AN EVALUATION OF FLIGHT SOFTWARE PLATFORMS FOR SMALL SATELLITE MISSIONS

Miguel Nunes[(1)], Ethan Chee[(1)], Osiel Montoya[(1)], William Edmonson[(2)],
Scott Ginoza[(1)], Eric Pilger[(1)]
[(1)]*Hawaii Space Flight Laboratory, Honolulu HI 96882, ph +1 808 956 0441,*
miguel.nunes@hsfl.hawaii.edu
[(2)] *Sadaina LLC, Norfolk, VA 23505, +1 919.757.6205, wwedmonson@sadaina.com*

Abstract

Developing and implementing reliable flight software is time-consuming and challenging, particularly for small teams embarking on satellite missions. CubeSats have high failure rates at an average of 45% with no contact after deployment. Some of the causes of this failure statistically can be traced to software issues. Small teams often grapple with a crucial decision: whether to develop their flight software from the ground up, adapt partially completed software from previous missions, or attempt to utilize mature, pre-existing software. This paper aims to resonate with those who identify with these struggles and engage with those who have successfully navigated this path, deploying effective and robust flight software in space with limited resources and time. This paper summarizes existing flight software frameworks that are mature and have flight heritage for CubeSats and proposes attributes to help small teams better understand the capabilities of these software platforms.

INTRODUCTION

Small satellite missions have become ubiquitous among small organizations such as universities, but also with national space organizations such as NASA and ESA. The reduced cost to build and launch is feeding an exciting ecosystem of novel research and more aggressive space missions with CubeSats and Nanosatellites in general, from LEO, to the Moon and Deep Space. Despite the increased number of missions, the complexity of such small satellites is increasing at a fast pace. The hardware and software of such missions are at a level of complexity that small organizations with a handful of students or engineers cannot easily manage. This leads to significant delays, cost overruns, frustrations within the team, and even mission failure. Once the satellite hardware has been properly validated, the mission success hinges on the reliability of its flight software. We will only focus on the flight software development for this work. Future research will bridge the complexity gap between hardware and software.

This paper delves into the intricacies of what constitutes a reliable flight software platform, examining how modern platforms measure up to these benchmarks. An ideal platform is characterized by several key attributes: **modularity**, ensuring flexible and adaptable components; **portability**, allowing software to be used across various systems; a **small memory and processing footprint**, crucial for space computers with limited resources; comprehensive **unit testing** capabilities, to assure software reliability; a **standardized packet protocol(s)**, for consistent data handling; and **thorough documentation** and examples and tutorials, aiding in ease of use and future development.

Miranda and others [1] have also studied flight software frameworks for 'New Space' missions. In our analysis, we study existing open-source flight software platforms, particularly focusing on F Prime from JPL [2], NASA's cFS [3], and the Hawai'i Space Flight Lab (HSFL) COSMOS flight software [4]. These were implemented in CubeSat projects at the University of Hawai'i at the Mānoa Hawaii Space Flight Laboratory.

This paper proposes a systematic evaluation metric to facilitate a more objective evaluation of flight software platforms for different mission requirements and objectives. This system aims to assess the performance and suitability of a software platform in comparison to an idealized model. The metric covers a range of parameters including ease of implementation, scalability, reliability, security features, and efficiency in resource utilization. This evaluation model is designed to aid small teams in making informed decisions when selecting or developing flight software for satellite missions.

Furthermore, the paper explores the challenges and solutions encountered during the implementation of these platforms. Lessons learned are summarized from case studies, including pitfalls and best practices, offering valuable lessons for teams engaged in similar endeavors. One of the key discussions revolves around the balance between customizability and user-friendliness. While tailor-made solutions offer the advantage of being optimized for a particular mission, they often come with increased complexity and a steep learning curve. Conversely, more generic platforms may sacrifice some degree of optimization but offer greater ease of use and faster implementation times.

Another aspect of the paper is the exploration of community support and resources available for flight software development. The role of open-source communities, academic-industry collaborations, and governmental support in fostering a more accessible and robust ecosystem for flight software development is examined.

The paper concludes by highlighting the future trajectory of flight software development, emphasizing the need for more collaborative approaches, better standardization, and the integration of emerging technologies like Model-Based Systems Engineering, machine learning, and artificial intelligence. These advancements promise to enhance the capability, efficiency, and reliability of flight software, potentially revolutionizing satellite missions, particularly for small teams with limited resources.

A standardized architecture for all flight software must implement an OS abstraction layer that handles essential OS functionalities, such as a file system, task scheduler, network manager, memory manager, I/O manager, clocks and timers, synchronization, messenger interpreter, cryptography, and system health. These can be interchanged with other OS abstractions to meet the hardware requirements without interfering with other platform components. Above this OS abstraction layer is an application layer consisting of essential flight software core components that all spacecraft adapts, such as commanding, telemetry, file uplink and downlink, and packet protocol validation.

FLIGHT SOFTWARE PLATFORMS IN SCOPE

When discussing software platforms for space missions, there are several prominent options developed by various organizations. The scope of this paper is limited to the platforms listed in Table 1: NASA's cFS, JPL's F' (F Prime), and HSFL COSMOS.

Table 1: Flight Software Platforms

Core Flight System (cFS)	Developer	NASA's Goddard Space Flight Center.
	Purpose	A platform for spaceflight software development with a reusable framework and a set of reusable applications.
	Features	Highly scalable and configurable. It is designed to be portable across various embedded hardware and operating system platforms. The architecture is component-based, making it easy to add, remove, or replace components.
	Usage	Widely used in various NASA and other governmental projects. It's known for reliability and robustness in mission-critical applications.
F Prime (F')	Developer	NASA's Jet Propulsion Laboratory.
	Purpose	An open-source framework to support the development and deployment of embedded systems and spacecraft.
	Features	It includes a component architecture, a robust set of development tools, and a deployment model that supports a variety of platforms (from desktop systems to spacecraft).
	Usage	It has been used in various space missions, including experimental CubeSats and instruments for Mars exploration.
HSFL COSMOS	Developer	The Hawaii Space Flight Laboratory (HSFL).
	Purpose	COSMOS is designed to control ground-based and space-based systems.
	Features	Provides capabilities for command and control, communication, automation, and display of system data. It's known for its flexibility and ease of integration with other systems.
	Usage	Primarily used for small satellite missions and has been adopted for various educational and research-oriented satellite projects.

A summary comparison between these flight software platforms can be performed using broad criteria:

- Scope and Usage: While cFS and F' are heavily used in government-led space missions, COSMOS is more oriented toward educational and smaller-scale projects.
- Community and Support: NASA's cFS and JPL's F' benefit from strong support from NASA and an extensive community of users and developers. COSMOS, while perhaps less globally recognized, still supports a niche community focused on satellite operations.
- Technical Features: All these frameworks offer high modularity and are designed for mission-critical applications. However, they differ in their primary focus—cFS and F' on general space missions, and COSMOS on satellite management.

Each of these systems brings unique strengths to space mission development, and the choice between them would depend on the specific needs of the mission, such as the focus on robotics, general spacecraft operations, or educational purposes.

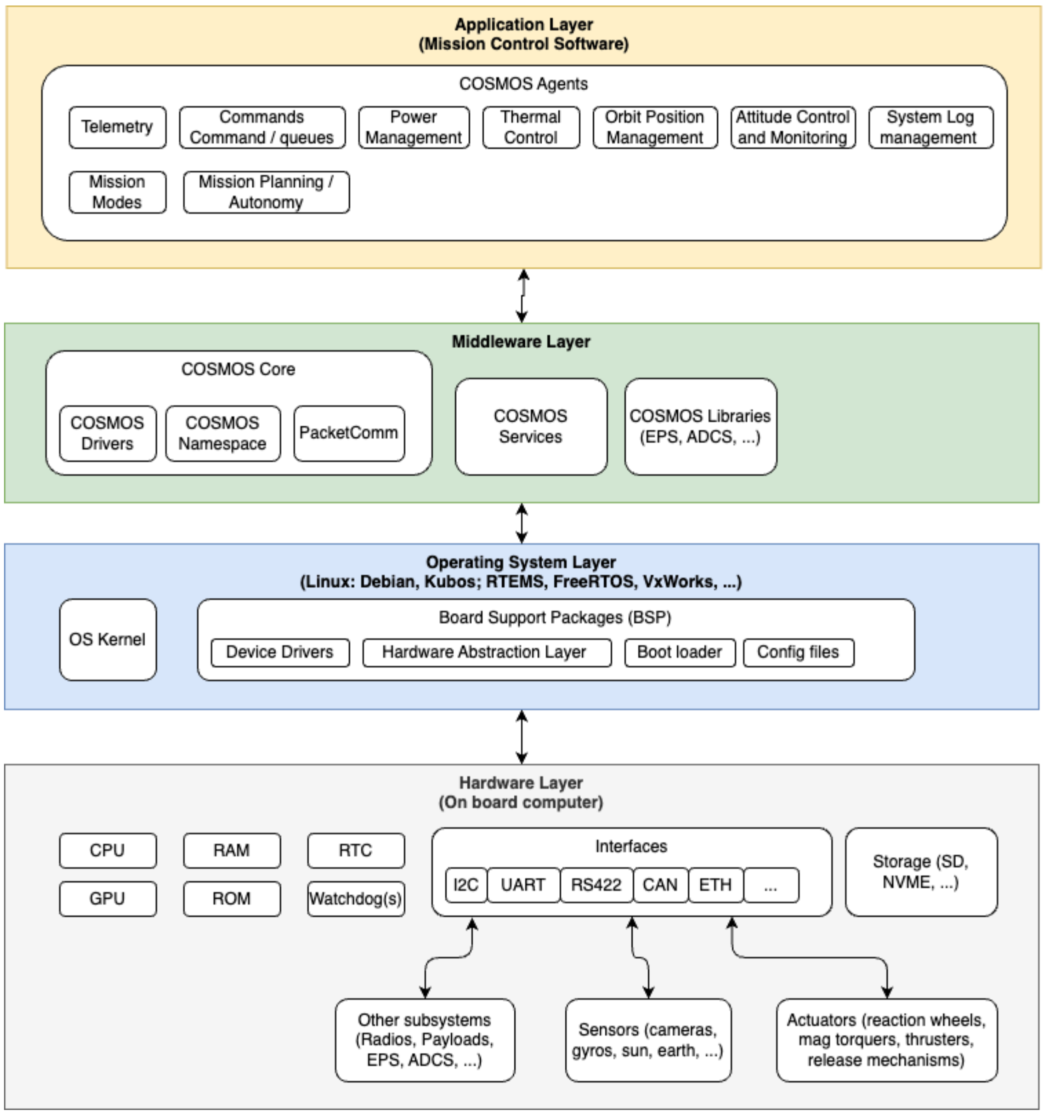
Application Layer
(Mission Control Software)
COSMOS Agents
Telemetry
Commands Command / queues
Power Management
Thermal Control
Orbit Position Management
Attitude Control and Monitoring
System Log management
Mission Modes
Mission Planning / Autonomy
Middleware Layer
COSMOS Core
COSMOS Drivers
COSMOS Namespace
PacketComm
COSMOS Services
COSMOS Libraries (EPS, ADCS, ...)
Operating System Layer
(Linux: Debian, Kubos; RTEMS, FreeRTOS, VxWorks, ...)
OS Kernel
Board Support Packages (BSP)
Device Drivers
Hardware Abstraction Layer
Boot loader
Config files
Hardware Layer
(On board computer)
CPU
RAM
RTC
GPU
ROM
Watchdog(s)
Interfaces
I2C
UART
RS422
CAN
ETH
...
Storage (SD, NVME, ...)
Other subsystems (Radios, Payloads, EPS, ADCS, ...)
Sensors (cameras, gyros, sun, earth, ...)
Actuators (reaction wheels, mag torquers, thrusters, release mechanisms)

Figure 1: COSMOS OSI equivalent model

DEFINITIONS TO ANALYZE THE VARIOUS FSW

Several terms describe flight software, and to avoid confusion, we present the following definitions.

Table 2: Flight Software Terminology

Open-source	Freely available on the internet in an open code repository, e.g., GIT
Flight Heritage	Flown successfully on at least one or more successful small satellite missions
Small Footprint	Minimal memory and CPU usage
Requirements Traceability	Requirements can be individually traced to their implementation with verification
Modularity	Compartmentalizable code as a function of FSW architecture, i.e., ease of code reusability.
Portability	Can be used on multiple types of embedded processors and RTOS
Reliability	Reliability of code through integrated analysis tools with significant coverage of system SW

FLIGHT SOFTWARE FRAMEWORK ATTRIBUTES

We list a number of attributes that were determined to be required for selecting or developing a flight software framework. These attributes will have different weights for different users:

- **“Inside” software developers:** will be concerned about developing their flight software for their mission success
- **“Outside” software developers**: will be concerned about using a third-party/existing flight software for their mission success

These attributes contribute significantly to the success of the software platforms used in aerospace missions, helping to ensure that they meet the rigorous demands of space environments. Each platform (cFS, F', and COSMOS) may emphasize different attributes based on their specific applications and target user communities. We have proposed some quantitative metrics to evaluate the attribute as a way to create a model for evaluating the various software frameworks.

Table 3: Summary List of desirable attributes for an ideal flight software platform

Reliability and Fault Tolerance	Definition	Software must operate correctly under various and often extreme conditions without failure. Fault tolerance mechanisms, such as error detection, correction, and redundancy, are crucial.

	Necessity	Space missions often last for extended periods, and the possibility of maintenance or repair is minimal to none. Systems must continue to function despite hardware or software failures.
Scalability and Modularity	Definition	The software should be capable of scaling from small to large systems and be modular to facilitate updates and maintenance. It should also be capable of “plug-and-play” with various missions.
	Necessity	Missions vary in size and complexity, and software may need to be reused across different projects with varying requirements.
Portability	Definition	Software should run on various hardware and operating systems, as missions might use different platforms. Implementation of an OS abstraction layer.
	Necessity	This increases the software's usability across different missions and reduces development costs.
Real-time Operations	Definition	Capable of meeting real-time computing requirements, ensuring that tasks are performed exactly when needed.
	Necessity	Many space operations, such as satellite attitude control and robotic maneuvers, require precise timing to be successful.
Resource Efficiency	Definition	Efficient in terms of processing power, memory, and energy, as space missions often have limited resources.
	Necessity	Spacecraft resources like power and computing capacity are constrained, so the software must maximize performance with minimal resource use.
Security	Definition	Must include robust security features to prevent unauthorized access and ensure data integrity.
	Necessity	The increasing connectivity of space systems, especially with the growth of commercial space activities, opens them up to potential cyber threats.
Interoperability	Definition	Ability to communicate and operate in conjunction with other systems, software, and standards.

	Necessity	Missions often involve collaboration between multiple systems and organizations, requiring seamless data and command exchange.
Testability	Definition	Software design that facilitates thorough testing and validation before deployment.
	Necessity	Due to the high costs of failure in space missions, software must be extensively tested to ensure it behaves as expected under all conditions.
Documentation and Community Support	Definition	Well-documented software with strong community and developer support.
	Necessity	Good documentation and support can reduce the learning curve and facilitate problem-solving and enhancements, which are critical for the success and longevity of space mission software.
Flight Heritage	Definition	Flight heritage refers to the track record of a technology, component, or system having been successfully flown and operated in space.
	Necessity	This reduces the perceived risk of integrating the software into new missions.
Source Code Availability	Definition	Source Code Availability refers to the extent to which the source code of a software application or system is accessible for viewing, modification, or redistribution. Distributing the source code via Github, Bitbucket and Gitlab are the most common distribution platforms currently available.
	Necessity	For small teams this is a very important aspect as it makes the source code available for free but also open to changes for specific mission needs.

We describe in more detail a selection of these flight software attributes. We introduce evaluation metrics to help developers and users better understand how they can evaluate each software framework and their own software development using these frameworks. In future work, we will present a more comprehensive evaluation system.

Modularity

Definition: Modularity in flight software refers to a system design where components can be added, removed, or replaced without compromising the overall system's integrity. A component is considered modular if it can operate independently and can also be used by other parts of the system.

Example: In the context of spaceflight operations, modularity allows for the independent software development of unique hardware or interface modules such as a radio, camera, sensors, or serial protocols. Developers are able to add, remove, or modify a given component that fits their project requirements, with no modifications to other components. For instance, if an Inertial Measurement Unit (IMU) driver needs to be implemented, the developer should be able to write their own module for the unit or utilize an existing module to adapt to their current project without alterations to other modules. The IMU module can then be connected to a serial interface module such as I2C. If the model of the IMU were to be changed, the current IMU module should be easily swapped with a new module without modifications to other modules. If multiple IMU devices exist within the same system, a single IMU module can be used for multiple instances to mitigate code duplication. A module should also be able to be connected to any other module that supports its interfaces. An example would be connecting an IMU module to an I2C module.

Evaluation Metric: The quality of a platform's modularity can be classified by these requirements:

- λ_1: A module must be independent and cannot rely on the functions of other modules
- λ_2: The capability of facilitating "design-first." The architecture of components and their connections can be designed before being implemented in the software
- λ_3: Capability of adding, removing, or modifying a module with no software modifications to any other module within the same system. $\lambda_3 = \frac{1}{a}$ where a is the total number of modules modified.
- λ_4: Capability of deploying multiple instances of a single module
- λ_5: Capability of inter-module connections where applicable

OS Abstraction Layer

Definition: An OS abstraction layer (OSAL) is an interface that provides a standardized API to allow platforms to interact with the underlying operating system.

Example: Many small satellites use a Linux-based OS, no OS, or a combination of both. If the platform were to support both Linux and bare metal applications, an OSAL must be present. Let's use a Raspberry Pi running Ubuntu and a Teensy 4.1 using the Teensyduino (Arduino) package for comparison. Ubuntu has a file system and provides its own functions to open, read, write, move, etc., a file. However, a Teensy board does not have any OS and therefore does not exhibit a file system, so the Teensyduino package does not provide any file system functionalities. The flight software platform must not directly call Linux file system functions like "open()" since the Teensyduino compiler does not recognize it. Instead, the platform must call a standardized file system API which interfaces with the OSAL.

Evaluation Metric: A platform that adapts an OSAL must have these requirements:

- λ_1: Capability of being replaced (either partially or completely) by another OSAL
- λ_2: The platform must not directly call OS-specific functions or system calls, and instead be replaced with platform aliases.

- λ_3: The OSAL must be designed to support these modules: file systems, clocks and timers, threads (or virtual threads), synchronization, system health, I/O drivers, message interpreter, memory interface, and task scheduling. If the hardware/OS does not support a module, it should be replaced with stubs that have no internal function.

Portability

Definition: Portability in flight software is the ability for the platform to be used by different computer hardware and/or operating systems with no modifications to any of the core files of the platform. Core files exclude OS abstraction files (if applicable) and any project or hardware-specific modules.

Example: Flight hardware across numerous small satellite missions varies depending on size, budget, mission requirements, and more. The onboard computer hardware may vary to meet those mission constraints and requirements, and thus, the software to interface with the computer will vary. Operating systems like Linux, VxWorks, RTEMS, and FreeRTOS [19], and even those that have no OS, have all been deployed in small satellite missions. The flight software platform must be portable so that developers can deploy it onto any OS of their choice without modifying the core features of the platform. However, external developers may still need to create their own OS abstraction to interface with their hardware and operating system if such OS abstraction does not already exist.

Evaluation Metric: The quality of a platform's portability to be supported by different hardware/OS. According to NASA, the four operating systems commonly used in small satellites are Linux, VxWorks, RTEMS, and FreeRTOS.

Driver Templates

Definition: Driver templates are modules that developers can import into any project, or be made very generically such that developers can easily modify them to suit their requirements. Driver templates can both relate to modularity and portability. They relate to modularity in a way that new projects can reuse existing modules from other projects with no modifications; essentially "plug-and-play" functionality. They relate to portability in a way that the templates are generic enough that they can be customized to be integrated with other hardware/OS.

Example: Serial communication protocols like UART, SPI, CAN, and I2C are very common among embedded systems. Drivers should be designed so that these protocols can be implemented into a project without needing to be redesigned. The F Prime Arduino package (fprime-arduino) provides several driver modules that can be reused in any F Prime Arduino project: UART, SPI, I2C, TCP Client, GPIO, Analog, and PWM. Developers can easily import the package and use these drivers without any modifications. Driver templates increase in demand for complicated drivers, including, but not limited to, radios, ADCS, and propulsion systems.

Evaluation Metric: The quality of a platform's ability to provide driver templates can be classified by these requirements and scores:

- λ_1: Capability of automatic code generation for generic modules

- λ_2: Provide out-of-the-box drivers for commonly used embedded system interfaces (i.e. UART, I2C, SPI, GPIO). $\lambda_2 = \frac{a}{4}$ where a is the number of "plug-and-play" interfaces provided out-of-the-box for the interfaces described above
- λ_3: Capability to import drivers from existing projects or repositories

Community Engagement

Definition: Flight software, like any form of software, is written by software developers for the benefit of users. The relationship between developers and users forms the basis for a community and can either accelerate or hinder flight software development. By analyzing this relationship, we can classify the quality of community engagement.

In *The Cathedral and the Bazaar*, Eric Raymond contrasts "Cathedral" and "Bazaar"-type software development in open-source projects [6, p.21]. These development styles differ in how developers interact with users.

"Cathedral" style software development is characterized by limited engagement between the community and developers. It is typified by a "wizards in an ivory tower" attitude, with developers writing code in isolation from the users. Communication is usually in one direction, from the developers to the users, and only a select few developers within the developing organization can change the code.

"Bazaar" style software development, by contrast, is typified by open and frequent communication between developers and the community. Flight software code can be contributed by the community, and it rapidly iterates. Discussion between developers both within and outside the developing organization is lively, reflecting the opinions and priorities of developers. The choices made in flight software development are community-led, reflecting the needs of users.

It is important to note that an open-source flight software project can still be developed in the "Cathedral" style. The designation of the software development style refers to the relationship between users and developers. Thus, a closed-source project will always be considered "Cathedral" style, but an open-source project is not automatically "Bazaar" style.

Example: The Linux kernel is a good example of the Bazaar style of development. Kernel development is distributed among thousands of developers, each specializing in a particular aspect of the codebase. Although a defined hierarchy exists for approving changes to the code, the actual development itself is delegated to the community as a whole. There is no restriction on who you must be in order to contribute; the only requirement is the technical ability and desire to take responsibility for an aspect of the kernel.

Evaluation Metric: Community engagement can be quantified by evaluating flight software development metrics. These metrics measure the pace of software development and identify who is driving software development. Common software development platforms such as GitHub and GitLab often provide these metrics.

Coding Standards

Definition: Coding Standards and Coding Style are often conflated [7]. For the purposes of this evaluation metric, *Coding Standards* are a set of guidelines or rules defined by industry, groups, or organizations to define how the code should be designed. *Coding Style* describes specific choices to be made in the process of writing code that is to be adhered to by all developers in the organization.

Coding Standards are generally high-level and subjective. An example is “Task synchronization shall not be performed through the use of task delays” from the JPL institutional coding standards [8]. This standard reflects the priorities and needs of the software in the context in which it will run. For this example, the embedded and safety-critical context informs the design decision.

Coding Style, on the other hand, refers to specific choices within the programming language's syntax. A common example is forbidding the use of tab indentation and enforcing indentation through spaces only. Linting tools such as clang-format [9] can be used to enforce such decisions across an entire organization.

The ideal flight software has clearly defined coding standards and style. Tools are provided to ensure that style is being enforced. Procedures exist to ensure that changes to the code maintain the chosen coding standards.

Example: Coding Standards relevant to CubeSat applications are provided by NASA [8][10][11], MISRA [12], LLVM [9], and the “Power of 10” rules for safety-critical code [13]. Similarly, different organizations publish their Coding Styles to ensure all code written by their developers maintains the same syntax. For example, LLVM [9], Google [14], Chromium [15], Mozilla [16], Apple [17], and GNU [18] all publish publicly available Code Style guides for the C++ language.

Evaluation Metric: When evaluating flight software frameworks, a CubeSat software development team should inspect a candidate framework by the following criteria:

- A coding standard is explicitly defined and provided as part of the flight software documentation.
- Merge commits require analysis against the coding standard.
- A coding style is explicitly defined.
- Instructions are included to configure the development environment to apply the coding style.
- The coding style is automatically enforced using CI/CD.

Memory Requirements

Definition: The compilation size of flight software must be small enough to fit on the satellite's onboard computer. Also, the runtime memory usage must be within the memory available by the processor. The processor or computer chosen for the satellite hardware design is often constrained by factors such as energy consumption. These constrictions necessarily limit the code storage memory and RAM available for the flight software. That a flight software framework is able to produce a compiled program that meets the constraints of the hardware is a hard requirement, and the evaluation metric should be adjusted accordingly.

Examples:

- Microcontroller Level: Teensy (8Mb Flash Memory, 1Mb RAM), ISIS OBC NOR (1Mb NOR Flash)
- Single Board Computer Level: Raspberry Pi Zero W, Beaglebone (>4GB SD Card, 512MB RAM), ISIS OBC (>4GB SD Card, 64 MB SDRAM)
- High-performance computer: Unibap Space Cloud iX5-100 (128 GB x2 of M.2 SSD)

Evaluation Metric: Three scales of processors are considered: microcontroller level, single-board computer, and high-performance flight computer. Depending on the complexity of the flight software, the processor shall be able to produce a compiled program that meets the memory requirements of the desired computer level.

Ease of Operations

Definition: Various aspects of satellite software development can consume development time, which could be conserved with a flight software framework that supplies a solution. This metric considers operational development as implemented in the satellite and on the ground. Additionally, flight software does not exist as a standalone solution – it must be operated from the ground by human operators. Integration with available ground software is desirable to save development time. Finally, the operator should have access to the satellite's current state and timeline of states.

Evaluation Metric:

A: Availability of integration with existing ground software

B: State knowledge:

- B_1: Can control telemetry frequency and logging during comm-link and non-comm-link
- B_2: Command delivery is acknowledged at the ground level
- B_3: The historical state and execution of commands of the satellite is known
- B_4; Error states are definable, recorded, and contained in the telemetry

When evaluating flight software platforms like NASA's cFS, JPL's F', and HSFL COSMOS, it is essential to assess how each aligns with critical attributes necessary for successful space missions. These attributes include reliability, modularity, portability, real-time capabilities, resource efficiency, security, interoperability, testability, documentation, and community support. Below is a summary of how these platforms might be evaluated based on these attributes.

Table 4: Flight Software vs Attributes Evaluation Summary

FSW Attributes	cFS	F Prime	HSFL COSMOS
1. Reliability and Fault Tolerance	Yes	Yes	Partial
2. Scalability and Modularity	Yes	Yes	Partial
3. Portability	Yes	Yes (x86, ARM, etc. OS: Linux, macOS, Baremetal, RTEMS, etc.)	Yes (portable to x86, Arm, Linux, Win, macOS, etc.)

4. Real-time Operations	Yes	Yes	Partial (near real-time, implementation is non-deterministic)
5. Resource Efficiency	Yes	Yes	Partial
6. Security	-	Yes	Yes
7. Interoperability	-	Yes	Yes
8. Testability	Yes	Yes	Yes
9. Documentation and Community Support	Yes	Yes	Partial
10. Flight Heritage / Reliability assurance	Yes	Yes	Partial (3 missions)
11. Open Source	Yes	Yes	Yes

CONCLUSIONS AND FUTURE WORK

This work is an introduction to what to expect from a flight software architecture for small satellites. This is a complex topic, and future work will dive deeper into the understanding of the details of what proper constructs are needed for reliable flight software with low cost. We evaluated three of the most relevant flight software frameworks currently available for the small satellite community: NASA’s cFS, F Prime, and HSFL COSMOS. We also proposed a number of attributes that flight software should comply with. The open source attribute is very important for small organizations that have limited resources or are just starting to design new missions.

In conclusion, the evaluation of flight software platforms such as NASA's cFS, JPL's F Prime, and HSFL COSMOS reveals a complex landscape of attributes that are critical for successful space missions. Each platform offers distinct advantages tailored to specific needs, ranging from extensive reliability and modularity in NASA's cFS. The choice of software depends on a variety of factors including mission requirements, environmental constraints, resource limitations, and the need for robust security measures. Understanding the strengths and limitations of each platform is essential for stakeholders to make informed decisions that align with their strategic objectives and operational demands.

Furthermore, as space missions continue to evolve with increasing complexity and broader objectives, the importance of selecting the right flight software cannot be overstated. Platforms like JPL's F Prime and HSFL COSMOS show how specialized adaptations and enhancements in software can lead to significant improvements in mission performance and operational efficiency. The ongoing development and support of these platforms also highlight the dynamic nature of space exploration technology, encouraging continuous innovation and collaboration within the aerospace community. This not only enhances current mission capabilities but also paves the way for future advancements in space technology.

REFERENCES

[1] Miranda DJF; Ferreira M; Kucinskis F; McComas D (2019) A Comparative Survey on Flight Software Frameworks for ‘New Space’ Nanosatellite Missions. J Aerosp Technol Manag, 11: e4619. https://doi.org/10.5028/jatm.v11.1081

[2] Bocchino Jr. RL, Canham TK, Watney GJ, Reder LJ, Levison JW (2018) F Prime: an open-source framework for small-scale flight software systems. Presented at: 32nd Annual AIAA/USU Conference on Small Satellites; Logan, USA.

[3] Core Flight System (2017) About the technology and why cFS. Core Flight System; [accessed 2018 May 18]. http://coreflightsystem.org/why-cfs/

[4] Sorensen, T. C., Pilger, E. J., Wood, M. S., Nunes, M. A. & Yost, B. D. Development of a Comprehensive Mission Operations System Designed to Operate Multiple Small Satellites. in AIAA/USU Small Satellite Conference, Logan, UT. #SC11-IX-3 1 (2011).

[5] Wilmot J, Fesq L, Dvorak D (2016) Quality attributes for mission flight software: a reference for architects. Presented at: IEEE Aerospace Conference; Big Sky, USA. https://doi.org/10.1109/AERO.2016.7500850

[6] E. S. Raymond, The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary, Revised. Sebastopol, CA: O’Reilly & Associates, 2001.

[7] Y. Wang, B. Zheng, and H. Huang, “Complying with Coding Standards or Retaining Programming Style: A Quality Outlook at Source Code Level,” *J. Software Engineering & Applications*, vol. 1, no. 1, Art. no. 1, Dec. 2008, doi: 10.4236/jsea.2008.11013.

[8] “JPL Institutional Coding Standard for the C Programming Language.” Mar. 03, 2009. Accessed: Apr. 14, 2024. [Online]. Available: https://yurichev.com/mirrors/C/JPL_Coding_Standard_C.pdf

[9] LLVM Project, “LLVM Coding Standards — LLVM 19.0.0 git documentation,” LLVM Coding Standards. Accessed: Mar. 14, 2024. [Online]. Available: https://llvm.org/docs/CodingStandards.html

[10] J. Doland and J. Valett, “C style guide,” SEL-94-003, Aug. 1994. Accessed: Mar. 14, 2024. [Online]. Available: https://ntrs.nasa.gov/citations/19950022400

[11] “Chapter 3: Software Management Requirements.” Mar. 08, 2022. Accessed: Mar. 14, 2024. [Online]. Available: https://nodis3.gsfc.nasa.gov/displayDir.cfm?Internal_ID=N_PR_7150_002D_&page_name=Chapter3

[12] “MISRA.” Accessed: Mar. 14, 2024. [Online]. Available: https://misra.org.uk/

[13] G. J. Holzmann, “The power of 10: rules for developing safety-critical code,” *Computer*, vol. 39, no. 6, pp. 95–99, Jun. 2006, doi: 10.1109/MC.2006.212.

[14] “Google C++ Style Guide.” Accessed: Apr. 14, 2024. [Online]. Available: https://google.github.io/styleguide/cppguide.html

[15] “Chromium C++ style guide.” Accessed: Apr. 14, 2024. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/refs/heads/main/styleguide/c++/c++.md

[16] “C++ Coding style — Firefox Source Docs documentation.” Accessed: Apr. 14, 2024. [Online]. Available: https://firefox-source-docs.mozilla.org/code-quality/coding-style/coding_style_cpp.html

[17] “Code Style Guidelines,” WebKit. Accessed: Apr. 14, 2024. [Online]. Available: https://www.webkit.org/code-style-guidelines/

[18] “GNU Coding Standards,” GNU Coding Standards. Accessed: Apr. 14, 2024. [Online]. Available: https://www.gnu.org/prep/standards/standards.html

[19] NASA State-of-the-Art of Small Spacecraft Technology, https://www.nasa.gov/smallsat-institute/sst-soa/small-spacecraft-avionics/#8.4.3