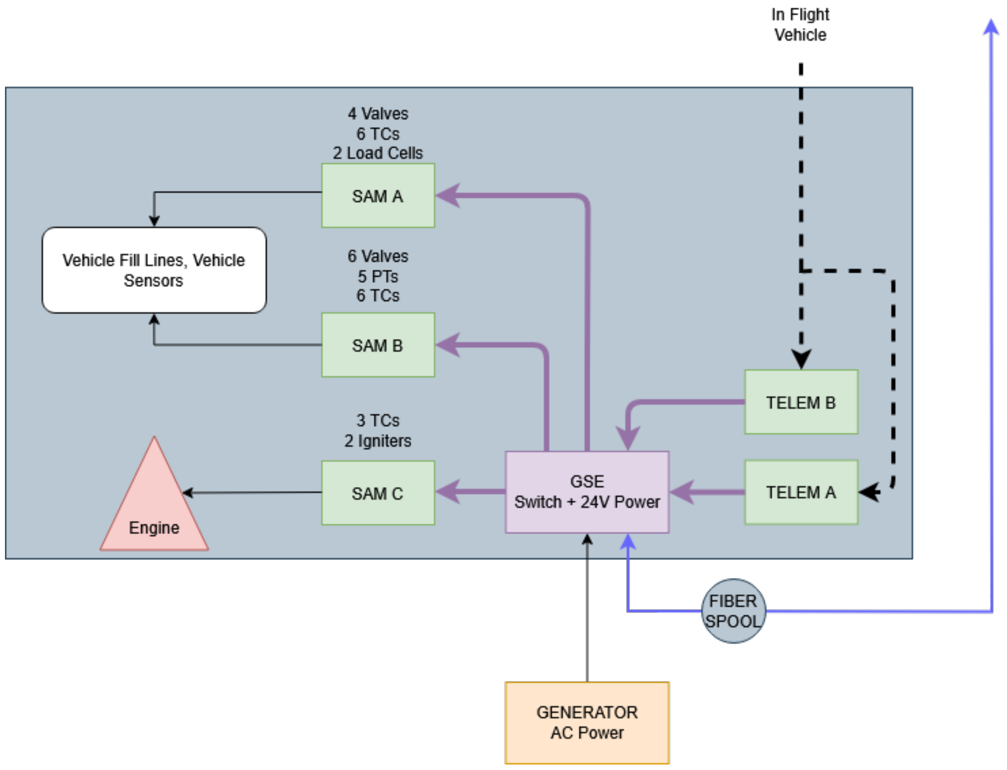


Fig. 2 Diagram of the Vehicle Ground System

The Darcy flight avionics system consists of the Battery Management System (BMS), Blackbox (BBX), Recovery (RECO), Attitude Heading and Reference System (AHRS), and Telemetry (TELEM). These sub-components were all chosen based on the functional requirements for the vehicles we had in mind. The BMS is primary responsible for the managing of the lithium-ion cell-based battery pack and distributing protected power to the rest of the system using load-switches. BBX is the node that houses the Flight Computer (FC) and is responsible for storing all flight data and relaying vehicle location information back to the ground station via the Iridium satellite constellation. RECO is designed as a single fault-tolerant system that determines the position of the rocket during flight with a Kalman filter and deploys parachutes at the proper times. AHRS has a high fidelity IMU, barometer, and magnetometer to collect data that can be analyzed after a launch of the vehicle. Lastly, TELEM is the board that has two RF bands and a GPS to deliver real time data to the ground station and to collect vehicle location data during its flight. The functional requirements were separated into these different boards so that each one could be worked on independently from each other with the only connections between them being power or communications.

The ground avionics and the flight avionics are on separate networks, only interacting through transmissions over telemetry. The ground system utilizes the SAM boards in a similar way to HETS. The GSE boxes containing the SAM boards are used to be able to rapidly connect pressure transducers, valves, load cells, and thermocouples, and to be able to control the system from the mission control trailer. The flight avionics peripherals are connected as nodes in a star topology network system, all being connected to an Ethernet switch and allowing for the seamless addition of new system components.

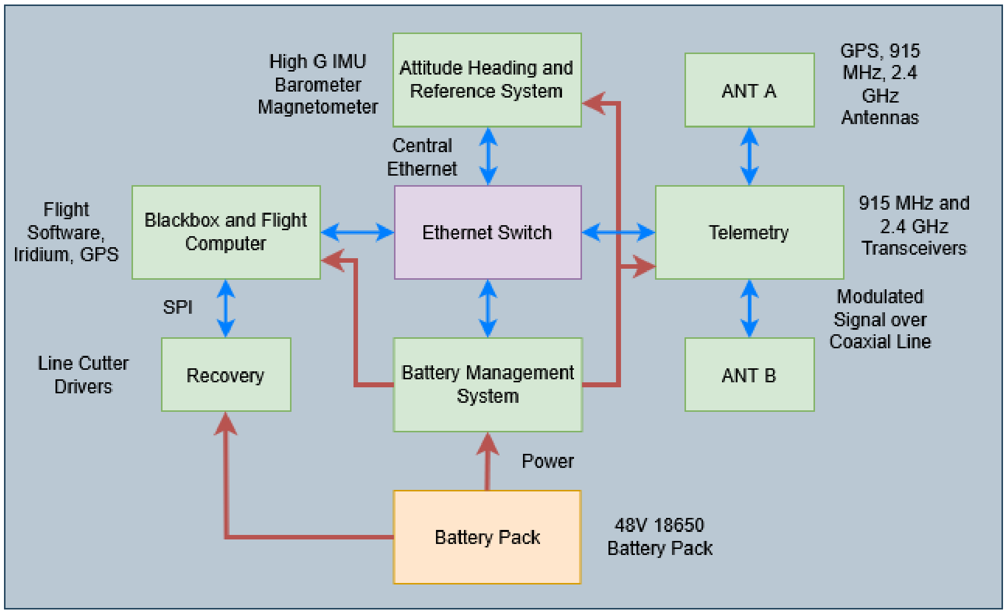

High G IMU
Barometer
Magnetometer
Attitude Heading and Reference System
ANT A
GPS, 915 MHz, 2.4 GHz Antennas
Central Ethernet
Flight Software, Iridium, GPS
Blackbox and Flight Computer
Ethernet Switch
Telemetry
915 MHz and 2.4 GHz Transceivers
SPI
Modulated Signal over Coaxial Line
Line Cutter Drivers
Recovery
Battery Management System
ANT B
Power
Battery Pack
48V 18650 Battery Pack

Fig. 3 Diagram of the Flight Avionics Bay

C. Vespula

The Vespula program avionics requirements include those from both HETs and Darcy while incorporating additional functionality. The vehicle avionics shares the five sub-components that Darcy uses and stores them below the rocket's nose cone. Allowing for compatibility between both Darcy's and Vespula's avionics systems made the development process smoother in that we only had to develop a single avionics system instead of two. The vehicle portion of the system has the unique challenge of requiring localized data acquisition of sensor data and control of various solenoid valves. This is accomplished with the flight version of the SAM that is smaller and optimized for a flight configuration. The hardware design is identical between the flight version and ground versions except for the connectors and some additional requirements for the flight version. This made it so that the software written for the ground SAMs could be seamlessly adapted for the flight SAMs further decreasing the development time due to modularization. Shown in Figure 4, the three flight SAMs are positioned in each of the inter-tanks which was done so that the wiring harness going down the rocket was reduced to power and communications. This resulted in a simpler integration and faster debugging of issues.

The nodes of the Vespula Avionics system are connected on a shared Ethernet network where each node has a statically assigned IP based on the type of node and its identification number. On the flight system, there is one Ethernet switch dedicated to the three flight SAM boards which lives in the second inter-tank and there is one Ethernet switch in the avionics bay that connects the flight SAM switch, avionics bay boards, and provides a connection to the ground system. The ground system network has one switch that bridges the ground SAMs, flight system, and our mission control trailer. This is one difference to the Darcy Avionics in that Vespula requires the ability to control the vehicle propulsion system on the ground.

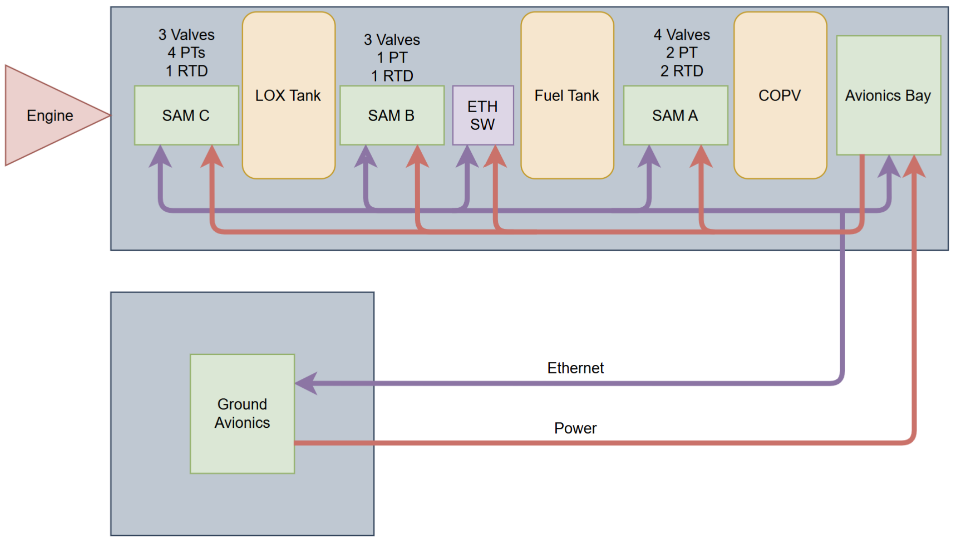


Fig. 4 Vespula Vehicle Avionics System Diagram

Reducing risk by applying modular hardware from different programs was the key to creating a solution that met the goals of the Vespula program. At the same time, developing a complementary software system to work with our modular hardware was critical to the success of our programs.

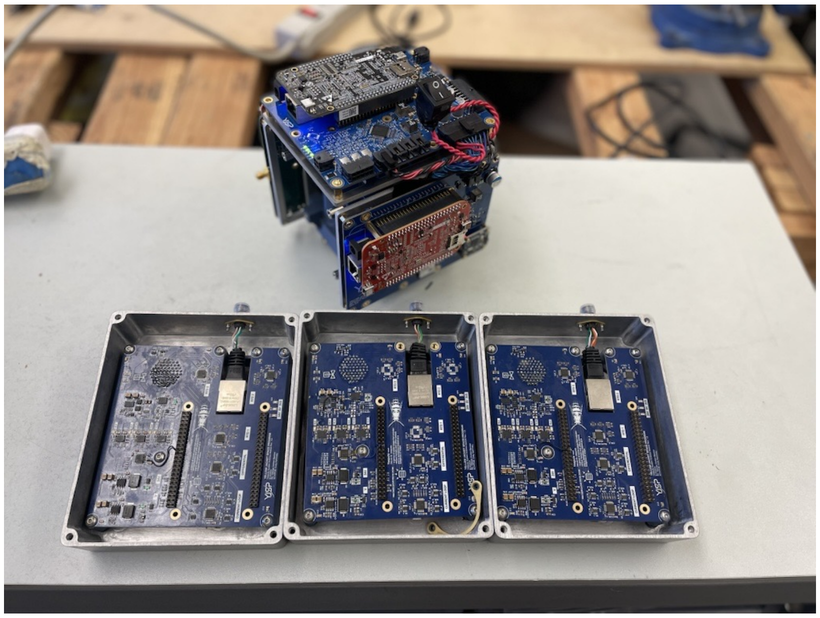


Fig. 5 Vesupla SAM boards alongside the Avionics Bay holding the Flight Computer, AHRS, BMS, RECO and TELEM.

III. Software Implementation

The software stack for our avionics system is designed to provide seamless control and monitoring over all mission-critical components. It consists of three major software nodes (Flight Computer, Control Server, and GUI) that communicate over a networked architecture. Each node is explained in detail below.

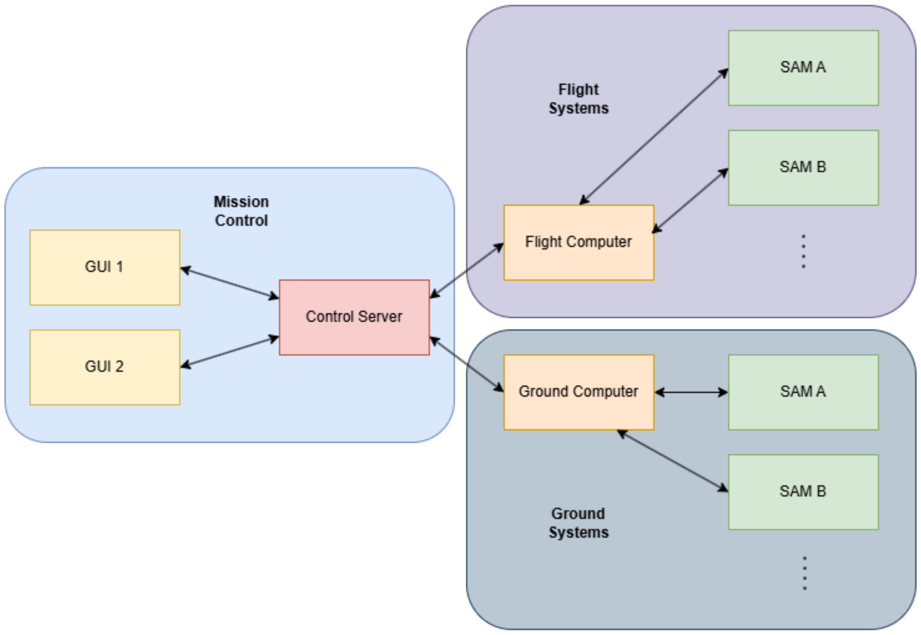


Fig. 6 Software Architecture for Mission Control, Flight, and Ground Systems

A. Flight Computer

The Flight Computer (FC) is a computer program written in the Rust programming language and designed for use on Unix-based operating systems. Some responsibilities include, but are not limited to: centralizing communication between the control server and the vehicle's hardware subsystems, maintaining vehicle software integrity, and executing sequences sent from the control server.

The FC centralizes communication by receiving sensor data from all registered subsystems, processing and synthesizing the data into a single message, and sending the message to the control server. This entire process is conducted at a constant rate. Any subsystem that connects to the FC can begin sending sensor data, without needing to coordinate with the other registered subsystems. If a subsystem stops communicating, the FC will deregister the subsystem and execute the relevant logic to handle the disconnection. The entire connection and disconnection process is automated, and was designed from the ground up to be modular and hands-off.

Part of the goal of the FC is to abstract the individual components of the vehicle into a single entity. Operators can interact with the vehicle through the FC with a single mechanism known as sequences. A sequence is a script of Python code that has access to all the valves and sensors on the vehicle (see Appendix A). Operators can then write closed-loop control procedures using available vehicle data, all within the Python language. The operators do not need to concern themselves with the specific subsystems connected to the FC. As long as those subsystems have access to the valves and sensors used by the sequence, the sequence can be reused as many times as necessary without modification.

Additionally, the FC software is hardware-agnostic; so long as the FC has access to the network and the underlying operating system is Unix-like, the FC will work without requiring modifications to the source code. This allows the same FC software to run on both the physical onboard Flight Computer, such as when its running on the Vespula rocket, or on the physical Ground Computer, such as when engines are tested during hot fires. Figure 6 demonstrates this modularity.

B. Control Server

The control server is the central device which coordinates actions and data between all other parts of the system. It maintains a direct connection to the flight and ground computers, as well as direct HTTP connections to all operator GUI computers. The control server translates the human-readable operator commands from the GUI computers into specific, actionable commands for hardware and forwards them to the flight and/or ground computer. It is also responsible for logging all data originating from the flight and ground computers to a database so that it can be analyzed later. The control server is a crucial component for our modular architecture because it is the one component that knows about all others and is positioned to connect the operators to the Vehicle system. The control server can support an arbitrary number of operator GUIs, receiving commands and forwarding out live data to all of them at once. It supports simultaneous connection of a flight computer in the vehicle and a ground computer controlling ground support equipment. It does so by relaying operator commands to the proper destination.

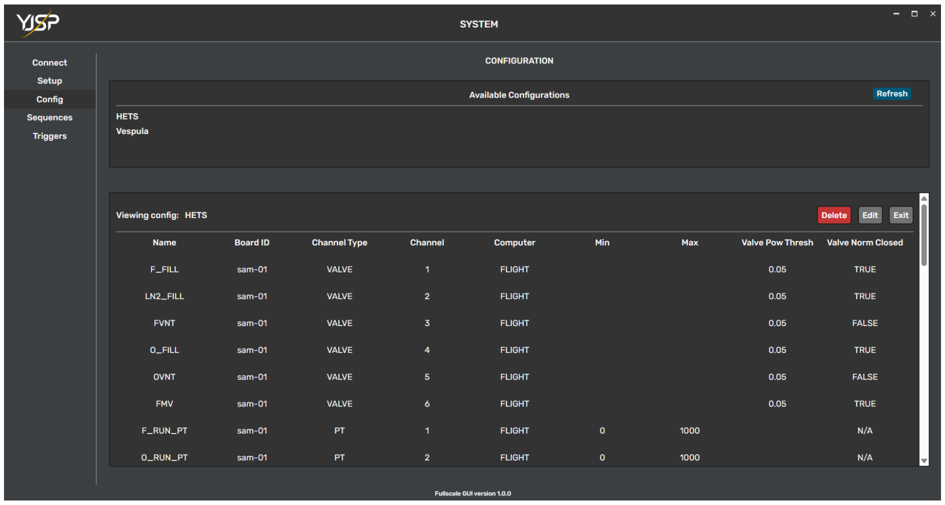
The architecture of the control server software is much like that of a traditional web server, and it uses several web technologies to function. It uses a Rust application that hosts an HTTP server and exposes several HTTP routes for commands, data streaming, setting mappings, and performing administrative actions. All messages are passed in human-readable JSON format between the control server and GUI. This software design decision improves the modularity and the ability to debug the system, as Avionics operators can use ordinary networking tools to inspect, modify, and patch the system with relative ease in a human-readable form.

The control server has another element of its architecture which connects to the ground and/or flight computers. The flight computer automatically detects the existence of the control server by locating its network hostname and attempting to connect. It has a predefined list of hostnames that it cycles through until a connection is formed. As a consequence of this auto-discovery, we can adopt different network topologies for each Vehicle's networking infrastructure, and yet the ground server and flight computer always find a path to each other with no extra configuration required.

C. Graphical User Interface (GUI)

The GUI serves as the primary interface for operators to interact with the system, providing real-time control, monitoring, and configuration management for valves, sensors, and sequences. Designed for both ease of use and operational flexibility, the GUI ensures seamless management of avionics systems across all programs, including HETS, Vespula, and Darcy. Built using a Rust-based backend for robust state management and a TypeScript + Solid.js frontend, the GUI leverages the Tauri framework to enable efficient, lightweight, and native application deployment across multiple operating systems.

A key feature of the GUI is its multi-window architecture, which allows operators to configure their workspace dynamically. Users can open multiple windows to display specific system views, such as real-time sensor data, valve states, and sequence execution. This adaptability ensures that each operator can tailor their interface based on their specific role and responsibilities, enhancing situational awareness and workflow efficiency.



HETS
Vespula

Viewing config: HETS

Name	Board ID	Channel Type	Channel	Computer	Min	Max	Valve Pow Thresh	Valve Norm Closed
F_FILL	sam-01	VALVE	1	FLIGHT			0.05	TRUE
LN2_FILL	sam-01	VALVE	2	FLIGHT			0.05	TRUE
FVNT	sam-01	VALVE	3	FLIGHT			0.05	FALSE
O_FILL	sam-01	VALVE	4	FLIGHT			0.05	TRUE
OVNT	sam-01	VALVE	5	FLIGHT			0.05	FALSE
FMV	sam-01	VALVE	6	FLIGHT			0.05	TRUE
F_RUN_PT	sam-01	PT	1	FLIGHT	0	1000		N/A
O_RUN_PT	sam-01	PT	2	FLIGHT	0	1000		N/A

Fig. 7 Configuration Viewing and Editing Page in the GUI

The GUI also provides extensive configuration and sequence editing capabilities, enabling operators to define and modify sensor and valve mappings and automate control logic through sequences as needed. These configurations ensure that each mission setup is precisely tailored to its operational requirements, allowing for quick adaptation between different test environments. The GUI communicates with the control server over Ethernet, sending commands as HTTP requests and receiving real-time data through a webstream, a simple communication setup allowing for reliable operation. With the ability to run multiple instances of the GUI that can connect to various instances of the control server, our GUI offers a scalable control system, making it a powerful tool for modular liquid rocketry avionics operations.

IV. Conclusion

The modularity that we built into the system was able to solve the issues we had with our previous avionics systems. Despite having limited resources in people and time, the above avionics systems have been designed and implemented in the last two and a half years. Starting with the HETS avionics system, a lot of experience was gained with the SAM boards, Control Server, Ground Computer, and GUI which culminated in ten engine hot fires over the course of four weeks in the Spring of 2024. This series of rapid testing was a result of the modularity in the hardware and software systems. When there were problems with SAM boards, we could seamlessly exchange them for a working one without additional delays to the schedule. Operators were able to quickly learn how the system worked to the extent that they could run the software without avionics assistance. In the Fall of 2024, both the Vespula and Darcy programs began component testing which was able to be supported with a modified ground system using a single SAM which again proved the usefulness of our modular system. Once the vehicles began being integrated in the Spring of 2025 with all the sub-components defined in the paper, it took less than three weeks before the vehicle's avionics were fully integrated and supporting testing. The modularity of the system allowing for fast development, quick modifications, and easy configuration led to these successes and will enable YJSP's goal to be the first collegiate team to launch a liquid rocket to space.

V. Appendix

A. An Example of a Sequence

Sequences are written in Python by the operators, allowing for adaptability and flexibility. Specific phases can be easily adjusted this way without altering the entire workflow.

```
# flow sequence
# start sequence at T - 3.000
import time

print('Flow Sequence has begun')
ftpt_target = 420 * psi # psia, target fuel tank pressure
otpt_target = 360 * psi # psia, target lox tank pressure
switch_open = 3486 * psi # psig, COPV pressure to open switch valves

ran = 5 * psi # psi, range for bang bang algorithm
wait_timing = 5 # ms, for loop check timing
press_time = 3000 # ms, time before sequence starts
flow_time = 27.5 # s, time we are flowing for

fmv_open = 86 # ms, observed FMV actuation open time
fmv_close = 100 # ms, estimated FMV actuation close time
omv_open = 62 # ms, observed OMV actuation open time
omv_close = 100 # ms, estimated OMV actuation close time

lox_open_lead = 200 # ms, targeted lox lead time
lox_close_lead = 500 # ms, time to close OMV before FMV to avoid ox rich

start_wait = press_time - omv_open - lox_open_lead # time to actuate OMV at
fmv_open_wait = omv_open + lox_open_lead - fmv_open # wait between opening omv and fmv
lox_close_wait = omv_close + lox_close_lead - fmv_close # time to wait after closing omv

wait_for(start_wait * ms)
OMV.open()
wait_for(fmv_open_wait * ms)
FMV.open()
wait_for(fmv_open * ms)

print('Main Valves have Opened and Flow has begun')
curr_time = time.time()
# start flow
while time.time() < curr_time + flow_time:
  # determine if switch valves need to open
  if PRPT.read() < switch_open and OSWV.is_closed():
    FSWV.open()
    OSWV.open()
    print('The switch valves have opened')
  # bang bang algorithm
  if FTPT.read() > ftpt_target+ran and FBB.is_open():
    FBB.close()
  if FTPT.read() < ftpt_target-ran and FBB.is_closed():
    FBB.open()
  if OTPT.read() > otpt_target+ran and OBB.is_open():
    OBB.close()
  if OTPT.read() < otpt_target-ran and OBB.is_closed():
    OBB.open()

# close OMV before FMV
OMV.close()
OBB.close()
wait_for(lox_close_wait * ms)
FMV.close()
FBB.close()
OVNT.open()
FVNT.open()
print('Flow Sequence has ended')
```

B. Github

All software written for YJSP's avionics is open-sourced and available for contribution on the Yellow Jacket Space Program Github, under GT Space.

Acknowledgments

In addition to the authors of this paper, there are many individuals who played a role in designing and implementing the YJSP Avionics system. We would like to acknowledge those with substantial contributions, including Billy Ewles, George Golden-Bankier, Rithvik Nagarajan, Andrew Carlisle, Renee Garg, Aparupa Brahma, Panya Bhinder, Patrick Sliwinski, Madhawi Alharbi, Veronica Mok, Donald Bailie, Andrew Kim, Syon Gupta, Pratham Ingale, and Chandini Kalidi.