Dedication

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

May you never have to read this.

Acknowledgements

I would first like to thank my mentor, Professor Chris Rossbach. Chris' Advanced Operating Systems class and subsequent mentorship inspired me to 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

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, implementation, and verification of the Avionics Software Platform are presented in this paper.

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 paper.

1.3 PAPER STRUCTURE

The paper is structured as follows. First, the requirements for the Avionics Software Platform are presented. Second, the Platform's architecture and component design are described. This chapter includes the rationale behind high-level system decisions such as the distributed architecture, flight computer, and network protocol. Third, the Platform implementation details are discussed. Fourth, the testing and performance measurements of the 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 5.

- 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, system architecture, assignment of functions to Platform software components, and the system initialization and loop sequences.

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.

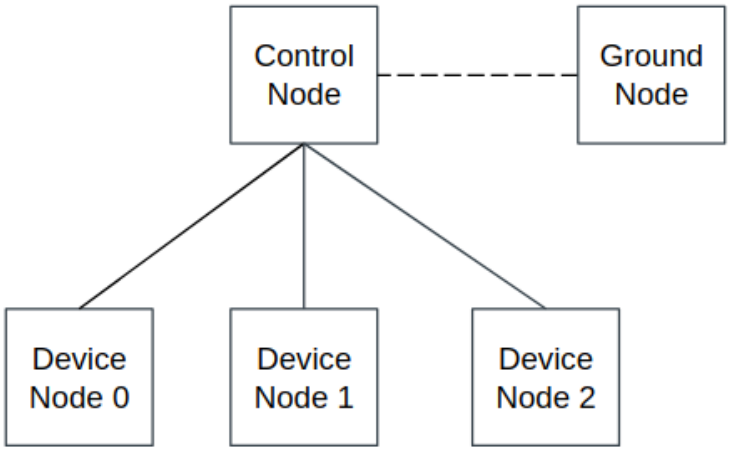

Figure 1: Parent-Child Distributed Architecture.

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.

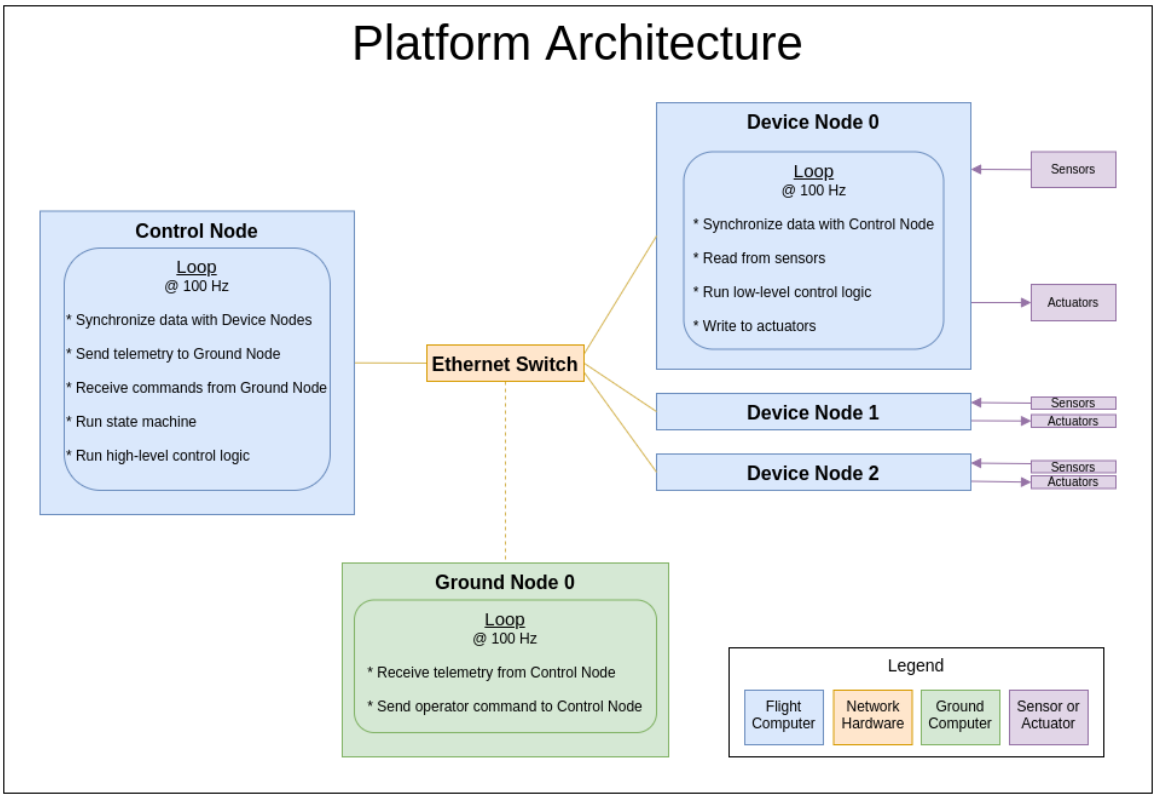

Figure 2: Platform Architecture.

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.

3.4 Function Assignment to 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.

3.4.1 Data Handling

The Platform is responsible for the management of all data on the rocket. This includes giving control logic a way to read and write the rocket's data, synchronizing data across flight computers, providing an API for moving data between software and hardware components, and providing a mechanism for streaming telemetry. The following Platform software components accomplish this function.

3.4.1.1 Data Vector

The Data Vector is a configurable data abstraction that stores all of the rocket's data in memory and provides all other software components with read and write access to the data. A Data Vector instance runs on each node, with the Data Vectors running on Device Nodes containing only data relevant to that node, and the Data Vector running on the Control Node containing all of the rocket's data. By centralizing data storage on the rocket, the Data Vector simplifies system monitoring, software debugging, and the streaming of data snapshots to the Ground Node for telemetry purposes. The Data Vector is configurable, expecting a list of element names and their corresponding data types and initial values on initialization. To synchronize data across flight computers, groups of related elements called Regions are sent between nodes.

3.4.1.2 Network Manager

The Network Manager is responsible for handling all communications between nodes over the flight network. Communications are done over an Ethernet network using UDP for low-latency messaging. The Network Manager is configurable, expecting a list of nodes on the network and valid communication channels on initialization.

3.4.1.3 Data Synchronization Protocol

At the top of each 100Hz loop, the Control Node synchronizes data with the Device Nodes and Ground Node. The Data Synchronization Protocol defines exactly when and how this synchronization occurs so that the synchronization is done efficiently and reliably.

3.4.1.4 FPGA Interface

The FPGA Interface is a C API that gives the CPU read and write access to the FPGA. This access is required to read from and write to the sbRIO's native digital and analog I/O. Once the FPGA is programmed using LabVIEW, the API is automatically generated using a LabVIEW plugin.

3.4.2 Control Logic Abstractions

The Platform is responsible for providing control abstractions to the Avionics Software Subsystem layer. These abstractions allow the subsystem logic to run on the Platform in a standardized and configurable way. There are four control logic abstractions.

3.4.2.1 State Machine

The Platform's State Machine software component is a configurable state machine that manages the high-level state of the rocket. Each state in the State Machine consists of an action sequence and a list of legal transitions. The action sequence is a list of times and corresponding actions to take place at that time. Each action is a write to the Data Vector, which can be anything from raising an error flag to setting an actuator value. The list of legal transitions is a list of conditions (e.g. altitude is greater than 100 km, error flag is true) and target state to transition to if that condition is met. The Data Vector is read to determine if the transition condition has been met. On initialization the State Machine is built from a provided config, which defines the initial state and a list of states with their corresponding action sequence and legal transitions.

3.4.2.2 Controllers

The Platform's Controller software component is a base class that is extended by the Avionics Software Subsystem layer to implement control algorithms. These algorithms read system data from (e.g. sensor values) and write data to (e.g. actuator commands) the Data Vector. Controllers accept a config on initialization for providing algorithm parameters that frequently require tuning, such as PID terms or hysteresis limits. The base Controller class requires a runEnabled () and runSafed () method to be defined by the derived classes. This is a safety technique that forces the Subsystem layer to define what it means for each Controller to be in a safed state, so that a Controller cannot accidentally cause harm to someone. An example of where this is used is in the RecoveryController. The Recovery subsystem deploys parachutes using black powder charges, which pose a danger to people if they are deployed at the incorrect time. The RecoveryController is therefore put into its safe mode until the rocket has been launched to prevent the Controller from accidentally deploying at an incorrect and unsafe time.

3.4.2.3 Devices

The Platform's Device software component is a base class that is extended by the Avionics Software Subsystem layer to interface with specific sensor and actuator hardware devices via the FPGA Interface. Sensor Devices read input values from sensor hardware and write them to the Data Vector. Actuator Devices read control values from the Data Vector and write them to actuator hardware. Devices accept a config on initialization for the client to provide the interface logic, such as calibration values or I/O pin numbers. The base Device class requires a run () method to be defined by the derived classes.

3.4.2.4 Command Handler

The Platform's Command Handler provides two avenues for ground operators to command the rocket. First, it accepts LAUNCH and ABORT commands, which are read by the State Machine. If the current state has a defined transition dependent on either of these commands, the State Machine will transition to the appropriate launch or abort state. Second, the Command Handler provides ground operators with direct access to the Data Vector. The WRITE command allows operators to directly write to the Data Vector and control things like an actuator value or whether a Controller is enabled or safed. Writing directly to the Data Vector is a powerful capability that is only intended to be used in off-nominal pre-launch or recovery scenarios and will be disabled during flight.

3.4.3 Time Management

The Platform is responsible for providing a monotonic and linear clock on each flight computer. Monotonic means that time is always increasing, and linear means that the clock's rate of change is constant. Furthermore, since each flight computer has its own clock, the clocks must be synchronized so that timestamps taken from one flight computer can be used on another flight computer. This is particularly important for detecting stale data. The following two Platform components accomplish this function.

3.4.3.1 Clock Synchronization

The Platform's Clock Synchronization software component is responsible for synchronizing all flight computer clocks to an accuracy of +/- 100ms. This is the level of accuracy required by the Avionics Software Subsystem layer, which needs to be able to detect if sensor data is stale.

3.4.3.2 Time Module

The Time Module provides flight software with direct access to time in nanoseconds since the Epoch. The underlying clock used in the Time Module's implementation is the same clock synchronized by the Clock Synchronization component. On initialization, the Time Module will fail if the underlying clock is anywhere near overflowing, which would cause the clock to jump back in time and have undefined results.

3.4.4 Software Error Handling

Software can fail in many ways, and the Platform is responsible for detecting, surfacing, and handling when an error occurs. The following software component delivers this functionality.

3.4.4.1 Flight Software Error Handling Framework

The Flight Software (FSW) Error Handling Framework provides a software pattern that all other flight software follows. This pattern requires all functions to return a success or error status code and all call-sites to check and handle the returned code. For library or system calls not written by the Avionics Software team, the call-site must verify the function succeeded as specified by the API in use. The FSW Error Handling Framework provides a status enumeration and helper functions for handling errors. All software errors are returned up the call stack to the Control/Device/Ground Main components, which handle them in one of three ways. If a critical error occurs during initialization, the program reports the error and exits. If a critical error occurs after system initialization, an error flag is raised and the State Machine transitions the system to an abort state, which then executes an abort sequence. If a non-critical error occurs after system initialization, the error is logged to the Data Vector.

3.4.5 Thread Scheduling

The Platform is responsible for providing CPU access to the flight software application so that it can run at 100Hz. This includes providing the main functions that initialize and run the flight software and making sure non-critical threads do not take CPU time from the flight software application. This function is provided by four software components.

3.4.5.1 Thread Manager

The Platform's Thread Manager component has two responsibilities. First, on initialization it configures the priorities of existing operating system threads so that the flight software application can run without other non-critical threads taking over the CPU. Running at 100Hz (every 10ms), the flight software cannot tolerate unnecessary threads temporarily taking over the CPU and forcing the flight software to pause for hundreds of microseconds at a time. Second, the Thread Manager provides thread creation functions used to create one-off as well as periodic threads. Periodic threads are used to schedule the flight software logic at 100Hz.

3.4.5.2 Control Node Main

The Control Node Main is the entry point for the flight software running on the Control Node. This Platform component is the glue of the flight software, responsible for initializing all other Platform components and running a periodic thread to execute the Control Node's loop at 100Hz.

3.4.5.3 Device Node Main

The Device Node Main is the entry point for the flight software running on each of the Device Nodes. Similar to the Control Node Main, this Platform component is the

glue of the flight software, responsible for initializing all other Platform components and running a thread that is loop-synchronized with the Control Node's periodic thread. This loop synchronization is done via a trigger, meaning the Control Node sends a trigger that tells the Device Node to start its loop. Since the Control Node sends a data synchronization message to each Device Node at the top of each loop, this message is also used as the loop synchronization trigger.

3.4.5.4 Ground Node Main

The Ground Node Main is the entry point for the ground software running on the ground computer. This Platform component initializes the handful of Platform software components running on the ground computer and runs an infinite loop that receives telemetry from and sends operator commands to the Control Node.

3.5 System Initialization and Loop Sequences

The 15 Avionics Software Platform components work together to deliver data handling, control logic abstractions, thread scheduling, time management, and software error handling functionality to the rocket. Figure 4 shows each node's initialization sequence and Figure 5 shows the loop sequences. The FSW Error Handling framework is not included in the diagrams, as it is an implementation pattern followed by all Platform software components.

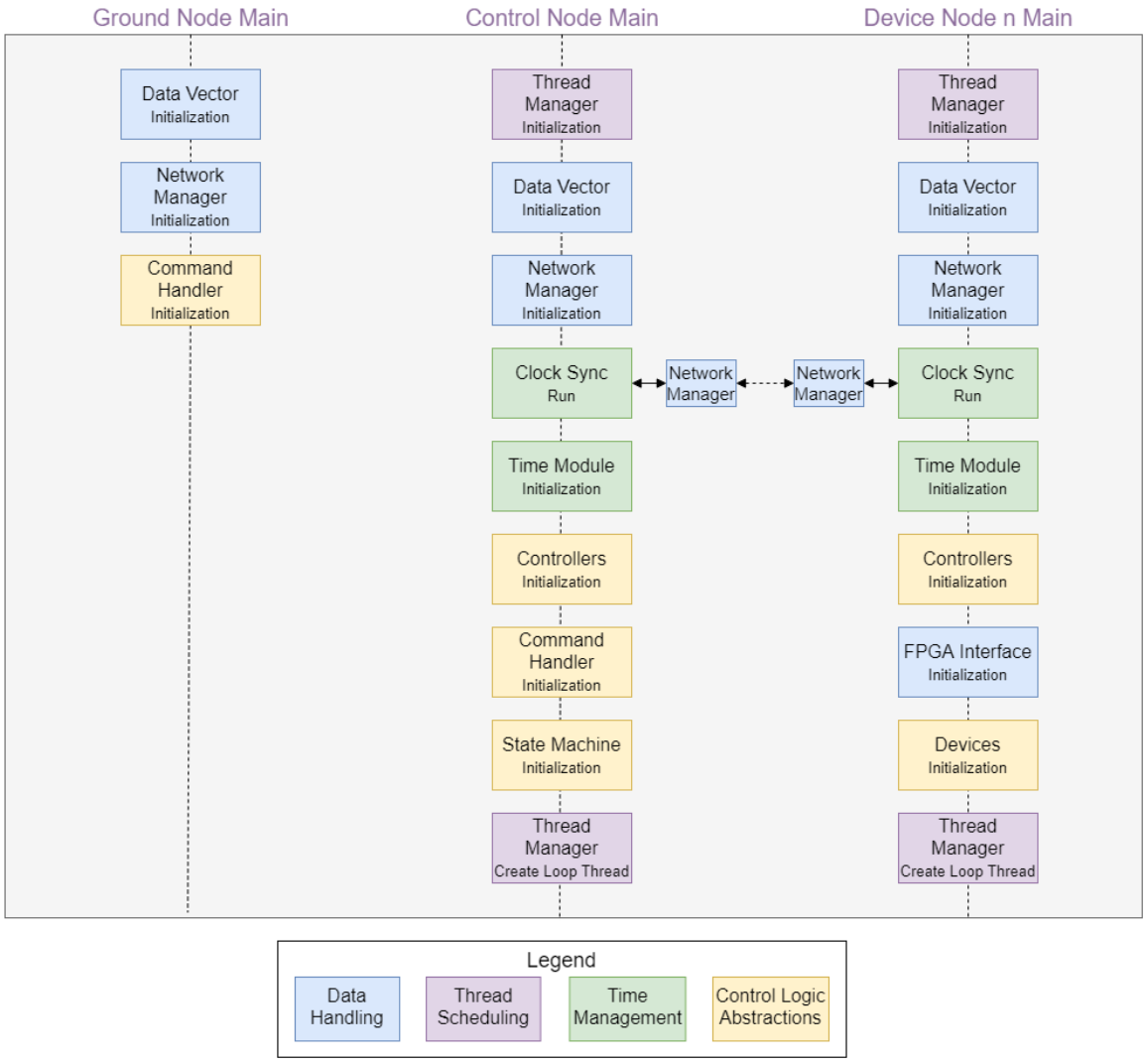

Figure 4: Flight Software Initialization Sequence Diagram.

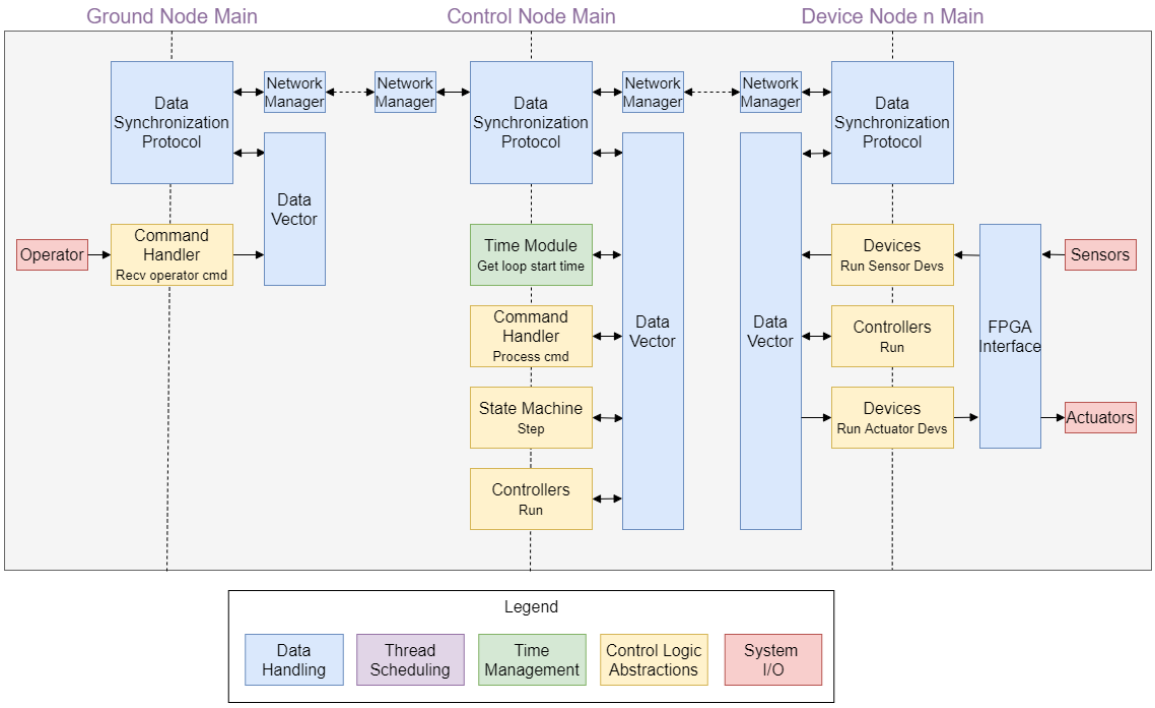

Figure 5: Flight Software Loop Sequence Diagram.

Chapter 4: Implementation

The following chapter describes the detailed implementation of each of the 15 Avionics Software Platform components.

4.1 DATA VECTOR

The Data Vector is a data abstraction that can be thought of as a vector of differently-typed data elements. Contiguous, related data elements are grouped into Regions. The grouping of elements into Regions is done to facilitate data synchronization across flight computers. By grouping elements based on whether they will be sent to another node or received from another node, data synchronization is accomplished by passing Data Vector Regions between computers. Figure 6 shows a simple example Data Vector with two Regions, with each Region containing two data elements.

Region	device_node_0_to_control_node		control_node_to_device_node_0	
Element	lox_tank_temp_k	lox_valve0_fdbck	lox_valve0_control	engine_ign_control
Value	66.23	true	true	true

Figure 6: Example Data Vector.

The underlying implementation of the Data Vector uses a byte buffer to store the raw data compactly. Supporting data structures are built on initialization to facilitate efficient read/write access to the byte buffer. These supporting data structures map each data element and Region to their starting index in the byte buffer and size.

4.1.1 Configuration

The Data Vector config is a struct passed by the client on initialization. The config is a list of Regions, with each Region containing a list of element names, types, and initial values. Figure 7 shows an example config that would be used to generate the Data Vector in Figure 6. The Data Vector supports 11 data element types: uint8_t, uint16_t, uint32_t, uint64_t, int8_t, int16_t, int32_t, int64_t, float, double, and bool. On initialization the config is first verified against the following rules:

- Config not empty.
- A Region must have at least 1 element.
- Duplicate Region names are not permitted.
- Duplicate Element names are not permitted.
- Valid element type.

Region	Element	Type	Initial Value
device_node_0_to_control_node	lox_tank_temp_k	float	0
	lox_valve0_fdbck	bool	false
control_node_to_device_node_0	lox_valve0_control	bool	false
	engine_ign_control	bool	false

Figure 7: Example Data Vector Config.

4.1.2 Time & Space Complexity

The Data Vector is the most commonly used Platform component. Almost every other flight software component uses the Data Vector, so it is critical that the read and write operations are efficient in terms of time so that accessing data does not use up a significant amount of the 100Hz flight software loops. The Data Vector's primary

operations include reading and writing a single data element, a Region (for data synchronization purposes), and the entire Data Vector (for telemetry purposes).

While the Data Vector must be time efficient once the flight software begins its 100Hz loops, there is no timing constraint on initialization, which is only done once at system start. There is no space constraint as each flight computer has more than enough memory. The lack of a timing constraint on initialization and space constraint throughout allows the Data Vector to spend time and space on initialization to facilitate efficient reads and writes later on.

On initialization the Data Vector builds supporting data structures mapping elements and Regions to their starting index in the byte buffer and size. These structures are implemented using the unordered_map container, which is implemented using a hash table. As such, access to a single item in these maps takes constant time on average.

Using these maps, a read or write of a single data element in the Data Vector is done in constant time. The operation first gets the element's starting index and size by accessing the elements supporting data through a constant-time access to the unordered_map. Then, the Data Vector's byte buffer is read/written directly in constant time. Reading and writing a Region or the entire Data Vector takes time linear to the size of the data being read or written. Since Regions and the Data Vector overall consist of contiguous elements, they are read by copying data from a single segment of the byte buffer to the copy buffer and vice versa for a write.

4.1.3 Copying Data vs. Passing Pointers

The operations to read a Region or read the entire Data Vector were initially implemented to run more efficiently (constant time) by returning pointers to the underlying data instead of a copy of the data. However, passing around pointers makes

software more difficult to follow and more prone to pointer-related bugs. It is much simpler, although less efficient, to instead return a copy of the underlying Data Vector data. To better understand the efficiency tradeoff, an experiment was run to measure the time to copy 2kB of data (the maximum expected data size) to copy buffers initialized using various strategies. The results are shown in Table 2. A dynamically allocated copy buffer that does not have its size initialized before the copy takes the largest amount of time. This is likely due to the copy buffer repeatedly being resized and possibly moved in memory multiple times to fit the entire 2kB of data. Initializing the copy buffer's size before the copy significantly reduces the time to copy. By statically allocating the copy buffer, time to copy is reduced further.

Copy Buffer Allocation	Copy Buffer Size Initialized	Avg Time To Copy
Dynamic	No	397.1 us
Dynamic	Yes	5.3 us
Static	Yes	2.4 us

Table 2: Byte Buffer Copy Experiment Results.

As a result of this experiment, the operations to read a Data Vector Region or the entire Data Vector were rewritten to return a copy of the underlying data rather than a pointer to that data. Callers are required to have the copy buffer's size initialized to the expected data size. The operations return an error if the buffer is not the same size as the data to be copied. Since the Data Vector does not allow adding/removing elements or changing the size of elements or Regions after initialization, any copy buffers used to store copied data from the Data Vector can be statically allocated and initialized with the

expected buffer size once. To facilitate initializing these copy buffers, operations to get the size of Regions and the entire Data Vector were added.

4.1.4 Data Vector Logger

The Data Vector Logger object was created to log Data Vector data to a csv-formatted file. The Logger is run on the Ground Node during integration and system tests so that each telemetry snapshot sent from the Control Node is logged. These logs are used to verify system behavior during the tests.

4.2 NETWORK MANAGER

The Network Manager is responsible for all communications between flight computers and runs on each node. The flight network is represented as an undirected graph, where each node in the graph is a computer on the network and each edge is a bidirectional communication channel defined by an IP-pair and a port (Figure 8).

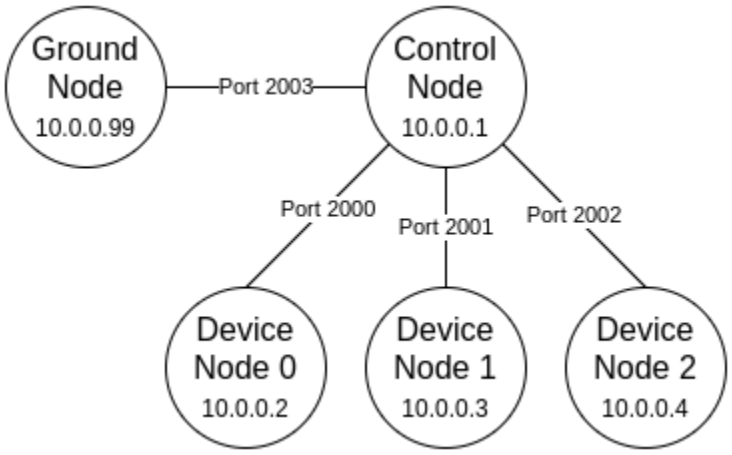

Figure 8: Flight Network Graph.

On initialization the Network Manager creates a UDP socket per channel, which is used to send and receive data over the flight network. To send a message the caller specifies a destination node and passes in a byte buffer to send. The Network Manager uses supporting data structures to find the relevant socket and send the message. To receive a message the caller uses one of three receive methods:

- *recvBlock:* Block until a message is received from a specified node.
- *recvNoBlock:* Attempt to receive a message from a specified node. Returns immediately, even if no message is received.
- *recvMult:* Attempt to receive multiple messages from multiple nodes for a specified amount of time.

To facilitate system health monitoring and debugging, on every successful message send and receive event the Network Manager increments a Data Vector element counting the number of messages sent and received, respectively.

4.2.1 Configuration

On initialization the Network Manager is passed a config defining the flight network nodes and channels. Figure 9 shows an example config that would generate the flight network graph in Figure 8. The config is verified against the following rules:

- All nodes have valid static IP's.
- All IP's are unique.
- Each channel has a valid IP-pair.
- Each channel uses a valid port number.

Nodes	*Node*	*IP*	
	control_node	10.0.0.1	
	device node 0	10.0.0.2	
	device_node_1	10.0.0.3	
	device node 2	10.0.0.4	
	ground node	10.0.0.99	
Channels	*First Node*	*Second Node*	*Port*
	control node	device node 0	2000
	control_node	device_node_1	2001
	control node	device node 2	2002
	control node	ground node	2003

Figure 9: Example Network Manager Config.

4.3 DATA SYNCHRONIZATION PROTOCOL

The Data Synchronization Protocol synchronizes Data Vectors across flight computers, sends telemetry to the ground computer, and receives operator commands from the ground computer. The unit of synchronization is a Data Vector Region. Each Device Node's Data Vector has two Regions. One Region contains all data written to by a Device Node and read by the Control Node, and the second Region contains all data written to by the Control Node and read by the Device Node. During data synchronization, the Device Node sends the first Region to and receives the second Region from the Control Node. The Ground Node has a single Region, where operator commands are written to. This Region is sent to the Control Node during data synchronization. The Control Node's Data Vector is the superset of all Data Vectors

running on a flight or ground computer, and therefore contains two Regions per Device Node, a single Region to for the Ground Node's operator command data, and a Region containing data only relevant to the Control Node (e.g. loop number, error logging). Overall, the Control Node contains the following Regions, where x is the Device Node number:

- *CN_Reg:* Data elements relevant only to the Control Node.
- *CN_to_DNx_Reg:* Data elements written to by the Control Node and used by Device Node x. There are n number of these Regions, where n is the number of Device Nodes on the flight network.
- *DNx_to_CN_Reg:* Data elements written to by Device Node x and used by the Control Node. There are n number of these Regions, where n is the number of Device Nodes on the flight network.
- *GN_to_CN_Reg:* Data elements written to by the Ground Node and used by the Control Node. These are all elements related to an operator command.

There is no CN_to_GN_Reg, since the Control Node sends the entire Data Vector to the Ground Node as telemetry.

The Data Synchronization Protocol is coordinated by the Control Node and runs at the top of each node's 100Hz loop (Figure 10). The protocol starts when the Control Node sends the relevant CN_to_DNx_Reg data to each Device Node. The Device Nodes are blocked at the top of each loop waiting for this message. Once the Device Node receives the Region, they each respond with the relevant DNx_to_CN_Reg data and continue executing the rest of their loop. After sending a Region to each Device Node, the Control Node sends a copy of its entire Data Vector to the Ground Node as telemetry and checks if the Ground Node sent the GN_to_CN_Reg data message. This will only be

sent if an operator has initiated a command. The Control Node then waits for a specified amount of time for the Device Nodes to respond with their Regions. If the Device Node Regions are not received before the timeout expires, a missed message error is logged. A timeout is required to ensure the Control Node's loop does not become blocked on a delayed or dropped message.

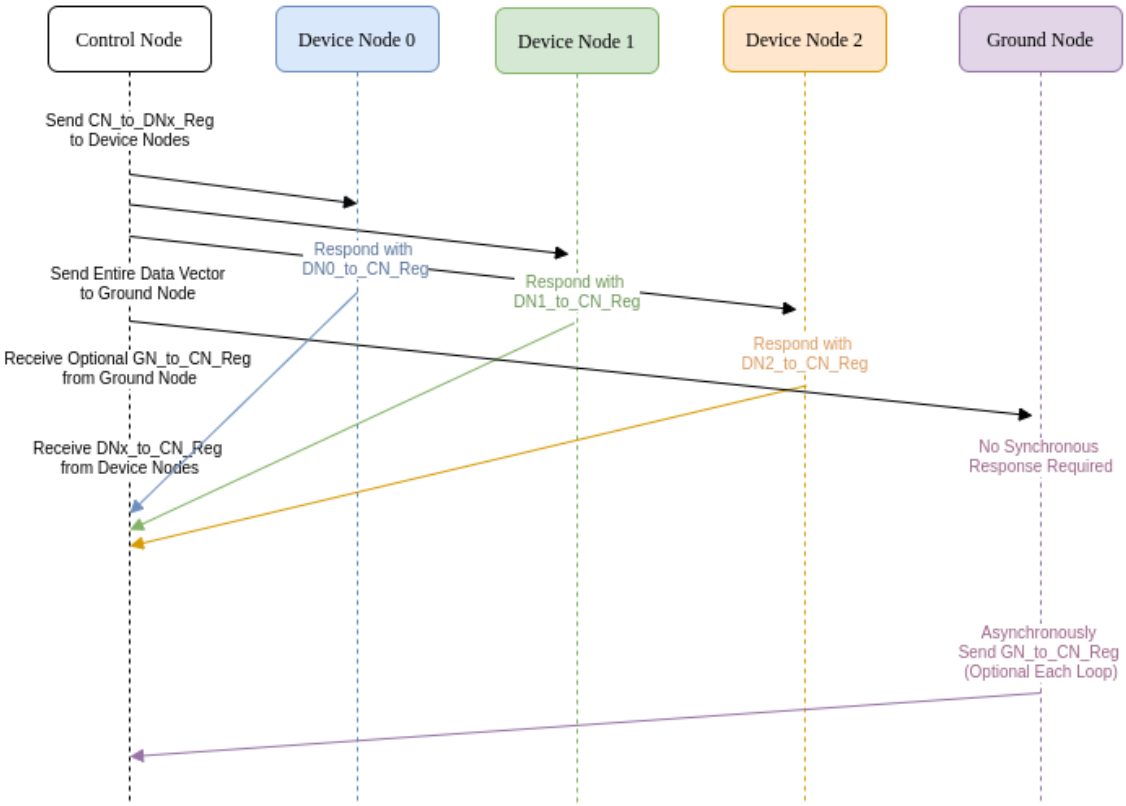

Figure 10: Data Synchronization Protocol Sequence Diagram.

4.3.1 Investigating & Resolving Data Synchronization Time Spikes

When stress testing the Data Synchronization Protocol, spikes in time to complete the synchronization were seen (Figure 11). While sync time would on average take less than 2ms to complete, every so often a multi-second spike was seen. In some cases 20

second spikes were seen. These message delays resulted in a network reliability of 99.98%, which does not meet the Avionics Software Platform's requirement that network reliability be 99.999%.

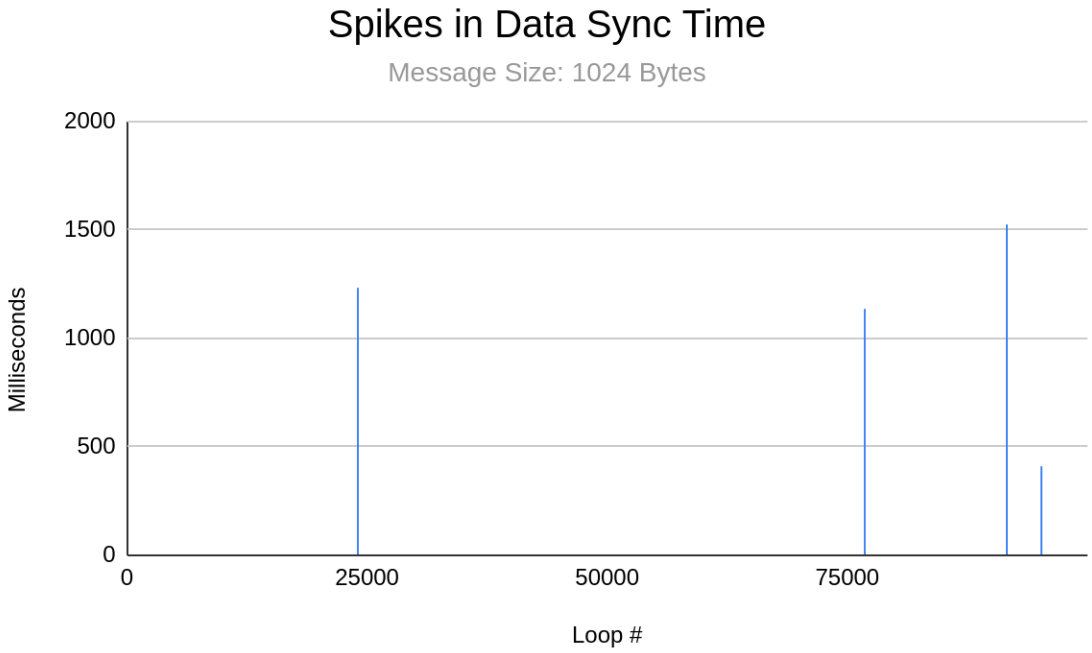

Figure 11: Spikes in Time to Complete Data Synchronization.

The path a message takes when it is sent from one node to another over the flight network is shown in Figure 12. The flight software application tells the operating system (OS) to send the message. The OS then passes the message to the computer's network interface controller (NIC) hardware, which then sends the message over the wire to the Ethernet switch. The switch passes the message to the receiving node's NIC, the NIC passes the message to the OS, and the OS passes the message up to the receiving node's flight software application.

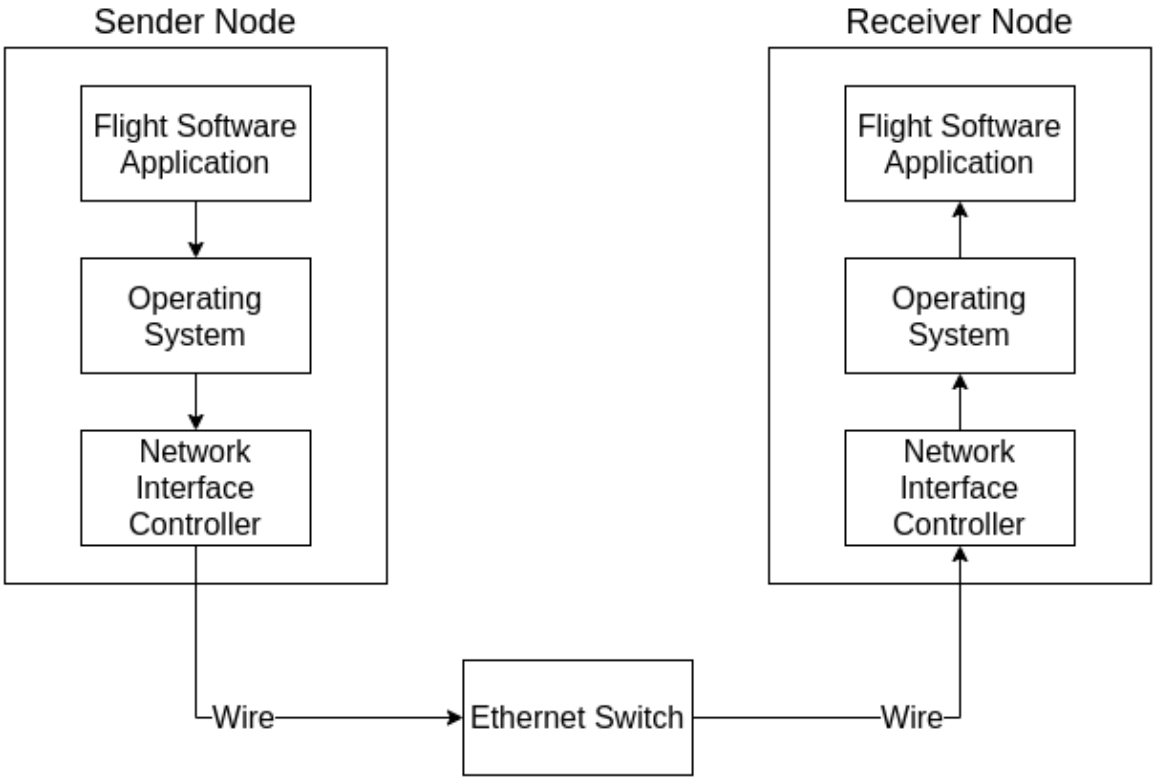

Figure 12: Flight Network Message Path.

The first step in investigating the root cause of the spikes was to add timestamps to each step in a message's path. These granular timestamps showed that the spikes were always occurring on the receiver's end, when the flight software application called the receive system call. So messages were being successfully sent by the sender node, but sometimes the receiver node would be stuck for seconds waiting for the message. This meant that the issue was therefore caused by one of the following:

- Wire
- Switch
- Receiving node's NIC
- Receiving node's OS

The Linux tcpdump utility was used next to further narrow down the root cause. Although the flight software application was seeing a delay on the receive call, tcpdump showed that the message had in fact reached the NIC successfully without delay. The tcpdump utility reads packets from the NIC directly, so this further narrowed down the issue to either the NIC receiving but not passing the message along to the OS or the OS receiving but not passing along the message to the application [9].

A possible reason the OS would cause a delay is if some other process was running on the CPU, causing the flight software application to be starved. To investigate this as a potential root cause all processes that ran while the flight software application was blocked on the message receive call were logged via a script monitoring the /proc/<pid> files. No unexpected processes were found to be running, so CPU starvation was eliminated as the root cause.

Finally, it was discovered that the Gigabit Ethernet Controller (the flight computer's NIC) had a known issue where messages could get stuck in the receive queue [10]. There is no known software workaround, however the message becomes unstuck when the next Ethernet message comes in.

As a result of this investigation, the Network Manager was modified to send a noop message (a 1 byte UDP message) after every data message sent. The noop message is sent to the same destination node, but to an unused port. After adding these noop messages, the spikes went away.

4.3.2 Selecting the Data Synchronization Timeout

The Data Synchronization Protocol includes a step where the Control Node waits for the Device Nodes to respond with their Data Vector Regions. To determine the appropriate timeout to use, an experiment was run measuring the time it takes to

complete the Data Synchronization Protocol as a function of Region size and which CPU core the flight software application was run on (Figure 13). The protocol was run 1M times per Region size/CPU combination.

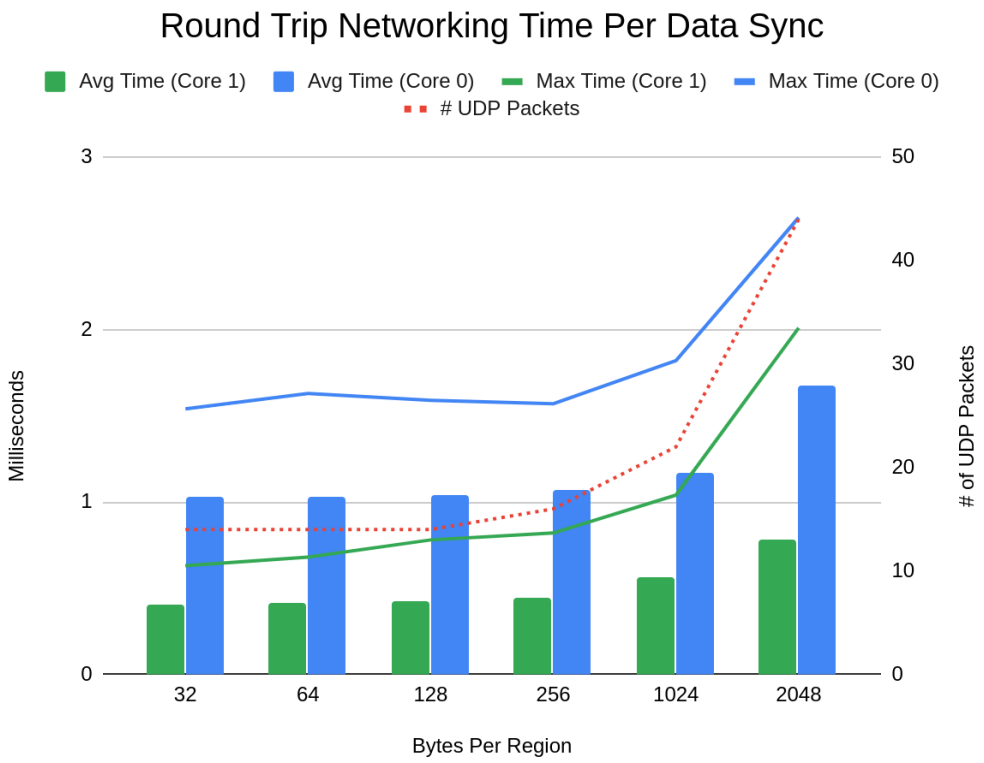

Figure 13: Data Synchronization Protocol Timing Experiment Results.

The results showed that running the flight software application on Core 1 is much more efficient than on Core 0. This is because many OS processes (in particular the process that handles incoming Ethernet messages) are higher priority than the flight software application and run only on Core 0. With the added noop messages, the Ethernet OS process has even more work to do. Running the flight software application on Core 1 therefore means less competition for the CPU. The results also show that as the size of the Region increases, the time to complete the protocol also increases. This increase is

particularly impacted if a Region size increase results in more UDP packets being sent. Each UDP packet has a maximum data payload size of 1472 bytes.

The outcome of this experiment was to fix the flight software applications to Core 1 on all flight computers, to set the maximum Data Vector Region size to 1024 bytes, and to give the Data Synchronization Protocol 2ms to complete. At an average of 4 bytes per data element, 1024 bytes allows for 256 elements per Region, which is well over what the system is expected to utilize. Setting the timeout to 2ms gives plenty of time for the Data Synchronization Protocol to complete, while still reserving the majority of each loop for executing the rest of the flight software.

4.3.3 Reliability & Tolerating Delayed and Dropped Messages

With the addition of noop messages to tolerate the NIC issue, a maximum Region size of 1024 bytes, and a 2ms window for the Data Synchronization Protocol to complete, profiling of the network (discussed in Chapter 5) shows that 1 out of every 20M messages will be delayed or dropped message. While this network reliability (99.999994%) is significantly better than the Platform's requirement (99.999%), delayed and dropped messages are still likely to occur during flight and must be handled.

Since the Device Node loops are synchronized to the Control Node's loop via the Data Synchronization Protocol, if a message sent from the Control Node to a Device Node is dropped, the Device Node will not know that the loop has begun. This means the Device Node will not send a response and will not run that loop. After discussions with the GNC team, it was verified that the system can tolerate a Device Node missing a loop once every 20M loops, so no changes were made to eliminate or reduce the frequency of this case. If a message from a Device Node to a Control Node is dropped, the Control

Node will timeout waiting for the message, log the miss, and continue with its loop. This ensures that the Control Node loop always occurs.

Delayed messages are slightly more complicated. If a message is delayed, the receiving node will have two messages in the receive queue when it attempts to receive a message the next loop. To tolerate this, each loop the Control Node receives as many messages as are in the receive queue (constrained by the timeout) and the Device Node will attempt to receive up to two messages.

4.3.4 Endianness

Since the Data Synchronization Protocol passes around Regions, which are byte buffers interpreted by the Data Vector, the protocol is sensitive to the endianness of each computer on the flight network. All flight and ground computers used thus far have a little endian architecture, so effort has not been invested in supporting big endian architectures as well. If a big endian computer is added to the flight network, the Data Vector will need to be updated so that it can interpret the byte buffer differently depending on the node's endianness.

4.4 FPGA INTERFACE

On the flight computer hardware (sbRIO-9637) the CPU is required to go through the FPGA to access all digital and analog I/O on the board. Programming of the FPGA is done using LabVIEW, and from this FPGA program the FPGA Interface is automatically generated using National Instruments' FPGA Interface C API plugin.

The FPGA Interface allows the flight software to set a digital I/O pin as input or output, set the value of an output pin, and read the value of an input or output pin. For the analog output pins, the interface allows the flight software to set the output voltage. For

the analog input pins, the interface allows the flight software to set a single pin to be a referenced single ended (RSE) input or a pair of pins to be a differential (DIFF) input. The analog input range can be set as +/- 10, 5, 2, or 1 volts, with smaller ranges allowing for higher resolution readings. The current FPGA program and FPGA Interface support digital and analog I/O, and the implementation can be extended to offload CPU computation such as running an I2C protocol, calibrations, and unit conversions.

4.5 STATE MACHINE

The State Machine component is implemented using 6 objects related to each other as shown in the UML class diagram shown in Figure 14. A single StateMachine object is associated with one or more States, and each State object is associated with a single Actions object and Transitions object. The Actions and Transitions objects encapsulate a State's action sequence and list of legal transitions, respectively. Each action in the action sequence is a write to a Data Vector element (e.g. set a Controller's mode to be enabled, set the engine igniter control value to true). Each action is defined by a Data Vector element (e.g. engine_igniter_control), the value to write to that element (e.g. true), and a time elapsed in the state at which the value should be written. Each transition in the set of legal transitions consists of a Data Vector element, the value to compare that element's value to, comparison type (e.g. equal to, greater than), and the state to transition to if the comparison returns true.

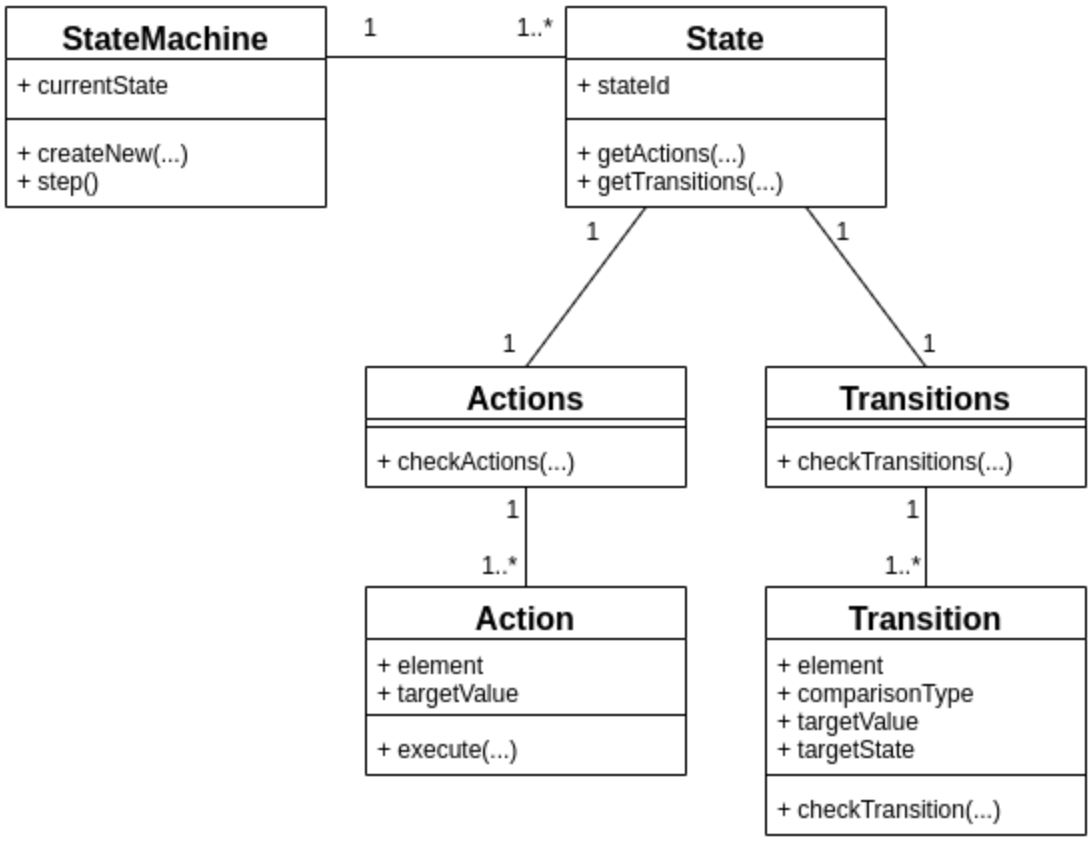

Figure 14: State Machine UML Class Diagram.

The StateMachine object is the only client-facing object in the State Machine and is responsible for initializing the entire suite of objects from a provided config and stepping the state machine forward each loop. The StateMachine's step function checks if any of the current state's transitions should occur. If yes, the StateMachine transitions to the transition's target state. The step function then checks if any of the current state's actions should be executed based on each actions' associated time to execute. If that time has elapsed in the current state, the corresponding actions are executed.

4.5.1 Configuration

On initialization the State Machine is passed a config defining the states in the State Machine. Each state's config includes the action sequence and legal transitions (Figure 15). The initial state is set to the default value set in the Data Vector's state element. The config is verified against the following rules:

- No duplicate states.
- Each action must reference a Data Vector element that exists in the configured Data Vector.
- Each transition must reference a Data Vector element that exists in the configured Data Vector.
- Each transition's target state must be a state defined by the config.

State Name	pre_launch_state		
Action Sequence	*Seconds Elapsed In State*	*Data Vector Element*	*Value To Write*
	0	TankHealthController_mode	enabled
	0	RecoveryController_mode	safed
	...	...	...
	1.3	ground_trans_telem	true
	1.5	ground_eth_telem	false
	2	pre_launch_complete	true
Transitions	*Transition Condition*	*Target State*	
	tank_health == degraded	abort_state	
	ground_command == abort	abort_state	
	pre_launch_complete == true	launch_ready_state	

Figure 15: Example State Machine Config.

4.6 CONTROLLERS

The Controller software component is implemented using an abstract class that is derived by the Avionics Software Subsystem layer (Figure 16). Every loop the flight software calls the run method on each of the Controllers, allowing them to make progress. The run method is defined by the abstract class and checks what mode the Controller is in by reading the relevant data element in the Data Vector. If the Controller is enabled, the run method calls the derived Controller's runEnabled implementation. If the Controller is safed, the run method calls the derived Controller's runSafed implementation.

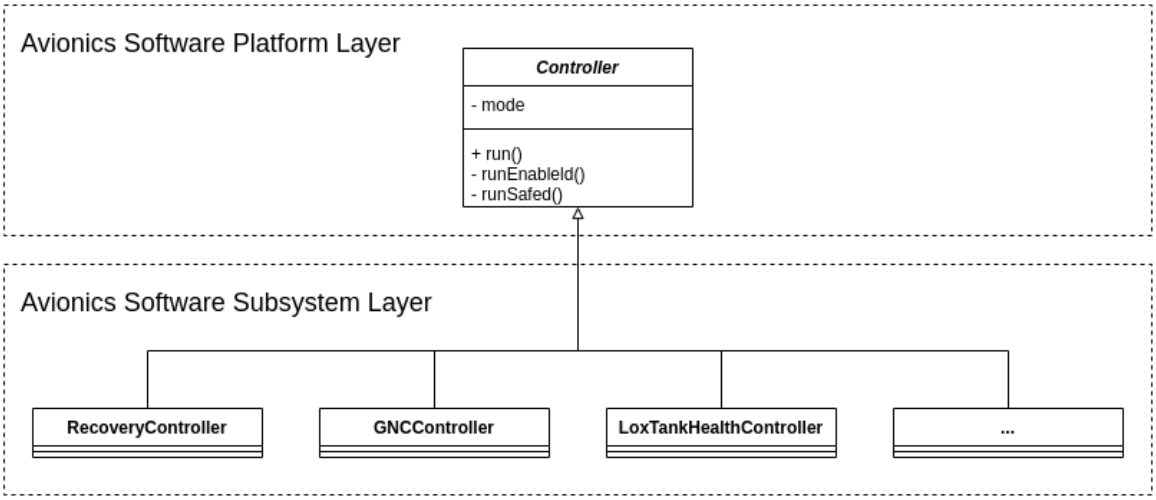

Figure 16: Controller UML Class Diagram.

Controllers accept a config on initialization, however since the configuration data is dependent on the specific Controller instance, it is the responsibility of the derived Controller to define what the config data structure looks like.

4.6.1 Setting a Controller's Mode

The established pattern for setting a Controller's mode is to do so as part of a state's action sequence in the State Machine. The most typical time to enable or safe a Controller is right after a state transition has occurred. This is done by adding an action set to execute at 0 seconds into the state to set each relevant Controller's mode. Figure 15 above showed an example of this by setting the TankHealthController_mode Data Vector element to enabled at the start of the pre_launch_state.

4.7 DEVICES

The Device software component is implemented similarly to the Controller component. An abstract class is defined as part of the Avionics Software Platform layer, and the Avionics Software Subsystem layer is responsible for implementing derived instances to interface with specific sensors and actuators (e.g. an IMU, pressure transducer, etc.) (Figure 17). Every loop the flight software calls the run method on each of the Devices. For Sensor Devices, the run method reads the sensor hardware data via the FPGA Interface, converts electrical units to engineering units, applies any necessary calibrations, and writes the results to the Data Vector. For Actuator Devices, the run method reads the relevant Data Vector data element, applies any necessary calibrations, converts engineering units to electrical units, and writes to the actuator hardware via the FPGA Interface. Devices do not have a mode. Safing an Actuator Device is done by safing the Controller that manages the Device. The safed Controller will set the appropriate data element in the Data Vector, which will then be read by the Actuator Device.

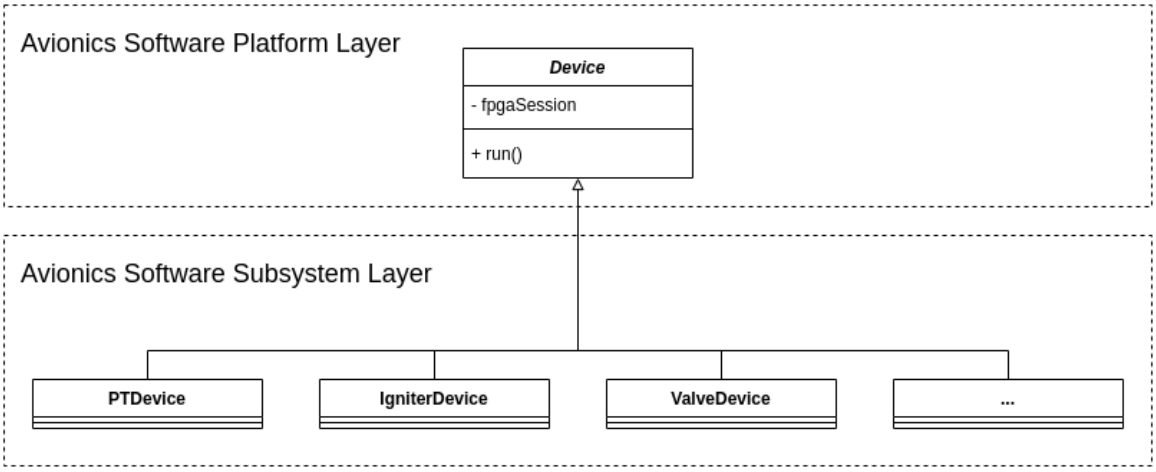

Figure 17: Device UML Class Diagram.

Devices accept a config on initialization, however since the configuration data is dependent on the specific Device instance, it is the responsibility of the derived Device to define what the config data structure looks like (e.g. Digital I/O pin number, calibration data).

4.8 COMMAND HANDLER

The Command Handler has three internal states: Idle (no operator command request received), Command Requested (operator command request received), and Command Active (operator command request active) (Figure 18). The Command Handler starts in the Idle state. To initiate a command, the Ground Node sends a command request to the Control Node. The command request includes the command, the unique command number (incremented after each request), and supporting command data if necessary. After the command request is received by the Control Node, the Command Handler moves into the Command Requested state. If it is an invalid command, the Command Handler rejects the command request and returns to the Idle state. If it is a valid command, the Command Handler processes the command and moves into the

Command Active state. The next loop the Command Handler will clear the active command and return to the Idle state.

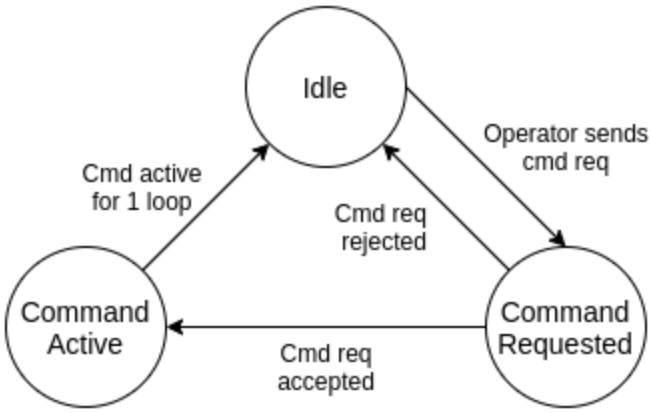

Figure 18: Command Handler Internal State Machine.

The Command Handler supports 3 commands. The first two are the LAUNCH and ABORT commands. The Command Handler processes these commands by storing the validated command in a Data Vector element that is read by the State Machine. If the current State Machine state has a transition condition triggered by the LAUNCH or ABORT command, the next time the State Machine's step method is called, the system will transition to the transition's target state. If the state does not have such a transition, no system change will occur and the command will be cleared the next loop. The third command is the WRITE command. This command requires a Data Vector element and target value as supporting data. If the element is a valid Data Vector element, the target value will be written to the Data Vector element.

Currently the Command Handler validates that the requested command is a defined command and, for the WRITE command, that the Data Vector element exists in the Data Vector. The next iteration of the Command Handler will have a state-based

command whitelist so that certain rules can be enforced (e.g. the only valid command during flight is ABORT).

4.9 CLOCK SYNCHRONIZATION

Clock Synchronization is implemented using the Linux ntpd and ndtpdate services. Although the Precision Time Protocol (PTP) is more accurate than the Network Time Protocol (NTP), PTP requires hardware that the selected flight computers do not support. Furthermore, the Avionics Software Subsystem layer only requires the flight computer clocks to be within 100ms of each other, an accuracy easily achieved using NTP.

The simplest and most efficient way to implement clock synchronization is to do it once, when the flight computers are initializing. This is possible if the clock drift between flight computers is no more than +/- 100ms over the maximum expected mission duration. However, the flight computer datasheet only claims a clock accuracy of 5 ppm [11]. With a maximum mission duration of 24 hours, this would be a maximum clock offset of 432ms assuming the clocks were perfectly synchronized at the beginning of the 24-hour period.

To determine the actual clock drift of the lab's four flight computers, an experiment was run where the computers were synchronized once and then the clock offsets were measured over the next 24 hours. The results show that the maximum clock drift between two flight computers was only 34ms over 24 hours, well under the maximum allowed offset and the sbRIO's specification (Figure 19).

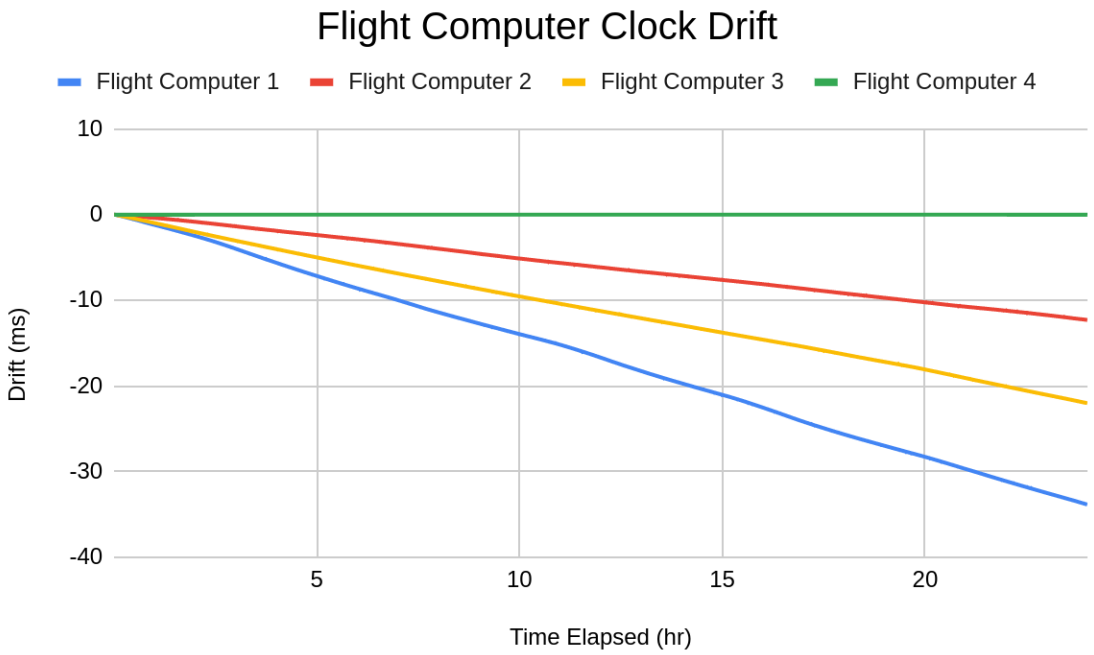

Figure 19: Flight Computer Clock Drift.

The Clock Synchronization component is implemented using a server/client model, with the Control Node acting as the NTP server and each of the Device Nodes acting as the NTP client (Figure 20). On initialization the Control Node starts the ntpd service and notifies the Device Nodes that the server is ready. The Device Nodes then run the ntpdate service to synchronize their clocks to the Control Node once. If this fails or the offset after synchronization is too high, the Device Node sends a failure response. Otherwise, the Device Node sends a success response. The server ready, success, and fail messages are sent and received using the Network Manager.

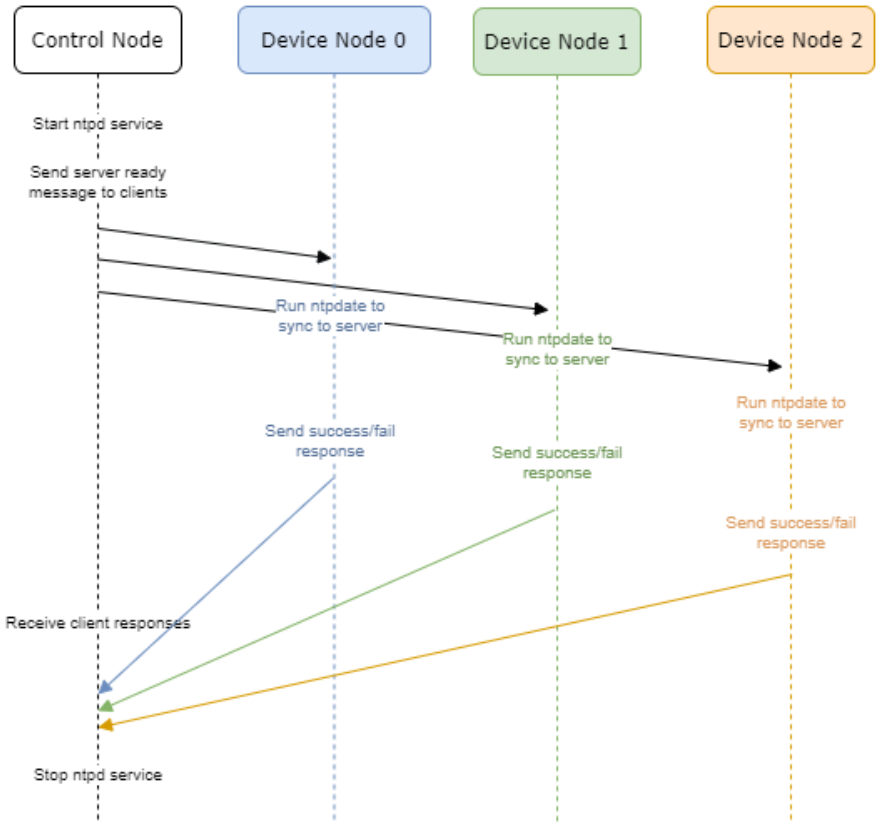

Figure 20: Clock Synchronization Protocol.

4.10 TIME MODULE

The Time Module is responsible for providing a centralized, monotonic, and linear time to the rest of the flight software. The Time Module is implemented using the clock_gettime system call with the CLOCK_REALTIME clock. CLOCK_REALTIME directs clock_gettime to use the Linux system clock, which represents time since Epoch. Since the Time Module is dependent on CLOCK_REALTIME instead of a strictly monotonic clock like CLOCK_MONOTONIC, steps must be taken to ensure the system clock does not jump backwards or forwards after the Time Module is initialized.

4.10.1 Guaranteeing Monotonicity and Linearity

The Time Module must guarantee that the time provided is monotonic (always increasing) and linear (always changing at the same rate). This means that once initialized the time cannot jump backwards or forward. To guarantee monotonicity and linearity it must be guaranteed that after initialization of the Time Module, the underlying system clock cannot be modified. The Linux system clock can be modified through the following methods:

- Hardware clock sets system clock on boot [12]
- NTP service is run
- Timezone or daylight savings change occurs
- User sets the clock via a system call (e.g. timedatectl, adjtime)

The first two methods only occur on system boot and during clock synchronization, so as long as the Time Module is initialized after the Clock Synchronization component runs, the system clock will not be affected. To prevent timezone or daylight savings changes from impacting the system clock, each flight computer's clock settings are configured to ignore both. Finally, the flight software is not permitted to use system calls that directly change the system clock, so no changes come from this method either.

4.10.2 Preventing Overflow

While the Linux system clock is not expected to overflow until 2038, a clock overflow would be catastrophic to the system. To ensure this never happens the time module returns an error on initialization if overflow will occur within the next year.

4.11 FSW Error Handling Framework

The FSW Error Handling Framework requires all flight software functions to return an error or success status, and every call-sites to check and handle the returned status. The status is represented as an enum called Error_t. All function prototypes following this pattern:

Error_t functionName (...params...);

If the function needs to return data, this is done by passing return variables by references. For example:

Error_t sum (int x, int y, int& result);

4.11.1 Handling Errors

Errors are handled by the highest-level software components on each node. These components are the Control/Device/Ground Node Main components. All other software components surface errors to the caller via the pattern described above. The Main components then handle errors in one of three ways:

1. *Exit Program:* This is done if an error occurs during system initialization. The issue is then investigated and resolved.
2. *Transition to Error State:* For critical errors that occur during flight, the Main components can set an error flag that is used by the State Machine to transition to an error state. This transition must be configured for the relevant states.
3. *Log Error:* For non-critical errors (e.g. a loop deadline was missed), an error is logged to the Data Vector.

4.12 Thread Manager

The Thread Manager is implemented as a singleton class and relies heavily on the pthreads library to create and configure threads. The Thread Manager is responsible for

initializing the CPU scheduling environment so that the flight software threads are only interrupted by critical Linux threads and responsible for managing the creation and cleanup of the flight software threads.

4.12.1 CPU Scheduling

The scheduling of threads on the flight computers needs to be tightly controlled so that the 10ms flight software loop deadlines can be met reliably. If scheduling is not tightly controlled, the flight software application will be prone to other applications taking over the CPU and delaying the flight software from making progress. To avoid this, the flight software threads are configured to use the Linux SCHED_FIFO scheduling policy, which prioritizes the flight software threads over any threads with the SCHED_OTHER policy (the most commonly used policy). There are, however, some Linux threads that use the SCHED_FIFO as well. Some of these are flight-critical threads that need to be able to take the CPU from the flight software threads (e.g. handling a timer interrupt) and others are not flight-critical, and therefore should not be able to take the CPU from the flight software. To prioritize between threads within the SCHED_FIFO policy, each thread is assigned a priority between 1 and 99, with 99 being the highest priority. On initialization the Thread Manager sets the priorities of 5 categories of SCHED_FIFO Linux threads:

1. *Hardware IRQ Threads:* These threads service the top half of hardware interrupts such as a keystroke, timer expiring, or Ethernet frame being received. Their default priority is 15, so all flight software threads must have a priority lower than this to make sure there is minimal latency in servicing these interrupts. Some of these threads do not do the full computation required to service the interrupt, but instead schedule a software IRQ thread to finish processing the interrupt.

2. *Software IRQ Threads:* These threads are sometimes scheduled by the hardware IRQ threads to finish servicing a hardware interrupt. The software IRQ threads relevant to the flight software are the ksoftirqd/N and ktimersoftirqd/N threads, where N is the core ID the thread runs on. These threads are critical for the periodic thread implementation in the Thread Manager, which relies on the hardware timer, and must not be starved. Their default priorities are 1 (timer) and 8 (soft). On initialization the Thread Manager increases both priorities to 14, just below the hardware IRQ thread priorities.
3. *Flight Software (FSW) Init Thread:* This is the thread per node that initializes all of the software components and creates the main loop thread. This thread has a priority below the software IRQ threads and above the rest of the FSW threads so that all of the other FSW threads can be initialized without the init thread being blocked. The loop thread then gets control when the init thread blocks.
4. *FSW Loop Thread:* This is the thread per node that runs the FSW loops. The priority of this thread can be set between 2 and 12 so that the hardware and software IRQ threads as well as the FSW Init Thread have no risk of starvation.
5. *RCU Threads:* These are a set of Linux threads that implements a synchronization method called read-copy update to improve concurrency in the Linux kernel. The default priorities of these threads is 1. These threads have no measurable impact on the flight software application, so their priority is set to be lower than the flight software threads.

After the Thread Manager is initialized, the following thread priorities are set:

- Linux Hardware IRQ Threads = 15
- Linux Software IRQ Threads = 14

- FSW Init Thread = 13
- FSW App Thread = 2 through 12
- Linux RCU Threads = 1

To verify that no unexpected threads were interrupting the flight software, an experiment was designed to spin the flight software indefinitely and log all threads that were able to interrupt the application and run. The results of this experiment surfaced 2 unexpected National Instruments threads that were interrupting the flight software application. The first was for a feature unused by the flight software, so this feature was uninstalled. The second was a watchdog thread for a National Instruments program that is used occasionally to configure the flight computers. Since the watchdog only runs for a handful of microseconds per second, the program was left as is.

4.12.2 Managing Flight Software Threads

The Thread Manager provides two methods for creating threads. The first, createThread, creates a one-off thread. The caller provides a function for that thread to execute, a priority to set the thread at (must be between 2 and 12), and a CPU affinity to set (core 0, core 1, or both). The second method, createPeriodicThread, creates a thread that runs with a specified period. The caller provides the function to be executed, the period at which to execute the function, a priority to set the thread to, a CPU affinity, and a callback function to be called if the function misses its period deadline.

A periodic thread is implemented internally by using a wrapper function and a timer (using the timerfd API). The wrapper function sets the time, calls the caller-provided function, and then checks to see if the deadline was missed when the caller-

provided function returns. If the deadline was missed, the caller-provided callback function is run to handle the missed deadline.

4.13 CONTROL NODE MAIN

The Control Node Main component is the configurable entry point for the flight software running on the Control Node. This component defines an entry() and loop() function. The entry() function initializes all flight software components and creates a 100Hz periodic thread that executes the loop() function.

4.13.1 Entry Function

The entry function initializes the Data Vector, Thread Manager, Network Manager, Time Module, and State Machine software components using passed in configs. Because derived Controllers have differently-typed configs, the caller must provide a Controller initialization function that the entry function calls. This is to avoid hardcoding the Controllers that are initialized by Control Node Main. If any software components return an error on initialization, the error is printed and the program exits. The final step in the entry function is to create the 100Hz periodic thread that executes the loop function. The periodic thread is fixed to CPU 1 to take advantage of the efficiency gains shown in Figure 13. The entry function then waits on the periodic thread, which causes it to block. The periodic thread is expected to never return, but if it does, the entry function will print the returned error and exit the program.

4.13.2 Loop Function

The loop function runs the Data Synchronization Protocol, reads the current time from the Time Module and stores it in the Data Vector, runs the Command Handler, steps the State Machine, and then runs each Controller. The Data Synchronization Protocol is

given 2ms to complete. If the loop function misses this deadline, a data synchronization deadline miss error is logged to the Data Vector. If a Region is not received from one or more of the Device Nodes, a missed message error is logged.

The time stored at the top of the loop is used by the State Machine to execute the action sequence and used by the Controllers to run any time-based logic. Using a single timestamp for the entire loop makes each loop more deterministic, since the State Machine and Controller results will not depend on small variations in when exactly in the loop they run. Additionally, establishing this pattern makes it easier to debug the system using just the Data Vector telemetry logs. The Command Handler runs next so that a new operator command request can be processed before the State Machine runs, which can be affected by operator commands. The Controllers run last, as they are dependent on the State Machine. The most common dependency is the State Machine setting a Controller's enabled/safed mode. If any of these software component return an error, a generic error is logged to the Data Vector. If the loop function misses its 10ms deadline, a loop deadline error is logged.

4.14 DEVICE NODE MAIN

The Device Node Main component is the configurable entry point for the flight software running on each Device Node. Similar to Control Node Main, this component defines an entry() and loop() function. The entry() function initializes all flight software components and creates a thread that executes the loop() function.

4.14.1 Entry Function

The entry function initializes the Data Vector, Thread Manager, Network Manager, Time Module, and FPGA Interface software components using passed in

configs. Because derived Controllers and Devices have differently-typed configs, the caller must provide a Controller and Device initialization function. This again avoids hardcoding which Controllers and Devices are initialized in the entry function. If any software components return an error on initialization during the entry function, the error is printed and the program exits. The final step in the entry function is to create a one-off thread that executes the loop function. The thread is fixed to CPU 1 to take advantage of the efficiency gains shown in Figure 13. The entry function then waits on the thread, which causes it to block. The thread is expected to never return, but in case it does, entry will print the returned error and exit the program.

4.14.2 Loop Function

The loop function runs an infinite while loop that is synchronized to the Control Node's loop by the Data Synchronization Protocol. During the Data Synchronization Protocol, a Data Vector Region is received from the Control Node. The Device Node sends a Region in response and begins executing the rest of its loop. The next step in the loop is to run all Sensor Devices, Controllers, and then Actuator Devices. Sensor Devices are run before the Controllers so that low-level Controllers running on the Device Node have the most up-to-date sensor data. The Actuator Devices are run after the Controllers so that the actuators are set to the most up-to-date control values. If any of these software components returns an error, a generic error is logged to the Data Vector.

4.15 Ground Node Main

The Ground Node Main software component is responsible for initializing the software components that run on the ground computer and executing an infinite while loop. The loop receives telemetry from and sends operator commands to the Control

Node. A unique attribute of the Ground Node is that it has two instances of the Data Vector. The first instance is a copy of the Control Node's Data Vector, which is used to store each copy received from the Control Node as telemetry. The second Data Vector instance is used to store operator command requests and Network Manager stats.

One of the sub-teams on the Avionics Software team is the Mission Control System team. This team is responsible for developing the ground software running on the Ground Node that will display telemetry to ground operators and allow operators to send commands to the Control Node via a GUI. Future versions of Ground Node Main will integrate with this system.

Chapter 5: 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.

5.1 TESTING

5.1.1 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 [13]. 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.

5.1.2 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.

5.1.3 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.

5.1.3.1 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 21). 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.

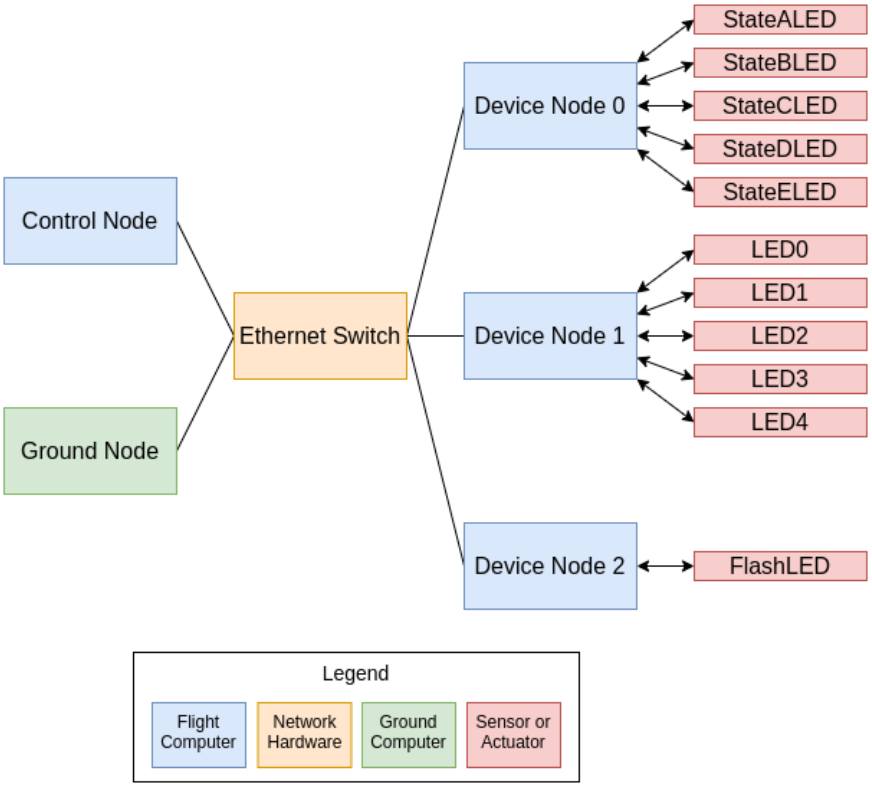

Figure 21: LED Platform System Test Setup.

The state machine for the LED test is shown in Figure 22. 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 23-27 show pictures of the system as it moves through the system's five states.

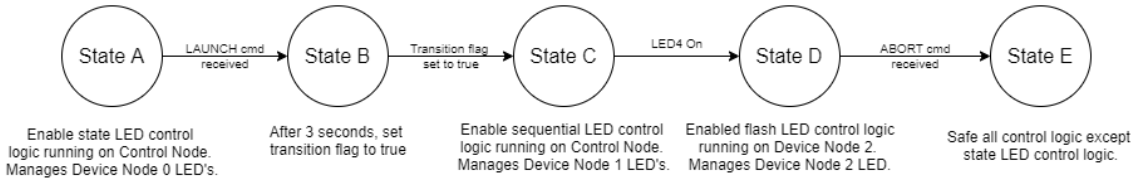

Figure 22: LED Platform System Test State Machine.

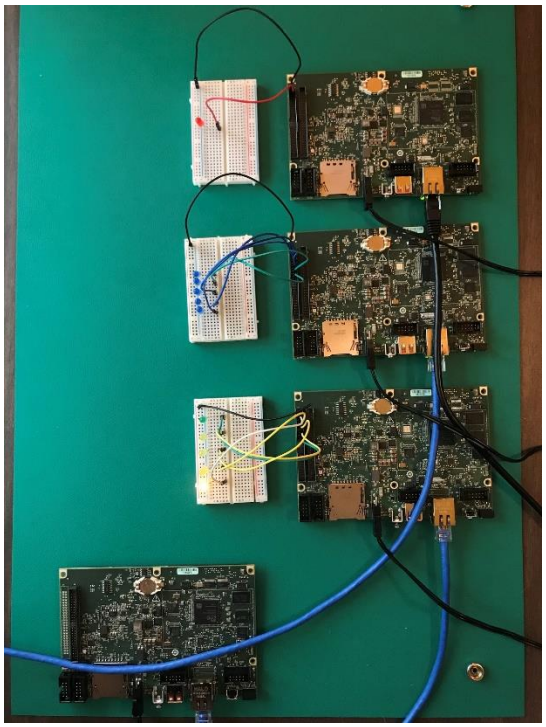

Figure 23: LED Platform System Test State A.

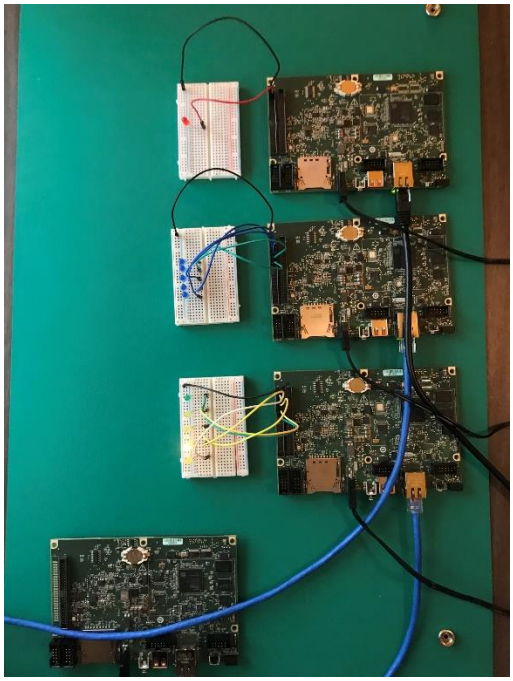

Figure 24: LED Platform System Test State B.

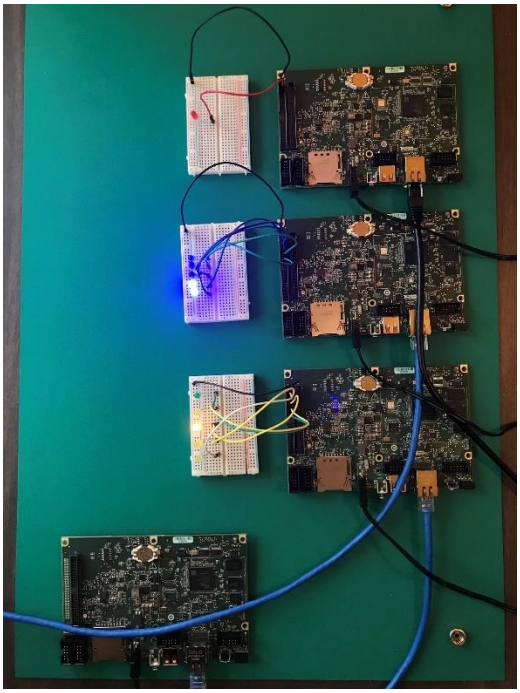

Figure 25: LED Platform System Test State C.

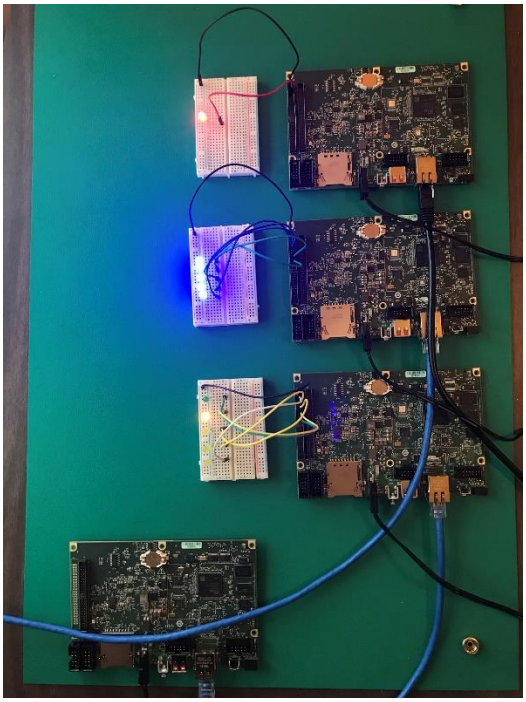

Figure 26: LED Platform System Test State D.

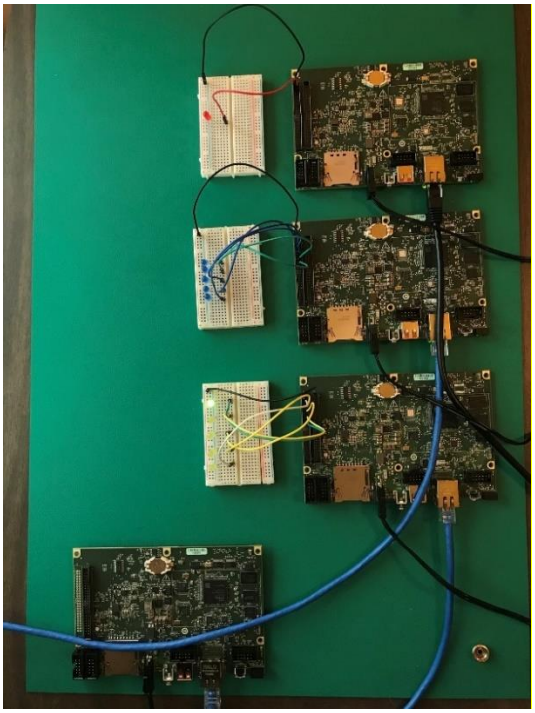

Figure 27: LED Platform System Test State E.

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.

5.1.3.2 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 28 shows the flight computer setup, Figure 29 shows the igniter in black powder, and Figure 30 shows the successful black powder ignition.

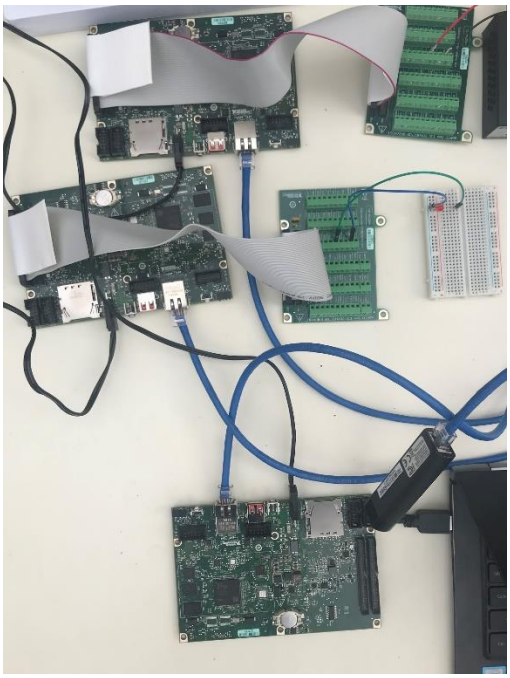

Figure 28: Recovery Igniter System Test Flight Computer Setup.

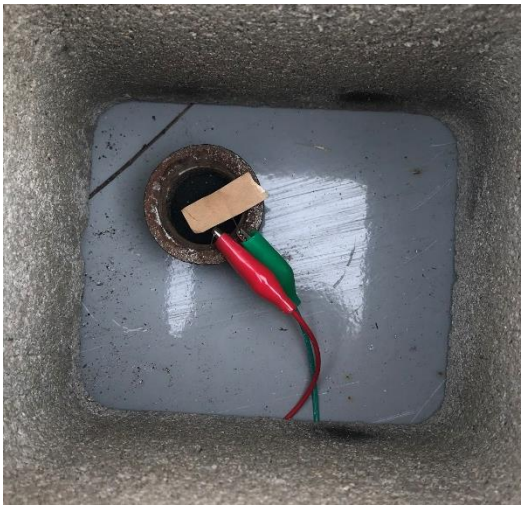

Figure 29: Recovery Igniter System Test Igniter Setup.

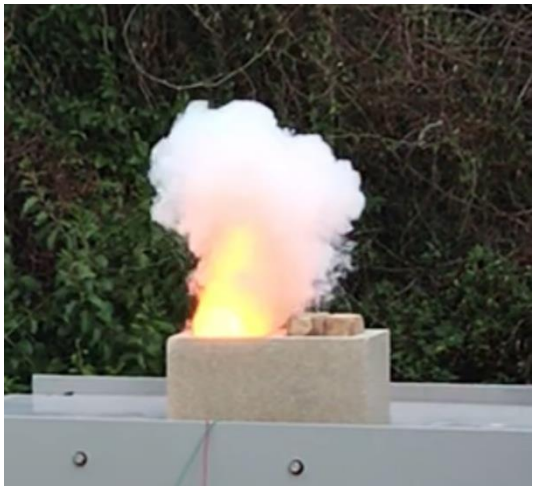

Figure 30: Recovery Igniter System Test Ignition Success.

5.2 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.

5.2.1 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.

5.2.2 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 31 for the control logic running on the Control Node and Figure 32 for the control logic running on the Device Nodes.

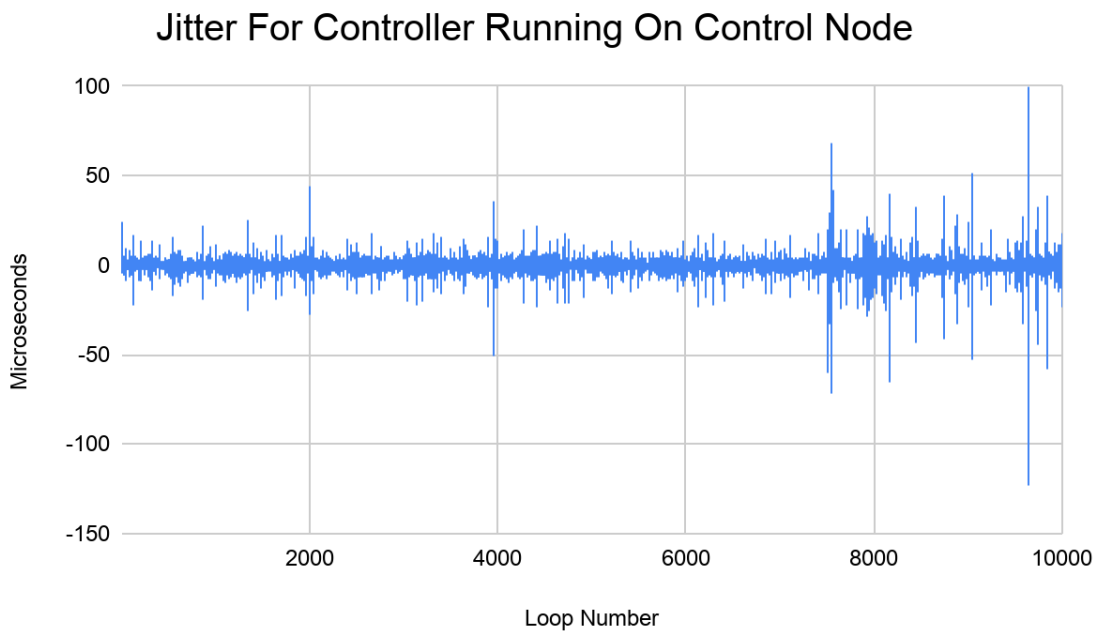

Figure 31: Control Node Jitter.

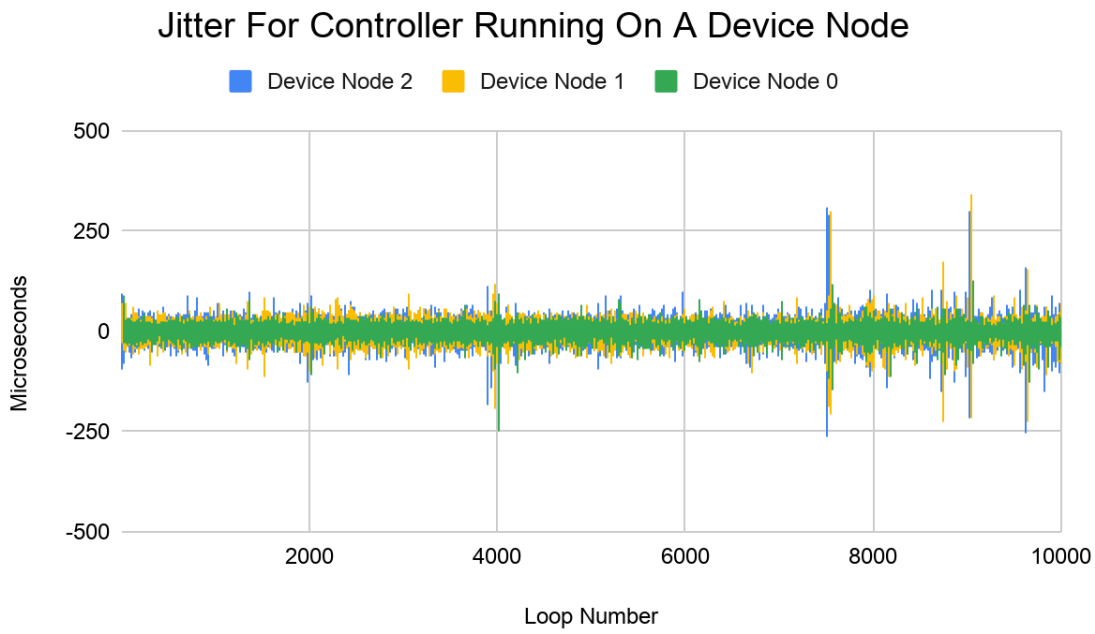

Figure 32: Device Node Jitter.

The timeseries data and aggregated results (Table 3) 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 3: 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.

5.2.3 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 4).

	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 4: Platform Reaction Time Results.

5.2.4 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.

5.2.5 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 5).

	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 5: 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.

5.3 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 6. 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 6: 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 6: Platform Requirements Verification Status.

Chapter 6: 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.

6.1 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.

6.2 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.

6.3 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.

6.4 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 7: 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] Lawrence Berkeley National Laboratory. “Manpage of TCPDUMP.” TCPDUMP, 2 Mar. 2020, www.tcpdump.org/manpages/tcpdump.1.html. (accessed 4 Apr. 2020)

[10] Xilinx. Zync-700 SoC Technical Reference Manual. 1 July 2018, www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf. (accessed 4 Apr. 2020)

[11] National Instruments. NI sbRIO-9637 Specifications. 15 Jan. 2018, http://www.ni.com/pdf/manuals/375279b.pdf (accessed 4 Apr. 2020)

[12] Arch Linux. “System Time.” System Time, ArchWiki, 4 Feb. 2020, wiki.archlinux.org/index.php/System_time. (accessed 4 Apr. 2020)

[13] 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)