2021

Increasing the Reliability of Software Systems on Small Satellites Using Software-Based Simulation of the Embedded System

Matthew D. Grubb
mgrubb1@mix.wvu.edu

Increasing the Reliability of Software Systems on Small Satellites Using Software-Based Simulation of the Embedded System

Matthew David Grubb

Thesis submitted to the
Benjamin M. Statler College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

Powsiri Klinkhachorn, Ph.D., LCSEE, Committee Chair
Roy Nutter, Ph.D., LCSEE
Jason Gross, Ph.D., MAE

Lane Department of Computer Science and Electrical Engineering
Morgantown, WV
2021

Keywords: Software-only Simulation, Satellite, SmallSat, CubeSat, Flight Software, cFS, NOS^3

Abstract

Increasing the Reliability of Software Systems on Small Satellites Using Software-Based Simulation of the Embedded System

Matthew D. Grubb

The utility of Small Satellites (SmallSats) for technology demonstrations and scientific research has been proven over the past few decades by governments, universities, and private companies. While the research and technology demonstration objectives that can be provided by these SmallSats are becoming similar to larger spacecraft, their reliability still falls behind. This is in part due to the reduced cost of SmallSat missions in comparison to large spacecraft, which requires cheaper components, rapid development schedules, and accepted risk. In these missions, the importance of the flight software is often overlooked, and the software is rushed through development and not fully tested to provide the reliability required for on-orbit operations.

This research aims to investigate the common causes of failures on SmallSats, and to provide a solution to the problem of developing and testing reliable flight software, through the use of software-based simulation of the full embedded satellite system. During this research, an open-source product was developed and released to the public to assist SmallSat missions, which is currently in use by public and private institutions across the country. The resulting product, the NASA Operational Simulator for Small Satellites, commonly referred to as NOS^3, will be discussed in detail. The results of NOS^3 will be viewed through a case study of the application of NOS^3 to a SmallSat mission.

Acknowledgments

I would like to start by thanking my graduate advisor, committee chair, and long-time mentor Dr. Powsiri Klinkhachorn. Dr. Klink encouraged me to complete the research contained within this paper and provided me with a great amount of knowledge throughout his courses.

I would also like to thank Dr. Roy Nutter and Dr. Jason Gross for taking the time to sit on my graduate committee and advise in this research.

Thank you also to Dr. Matthew Valenti for all the knowledge I learned on communications systems from his courses, and all the professors I had along the way at WVU.

A major thank you must be provided to the entire STF-1 team, too many names to list here, for their critical assistance in the design, development, and operations of West Virginia's first spacecraft, the results of which were critical for this research.

A special thanks to the managers of the NASA JSTAR team Justin Morris and Scott Zemerick for allowing me to work on STF-1 and NOS^3. Thanks to the developers who had input into this research including John Lucas, Mark Suder, Dustin Geletko, Steve Yokum, Tim Reilly, Jason Battin, Justin McCarty, Robert Brown, Cody Cutright, as well as Justin and Scott.

I must also thank the developers of the open-source tools such as cFS, 42, and COSMOS that are critical components in this research for taking the time to release useful code to the public.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction

1.1 Background

The National Aeronautics and Space Administration (NASA) classifies Small Satellites, commonly referred to as SmallSats, as any spacecraft with a mass of 180kg or less [1]. The size, weight, power, and cost (SWaP-C) of these spacecraft vary drastically within the category, from femtosatellites as small as .001kg to minisatellites from 100kg to 180kg. Thanks to the standardization effort between Prof. Jordi Puig-Suari at California Polytechnic State University, San Luis Obispo, and Prof. Bob Twiggs at Stanford University's Space Systems Development Laboratory in 1999, the majority of SmallSats launched in the past decade have been CubeSats [2]. Puig-Suari and Twiggs realized that in order to increase access to launch opportunities, the form factor of the spacecraft needed to follow a strict standard, which would allow a common deployment mechanism to be used on numerous different launch vehicles. The original CubeSat Design Specification defined the spacecraft as a 10x10x10cm body, in addition to specific contacting points for the Poly Picosatellite Orbital Deployer (P-POD) [2]. Three of these CubeSats would fit into the P-POD, and multiple P-PODs could be affixed to a launch vehicle as a secondary payload [3]. As a secondary payload, the CubeSats take advantage of "ridesharing" where there is unused payload capacity in the launch vehicle, and the secondary payloads are determined to pose no risk to the primary mission. Launches were then able to deploy numerous CubeSats at one time, greatly increasing the number of missions able to access space.

The CubeSat standard has continued to evolve, and the 10cm cube is no longer the only size recognized. The original standard is now referred to as a 1 Unit (1U) CubeSat, but developers can select a size that fits their mission requirements from 1, 1.5, 2, 3, 6, and 12U. While 1-3U

CubeSats are still the most common due to launch availability, 6 and 12U deployment canisters are available. With the expansion of the CubeSat Design Specification, and a growing number of rideshare, and dedicated launch opportunities, the number of CubeSats launched has grown rapidly since 1999 [4].

These use of CubeSats vary greatly from mission to mission, depending on the mission's objectives. For small missions, such as the NASA Ames PhoneSat [5], the total hardware cost for a 1U CubeSat was limited to $3500. The purpose of the PhoneSat mission was to use a commercial cellular phone as the processing system on a CubeSat. This is referred to as a technology demonstration mission, where the objective is to prove the functionality of some technology, in this case, a Nexus cellular phone, when operating on-orbit. Technology demonstrations are a very common use of a CubeSat, however, it is not the only one.

A much more complex mission, such as the NASA JPL MarCO pair of CubeSats, cost $18.5 million [6]. The MarCO CubeSats were the first interplanetary CubeSats to be launched, with the objective of relaying data from the Mars lander InSight back to Earth during InSight's descent to the planet. The complexity of MarCO required more money than PhoneSat due to the size, weight and power restrictions imposed by a much more difficult to achieve objective. Although $3500 to $18.5 million seems like a major leap in cost, the MarCO mission was still just a fraction of a typical spacecraft's cost. As an example, the NASA Mars Reconnaissance Orbiter mission cost $716.6 million, $90 million of which was for the launch alone [7]. With cost being an important factor in mission planning and development, more science-based missions are looking to CubeSats and SmallSats to accomplish their objectives. Launch opportunities are also available for qualified missions via the NASA CubeSat Launch Initiative (CSLI) at no cost to the missions [8].

1.2 Problem Statement

With a continuous increase in the number of SmallSats being launched, there remains a large number of these spacecraft that fail to meet their mission objectives [4]. The development cost and mission timelines of these spacecraft are a fraction of the typical flagship spacecraft, and it is typical for the mission to accept a certain amount of risk [9]. One of the areas commonly overlooked for SmallSats is the flight software (FSW). Flight software is a critical component in a successful mission, but it is routinely scheduled late, during spacecraft integration, in an already short development cycle [10]. This leads to limited testing of the FSW, which can hide the presence of errors and lead to workarounds on orbit, or at worst mission failure.

Large missions, with longer development time and a much larger budget, utilize simulations to increase the amount of testing that can be performed and increase their FSW reliability. With more time and money, the FSW can be tested to near full coverage, and the resulting product expects to operate without a catastrophic failure when in orbit. However, SmallSats accept many risks, FSW only being one of which, and are known to fail. The time spent in development and the overall cost are low enough, that another attempt can, and commonly will occur [4].

The subject of this research is, can the implementation of simulations utilized during development and testing for large missions be tailored to fit the needs of SmallSats thus increasing the reliability of the resulting FSW?

1.3 Research Objectives

This research will: 1.) Evaluate the root causes that lead to lower cost and rapidly developed FSW in SmallSat missions, such as short mission schedules, lack of hardware for flight software testing, and the complexity required for successful FSW. 2.) Evaluate potential solutions that already exist for simulation, such as commercial products like Wind River SIMICS, and open-source software such as NASA's 42 Spacecraft Simulation. 3.) Provide an open-source solution that allows missions to increase the reliability of their FSW.

The resulting solution, a new suite of software that integrates multiple other open-source tools, named the NASA Operational Simulator for Small Satellites, or NOS^3, has been released to the public and is in use by government agencies, universities, and private companies at the time of this writing. NOS^3 combines multiple proven open-source software packages, such as NASA's 42 and NASA's core Flight System (cFS), with newly a developed simulation framework for modeling SmallSat hardware in a software-only environment.

1.4 Organization

This section of Chapter 1 concludes the introduction of the research. Chapter 2 contains the literature review that will provide more details on SmallSat requirements, determine the success and failure rates of these SmallSats, and investigate potential solutions that may solve the problem of FSW reliability. Chapter 3 details the design of the developed solution, NOS^3, starting with the requirements necessary to solve the problem, the initial proof-of-concept for the solution, then explaining the final solution and sub-components, and describing how it can be deployed and utilized. Chapter 4 will provide the results of the solution through a case study of an actual

SmallSat mission, testing performed to verify the solution, known limitations, and applications. Chapter 5 concludes the research with a summary and describes how the NOS^3 can be enhanced through future work.

Chapter 2. Literature Review

2.1 CubeSat Design Specification and CubeSat Launch Initiative

While the CubeSat Design Specification (CDS) does not mention FSW explicitly, it does contain critical information that explains why FSW is an important component of the spacecraft. It also provides the spacecraft requirements that can lead to design decisions that affect the FSW development and design. The 13th edition of the CDS was released in February of 2014, and a new release is currently in the draft stage [2].

The CDS provides a detailed specification of the dimensions of the spacecraft, the interfaces required on the spacecraft for the P-POD deployer, and a list of requirements "*to ensure the safety of the CubeSat and protect the launch vehicle (LV), primary payload, and other CubeSats*" [2]. SmallSats are typically launched as a secondary payload, meaning the SmallSat is not a requirement of either the primary payload or the launch vehicle itself. The secondary payloads take up unused payload capacity and space on the launch vehicle that has already been funded by the primary mission [3]. To make the interface between the secondary payload and the launch vehicle easier, the CDS contains a standard for the dimensions of their P-POD deployer.

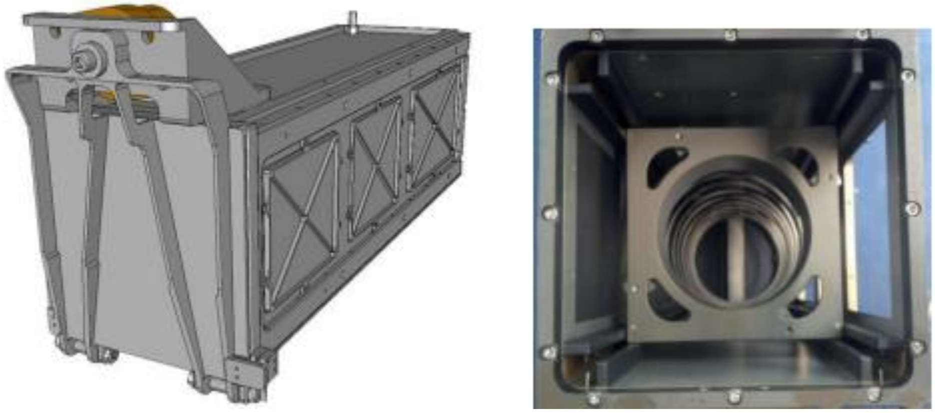


Figure 1 - CalPoly P-POD Deployer [2]

There are an increasing number of dedicated launch vehicles for SmallSats, but these come with a cost that must be paid by the mission. The cost of a dedicated launch from the small launch provider Rocket Lab USA costs close to 5 million USD [11]. The Rocket Lab Electron dedicated launch vehicle, used in the NASA Educational Launch of Nanosatellites (ELaNa-19) mission, deployed 13 CubeSats to low Earth orbit (LEO) [12]. NASA CubeSat Launch Initiative provides zero-cost launch services for a select number of SmallSats each year. For the reason of cost alone, most US-based missions elect to launch via the CSLI. The CSLI launch services depend on the launch vehicle that is providing the rideshare, but one consistent requirement is to meet, and in certain cases exceed, the requirements outlined in the CDS [13]. As of May 2020, NASA CSLI has selected and prioritized 220 CubeSat missions from 102 unique organizations representing 41 states and the District of Columbia, including one from West Virginia [8].

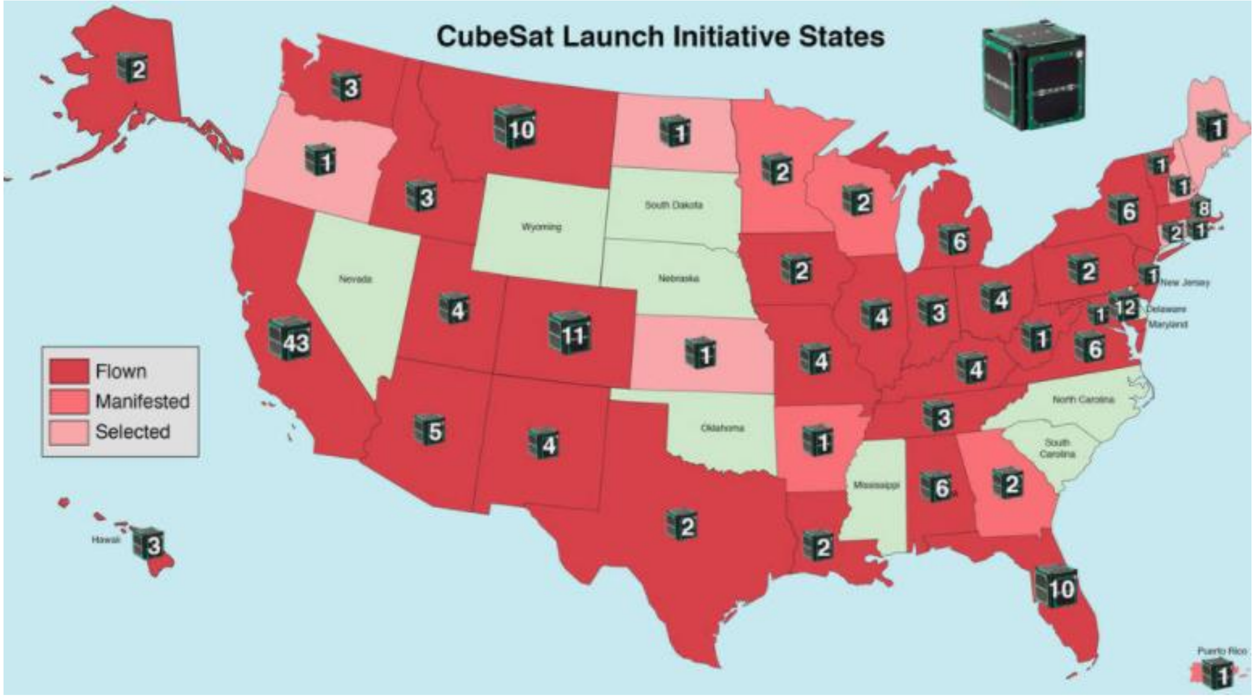


Figure 2 - NASA CSLI Launches by State [8]

The CDS definitions for the size and mass of 1-3U CubeSats are shown in the following table.

Table 1 - CDS Definitions of CubeSat Unit Size

U Definition	Width (mm)	Length (mm)	Height (mm)	Mass (kg)
1U	100.0	100.0	113.5	1.33
1.5U	100.0	100.0	170.2	2.00
2U	100.0	100.0	227.0	2.66
3U	100.0	100.0	340.5	4.00

The dimensions and mass requirements above are limiting in terms of the size of the components that need to fit within the spacecraft. A typical spacecraft will need to have a command and data handling (C&DH) subsystem, an electrical power subsystem (EPS), a communications subsystem, an instrument or technology demonstration, and optionally an attitude determination and control subsystem (ADCS). The following figure shows the dimensions of a 1U CubeSat that would need to contain these subsystems.

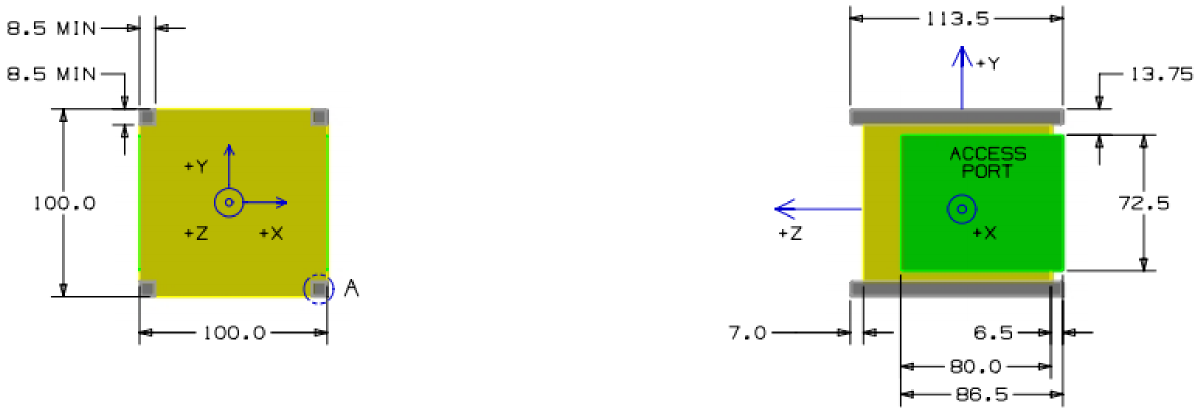


Figure 3 - 1U CubeSat Dimensions [2]

Fitting these subsystems into the small form factor poses a mechanical challenge, but it also poses a technology challenge. Smaller components need to be selected than are typical on large missions, and it is common for commercial components to be repurposed for SmallSats [14]. These commercial parts may have flight heritage from another mission using them in the past, but that does not reflect on the reliability for all missions. For this reason, the FSW must be robust enough to detect and correct issues that occur due to component failures. For example, if the communications and C&DH cannot pass commands and telemetry due to a failure in the microchips that control the data bus, then the mission will be over. It is up to the FSW to attempt to recover these subsystems and preserve functionality.

The small form factor also routinely requires deployable components that will expand when the orbit is achieved. This poses two challenges to FSW, the first being that there are strict deployment requirements in the CDS that must be adhered to and verified via testing. The CDS specifies "*All deployables such as booms, antennas, and solar panels shall wait to deploy a minimum of 30 minutes after the CubeSat's deployment switch(es) are activated from PPOD ejection*" [2]. The second is that these deployments in most cases must occur for the mission to succeed. For example, if the antenna for the communications system is stowed inside or the solar arrays are stowed on the body of the spacecraft, then the spacecraft could lose either communications or power if a deployment is not achieved. This adds FSW complexity to meet the CDS requirements for the deployments, and to ensure that the deployments occur in time to meet the mission requirements.

The CSLI provides a useful document to guide first-time missions through the process of CubeSat development. The CubeSat 101: Basic Concepts and Processes for First-Time Developers, gives a new team information on the development process from Concept Development to Mission Operations. One of the best things this document does is give a mission a timeline to follow while highlighting requirements along the way. "*A CubeSat can be designed, built, tested, and delivered in as little as 9 months, but typically takes 18 to 24 months to complete*" [9]. The following chart provided by CubeSat 101 shows this rapid development schedule.

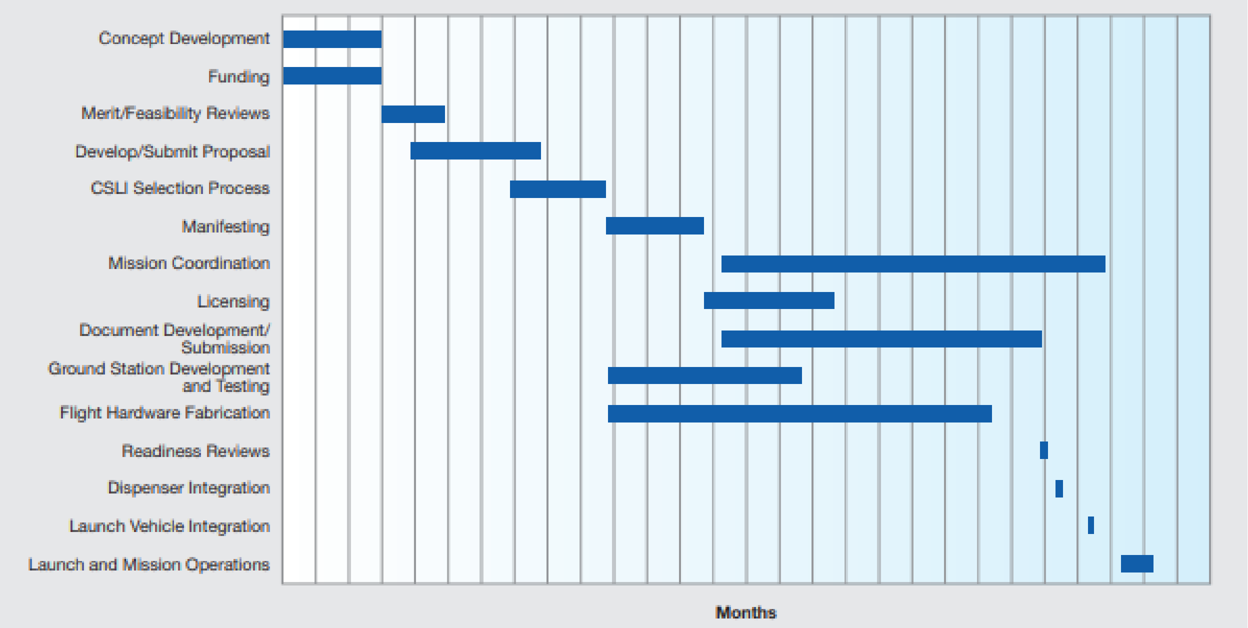


Figure 4 - CubeSat 101 Mission Development Schedule [9]

Out of the entire document, the FSW for the spacecraft is only mentioned in two sections, and these references are surprisingly minimal. The first reference is when describing the use of Engineering Test Units (ETUs) of hardware. CubeSat 101 states "*[An ETU] can be used to practice putting the components together, fit checks, hardware and software testing, and anything else that you don't want to try for the first time on your valuable flight unit*" [9]. This is a valuable statement for missions to remember for testing their spacecraft, in that ETUs are a very valuable resource for software testing. These ETUs however come with a price tag and most missions will have to determine where the budget can afford an ETU, and where the flight unit will have to suffice. **Can**

simulation of the spacecraft hardware act as another alternative to flight software testing on ETUs where they are either capitalized in integration testing or unavailable due to budget constraints?

The second mention of FSW comes in the section describing the required “Day-in-the-Life” (DITL) testing. *“This test shows that your CubeSat’s electronics and flight software work as expected. The ICD will have requirements for when the CubeSat is allowed to release its deployables and when it can start transmitting after being ejected from the dispenser”*. *“This test must be run with the final flight software and in most cases will be required to be completed prior to environmental testing”* [9]. As quoted above, the final FSW must be proven through a DITL test to meet the requirements for launch, and this should take place before environmental testing. This implies from the schedule provided in the guide, that the final FSW will need to be developed in 2-12 months during the “CubeSat hardware fabrication and testing” segment of the development cycle. A 12-month period for a first-time mission to develop working FSW is very short and could explain why many missions have inoperable CubeSats when deployed on orbit. CubeSat 101 gives missions no other guidance in the area of flight software, and missions using this document to get them to launch may find FSW is a much more difficult component of their spacecraft than is alluded to in this guide.

2.2 Reliability and Failures

“*Despite the reduced size of these spacecrafts, their Flight Software (FSW) complexity is not proportional to the satellite volume, thus creating a great barrier for the entrance of new* *players on the nanosatellite market*” [10]. Even though the SWaP-C of SmallSats is a fraction of that of their flagship counterparts, the flight software complexity does not scale accordingly.

Over many years, Dr. Swartwout of St. Louis University has maintained an online database to document all CubeSats launched, as well as the status of each of the missions [4]. The database is openly available to the public, and mission status is provided either from publicly available information, or a contribution from an individual mission. The mission status of all CubeSats launched from 2000 to 2019, with the exclusion of constellations, is shown in the following chart.

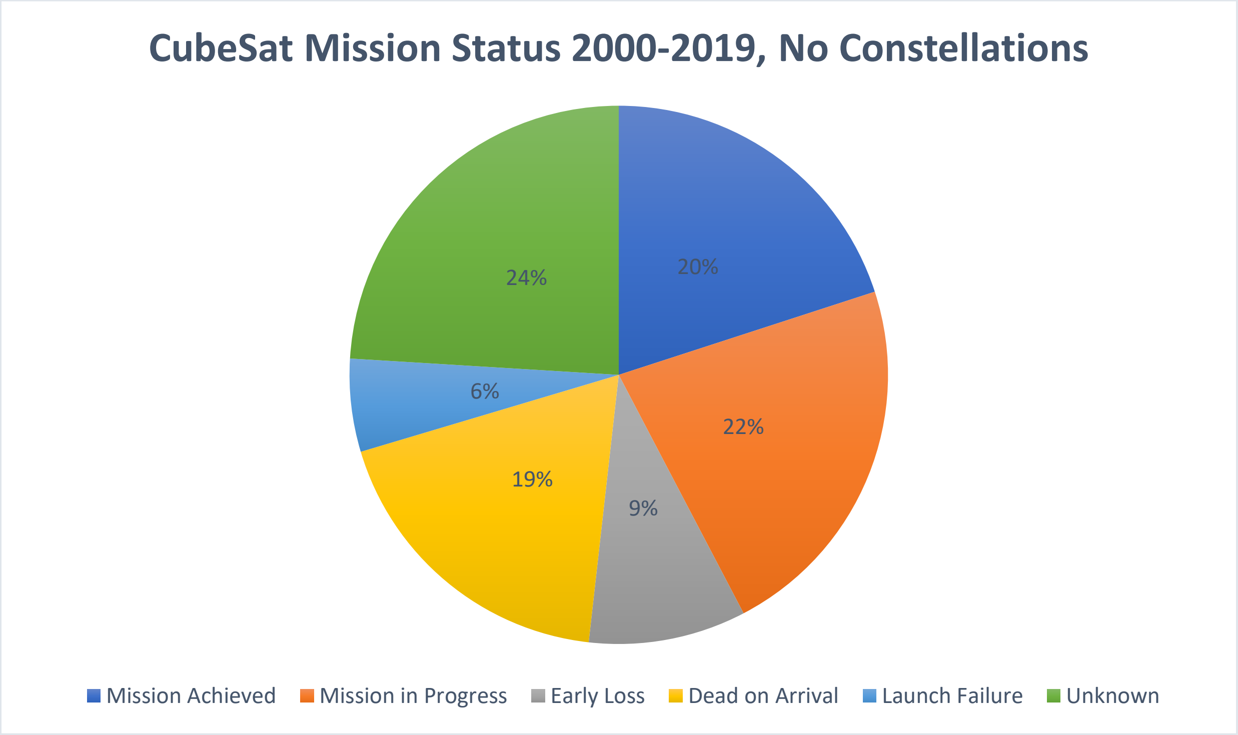


Figure 5 - CubeSat Mission Status 2000-2019 [4]

Constellations are excluded from this data because “*With Planet and Spire contributing more than 500 CubeSats between them, their missions dominate any chart that I could produce”* [4]. Constellations are typically a suite of identical spacecraft that can be mass-produced, and as such their development process differs drastically from that of a university or first-time spacecraft builder. For this research, constellation spacecraft will also be excluded.

The most glaring data from the chart is the percentage of CubeSats that are either dead-on-arrival (DOA) or an early loss. Combining for over 27 percent of all missions are CubeSats that have been reported as failing to complete their mission. There is the possibility that that number is even higher given the over 23 percent of missions in which their status is currently unknown. In an investigation by Dr. Swartwout of "*University-class satellites – that is, spacecraft built by university students for the express purpose of student training*" the failure rate is even larger as shown in the following chart [15].

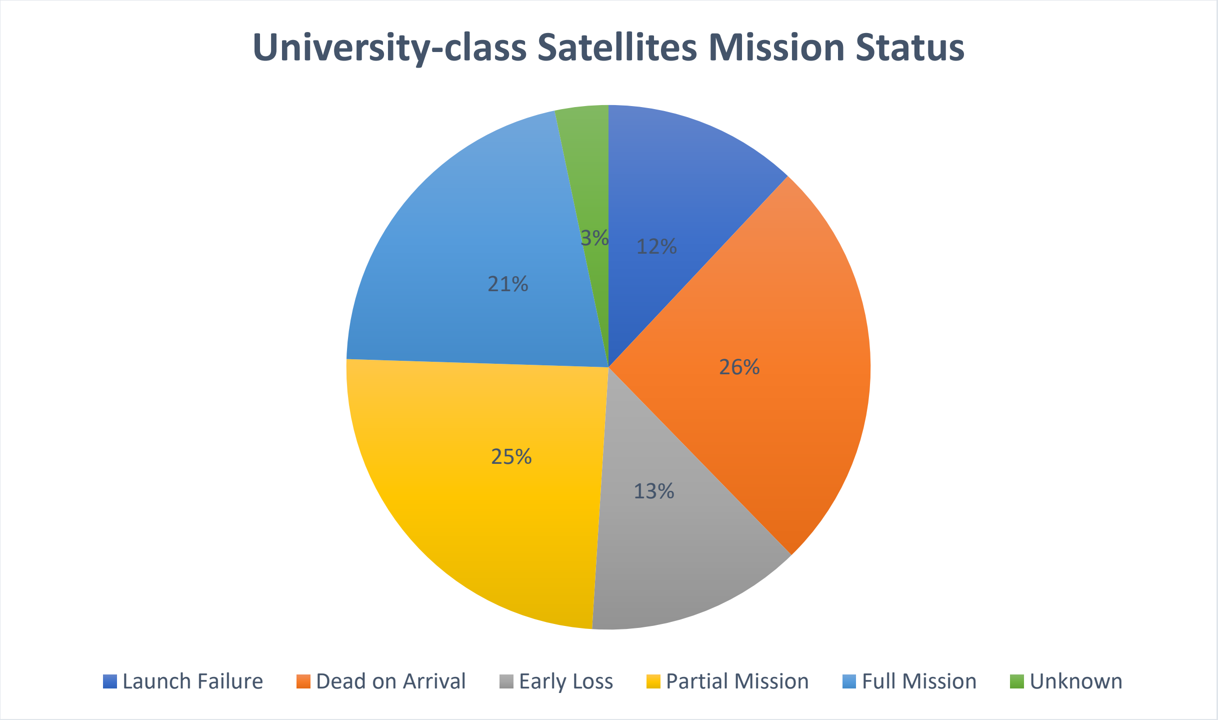


Figure 6 – University-class Satellites Mission Status [15]

It would be impossible to determine the root cause of all of these failures because a spacecraft that is DOA could have had any number of failures, however, the author does state that *"in [his] twenty years of experience with university-class missions, he has noted that student-led projects often fail because of a lack of time/resources given to systems-level testing"* [15]. This is an important point that reflects on the rapid development and testing schedule that these missions typically follow. **Can simulation of the spacecraft hardware allow more time and resources for systems-level testing?**

In another investigation into the failure rates of CubeSats titled "Reliability of CubeSats – Statistical Data, Developers' Beliefs and the Way Forward", the authors solicited feedback from developers of CubeSat missions that had failed. "*Of the surveys sent out to 987 individuals, 113 were returned fully completed*" and "*73% of the participants considered themselves not as a beginner or as without knowledge in risk and failure analysis*" [16]. From these solicitations, "*the experts also had to subjectively assess what reason might have caused the assumed critical failure of the satellite*" [16]. This is perhaps the best data point that was found concerning software being a contributing factor to failures, and not surprisingly, "software design failure" was the highest percentage of assumed failure with "fault in electronics" being a close second. The following chart shows the responses for a percentage likelihood of the critical failure with "Software Design Error" being roughly 34% chance of likelihood.

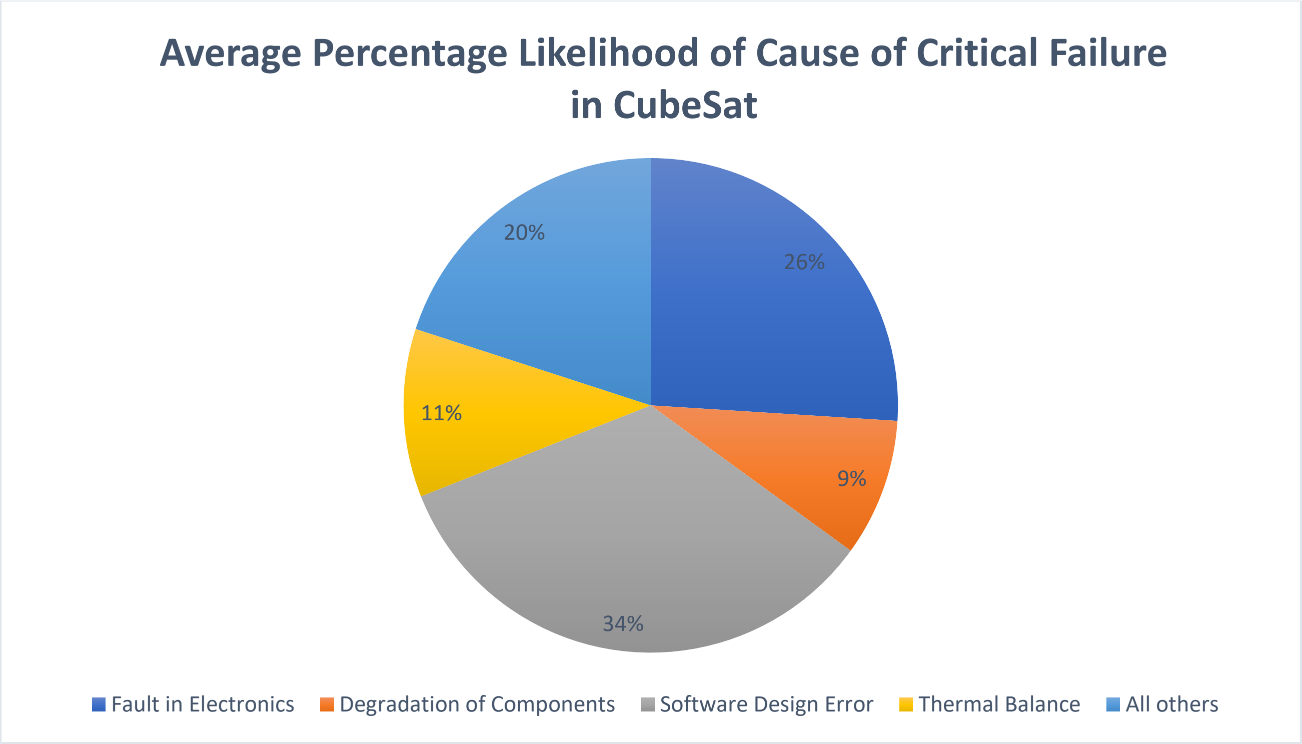


Figure 7 – Percentage Chance of Software Causing Critical Failure [16]

From this investigation, it is apparent that many missions did not have confidence that their software was operating reliably, even if other factors could have also caused the critical failure.

2.3 Software Simulation Solutions

2.3.1 SIMICS

The SIMICS® product from Wind River [17] provides users with a means to create a digital twin of an embedded system for the purpose of testing. "*By using virtual platforms and simulation, software developers can decouple their work from physical hardware and its limitations during development*" [17]. SIMICS is a powerful solution for a simulation that allows the target system, in this case, a spacecraft, to be modeled such that the binaries built to run on the target will run directly in SIMICS. This is a true test-as-you-fly simulation, that can test the actual flight code as compiled for the target processor. A low-level emulator can run the processor instructions for a select set of supported processors, such as ARM, Intel, and PowerPC. Above the processor emulator sits code written in the Device Modeling Language (DML) to model the remained of the flight computer board such as memory, timers, etc. Commercial spacecraft flight computer models are available for purchase from Wind River, and custom models can be developed by the consumer.

SIMICS allows step-by-step instruction running, in a synchronous environment. Debug tools are standard for viewing any location in memory, injecting faults, and running test scenarios. This resource is commonly used by the NASA Independent Verification and Validation Facility (IV&V) Jon McBride Software Testing and Research (JSTAR) team to model flagship missions. One example of a flagship NASA mission modeled by the JSTAR team using SIMICS is Global Precipitation Measurement (GPM), resulting in the GPM Operational Simulator (GO-SIM) [18].

SIMICS is a great solution for large mission simulations, where budget and schedule time are adequate, however, the cost and time to deploy are prohibitive enough to make this option unreasonable for most SmallSats. SIMICS cost is not readily available online and must be requested from the vendor, but an estimate from one source is provided as $200,000 to $300,000 for 10-15 developers [19]. The time to develop the board-level models of all the subsystems in the spacecraft would be more involved than writing the FSW itself and would require users to learn the DML. While commercial board models can be purchased from SIMICS, SmallSat flight computer boards vary drastically from mission to mission and are unlikely to be available.

2.3.2 QEMU

QEMU is an open-source emulator meant to run target binaries as compiled, much like SIMICS. QEMU is commonly used as a test system for the LEON3 flight computer, "*a synthesizable VHDL model of a 32-bit processor compliant with the SPARC V8 architecture*" [20]. The RTEMS real-time operating system (OS), a common OS running on spacecraft, provides a LEON3 board support package and runs directly on a patched version of QEMU [21]. This allows developers to run their code targeted for a LEON3 with RTEMS on their laptop for debugging. QEMU uses the QEMU Object Model (QOM) framework for user-defined targets [22]. Users can model their flight computer board if one is not available from the open-source community, like the basic LEON3.

QEMU is a useful solution for the emulation of the target flight computer, and it comes at no cost to the mission. Like SIMICS, the time to develop the board-level models of the spacecraft will be more involved than writing the flight software and makes this solution not ideal for a first-time mission.

2.3.3 Core Flight System

"*The core Flight Software System (cFS) has reduced the costly and time-consuming process of developing software for spaceflight missions. Its flexible, layered architecture creates a development environment where system integrators can rapidly assemble a significant portion of a software system for new missions, test platforms, and technology prototypes, resulting in reduced technical, schedule, and cost risks"* [23].

The Core Flight System, or cFS, is a NASA Goddard Space Flight Center (GSFC) developed flight software that is available open-source at www.github.com/nasa/cfs. The cFS has flight heritage on large NASA missions such as GPM, Magnetospheric Multiscale Mission, Parker Solar Probe, and SmallSat missions such as Lunar Atmosphere and Dust Environment Explorer, Dellingr, and Simulation-to-Flight 1 (STF-1).

To overcome the issue of "clone-and-own" flight software from mission to mission, GSFC put together a team to analyze the heritage code from numerous missions and look for commonality. "*The analysis concluded existence of commonality in the flight software amongst all the missions especially in the discipline of Command and Data Handling (C&DH) flight software systems. The results of the analysis determined qualified C&DH flight software could be redesigned into a reusable, platform-independent software product line. The reuse would not only include the software, but the artifacts such as requirements, design, test procedures and results, and documentation, saving projects and missions the cost and effort*" [23].

The cFS is a platform-independent FSW, that runs in a layered architecture to provide abstraction from the target platform. At the first level is the Platform Support Package (PSP) that provides an API that interfaces with the board level memory, timers, etc. that can be utilized by the upper layers. Next in the stack is the Operating System Abstraction Layer (OSAL) that provides an operating system level API that can be used by the upper layers. On top of the OSAL and PSP is the Core Flight Executive (cFE), which provides commonly required software services including, time management, executive services, event services, table management, and a software communications bus. These five services can be called from the final layer of code which is the application layer of cFS. The application layer consists of modular and configurable "apps" which missions can use at their discretion. Common apps are a scheduler, house-keeping, command ingest, telemetry output, and numerous others. The apps are configured via tables managed by the cFE that provide missions a way to tailor the cFS to their mission requirements. The layered architecture of the cFS can be seen in the following figure.

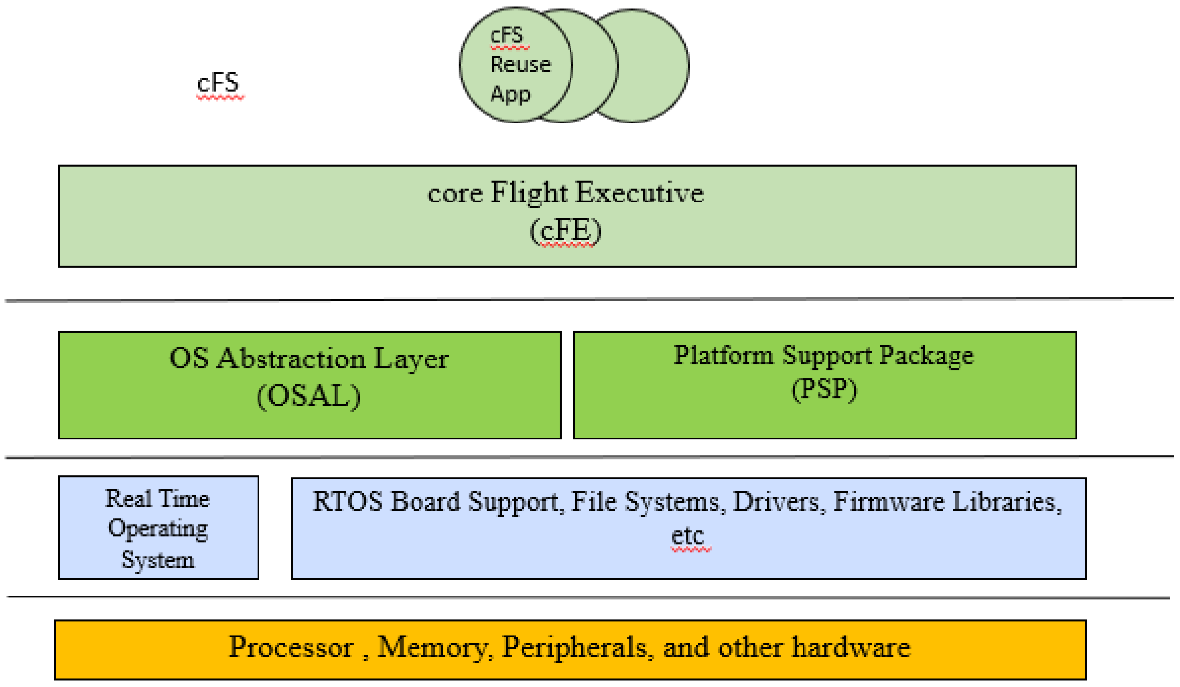

cFS
cFS Reuse App
core Flight Executive (cFE)
OS Abstraction Layer (OSAL)
Platform Support Package (PSP)
Real Time Operating System
RTOS Board Support, File Systems, Drivers, Firmware Libraries, etc
Processor , Memory, Peripherals, and other hardware

Figure 8 - cFS Layered Architecture

The cFS is built using the CMake build system and can compile for either the target system or for the Linux host. This capability allows a developer to cross-compile their code using the compiler for the flight computer, to run on the spacecraft, while also generating a version that can run on a personal computer or server. It is typical for a mission to write “simulated” applications that mimic the software bus communications of hardware interfacing applications for testing. This is a valuable test, but it is far from a test-as-you-fly configuration. The simulated apps do not model the behavior of the hardware but stub it out to allow the rest of the FSW to operate as if it were available. cFS is an excellent option for SmallSat missions to base their FSW at no cost, with substantial flight heritage from NASA and external organizations.

2.3.4 "42" Spacecraft Simulation

42 is another tool from NASA GSFC, written by Eric Stoneking and provided to the open-source community. "*42 is a comprehensive general-purpose simulation of spacecraft attitude and orbit dynamics. Its primary purpose is to support design and validation of attitude control systems, from concept studies through integration and test. 42 accurately models multi-body spacecraft attitude dynamics"* [24]. 42 is highly configurable through a series of ASCII-based text files that are easily read by the user. The orbit, spacecraft body, and external forces can all be accurately modeled to provide a simulation of the dynamic environment. In newer versions of 42, users can write and command their own "FSW" using the included attitude control module. FSW is in quotes because this tool is only modeling the commands from FSW to the actuators modeled in 42. These actuators can be reaction wheels, magnetic torquers, thrusters, or gimbals, all of which change the spacecraft dynamics. Sensors are also included in 42 and can be reported via strings either over a socket connection or through an included text file generator. 42 is a very capable tool for a dynamic simulation that is free to use, and easily configurable for SmallSats.

Chapter 3. Design and Development

3.1 Design Objectives

The research performed to investigate the causes of mission failures point to FSW as a likely cause of critical mission failures and shows that even the development teams have low confidence in the FSW they have deployed on their spacecraft. Simulation technologies are already used with success on many large missions but for rapidly developed and low-cost SmallSats the time and budget do not support the same paradigm. Numerous individual tools allow missions to perform some level of simulation, but a low-cost solution that closely resembles the spacecraft is not currently available.

The objective of this design is to provide missions with a zero-cost simulation tool that can be used to model their spacecraft, develop and test their FSW, and train spacecraft operators with the real command and telemetry interface. To meet this objective, heritage software such as cFS and 42 will be integrated with software to connect and model the components of the spacecraft. Commonly selected components will be modeled as a sample, or starting point for the missions to model their full spacecraft. An open-source ground software will be included to build and enact the command and telemetry database for communicating with the FSW. To reduce the time necessary for a new mission to utilize the simulation, it must be made easy to procure and deploy.

3.2 Design Revision 1: CubeSat in a VM

The first revision of the design was used as a proof-of-concept for the utility of such a simulator contained in a single Virtual Machine (VM) that could run on a standard laptop computer. cFS was built and configured for a generic SmallSat mission that included a minimal set of components including a C&DH, EPS, GPS, and communications subsystem. In the initial revision, the orbit of the International Space Station (ISS) was simulated via 42 in the VM, and telemetry reports were generated that included an Earth Centered Earth Fixed (ECEF) location for the GPS and a sunlight or eclipse indicator to simulate charging and discharging of the battery in the EPS. The ISS was selected as it is a common deployment opportunity for secondary payloads such as SmallSats. cFS includes a python-based tool to command the FSW via the software bus, and this was the communications system. The following figure shows the architecture of the first revision of the design.

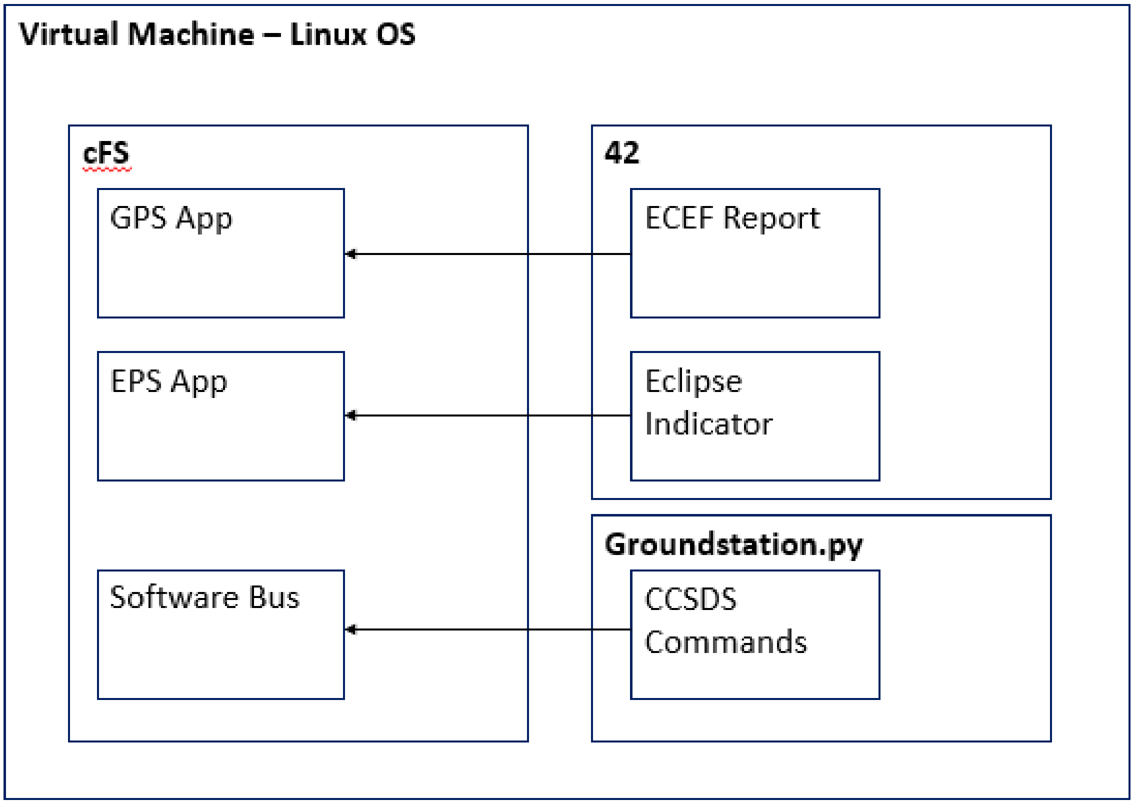


Figure 9 - CubeSat in a VM Architecture

This design proved that the concept would work, however, it was quickly constructed and does not meet the objectives laid forth. As visualized in the diagram, the information only flows into the cFS, and telemetry is not available outside of that process. Also, the hardware of the spacecraft itself was not modeled, only the resulting outputs of that hardware were mimicked by the cFS apps. The telemetry reports generated by 42 were also generated before running the cFS and were loaded by the application at runtime. This constrained the entire system to the execution time of the reports from 42. The work performed on revision one was very valuable in the long run in showing how not to design the system, as opposed to providing the final solution.

3.3 NASA Operational Simulator for Small Satellites

Learning from the mistakes made in the first revision, and with better knowledge of how to utilize cFS and 42, the NASA Operational Simulator for Small Satellites was designed. To meet the objectives laid out in section 3.1 more components needed to be added to the system to complete the design.

To model the spacecraft hardware, a “middleware” is needed to control communications and timing between the components of the simulation. NOS Engine, a heritage middleware developed by the JSTAR team at NASA IV&V was selected due to familiarity and ease of use. Using cFS as the FSW and 42 as the spacecraft dynamics simulator were both proven in revision one although the interfaces needed to be enhanced for the final design. The python-based ground software is useful but does not meet the objective, and as such the COSMOS Ground Software from Ball Aerospace was added.

In revision one, cFS apps were written to stub-out the hardware, which is not close to the real configuration of a spacecraft. For this reason, a hardware modeling layer was required to model the real behavior and input/output (I/O) characteristics of any given piece of spacecraft hardware. This layer was written using a “factory model” in C++ that can inherit common characteristics needed for the modeling framework while allowing the developer to write a plugin model of the spacecraft hardware component. Example hardware models were written based on commonly selected SmallSat components for an EPS and GPS subsystem.

Finally, a low-level interface was necessary to connect cFS to the middleware. The FSW apps need to be written once and used in either the simulation or on the flight target. There should be no code changes in the application, so the simulation is as close to "test-as-you-fly" as possible. The following diagram shows the architecture of NOS^3.

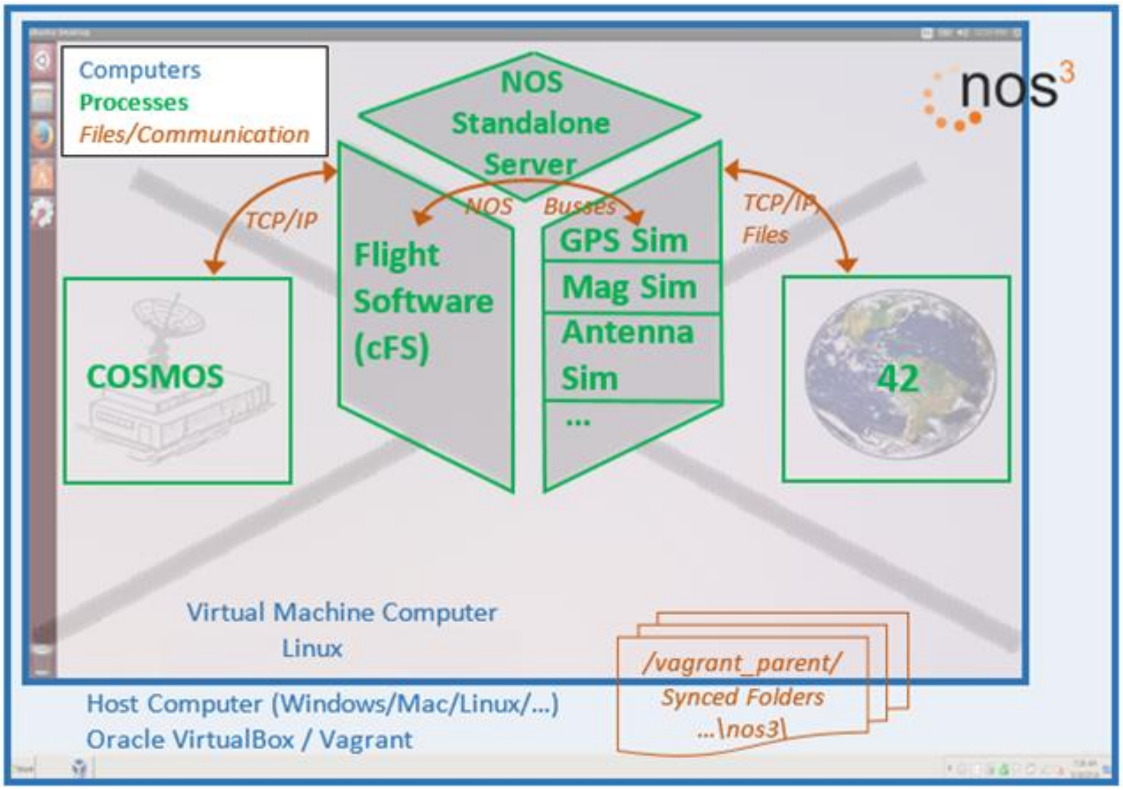


Figure 10 - NOS3 Architecture

3.4 NOS^3 Architecture and Components

3.4.1 NOS Engine

One of the primary components of NOS^3 is the NOS Engine simulation middleware that abstracts the hardware interfaces, such as Inter-Integrated Circuit (I2C), Serial Peripheral Interface

(SPI), and Universal Asynchronous Receiver-Transmitter (UART), and connects the flight software with the simulated components. The NOS Engine is a JSTAR developed software suite that provides a library of functions to simulate the hardware communication protocols that are commonly utilized in spacecraft systems. NOS Engine also provides support for various underlying protocols such as TCP/IP, inter-process communication protocol (IPC), and shared memory to transport software bus messages that represent the actual hardware bus communication. This functionality provides several unique advantages: extremely fast communications; shared memory on a single computer running the flight software and the software simulators; and distributed processing such as TCP/IP on multiple computers.

One of the challenges of simulated communications protocols (e.g., UART, I2C, SPI, etc.) is being able to represent their hardware time synchronization clocks within a software-only environment. To meet this requirement, the NOS Engine library contains methods to manipulate and distribute time between various components that are connected via software busses in place of what would normally be hardware busses. For example, within NOS^3 the NOS Engine is used to control epochs and periodic clock signals between devices. Each of the hardware models, the cFS FSW, and 42 dynamics are driven synchronously using the NOS Engine timing system so that the simulation runs each component with consistent timing.

3.4.2 NOS3 Hardware Model Framework

A component simulator development framework for adding custom mission simulators was written for NOS^3. This framework includes functionality for logging data from the simulators, XML file-based configuration of simulators, facilitates integrating custom spacecraft hardware, assists with integrating data providers such as the dynamic simulator 42 or file-based data that can be recorded from hardware components, and provides the connections to the NOS Engine middleware for communications and timing.

Several simulators have been developed for common hardware components, such as a GPS receiver, and the electrical power system. While these simulators have features that are specific to the hardware components in which they were modeled, they also present several elements useful to other developers with different components. For instance, they provide detailed, practical examples showing how simulators can be written for real-world hardware components, how to use the NOS Engine communication and timing busses, and how to receive dynamic data from 42. The framework allows the user to create software simulators of a hardware component early in the mission lifecycle, to support FSW development and testing. These simulators can be written by referencing hardware interface control documents (ICDs) or datasheets, and further augmented with characteristic data from the hardware, when available. The following figure shows the interfaces between the components of the hardware model framework.

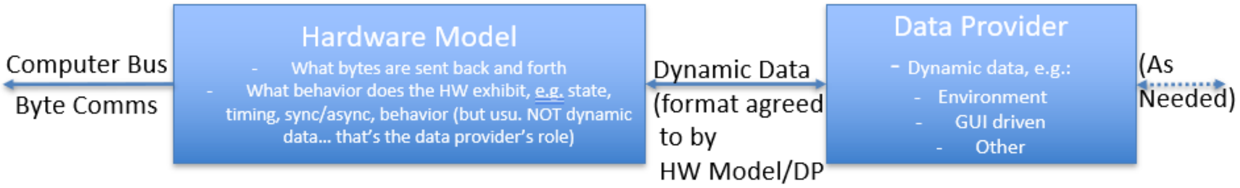

Computer Bus
Byte Comms
Hardware Model
- What bytes are sent back and forth
- What behavior does the HW exhibit, e.g. state, timing, sync/async, behavior (but usu. NOT dynamic data… that's the data provider's role)
Dynamic Data
(format agreed to by HW Model/DP
Data Provider
- Dynamic data, e.g.:
- Environment
- GUI driven
- Other
(As Needed)

Figure 11 - NOS3 Hardware Model Framework

The hardware model framework can be used to develop models that are highly complex or very simple, depending on the mission needs. For some hardware components, such as an on-board instrument that only passes data to a ground station or non-critical subsystem, the model can pass "hard-coded" values from the hardware model to the FSW if these values match the format of I/O from the actual hardware. For more critical components, such as a deployment mechanism or electrical power system, it is useful to model the hardware in as much detail as possible so the FSW can be tested extensively. The hardware model framework makes the process of writing models of any complexity easy for new developers. The necessary functions to be implemented by a hardware model depend on the device itself, but will typically be a streaming data function, an I/O callback function, or a combination of both. The UART callback function from the sample simulator is shown in the following code example. In this example, the hardware model checks the header and trailer of the command for validity and sets a configuration register in the hardware for a valid command. If the command was valid, the hardware will echo the command to the FSW to verify a successful command receipt.

```
    void SampleHardwareModel::uart_read_callback(const uint8_t *buf, size_t len)
    {
        // Retrieve data and log received data in man readable format
        std::vector<uint8_t> in_data(buf, buf + len);
        sim_logger->debug("SampleHardwareModel::uart_read_callback:  REQUEST %s",
            SimIHardwareModel::uint8_vector_to_hex_string(in_data).c_str());

        // Check if message is incorrect size
        if (in_data.size() != 9)
        {
            sim_logger-
>debug("SampleHardwareModel::uart_read_callback:  Invalid command size of %d received
!", in_data.size());
            return;
        }

        // Check header - 0xDEAD
        if ((in_data[0] != 0xDE) || (in_data[1] !=0xAD))
        {
            sim_logger-
>debug("SampleHardwareModel::uart_read_callback:  Header incorrect!");
            return;
        }

        // Check trailer - 0xBEEF
        if ((in_data[7] != 0xBE) || (in_data[8] !=0xEF))
        {
            sim_logger-
>debug("SampleHardwareModel::uart_read_callback:  Trailer incorrect!");
            return;
        }

        // Process command type
        switch (in_data[2])
        {
            case 1:
                sim_logger-
>debug("SampleHardwareModel::uart_read_callback:  Configuration command received!");
                _millisecond_stream_delay = (in_data[3] << 24) +
                                            (in_data[4] << 16) +
                                            (in_data[5] << 8 ) +
                                            (in_data[6]);
                sim_logger-
>debug("SampleHardwareModel::uart_read_callback:  New millisecond stream delay of %d"
, _millisecond_stream_delay);
                _second_stream_delay = double(_millisecond_stream_delay) / 1000;
                break;

            case 2:
                sim_logger-
>debug("SampleHardwareModel::uart_read_callback:  Other command received!");
                break;

            default:
                sim_logger-
>debug("SampleHardwareModel::uart_read_callback:  Unused command received!");
                break;
        }

        // Prepare to echo back valid command
        std::vector<uint8_t> out_data = in_data;

        // Log reply data in man readable format and ship the message bytes off
        sim_logger->debug("SampleHardwareModel::uart_read_callback:  REPLY %s",
            SimIHardwareModel::uint8_vector_to_hex_string(out_data).c_str());
        _uart_connection->write(&out_data[0], out_data.size());
    }
```

3.4.3 42

As the spacecraft is in orbit, variables such as its position, velocity, orientation, solar radiation direction and intensity, and magnetic field direction and intensity change over time. While the actual hardware signals corresponding to dynamic inputs can be determined from hardware datasheets and user manuals, the dynamic inputs must also be identified for an accurate simulation. To connect 42 properly to the simulated hardware models, the 42 socket layer is utilized versus the log files used in revision one. This allows the 42 simulator to send updated dynamic data synchronously to the simulated hardware model for packaging into the proper I/O format. As 42 executes its time-step all data relevant to the simulator, such as the ECEF position for the GPS, is sent over a socket to each simulator. The hardware model for the simulator must then parse the message from the socket to collect the value, and format its message to the FSW exactly as sent by the real piece of hardware.

To keep time synchronized, a NOS Engine timing option needed to be added to the 42 source code to drive its time-step at the same pace as the rest of the NOS^3 components. This work was performed and contributed back to the open-source 42 repository where it was eventually added to the maintained release of 42. When a user wants to utilize the NOS Engine time synchronization functionality in 42, they can specify "NOS3" as the time option in the 42 configuration files. An additional file is then required to configure the NOS Engine time bus, labeled Inp_NOS3.txt. An example of each of these configuration files is shown in the following code examples.

```
/* Inp_Sim.txt */
<<<<<<<<<<<<<<<<<  42: The Mostly Harmless Simulator  >>>>>>>>>>>>>>>>>
************************** Simulation Control **************************
NOS3                            !  Time Mode (FAST, REAL, EXTERNAL, or NOS3)
10000.0   0.1                   !  Sim Duration, Step Size [sec]
```

Figure 12 - 42 Inp_Sim.txt

```
/* Inp_NOS3.txt */
<<<<<<<<<<<<<<<<  42 NOS3 Time Configuration File  >>>>>>>>>>>>>>>>>>>
command                         !  NOS3 Time Bus
tcp://127.0.0.1:12001           !  NOS3 Time Connection String
```

Figure 13 - 42 Inp_NOS3.txt

3.4.4 cFS and the Hardware Library

Much like the cFS provides the OSAL and PSP layers to abstract the platform upon which the FSW will execute, a Hardware Library (HWLIB) was written to abstract the hardware I/O drivers that communicate with either NOS^3 or real hardware. To achieve this the drivers, such as I2C, UART, and SPI were written with a common API that can be called in the cFS applications. The source code for this API is then targeted at build time, and both the simulator and a cross-compiled version for the flight computer are generated. The same cFS application code is applied to both targets, with only the low-level drivers abstracted. For example, the following code example shows the API function call to perform a transaction between the FSW and an I2C component, contained in libi2c.h and included by the cFS application, followed by the implementation of those calls for the NOS^3 target, contained in libi2c.c and built at compile time.

```c
/* libi2c.h */
int32_t i2c_master_transaction(int32_t handle, uint8_t addr, void * txbuf, uint8_t tx
len, void * rxbuf, uint8_t rxlen, uint16_t timeout);
```

Figure 14 - NOS3 libi2c.h

```c
/* libi2c.c */
int32_t i2c_master_transaction(int32_t handle, uint8_t addr, void * txbuf, uint8_t tx
len, void * rxbuf, uint8_t rxlen, uint16_t timeout)
{
    int32_t result = OS_ERROR;
    if(handle < NUM_I2C_DEVICES)
    {
        OS_MutSemTake(nos_i2c_mutex);
        /* get i2c device handle */
        NE_I2CHandle **dev = &i2c_device[handle];
        if(*dev == NULL)
        {
            /* get nos i2c connection params */
            const nos_connection_t *con = &nos_i2c_connection[handle];
            /* try to initialize master */
            *dev = NE_i2c_init_master3(hub, 10, con->uri, con->bus);
            if(*dev == NULL)
            {
                OS_printf("nos i2c_init_master failed\n");
            }
        }
        /* i2c transaction */
        if(*dev)
        {
            if(NE_i2c_transaction(*dev, addr, txbuf, txlen, rxbuf, rxlen) == NE_I2C_S
UCCESS)
            {
                result = OS_SUCCESS;
            }
        }
        OS_MutSemGive(nos_i2c_mutex);
    }
    return result;
}
```

Figure 15 - NOS3 libi2c.c

The application can utilize the i2c_master_transaction function, and the underlying implementation will send the data over the NOS Engine middleware to the component simulator. This HWLIB allows for the lowest level of simulation that can be achieved when running on the host OS, like Linux for NOS^3.

When a mission selects their flight computer, they will need to implement the I2C API, like shown in the code example for NOS^3, for their specific device. The dual-target model, simulation and flight, is shown in the following diagram.

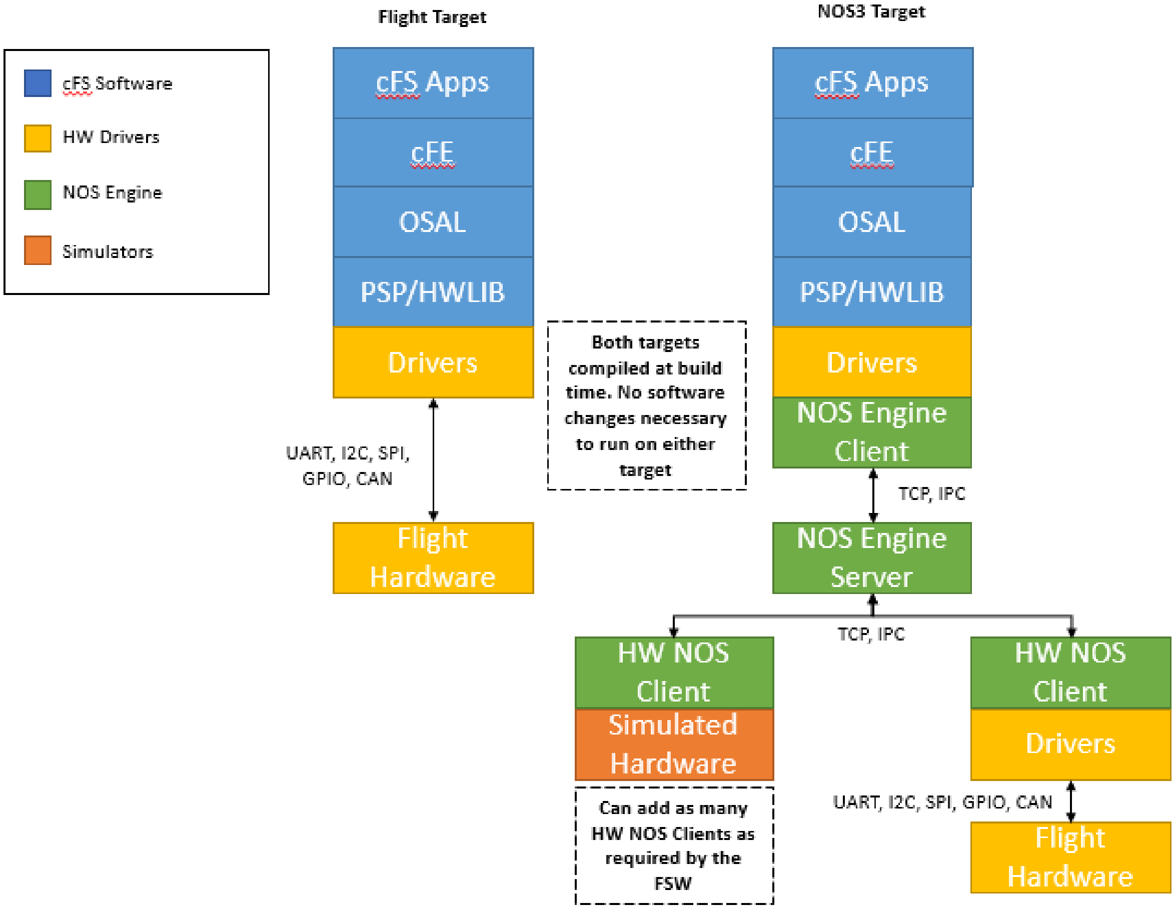


Figure 16 - NOS3 HWLIB Target Layers

3.4.5 COSMOS

COSMOS, an open-source command and control software package, was integrated into NOS3 to allow end-to-end testing of the FSW and to enable the "test as we fly" philosophy. COSMOS provides a sophisticated framework for the command and control of satellites and other embedded systems. COSMOS was integrated into NOS3 using a collection of text configuration files. A single text file provides the TCP/IP socket configuration information, while additional text files are autogenerated to define the byte patterns representing telemetry and command data sent from the spacecraft to the ground, and vice versa. It should be noted that despite COSMOS being integrated into NOS^3, it is not architecturally required, and could be replaced by a similar command and control software that supports UDP connection. If a mission selects ground software that is not COSMOS, such as the Jet Propulsion Laboratory's AMMOS Instrument Toolkit, it can be used for the same purpose. The important takeaway for the mission is to use the same ground software that will be used for flight. This will allow the same command and telemetry definitions to be fully tested in NOS^3.

COSMOS is connected to NOS^3 via a UDP connection to the cFS Command Ingest (CI) and Telemetry Output (TO) applications. The CI and TO apps have a "custom" interface to define the I/O characteristics of the communications system. As mentioned, this interface is currently UDP to make NOS^3 generic, but when a mission selects their radio, they can generate a simulated hardware model and utilize the previously described HWLIB in this custom interface. This will allow a more accurate simulation of the communications system for the spacecraft. The following diagram shows how the final end-to-end system should be architected.

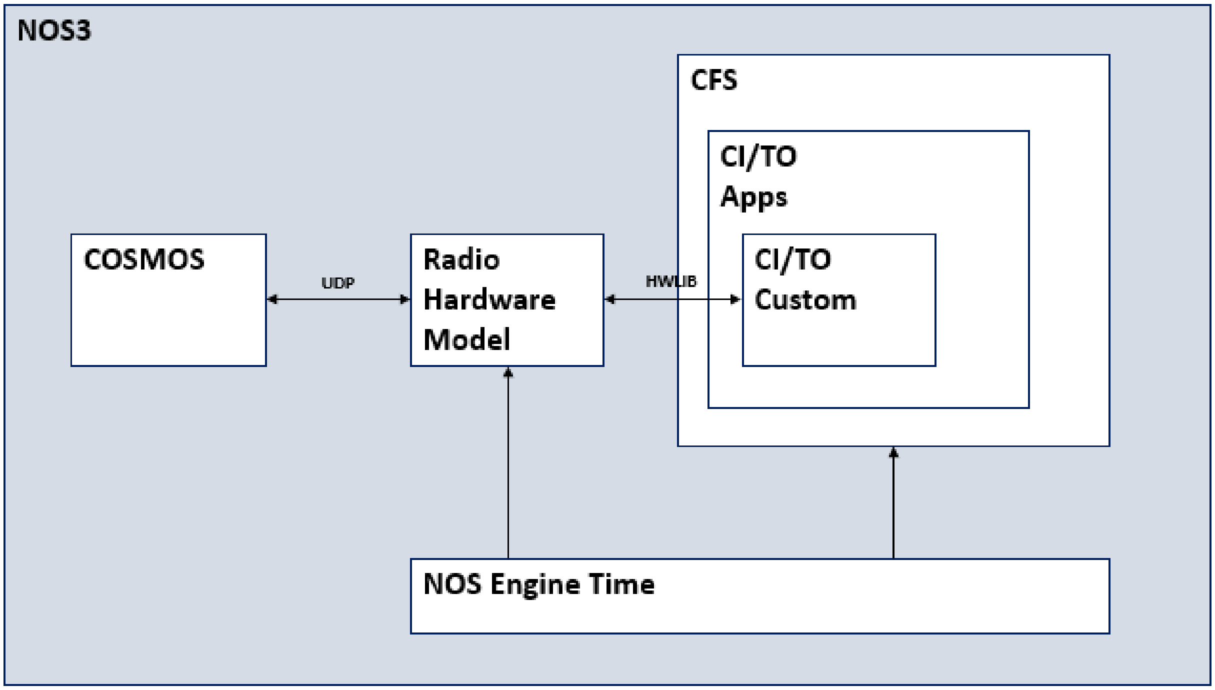


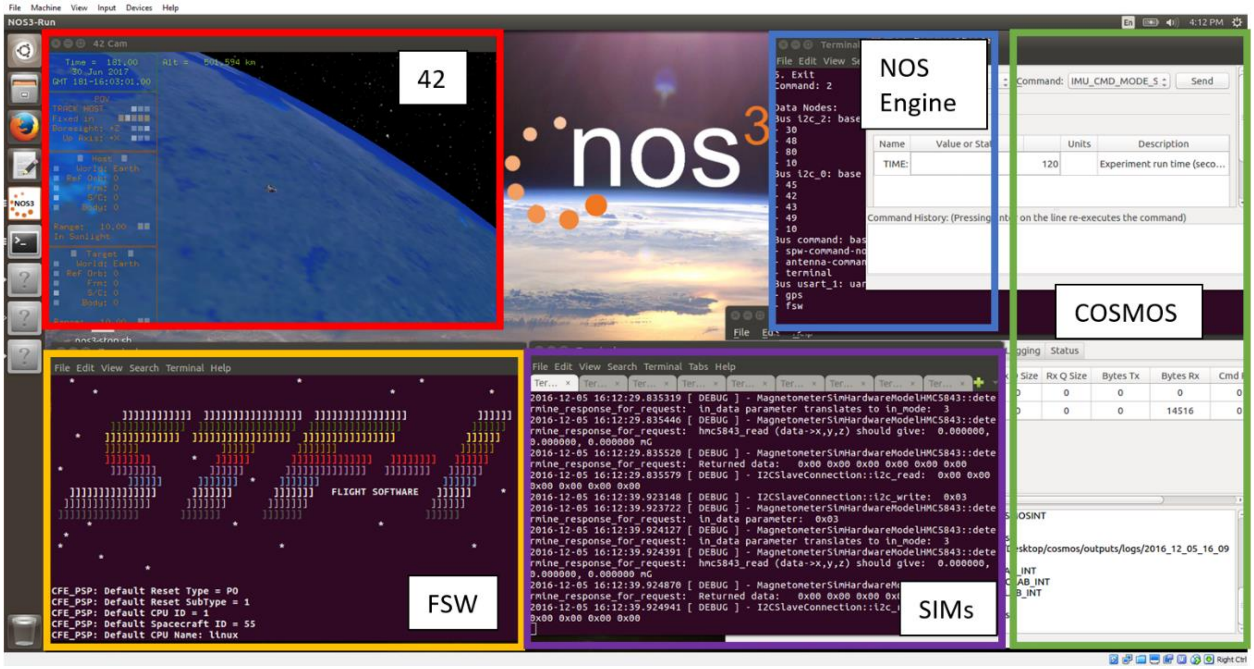
Figure 17 - NOS3 COSMOS Block Diagram

3.4.6 Deployment and Virtual Environment

To make NOS[3] as easy to use as possible, an automated deployment solution was written using a combination of Vagrant and Ansible scripting to generate a virtual machine that runs in Oracle's Virtual Box software. All three software products, Vagrant, Ansible, and Virtual Box are available at no cost to the user.

This automated deployment solution allows a mission to generate as many copies of the NOS[3] as they would like to support their development and testing. Each environment will be the same when the installation completes, and because the virtual environment is executed on Virtual Box, it can be used on any host operating system such as Windows, macOS, or Linux. This provides maximum flexibility for the development team. Having an easily deployable solution, with no limit on the number of instances, reduces risk and increases the number of testing resources available to a mission.

The NOS^3 virtual environment comes with all of the previously described components installed and ready to run. Users can use the included ease-of-use scripts to build their FSW and simulated hardware models by calling *make*, configure and run the full NOS^3 suite by calling *make launch*, and halt the simulation by calling *make stop*. Upon calling *make launch* each of the components will start execution and the view of the virtual environment will look similar to the following screenshot.



```
5. Exit
Command: 2

Data Nodes:
Bus i2c_2: base
- 30
- 48
- 80
- 10
Bus i2c_0: base
- 45
- 42
- 43
- 49
- 10
Bus command: bas
- spw-command-no
- antenna-comman
- terminal
Bus usart_1: uar
- gps
- fsw
```

Name	Value or Stat		Units	Description
TIME:		120		Experiment run time (seco…

Command History: (Pressing Enter on the line re-executes the command)

Size	Rx Q Size	Bytes Tx	Bytes Rx	Cmd
0	0	0	0	0
0	0	0	14516	0

```
CFE_PSP: Default Reset Type = PO
CFE_PSP: Default Reset SubType = 1
CFE_PSP: Default CPU ID = 1
CFE_PSP: Default Spacecraft ID = 55
CFE_PSP: Default CPU Name: linux
```

```
2016-12-05 16:12:29.835319 [ DEBUG ] - MagnetometerSimHardwareModelHMC5843::dete
rmine_response_for_request:  in_data parameter translates to in_mode:  3
2016-12-05 16:12:29.835446 [ DEBUG ] - MagnetometerSimHardwareModelHMC5843::dete
rmine_response_for_request:  hmc5843_read (data->x,y,z) should give:  0.000000,
0.000000, 0.000000 mG
2016-12-05 16:12:29.835520 [ DEBUG ] - MagnetometerSimHardwareModelHMC5843::dete
rmine_response_for_request:  Returned data:   0x00 0x00 0x00 0x00 0x00 0x00
2016-12-05 16:12:29.835579 [ DEBUG ] - I2CSlaveConnection::i2c_read:  0x00 0x00
0x00 0x00 0x00 0x00
2016-12-05 16:12:39.923148 [ DEBUG ] - I2CSlaveConnection::i2c_write:  0x03
2016-12-05 16:12:39.923722 [ DEBUG ] - MagnetometerSimHardwareModelHMC5843::dete
rmine_response_for_request:  in_data parameter:  0x03
2016-12-05 16:12:39.924127 [ DEBUG ] - MagnetometerSimHardwareModelHMC5843::dete
rmine_response_for_request:  in_data parameter translates to in_mode:  3
2016-12-05 16:12:39.924391 [ DEBUG ] - MagnetometerSimHardwareModelHMC5843::dete
rmine_response_for_request:  hmc5843_read (data->x,y,z) should give:  0.000000,
0.000000, 0.000000 mG
2016-12-05 16:12:39.924870 [ DEBUG ] - MagnetometerSimHardwareMo
rmine_response_for_request:  Returned data:   0x00 0x00 0x00 0x
2016-12-05 16:12:39.924941 [ DEBUG ] - I2CSlaveConnection::i2c_
0x00 0x00 0x00 0x00
```

Figure 18 - NOS3 Virtual Environment Display

Chapter 4. Results

4.1 The Simulation-to-Flight 1 Mission

The best way to prove the utility of NOS^3 is the direct application of the simulator on a CubeSat mission. Simulation-to-Flight 1, or STF-1, is a CubeSat mission led by the NASA IV&V Facility's JSTAR team, in partnership with West Virginia University (WVU), TMC Technologies, the West Virginia Space Grant Consortium, and the West Virginia High Tech Foundation. The picture below shows STF-1 integrated into the deployment canister in preparation for launch.

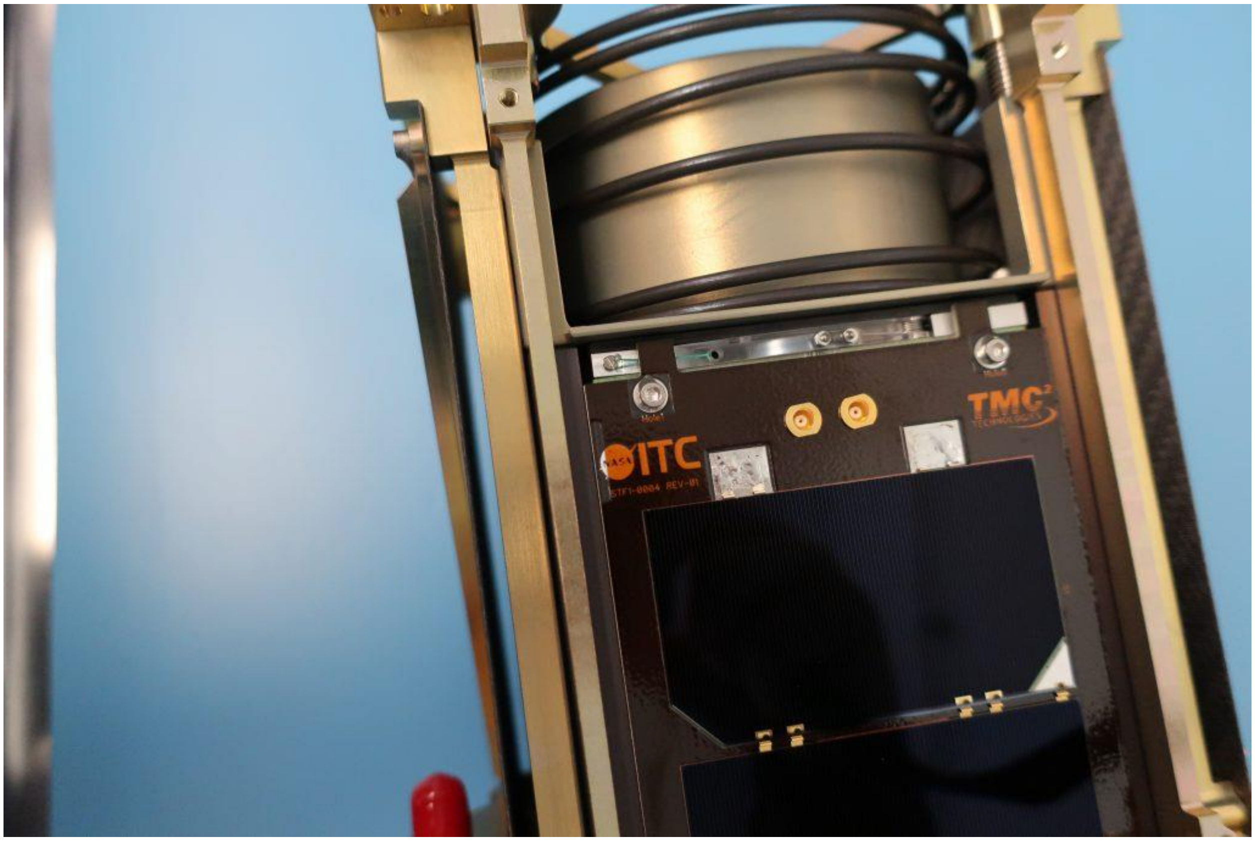


Figure 19 - STF-1 Integrated in Deployment Canister

As a result of the demonstrated successes of JSTAR software-only simulation environments and an opportunity to launch a spacecraft to demonstrate technologies that benefit NASA programs through the CSLI, the STF-1 team was formed. The primary purpose of STF-1 was to determine and demonstrate the value of developing, utilizing, and maintaining a software-only simulation during the project lifecycle. A diverse set of science experiments, provided by WVU, allowed the project to expand the mission's overall objective to include science experiments and technology demonstrations. The instruments include a cluster of Micro-Electro-Mechanical Systems (MEMS) Inertial Measurement Units (IMU) to produce attitude knowledge; a space-weather experiment including a Geiger counter and Langmuir probe; a III-V Nitride-based materials optoelectronics experiment; and a Novatel OEM615 GPS coupled with advanced algorithms for precise orbit determination. The science experiments enhanced the mission capabilities, as well as providing a diverse set of instruments to assess how the simulator would support instrument development. The following figure shows the number of STF-1 components that were modeled in NOS[3].

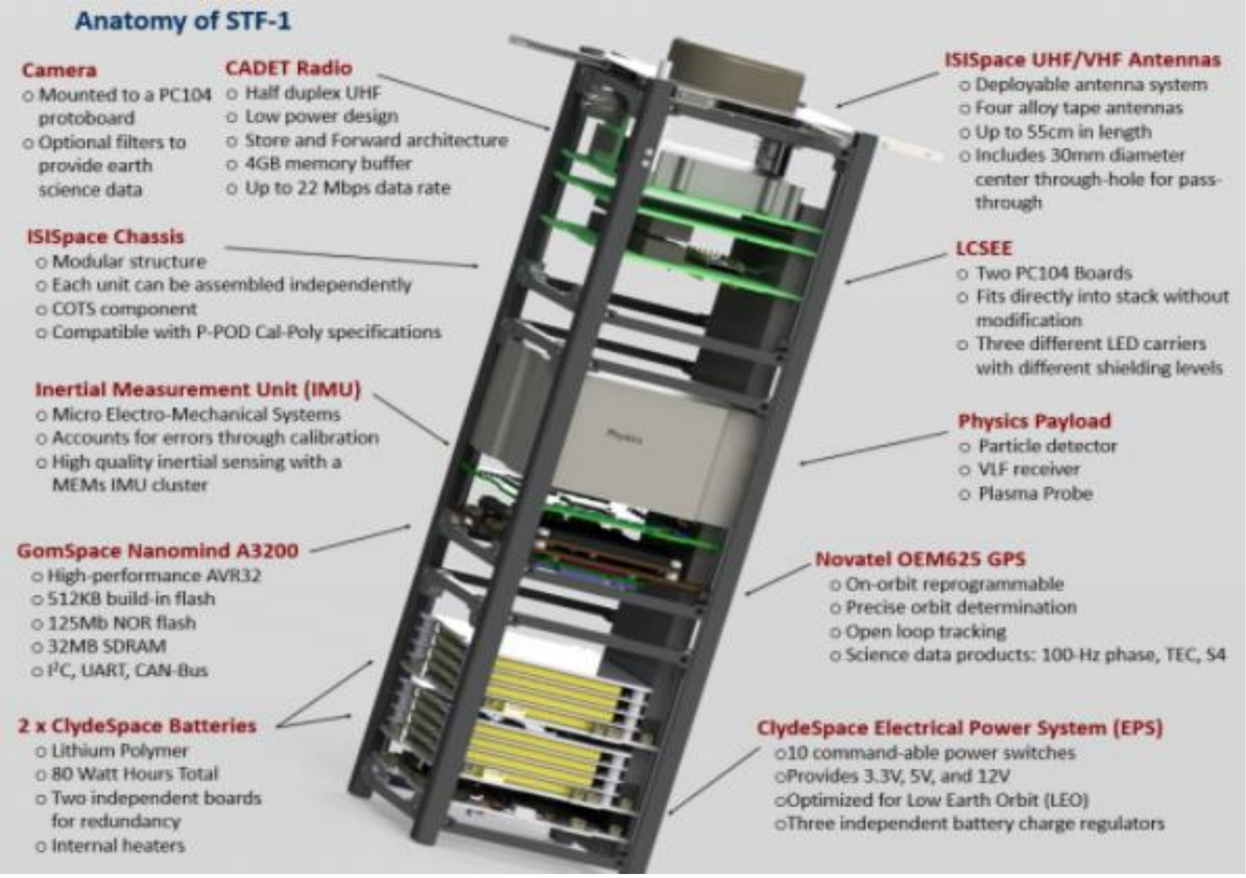

Anatomy of STF-1
Camera
○ Mounted to a PC104 protoboard
○ Optional filters to provide earth science data
CADET Radio
○ Half duplex UHF
○ Low power design
○ Store and Forward architecture
○ 4GB memory buffer
○ Up to 22 Mbps data rate
ISISpace UHF/VHF Antennas
○ Deployable antenna system
○ Four alloy tape antennas
○ Up to 55cm in length
○ Includes 30mm diameter center through-hole for pass-through
ISISpace Chassis
○ Modular structure
○ Each unit can be assembled independently
○ COTS component
○ Compatible with P-POD Cal-Poly specifications
LCSEE
○ Two PC104 Boards
○ Fits directly into stack without modification
○ Three different LED carriers with different shielding levels
Inertial Measurement Unit (IMU)
○ Micro Electro-Mechanical Systems
○ Accounts for errors through calibration
○ High quality inertial sensing with a MEMs IMU cluster
Physics Payload
○ Particle detector
○ VLF receiver
○ Plasma Probe
GomSpace Nanomind A3200
○ High-performance AVR32
○ 512KB build-in flash
○ 125Mb NOR flash
○ 32MB SDRAM
○ I^2C, UART, CAN-Bus
Novatel OEM625 GPS
○ On-orbit reprogrammable
○ Precise orbit determination
○ Open loop tracking
○ Science data products: 100-Hz phase, TEC, S4
2 x ClydeSpace Batteries
○ Lithium Polymer
○ 80 Watt Hours Total
○ Two independent boards for redundancy
○ Internal heaters
ClydeSpace Electrical Power System (EPS)
○10 command-able power switches
○Provides 3.3V, 5V, and 12V
○Optimized for Low Earth Orbit (LEO)
○Three independent battery charge regulators

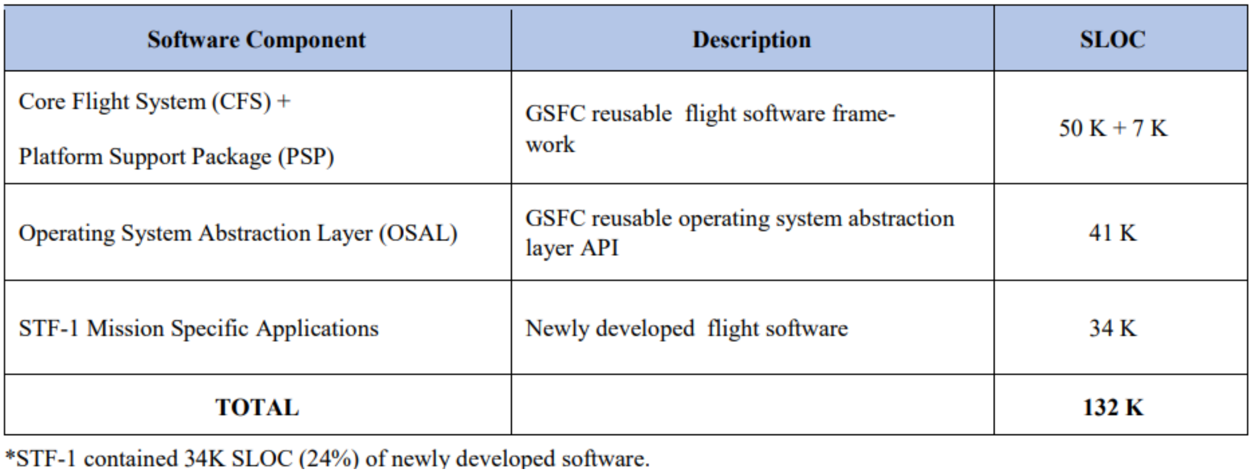
Figure 20 - Anatomy of STF-1

The STF-1 spacecraft was launched on December 16th of 2018 and continues to operate nominally on orbit. The following subsections describe the complexity of the STF-1 FSW, and the major benefits of NOS^3 realized through the STF-1 mission.

4.1.1 STF-1 FSW Complexity

As a metric to assess the overall software complexity, the Source-Lines-of-Code (SLOC) utility (SLOCCount) was executed against the STF-1 flight software [25]. This utility measures the size of a computer program by counting the number of lines in the program's source code. Additionally, the results of the SLOC utility were used as an indicator of software size for the Constructive Cost Model, a procedural cost estimation model [26]. Table 2 lists the STF-1 SLOC count, with the RTOS and drivers not included because they were vendor-provided. Of the 132,000 total SLOC, roughly 25% of the software was newly developed for the STF-1 mission.

Table 2 - STF-1 SLOCCount



Software Component	Description	SLOC
Core Flight System (CFS) + Platform Support Package (PSP)	GSFC reusable flight software frame-work	50 K + 7 K
Operating System Abstraction Layer (OSAL)	GSFC reusable operating system abstraction layer API	41 K
STF-1 Mission Specific Applications	Newly developed flight software	34 K
TOTAL		**132 K**

*STF-1 contained 34K SLOC (24%) of newly developed software.

Using the Constructive Cost Model, SLOC Count estimates that the STF-1 applications take 8.25 person-months for development, but this metric does not consider integration testing time, and access to flight hardware for testing, which as previously discussed are limiting factors for SmallSats. The STF-1 flight software is not trivial, due to semi-autonomous on-orbit operations needed to perform science experiments, record science data, and transmit the data to the ground station during downlink periods of only a few minutes in duration. The flight software was required to simultaneously provide the following core functionalities: 1) operate without communications from the ground station; 2) monitor the power level of the EPS for safely executing time-lapse science experiments; 3) start, stop, and pause experiments; 4) communicate with various STF-1 hardware components such as sensors, radio, camera, and the deployable antenna. This flight software complexity results in increased mission risk with respect to development and testing schedule. This type of embedded hardware testing is typically dependent upon either flight hardware or ETU availability.

4.1.2 Reduced Reliance on Hardware Resources

NOS^3 enabled multiple STF-1 developers to work in parallel without monopolizing either a single simulator, ETU, or spacecraft flight computer, thus reducing the STF-1 mission's reliance on hardware resources. For example, while one engineer was developing the EPS software, another engineer was developing the communications software. Neither engineer needed to use the hardware for their development and initial testing. NOS^3 was used extensively by the STF-1 software development team for all aspects of flight software development and testing. Throughout the three person-months in which the majority of STF-1 software development was accomplished, each team member maintained a copy of the NOS3 virtual environment. The virtual environment provided realistic inputs and feedback to the flight software while under development and testing. Additionally, NOS^3 provided a suitable test environment to support STF-1 flight software integration testing. Like other SmallSat missions, the STF-1 spacecraft hardware was expensive, limited in supply with few spares or ETUs, and it needed to be configured and set up quickly to support FSW development testing. NOS^3 provided the ability to develop and test most FSW functionality without requiring a hardware-in-the-loop test configuration. Hardware is still needed to test certain performance and timing requirements, but this requirement is minimal in comparison to testing the full FSW stack. Without NOS^3, STF-1 developers would not have been able to develop and test software applications in parallel to these activities.

4.1.3 Risk Reduction and Comprehensive testing

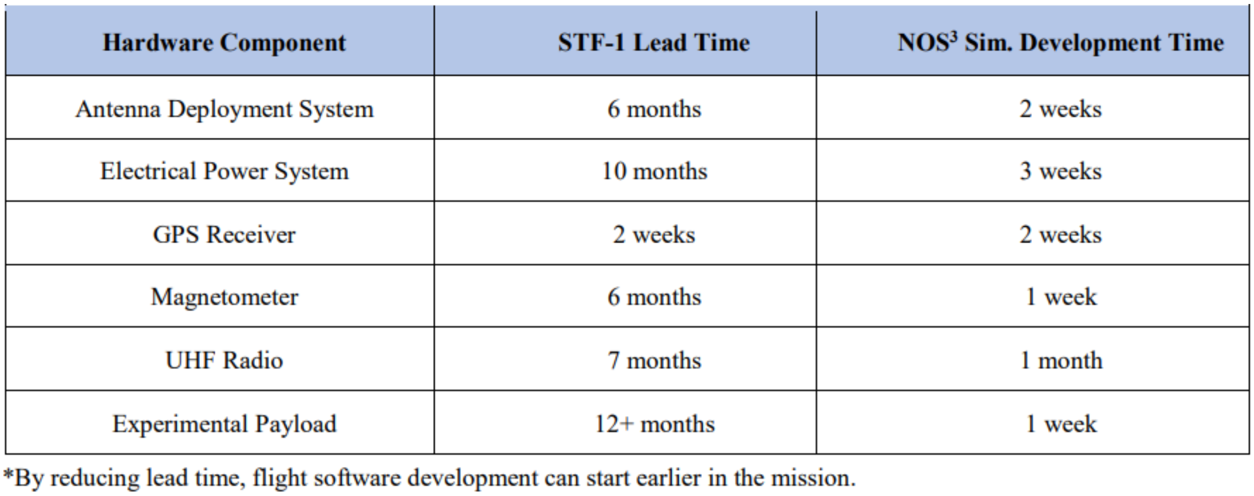
The effortless deployment process of the NOS^3 software allowed the STF-1 team to set up and configure a large number of identical NOS^3 environments to cross-train personnel and to support risk reduction testing during the STF-1 software development. For example, NOS^3 was provided to multiple interns during the summer months to support mission understanding, perform static code analysis, and complete additional software testing of custom STF-1 software applications. The additional simulation resources allowed the team to test how the various STF-1 software applications would respond to adverse conditions, thus ensuring STF-1 software robustness. One of the most critical STF-1 software applications, the manager application, which is responsible for semi-automating the spacecraft operations, was exhaustively tested using NOS^3. The test engineer, who was unfamiliar with NOS^3 and the STF-1 mission, was able to quickly and effectively test the entry and exit conditions and fault detection and mitigation logic contained in the manager application. This led to complete confidence in the semi-autonomous operations of STF-1 while on-orbit and has since been proven through over two years of nominal operations of the spacecraft.

NOS^3 also allows the tester to introduce fault conditions that are too dangerous or expensive to test using hardware, which further reduced mission risk and raised confidence in the flight software. For example, the EPS subsystem is very expensive and had a lead-time of over 10 months and as such is not able to be shorted or otherwise damaged during testing of the FSW. Through NOS^3 individual faults were able to be injected into the simulation, such as a short in one of the power busses, to ensure the FSW would respond properly to save the health of the spacecraft.

4.1.4 Software Development and Testing Schedule

NOS^3 was able to increase the STF-1 development team's control of the software development schedule and to demonstrate how future software development effort schedules can be shifted ahead of the receipt of hardware components. By decoupling the software development schedule from the availability of flight hardware and ETUs, FSW development can occur very early in the already tight mission schedule. Table 3 reports the lead times associated with the major STF-1 flight components as compared with the associated development time for the NOS^3 hardware simulator.

Table 3 - STF-1 Component Lead Times



Hardware Component	STF-1 Lead Time	NOS^3 Sim. Development Time
Antenna Deployment System	6 months	2 weeks
Electrical Power System	10 months	3 weeks
GPS Receiver	2 weeks	2 weeks
Magnetometer	6 months	1 week
UHF Radio	7 months	1 month
Experimental Payload	12+ months	1 week

*By reducing lead time, flight software development can start earlier in the mission.

From this data, it is evident that the level of effort required to develop a hardware-equivalent simulator for the STF-1 mission with NOS3 was rather minimal. Furthermore, a NOS^3 hardware simulator can be scoped, and planned (level of effort, required simulator fidelity, etc.), whereas hardware lead times from vendors change and are regularly delayed. NOS^3 allowed STF-1 software development to begin as scheduled, rather than after the hardware was delivered. As a result, the FSW was developed and tested before environmental testing, and the team benefited from an additional 6 months of schedule time for operations training, and day-in-the-life testing before the spacecraft was required to be delivered to CSLI.

4.1.5 Results of NOS^3 as Proven by STF-1

From the direct application of NOS^3 to a SmallSat mission, the primary questions of this research can be answered. The problem statement in Chapter 1 asks, can the implementation of simulations performed for large missions, be tailored to fit the needs of SmallSats, and increase the reliability of the resulting FSW. The results of the application of NOS^3 to the STF-1 mission would mean yes, the same simulation techniques used for large missions can be made cost-effective enough for a SmallSat and did increase the reliability of the STF-1 FSW. Two additional questions arose from the research into failures of SmallSats being 1.) can simulation of the spacecraft hardware act as another alternative to flight software testing on ETUs where they are either capitalized in integration testing or unavailable due to budget constraints; and 2.) can simulation of the spacecraft hardware allow more time and resources for systems-level testing. The answer to question 1, as laid out in section 4.2 is yes, and as laid out in section 4.4 the answer to question 2 is also yes. NOS^3 has the proven capability to increase testing resources and allow more time for testing, while not being dependent on the availability of flight hardware or ETUs.

4.2 NOS^3 Testing

4.2.1 NOS^3 Test Configuration

Testing of the NOS^3 simulation environment was performed using the NOS^3 Sample application and simulator. As discussed in section 3.4.2, NOS^3 includes a hardware modeling framework to write simulators of hardware components. To provide users with a proper example of a cFS application, and corresponding NOS^3 hardware model, a sample application and simulator were written and provided in the open-source suite.

The Sample hardware model is written for an arbitrary spacecraft sensor, that sends a single telemetry point as a float data type. To provide an example of receiving dynamic data from 42 the sun vector in the body frame is sent to the hardware model via the socket interface. The sample application interfaces to the simulator using the UART interface from the HWLIB, which then sends and receives bytes over the NOS Engine middleware. This sample sensor simulator takes a single command that configures the frequency of the telemetry point being output over the UART. Although this is an arbitrary device, it is based upon typical sensors such as an inertial measurement unit or a magnetometer. The end-to-end sample test setup is shown in the following diagram.

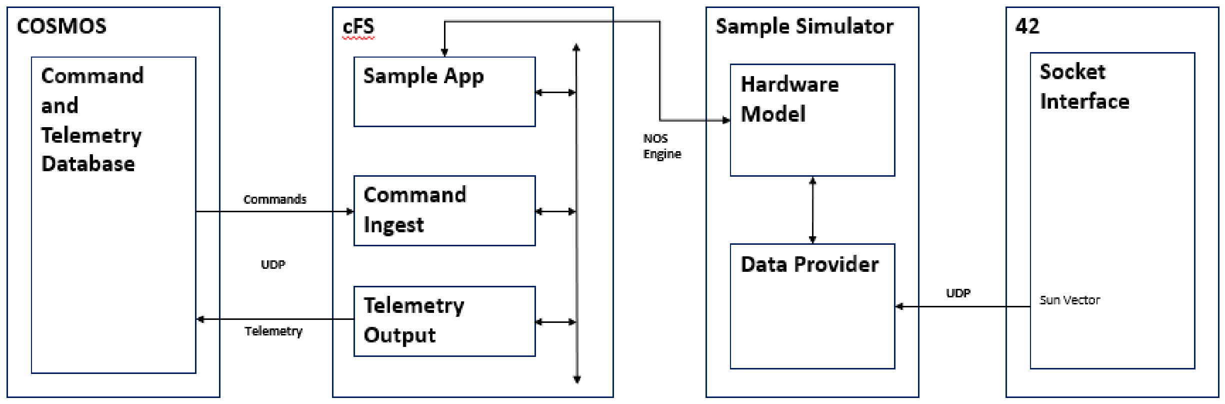


Figure 21 - NOS3 Sample Test Setup Diagram

4.2.2 NOS^3 Sample Test Results

To perform the test, the output frequency was stepped from the default frequency of 1Hz, up to 1000Hz via commands from COSMOS, and the number of telemetry packets that are received in COSMOS from the Sample app was recorded. The following table shows the results of the test performed for 60 seconds.

Table 4 - NOS Sample Test Results Initial

Output Frequency	**Data Points from 42**	**Data Points from Sim**	**Telemetry Points Received in COSMOS**	**Throughput Percentage**
1Hz	60	60	60	100%
10Hz	600	600	600	100%
100Hz	6000	6000	5643	94.05%
1000Hz	60000	60000	20520	34.20%

In the first attempt of the test, the results at 100 and 1000Hz showed a loss of roughly 6% at 100Hz and roughly 70% at 1000Hz. As there should be no packet loss in the system, and NOS Engine has been exhaustively tested, the code in the HWLIB was inspected for faults. The initial thought was the simulated UART buffer was overflowing due to the increased frequency of data, and the FSW was not able to process the data quickly enough before the next message is received. However, it was found that a mutex in the HWLIB was the cause of the issue. For I2C and SPI devices, where multiple devices can operate on a single bus, a mutex is necessary to protect the HWLIB in the cFS application from being interrupted by another application before it completes. For a UART device, the communications are point-to-point, from the flight computer to the device, and the mutex caused this issue because a single mutex was being used across multiple UART devices like required for I2C. Upon removing the mutex from the UART HWLIB throughput increased to the expected 100%.

Table 5 - NOS3 Sample Test Results after Patch

Output Frequency	**Data Points from 42**	**Data Points from Sim**	**Telemetry Points Received in COSMOS**	**Throughput Percentage**
1Hz	60	60	60	100%
10Hz	600	600	600	100%
100Hz	6000	6000	6000	100%
1000Hz	60000	60000	60000	100%

4.3 NOS3 Known Limitations

While NOS^3 has been proven to be a solution for the problem presented in this research, there are known limitations that should be noted. The first, and most important to note is not a fault of NOS^3, but a problem with the assumptions made in developing NOS^3 hardware models and cFS applications. The assumption is when a hardware component is being modeled and the cFS application is being developed and tested, that the vendor-provided documentation accurately describes the behavior of the hardware. Through direct application on the STF-1 mission, this has proven to be false. It is typical for vendor documentation to either inaccurately describe the byte-level communications or omit critical information from the interface. This requires the hardware models to be corrected to simulate the actual behavior of the device. These corrections would be required at integration and test time even if NOS^3 is not used. In the case of NOS^3 simple corrections can be made during integration and testing of the flight hardware or ETUs, whereas without NOS^3 the FSW would be fully untested at the time of integration and testing of the hardware.

The second limitation of NOS^3 is modeling precise timing with the spacecraft hardware. Time is a critical component in NOS^3; however, extremely small timing requirements cannot be met in a virtualized and simulated system. For example, a 10-microsecond timeout, between an I2C write and a subsequent I2C read to a device cannot be met in this simulation. NOS Engine and NOS^3 were tested up to 1000Hz for throughput, as no common spacecraft component sends data at such a high frequency. As such the clocks in NOS^3 are configured to operate at 100Hz, and a 10-microsecond timeout cannot be guaranteed. This is in part due to the clock, but also from running NOS^3 on Linux, as opposed to a real-time operating system. When building the NOS^3 target for simulation, and not for flight, the timeout settings in the I/O interface are ignored.

4.4 NOS^3 Usage

On August 3rd, 2018, the open-source release of NOS^3 was approved by NASA and made available to the public on the internet hosting and Git version control website called GitHub. This open-source release can be viewed by anyone with a GitHub account and cloned or forked for their use. Since the open-source release, the code has been starred by 85 users and forked by 33. These users range from individuals to government agencies, to universities, to private companies. STF-1 is the best example of NOS^3 being applied directly to a SmallSat mission, and as such NOS^3 won the prestigious award of runner-up for the NASA Software of the Year in 2019. This only represents the publicly available information on the open-source software. Being open-source, NOS^3 could be cloned and in use by any number of projects or companies.

Chapter 5. Conclusions and Future Work

5.1 Summary

Small Satellites, or satellites with a mass of less than 180kg as defined by NASA, have been increasing in the number launched each year over the past few decades. SmallSats can meet many research and tech demonstration objectives and are lower-cost options than their large spacecraft counterparts. The number of missions has greatly increased since 1999 thanks to the release of the CubeSat standardized form-factor that allows SmallSats to fit into a standard deployment container and launch as secondary payloads. As the number of SmallSats launched increases, the number of missions that arrive on orbit Dead-on-Arrival or fail to meet their mission objective remains high. For first-time university-developed missions these failure rates are even higher.

The causes of these failures were researched in detail and the majority of the failures point to a lack of development and testing of the spacecraft. Even in the NASA provided document for new missions, flight software receives very minimal attention. The flight software is a critical component of a spacecraft that must be developed for each mission and requires a substantial amount of testing to be reliable. Although the size, weight, power, cost, and development schedules of SmallSats are much less than a typical large satellite mission, the FSW does not scale accordingly. According to mission developers who were queried on the suspected cause of failure on their missions, the majority noted the FSW as the most likely cause.

Due to the nature of SmallSats, being developed rapidly, at a low cost, and typically by a first-time team, the development and testing practices used on larger spacecraft are not possible to be employed. Large missions typically require a test to be run on a full simulation of the system, including all subsystems and dynamics. The research performed looked into simulation products that are available that could help missions perform simulation of the spacecraft quickly and at a low cost. Although there are multiple open-source tools for simulation such as QEMU and 42, and an open-source FSW called cFS, an end-to-end spacecraft simulation did not exist. Over multiple revisions, the NASA Operational Simulator for Small Satellites was designed, implemented, and deployed. NOS^3 combines numerous open-source tools including cFS, 42, and COSMOS into a Virtual Box environment. New software products were written to interface each of the individual components with the NASA developed NOS Engine middleware. The final solution is an easy to deploy, easy to configure, and free software suite that allows a mission to simulate their entire spacecraft to aid in the development and testing of their FSW.

To show the utility of NOS^3 it was applied directly in the development and testing process of the Simulation-to-Flight 1 missions. STF-1 is a NASA IV&V led mission with science and technology demonstration payloads from West Virginia University. Through the implementation of NOS^3 in the mission development cycle, the development of FSW was decoupled from the availability of spacecraft hardware and engineering test units. This allowed the FSW to be written by numerous FSW developers in parallel and tested before attempting to deploy the FSW to the spacecraft hardware. Throughout the mission lifecycle, an additional 6 months of testing time was gained by using NOS^3 for FSW development. STF-1 has been operating without FSW bugs or failures since December of 2018.

There are some limitations to the NOS^3 system that do not fully replace integration testing on spacecraft hardware. Notably, precise timeouts that would be handled by the real-time operating system on the flight computer, cannot be achieved when running on Linux in the simulated environment. For this reason, NOS^3 is not a 100% replacement for a typical development and testing process, but it greatly increases the number of testing resources.

NOS^3 was proven to be a solution for the problem of limited development and testing time available to SmallSats, especially through the results of the STF-1 mission. NOS^3 is available open-source on GitHub at www.github.com/nasa/nos3 and has been used by government, universities, and public and private entities throughout the country. NOS^3 was awarded the prestigious award of runner-up for the 2019 NASA Software of the Year.

5.2 Future Work

NOS^3 is maintained by the JSTAR team at the NASA IV&V Facility. Although NOS^3 was determined to be a solution to the questions proposed by this research, there is still room for improvement. In order to make NOS^3 even simpler to use by first-time missions, sample applications and simulators for all subsystems on a spacecraft could be developed and included. While some missions may prefer to ignore the samples altogether to develop their apps and simulators for the real hardware, many missions and new users have requested more examples. A full simulated spacecraft would give teams an example of how the final NOS^3 modeled spacecraft should look and operate. The list below shows the sample subsystems provided in the current release of NOS^3.

1. *EPS/Battery*
2. *GPS*
3. *Reaction Wheels*
4. *Camera (payload example)*

The list below shows the rest of the component applications and simulators that could be added to model a full SmallSat. Not all these components will be included on every SmallSat.

1. *Radio*
2. *Magnetorquer*
3. *Magnetometer*
4. *Inertial Measurement Unit or Gyro*
5. *Sun Sensor*
6. *Temperature Sensors*

References

[1] NASA, "NASA CubeSats Overview," 14 Feb 2018. [Online]. Available: https://www.nasa.gov/mission_pages/cubesats/overview. [Accessed 2021].

[2] The CubeSat Program, CalPoly SLO, "CubeSat Design Specification Rev. 13," 6 April 2015. [Online]. Available: https://www.cubesat.org/s/cds_rev13_final2.pdf.

[3] G. Norris, "Secondary Payloads Overview," 6 Jan 2014. [Online]. Available: https://www.nasa.gov/sites/default/files/files/Secondary-Payloads-Overview-Rev2.pdf. [Accessed 2021].

[4] D. M. Swartwout, "CubeSat Database," Saint Louis University, 2019. [Online]. Available: https://sites.google.com/a/slu.edu/swartwout/home/cubesat-database. [Accessed 2021].

[5] "Phonesat The Smartphone Nanosatellite," NASA, 20 Nov 2013. [Online]. Available: https://www.nasa.gov/centers/ames/engineering/projects/phonesat.html. [Accessed 2021].

[6] "MarCO (Mars Cube One)," NASA, 11 Feb 2021. [Online]. Available: https://solarsystem.nasa.gov/missions/mars-cube-one/in-depth/. [Accessed 2021].

[7] "Planetary Science Budget Dataset," Planetary Society, 2021. [Online]. Available: https://docs.google.com/spreadsheets/d/12frTU01gfT1CXGWFimN3whf4348F_r3XolTqBt02OyM/edit#gid=244635107.

[8] NASA, "About CubeSat Launch Initiative," 28 May 2020. [Online]. Available: https://www.nasa.gov/content/about-cubesat-launch-initiative. [Accessed 2021].

[9] N. C. L. Initiative, "NASA: CubeSat 101," Oct 2017. [Online]. Available: https://www.nasa.gov/sites/default/files/atoms/files/nasa_csli_cubesat_101_508.pdf. [Accessed 2021].

[10] D. J. F. Miranda, "A Comparative Survey on Flight Software Frameworks for ‘New Space’ Nanosatellite Missions," *Journal of Aerospace Technology and Management*, vol. 11, 2019.

[11] D. Duemler, "Rocket Lab USA poised to change the space industry," BusinessWire, 28 July 2014. [Online]. Available: https://www.businesswire.com/news/home/20140728006338/en/Rocket-Lab-USA-Poised-to-Change-the-Space-Industry. [Accessed 2021].

[12] "ELANA XIX Press Kit," Dec 2018. [Online]. Available: https://www.rocketlabusa.com/assets/Uploads/NASA-ELANA19-Presskit-December2019.pdf. [Accessed 2021].

[13] NanoRacks, "NanoRacks CubeSat Deployer ICD," 4 June 2018. [Online]. Available: https://nanoracks.com/wp-content/uploads/Nanoracks-CubeSat-Deployer-NRCSD-IDD.pdf. [Accessed 2021].

[14] C. Wicks, "BeagleBoard," 19 Aug 2019. [Online]. Available: https://beagleboard.org/blog/2019-08-19-oresat-with-pocketbeagle-selfie-stick-from-space. [Accessed 2021].

[15] D. M. Swartwout, "University-Class Spacecraft by the Numbers," in *Small Satellite Conference*, Logan, Utah, 2016.

[16] M. Langer, "Reliability of CubeSats – Statistical Data, Developers' Beliefs and the Way Forward," in *Small Satellite Confernce*, Logan, Utah, 2016.

[17] WindRiver, "WindRiver SIMICS Product Overview," Nov 2019. [Online]. Available: https://windriver.com/themes/Windriver/pdf/Wind-River-Simics_Product-Overview.pdf. [Accessed 2021].

[18] WindRiver, "NASA Meets Satellite Project Testing and Verification," April 2020. [Online]. Available: https://www.windriver.com/themes/Windriver/pdf/NASA_IVV_SS_0113.pdf. [Accessed 2021].

[19] R. Goering, "In Hindsight SW debug can run backward," EDN, 7 Mar 2005. [Online]. Available: https://www.edn.com/in-hindsight-sw-debug-can-run-backward/. [Accessed 2021].

[20] C. G. AB, "LEON3 Processor," 2021. [Online]. Available: https://www.gaisler.com/index.php/products/processors/leon3. [Accessed 2021].

[21] RTEMS, "LEON3/4 BSP," 9 Nov 2015. [Online]. Available: https://devel.rtems.org/wiki/TBR/BSP/Leon3. [Accessed 2021].

[22] QEMU, "QEMU Developers Guide," 2020. [Online]. Available: https://qemu.readthedocs.io/en/latest/. [Accessed 2021].

[23] NASA, "cFS Introduction," 10 July 2020. [Online]. Available: https://cfs.gsfc.nasa.gov/Introduction.html. [Accessed 2021].

[24] E. Stoneking, "42 Spacecraft Simulation," 30 Oct 2020. [Online]. Available: https://www.github.com/ericstoneking/42. [Accessed 2021].

[25] D. Wheeler, "SLOCCount," 2021. [Online]. Available: https://dwheeler.com/sloccount/. [Accessed 2021].

[26] R. Madachy, "Software Cost Tools COCOMO II," 2021. [Online]. Available: http://softwarecost.org/tools/COCOMO/. [Accessed 2021].

Appendix A. NOS^3 Hardware Library

```c
/* Copyright (C) 2009 - 2016 National Aeronautics and Space Administration. All Foreign Rights are Reserved to the U.S. Government.

This software is provided "as is" without any warranty of any, kind either express, implied, or statutory, including, but not
limited to, any warranty that the software will conform to, specifications any implied warranties of merchantability, fitness
for a particular purpose, and freedom from infringement, and any warranty that the documentation will conform to the program, or
any warranty that the software will be error free.

In no event shall NASA be liable for any damages, including, but not limited to direct, indirect, special or consequential damages,
arising out of, resulting from, or in any way connected with the software or its documentation.  Whether or not based upon warranty,
contract, tort or otherwise, and whether or not loss was sustained from, or arose out of the results of, or use of, the software,
documentation or services provided hereunder

ITC Team
NASA IV&V
ivv-itc@lists.nasa.gov
*/

#include "nos_link.h"
#include <stdint.h>
#include <stdlib.h>

/* psp */
#include <cfe_psp.h>

/* osal */
#include <osapi.h>

/* nos */
#include <Can/Client/CInterface.h>

#include "libcan.h"

/* can device handles */
static NE_CanHandle *can_device[NUM_CAN_DEVICES] = {0};

/* can mutex */
static uint32 nos_can_mutex = 0;

/* public prototypes */
void nos_init_can_link(void);
void nos_destroy_can_link(void);

/* get spi device */
static NE_CanHandle* nos_get_can_device(can_info_t* device)
{
    NE_CanHandle *dev = NULL;
    if(!strcmp(device->handle, CW_CAN_HANDLE_STR))
    {
        dev = can_device[CW_CAN_HANDLE];
        if(dev == NULL)
        {
            can_init_dev(device);
            dev = can_device[CW_CAN_HANDLE];
        }
    }
    else
    {
        dev = can_device[1];
        if(dev == NULL)
        {
            can_init_dev(device);
            dev = can_device[1];
        }
    }

    return dev;
}

/* initialize nos engine can link */
void nos_init_can_link(void)
{
    /* create mutex */
    int32 result = OS_MutSemCreate(&nos_can_mutex, "nos_can", 0);

}

/* destroy nos engine can link */
void nos_destroy_can_link(void)
{
    OS_MutSemTake(nos_can_mutex);

    /* clean up can buses */
    int i;
    for (i = 0; i < NUM_CAN_DEVICES; i++)
    {
        NE_CanHandle *dev = can_device[i];
        if (dev)
            NE_can_close(&dev);
    }

    OS_MutSemGive(nos_can_mutex);

    /* destroy mutex */
    int32 result = OS_MutSemDelete(nos_can_mutex);
}

// Bring CAN network interface
int32_t can_init_dev(can_info_t* device)
{
    int32 result = OS_SUCCESS;
    NE_CanHandle **dev;
    const nos_connection_t *con;

    if(!strcmp(device->handle, CW_CAN_HANDLE_STR))
    {

        dev = &can_device[CW_CAN_HANDLE];
        if (*dev == NULL)
        {
            /* get nos can connection params */
            con = &nos_can_connection[CW_CAN_HANDLE];
        }
    }
    else
    {
        // TODO - UPDATE with mission defined device strings
        dev = &can_device[1];
        if (*dev == NULL)
        {
            /* get nos can connection params */
            con = &nos_can_connection[1];
        }
    }

    /* try to initialize master */
    *dev = NE_can_init_master3(hub, 10, con->uri, con->bus);
    if (*dev == NULL)
    {
        result = OS_ERROR;
        OS_printf("LIBCAN: %s:  FAILED TO INITIALIZE NOS CAN MASTER\n", __FUNCTION__)
;
    }

    OS_MutSemGive(nos_can_mutex);
    return result;
}

// TODO: NOT IMPLEMENTED!
int32_t can_set_modes(can_info_t* device)
{
    return CAN_SUCCESS;
}

// Write out to CAN bus from CAN device specified by `device`.
int32_t can_write(can_info_t* device, uint32_t can_id, uint8_t* buf, const uint32_t l
ength)
{
    return can_master_transaction(CW_CAN_HANDLE, can_id, buf, length, NULL, 0, 0, 0);
}

// Read a can_frame from CAN interface specified by `device-
>handle`. Does a blocking read call.
int32_t can_blocking_read(can_info_t* device, struct can_frame* readFrame, const uint
32_t length)
{
    return can_master_transaction(CW_CAN_HANDLE, readFrame-
>can_id << 3, NULL, 0, &(readFrame->data[0]), length, 0, 0);
}

// Read a can_frame from CAN interface specified by `device-
>handle`. Does a nonblocking read call.
int32_t can_nonblocking_read(can_info_t* device, struct can_frame* readFrame, const u
int32_t length, uint32_t second_timeout, uint32_t microsecond_timeout)
{
    return can_master_transaction(CW_CAN_HANDLE, readFrame-
>can_id << 3, NULL, 0, &(readFrame->data[0]), length, 0, 0);
}

//int32 can_master_transaction(int handle, uint32_t identifier, void * txbuf, uint8_t
 txlen, void * rxbuf, uint8_t rxlen, uint16_t timeout)
int32_t can_master_transaction(can_info_t* device, uint32_t can_id, uint8_t* txbuf, c
onst uint32_t txlen, uint8_t* rxbuf, const uint32_t rxlen, uint32_t second_timeout, u
int32_t microsecond_timeout)
{
    int result = OS_ERROR;

    /* get can device handle */
    NE_CanHandle *dev = nos_get_can_device(device);

    /* can transaction */
    OS_MutSemTake(nos_can_mutex);
    if(dev)
    {
        if ( (can_id & 0xF) == CW_WHL1_MASK || (can_id & 0xF) == CW_WHL2_MASK || (can
_id & 0xF) == CW_WHL3_MASK )
        {
            result = NE_can_transaction(dev, CW_ADDRESS, txbuf, txlen, rxbuf, rxlen);

        }

        else if (can_id == CW_ADDRESS)
        {
            result = NE_can_transaction(dev, CW_ADDRESS, txbuf, txlen, rxbuf, rxlen);

        }

        else
        {
            //OS_printf("LIBCAN: %s:  CAN IDENTIFIER IS NOT CW_ADDRESS, NOR CONTAINS
THE WHEEL CAN MASKS\n", __FUNCTION__);
            result = NE_can_transaction(dev, can_id, txbuf, txlen, rxbuf, rxlen);
        }
    }

    OS_MutSemGive(nos_can_mutex);

    return result;
}

// Bring CAN network interface down
```

```c
int32_t can_close_device(can_info_t* device)
{
    OS_MutSemTake(nos_can_mutex);

    /* clean up can device */
    NE_CanHandle *dev = can_device[CW_CAN_HANDLE];
    if(dev) NE_can_close(&dev);

    OS_MutSemGive(nos_can_mutex);

    /* destroy mutex */
    return NE_CAN_SUCCESS;
}
```

```c
/* Copyright (C) 2009 - 2016 National Aeronautics and Space Administration. All Foreign Rights are Reserved to the U.S. Government.

This software is provided "as is" without any warranty of any, kind either express, implied, or statutory, including, but not
limited to, any warranty that the software will conform to, specifications any implied warranties of merchantability, fitness
for a particular purpose, and freedom from infringement, and any warranty that the documentation will conform to the program, or
any warranty that the software will be error free.

In no event shall NASA be liable for any damages, including, but not limited to direct, indirect, special or consequential damages,
arising out of, resulting from, or in any way connected with the software or its documentation.  Whether or not based upon warranty,
contract, tort or otherwise, and whether or not loss was sustained from, or arose out of the results of, or use of, the software,
documentation or services provided hereunder

ITC Team
NASA IV&V
ivv-itc@lists.nasa.gov
*/

#include "nos_link.h"
#include <stdint.h>
#include <stdlib.h>

/* psp */
#include <cfe_psp.h>

/* osal */
#include <osapi.h>

/* nos */
#include <I2C/Client/CInterface.h>

/* hwlib API */
#include "libi2c.h"
```

```
/* i2c device handles */
static NE_I2CHandle *i2c_device[NUM_I2C_DEVICES] = {0};

/* i2c mutex */
static uint32 nos_i2c_mutex = 0;

/* public prototypes */
void nos_init_i2c_link(void);
void nos_destroy_i2c_link(void);

/* initialize nos engine i2c link */
void nos_init_i2c_link(void)
{
    /* create mutex */
    int32 result = OS_MutSemCreate(&nos_i2c_mutex, "nos_i2c", 0);

}

/* destroy nos engine i2c link */
void nos_destroy_i2c_link(void)
{
    OS_MutSemTake(nos_i2c_mutex);

    /* clean up i2c buses */
    int32_t i;
    for(i = 0; i < NUM_I2C_DEVICES; i++)
    {
        NE_I2CHandle *dev = i2c_device[i];
        if(dev) NE_i2c_close(&dev);
    }

    OS_MutSemGive(nos_i2c_mutex);

    /* destroy mutex */
    int32 result = OS_MutSemDelete(nos_i2c_mutex);
}

/* nos i2c transaction */
int32_t i2c_master_transaction(int32_t handle, uint8_t addr, void * txbuf, uint8_t tx
len,
                               void * rxbuf, uint8_t rxlen, uint16_t timeout)
{
    int32_t result = OS_ERROR;

    if(handle < NUM_I2C_DEVICES)
    {
        OS_MutSemTake(nos_i2c_mutex);

        /* get i2c device handle */
        NE_I2CHandle **dev = &i2c_device[handle];
        if(*dev == NULL)
        {
            /* get nos i2c connection params */
            const nos_connection_t *con = &nos_i2c_connection[handle];
            /* try to initialize master */
            *dev = NE_i2c_init_master3(hub, 10, con->uri, con->bus);
            if(*dev == NULL)
            {
                OS_printf("nos i2c_init_master failed\n");
            }
        }

        /* i2c transaction */
        if(*dev)
        {
            if(NE_i2c_transaction(*dev, addr, txbuf, txlen, rxbuf, rxlen) == NE_I2C_S
UCCESS)
            {
                result = OS_SUCCESS;
            }
        }

        OS_MutSemGive(nos_i2c_mutex);
    }

    return result;
}
```

```
/* Copyright (C) 2009 - 2016 National Aeronautics and Space Administration. All Forei
gn Rights are Reserved to the U.S. Government.

This software is provided "as is" without any warranty of any, kind either express, i
mplied, or statutory, including, but not
limited to, any warranty that the software will conform to, specifications any implie
d warranties of merchantability, fitness
for a particular purpose, and freedom from infringement, and any warranty that the do
cumentation will conform to the program, or
any warranty that the software will be error free.

In no event shall NASA be liable for any damages, including, but not limited to direc
t, indirect, special or consequential damages,
arising out of, resulting from, or in any way connected with the software or its docu
mentation.  Whether or not based upon warranty,
contract, tort or otherwise, and whether or not loss was sustained from, or arose out
 of the results of, or use of, the software,
documentation or services provided hereunder

ITC Team
NASA IV&V
ivv-itc@lists.nasa.gov
*/

#include "nos_link.h"
#include <stdint.h>
#include <stdlib.h>

/* psp */
```

```
#include <cfe_psp.h>

/* nos */
#include <Spi/Client/CInterface.h>

/* hwlib API */
#include "libspi.h"

/* spi bus mutex */
spi_mutex_t spi_bus_mutex[MAX_SPI_BUSES];
uint32_t handle_count = 0;

/* spi device handles */
static NE_SpiHandle *spi_device[NUM_SPI_DEVICES] = {0};

/* public prototypes */
void nos_init_spi_link(void);
void nos_destroy_spi_link(void);

/* private prototypes */
static NE_SpiHandle* nos_get_spi_device(spi_info_t* device);

/* initialize nos engine spi link */
void nos_init_spi_link(void)
{
    // Do nothing
}

/* destroy nos engine spi link */
void nos_destroy_spi_link(void)
{
    /* clean up spi buses */
    int i;
    for(i = 0; i < NUM_SPI_DEVICES; i++)
    {
        NE_SpiHandle *dev = spi_device[i];
        if(dev) NE_spi_close(&dev);
    }
}

/* nos spi init */
int32 spi_init_dev(spi_info_t* device)
{
    int     status = SPI_SUCCESS;
    char    buffer[16];

    // Initialize the bus mutex
    if (device->bus < MAX_SPI_BUSES)
    {
        if (spi_bus_mutex[device->bus].users == 0)
        {
            snprintf(buffer, 16, "spi_%d_mutex", device->bus);
            status = OS_MutSemCreate(&spi_bus_mutex[device-
>bus].spi_mutex, buffer, 0);
            if (status != OS_SUCCESS)
            {
                CFE_EVS_SendEvent(SPI_ERR_MUTEX_CREATE, CFE_EVS_ERROR, "HWLIB: Create
 spi mutex error %d", status);
                return status;
            }
        }
        spi_bus_mutex[device->bus].users++;
    }
    else
    {
        CFE_EVS_SendEvent(SPI_ERR_MUTEX_CREATE, CFE_EVS_ERROR, "HWLIB: Create spi mut
ex error %d, bus invalid!", status);
        return status;
    }

    if (OS_MutSemTake(spi_bus_mutex[device->bus].spi_mutex) == OS_SUCCESS)
    {
        /* get spi device handle */
        NE_SpiHandle **dev = &spi_device[device->handle];
        if(*dev == NULL)
        {
            /* get nos spi connection params */
            const nos_connection_t *con = &nos_spi_connection[(device-
>bus * 10) + device->cs];

            /* try to initialize master */
            *dev = NE_spi_init_master3(hub, con->uri, con->bus);
            if(*dev)
            {
                status = SPI_SUCCESS;
            }
            else
            {
                OS_MutSemGive(spi_bus_mutex[device->bus].spi_mutex);
                CFE_EVS_SendEvent(SPI_ERR_FILE_OPEN, CFE_EVS_ERROR, "HWLIB: Open SPI
device \"%s\" error %d", device->deviceString, status);
                return status;
            }
        }
    }
    OS_MutSemGive(spi_bus_mutex[device->bus].spi_mutex);

    // Set open flag
    device->isOpen = SPI_DEVICE_OPEN;

    return status;
}

/* get spi device */
static NE_SpiHandle* nos_get_spi_device(spi_info_t* device)
{
    NE_SpiHandle *dev = NULL;
    if(device->handle < NUM_SPI_DEVICES)
    {
        dev = spi_device[device->handle];
        if(dev == NULL)
        {
            spi_init_dev(device);
            dev = spi_device[device->handle];
        }
    }
    return dev;
}

/* nos spi chip select */
int32 spi_select_chip(spi_info_t* device)
{
    uint32_t status = SPI_SUCCESS;

    status = OS_MutSemTake(spi_bus_mutex[device->bus].spi_mutex);

    NE_SpiHandle *dev = nos_get_spi_device(device);
    if(dev)
    {
        NE_spi_select_chip(dev, device->cs);
    }

    return status;
}

/* nos spi chip unselect */
int32 spi_unselect_chip(spi_info_t* device)
{
    uint32_t status = SPI_SUCCESS;

    status = OS_MutSemGive(spi_bus_mutex[device->bus].spi_mutex);

    NE_SpiHandle *dev = nos_get_spi_device(device);
    if(dev)
    {
        NE_spi_unselect_chip(dev);
    }

    return status;
}

/* nos spi write */
int32 spi_write(spi_info_t* device, uint8 data[], const uint32 numBytes)
{
    int status = SPI_SUCCESS;

    NE_SpiHandle *dev = nos_get_spi_device(device);
    if(dev)
    {
        if(NE_spi_write(dev, data, numBytes) != NE_SPI_SUCCESS)
        {
            status = SPI_ERROR;
        }
    }
    return status;
}

/* nos spi read */
int32 spi_read(spi_info_t* device, uint8 data[], const uint32 numBytes)
{
    int status = SPI_SUCCESS;

    NE_SpiHandle *dev = nos_get_spi_device(device);
    if(dev)
    {
        if(NE_spi_read(dev, data, numBytes) != NE_SPI_SUCCESS)
        {
            status = SPI_ERROR;
        }
    }

    return status;
}

int32 spi_transaction(spi_info_t* device, uint8_t *txBuff, uint8_t * rxBuffer, uint32
_t length, uint16_t delay, uint8_t bits, uint8_t deselect)
{
    int status = SPI_SUCCESS;

    NE_SpiHandle *dev = nos_get_spi_device(device);
    if(dev)
    {
        if(NE_spi_transaction(dev, txBuff, length, rxBuffer, length) != NE_SPI_SUCCES
S)
        {
            status = SPI_ERROR;
        }
    }

    return status;
}
```

```
/* Copyright (C) 2009 - 2016 National Aeronautics and Space Administration. All Forei
gn Rights are Reserved to the U.S. Government.

This software is provided "as is" without any warranty of any, kind either express, i
mplied, or statutory, including, but not
limited to, any warranty that the software will conform to, specifications any implie
d warranties of merchantability, fitness
for a particular purpose, and freedom from infringement, and any warranty that the do
cumentation will conform to the program, or
any warranty that the software will be error free.

In no event shall NASA be liable for any damages, including, but not limited to direc
t, indirect, special or consequential damages,
arising out of, resulting from, or in any way connected with the software or its docu
mentation.  Whether or not based upon warranty,
contract, tort or otherwise, and whether or not loss was sustained from, or arose out
 of the results of, or use of, the software,
documentation or services provided hereunder

ITC Team
NASA IV&V
ivv-itc@lists.nasa.gov
*/

#include "nos_link.h"
#include <stdint.h>
#include <stdlib.h>

/* psp */
#include <cfe_psp.h>

/* osal */
#include <osapi.h>

/* nos */
#include <Uart/Client/CInterface.h>

/* hwlib API */
#include "libuart.h"

/* size of uart buffer */
#define USART_RX_BUF_SIZE    512

/* usart device handles */
static NE_Uart *usart_device[NUM_USARTS] = {0};

/* usart mutex */
static uint32 nos_usart_mutex = 0;

/* public prototypes */
void nos_init_usart_link(void);
void nos_destroy_usart_link(void);

/* private prototypes */
static NE_Uart* nos_get_usart_device(int handle);

/* initialize nos engine usart link */
void nos_init_usart_link(void)
{
    /* create mutex */
    int32 result = OS_MutSemCreate(&nos_usart_mutex, "nos_usart", 0);

}

/* destroy nos engine usart link */
void nos_destroy_usart_link(void)
{
    int i;

    OS_MutSemTake(nos_usart_mutex);
    /* clean up usart buses */

    for(i = 0; i <= NUM_USARTS; i++)
    {
        NE_Uart *dev = usart_device[i];
        if(dev) NE_uart_close(&dev);
    }

    OS_MutSemGive(nos_usart_mutex);

    /* destroy mutex */
    int32 result = OS_MutSemDelete(nos_usart_mutex);
}

/* init usart */
int32 uart_init_port(uart_info_t* device)
{
    int32_t status = OS_SUCCESS;
    if(device->handle >= 0)
    {
        OS_MutSemTake(nos_usart_mutex);

        /* get usart device handle */
        NE_Uart **dev = &usart_device[device->handle];
        if(*dev == NULL)
        {
            /* get nos usart connection params */
            const nos_connection_t *con = &nos_usart_connection[device->handle];

            /* try to initialize usart */
            *dev = NE_uart_open3(hub, "fsw", con->uri, con->bus, device->handle);

            if(*dev)
            {
                /* set default queue size */
                NE_uart_set_queue_size(*dev, USART_RX_BUF_SIZE);

                device->isOpen = PORT_OPEN;
            }
            else
            {
                OS_printf("nos uart_open failed\n");
                device->isOpen = PORT_CLOSED;
                status = OS_ERR_FILE;
            }
        }
        OS_MutSemGive(nos_usart_mutex);
    }
    else
    {
        OS_printf("Handle not found\n");
        device->isOpen = PORT_CLOSED;
        status = OS_ERR_FILE;
    }
    return status;
}

/* get usart device */
static NE_Uart* nos_get_usart_device(int handle)
{
    NE_Uart *dev = NULL;
    if(handle < NUM_USARTS)
    {
        dev = usart_device[handle];
    }
    return dev;
}

/* usart write */
int32 uart_write_port(int32 handle, uint8 data[], const uint32 numBytes)
{
    int32_t status = OS_ERR_FILE;
    NE_Uart *dev = nos_get_usart_device((int)handle);
    if(dev)
    {
        OS_MutSemTake(nos_usart_mutex);
        status = NE_uart_write(dev, (const uint8_t*)data, numBytes); //Can this funct
ion return -1?
        OS_MutSemGive(nos_usart_mutex);
    }
    return status;
}

/* usart read */
int32 uart_read_port(int32 handle, uint8 data[], const uint32 numBytes)
{
    uint32 status = OS_ERR_FILE;

    if (data != NULL) //Check that there is actually data to read
    {
        char c = 0xFF;
        int  i;
        int stat;
        NE_Uart *dev = nos_get_usart_device((int)handle);
        if(dev)
        {
            OS_MutSemTake(nos_usart_mutex);
            for (i = 0; i < (int)numBytes; i++) //TODO: Add ability to switch between
 blocking and non-blocking?
            {
                /*
                //NON BLOCKING MODE
                stat = NE_uart_getc(dev, (uint8_t*)&c); //Returns 0 if byte read, 1 i
f no byte actually read
                if(stat == 1)
                {
                    return i; //Causes app to immediately enter service mode
                }
                else {
                    data[i] = c;
                }
                */
                //BLOCKING MODE
                do {
                    stat = NE_uart_getc(dev, (uint8_t*)&c);
                } while(stat);
                data[i] = c;
            }
            OS_MutSemGive(nos_usart_mutex);
            status = numBytes;

            return status;
        }

        return status; //There is data, but can't read from device
    }

    return status; //Following arm_inux model
}

/* usart number bytes available */
int32 uart_bytes_available(int32 handle)
{
    int bytes = 0;
    NE_Uart *dev = nos_get_usart_device((int)handle);
    if(dev)
    {
        OS_MutSemTake(nos_usart_mutex);
        bytes = (int)NE_uart_available(dev);
        OS_MutSemGive(nos_usart_mutex);
    }
    return bytes;
}

int32 uart_close_port(int32 handle)
{
    NE_UartStatus status;
    NE_Uart *dev = nos_get_usart_device((int)handle);
    if (handle >= 0)
    {
        status = NE_uart_close(&dev);
    }
    if (status == NE_UART_SUCCESS) {
        return OS_SUCCESS;
    }
    else
    {
        return OS_ERROR;
    }

}
```

Appendix B: NOS^3 Sample Simulator

```
#include <ItcLogger/Logger.hpp>
#include <sim_config.hpp>

namespace Nos3
{
    ItcLogger::Logger *sim_logger;
}

int
main(int argc, char *argv[])
{
    std::string simulator_name = "sample_sim"; // this is the ONLY simulator specific
 line!

    // Determine the configuration and run the simulator
    Nos3::SimConfig sc(argc, argv);
    Nos3::sim_logger->info("main:  %s simulator starting",
        simulator_name.c_str());
    sc.run_simulator(simulator_name);
    Nos3::sim_logger->info("main:  %s simulator terminating",
        simulator_name.c_str());
}
```

```
#include <sample_hardware_model.hpp>

namespace Nos3
{
    REGISTER_HARDWARE_MODEL(SampleHardwareModel,"SAMPLE");

    extern ItcLogger::Logger *sim_logger;

    SampleHardwareModel::SampleHardwareModel(const boost::property_tree::ptree& confi
g) : SimIHardwareModel(config), _keep_running(true)
    {
        // The sim logger prints to both the terminal and to a file
        sim_logger-
>trace("SampleHardwareModel::SampleHardwareModel:  Constructor executing");

        // Initialize configuration values to hard coded defaults in case configurati
on file is incomplete
        std::string time_bus_name = "command";
        std::string bus_name = "usart_29";
        int node_port = 29;
        _counter = 0;
        _init_time_seconds = 5.0;
        _millisecond_stream_delay = 1000;
        _second_stream_delay = 1.0;

        // Get the time connection string
        std::string connection_string = config.get("common.nos-connection-string", "tcp://127.0.0.1:12001");

        // Loop through the configuration file
        if (config.get_child_optional("hardware-model.connections"))
        {
            BOOST_FOREACH(const boost::property_tree::ptree::value_type &v, config.get_child("hardware-model.connections"))
            {
                // Find the time information
                if (v.second.get("type", "").compare("time") == 0)
                {
                    time_bus_name = v.second.get("bus-name", "command");
                    sim_logger->info("SampleHardwareModel::SampleHardwareModel:  Found time info!");
                }
            }
        }

        // Loop through the configuration file
        if (config.get_child_optional("simulator.hardware-model.connections"))
        {
            BOOST_FOREACH(const boost::property_tree::ptree::value_type &v, config.get_child("simulator.hardware-model.connections"))
            {
                // Find the uart information
                if (v.second.get("type", "").compare("usart") == 0)
                {
                    bus_name = v.second.get("bus-name", bus_name);
                    node_port = v.second.get("node-port", node_port);
                    sim_logger->info("SampleHardwareModel::SampleHardwareModel:  Found uart info!");
                }

                // Find the initialization time information
                if (v.second.get("type", "").compare("period") == 0)
                {
                    _init_time_seconds = v.second.get("init-time-seconds", _init_time_seconds);
                    _millisecond_stream_delay = v.second.get("ms-period", _millisecond_stream_delay);
                    _second_stream_delay = double(_millisecond_stream_delay) / 1000;
                    sim_logger->info("SampleHardwareModel::SampleHardwareModel:  Found period info!");
                }
            }
        }

        // Reset time connection
        _time_bus.reset(new NosEngine::Client::Bus(_hub, connection_string, time_bus_name));

        // Reset and open uart
        _uart_connection.reset(new NosEngine::Uart::Uart(_hub, config.get("simulator.name", "sample_sim"), connection_string, bus_name));
        _uart_connection->open(node_port);
        _uart_connection-
>set_read_callback(std::bind(&SampleHardwareModel::uart_read_callback, this, std::pla
ceholders::_1, std::placeholders::_2));

        // Setup the data provider
        std::string dp_name = config.get("simulator.hardware-model.data-
provider.type", "SAMPLE_PROVIDER");
        _sample_dp = SimDataProviderFactory::Instance().Create(dp_name, config);

        // Calculate next time to send streaming data
        _next_time = _absolute_start_time + _init_time_seconds;

        // Prepare streaming data header - 0xDEAD
        _streaming_data.push_back(0xDE);
        _streaming_data.push_back(0xAD);
        // Prepare streaming data counter
        _streaming_data.push_back(0x00);
        _streaming_data.push_back(0x00);
        _streaming_data.push_back(0x00);
        _streaming_data.push_back(0x00);
        // Prepare streaming data payload
        _streaming_data.push_back(0x00);
        _streaming_data.push_back(0x00);
        _streaming_data.push_back(0x00);
        _streaming_data.push_back(0x00);
        // Prepare streaming data trailer - 0xBEEF
        _streaming_data.push_back(0xBE);
        _streaming_data.push_back(0xEF);

        // Add callback for streaming data
        _time_bus-
>add_time_tick_callback(std::bind(&SampleHardwareModel::send_periodic_data, this, std
::placeholders::_1));

        sim_logger-
>trace("SampleHardwareModel::SampleHardwareModel:  Time node, UART node, data provide
r created; constructor exiting");
    }

    SampleHardwareModel::~SampleHardwareModel(void)
    {
        sim_logger-
>trace("SampleHardwareModel::SampleHardwareModel:  Destructor executing");

        // Clean up the data provider we got
        delete _sample_dp;

        // Reset the time bus so the unique pointer does not try to delete the hub
        _time_bus.reset();
        //Do not destroy the time node, the bus will do it

        // Close the uart
        _uart_connection->close();
    }
    void SampleHardwareModel::run(void)
    {
        int i = 0;
        while(_keep_running)
        {
            sim_logger-
>info("SampleHardwareModel::run:  Loop count %d, time %f", i++,
                _absolute_start_time + (double(_time_bus-
>get_time() * _sim_microseconds_per_tick)) / 1000000.0);
            sleep(5);
        }
    }

    void SampleHardwareModel::command_callback(NosEngine::Common::Message msg)
    {
        // Here's how to get the data out of the message
        NosEngine::Common::DataBufferOverlay dbf(const_cast<NosEngine::Utility::Buffe
r&>(msg.buffer));
        sim_logger-
>info("SampleHardwareModel::command_callback:  Received command: %s.", dbf.data);

        // Do something with the data
        std::string command = dbf.data;
        std::string response = "SampleHardwareModel::command_callback:  INVALID COMMA
ND! (Try STOP SAMPLE)";
        boost::to_upper(command);
        if (command.compare("STOP SAMPLE") == 0)
        {
            _keep_running = false;
            response = "SampleHardwareModel::command_callback:  STOPPING SAMPLE";
        }

        // Here's how to send a reply
        _command_node-
>send_reply_message_async(msg, response.size(), response.c_str());
    }

    void SampleHardwareModel::send_periodic_data(NosEngine::Common::SimTime time)
    {
        // Determine current simulator time
        double sim_time = _absolute_start_time + (double(time * _sim_microseconds_per
_tick)) / 1000000.0;

        // Check if time to send data
        if (_next_time < sim_time)
        {
            sim_logger-
>trace("SampleHardwareModel::send_periodic_data:  Time to send more data!");

            // Get a new data point
            const boost::shared_ptr<SampleDataPoint> data_point = boost::dynamic_poin
ter_cast<SampleDataPoint>(_sample_dp->get_data_point());

            // Call streaming data function
            stream_data(*data_point, _streaming_data);

            // Determine next time
            _next_time = _next_time + _second_stream_delay;
        }
    }

    void SampleHardwareModel::stream_data(const SampleDataPoint& data_point, std::vec
tor<uint8_t>& out_data)
    {
        // Update Payload - Counter
        _counter++;
        out_data[2] = (_counter >> 24) & 0x000000FF;
        out_data[3] = (_counter >> 16) & 0x000000FF;
        out_data[4] = (_counter >>  8) & 0x000000FF;
        out_data[5] = _counter & 0x000000FF;
        // Update Payload - Data
        float payload = static_cast<float>(data_point.get_sample_data());
        uint32_t* value = (uint32_t*) &payload;
        out_data[6] = value[3];
        out_data[7] = value[2];
        out_data[8] = value[1];
        out_data[9] = value[0];

        // Log reply data in man readable format and ship the message bytes off
        sim_logger->debug("SampleHardwareModel::stream_data:  %s",
            SimIHardwareModel::uint8_vector_to_hex_string(out_data).c_str());
        _uart_connection->write(&out_data[0], out_data.size());
    }

    void SampleHardwareModel::uart_read_callback(const uint8_t *buf, size_t len)
    {
        // Retrieve data and log received data in man readable format
        std::vector<uint8_t> in_data(buf, buf + len);
        sim_logger->debug("SampleHardwareModel::uart_read_callback:  REQUEST %s",
            SimIHardwareModel::uint8_vector_to_hex_string(in_data).c_str());

        // Check if message is incorrect size
        if (in_data.size() != 9)
        {
            sim_logger-
>debug("SampleHardwareModel::uart_read_callback:  Invalid command size of %d received
!", in_data.size());
            return;
        }

        // Check header - 0xDEAD
        if ((in_data[0] != 0xDE) || (in_data[1] !=0xAD))
        {
            sim_logger-
>debug("SampleHardwareModel::uart_read_callback:  Header incorrect!");
            return;
        }

        // Check trailer - 0xBEEF
        if ((in_data[7] != 0xBE) || (in_data[8] !=0xEF))
        {
            sim_logger-
>debug("SampleHardwareModel::uart_read_callback:  Trailer incorrect!");
            return;
        }

        // Process command type
        switch (in_data[2])
        {
            case 1:
                sim_logger-
>debug("SampleHardwareModel::uart_read_callback:  Configuration command received!");
                _millisecond_stream_delay = (in_data[3] << 24) +
                                            (in_data[4] << 16) +
                                            (in_data[5] << 8 ) +
                                            (in_data[6]);
                sim_logger-
>debug("SampleHardwareModel::uart_read_callback:  New millisecond stream delay of %d"
, _millisecond_stream_delay);
                _second_stream_delay = double(_millisecond_stream_delay) / 1000;
                break;

            case 2:
                sim_logger-
>debug("SampleHardwareModel::uart_read_callback:  Other command received!");
                break;

            default:
                sim_logger-
>debug("SampleHardwareModel::uart_read_callback:  Unused command received!");
                break;
        }

        // Prepare to echo back valid command
        std::vector<uint8_t> out_data = in_data;

        // Log reply data in man readable format and ship the message bytes off
        sim_logger->debug("SampleHardwareModel::uart_read_callback:  REPLY %s",
            SimIHardwareModel::uint8_vector_to_hex_string(out_data).c_str());
        _uart_connection->write(&out_data[0], out_data.size());
    }
}
```

```
#include <sample_42_data_provider.hpp>

namespace Nos3
{
    REGISTER_DATA_PROVIDER(Sample42DataProvider,"SAMPLE_42_PROVIDER");

    extern ItcLogger::Logger *sim_logger;
    Sample42DataProvider::Sample42DataProvider(const boost::property_tree::ptree& con
fig) : SimData42SocketProvider(config)
    {
        sim_logger-
>trace("Sample42DataProvider::Sample42DataProvider:  Constructor executed");

        connect_reader_thread_as_42_socket_client(
            config.get("simulator.hardware-model.data-
provider.hostname", "localhost"),
            config.get("simulator.hardware-model.data-provider.port", 4242) );

        _sc = config.get("simulator.hardware-model.data-provider.spacecraft", 0);
    }

    Sample42DataProvider::~Sample42DataProvider(void)
    {
        sim_logger-
>trace("Sample42DataProvider::~Sample42DataProvider:  Destructor executed");
    }

    boost::shared_ptr<SimIDataPoint> Sample42DataProvider::get_data_point(void) const
    {
        sim_logger->trace("Sample42DataProvider::get_data_point:  Executed");

        // Get the 42 data
        const boost::shared_ptr<Sim42DataPoint> dp42 =
            boost::dynamic_pointer_cast<Sim42DataPoint>(SimData42SocketProvider::get_
data_point());

        // Prepare the specific data
        SimIDataPoint *dp = new SampleDataPoint(_sc, dp42);

        return boost::shared_ptr<SimIDataPoint>(dp);
    }
}
```

```
#include <sample_data_point.hpp>

namespace Nos3
{
    extern ItcLogger::Logger *sim_logger;

    SampleDataPoint::SampleDataPoint(void)
    {
        sim_logger-
>trace("SampleDataPoint::SampleDataPoint:  Empty constructor executed");
    }

    SampleDataPoint::SampleDataPoint(double data)
    {
        sim_logger-
>trace("SampleDataPoint::SampleDataPoint:  Defined constructor executed");
```

```
        // Option to do calculations on provided data at this point
        _sample_data.push_back(data * 2);
    }

    SampleDataPoint::SampleDataPoint(int16_t spacecraft, const boost::shared_ptr<Sim4
2DataPoint> dp)
    {
        sim_logger-
>trace("SampleDataPoint::SampleDataPoint:  42 Constructor executed");

        // Declare 42 telemetry string prefix
        // 42 variables defined in `42/Include/42types.h`
        // 42 data stream defined in `42/Source/IPC/SimWriteToSocket.c`
        std::ostringstream MatchString;
        MatchString << "SC[" << spacecraft << "].svb = ";
        size_t MSsize = MatchString.str().size();

        // Parse 42 telemetry
        std::vector<std::string> lines = dp->get_lines();
        try
        {
            for (int i = 0; i < lines.size(); i++)
            {
                // Compare prefix
                if (lines[i].compare(0, MSsize, MatchString.str()) == 0)
                {
                    size_t found = lines[i].find_first_of("=");
                    // Parse line
                    std::istringstream iss(lines[i].substr(found+1, lines[i].size()-
found-1));
                    _sample_data.clear();
                    for (std::string s; iss >> s; )
                    {
                        _sample_data.push_back(std::stod(s));
                    }
                    sim_logger-
>trace("SampleDataPoint::SampleDataPoint:  Parsed svb = %f %f %f", _sample_data[0], _
sample_data[1], _sample_data[2]);
                }
            }
        }
        catch(const std::exception& e)
        {
            sim_logger-
>error("SampleDataPoint::SampleDataPoint:  Parsing exception %s", e.what());
        }
    }

    SampleDataPoint::~SampleDataPoint(void)
    {
        sim_logger->trace("SampleDataPoint::~SampleDataPoint:  Destructor executed");
    }

    std::string SampleDataPoint::to_string(void) const
    {
        sim_logger->trace("SampleDataPoint::to_string:  Executed");

        std::stringstream ss;

        ss << std::fixed << std::setfill(' ');
        ss << "Sample Data Point: ";
        ss << std::setprecision(std::numeric_limits<double>::digits10); // Full double precision
        ss << " Sample Data: "
           << _sample_data[0];

        return ss.str();
    }
}
```