The Thesis Committee for Sarah Masimore
Certifies that this is the approved version of the following Thesis:

A Distributed Avionics Software Platform for a Liquid-Fueled Rocket

APPROVED BY
SUPERVISING COMMITTEE:

Christopher J. Rossbach, Supervisor

Leon W. Vanstone

A Distributed Avionics Software Platform for a Liquid-Fueled Rocket

by

Sarah Masimore

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Computer Science

The University of Texas at Austin

May 2020

Dedication

To my wife, family, friends, and mentors who have brought joy and meaning to my life.

May you never have to read this thesis.

Acknowledgements

I would first like to thank my thesis adviser and mentor, Professor Chris Rossbach. Chris' Advanced Operating Systems class and subsequent mentorship inspired me to pursue a thesis and dive deeper into the systems world. I would also like to thank Dr. Leon Vanstone, who is a founder and the leader of the Texas Rocket Engineering Lab. His thoughtful leadership has fostered a diverse and productive rocket lab that has given hundreds of graduate and undergraduate students the opportunity to work on and lead industry-level projects. Finally, I would like to thank the intelligent, kind, and dedicated members of the TREL Avionics Software team, who contributed to the design, implementation, and verification of the presented Avionics Software Platform. In particular I would like to thank Stefan deBruyn, Kevin Liang, Simoni Maniar, Sophia Xu, Jennifer Dahm, Sahil Ashar, Matthew Yu, Wilson Watson, Jonathan Baurer, and Mithilesh Konakanchi.

Abstract

A Distributed Avionics Software Platform for a Liquid-Fueled Rocket

Sarah Masimore, M.S.COMP.SC.

The University of Texas at Austin, 2020

Supervisor: Christopher J. Rossbach

A distributed avionics software platform was developed as part of the Texas Rocket Engineering Lab's efforts to become the first university lab to launch a liquid-fueled rocket to the edge of space (100km) and recover it successfully. There are four flight computers on the rocket, each running a real-time version of Linux and connected over an Ethernet network. The Avionics Software Platform runs on all flight computers, providing data handling, thread scheduling, clock synchronization, software error handling, and control logic abstractions for the rocket's avionics system. The Platform executes the rocket's control logic and device drivers at 100Hz, with only 6.2% of the total CPU time dedicated to Platform functions. The requirements, design, and verification of the Avionics Software Platform are presented in this thesis.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

For the first time since the Space Shuttle program, the NewSpace movement marks a step change in expanding access to space and improving launch vehicle and spacecraft technology. Key drivers of the NewSpace movement have been private companies like SpaceX and Blue Origin, who have successfully developed reusable launch vehicles that can reach orbital or suborbital spaceflight. Reusability of launch vehicles has greatly reduced the cost of accessing space, and as a result new industries have emerged in areas such as satellite imaging and internet systems as well as commercial space flight.

This new fleet of reusable and low-cost launch vehicles all have a common and fundamental design feature: they are liquid-fueled rather than solid-fueled rockets. A liquid-fueled rocket uses a liquid fuel (e.g. methane) and a liquid oxidizer (e.g. cryogenically cooled oxygen) to fuel the rocket. These propellants are fed from tanks on the rocket to the engine via tubing, regulators, and control valves. The rocket's thrust can be regulated (even stopped and restarted) by controlling the flow of the propellants to the engine. This capability has enabled SpaceX and Blue Origin to deliver a payload to space, reignite their rocket engines on descent, land the rocket, and reuse the same rocket multiple times. A solid-fueled rocket, on the other hand, uses a mixture of solid fuel and solid oxidizer. Since the fuel does not require a cryogenic environment or plumbing to feed liquids into the engine, the rocket can be much simpler. Solid-fueled rockets, however, do not easily allow regulation or precise control of the rocket's thrust and are therefore much more difficult to reuse. While liquid-fueled rockets are more complex to

develop, companies like SpaceX and Blue Origin have demonstrated that the significant cost reduction per launch enabled by a high-level of reusability is well worth it.

1.1 THE TEXAS ROCKET ENGINEERING LAB

The Texas Rocket Engineering Lab (TREL) is a research laboratory at the University of Texas at Austin. TREL was founded in the fall of 2018 in partnership with Firefly Academy, a nonprofit organization run by Firefly Aerospace. The mission of TREL is to develop capable professionals to meet the needs of new space, commercial launch, and NASA [1]. The lab is composed of about 150 graduate and undergraduate students and is currently focused on designing, building, and testing an approximately 30-foot, liquid-fueled rocket named Halcyon as part of the Base 11 Space Challenge. The Base 11 Space Challenge is a $1 million+ prize for a student-led university team to design, build, and launch a liquid-propelled, single-stage rocket to an altitude of 100 kilometers (the Karman Line) by December 30, 2021 [2].

1.2 HALCYON

Halcyon is a liquid-fueled, single-stage rocket designed to reach a flight apogee of 122km, an altitude higher than the 100km boundary of space. The rocket is currently being designed and developed by TREL as part of the Base 11 Space Challenge. The rocket's engine is pressure-fed propellant from its tanks using valves and regulators to control the flow. Once the engine is ignited, the rocket's attitude is controlled using a thruster-based reaction control system and actuated fins. On descent, a series of three parachutes is used to recover the rocket safely. Except for an operator-commanded abort, the rocket is designed to be 100% automated during flight.

There are 8 subsystems on the rocket. The Structures subsystem includes the rocket's airframe, tanks, fins, electronics enclosures, and final integration of all flight hardware. The Propulsion subsystem includes the rocket's engine, including the igniter, injector, nozzle, and associated cooling systems. The Fluids subsystem manages all fluids in the system, including the plumbing that feeds fuel and oxidizer to the rocket's engine and reaction control system. The Recovery subsystem is responsible for the rocket's nose cone and parachute system for recovering the rocket safely and intact. The Payload subsystem includes the payload that will be deployed at apogee. The Guidance, Navigation, and Control (GNC) subsystem is responsible for the control algorithms that manage the stability and flight path of the rocket. The Avionics Hardware subsystem includes electronic hardware on the rocket, including power, telemetry transceiver, sensors, and actuators, while the Avionics Software subsystem is responsible for the flight computers, onboard communications network, and all software on the rocket.

1.2.1 Avionics Software Subsystem

The Avionics Software subsystem on the rocket is broken down into two layers. First, the Avionics Software Platform layer provides the high-level architecture of the rocket's software systems, including the flight computers, flight network, operating system, data handling, and control abstractions for the rocket. The Platform is designed to be configurable so that it can be used without modification for various iterations of the Halcyon rocket and future TREL rockets. Second, the Avionics Software Subsystem layer runs on top of the Platform, defining the rocket's control logic and drivers (e.g. parachute deploy logic and igniter device drivers). The Avionics Software Platform is the focus of this thesis.

1.3 THESIS STRUCTURE

The thesis is structured as follows. First, the requirements for the Avionics Software Platform are presented. Second, the Platform's design is described. This chapter includes the rationale behind high-level system decisions such as the distributed architecture, flight computer, and network protocol. Third, the testing and performance measurements of the implemented system are presented. Finally, opportunities for future work are discussed and a summary of outcomes is presented.

Chapter 2: Requirements

2.1 USE CASES

In order to define a set of system requirements for the Avionics Software Platform, a list of representative use cases was first collected from the rocket's subsystems. A use case for the rocket is defined as an interaction between a subsystem or ground operator and the Platform. The following use cases represent 10 key interaction types:

1. *Parachute Deployment:* The Recovery subsystem reads an altitude sensor value indicating that a parachute should be deployed within the next few seconds via igniting a black powder charge. This use case describes the need for subsystem software to automatically read sensors, run control logic, and command actuators.
2. *Rocket Attitude Control:* GNC subsystem reads various rocket sensors, which indicate the rocket's attitude is off course, and must set new fin target angles within 50 milliseconds. Describes subsystem need to run high-level control logic that manages various components of the rocket.
3. *Single Fin Control:* A single fin requires a software control loop running at a high frequency to reach and maintain the fin's target angle against environmental factors. Describes subsystem need to run low-level control logic that manages a single rocket component at a high frequency.
4. *Coordinated Rocket Control:* Valves, attitude control mechanisms, and ignition must be coordinated to provide stable control of the rocket. Describes the need to coordinate the timing of actions across the rocket within some acceptable timing variability.

5. *Telemetry Transmission to Ground:* Ground operators use a low-latency telemetry stream to prepare for launch, monitor the rocket during flight, and recover the rocket. Describes the need to integrate with ground systems and provide snapshots of the entire system's state throughout the rocket's day-of-launch activities.
6. *Hardware-In-the-Loop (HIL) Verification:* Test team runs system-level tests using a HIL test harness and requires telemetry data throughout to verify that the integrated avionics software system behaves as expected. Describes the need for continuous streaming of system state snapshots in a testing environment.
7. *Pre-Launch Preparations*: To prepare and OK the rocket to launch, sensors must be enabled, telemetry stream active, pre-launch tests run, and all actuators set to a stable state. Describes the need for constant data monitoring of the system and the capability to test and directly control actuators before launch. Similar need exists during the rocket recovery phase.
8. *Automated Abort:* If a fault occurs, the system needs to automatically activate the Flight Termination System (FTS). Describes the need to automatically monitor the health of the system and change the system's state to an abort state.
9. *Ground Operator Commands Abort:* Ground operators may transmit the abort command to the rocket, which results in FTS activation. Describes the need to change the system's state via an operator command.
10. *Flight Computer Failure:* Strong desire to have redundancy so that the rocket can be recovered intact. Describes the desire to have physical redundancy of the flight computers.

2.2 REQUIREMENTS

From these use cases the following Platform requirements were derived:

- The flight software shall be deterministic.

Given the same inputs and assuming 100% network reliability, the flight software must produce the same output every time. This is required to debug, test, and qualify the flight software. Otherwise, if the flight software passes a test, there is no guarantee that the software will pass the identical test again in the future. This applies to both software running on a single flight computer and any coordination between flight computers in a distributed setting.

- Network reliability shall be at least 99.999% in the nominal case.

Network reliability in this requirement is defined by the percentage of messages successfully sent by the sender node and successfully received by the receiver node within a usable time period. This reliability metric detects dropped or delayed messages. While network communications between flight computers cannot be guaranteed to be deterministic (e.g. a message could be dropped due to electrical interference and a subsequent checksum error), message drops must occur less than .001% of the time. This is essential for the system to be able to react quickly to new data and to minimize non-determinism in the system caused by dropped or delayed messages. In this requirement the nominal case is defined as all nodes, the network switch, and network connections being functional. If any of these components are not functional, network reliability will be significantly degraded. At 99.999% reliability, the Platform is providing a guarantee that no more than 1 out of every 100,000 messages will be dropped or delayed beyond usability.

- The flight software shall have a reaction time of no more than 50ms for high-level control logic and 10ms for low-level control logic.

Reaction time is defined as the time between new sensor data being available and an updated actuator value being set based on that new sensor data. This requirement is primarily informed by the GNC team, which is responsible for writing the control algorithms that react to the environment and guide the rocket to apogee. There are two levels of control logic with different reaction time requirements. First, high-level control logic is responsible for managing the overall rocket, relying on multiple sensors and setting target values for multiple actuators. For example, the GNC team reads multiple IMU's and barometers to determine the position and attitude of the rocket and commands multiple fins and the reaction control system to modify the rocket's position and attitude. Second, low-level control logic manages a single actuator. For example, high-level control logic sets a target angle for all fins, and low-level control logic makes sure the fin is set to that angle. While high-level control logic can tolerate a reaction time of up to 50ms, the low-level control logic requires a faster reaction time of up to 10ms to maintain stability of the rocket.

- The flight software shall run control logic and hardware drivers at a frequency of 100Hz.

The frequency at which the control logic and hardware drivers run directly impacts the rocket's reaction time. Similar to the reaction time requirement above, this requirement is primarily informed by the GNC team. Spacecraft like the Space Shuttle ran their software loops at 25Hz in order to maintain stability and control of the spacecraft [3]. Because Halcyon is significantly smaller than the Space Shuttle, Halcyon can destabilize more quickly, meaning the frequency the control logic runs must be

higher. While 100Hz may end up being faster than the GNC team requires for controlling Halcyon, based on the team's simulations this frequency is guaranteed to satisfy their needs.

- The flight software shall support up to 100 sensors and 100 actuators.

This requirement is informed by all subsystems on the rocket. As a liquid-fueled, actively controlled rocket, a large number of sensors are required to understand the state of the rocket for both control and engineering feedback purposes. Similarly, a large number of actuators are required to control things like valves, igniters, fins, power systems, and the reaction control system. While it is unlikely Halcyon will require 100 sensors and 100 actuators, it is guaranteed that the subsystems in aggregate will not need more than this number. The number as well as the types of sensors and actuators supported by the Platform is bounded by the system's network bandwidth, physical I/O, and CPU availability. This is discussed further in Chapter 4.

- The flight software shall support configurable system states.

A state machine is an effective and intuitive way of representing the complexities of a system's behavior, and is therefore a common abstraction used to manage a system's state. Each state represents a grouping of related activities (e.g. running through pre-flight sensor tests, controlling the rocket to apogee, executing an abort sequence), as well as a list of transition conditions (e.g. pre-flight tests complete). If a transition condition is met, the system transitions to a new state. Additionally, a state machine provides a clean way to ensure certain safety rules are not violated throughout flight. For instance, parachute deployment must never be armed before engine ignition, as it is a safety hazard for ground personnel. This can easily be enforced using a state machine by only arming the

deployment system in a post-launch state. For these reasons, Halcyon is required to use a state machine. Since the exact state machine used by Halcyon will change frequently during development and testing, states must be easily configurable.

- The flight software shall support user-commanded, time-based, and flight data-based state transitions.

Launch operators are required as a part of the Base 11 requirements to be able to initiate an abort sequence at any point. The rocket's state machine therefore must support operator commanded transitions to an abort state. An operator command will also be required to initiate the final launch sequence. During all other operations, the system must transition through the rocket's states using time-based logic (e.g. after running a 60 second pre-flight test) and data-based logic (e.g. a flag indicating apogee has been reached).

- The flight software shall provide a mechanism for streaming snapshots of the rocket's state with a latency of no more than 30ms.

Launch and recovery operators must have insight into the rocket's state in order to verify nominal conditions for launch, command an abort, and verify the rocket is safe to approach after flight. Furthermore, if the rocket fails, having snapshots of the rocket's state is critical to investigating a failure. The flight software must therefore support streaming snapshots of the rocket's state throughout the mission. This snapshot data will be streamed over a physical connection during pre-flight operations and over a transceiver during flight and recovery operations. Before flight, the rocket's entire state must be streamed. During flight and recovery operations, the telemetry data bandwidth will be bound by the transceiver hardware. In the case of a failure, a low latency is

critical to investigators having as much data as possible before the telemetry systems are destroyed. It is desired that the overall latency is no more than 50ms due to how quickly a rocket can fail, and 30ms of latency in the software gives the transceiver hardware an additional 20ms to meet this desired overall latency. The Avionics Software team is responsible for the software mechanisms involved in streaming telemetry, and the Avionics Hardware team is responsible for the physical mechanisms (e.g. a transceiver).

- All flight software shall detect and handle all software errors such that the system transitions to a predefined state.

If a software component fails, the failure needs to be detected and handled appropriately depending on what error has occurred and when in the mission it has occurred. This requirement is specifically related to software errors (e.g. a scheduling deadline is missed, a bad parameter is passed, an exception is thrown), rather than rocket errors (e.g. a tank temperature is too high or a sensor is returning no data). While the Platform does provide abstractions to detect and handle these types of rocket errors, the specific rocket error detection and handling logic is part of the Avionics Software Subsystem layer. It is important to note that this requirement is not a fault tolerance requirement. Handling an error in this case means making an explicit decision on what to do when the error occurs. For some errors, this may mean logging the error and continuing, and for others it may mean initiating an abort.

2.2.1 Exclusion of a System-Level Fault Tolerance Requirement

There are many failure points in an avionics system. A flight computer could become unresponsive, a network switch could lose power, a sensor could start producing faulty data, a connector could come loose, or a software bug could cause undefined

behavior. Depending on the consequence of a rocket's failure (e.g. risk to human life or destruction of a billion dollar satellite) and the system's time, mass, and financial budget, various fault tolerance strategies have been employed in the space industry. The Space Shuttle, for example, had four redundant flight computers and a fifth independently programmed backup flight computer in case the four redundant flight computers suffered from a common mode software bug [4]. Launch vehicles and spacecraft commonly use what is known as a 3-string architecture, where all avionics components are triplicated to tolerate byzantine as well as component fail-stop failures. 3-string architectures are 1-fault tolerant, meaning they can tolerate any single component failing and still be able to complete the mission. A single-string architecture does not implement system-wide redundancy and is therefore 0-fault tolerant, however the architecture as a result is much simpler.

A *system-level* fault tolerance requirement was intentionally excluded from the Platform requirements due to the additional financial, mass, and development time cost. However, redundancy is added at the *component-level*. As more is learned about the failure modes and failure rates of the various avionics hardware and software components via stress and environmental testing, redundancy is added to specific components. This strategy allows for a more cost-effective approach to redundancy, although the trade-off is that the system as a whole is not 1-fault tolerant. An additional risk of taking this approach is that component failure risks may be discovered late in the development of the Platform, making it more difficult to modify the system. To mitigate this risk, a significant amount of effort has been applied to developing a simple, well-documented, and highly modular Avionics Software Platform, as well as following a strategy of testing early and often. Furthermore, the iterative software development lifecycle used by the Avionics Software team facilitates a test and learn cycle and sets the expectation that the

flight software will change as it is tested and integrated into the rocket. This helps the team avoid the sunk cost fallacy and remain willing to change or even throw away previously implemented software that is shown to be insufficient in testing.

2.3 Requirements Verification

Each Avionics Software Platform requirement must be verifiable and have a corresponding verification strategy. The following verification techniques are used to verify the Platform with a verification strategy for each requirement shown in Table 1:

- Safety-Critical Coding Standard

All avionics software that will fly on the rocket is required to follow an industry-level coding standard. Coding standards include style and documentation guidelines as well as software patterns to follow or avoid when writing software for safety-critical systems. The selected coding standard is High-Integrity C++ developed by Perforce, which is also used by SpaceX's flight software team [5]. The standard is enforced in code review and will be enforced using static analysis later this year. The selection of C++ as the flight software programming language is discussed in Chapter 3.

- Code Review

All flight software code additions, removals, and modifications are required to be reviewed and approved by another member of the Avionics Software team. Code reviewers validate the design and verify the implementation against component requirements; they check for adherence to the safety-critical coding standards and ensure documentation is present and understandable throughout the code; and they verify the automated unit and integration tests sufficiently exercise both the success and failure

cases of the software. Software is version-controlled using git with a protected master branch so that code review is required before any changes can be merged.

- Unit Testing

All flight software components are required to have a corresponding unit test suite written using an automated unit testing framework. A unit test suite is a set of tests designed to verify a specific software component's success and failure cases. An example unit test would be executing a function with invalid parameters and checking that the function returns an expected error status. Another example would be executing the same function with valid parameters and checking that the function returns a success status and the software's state changes as expected. An automated unit testing framework is a tool that enables these unit tests to be written in software and automatically run via a script. The Avionics Software team uses Cpputest, a unit testing framework and memory leak detection tool commonly used for C and C++ embedded projects. The automated nature of the test framework allows the unit test suites to also serve as regression tests, as all unit tests must run successfully before a new code change can be merged into master.

- Integration Testing

Integration tests are designed to test the behavior of multiple software components working together in an integrated setting. An example integration test for the Avionics Software Platform is testing the software running on a single flight computer, while simulating the other flight computers on the network.

- System Testing

A system test is designed to test the entire Avionics Software Platform system running across all flight computers. Various versions of the Avionics Software Subsystem layer are run on top of the Platform layer to test different features of the Platform. This testing technique includes testing multiple rocket subsystems together (e.g. Platform & Recovery Subsystem, Platform & Ground Infrastructure). The telemetry stream is used to verify the results of the test.

- System Profiling

System profiling is done to measure the Platform's overall performance. To profile the Platform, custom Subsystem layers are designed to non-intrusively measure various features of the Platform. For example, a simple Subsystem layer is designed to measure the system's reaction time by measuring how long it takes for an actuator to be commanded based on new sensor data.

- Hardware-In-the-Loop (HIL) Testing

HIL testing is the highest fidelity flight software test that can be done without actually launching the rocket. A HIL tool sends electronic signals to the flight computers to simulate sensors and then reads the flight computer electrical outputs and telemetry to verify the system is functioning as expected. The Avionics Software team is currently building a HIL test platform, which will be used for final qualification of the flight software later this year.

- Static Analysis

Static analysis tools inspect software without executing the code. This technique can detect issues such as an out-of-bounds array access or unhandled exception. Furthermore, adherence to the High Integrity C++ coding standards can be verified. Static analysis of the flight software will be done later this year.

Requirement	Verification Strategy
The flight software shall be deterministic.	• *Code Review:* Check for dependencies on non-determinism, such as time or random number. • *Unit & Integration Testing:* System output for every run must be the same, assuming 100% network reliability.
Network reliability shall be at least 99.999% in the nominal case.	• *System Profiling:* Measure % of messages sent that are successfully received over a time period of 2x the expected mission time.
The flight software shall have a reaction time of no more than 50ms for high-level control logic and 10ms for low-level control logic.	• *System Profiling:* Determine the maximum reaction time of the system.
The flight software shall run control logic and hardware drivers at a frequency of 100Hz.	• *Integration & System Testing:* The Platform must be able to run at 100Hz without missing its 10ms deadline. • *System Profiling:* Run the Platform for 2x the expected mission time and verify no deadline misses.
The flight software shall support up to 100 sensors and 100 actuators.	• *System Profiling:* Compare network bandwidth, flight computer physical I/O, and CPU time availability to that required to support 100 sensors and 100 actuators.
The flight software shall support configurable system states.	• *Unit, Integration, & System Testing:* Verify the Platform's state machine's ability to run various configured states.

Table 1: Continued on next page.

The flight software shall support user-commanded, time-based, and flight data-based state transitions.	• *Unit, Integration, & System Testing:* Verify the state machine's ability to transition state based on user commands, time elapsed, and arbitrary flight data.
The flight software shall provide a mechanism for streaming snapshots of the rocket's state with a latency of no more than 30ms.	• *Integration & System Testing:* Verify the Platform streams snapshots of the rocket's state. • *System Profiling:* Determine the maximum latency of the rocket's state snapshots.
All flight software shall detect and handle all software errors such that the system transitions to a predefined state.	• *Code Review:* Verify Platform error detection and handling patterns are implemented. Check for unhandled exceptions and verify safety-critical standards are followed. • *Unit & Integration & System Testing:* Verify Platform handles software errors. • *Static Analysis:* Check for unhandled exceptions and that safety-critical standards are followed.

Table 1: Platform Requirements and Verification Strategy.

Chapter 3: Design

The following chapter describes the Avionics Software Platform's design, including design goals, the selected development life cycle, and system architecture.

3.1 Design Goals

The first step in moving from the Avionics Software Platform requirements to a system architecture was establishing a set of five design goals to guide design decisions. The first design goal is scalability. Halcyon's subsystem requirements are in continual flux as teams go through their own test and learn cycles. As such, the Platform should be able to scale up or down to meet CPU and I/O needs of the subsystems without requiring a fundamental design change.

The second design goal is modularity. A modular system, whose complexity and functionality is split across discrete and simple modules, is easier to design, develop, test, and modify. Additionally, the Platform will go through iterations as the software and hardware is tested. It is critical, therefore, that the system be modifiable. Modularity helps ensure this.

The third design goal is to avoid over-engineering. Over-engineering the Avionics Software Platform to satisfy every possible "what if" scenario and optimized against every system metric will result in a complex system that is difficult to understand, develop, and test. Instead, the system is strictly designed to handle only known scenarios. The tradeoff is that as the Platform goes through test and learn loops, some of the "what if" scenarios previously considered but not included in the design will turn out to be known scenarios. While it can be more difficult to add features later in a system's development lifecycle, the Platform's emphasis on modularity and avoiding over-engineering results in a simpler design that is easier to modify when necessary.

The fourth design goal is configurability. As the overall rocket design matures, many aspects will change. In particular the number and types of sensors and actuators used, the exact behavior of the state machine, hardware calibration values, and control logic parameters will change. The Platform software components are therefore designed to be configurable to facilitate these types of changes.

The final design goal is testability. The Platform must be testable at the unit, integration, and system level to ensure the system correctly implements desired behavior. The easier it is to test a system, the more the system will be tested. This means the system and supporting test tools must be designed so that adding test cases and verifying system behavior is trivial.

3.2 Software Development Life Cycle

The Avionics Software team follows an iterative software development life cycle. An iterative model uses the waterfall approach (requirements → design → develop → test → deploy) over multiple iterations. This approach has the benefits of thorough design, development, and test phases, while also providing the flexibility to iterate on the system's design over time and integrate feedback from software and hardware testing into future iterations. The initial iteration of the Platform, for example, was the minimum viable product (MVP) that included data handling, a single control logic abstraction, and I/O capabilities. The next iteration included a state machine, clock synchronization, and operator command capabilities, as well as improvements in networking reliability to address issues discovered during network testing.

3.3 System Architecture

The Platform uses a single-string, distributed avionics architecture. A single-string architecture provides no system-wide fault tolerance, however it allows for a simpler and more cost-effective system. A distributed architecture uses multiple flight computers (or nodes) that work together to meet the system's I/O and CPU requirements. While using a single, more powerful flight computer would be simpler, there are three key benefits to using a distributed architecture. First, a distributed architecture can scale up or down to meet I/O and CPU needs by adding or removing nodes. While designed to run on multiple flight computers, the Platform can also scale down to running on a single computer. Second, using many cheaper, less powerful flight computers instead of a single, more expensive and more powerful flight computer allows the Avionics Software team to reduce resource contention during development. Developing and testing much of Halcyon's flight software requires running the software on a physical flight computer, so using many cheaper flight computers facilitates parallel development and testing. Finally, industry-scale rockets typically use distributed architectures, and building experience working in these complex systems is critical to preparing TREL software engineers for industry.

3.3.1 Parent-Child Model

In a distributed architecture the role of each node and their relationship to every other node must be defined. The Platform uses a parent-child model, where there is a single parent node and multiple child nodes, each running on their own flight computer (Figure 1). The parent node coordinates all child node actions, and the child nodes communicate only with the parent node. The parent node in the Platform architecture is

called the Control Node, and its primary role is to manage the overall system and execute the rocket's high-level control logic.

There are two types of child nodes. The first is a Device Node, whose primary role is to interface with the rocket's sensor and actuator devices. A Device Node's secondary role is to execute low-level control logic, for example running a PID controller to manage the angle of a fin. This is possible as long as the sensors and actuators used by the controller (e.g. the fin's angle sensor and motor) are directly connected to the same Device Node. The Platform is designed to support up to three Device Nodes on the rocket. With every additional Device Node, the system adds I/O and CPU capabilities. However, each additional node also means spending more time synchronizing data across the system. Selecting three Device Nodes as the maximum supported enables the Platform to satisfy the 100 sensors and 100 actuators requirement through sufficient physical I/O, while still meeting the 100Hz loop frequency requirement.

The second type of child node is called a Ground Node. Unlike the Control Node and Device Nodes, which run on flight computers on the rocket, the Ground Node runs on a ground computer. The Ground Node is used by ground operators to receive telemetry from and send commands to the rocket. During launch preparations the Control Node communicates with the Ground Node over the flight network, and during the flight and recovery phases communication between the two nodes is done over a transceiver.

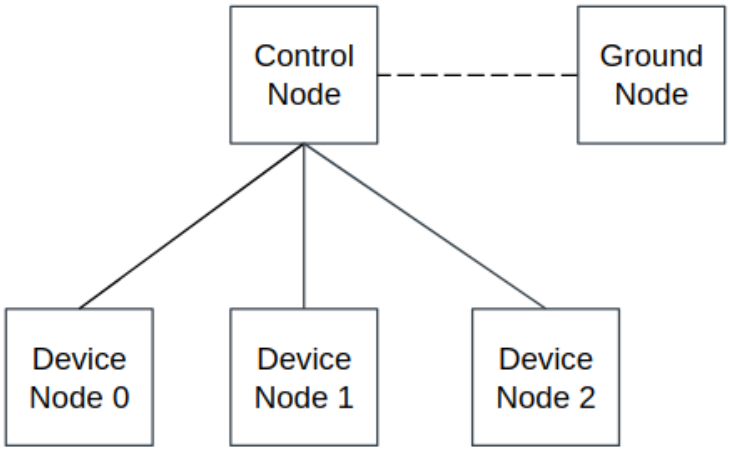

Figure 1: Parent-Child Distributed Architecture.

The parent-child model was chosen to simplify coordination between nodes and to increase determinism in the system. It is much easier to design, develop, debug, and test a distributed system that has a single coordinator rather than a distributed system with all nodes running independently. A system with a single coordinator can deterministically control things like network traffic, data synchronization across nodes, and when each node runs. Additionally, the splitting of node roles between control and device nodes follows a similar pattern used in other spacecraft, such as SpaceX's Dragon vehicle which relies on remote input/output modules to interface with hardware devices [6].

3.3.2 Flight Software Loops

The Control Node and Device Nodes execute their control logic and device drivers in synchronized, single-threaded 100Hz loops (Figure 2). The loops are synchronized by the Control Node so that they start at approximately the same time, increasing determinism in the system. A single-threaded loop for each node was chosen to avoid the complexities of a multi-threaded environment, where careful synchronization

between threads is required to prevent race conditions and deadlock. While a single-threaded environment is simpler, it is also typically less efficient, since the software cannot be parallelized. This is a perfect example of where the “Avoid Over-Engineering” design goal guides the design decision. There is no evidence that the system requires the efficiency of a multi-threaded environment, so the simpler, single-threaded design is selected.

Since the sensors, actuators, and control logic are distributed across flight computers, regular synchronization of the system’s data is required to ensure each computer has the information it needs to run its loop. At the top of each loop, data across the system is synchronized. After data synchronization is complete, the Control Node sends telemetry to the Ground Node, receives an optional operator command, and runs the state machine and high-level control logic. The Device Nodes, on the other hand, read data from sensors, run low-level control logic, and update actuator values.

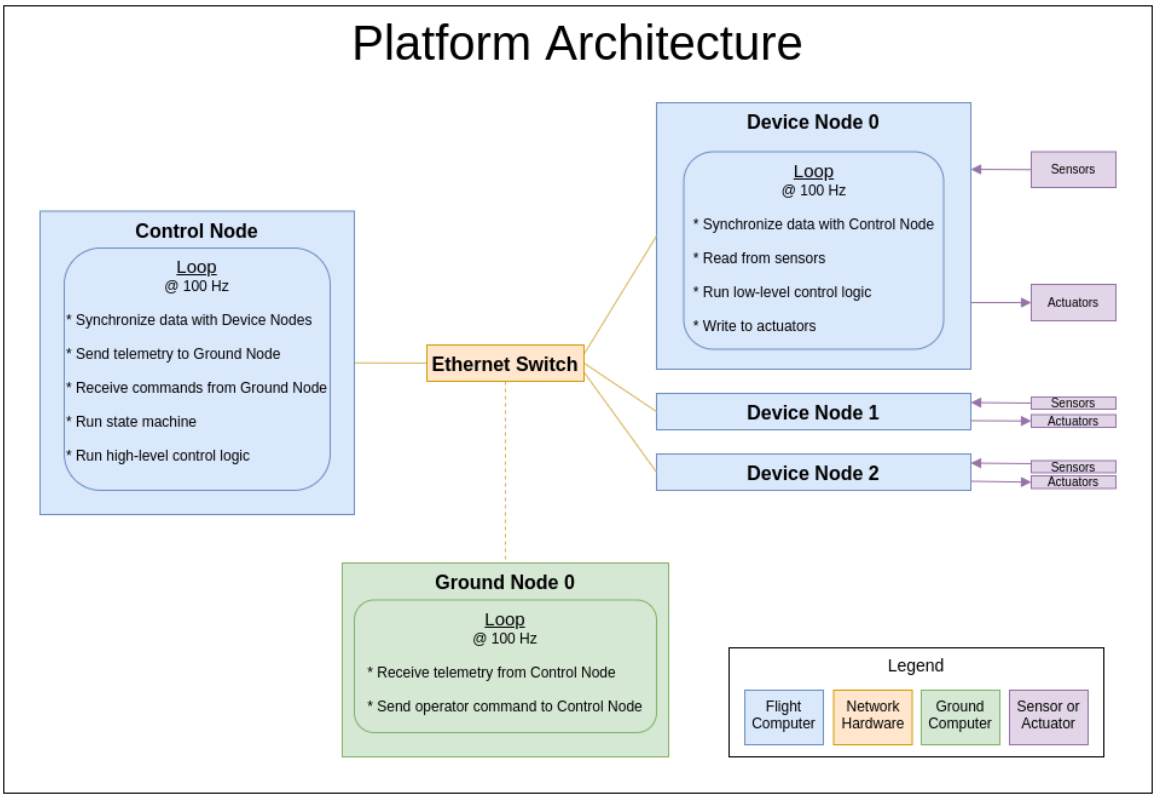

Figure 2: Platform Architecture.

3.3.3 Flight Network

The flight network is implemented using Ethernet in a star topology (Figure 2). In a star topology all nodes are connected via a central connection point. As compared to using an Ethernet bus topology, a star topology is simpler to implement (use an Ethernet switch as the central point) and ensures there are no message collisions. A CAN (Controller Area Network) bus was considered as an alternative to Ethernet, since more engineers on the team had experience with CAN. However, in order to support 100 sensors and 100 actuators, the Platform requires a bandwidth of at least 1.3 Mbps (assuming each sensor and actuator requires an average of 64 bits of data and data is

synchronized each loop). CAN only supports up to 1 Mbps, while Ethernet can typically support up to 1 Gbps.

3.3.4 Flight Computer and Runtime Environment

National Instrument (NI) single-board controllers were selected as Halcyon's flight computers. Specifically, the NI sbRIO-9637 model was selected (Figure 3).

Figure 3: Flight Computer – NI sbRIO-9637.

The sbRIO is a compact, high-performance embedded computer, with the following CPU and I/O specifications:

- 667 MHz dual-core ARM processor
- 512 MB volatile memory
- 512 MB non-volatile memory
- Programmable Xilinx FPGA
- 1 Gigabit Ethernet
- 16 single-ended analog inputs (or 8 differential)
- 4 analog outputs
- 28 bidirectional digital I/O channels

- 2 RS-232 serial ports
- 1 RS-485 serial port

While the sbRIO is not space-rated, it has been used successfully in space in CubeSats [7]. The sbRIO will undergo environmental stress testing later this year to determine acceptable temperature, vibrational, and pressure bounds. An enclosure designed by the Structures subsystem will be used to maintain an acceptable environment throughout flight.

Using an NI controller also allows the team to take advantage of the NI tool stack, including LabVIEW for programming the FPGA and NI's fork of Linux called NI Linux Real-Time. NI Linux Real-Time currently uses the 4.14.87 Linux kernel with the PREEMPT_RT patch, which allows Linux to run more like a real-time operating system. Real-time operating systems are used for systems that have strict timing guarantees (e.g. running a software loop at 100Hz). A version of Linux was the preferred operating system for the Avionics Software team as it is open-sourced, has a large user base, and comes with useful utilities. A version of Linux with the PREEMPT_RT patch is also used by SpaceX flight computers, demonstrating the successful application of Linux as a rocket's operating system [8].

The flight software is run as a single application in the user space on each flight computer. The software is written in C++11, which allows for object-oriented programming as well as direct control of memory management. The 11 version of C++ is used as it has been widely used in embedded systems and the nuances and issues of the version are well-understood. One nuance of the sbRIO is that all digital and analog I/O is connected to the CPU through the FPGA. The FPGA is primarily programmed using

LabVIEW to give the CPU direct access to the I/O as well as to run protocols such as I2C.

3.3.5 Software Components

The Platform provides 5 key functions:

1. *Data handling* capabilities to efficiently move data around the system, synchronize flight computer data, and stream telemetry to the Ground Node.
2. *Control logic abstractions* for the Avionics Software Subsystem layer to implement state machine logic, control algorithms, and device drivers.
3. *Managing time* so that the flight software has a monotonic, linear, and globally relevant clock that will not overflow.
4. *Software error handling* to detect, surface, and handle software errors.
5. *Scheduling* the flight software application on the CPU so that jitter and deadline misses are minimized.

To achieve a modular design these five Platform functions are split across 15 Platform software components with few interdependencies.

Chapter 4: Verification

The following chapter describes the testing and performance of the Avionics Software Platform. The system requirements are then verified using the testing and performance results.

4.1 TESTING

4.1.1 Unit Testing

Each Platform software component has a header file that documents the component's behavior. Unit testing is used to test individual Platform software components against this documented behavior. Each component has a corresponding test suite, which is used to verify the component and as regression testing whenever a new code change is introduced to the Platform. The unit tests are implemented using Cpputest, a unit testing framework and memory leak detection tool.

Overall, the Avionics Software Platform unit test package includes 235 success and failure test cases with 11k test assertions, an average of 15 test cases and 733 test assertions per software component. Statement coverage was measured using the gcov utility and showed 90.7% statement coverage for the Platform. The remaining uncovered statements were manually reviewed and almost entirely attributed to unreachable error handling code when making system calls (e.g. interfacing with sockets or files). In order for these lines to be reachable, the system calls will need to be stubbed to force an error to be returned.

Two unexpected results came out of the Platform unit tests, both related to testing the flight computer's FPGA. The first was a memory leak detected by Cpputest. This

turned out to be a known issue with NI's FPGA software, where a negligible amount of memory is leaked once per initialization of the FPGA [9]. This is not an issue for the flight software as the FPGA is initialized only once, so the memory leak is ignored. The second unexpected result was that the digital I/O pins float at a voltage of around .7V during FPGA initialization. On the rocket many of the digital I/O pins will be connected to igniters that are responsible for detonating black power during the recovery phase of the mission. Testing showed that the floating digital I/O pins could detonate the black powder, which is a significant safety risk to both personnel and the rocket. There is no known software fix for this issue, so a pulldown resistor is required to be used for all actuators relying on a digital signal to actuate.

4.1.2 Integration Testing

Integration testing is done to verify the integrated Platform running on a single flight computer. This is distinct from system testing, where the entire Platform is running on multiple flight computers. Integration testing is done by running the Control Node or Device Node software on CPU 1 and simulating the other flight and ground computers on CPU 0. Loopback IP addresses are used to communicate between the software under test and the simulated nodes. Both success and error cases (e.g. a network message being dropped, clock synchronization failing, etc…) are tested.

4.1.3 System Testing

System testing is done to verify the full Avionics Software Platform running on multiple flight computers. Different versions of the Avionics Software Subsystem layer are created to set up the relevant test conditions. Two system-level tests were run to verify the Platform.

4.1.3.1 LED Platform Test

The first system test designed was the LED Avionics Software Platform Test. This test defines the simplest Subsystem layer required to exercise every Platform feature in the nominal case.

The test setup uses four flight computers (one Control Node and three Device Nodes), a ground computer (Ground Node), an Ethernet switch connecting the five nodes, and 11 LED's (Figure 4). Five LED's were connected to Device Node 0, one for each state the test will move through. These LED's are managed by control logic running on the Control Node that checks the system state and turns on the corresponding LED. Five LED's were also connected to Device Node 1. When in a specific system state, these LED's light up sequentially. Finally, there is one LED connected to Device Node 2. This LED is managed by control logic running on Device Node 2, which commands the LED to flash at 1Hz.

To interface with each LED a device driver was created that can be commanded to set a digital I/O pin high or low. The driver also reads the current actual state of the digital I/O pin (high or low) to provide feedback to the control logic. This allows the control logic to verify that a pin value was successfully set.

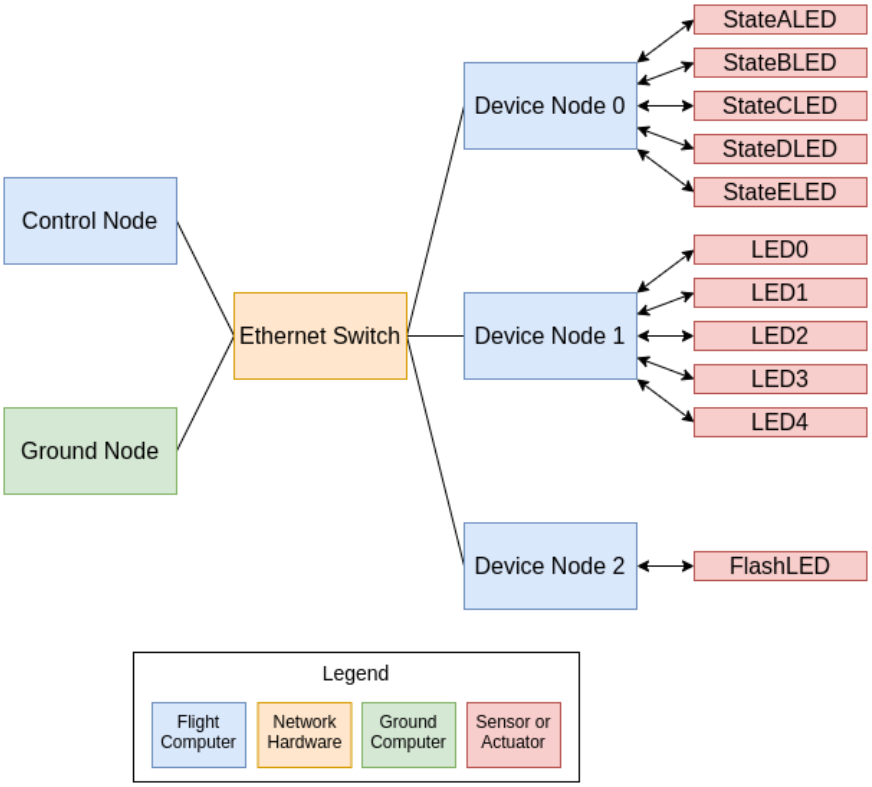

Figure 4: LED Platform System Test Setup.

The state machine for the LED test is shown in Figure 5. State A is the initial state. The only action in this state is to enable the control logic managing the Device Node 0 LED's, which results in StateALED turning on. The system then remains in State A until the LAUNCH command is received from the Ground Node, which initiates a transition to State B. In State B the system loops until three seconds have elapsed at which point a transition flag is set to true. This initiates a transition to State C. In State C the control logic managing the LED's connected to Device Node 1 is enabled. This control logic sequentially turns the five LED's on over five seconds. After the last LED (LED4) is enabled, LED4's feedback value should now read true. After this occurs the system transitions to State D. In State D the control logic running on Device Node 2 is enabled. This results in FlashLED flashing at 1Hz. Once the system receives the ABORT

command from the Ground Node, the system transitions to State E. In State E only the state LED control logic is left enabled, so the only LED to remain on is StateELED. Figures 6-10 show pictures of the system as it moves through the system's five states.

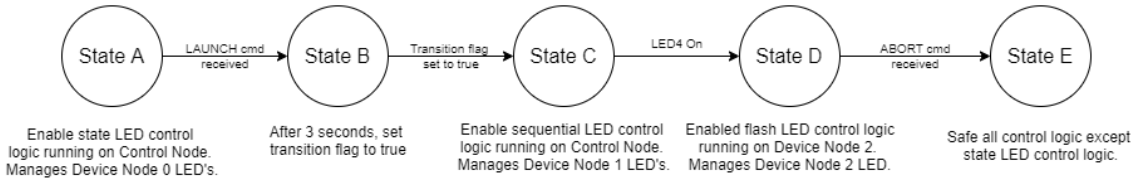

Figure 5: LED Platform System Test State Machine.

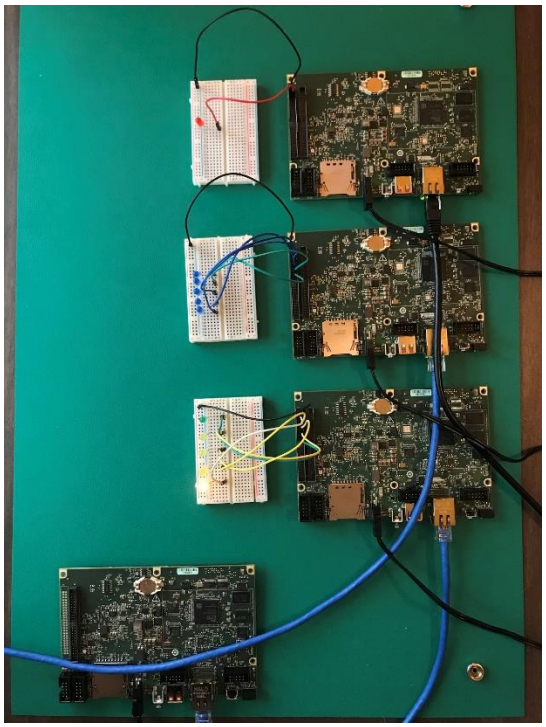

Figure 6: LED Platform System Test State A.

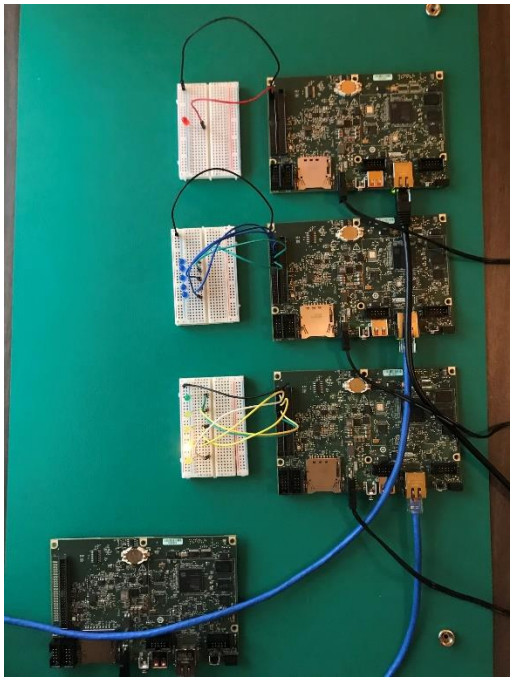

Figure 7: LED Platform System Test State B.

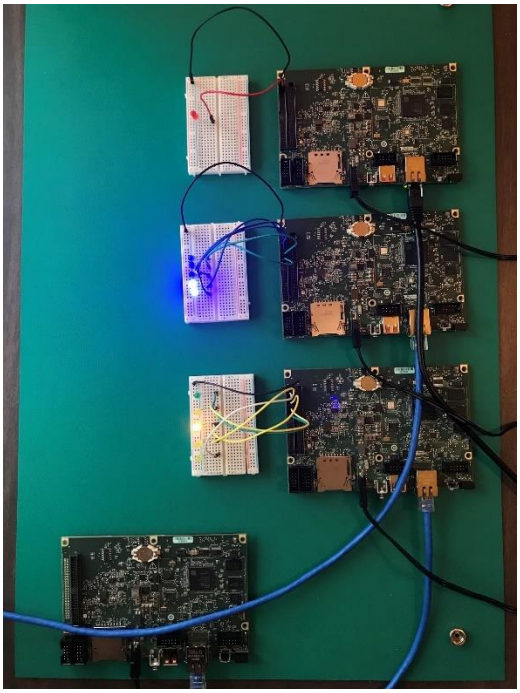

Figure 8: LED Platform System Test State C.

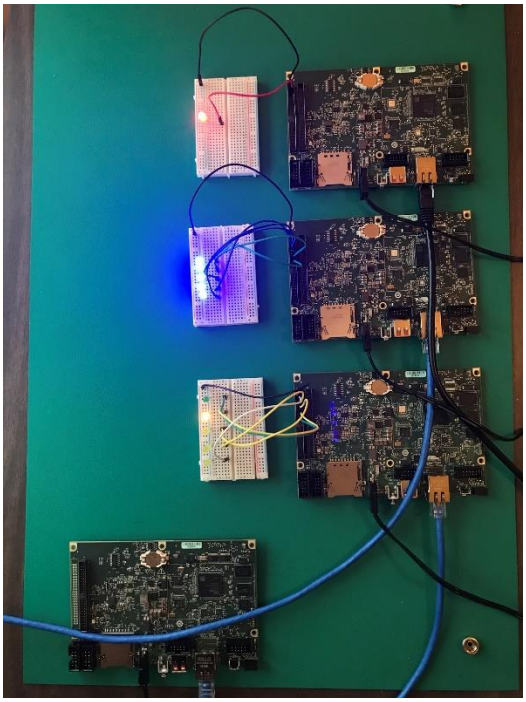

Figure 9: LED Platform System Test State D.

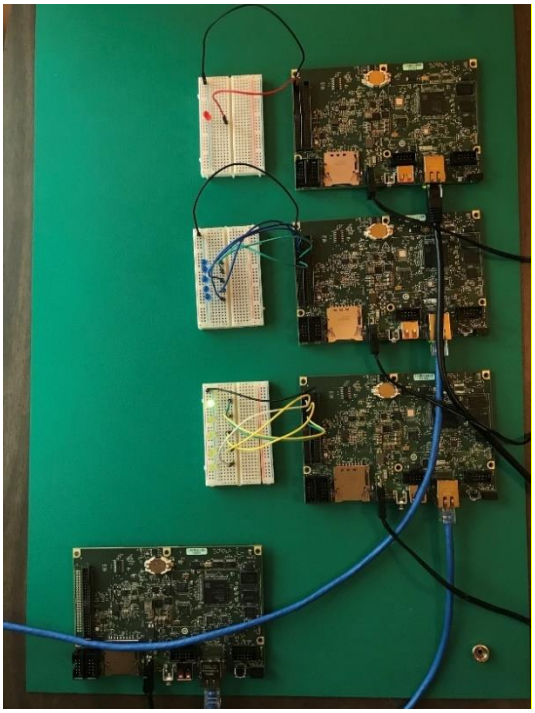

Figure 10: LED Platform System Test State E.

After running the LED System Test, the telemetry log on the Ground Node was inspected. The log showed the system progressed as expected and ended in the correct final state. No software errors, message drops, or deadline misses occurred.

4.1.3.2 Recovery Igniter Test

The second system test to verify the Platform was the Recovery Igniter Test. The Recovery Subsystem layer software was developed by the Avionics Software Integration team in partnership with the Recovery team. The Recovery Igniter Test was designed to verify the Recovery igniter software and hardware integration with the Avionics Software Platform. The test ran on three flight computers (one Control Node and two Device Nodes) and one ground computer. Control logic running on the Control Node sends an ignition command to the Device Nodes. Redundant control logic running on each of the Device Nodes checks the command against a set of safety rules before signaling to the igniter device driver to set a digital output pin to high. The output pin is connected to the igniter circuit, which converts the digital signal to the required current to actuate the igniter. The test successfully ignited black powder, which is the parachute deployment mechanism on the rocket. Figure 11 shows the flight computer setup, Figure 12 shows the igniter in black powder, and Figure 13 shows the successful black powder ignition.

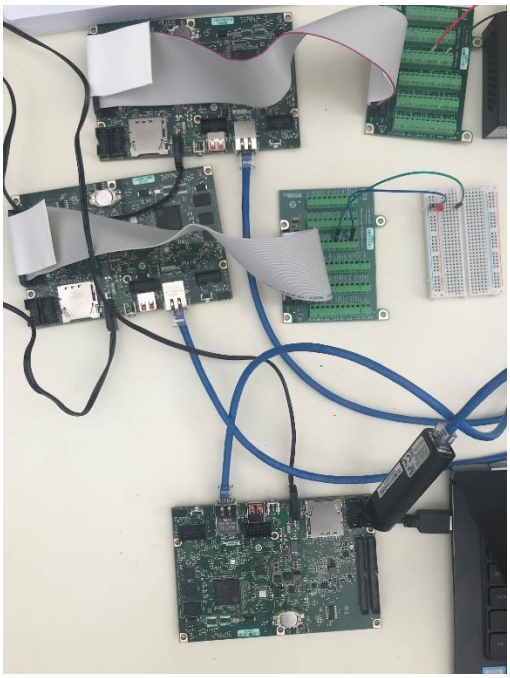

Figure 11: Recovery Igniter System Test Flight Computer Setup.

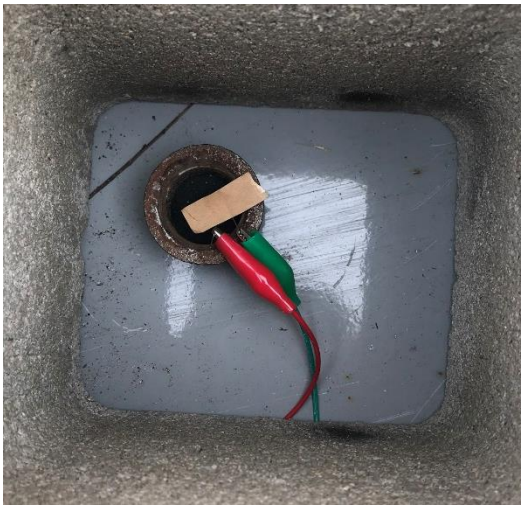

Figure 12: Recovery Igniter System Test Igniter Setup.

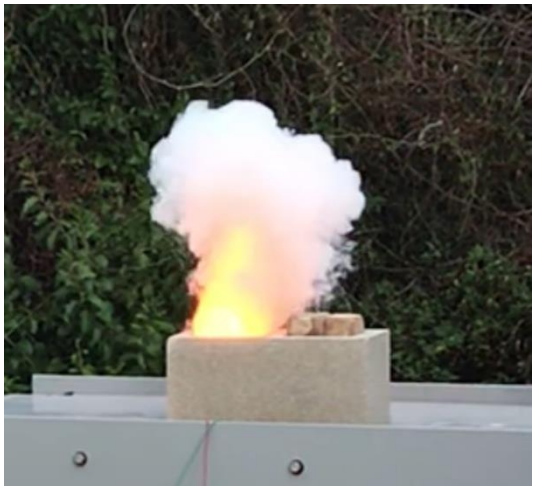

Figure 13: Recovery Igniter System Test Ignition Success.

4.2 PERFORMANCE

The key performance metrics for the system are network reliability, jitter when control logic runs each loop, reaction time of the system, the maximum number of sensors and actuators supported by the software, and the Platform CPU overhead.

4.2.1 Network Reliability

To measure network reliability the Platform was run across four flight computers (one Control Node and three Device Nodes) with the maximum allowed message sizes over a 24-hour period. The number of delayed or dropped messages was then measured. Over the 24-hour period, approximately 52 million messages were sent between flight computers. Of these messages, 2 were delayed and 1 was dropped, resulting in a network reliability of 99.999994%. Neither the data synchronization deadline nor the 10ms loop deadline was ever missed.

4.2.2 Jitter

Jitter was measured to understand the timing reliability on each node. Ideally the control logic would run exactly every 10ms. However, variability in network timing and CPU scheduling results in slight timing changes each run. To measure jitter control logic was run on each node. This control logic read the current time and compared it to the previous time it ran. The measured jitter is shown in Figure 14 for the control logic running on the Control Node and Figure 15 for the control logic running on the Device Nodes.

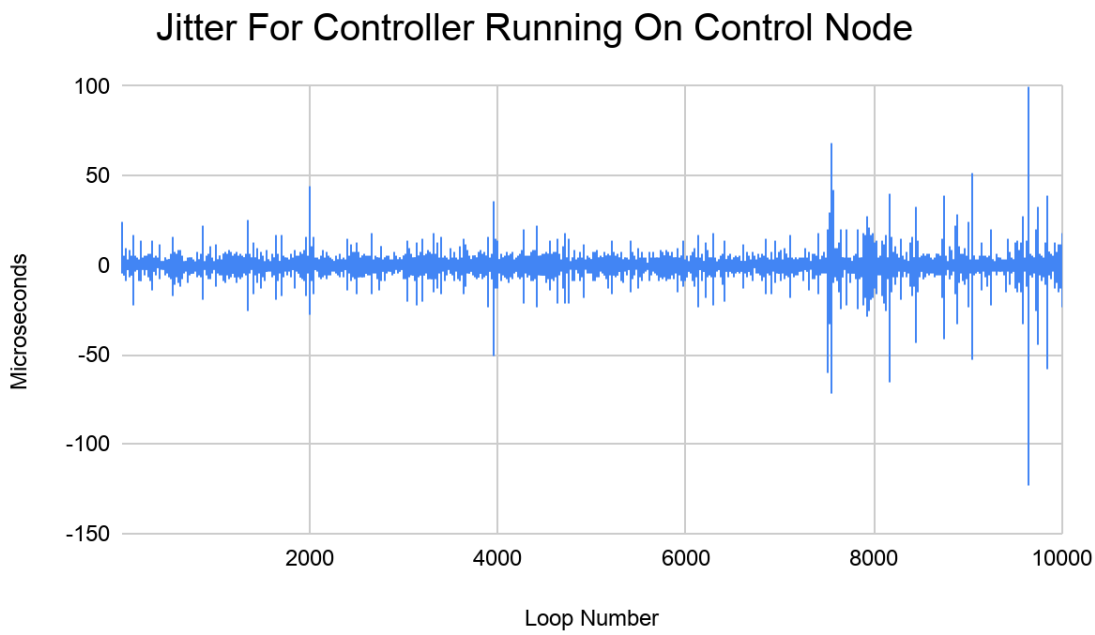

Figure 14: Control Node Jitter.

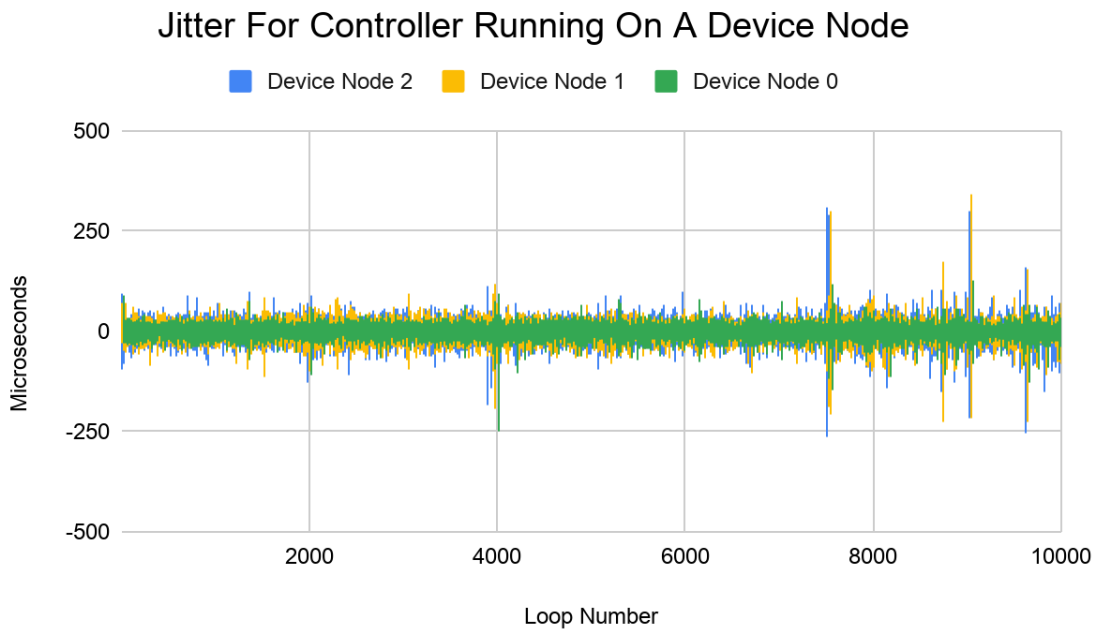

Figure 15: Device Node Jitter.

The timeseries data and aggregated results (Table 2) show that the maximum jitter for control logic running on any node is 265us. This is a maximum variability of 2.7% on the 10ms loop. The jitter is about twice as high for logic running on Device Nodes as compared to logic running on the Control Node. This is because the Control Node loop starts based on a hardware timer, whereas the Device Node loops start when they receive a network message from the Control Node. Dependency on the network introduces more timing variability.

	Control Node	Device Node 0	Device Node 1	Device Node 2
Avg Jitter	2.8 us	14.4 us	18.6 us	19.9 us
Max Jitter	123.8 us	251.3 us	228.8 us	265.2 us
Max Jitter (% of Loop)	1.2%	2.5%	2.3%	2.7%

Table 2: Platform Jitter Results.

One edge case not captured here is when a message from the Control Node to the Device Node is dropped. This will cause an additional spike jitter in the Device Node. Discussions with the subsystem teams concluded that the jitter both in the nominal case and during the occasional message drop is acceptable.

4.2.3 Reaction Time

Reaction time is defined as the time between new sensor data being read and an updated actuator value being written based on the new sensor data. While the sensor and actuator device drivers run on the Device Node connected to the sensor and actuator hardware, the control logic that uses the sensor data and provides an updated actuator value may not. If all sensors and actuators used by the control logic are connected to a single Device Node, the control logic can run on that same Device Node. Otherwise, the control logic must run on the Control Node, which has access to all sensors and actuators connected to the entire system. When control logic is running on a Device Node, the system reaction time is extremely fast as there is no dependency on sensors or actuators running on other nodes. For control logic running on the Control Node, the reaction time is slower as the sensor and actuator data must be communicated between nodes.

The minimum possible reaction time for both scenarios was measured by running control logic on the Control Node and each Device Node with no other logic running.

The average was calculated by assuming the full 10ms software loop was being used on each node, with reading sensors, running control logic, and writing actuators taking up equal parts of the loop. The maximum was calculated based on the worst case scenario that a network message was dropped (Table 3).

	Control Node	Device Node
Min Rxn Time	20.29 ms	.08 ms
Avg Rxn Time	25 ms	5 ms
Max Rxn Time	40 ms	10 ms

Table 3: Platform Reaction Time Results.

4.2.4 Maximum Number of Sensors and Actuators

The Platform is required to support up to 100 sensors and 100 actuators. Meeting this requirement is dependent on three factors:

1. Sufficient *network bandwidth* to synchronize the sensor and actuator data between the Device Nodes and the Control Nodes each loop.
2. Sufficient *physical I/O* available across the flight computers.
3. Sufficient *CPU time* to read and write the sensors and actuators

With respect to network bandwidth, assuming each sensor and actuator requires an average of 64 bits of data, the required network bandwidth is 1.3 Mbps ((100 sensors + 100 actuators) x 64 bits x 100 loops per second). The Platform supports a maximum of 49.15 kilobits of data to be transferred between flight computers each 10ms loop. With 100 loops per second the Platform supports a network bandwidth of 4.9 Mbps. This satisfies the network bandwidth requirement to support 100 sensors and 100 actuators.

With respect to physical I/O the four flight computers together have 112 digital I/O pins, 64 analog in pins, 16 analog out pins, and 12 serial ports readily available. While this technically means the Platform meets the requirement with 204 sensors and actuators supported directly by the flight computers, meeting the subsystem needs is dependent on the types of sensors and actuators selected. For example, if the subsystems required 100 analog in sensors, the Platform would require some modification to meet the requirement (e.g. adding a multiplexer). Similarly, if all analog input signals required differential rather than RSE support, an analog differential to RSE converter card would be needed. However, given the current breakdown of subsystem sensor and actuator types, the Platform supports the rocket's physical I/O needs.

With respect to the CPU time requirement there are four types of sensor and actuator signals used on the rocket: digital, analog, I2C, and serial (RS-232). All digital and analog I/O reads and writes go through the FPGA, which is accessed via direct reads and writes to memory. This means each access only takes a handful of microseconds. Profiling of the reads and writes shows the average access time is about 3us. The I2C protocol is also implemented using the FPGA. The protocol runs asynchronously and writes results to memory, which are then read by the CPU. So any hardware devices using I2C also require only a few microseconds of CPU time to read or write. Finally, the rocket requires three serial devices. Since all other sensors and actuators require so little CPU, even if each serial device took a full millisecond to read or write, there would be plenty of CPU time across the flight computers to support the sensors. The Platform therefore supports the CPU time required to read and write up to 100 sensors and 100 actuators.

4.2.5 CPU Overhead

While the Avionics Software Platform provides critical functionality to the Avionics Software Subsystem layer, the Platform must also minimize its CPU footprint so that the Subsystem layer has plenty of time to run the rocket's control logic and device drivers. The Platform's overhead was measured by implementing control logic that increases its spin time each loop until the 10ms deadline is missed. CLOCK_PROCESS_CPUTIME_ID was used to measure the amount of actual CPU time the control logic was able to use before a deadline was missed. The Platform's CPU overhead per node was then derived (Table 4).

	Control Node	Device Node
Max Time Control Logic Ran Before Deadline Miss	7.85 ms	9.89 ms
Platform CPU Overhead	21.5%	1.1%

Table 4: Platform Overhead Results.

With the four flight computers (one Control Node and three Device Nodes), every loop there is 40ms of CPU time to dedicate to the Platform and Subsystem layers. Across all flight computers, the Platform uses 2.48ms of this 40ms time per loop. This results in an overall Platform CPU overhead of 6.2%. Almost all of this overhead is used to synchronize data across nodes.

4.3 Requirements Verification

The Avionics Software Platform tests and performance measurements were used to verify the system's requirements. Each requirement and corresponding verification strategy and status are shown in Table 5. All but the final requirement have been verified.

The final requirement requires a static analysis tool integration and for any identified issues to be addressed before verification can be complete.

Requirement	Verification Strategy	Verification Status
The flight software shall be deterministic.	• Code Review • Unit & Integration Testing	**Pass:** Verified in code review. All unit and integration tests pass deterministically.
Network reliability shall be at least 99.999% in the nominal case.	• System Profiling	**Pass:** 99.999994% from network reliability measurements.
The flight software shall have a reaction time of no more than 50ms for high-level control logic and 10ms for low-level control logic.	• System Profiling	**Pass:** 40ms & 10ms maximums calculated, respectively, from system reaction time measurements.
The flight software shall run control logic and hardware drivers at a frequency of 100Hz.	• Integration & System Testing • System Profiling	**Pass:** Verified in integration and system testing as well as system profiling where no loop deadlines were missed.
The flight software shall support up to 100 sensors and 100 actuators.	• System Profiling	**Pass:** Required network bandwidth, physical I/O, and CPU time for sensors and actuators met.
The flight software shall support configurable system states.	• Unit, Integration, & System Testing	**Pass:** Verified in unit tests, integration tests, and LED System Test.
The flight software shall support user-commanded, time-based, and flight data-based state transitions.	• Unit, Integration, & System Testing	**Pass:** Verified in unit tests, integration tests, and LED System Test.

Table 5: Continued on next page.

The flight software shall provide a mechanism for streaming snapshots of the rocket's state with a latency of no more than 30ms.	• Integration & System Testing • System Profiling	**Pass:** Verified in integration tests, LED System Tests, and system profiling scripts, where telemetry is sent to Ground Node every loop with a nominal latency of up to 10ms (20ms if a telemetry message is dropped).
All flight software shall detect and handle all software errors such that the system transitions to a predefined state.	• Code Review • Unit & Integration & System Testing • Static Analysis	**In Progress:** Verified error handling pattern followed during code review and testing. Requires static analysis to complete verification.

Table 5: Platform Requirements Verification Status.

Chapter 5: Future Work

Halcyon is currently scheduled to launch in May 2021. With the most recent iteration of Avionics Software Platform complete, the Avionics Software team has shifted focus on developing the Avionics Software Subsystem layer, a hardware-in-the-loop test platform for testing and qualifying the full avionics software system, and a mission control system for operators to interface with the rocket. However, the Platform will continue to be iterated on based on continual feedback from testing and subsystem integrations. In particular, there are four areas that will be worked on before the 2021 launch.

5.1 Operator Command Rule System

The Platform's software to support operator commands will currently execute any provided command. This is problematic because it exposes the rocket to operator error and would allow an operator to command the rocket during flight, which is not permitted by the range officers that provide oversight of rocket launches. A configurable rule system is therefore being developed. Depending on the state the rocket is in, commands will need to be whitelisted to be considered valid.

5.2 Control Logic Off Mode

Ground operators have a need to have complete control of the rocket's actuators in some pre-launch procedures (e.g. manually controlling actuators during fueling). The current implementation of the Platform facilitates this by providing operators with direct command of actuators. However, manually written actuator commands may conflict with automated control logic. While this conflict can be handled in the Subsystem layer, a cleaner and more scalable solution is to resolve this conflict in the Platform layer.

5.3 Static Analysis

A static analysis tool will be integrated to further verify the flight software. In particular a static analysis tool will be used to surface unhandled exceptions and verify that the flight software follows the established safety-critical coding guidelines.

5.4 Tolerating an Unresponsive Flight Computer

During the Platform development a flight computer failed. The board powered off suddenly and could not be powered back on. While this was the only board failure seen across four flight computers over a 12-month period, this type of failure during flight would be catastrophic to the rocket. The root cause determined by National Instruments was an electrical component failure, which allowed 12V into a 1.8V power circuit causing the board to be unable to be powered on. This type of manufacturing issue, which occurred after months of successful usage of the board, would be very difficult to prevent or predict.

Future iterations of the Platform will likely tolerate an unresponsive flight computer. The current design in the works is to use a primary/backup model, which would cost less in terms of weight than a 3-string solution. Since the Platform has plenty of I/O and CPU available to the rocket, one possible design is to repurpose the four flight computer roles so that there is a primary Control Node and Device Node pair and a backup pair. This would allow the system to tolerate a single node becoming unresponsive without adding nodes to the rocket.

Since the Platform currently sends the rocket's state to the Ground Node via telemetry, maintaining state in a backup pair would be trivial, as the primary pair could send the rocket's state to the backup pair as well. This synchronization step would also be used to detect a primary failure. If the state is not sent after some timeout, the backup

would know the primary had failed. To protect against false positive failure detection, the backup would kill power to the primary. The backup node would then initialize its software components to run based on the most recently saved state. Some Platform components would need to be updated to support mid-flight initialization.

The trickiest part to the primary/backup model is transitioning control to the backups. There are many edge cases depending on the exact timing of the failure. For example, what if the primary Control Node sends an actuator command to the primary Device Node and then dies before the backup Control Node receives the updated state? Is it ok if the backup Control Node resends the command? These types of cases would need to be thoroughly evaluated to determine the desired system behavior and how to modify the Platform to accomplish this desired behavior.

Chapter 6: Conclusion

The Texas Rocket Engineering Lab is on track to launch Halcyon to the Karman Line in 2021, with the Avionics Software Platform providing a distributed flight software system that delivers data handling, control logic abstractions, thread scheduling, time management, and software error handling functionality. The Platform delivers this functionality using a configurable, scalable, and modular architecture that is designed to be easily tested and modified. The Platform executes control logic and device drivers at 100Hz with a network reliability of 99.999994%, an average jitter of 2.2%, an average system reaction time of 5ms for control logic running on Device Nodes and 25ms for control logic running on the Control Node, and support for over 100 sensors and actuators. This functionality and performance is delivered by the Avionics Software Platform at a cost of just 6.2% of the total CPU time.

References

[1] UT Austin Cockrell School of Engineering. “UT Launches New Rocket Engineering Program Thanks to $1M Gift from Firefly Academy.” 5 Dec. 2018, www.engr.utexas.edu/news/archive/8575-firefly-at-ut-1m-gift-rocket-engineering. (accessed 4 Apr. 2020)

[2] Base 11 Space Challenge. “Base11 Space Challenge: $1 MILLION+ STUDENT ROCKETRY PRIZE.” base11spacechallenge.org/. (accessed 4 Apr. 2020)

[3] Carlow, Gene. “Architecture of the Space Shuttle Primary Avionics Software System.” Communications of the ACM 27.9 (1984): 926–936. Web.

[4] Caulfield, J.t. “Application of Redundant Processing to Space Shuttle.” IFAC Proceedings Volumes 14.2 (1981): 2461–2466. Web.

[5] Hosein, Jinnah. “Engineer the Future.” USENIX, 7 Dec. 2016, www.usenix.org/conference/lisa16/conference-program/presentation/hosein. (accessed 4 Apr. 2020)

[6] NASA. “SpaceX CRS-1 Mission Press Kit.” NASA, Oct. 2012, www.nasa.gov/pdf/694166main_SpaceXCRS-1PressKit.pdf. (accessed 4 Apr. 2020)

[7] Carancho, Ted. “Millennium Space Systems' ALTAIR™ Satellite.” Millennium Space Systems' ALTAIR™ Satellite, National Instruments, www.ni.com/en-us/innovations/case-studies/19/millennium-space-systems-altair-satellite.html. (accessed 4 Apr. 2020)

[8] Edge, Jake. “ELC: SpaceX Lessons Learned.” [LWN.net], 13 Mar. 2013, lwn.net/Articles/540368/. (accessed 4 Apr. 2020)

[9] National Instruments. “LabVIEW 2019 FPGA Module Known Issues.” Bugs, 17 May 2019, www.ni.com/en-us/support/documentation/bugs/19/labview-2019-fpga-module-known-issues.html#660205_by_Date. (accessed 4 Apr. 2020)