SpaceOps-2023, ID # 347

Design and Implementation of a Flight Software for a Modular Satellite

Mohammed Eshaq[a*], Ibrahim Al-Midfa[b], Eisa Al-Shamsi[b], Zakareyya Al-Shamsi[b], Shadi Atalla[a], Saeed Al-Mansoori[b], Hussain Al-Ahmad[a]

[a] *MBRSC Lab, University of Dubai, Dubai, United Arab Emirates*
[b] *Mohammed Bin Rashid Space Center (MBRSC), Dubai, United Arab Emirates*
* *Corresponding Author,* meshaq@ud.ac.ae.

Abstract

Due to their modularity, standardization, and lower cost, CubeSats are gaining in popularity and expected to dominate space in the near future. They can be easily adapted for new missions by swapping out payload cubes without extensive hardware redesign. However, achieving such a high degree of modularity can pose a challenge for the Flight Software that governs the spacecraft's operation, which must also be modular and reusable. To address this challenge and streamline software development, we propose a Flight Software designed with specific features from the ground up.

App-Based Flight Software: The Flight Software presented in this work comprises many Applications (Apps). The goal is to have a modular and service-oriented architecture where Apps can be added/removed based on mission requirements. Command Line Interface: Satellites typically comprise an onboard computer and many subsystems communicating via various protocols. Each subsystem has its own set of commands and responses. A human-readable Command Line Interface (CLI) is added as an abstraction layer that assists the team on the ground in controlling the spacecraft without needing to deal with the plethora of native binary commands and responses used in each subsystem. These commands can then be executed when the satellite is on the ground using a Command Console, or they can be sent to the satellite while in-mission using CubeSat Space Protocol packets (CLI/CSP). Script Engine: After defining the CLI command set, the same commands can be used in a Script Engine. Script files are simply sequences of CLI commands. When a script file is triggered, commands are released and executed by the Script Engine in a timely fashion while orbiting, whether the satellite is still in contact or not. Bootloader: The ability to modify/update Flight Software while in-mission is a well-sought-after feature. It adds flexibility and makes the Flight Software future-proof.

Keywords: CubeSat, Flight Software, Onboard Computer, Command Line Interface, Script Engine, Bootloader.

1 Introduction

Technological advancements over the years have led to the creation of smaller satellites that can be easily held and handled by a single person. A popular category of such smaller satellites is the Nano Satellite category (weighing between 1 to 10 KG) [1]. An even more popular type of Nano Satellite is the Cube Satellite (or CubeSat), a miniaturized satellite built out of standardized cubes; The name comes from the *Cubed Units* used to build the satellite. Any CubeSat may have the size of 1 Unit (1U) or multiples of such Units where each Unit measures 10 x 10 x 10 cm. These units typically have a mass of about 1 kilogram per unit. A CubeSat of such magnitude can be quickly built with in-house-developed or commercial-off-the-shelf (COTS) electronics and components. CubeSats are typically used for space research and commercial use [2]. This meant that developers with limited to no experience in satellite technology away from large space agencies, such as students at universities and research centers or even enthusiasts at startup companies, to explore the realms of space with CubeSats [3]. CubeSats' standardization and adoption in academia and industry have led to mass production and an abundance of off-the-shelf components. Although this has resulted in a significant reduction in satellite development time and cost associated, the fact remains that a considerable amount of mission development time and effort is still spent on flight software development [4] unless the code is reusable [5].

A satellite is composed of multiple subsystems, each with a specific function such as power or communication, and an Onboard Computer (OBC). These subsystems are essentially embedded systems that are connected to the OBC. The OBC is responsible for processing commands and responses, managing communication with the ground, and overseeing the overall operation of the satellite. Designing and implementing the Flight Software (FSW) that runs on the onboard computer is a crucial aspect of this work. The flight software must provide "numerous services such as computer boot-up and initialization, time management, hardware interface control, command processing, telemetry processing, data storage management, flight software patch and load, and fault protection" [6]. Implementing flight software is more challenging than other commercial software; Once the satellite has launched, there is no interaction with users unless the satellite makes contact with the ground station. Even then, it can only communicate via uplink and downlink during a tiny time window. Most of the mission time, the FSW runs alone with no supervision from the ground. Thus, it must be able to recover from faults on its own [6].

Satellites are increasingly critical components of the Internet of Things (IoT) infrastructure, prompting the Mohammed Bin Rashid Space Center (MBRSC) to establish the Payload Hosting Initiative (PHI) program for Cube Satellites. The first iteration of the PHI program will host a 5G-capable communication subsystem as a payload.

This project aims to design and implement the Flight Software (FSW) for the first CubeSat included in the PHI program. To achieve this, we first review previous works in this area to determine the desired characteristics and features for our FSW. After evaluating multiple Flight Software Development Kits (FSDKs) and frameworks, we decide to proceed with our custom design. We present an FSW architecture based on our Literature Review's findings, featuring a Command Line Interface (CLI), CLI commands over CubeSat Space Protocol packets (CLI/CSP), Script Engine, and Bootloader.

We thoroughly demonstrate the aforementioned features before presenting the results of running the proposed FSW in a FlatSat environment. Our conclusions and recommendations are based on the results obtained.

2 Literature Review

2.1 Characteristics of a Good Flight Software

Through careful examination of existing works and solutions, we have identified the desired characteristics for our FSW architecture.

A **modular** app-based software architecture satisfies requirements, accelerates development, and allows for future mission reuse by selecting needed apps. The Norwegian University of Science and Technology developed an FSW for a nanosatellite onboard computer using a similar approach, but with heavier reliance on CSP for inter-task messaging instead of the more efficient built-in message queues in the Operating System [7]. The author in [8] took it further and asserted that modularity determines flight software quality. Every design aspect should be highly modular to make the code more flexible and modifiable. Furthermore, many additional efforts were made to develop flight softwares for CubeSats with a heavy emphasis on modularity [2] [5] [9] [10]. The work in [3] summarized several published papers describing FSWs of CubeSat missions and added that "modularity, extensibility, flexibility, robustness and fault-tolerance have been identified as the main features of the flight software for nanosatellites". This work also proposed an approach to evaluate modular FSW for CubeSats by tracking architectures. This approach can be used as an expansion of our work. Nonetheless, evaluating our FSW using an architecture-tracking approach is out of the scope of this paper.

In addition to modularity, the work also concluded that most of the examined solutions implemented the FSWs as ***multilayered architecture*** to exploit abstraction layers. For example, the FSW proposed in [8] is "divided into four modular layers: at the top are application-specific tasks, next are hardware specific hardware libraries, at the bottom are microcontroller specific drivers, and supporting all layers are general purpose software libraries." This approach is similar to ours; however, the final FSW architecture differs from our design due to dissimilarities in hardware. Furthermore, the launch sequence is hardwired into the code in this work, as in most FSWs. In our design, however, we can execute the launch sequence (*Launch and Early Orbit Phase - LEOP Script)* by letting the script engine execute a script file. The work in [11] also demonstrated a multilayered architecture. Furthermore, it stretched the FSW development process further by introducing a robust feedback software life cycle model and technical specification verification framework, both outside our work's scope.

The Real-Time **Operating System (RTOS)** for CubeSat missions is usually chosen based on the microcontroller selected. Commonly used RTOS options include GNU/Linux or **FreeRTOS**. The choice of microcontroller and RTOS are interrelated, as a more powerful microcontroller is needed for FSW that requires high-level programming language or features only available in GNU/Linux, resulting in increased power consumption. On the other hand, using a lightweight RTOS can conserve power at the expense of some processing capabilities. [3]. In our work, we design and implement an FSW also using the Free Real-Time Operating System (FreeRTOS) [12], as in [2], [4], [7], [13], [14], and [15]. In addition, the work in [10] also uses FreeRTOS yet focuses heavily on the design of FSW robustness by employing methods of 'Fault Detection, Isolation, and Recovery' (FDIR) and 'Failure Mode, Effects, and Criticality Analysis' (FMECA).

Moreover, the works in [2] and [1] are similar to ours because they also employ ***Bootloaders*** for their FSWs for in-mission image update capability. However, their implementation of the bootloader is quite different from ours. The work in [2] also implemented a ***Command Line Interface*** (CLI) used in their testing. However, we take CLI a step further and implement CLI over CSP (CLI/CSP).

The comprehensive work in [14] attempts to develop a generic FSW and seems very promising. However, we should keep in mind that many generic and simulated works are published just to lay the foundation for future FSW developers without actually implementing and testing the FSW in actual satellite missions. Works as such remain theoretical and lack space heritage. This also applies to the work in [16], which can be used as a basic guideline for CubeSat FSW architecture design. This work also foresees that as technology advances, new and more efficient subsystems will be available, and "nanosatellites will, then, increase their payload capacities and will likely require flight software capable of controlling and interfacing more subsystems." Therefore, it is essential to have scalable and flexible software architectures where modules can be added, modified, or removed without affecting the architecture [16].

2.2 Flight Software Frameworks and Development Kits

While various Flight Software Development Kits (FSDKs) and Frameworks are available to speed up software development, we chose to create a custom flight software for our mission at MBRSC, rather than relying on existing FSDKs. This approach differs from that of many CubeSat teams [13], [17]. Nevertheless, before starting the *custom* implementation of our FSW, we reviewed the following frameworks.

One of the few open-source and freely-available frameworks is the well-established **NASA's cFS** (Core Flight System) [18]. In fact, our software design was inspired by NASA's cFS framework. It employs a highly modular design where each functionality is managed by a software module and apps. Each app can be added or removed from the FSW based on the requirements, the mission, and the subsystems used. The works in [19] and [20] use cFS to develop nanosatellite FSWs. However, it is technically challenging to configure and deploy cFS due to its rooted history with large and complex satellites [4]. The OpenSatKit [21] was created to mitigate these problems; however, it adds a learning curve with even additional complexity [4] [9]. Even though this suite is tested in NASA's nanosatellite missions, its adoption is still not broad enough. Many current CubeSat FSWs are still implemented directly (without cFS) on top of simpler lightweight operating systems such as FreeRTOS [16].

Besides, NASA's cFS does not natively support FreeRTOS [22], the Real-Time Operating System we intend to use. Many attempts to *'port'* cFS to FreeRTOS have been made [23]; however, running a ported cFS on our MCU with only 1 MB of RAM is not recommended. [24] Another framework by NASA is the primitive open-source **F Prime** [25] made by Jet Propulsion Laboratory. "Unlike cFS, it is specifically tailored for small-scale systems, reducing complexity, and uses static topologies" [4].

The **KubOS** Framework is an open-source platform that equips developers with the necessary tools and libraries to rapidly deploy space-ready FSWs. The framework is designed to handle all aspects of the flight software, allowing developers to focus solely on implementing Mission Applications that dictate the satellite's behavior. The central component of this framework is the KubOS Linux operating system, which is a custom Linux distribution adapted to host the satellite applications [9]. KubOS and other Linux-based SDKs require OBCs with relatively high hardware specifications, such as a recommended RAM size of 64 MB. [26]. Other frameworks, such as CubedOS, are promising yet still work in progress [9]. Similarly, the **NanoSat MO** Framework, made by the European Space Agency, is an advanced framework for small satellites based on CCSDS Mission Operations services. [27]

The work described in [4] presents a comprehensive review of available flight software development frameworks and proposes a framework specifically tailored for CubeSat missions. While our approach shares some similarities with this proposed framework, we had to customize our approach to meet the specific mission requirements of MBRSC. Additionally, our approach heavily incorporates lessons learned from previous MBRSC missions, which influenced our application and task skeletons. Therefore, our flight software development process deviates from the proposed framework described in [4].

3 Overview of Hardware Architecture

A typical satellite is composed of various hardware components, including:

1. Onboard Computer: The primary computer responsible for mission operations, communication, and overall satellite governance. The Flight Software (FSW) runs on the OBC.
2. Electrical Power System: Collects energy from the Solar Panel Arrays and stores it in the satellite's batteries. The Power Conditioning and Control Module is an integral part of the power system and ensures that power is distributed to all other subsystems via the Distribution Board.
3. Communication Subsystem: Facilitates communication with the Ground. The most frequently utilized frequency bands in CubeSat missions include UHF, VHF, L-band, X-band, Ku-band, or Ka-band [28].
4. Attitude Determination and Control Subsystem: Determines the satellite's location and orientation and performs necessary maneuvers.
5. Payload Subsystems: The subsystems delivered to space to accomplish mission objectives, such as imaging instruments, specialized communication devices, or new technology that requires testing in space.

3.1 CubeSat Hardware Components

3.1.1 The Onboard Computer

The "flight software typically runs on radiation-hardened processors and microcontrollers that are relatively slow and memory-limited" [6]. Our satellite uses an ARM Cortex M7 microcontroller from STMicroelectronics. This microcontroller out-of-the-box supports third-party software libraries such as FreeRTOS, and FATFS, which are used in this work. Unfortunately, it has a RAM of only 1 MB, which is a limitation that must be kept in mind during the implementation. The ARM processor can be considered a System on Chip. The processor is combined with a set of peripherals on the die. These peripherals (such as UART, SPI, I^2C, and CAN) can be configured by writing to memory-mapped registers [1]. In addition, Error Correcting Codes (ECC) were applied for the RAM and the Flash Memory, where the FSW images are stored.

3.1.2 The Subsystems

The subsystems that make up our CubeSat are as follows: an Electrical Power Subsystem (EPS) that has a Distribution Board (DB) and two Power Conditioning and Control Modules (PCCM) for redundancy, A Communication Subsystem (COM) that has both UHV/VHF and S-Band, an Attitude Determination and Control Subsystem (ADCS), two Payload Subsystems; one is a 5G IoT device, and the other is a green Propulsion technology to be tested in space (PROP). Refer to Fig. 1 CubeSat Hardware Components (Subsystems) below.

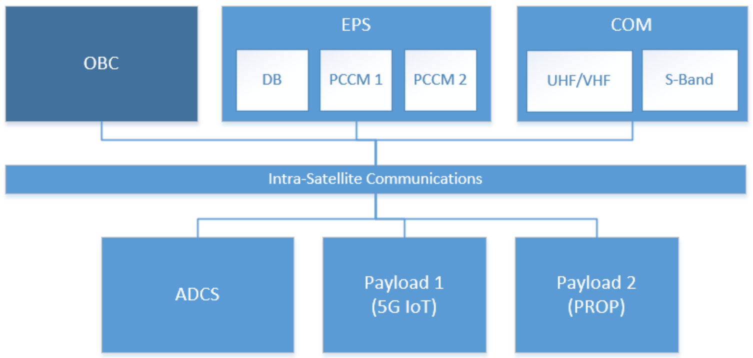


Fig. 1 CubeSat Hardware Components (Subsystems)

Regrettably, CubeSats have a poor track record in terms of mission success. This is due to a common practice among researchers and developers, who tend to prioritize subsystems and hardware design over the development of flight software and the integration of the satellite as a complete system. To address this issue, it is crucial to invest more resources into the architecture, design, and implementation of the flight software, as well as the development of fault tolerance mechanisms. [2]

4 Software Design and Architecture

The Flight Software is composed of various Applications (Apps) and supporting libraries, each with a specific purpose and function. The aim is to create a modular and service-oriented architecture, where Apps can be added or removed based on mission requirements and reused for future missions. As such, our FSW architecture can be described as layered, modular, and service oriented. The FSW architecture is illustrated in Fig. 1 below.

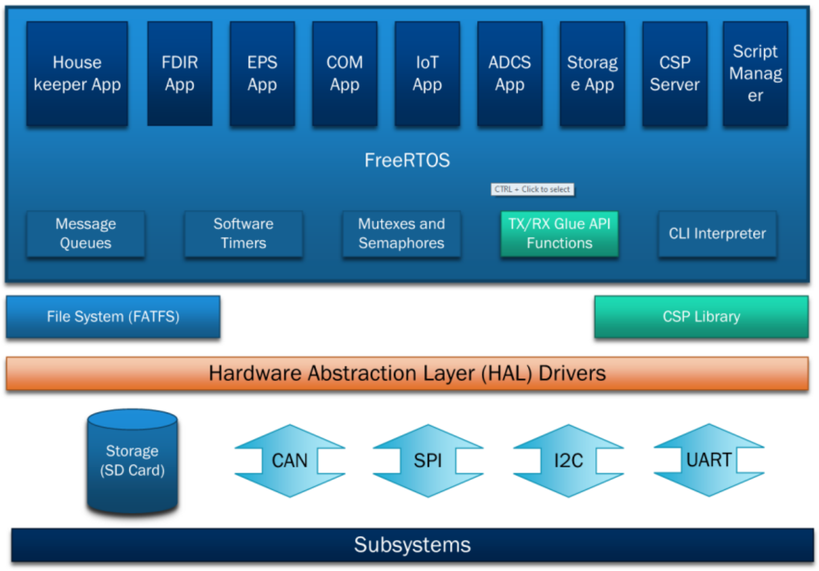


Fig. 2. Layered Software Model

The Flight Software (FSW) consists of multiple applications (Apps) and supporting libraries, each with its own purpose and function.

The **Housekeeper App** is responsible for collecting spacecraft telemetry and storing it as Whole Orbit Data (WOD) files. The Housekeeper app is critical for ensuring the overall health and performance of the spacecraft, making it an essential component of the Flight Software in a CubeSat mission.

The **Fault Detection, Isolation, and Recovery (FDIR) App** continuously monitors the telemetry and determines the spacecraft's health. The FDIR employs Watch Points (WP) and Action Points (AP). The WP subsets of telemetry are constantly monitored to ensure that they remain within safe levels for the spacecraft's nominal operation. Each WP has an AP linked to it. If a WP falls outside of the safe range, the corresponding AP is executed to recover from the detected fault. If recovery fails, the spacecraft enters a 'safe mode' until ground intervention.

The **EPS App** manages the Electrical Power Supply subsystem. In a CubeSat mission, the EPS (Electrical Power Supply) application is responsible for managing the power subsystem, which includes managing the solar panels, batteries, and power distribution system.

The **Communication Application (COM App)** is responsible for managing all communication activities of the satellite. It handles communication between the satellite and the ground station through the communication subsystem.

The **Payload Application (PLD App)** manages the payload. It provides an interface to the payload and handles the payload's data flow.

The **Attitude Determination and Control Subsystem Application (ADCS App)** governs the spacecraft's attitude by controlling the ADC Subsystem.

The **Storage App** manages and ensures the integrity of the storage unit (SD Card). It mounts and initializes a File Allocation Table File System (FATFS) volume, handles all commands related to files and file systems from the ground, and carries out file transfers (large data transfers) via Uplink and Downlink.

A **CubeSat Space Protocol (CSP) Server App** is critical to the FSW. It accepts all incoming connections to the OBC, whether the connection is coming from an onboard subsystem or ultimately from the ground via the COM Subsystems. Several dedicated CSP ports are assigned and used. One port is dedicated to receiving and processing CLI commands sent in CSP packets (CLI/CSP), which are explained in the CLI Commands over CSP (CLI/CSP) Section below.

Another very central component of our Satellite Operation is the **Script Manager App**. Details on this App are highlighted in Section 4.4 below.

4.1 Operating System

Simple embedded softwares are now evolving into more sophisticated Embedded Operating Systems. Many of these Operating Systems are available as open-source libraries that can be *ported* to various microcontrollers and hardware platforms. Our operating system of choice was the Free Real-Time Operating System (FreeRTOS), distributed freely under the MIT open-source license [12]. We selected FreeRTOS due to many reasons. One was that it has recently gained popularity, especially in CubeSat missions, as established in the Literature Review. FreeRTOS is also often chosen due to its simplicity and large user community in the field of satellites and other embedded systems [7]. Another reason was that our microcontroller natively supports it.

4.2 File System

The File Allocation Table File System (FATFS) is a basic file system module for embedded systems that is also Windows-compatible [29]. It is a software library that allows managing and organizing files on a storage device, such as a disk drive or memories. The low-level disk I/O modules are entirely separate from the FATFS module [29]. FATFS is a middle layer that allows the FSW to be written independent of the underlying media storage device drivers or host controllers. The adoption of this middle layer aligns with our intended Multilayered Architecture. FATFS was selected for this project due to its simplicity and native support by the Microcontroller and IDE. In addition, having a file system allows all apps (tasks mentioned in Section 4 above) to have the ability to record events to their designated log files.

4.3 Command Line Interface (CLI)

To simplify the process of commanding the satellite, a Command Line Interface (CLI) layer is added, which provides a layer of abstraction that allows planners and operators on the ground to control the spacecraft without having to work with the many native binary commands and responses used in each subsystem. The CLI commands introduced are high-level and human-readable, implemented with the help of the FreeRTOS-Plus Command Line Interface Framework. Examples of such commands are presented in Table 1 below.

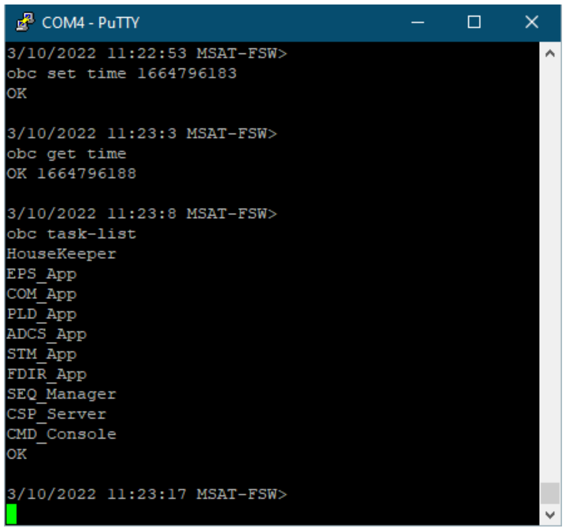
Table 1. CLI Commands and Replies

CLI Command (Input)	**Expected Reply (Output)**	**Description**
`obc get time`	`OK 1632133618`	Get tick time since EPOCH.
`obc set time 1632133620`	`OK`	Set current OBC time.
`adcs point-to nadir`	`OK`	Maneuver the Sat to Nadir.
`eps iot on`	`OK`	Power ON the IoT Payload.
`delay 15 mins`	`OK`	Delay the Script Engine for 15 mins.
`eps iot off`	`OK`	Power OFF the IoT Payload.

`msat set mode safe`	`OK`	Change the Sat mode to safe.
`boot install FSW.bin 2`	`Installation COMPLETE`	Install the FSW image to slot 2.
`boot from image 2`	`OK`	Boot from the FSW in slot 2.
`script run sequence.scr 1`	`OK`	Run the script stored in the file starting from line 1.

4.3.1 Command Console

Once CLI commands are defined, implemented, and registered with the FreeRTOS *CLI Interpreter*, they can be invoked anywhere within the FreeRTOS environment. Apps (FreeRTOS Tasks) can execute commands by sending a 'string' containing the command and any required parameter to the CLI interpreter function. The command is then processed (executed), and a reply is passed back to the caller. In light of this, A Command Console was implemented to exploit this mechanism. When testing the satellite on the ground (in a FlatSat test environment or Thermal Vacuum Chamber - TVAC Testing for the Flight Model - FM), the OBC was connected to via a serial port and a terminal. Then, the commands were typed in and executed, and outputs could be observed in real-time. This allowed us to test all of the CLI commands defined and registered. The Command Console also assisted in the Assembly, Integration, and Testing (AIT) process and in testing the operation scenarios of the satellite as a whole.



```
3/10/2022 11:22:53 MSAT-FSW>
obc set time 1664796183
OK

3/10/2022 11:23:3 MSAT-FSW>
obc get time
OK 1664796188

3/10/2022 11:23:8 MSAT-FSW>
obc task-list
HouseKeeper
EPS_App
COM_App
PLD_App
ADCS_App
STM_App
FDIR_App
SEQ_Manager
CSP_Server
CMD_Console
OK

3/10/2022 11:23:17 MSAT-FSW>
```

Fig. 3. Command Console via Serial Terminal

4.3.2 CLI Commands over CSP (CLI/CSP)

A dedicated port in the CSP Server (explained in Section 4 above) was assigned to accept incoming CLI commands strings encapsulated in CSP packets. Each command is then processed (executed), and a reply is sent back to the ground in another CSP packet as a reply. The team on the ground (using Mission Control Software) can communicate with the Satellite upon contact with an experience similar to using the Command Console just explained above.

By utilizing the CLI command set as an abstraction layer in the Mission Control Software (MCS) for communicating with the spacecraft, the MCS is no longer dependent on the specific hardware architecture of the satellite or the intra-satellite communication protocols between the On-Board Computer (OBC) and subsystems. As a result, this approach facilitates the reuse of the MCS for future missions.

4.4 Script Engine

The same CLI command set explained above can also be used in the Script Engine. A script file can be uploaded to the satellite pre-launch or upon contact. The script files are simply sequences of CLI commands optionally separated by time delays and stored in a text file. When a script file is triggered, commands are released and executed in a timely fashion while orbiting, whether the satellite is still in contact or not.

The script engine works by first having a script file marked as *'armed'*. This means that the file is now ready to be executed. Arming must be done by ground or by a previously running script. Next, the script engine reads the armed file and starts executing the (input) CLI commands stored in the script file line-by-line. Upon execution, an output file is created and filled with (output) reply lines corresponding to each CLI command executed. Upon contact, the ground team can then download and examine the output file, and new script files for subsequent satellite operation can be uploaded. At the end of each script file, another script file can also be 'armed'. Upon finishing the execution of the first one, the following script file is executed. This allows the ground team to queue or *'chain'* many script files covering satellite operations for many orbits spanning many upcoming days or weeks. This feature means satellite operations do not have to be hardcoded in the FSW. Instead, they can be defined as scripts and potentially reused in future missions.

4.5 Bootloader

A bootloader is a small piece of software that runs before the main flight software and is responsible for loading and executing the flight software. With a bootloader, a CubeSat can be updated with new software while in orbit, allowing for changes to be made to the spacecraft's operations and scientific objectives. It also allows for easier debugging and testing of new software without requiring physical access to the spacecraft. The FSW running on the OBC must be highly reliable, as maintenance is almost impossible after launch. Scenarios leading to faults and errors, and ultimately mission failure, must be predicted beforehand, and solutions to such issues should be prepared for in advance [7]. Having a bootloader can provide a safety net in case the flight software becomes corrupted or fails. The bootloader can detect a problem with the flight software and load a backup version, providing a fail-safe mechanism for the spacecraft. This makes the ability to modify/update Flight Software while in-mission a highly desirable feature. It may not directly contribute to the modularity of the Flight Software (FSW); however, it certainly adds flexibility to its design and development process and makes the FSW future-proof and thus reusable.

The bootloader in our CubeSat has four slots for storing FSW images. The first slot contains the main image (Golden Image), installed pre-launch, and can never be overwritten. Other slots may store additional FSW images as a form of software redundancy and backup. If an image is corrupt, the bootloader can boot from other images. Refer to Fig. 4 below.

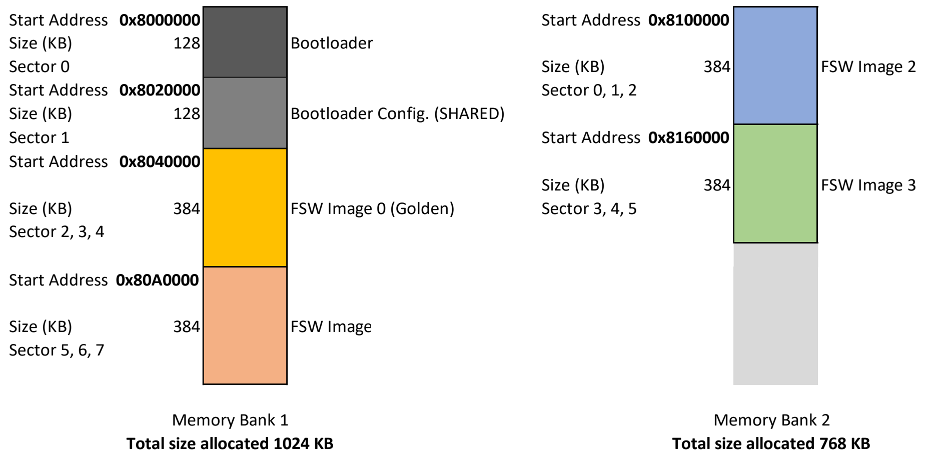


Fig. 4. Memory Organization for Bootloader in Flash Memory

New images can be uploaded to the spacecraft as a binary file during a mission. The image must then be installed in any of the other slots. The spacecraft is then instructed to boot from the newly uploaded image. The bootloader then starts by attempting to boot from the newly uplinked image. If it fails, it attempts to boot from the image the satellite managed to boot from the last time. If that also fails, it will start looking for images in other slots. In each attempt, the bootloader checks the slot area in the memory and sees if the slot is empty or not, then makes three attempts to boot from it. Finally, if all of the above fails, the bootloader defaults to the Golden Image and boots from it (refer to Fig. 5 below).

The bootloader achieves this by employing a watchdog. A watchdog is a hardware timer that monitors the operation of the system, and when a fault or an unknown state occurs, the watchdog timer will time out (after 4 seconds in our case) and it will restart the system or put the system back into a known state from which the system can recover. Watchdogs have been used in terrestrial and space systems and can improve system reliability in CubeSats [30]. When the Bootloader starts, the watchdog timer is also started, and a crash counter is incremented. Successful execution of the FSW will reset the watchdog timer and the crash counter. If the watchdog timer is not reset within a given time window, the watchdog will reset the OBC, forcing it to enter the Bootloader again to decide what to do next.

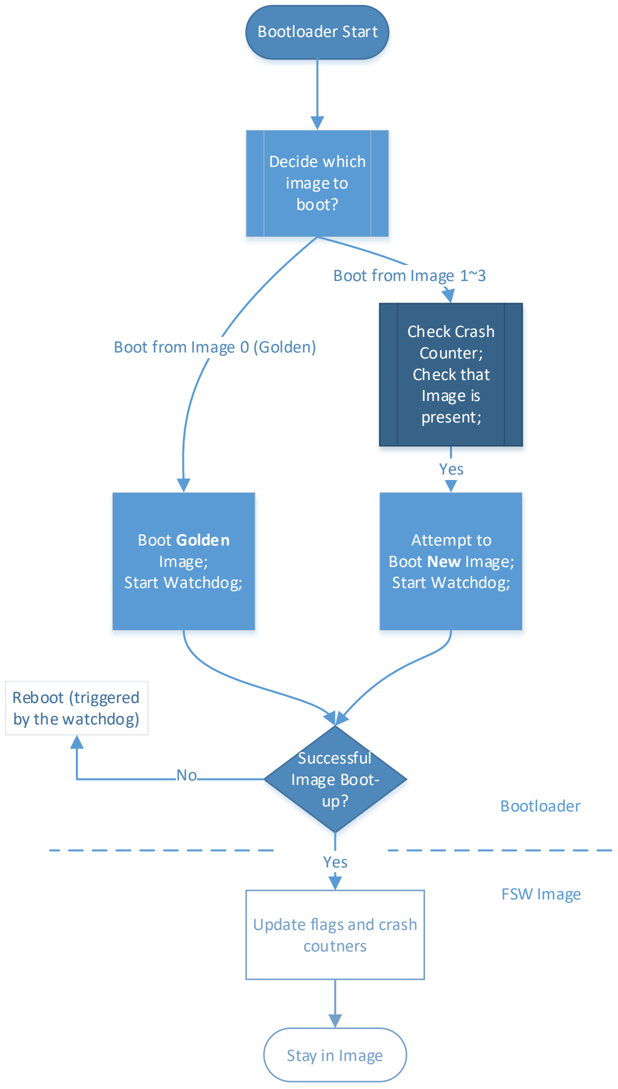

Bootloader Start
Decide which image to boot?
Boot from Image 1~3
Boot from Image 0 (Golden)
Check Crash Counter; Check that Image is present;
Yes
Boot Golden Image; Start Watchdog;
Attempt to Boot New Image; Start Watchdog;
Reboot (triggered by the watchdog)
No
Successful Image Boot-up?
Bootloader
Yes
FSW Image
Update flags and crash coutners
Stay in Image

Fig. 5 Bootloader Flowchart

5 Testing and Results

Time spent in testing accounts for more than half of the time spent on projects. The reward for finding a bug early in the implementation phase is at least a tenfold saving compared to finding it during integration time, or worse, after launch [7]. Moreover, the incredibly harsh environment of space can cause countless system failures; The ionized particles in space can trigger catastrophic software crashes and power failures. Also, extreme temperature swings may deteriorate batteries and other components [16]. CubeSats do not have an excellent mission success record [2] and still demand significant efforts to improve their robustness [16].

5.1 Test Setup

We used the Eclipse-based STM32CubeIDE for code development. STM32CubeIDE uses the GCC toolchain for development and GDB for debugging [31]. Moreover, a Version Control System is vital for such projects [8]. We used GitLab as a version-controlling tool to control software releases. We used FreeRTOS version 10.3.1, with CMSIS version 2.00. Cooperative Scheduling was used. The memory scheme used is heap 4. Generally speaking, it is not recommended to use dynamic memory in embedded systems. However, in a work that evaluates different memory schemes with the CubeSat project in mind, it was concluded that heap 4 was the best fit for its balance between the overhead code size and segmentation level. [7]

5.2 Testing Methodology

We derived test cases from the mission FSW functional and non-functional requirements. Test cases were performed, and results were recorded. Furthermore, a Long-Term Test was performed for the FSW, where the software kept running in FlatSat setup for over ten days, and all activities were logged.

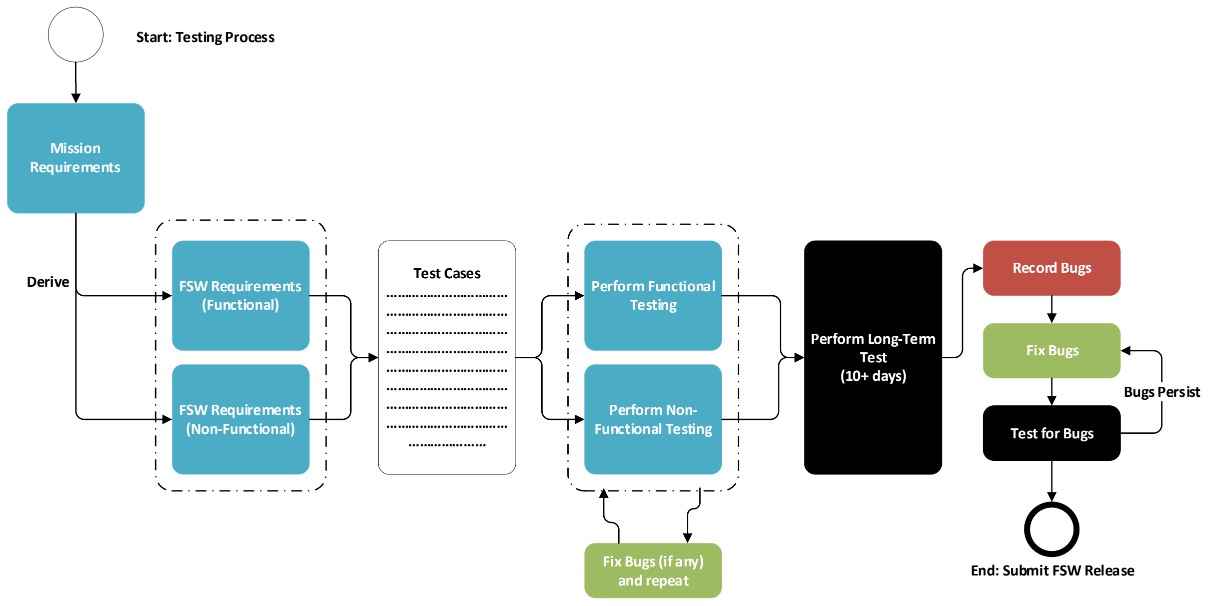

Start: Testing Process
Mission Requirements
Derive
FSW Requirements (Functional)
FSW Requirements (Non-Functional)
Test Cases
Perform Functional Testing
Perform Non-Functional Testing
Fix Bugs (if any) and repeat
Perform Long-Term Test (10+ days)
Record Bugs
Fix Bugs
Test for Bugs
Bugs Persist
End: Submit FSW Release

Figure 6 Testing Methodology

No crashes were recorded where FSW was completely stuck and unable to recover. Three types of bugs were recorded, which the FSW managed to recover from independently. One memory leak event was observed. Later, the bugs were fixed, and an FDIR WP/AP was added to rectify the situation if it ever occurred.

The following graph shows CPU usage over time for each task. This was recorded for the first 16 seconds of FSW startup. After startup, it is notable that, with all the included features in the FSW, the CPU utilization remained well below 40%.

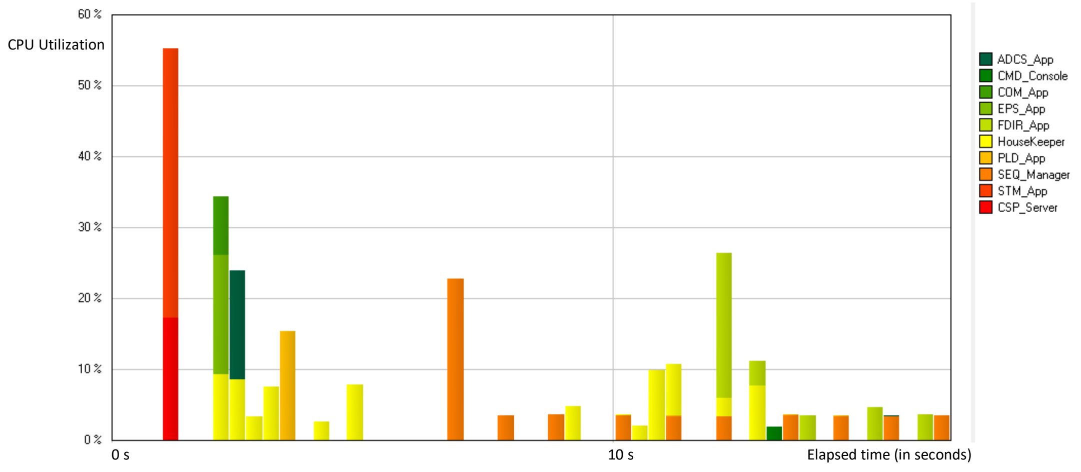


Fig. 7 CPU Utilization

5.3 *Discussions*

Our testing objective was to ensure the robustness of the software. The most important step in our testing was the long-term test that lasted for over 10 days. The idea was to keep the satellite as a whole (in a FlatSat setup) running while performing all the tasks that it is expected to perform in mission and observe the behavior. The FSW recovered from faults as expected. The Test Engineer were allowed to interact with the satellite pretending that they are in contact. However, no physical interaction (such as power cycling) was allowed with the FlatSat.

The subsequent tests and results gathered were to ensure that the OBC was not overloaded and running the FSW with ease. This is why CPU utilization was monitored as well as memory consumption. We believe faults accumulate and result in failures when systems are running at their maximum potential.

Moreover, we simply wanted to demonstrate that flight softwares for CubeSats can be packed with features (such as script engines) without compromising the performance. Modern CubeSat hardware allows for this, and FreeRTOS is lightweight indeed. Nevertheless, more time and effort should be invested on the FSW, and it has to be heavily tested.

6 Conclusions and Recommendations

CubeSats are becoming very popular. Nevertheless, the absence of a freely-available, truly modular and virtually reusable FSW, and the existence of a few frameworks and FSW development kits that require prior knowledge to fully exploit often force students and enthusiasts to develop their FSW. Achieving a truly modular, portable, and reusable FSW remains a challenge, yet this is what the scientific community should strive for.

Characteristics such as service-oriented architecture and layered software can help with software modularity to a great extent. In addition, limiting future missions to specific hardware platforms (such as the increasingly-popular FreeRTOS and, consequently, the microprocessor of choice) can drastically minimize the FSW development time in the future.

Having features in the FSW such as CLI Commands Interpreter and a Script Engine allows for the reusability of the Mission Control Systems (MCS) and Mission Operations to some extent. A bootloader is a highly recommended feature for future space engineers writing FSWs for their first mission. It is a feature that makes the FSW development process more flexible. CubeSats can be packed with such features without compromising the performance.

Employing CSP as the standard way of communication in satellites contributes to FSW's modularity. An FSW that uses CSP for all of its internal communication (intra-satellite) is deemed highly modular. Interchanging subsystems (cubes) with newer subsystems supporting CSP will reduce FSW development time. Communicating with Ground via CSP will make the FSW even more modular. We predict that the modular service-oriented architecture and use of CSP will become the de facto standard for all satellite flight software in the future.

7 Future Work

The true test of our flight software desired modularity will be the next iteration of the Payload Hosting Initiative (PHI) program, where the current FSW will be re-used in a new satellite. Having new and different payload subsystems means having new missions and thus new satellite operations. New scripts are going to be written for the new mission yet run on the same old script engine. Adding/removing/modifying apps for the new mission will also test our service-oriented FSW concept.

References

[1] D. D. Kekez, "Development of flight software and communication systems for the CanX-2 nanosatellite," 2006. [Online]. Available: https://tspace.library.utoronto.ca/handle/1807/120820. [Accessed 1 November 2022].

[2] K. V. C. K. de Souza, Y. Bouslimani and M. Ghribi, "Flight Software Development for a Cubesat Application," *IEEE Journal on Miniaturization for Air and Space Systems,* pp. 1-1, 2022.

[3] C. E. Gonzalez, C. J. Rojas, A. Bergel and M. A. Diaz, "An Architecture-Tracking Approach to Evaluate a Modular and Extensible Flight Software for CubeSat Nanosatellites," *IEEE Access,* vol. 7, pp. 126409-126429, 2019.

[4] A. K. El Allam, A.-H. M. Jallad, M. Awad, M. Takruri and P. R. Marpu, "A Highly Modular Software Framework for Reducing Software Development Time of Nanosatellites," *IEEE Access,* vol. 9, pp. 107791-107803, 2021.

[5] J. Hanson and L. J. Hansen, "Reusable, modular, and scalable flight software," in *2014 IEEE Aerospace Conference*, Big Sky, 2014.

[6] D. Dvorak, "NASA Study on Flight Software Complexity," in *AIAA Infotech@Aerospace Conference*, Seattle, 2009.

[7] M. Normann, "Software Design of an Onboard Computer for a Nanosatellite," Norwegian University of Science and Technology, 2016.

[8] S. F. Hishmeh, T. J. Doering and J. E. Lumpp, "Design of Flight Software for the KySat CubeSat Bus," in *2009 IEEE Aerospace Conference*, Big Sky, 2009.

[9] O. D. Quiros-Jimenez and D. d'Hemecourt, "Development of a flight software framework for student CubeSat missions," *TM,* vol. 32, no. 8, pp. 180-197, 2019.

[10] I. Latachi, T. Rachidi, M. Karim and A. Hanafi, "Reusable and Reliable Flight-Control Software for a Fail-Safe and Cost-Efficient Cubesat Mission: Design and Implementation," *Aerospace,* vol. 7, no. 10, p. 146, 2020.

[11] M. Tipaldi, C. Legendre, O. Koopmann, M. Ferraguto, R. Wenker and G. D'Angelo, "Development strategies for the satellite flight software on-board Meteosat Third Generation," *Acta Astronautica,* vol. 145, pp. 482-491, 2018.

[12] FreeRTOS, "FreeRTOS," Amazon Web Services, [Online]. Available: https://www.freertos.org/. [Accessed 11 July 2021].

[13] K. Lamichhane, M. Kiran, T. Kannan, D. Sahay, S. Sandya and H. G. Ranjith, "Embedded RTOS implementation for Twin Nano-satellite STUDSAT-2," in *IEEE Metrology for Aerospace (MetroAeroSpace)*, Benevento, 2015.

[14] A. E. Heunis, "Design And Implementation of Generic Flight Software for a CubeSat," December 2014. [Online]. Available: https://scholar.sun.ac.za/handle/10019.1/95911. [Accessed 31 October 2022].

[15] J. Ferreira, "ISTNanosat-1 Heart Processing and digital communications unit," Technical University of Lisbon, October 2012. [Online]. Available: https://fenix.tecnico.ulisboa.pt/downloadFile/395144736837/MScThesis-JFerreira_vFinal.pdf. [Accessed 28 July 2021].

[16] C. Araguz, M. Marí, E. Bou-Balust, E. Alarcon and D. Selva, "Design Guidelines for General-Purpose Payload-Oriented Nanosatellite Software Architectures," *Journal of Aerospace Information Systems,* vol. 15, no. 3, pp. 107-119, 2018.

[17] M. Doyle et al., "Flight Software Development for the EIRSAT-1 Mission," in *3rd Symposium on Space Educational Activities*, Leicester, 2019.

[18] National Aeronautics and Space Administration (NASA), "core Flight System (cFS)," NASA, 16 June 2021. [Online]. Available: https://cfs.gsfc.nasa.gov/. [Accessed 11 July 2021].

[19] W.-s. Choi, J.-H. Kim and H.-d. Kim, "A Study on developing Flight Software for Nano-satellite based on NASA CFS," *Journal of The Korean Society for Aeronautical & Space Sciences,* vol. 44, no. 11, pp. 997-1005, 2016.

[20] M. De la Vega-Martínez, M. C. Velázquez-García, M. F. Zavala-López, E. Hernández, R. A. Gutiérrez-Esparza, D. G. Arcos-Bravo, D. Medina, H. E. Gilardi-Velázquez and D. McComas, "Implementation of the cFS framework for the development of software in aerospace missions: first application in an undergraduate program in Mexico," in *IFAC Workshop on Aerospace Control Education WACE*, Milan, 2021.

[21] D. McComas, "Increasing flight software reuse with OpenSatKit," in *2018 IEEE Aerospace Conference*, Big Sky, 2018.

[22] D. Mitchell, "NASA - GSFC Open Source Software: OS Abstraction Layer (OSAL)," NASA's Goddard Space, 24 October 2019. [Online]. Available: https://opensource.gsfc.nasa.gov/projects/osal/. [Accessed 5 October 2022].

[23] A. Cudmore, G. Crum, S. Sheikh and J. Marshall, "Big Software for SmallSats: Adapting cFS to CubeSat Missions," [Online]. Available: https://digitalcommons.usu.edu/cgi/viewcontent.cgi?article=3303&context=smallsat. [Accessed 5 October 2022].

[24] "cFS portability on baremetal platforms," GitHub, Inc., 24 November 2021. [Online]. Available: https://github.com/nasa/cFS/discussions/374. [Accessed 5 October 2022].

[25] R. L. J. Bocchino, T. K. Canham, G. J. Watney, L. J. Reder and J. W. Levison, "F Prime: An Open-Source Framework for Small-Scale Flight Software Systems," in *Small Satellites Conference (SmallSat 2018)*, Logan, 2018.

[26] Kubos Corporation, "Porting KubOS to a New OBC," 2020. [Online]. Available: https://docs.kubos.com/1.21.0/obc-docs/porting-kubos.html. [Accessed 31 October 2022].

[27] European Space Agency, "NanoSat MO Framework," European Space Agency, 2022. [Online]. Available: https://nanosat-mo-framework.github.io/. [Accessed 29 August 2022].

[28] S. Liu, P. I. Theoharis, R. Raad, F. Tubbal, A. Theoharis, S. Iranmanesh, S. Abulgasem, M. U. A. Khan and L. Matekovits, "A Survey on CubeSat Missions and Their Antenna Designs," *Electronics,* vol. 11, no. 13, p. 2021, 2022.

[29] STMicroelectronics, "Developing applications on STM32Cube with FatFs," February 2019. [Online]. Available: https://www.st.com/resource/en/user_manual/dm00105259-developing-applications-on-stm32cube-with-fatfs-stmicroelectronics.pdf. [Accessed 11 July 2021].

[30] J. Beningo, "A Review of Watchdog Architectures and their Application to Cubesats," 2010. [Online]. Available: https://www.semanticscholar.org/paper/A-Review-of-Watchdog-Architectures-and-their-to-Beningo/3ffad01acede0bb48ff8d7cc040b1dd2079b74c3. [Accessed 8 September 2022].

[31] STMicroelectronics, "Integrated Development Environment for STM32," STMicroelectronics, 2021. [Online]. Available: https://www.st.com/en/development-tools/stm32cubeide.html. [Accessed 13 July 2021].