Flight Software Development for a University-Class Microsatellite Mission

Yumeka Nagano
Missouri University of Science and Technology

FLIGHT SOFTWARE DEVELOPMENT FOR A UNIVERSITY-CLASS MICROSATELLITE MISSION

by

YUMEKA NAGANO

A THESIS

Presented to the Graduate Faculty of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

in

AEROSPACE ENGINEERING

2023

Approved by:

Henry Pernicka, Advisor
Serhat Hosder
Daoru Han

ABSTRACT

The purpose of this research is to present the software development process used by the Command and Data Handling (C&DH) subsystem as a part of a microsatellite mission underway at the Missouri University of Science and Technology. The mission's objective is to evaluate the performance of an innovative cold gas propulsion system for use in formation flying applications and the incorporation of a stereoscopic imaging sensor used to measure the satellite pair's relative position and velocity vectors in real time. C&DH uses a Raspberry Pi 3 Model B as the flight computer running a Linux environment, and the flight code is written in C++. A State Machine controls all the actions and operations during the mission. C&DH is also responsible for handling sensor data and communicating with various systems. In this study, the communication method is established, and the interface codes are being created at the time of publication. There are various tests required for the C&DH software: unit testing, command execution testing (CET), and day-in-the-life (DITL) testing. The first is the unit testing used to validate the flight code's function. Next, CET tests all possible commands from the ground station to the integrated system. Finally, DITL is conducted to validate all functionality of the integrated system when simulating all possible scenarios expected on-orbit. This work is to establish a reference for software development of the C&DH subsystem and to provide an example that can be used to assist similar satellite programs.

ACKNOWLEDGMENTS

First and foremost, I am extremely grateful to my adviser, Dr. Pernicka, for his invaluable advice and continuous support during my master's study. His immense knowledge and plentiful experience have encouraged me in all the time of my academic research.

I thank the members of my advisory committee, Dr. Hosder and Dr. Han, for their valuable time and advice in the review of this thesis.

I am grateful to the members of the M-SAT Research Team for their constructive suggestions and guidance. I would like to thank a Chief Engineer of MR & MRS SAT, Jake Anderson, for his support and advice on my research.

I would also like to thank Ms. Wood for her technical editing support on my thesis.

Finally, I would like to thank my parents for their support throughout my Master's degree.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

The rate of small satellite launches continues to increase dramatically for a variety of mission purposes, including Earth observation, remote sensing, and communications [1]. According to the BryceTech report [2], small satellites ("SmallSats") usually include satellites with a mass less than 600 kilograms and represented 94% of spacecraft launched in 2021 (compared to 82% of spacecraft launched 2012 – 2021). Satellites continue to shrink in size; however, as technology advances, their capabilities are becoming more impactful. This trend has increased launch opportunities, not only for the space industry but also for universities, by reducing cost and size of satellites without compromising function. One advantage of small satellites is that they give college students early access to space, allowing them to gain practical experience in space science and engineering before working in the aerospace industry. One example of this is the Air Force Research Laboratory's (AFRL) University Nanosatellite Program (UNP). This program is designed to assist U.S. university students with designing, building, launching, and operating small satellites. According to the program, "UNP's objective is to train the next generation of space professionals by providing a rigorous concept-to-flight-ready spacecraft development process centered on systems engineering principles and practices."[3] The research team at the Missouri University of Science and Technology has had the privilege of participating in the UNP program since 2007.

1.1. MISSOURI ROLLA SATELLITE AND MISSOURI ROLLA SECOND SATELLITE

The Missouri S&T Satellite Research Team (M-SAT) was established by Dr. Henry Pernicka in 2001 at the Missouri University of Science and Technology. The team's first pair of microsatellites, Missouri-Rolla Satellite (MR SAT) and Missouri-Rolla Second Satellite (MRS SAT), are nearing the end of their development phase. The MR & MRS SAT project's objectives are to test cutting-edge technologies applicable to SmallSats. These tests include the investigation of an innovative cold gas propulsion system for use in formation flying applications and the incorporation of a stereoscopic imaging sensor used to measure the satellite pair's relative position and velocity vectors in real time. The M-SAT team was accepted into AFRL's UNP Nanosatellite-8 (NS-8) student competition in December 2012 with an official start date of January 2013. NS-8 was sponsored by AFRL and the protoflight spacecraft was completed by January 2015 for a final competition review with nine other universities. As of the publication date of this thesis, M-SAT continues to work with AFRL until all subsequent reviews are passed and the satellite is launched.

1.2. SOFTWARE DEVELOPMENT FOR UNIVERSITY SATELLITE TEAM

The author has been involved in developing the flight software for the MR & MRS SAT mission as a member of the Command and Data Handling (C&DH) subsystem. C&DH controls all autonomous actions made by the satellite and all uplinked or downlinked communications between the spacecraft and the ground station. In addition, C&DH handles all sensor data for the spacecraft, including those from the inertial measurement unit (IMU) and thermal sensor. For a mission to be successful, flawless command and data handling are essential. Flight code programming is the primary project of C&DH. For university students, there is a number of factors that make software development challenging, for example: (1) Team members are frequently replaced when they graduate. Due to the frequent turnover rates, students require a longer period of time to learn good coding practices. (2) There are often no consistent standards for writing code within the team. As a result, flight codes can lack adequate descriptions and are not always well-organized.

The general software development process using the M-SAT's satellite program as an example and the author's involvement are described in this paper. The primary objectives are to assist future students in the following ways: (1) Describe the current code structure and how it works. (2) Provide an organized instructional resource for developing flight codes.

2. LITERATURE REVIEW

2.1. COMMAND AND DATA HANDLING

All electronic subsystems, parts, instruments, and functional elements are referred to as Small Spacecraft Avionics (SSA), which are used with the flight subsystems Command and Data Handling (C&DH), Flight Software (FSW), and other critical flight subsystems [4]. The core integrated avionics system for controlling the spacecraft is C&DH, which includes command and telemetry processing, real-time control systems that use sensor inputs to determine the status of the spacecraft, network management, and data storage systems [5].

2.2. FLIGHT COMPUTER SELECTION

The selection of the flight computer greatly influences the avionics architecture design, hence the selection is critical for the C&DH subsystem. Several open-source hardware platforms show potential for small spacecraft systems [4]. According to the Open Source Hardware Association [6], "Open source hardware is hardware whose design is made publicly available so that anyone can study, modify, distribute, make, and sell the design or hardware based on that design." Small satellite researchers usually require low-cost solutions to achieve their missions. The rise of open-source and low-cost technologies, such as microcontrollers, 3D printers, sensors, and actuators, has led to an increase in the use of open-source hardware [7].

The Raspberry Pi is a high-performance, open-source, low-cost hardware platform, and its effectiveness has been demonstrated as a processor for satellites [8], [9]. The Arduino microcontroller is another open-source electronics platform, which has user-friendly hardware and software attributes [10]. The Arduino software is especially easy-to-use for beginners, yet flexible enough for advanced applications. One application using Arduino is

Ardusat, which contains a set of Arduino boards and sensors. With the full Arduino sensor suite on board, Ardusat provides the user the chance to design their own satellite experiments and obtain actual space data using the Arduino open-source prototyping platform [11].

2.3. FLIGHT SOFTWARE

According to [4], fundamentally, the FSW controls the spacecraft and sends commands to all systems to handle all tasks required for the mission. These tasks include all science objectives as commands to keep the spacecraft's functionality and ensure data storage and communication of the data. The FSW should contain all software operating on the various subsystems and payloads in addition to the C&DH avionics systems.

Primary considerations for FSW are the following:

1. Operating system (OS): controls computer hardware and software resources and offers standard services for programming [4]
2. Software language: creates a common format of instructions that can be translated from what humans can understand to a code recognized by a machine, which is written in binary [12]
3. Version control tool: coordinates work among programmers, a version control system is used to track changes [13]

2.3.1. Operating System.

Linux is a general-purpose operating system used in embedded devices. Linux is widely used because of its abundant hardware support, implementation of communication protocols and application programming interfaces, availability of development tools, attractive license terms, vendor independence, and affordability [14]. Linux has been used in various spacecraft, including many by Planet (Dove satellites) and SpaceX (Falcon 9 and Dragon) [15].

FreeRTOS is another example of a satellite OS. A real-time operating system (RTOS) is an OS that manages data and events with strictly defined time restrictions for real-time computing applications. FreeRTOS is open source and released under the Massachusetts Institute of Technology (MIT) license and is built with an emphasis on reliability and ease of use [16]. FreeRTOS is used in many satellite missions because of its core real-time functionalities [17], [18].

2.3.2. Programming Language. The major programming languages in the space-flight field are C, C++, Python, and Matlab. NASA has provided various open-source codes with these languages, including simulation tools and flight software [19]. Many space companies and administrations, such as SpaceX and NASA, have adopted C++ in Linux environments in order to access the sizeable Linux developer community [15].

2.3.3. Version Control Tool. Git is one of the most common version control tools for software development. It allows programmers to track all changes, and the code is mirrored on every programmer's computer [13]. GitHub and GitLab are common Internet hosting services, which allow programmers to serve content or host services connected to the Internet, using Git.

2.4. CODING GUIDELINES FOR SPACECRAFT

Flight code programming is the primary purpose of C&DH, and there are many considerations that researchers have to take into account to establish coding best practices. Therefore, most critical software development projects, such as spacecraft flight codes, have coding guidelines. These rules are intended to outline the basic rules for writing the program, including how it should be organized and which language features should and should not be used. According to [20], there are ten rules for developing safety-critical code:

1. Control flow constructs must be simple. Simple code flow is beneficial for verification and often results in improving code clarity for readers.

2. All loops must have a fixed upper bound. A loop must not continue forever and needs to have a maximum number of iterations or a time condition to end the loop.
3. Dynamic memory allocation must not be used. When a variable is declared, programmers must define a specific size of memory for the data. While some languages are able to handle data even if the size is not set, it will reduce runtime if the size of the data is defined beforehand.
4. No function should be unnecessarily long. Typically, one function should be less than 60 lines of code. The rule suggests that a function is doing too much if it runs beyond the screen. By breaking up the code into several small segments, it becomes simpler to read, comprehend, and debug.
5. Functions should have at least two assertions. Assertions are used to check if pre- and post-conditions of functions, parameter values, function return values, and loop invariants are valid states and not null. If it is not a valid state, error messages should be returned.
6. Data objects must be declared with the smallest possible scope. This minimizes the opportunity for other code to corrupt or misuse the data objects.
7. Incoming parameters and return values of functions must be checked to ensure they are in a valid state that is expected.
8. Only header files and simple macro definitions can be included as the preprocessor. Preprocessors are used before the main code executes, and this rule discourages large macro development to avoid developers spending additional time tracing and understanding the code.
9. The use of pointers should be limited. Pointers are essentially used to store the memory address of variables. Instead of passing actual data structures and objects, pointers are passed around in functions, which uses less memory. However, pointers can easily be misused, making it difficult to follow and analyze the flow of data in a program.

10. All codes must be compiled without any warnings. This rule requires that the code should be in a state that is ready to execute and deploy in a production environment.

The NASA Jet Propulsion Laboratory (JPL) used these rules for writing critical mission programs and showed encouraging outcomes. In addition, these rules can simplify the process for the developer and tester to establish the important elements of the programs.

2.5. SOFTWARE STRUCTURE

Flight software structure can often be illustrated by a layered architecture [21], [22]. The lowest layer has simple functions to access hardware interfaces. The next layer includes functions to communicate with various subsystems. Managing various data is done in the following layer. The processes, features, and drivers in the lower layers of the flight software structure are compiled in the top layer. The functions in the lower layers are required for all spacecraft, and the functions become mission-unique toward the upper layers.

2.6. MISSION CONTROL

A state machine is a mathematical model that transitions between states in response to certain inputs [23], and it can be used to control mission flow for flight software. There are several control approaches for the state machine. One is a centralized control that uses a single control component to execute state machine functions [24]. This approach is well suited for small spacecraft because the control simple logic can be easily understood and maintained. Another approach, hierarchical control, uses multiple control components to execute functions [24]. This approach is better suited for larger spacecraft because it can avoid creating bottlenecks.

2.7. INTEGRATION OF SOFTWARE AND HARDWARE

In order to determine the next action that a spacecraft will take, the C&DH subsystem must obtain data from various hardware systems, and the flight computer must send commands to each interface. Communication protocols are used to communicate between a flight computer and each interface. The protocols allow data to be sent from each sensor to the flight computer and commands to be sent from a flight computer to each system. The main communication protocols that can be used are described in the following sections.

2.7.1. Inter-Integrated Circuit.

Inter-Integrated Circuit (I^2C) is a simple communication protocol used to transfer data between a controller and peripherals and is a duplex two-wire serial bus using serial clock (SCK) and serial data (SDA) wires to send and manage data as shown in Figure 2.1. Each peripheral has an address so that the controller can send commands to a specific peripheral and identify from where the data originated. When multiple devices are added to the same bus, I^2C can reduce the complexity of the circuit [25].

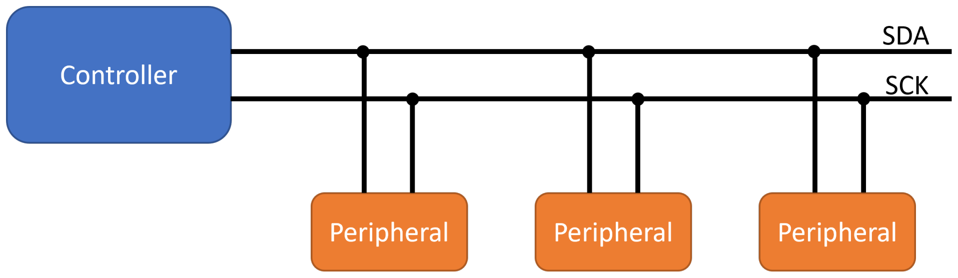


Figure 2.1. I^2C communication protocol [26]

2.7.2. Serial Peripheral Interface. The Serial Peripheral Interface (SPI) is used for high-speed data exchanges between devices on the bus. A serial clock (SCK), a peripheral in/controller out (PICO), a peripheral out/controller in (POCI), and a chip select (CS) signal form the minimum set of signals required. All devices on the bus share the SCK, PICO, and POCI signals. The controller device generates the SCK signal for synchronization, while the PICO and POCI lines are employed for data communication. Each peripheral device that is added to the bus also has its own CS line. The structure of the SPI communication protocol is shown in Figure 2.2. The protocol allows faster communication than I^{2}C; however, it requires at least four lines, which can make the system complicated [27].

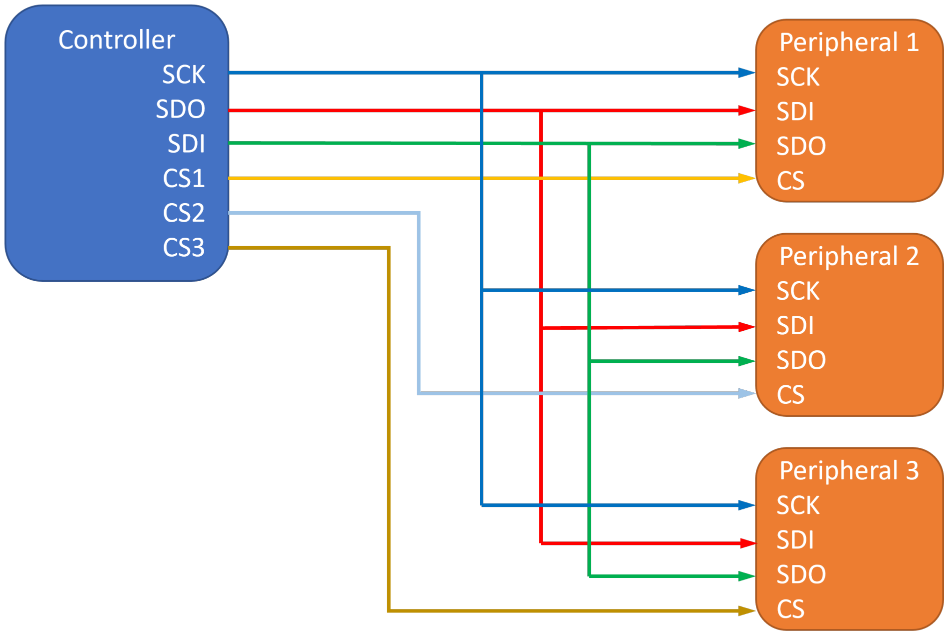


Figure 2.2. SPI communication protocol [26]

2.7.3. Universal Asynchronous Receiver/ Transmitter. A universal asynchronous receiver/transmitter (UART) uses only two wires to transmit and receive serial data. A transmitter (Tx) pin is used to send data, and a receiver (Rx) is used to receive the data as shown in Figure 2.3. The UART protocol allows faster transmission than SPI, although it is not possible to communicate with multiple peripherals [28].

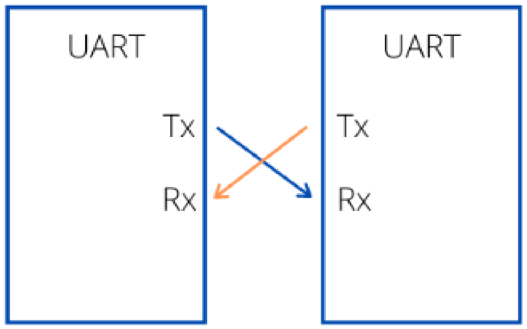


Figure 2.3. UART communication protocol; Source: [26]

2.7.4. 1-Wire. The 1-Wire bus is similar in concept to I^2C. According to [29], 1-Wire is the only voltage-based digital system capable of half-duplex bidirectional communication using data and ground connections. A 1-Wire system consists of a single controller and one or more peripherals. In a 1-Wire connection, a controller device initializes digital communication as well as self-timing peripherals that synchronize to the controller's signal.

2.7.5. Universal Serial Bus. A universal serial bus (USB) is a common interface that allows communication between different peripheral devices. The communication protocol is well-known and simple, but the functionality is limited [30].

2.7.6. Transmission Control Protocol. The Transmission Control Protocol (TCP) is designed to be used as a highly reliable host-to-host protocol to transmit packets by using Internet Protocol (IP). TCP has taken over as the Internet's primary connection-oriented, stream-based transport protocol. It is commonly utilized by common applications and frequently implemented by endpoints [31].

2.8. VERIFICATION AND VALIDATION

According to [32], software verification is testing to check that the code performs as desired, and validation is testing to prove that the verified code meets all requirements for the mission. The V-shaped software life cycle diagram in Figure 2.4 is often used to represent software verification and validation. The software development process starts at the top left corner, indicating that the development of the software requirements is the first phase. The functional requirements for the program are broken down into a software architecture, which is where the detailed software design and code are ultimately created. Software verification demonstrates the accuracy of the requirements progressive decomposition into the software code. The blue arrows represent the verification testing, and the red arrows represent the validation testing at each step. At the lowest validation level, each software unit is tested (unit testing), and the tests are continually performed until all the software units have been successfully tested. The code is then integrated into the full code and tested (integration testing). The final step of software testing is validation testing to check that the software functions correctly on the target platform.

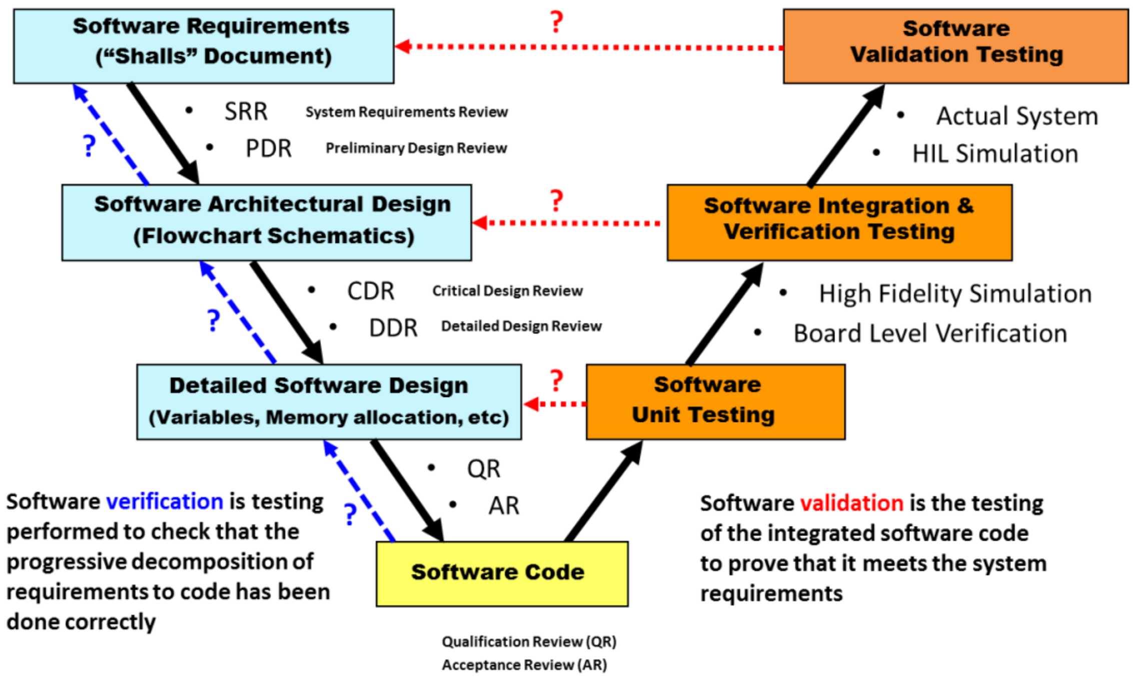

Software Requirements ("Shalls" Document)
Software Architectural Design (Flowchart Schematics)
Detailed Software Design (Variables, Memory allocation, etc)
Software Code
Software Unit Testing
Software Integration & Verification Testing
Software Validation Testing
SRR System Requirements Review
PDR Preliminary Design Review
CDR Critical Design Review
DDR Detailed Design Review
QR
AR
Actual System
HIL Simulation
High Fidelity Simulation
Board Level Verification
?
Software verification is testing performed to check that the progressive decomposition of requirements to code has been done correctly
Software validation is the testing of the integrated software code to prove that it meets the system requirements
Qualification Review (QR)
Acceptance Review (AR)

Figure 2.4. Software development lifecycle, verification and validation; Source: [32]

2.8.1. Unit Testing. Unit testing is the first step for software validation, as mentioned. This is a function-level test to check if each separate unit of the software meets its requirements. The unit test should be modified when a discrepancy or bug in the program is fixed, and a record of all the tests should be kept for reference [4].

2.8.2. Command Execution Testing. Command execution testing (CET) is performed to validate every command that may be uplinked from the ground station to the spacecraft. The FSW will be operating on the integrated satellite during the test, and the hardware will be reacting to uplinked commands. The CET tests all of the software and electrical interactions between all of the integrated satellite's components, in addition to finding any bugs or flaws in the flight software [22].

2.8.3. Day-in-the-Life Testing. Day-in-the-life (DITL) testing validates that satellite software is nominally functional, and that the full integration of hardware and software can perform all possible scenarios during the mission [33]. As stated in [34], mission scenarios are simulated, and commands are generated in the planning software. Those commands are uplinked from the ground station to the satellite and executed automatically with the satellite in a similar state as expected on-orbit. The results, including sensor data, are downlinked at the end of the scenario.

3. CONCEPTUAL DESIGN OF COMMAND AND DATA HANDLING SYSTEM

The C&DH subsystem is analogous to the central brain of the MR SAT spacecraft and is responsible for handling data and issuing commands to different subsystems. All data for functions are processed through the C&DH subsystem. C&DH controls all mission flows and the decision-making capabilities of the spacecraft.

3.1. HARDWARE AND SOFTWARE SELECTION FOR MR SAT

Software code and programs are thoroughly integrated with the hardware, requiring careful selection and implementation. Primary considerations for the C&DH subsystem are the flight computer, operating system (OS), and software language.

MR SAT's flight computer must satisfy the following general requirements [35]:

- Ability to execute complex calculations quickly
- Provide large data storage
- Ability to communicate over multiple data buses
- Affordable
- Small
- Minimal power draw

Open-source hardware platforms show potential for small spacecraft systems as mentioned in Section 2.2. Therefore, the Raspberry Pi 3 Model B was chosen to serve as the flight computer for MR SAT. This model can store large amounts of data on a micro SD card. Another attractive feature of the Pi 3 is the abundance of readily available pinouts and data connectors. These connectors allow the Pi 3 to take advantage of different data communication protocols at the same time. Considering its benefits and record of success as mentioned in Section 2.3, Linux was chosen for the MR SAT OS. Most of MR SAT's flight software for C&DH was created using C++ that supports object-oriented programming, with the exception of the microcontrollers (MCUs), which were written in C.

3.2. HARDWARE AND SOFTWARE SELECTION FOR MRS SAT

The MRS SAT spacecraft is a non-cooperative RSO with no control over its orbital or attitude motion. MRS SAT utilizes an ATMega644A MCU as the primary flight computer. This MCU was selected for its simplicity and interface options, as well as the familiarity that the team has with the hardware. The MRS SAT MCU flight code was written in C.

3.3. CODING GUIDELINES

Coding guidelines outline the basic rules for writing the flight code, including how it should be organized and which C++ language features should and should not be used. The following are the guidelines for MR SAT's flight code:

1. Functions should be used to organize the code. It is possible for the flight code to be written with only one large main function; however, in this case, it is then difficult for developers to track, debug, and test such code. Therefore, individual functions should be created for repeated tasks, and each function's lines of code should not exceed one screen page (at a standard font size).
2. The main control flow should be simple. The State Machine controlling the mission flow only has one switch case statement to transition to the eight different mission modes.
3. All loop statements must have a time limit or maximum iteration number to avoid an infinite loop.
4. Appropriate data object type and size should be defined. Using more data memory than necessary increases code run-time. Also, memory is limited, particularly for uplinking and downlinking, so choosing proper data types and sizes is important.
5. "Namespaces" must be used to differentiate similar variables in different libraries. For example, the IMU variables are declared in the health and power namespaces. HEALTH::IMU is used for the health status of IMU, and POWER::IMU is used for the power status of IMU.

6. A “class” or “struct” should be used to structure different data and functions. For example, multiple data on multiple thermal sensors are needed to be stored, such as thermal sensor ID, temperature reading, and thermal sensor location. In class or struct, each datum must be declared as a private object to prevent accessing the object from outside of the structure, and functions should be declared as public to access the object member in the structure.
7. “Const” should be used for data that are not changed in the entire code, such as the number of thermal sensors used and the temperature tolerance range. A const variable must be notated in all uppercase to distinguish it from normal variables.
8. “Static” should be used for data that can exist only one at a time and for functions that only need to access static variables. Using static member variables can prevent having existing several statuses for one object. For example, only one current mission mode can exist at a time.
9. The code must contain thorough comments. For each function, the function, input, and output description should be commented on before the function declaration. Within each function should be a description of each line and the to-do list for future work must be clearly given. These comments can help team members understand the code.

3.4. DEVELOPMENT PROCESS

The M-SAT team uses GitLab to track all changes made in the software and workflow, as shown in Figure 3.1. In this workflow, the master branch always reflects a ready state. Whenever a member is assigned a task in the GitLab issue list, they will create a new branch corresponding to the issue to work on the task. Once the task is completed and tested, the subsystem lead will review the code and, if required, make suggestions for changes. Then, the lead will merge the respective branch into the development branch of the project. The development branch will then be tested with a prior merge. Once the development branch's code is determined as stable by the lead, the development branch will be merged into the master branch.

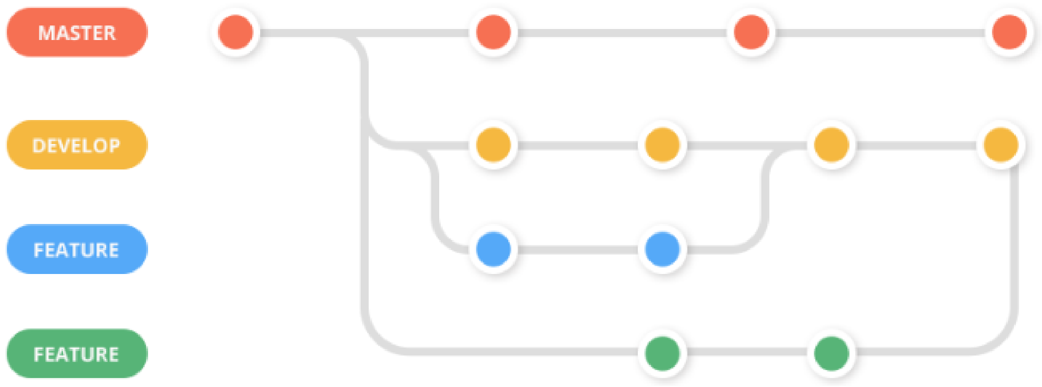


Figure 3.1. Git workflow with features and develop branch; Source: [36]

3.5. MISSION MODES

The MR & MRS SAT mission will simulate an inspector satellite performing proximity operations about a non-cooperative RSO. During the mission, the MR SAT spacecraft will serve as the inspector spacecraft, and MRS SAT will serve as the non-cooperative RSO. Image data analyzed and stored during the main mission modes will be used to estimate the position of MRS SAT relative to MR SAT. These images will also be used to generate a 3-D reconstruction of MRS SAT, providing the means to characterize the RSO. The minimum success criterion of the mission is defined as "The stereoscopic imager must acquire and store an image pair after separation of MRS SAT as confirmed through boolean verification in beacon data."[37] The necessary high-level mission operations during the MR & MRS SAT mission are shown in Figure 3.2. In brief, MR SAT and MRS SAT will launch together to the International Space Station (ISS). The satellite pair will initialize in a low power mode after separation from the ISS and "detumble" their attitude. The satellites will initialize to a fully operational condition and carry out the necessary health checks once the nominal attitude has been attained. MRS SAT will undock from MR SAT, and the assigned mission modes will start, contingent on the successful system checkouts of each subsystem and the uplinking of a separation command. Each mission mode is defined in the following subsections [37].

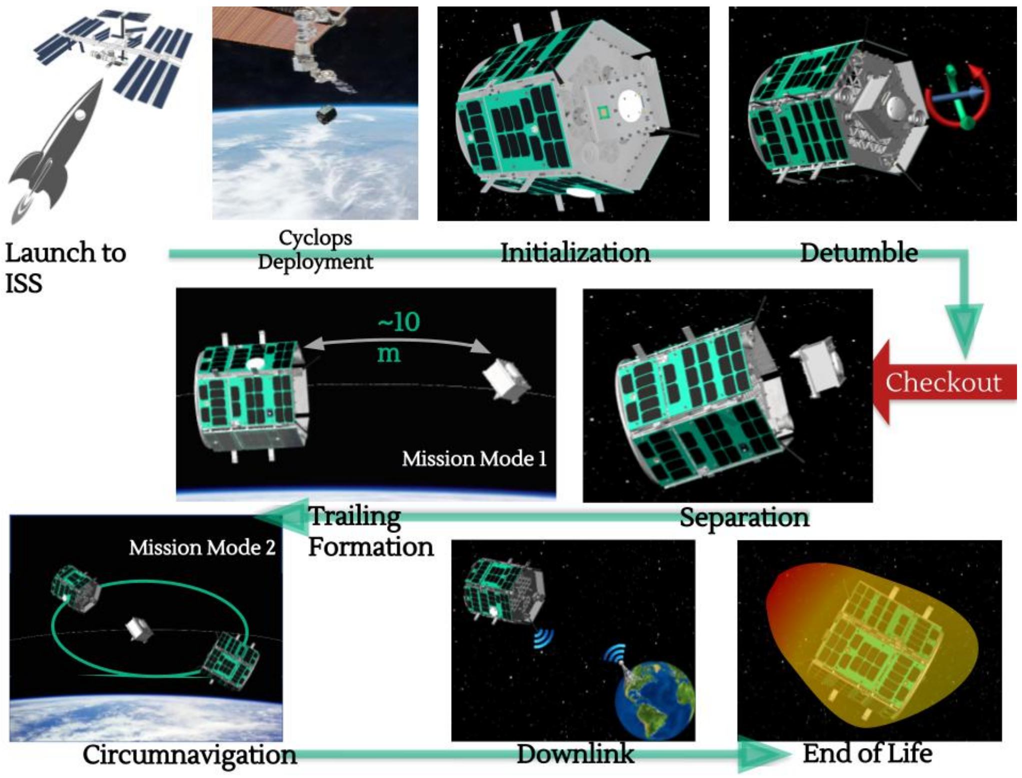


Figure 3.2. MR & MRS SAT mission flow; Source: [37]

3.5.1. Cyclops Deployment. The mated satellites are deployed from the ISS during the Cyclops Deployment Mode, which lasts from the time the satellites are integrated into the Cyclops deployment mechanism to the time the mated satellites are separated from the ISS. The mated satellites will be entirely powered down when in launch mode, and inhibitors will be in place to prevent any electrical current from the launch vehicle or other payloads from reaching the MR & MRS SAT spacecraft.

3.5.2. Initialization. The operations performed during the Initialization Mode are shown in Figure 3.3. The MR SAT spacecraft's inhibits are actuated once it separates from the ISS, and the solar arrays will then begin charging the batteries. At this point, the MR SAT's Electric Power System (EPS) will power up, enabling charging of its secondary batteries. The onboard Raspberry PI flight computer will automatically turn on once the EPS is powered and start a 45-minute timer that enforces radio silence. After the 45-minute silence, the MR SAT radio will power on and attempt to establish a link with the ground station while the flight software will begin powering on subsystems and executing commands. The MRS SAT spacecraft will power on and begin transmitting position data to the Iridium network via its Global Positioning System (GPS) antenna and Eyestar radio. After MR and MRS SAT are enabled, the flight computer will power on systems in the following order: GPS MCU, GPS antenna, GPS receiver, IMU, analog board, magnetometer, and Sun sensor boards. With the MR SAT sensors and Guidance Navigation and Control (GNC) system powered, the navigation filter can be initialized, and position and attitude can be determined. A systems health check will then be performed, and MR & MRS SAT will enter into the Detumble Mode. If MR SAT is unable to resolve any anomalies during the health check, the Checkout Mode will be activated.

3.5.3. Detumble. The operations performed during Detumble Mode are shown in Figure 3.4. Once the spacecraft pair enters Detumble Mode, the GNC system will execute detumble maneuvers. The GNC system will use three torque coils on MR SAT to control the satellite's attitude. The torque coils will be actuated for a brief period, followed by a

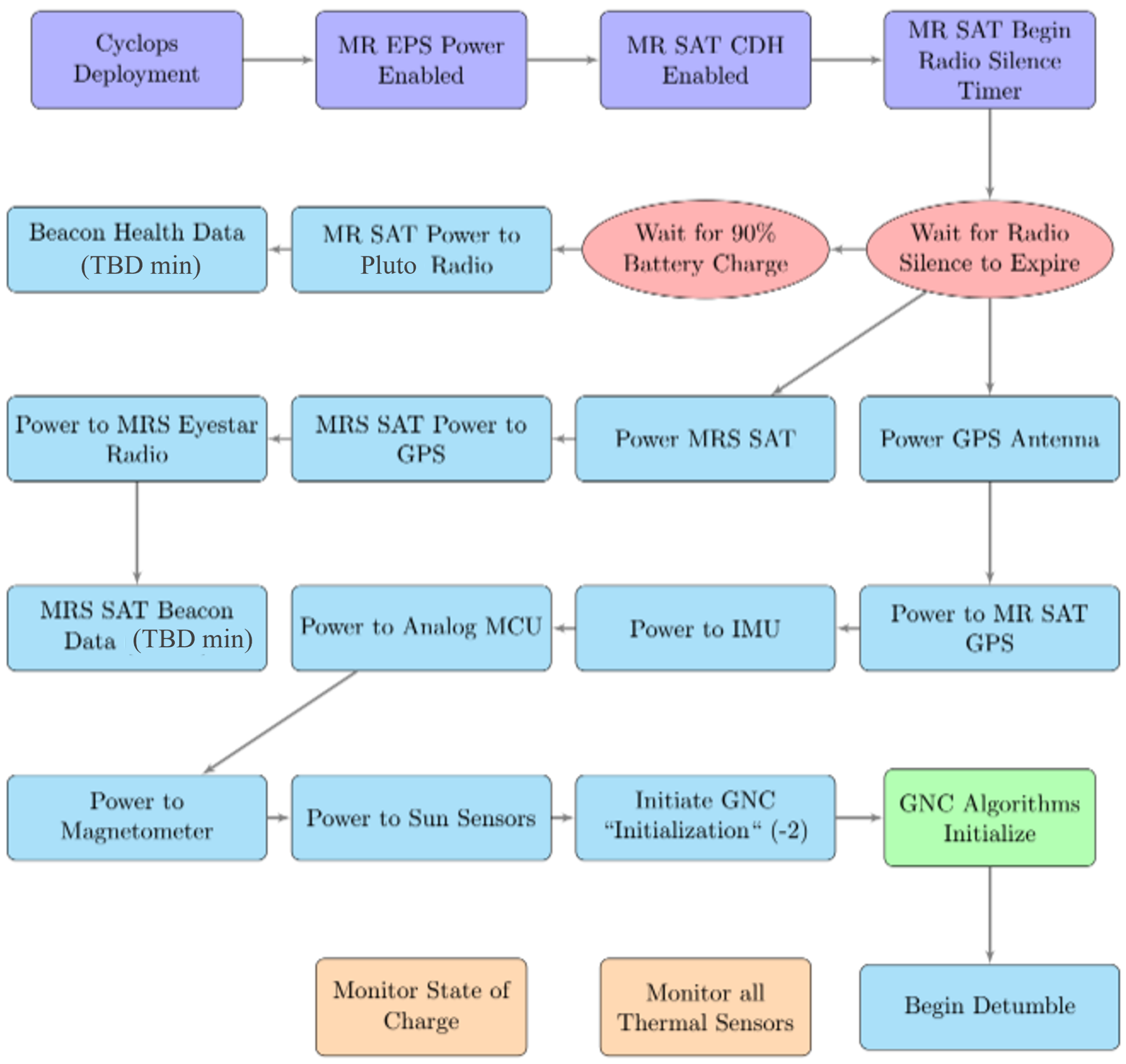

Cyclops Deployment
MR EPS Power Enabled
MR SAT CDH Enabled
MR SAT Begin Radio Silence Timer
Wait for Radio Silence to Expire
Wait for 90% Battery Charge
MR SAT Power to Pluto Radio
Beacon Health Data (TBD min)
Power GPS Antenna
Power MRS SAT
MRS SAT Power to GPS
Power to MRS Eyestar Radio
MRS SAT Beacon Data (TBD min)
Power to MR SAT GPS
Power to IMU
Power to Analog MCU
Power to Magnetometer
Power to Sun Sensors
Initiate GNC "Initialization" (-2)
GNC Algorithms Initialize
Begin Detumble
Monitor all Thermal Sensors
Monitor State of Charge

Figure 3.3. Initialization block diagram; Source: adapted from [37]

shutoff period during which the magnetometer readings and Sun sensor data will update MR SAT's attitude. This process is repeated until the nominal attitude is achieved. If the nominal attitude is not achieved within seven days, the satellite will enter Checkout Mode.

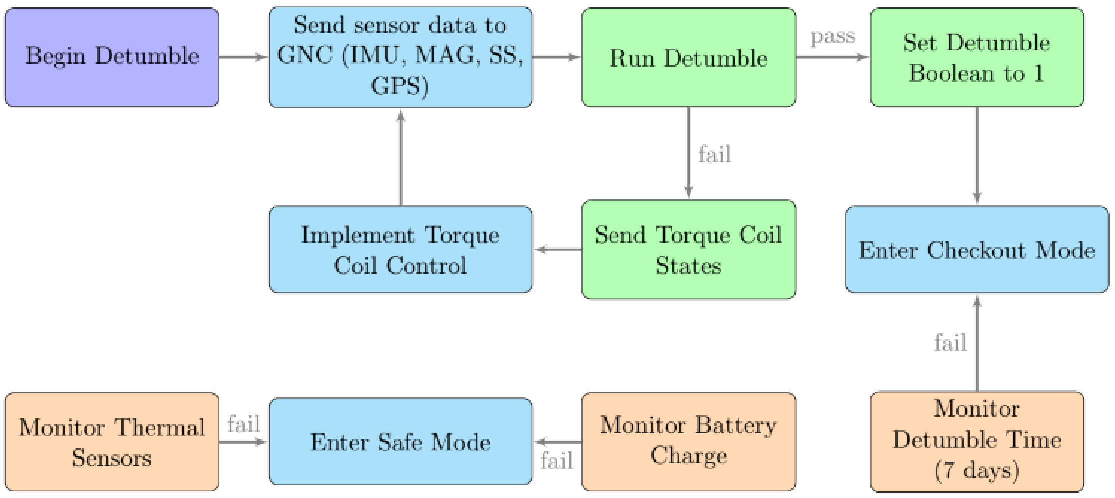


Figure 3.4. Detumble block diagram; Source: [37]

3.5.4. Checkout. The operations performed during the Checkout Mode are shown in Figure 3.5. Checkout Mode is intended to allow M-SAT engineers time to debug systems as needed. In this mode, MR SAT will maintain its attitude to preserve the alignment of the stereoscopic imager plane normal to the velocity direction. The spacecraft will wait for a command from the ground station to enter Separation Mode.

3.5.5. Separation. The operations performed during the Separation Mode are shown in Figure 3.6. The Separation Mode prepares the satellite for the proximity operations portion of the mission and performs one final health check. This mode is critical because there can be no more attempts at proximity operations once separation occurs, as it places the satellite in a power-negative state. It will systematically power on the final required systems (digital board, imaging computer, and stereoscopic cameras), perform a health check, begin physical separation, and then enter into Mission Mode 1. If the separation mechanism does not function correctly, the imaging system and digital board will be turned off, and the spacecraft will thereafter reenter Checkout Mode. If the Digital board fails a health check or connection to the imaging computer, the paired satellites will

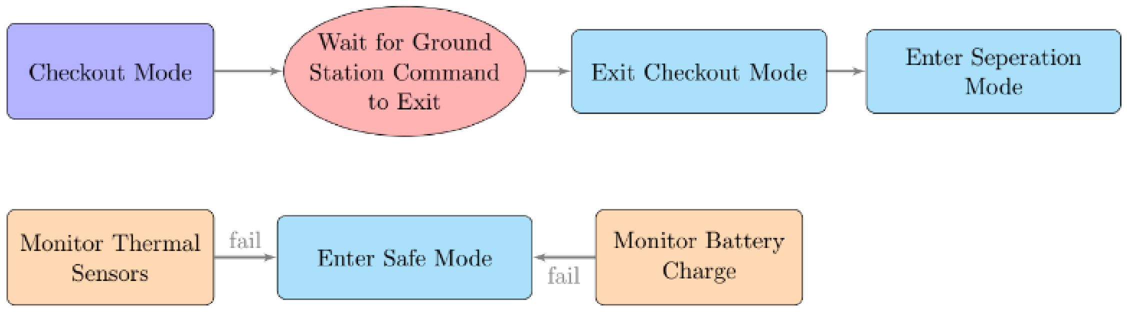


Figure 3.5. Checkout block diagram; Source: [37]

autonomously return to the Checkout Mode to allow ground station engineers to assess the anomaly. Once the diagnosis is complete and any error corrections are made, the M-SAT engineers will wait again for an ideal pass to reenter Separation Mode.

3.5.6. Mission Mode 1: Trailing Formation. The operations performed during Trailing Formation Mode are shown in Figure 3.7. MR SAT's primary mission is the proximity operation, which is executed by using the stereoscopic imaging system to estimate MRS SAT's relative position. MR SAT will nominally be positioned approximately 10 m behind MRS SAT in a trailing formation using MR SAT's propulsion system. The spacecraft will enter Mission Mode 2 if MR and MRS SAT maintain the appropriate trailing formation distance of 10 m +/- 2 m for at least five minutes. If a continuous five-minute period of proper trailing formation distance is not maintained, MR SAT will autonomously switch to Mission Mode 2 after 45 minutes. The spacecraft will autonomously switch to Downlink Mode if the battery charge or propellant reaches a critical value before 45 minutes. The position of MRS SAT will be determined by the GPS receiver on board MRS SAT. MRS SAT's Novatel GPS will beacon GPS data (with an error of approximately 2.5 m) and the supplementary uncertainty for future post-flight processing on the ground. The Iridium

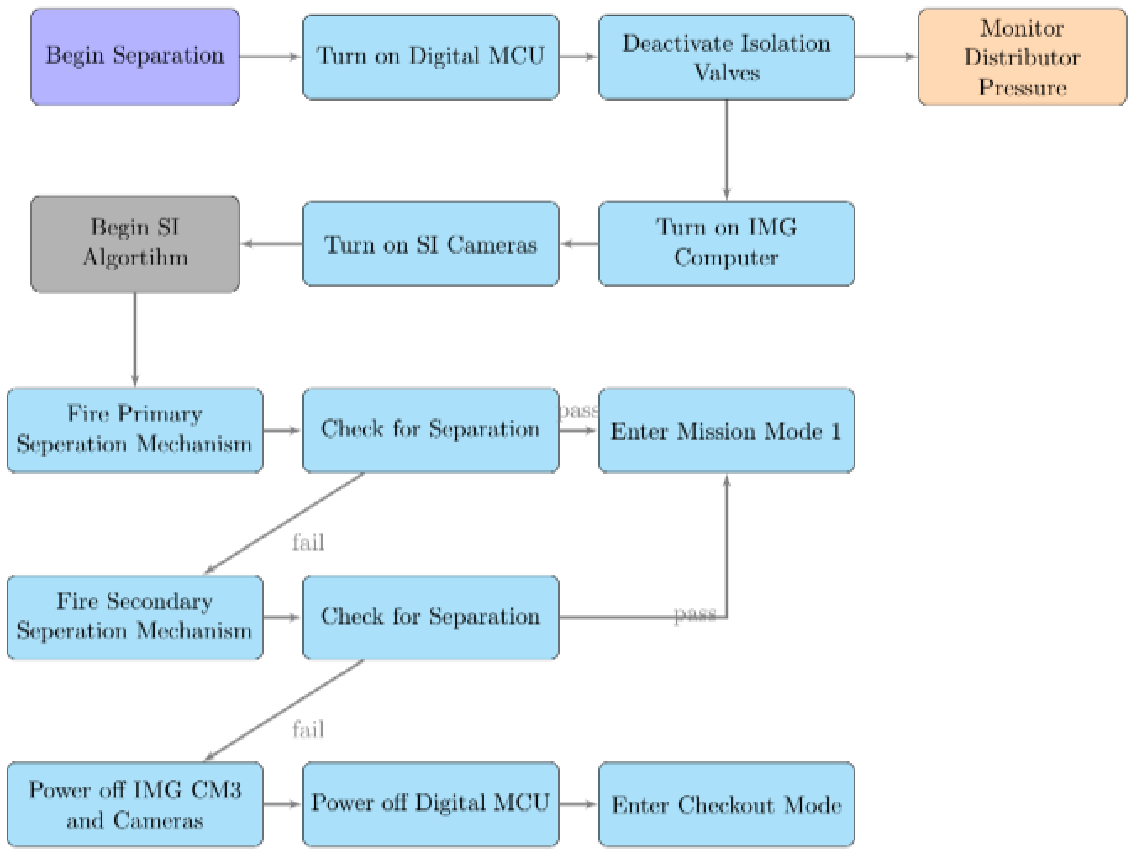


Figure 3.6. Separation block diagram; Source: [37]

network will receive and downlink these data, which will then be used to analyze and verify the relative position data estimated by the stereoscopic imaging system. This information will also be used to estimate the ballistic coefficient of MRS SAT.

3.5.7. Mission Mode 2: Circumnavigation. The operations performed during the Circumnavigation Mode are shown in Figure 3.8. MR SAT will continue to perform proximity operations by circumnavigating MRS SAT using stereoscopic imaging data and the cold-gas propulsion system (Figure 3.9). The circumnavigation orbit around MRS SAT will be maintained until all of the propellant has been consumed. Once all of the propellant has been consumed, MR SAT will enter Downlink Mode, and MRS SAT will enter End

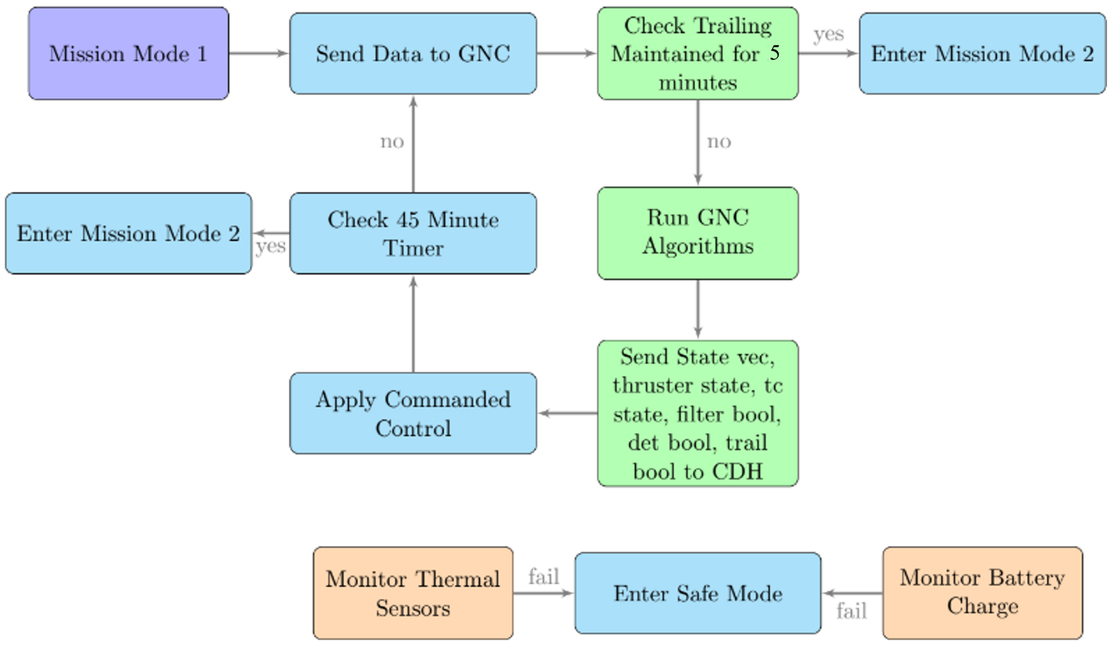


Figure 3.7. Trailing Formation block diagram; Source: [37]

of Life Mode. If MR SAT's power drops below a critical level, Mission Mode Two will be terminated, and the satellite will enter Downlink Mode. Images will be stored on board for downlinking to the M-SAT ground station. Three-dimensional image reconstruction of MRS SAT will be performed on the Missouri S&T campus using the images and data collected.

3.5.8. Downlink Mode. The operations performed during Downlink Mode are shown in Figure 3.10. In Downlink Mode, MR SAT will no longer have orbit control, but will still be able to maintain attitude via the torque coils. This mode is similar to Checkout Mode, except that the torque coils will now orient MR SAT into a near nadir-pointing attitude to enhance ground station communications. As MR SAT continues to orbit, mission data will be downlinked in order of priority: image pairs, engineering data, then log files. Once all data have been successfully received by the ground station, MR SAT will enter the End of Life Mode.

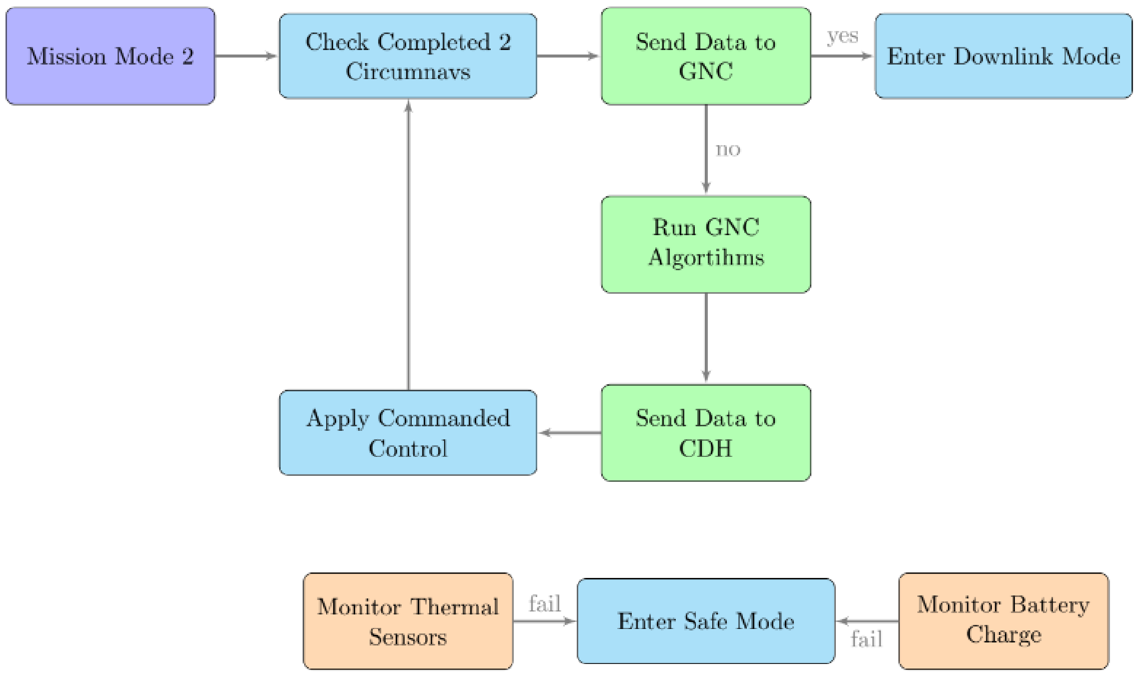


Figure 3.8. Circumnavigation block diagram; Source: [37]

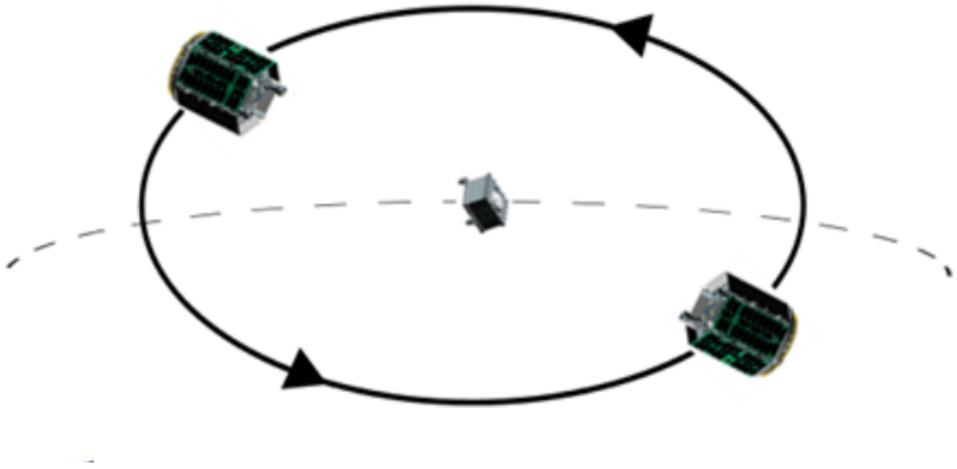


Figure 3.9. Circumnavigation orbit; Source: [38]

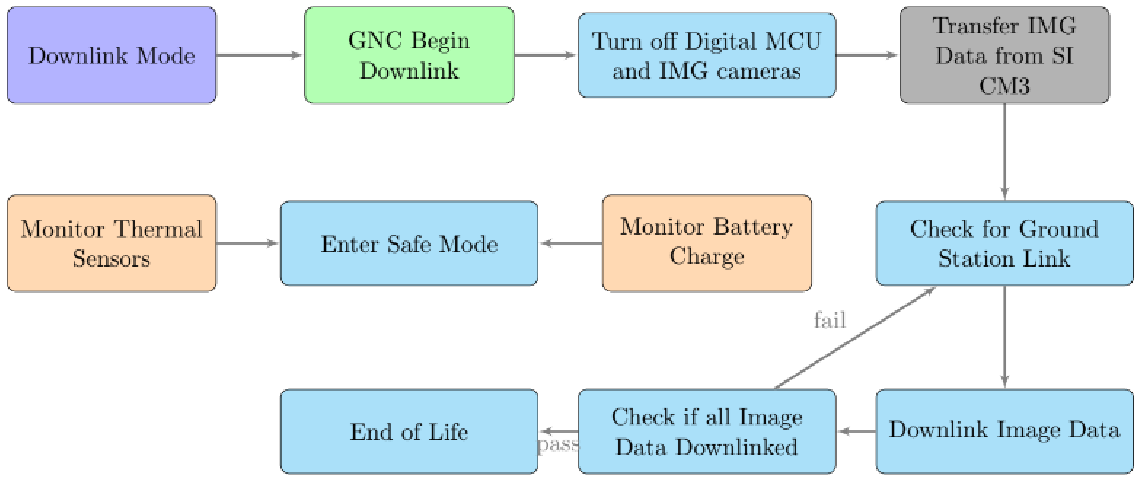


Figure 3.10. Downlink block diagram; Source: [37]

3.5.9. End of Life. The operations performed during End of Life Mode are shown in Figure 3.11. During End of Life, MRS SAT will drift away from MR SAT. The batteries on board MRS SAT will only operate for approximately four orbits after being deployed from MR SAT. MRS SAT will continue to beacon over the Iridium network during this period. Once the power is depleted, MRS SAT will then continue to orbit until the natural orbit decay results in atmospheric reentry. This reentry is expected to occur one to three years after deployment, which is well within the requisite 25-year timeframe (and even within the Federal Communications Commission (FCC) recently proposed five-year limit). When entering End of Life, MR SAT will power down all systems except sensors related to the GNC, Communication (COM), and C&DH systems. MR SAT will maintain communication with the ground station and will operate as a beaconing device while maintaining the ability to downlink engineering data. MR SAT's reentry is expected to occur four to twelve months after deployment from the ISS.

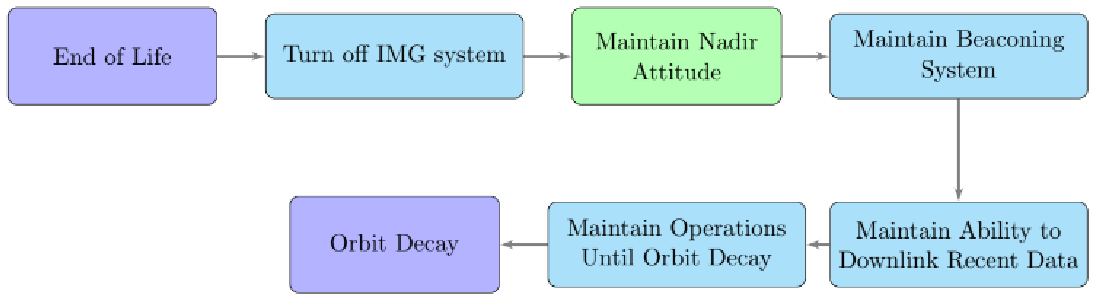

End of Life
Turn off IMG system
Maintain Nadir Attitude
Maintain Beaconing System
Maintain Ability to Downlink Recent Data
Maintain Operations Until Orbit Decay
Orbit Decay

Figure 3.11. End of Life block diagram; Source: [37]

3.5.10. Safe Mode. The operations performed during Safe Mode are shown in Figure 3.12. Safe Mode only activates if MR SAT's batteries drop below the critical threshold, which is defined as 115% of the minimum allowed depth of discharge, or if the temperature rises beyond the safe operating limit. When entering this mode, a system health check will be performed and logged to determine if a system is operating abnormally, which would trigger Safe Mode. After the health check, all non-critical systems will be powered down. Only the Power (PWR), C&DH, COM, GPS, and IMU will be operational in this mode. GNC will only utilize the GPS receiver and antenna to provide CDH position data, while CDH will generate data packages for COM. Beaconing and uplinking capabilities will be available on COM. In this mode, the batteries can be allowed to fully charge in the most power-positive mode for quick recharging to the battery capacity. Once the batteries are fully charged, the analog MCU will be powered on. If this mode is entered before Separation Mode, the spacecraft will return to Checkout Mode. If this mode is entered after MR SAT and MRS SAT have separated, the imaging computer will restart, and the spacecraft will enter Downlink Mode.

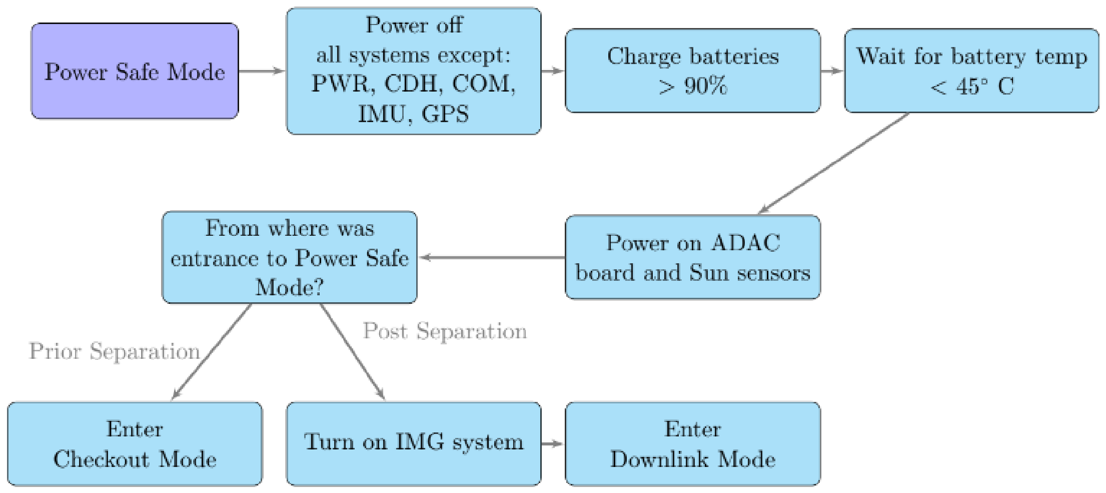

Power Safe Mode
Power off all systems except: PWR, CDH, COM, IMU, GPS
Charge batteries > 90%
Wait for battery temp < 45° C
Power on ADAC board and Sun sensors
From where was entrance to Power Safe Mode?
Prior Separation
Post Separation
Enter Checkout Mode
Turn on IMG system
Enter Downlink Mode

Figure 3.12. Safe Mode block diagram; Source: [37]

4. SOFTWARE DESIGN

4.1. SOFTWARE STRUCTURE

The software structure is shown in Figure 4.1. The flight code is split into five major components: (1) the State Machine, (2) the Data Manager, (3) the Interface Subthreads, (4) the State Info Class, and (5) the Hardware Interfaces. There are also two smaller components to the flight code: the SI Manager and the Comm Interface. The flight Pi will communicate with the SI Pi over an Ethernet line with TCP connection. A centralized control model was chosen such that only the State Machine is responsible for controlling the execution of the other components. This choice maintains and helps understand the control system more easily. All subprograms of the flight code run on their own thread to allow for simultaneous execution of MR SAT's flight code functions. The State Info Class and Thread Buffers are used to communicate between the various threads. The State Info Class is used to keep track of the state of MR SAT. The Thread Buffers are used to send commands and sensor data to the various threads.

4.2. STATE MACHINE

The State Machine will control the mission flow of the satellite and will be responsible for initializing the various subsystems of the satellite. The State Machine is created in the main function of the flight code. The private members of the State Machine include thread objects for COM, UART, I2C0, I2C1, 1-Wire, SPI, and the Data Manager. Vectors of command buffers for the interface subthreads are also included. The private members of the State Machine also include two more Thread Buffers: one for commands and one for sensor data sent to the Data Manager. Once the State Machine is created, StateMachine::start() is called in the main code to begin running the flight code. The State Info Class will be read to determine the current mission mode, and a switch case statement will

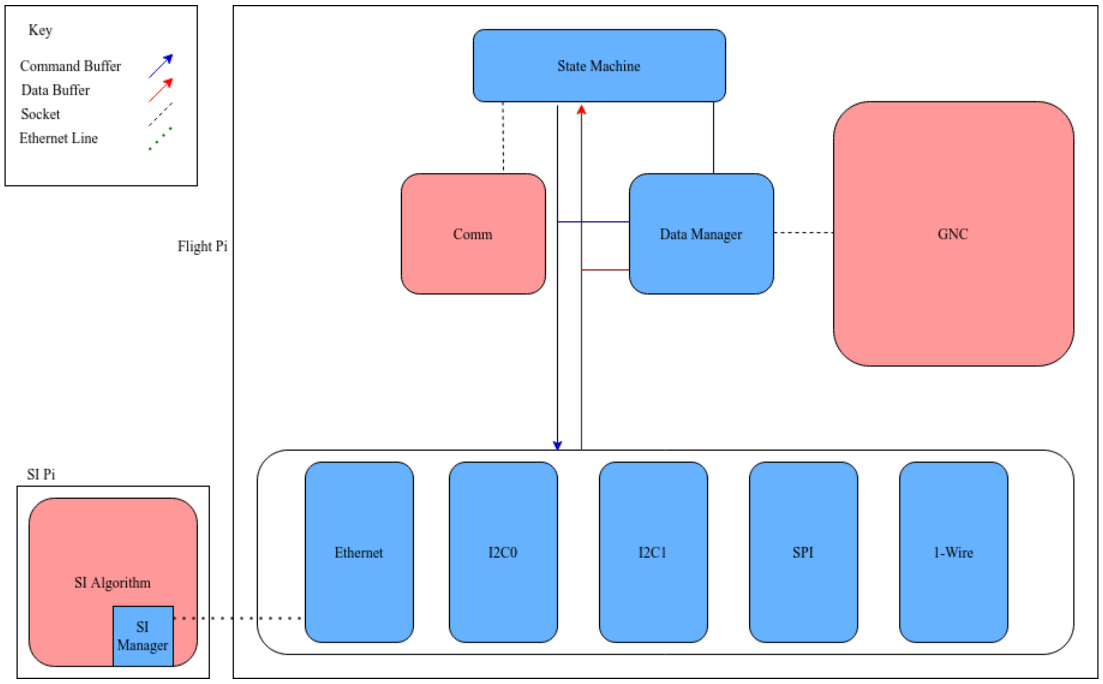


Figure 4.1. Flight code structure; Source: [39]

call the respective mission mode. Each mission mode has its own function that follows the MR SAT Mission Sequence outlined in Section 3.1. The flowchart of the State Machine is shown in Figures 4.2–4.4. Once each mission mode function completes, it will set the current mission mode to the next mode and return the switch and case statement that will call the next mission mode function. Commands will be sent over the Pluto radio from the ground station to the flight Pi and stored in the command list during the mission. A flightExceptionHandler function will be continually called whenever it is awaiting an external event to process the commands and monitor temperatures and battery voltage to determine whether entering the Safe Mode is necessary.

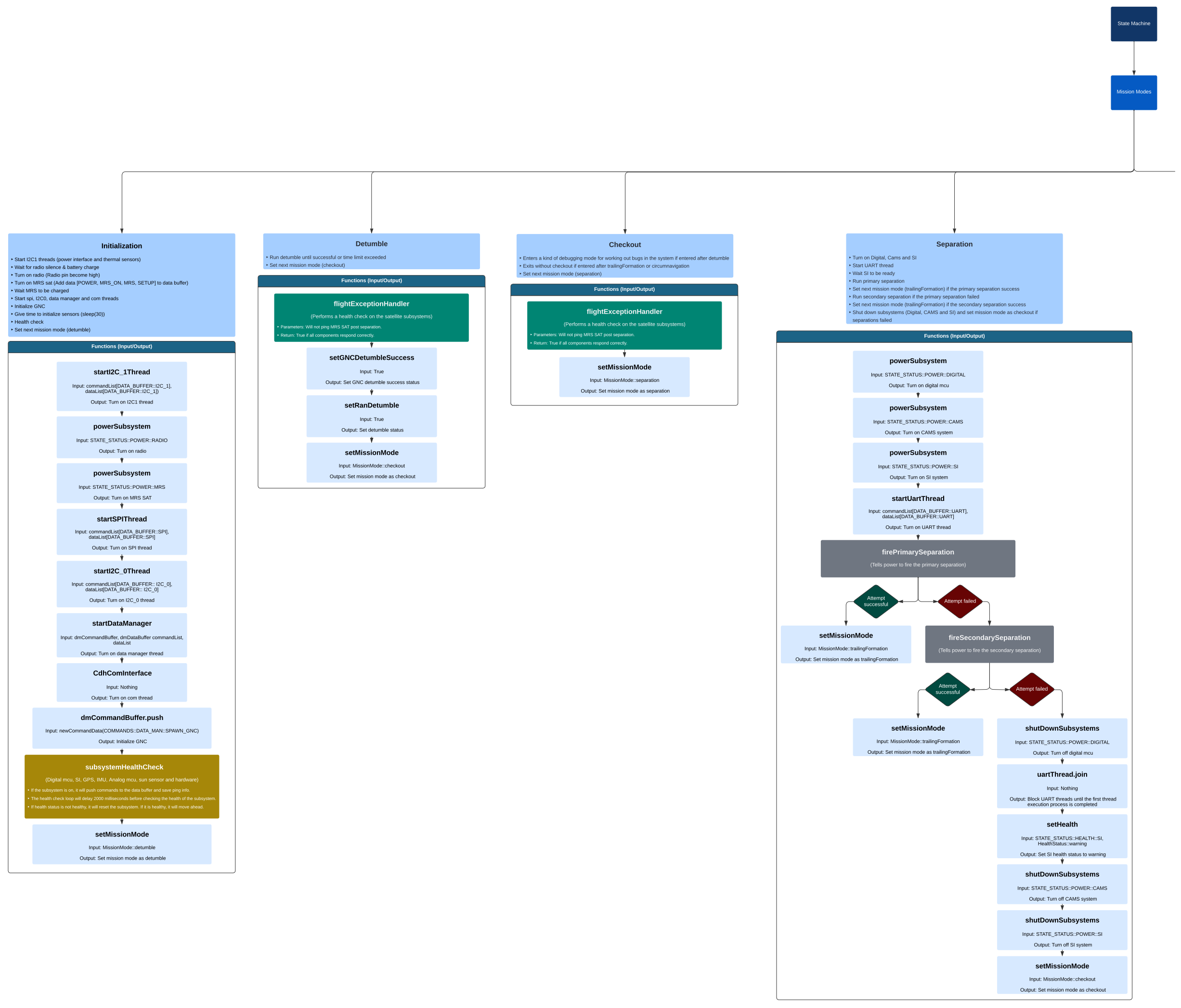

State Machine
Mission Modes
Initialization
Start I2C1 threads (power interface and thermal sensors)
Wait for radio silence & battery charge
Turn on radio (Radio pin become high)
Turn on MRS sat (Add data [POWER, MRS_ON, MRS, SETUP] to data buffer)
Wait MRS to be charged
Start spi, I2C0, data manager and com threads
Initialize GNC
Give time to initialize sensors (sleep(30))
Health check
Set next mission mode (detumble)
Functions (Input/Output)
startI2C_1Thread
Input: commandList[DATA_BUFFER::I2C_1], dataList[DATA_BUFFER::I2C_1])
Output: Turn on I2C1 thread
powerSubsystem
Input: STATE_STATUS::POWER::RADIO
Output: Turn on radio
powerSubsystem
Input: STATE_STATUS::POWER::MRS
Output: Turn on MRS SAT
startSPIThread
Input: commandList[DATA_BUFFER::SPI], dataList[DATA_BUFFER::SPI]
Output: Turn on SPI thread
startI2C_0Thread
Input: commandList[DATA_BUFFER:: I2C_0], dataList[DATA_BUFFER:: I2C_0]
Output: Turn on I2C_0 thread
startDataManager
Input: dmCommandBuffer, dmDataBuffer commandList, dataList
Output: Turn on data manager thread
CdhComInterface
Input: Nothing
Output: Turn on com thread
dmCommandBuffer.push
Input: newCommandData(COMMANDS::DATA_MAN::SPAWN_GNC)
Output: Initialize GNC
subsystemHealthCheck
(Digital mcu, SI, GPS, IMU, Analog mcu, sun sensor and hardware)
If the subsystem is on, it will push commands to the data buffer and save ping info.
The health check loop will delay 2000 milliseconds before checking the health of the subsystem.
If health status is not healthy, it will reset the subsystem. If it is healthy, it will move ahead.
setMissionMode
Input: MissionMode::detumble
Output: Set mission mode as detumble
Detumble
Run detumble until successful or time limit exceeded
Set next mission mode (checkout)
Functions (Input/Output)
flightExceptionHandler
(Performs a health check on the satellite subsystems)
Parameters: Will not ping MRS SAT post separation.
Return: True if all components respond correctly.
setGNCDetumbleSuccess
Input: True
Output: Set GNC detumble success status
setRanDetumble
Input: True
Output: Set detumble status
setMissionMode
Input: MissionMode::checkout
Output: Set mission mode as checkout
Checkout
Enters a kind of debugging mode for working out bugs in the system if entered after detumble
Exits without checkout if entered after trailingFormation or circumnavigation
Set next mission mode (separation)
Functions (Input/Output)
flightExceptionHandler
(Performs a health check on the satellite subsystems)
Parameters: Will not ping MRS SAT post separation.
Return: True if all components respond correctly.
setMissionMode
Input: MissionMode::separation
Output: Set mission mode as separation
Separation
Turn on Digital, Cams and SI
Start UART thread
Wait SI to be ready
Run primary separation
Set next mission mode (trailingFormation) if the primary separation success
Run secondary separation if the primary separation failed
Set next mission mode (trailingFormation) if the secondary separation success
Shut down subsystems (Digital, CAMS and SI) and set mission mode as checkout if separations failed
Functions (Input/Output)
powerSubsystem
Input: STATE_STATUS::POWER::DIGITAL
Output: Turn on digital mcu
powerSubsystem
Input: STATE_STATUS::POWER::CAMS
Output: Turn on CAMS system
powerSubsystem
Input: STATE_STATUS::POWER::SI
Output: Turn on SI system
startUartThread
Input: commandList[DATA_BUFFER::UART], dataList[DATA_BUFFER::UART]
Output: Turn on UART thread
firePrimarySeparation
(Tells power to fire the primary separation)
Attempt successful
Attempt failed
setMissionMode
Input: MissionMode::trailingFormation
Output: Set mission mode as trailingFormation
fireSecondarySeparation
(Tells power to fire the secondary separation)
Attempt successful
Attempt failed
setMissionMode
Input: MissionMode::trailingFormation
Output: Set mission mode as trailingFormation
shutDownSubsystems
Input: STATE_STATUS::POWER::DIGITAL
Output: Turn off digital mcu
uartThread.join
Input: Nothing
Output: Block UART threads until the first thread execution process is completed
setHealth
Input: STATE_STATUS::HEALTH::SI, HealthStatus::warning
Output: Set SI health status to warning
shutDownSubsystems
Input: STATE_STATUS::POWER::CAMS
Output: Turn off CAMS system
shutDownSubsystems
Input: STATE_STATUS::POWER::SI
Output: Turn off SI system
setMissionMode
Input: MissionMode::checkout
Output: Set mission mode as checkout

Figure 4.2. State Machine flowchart 1

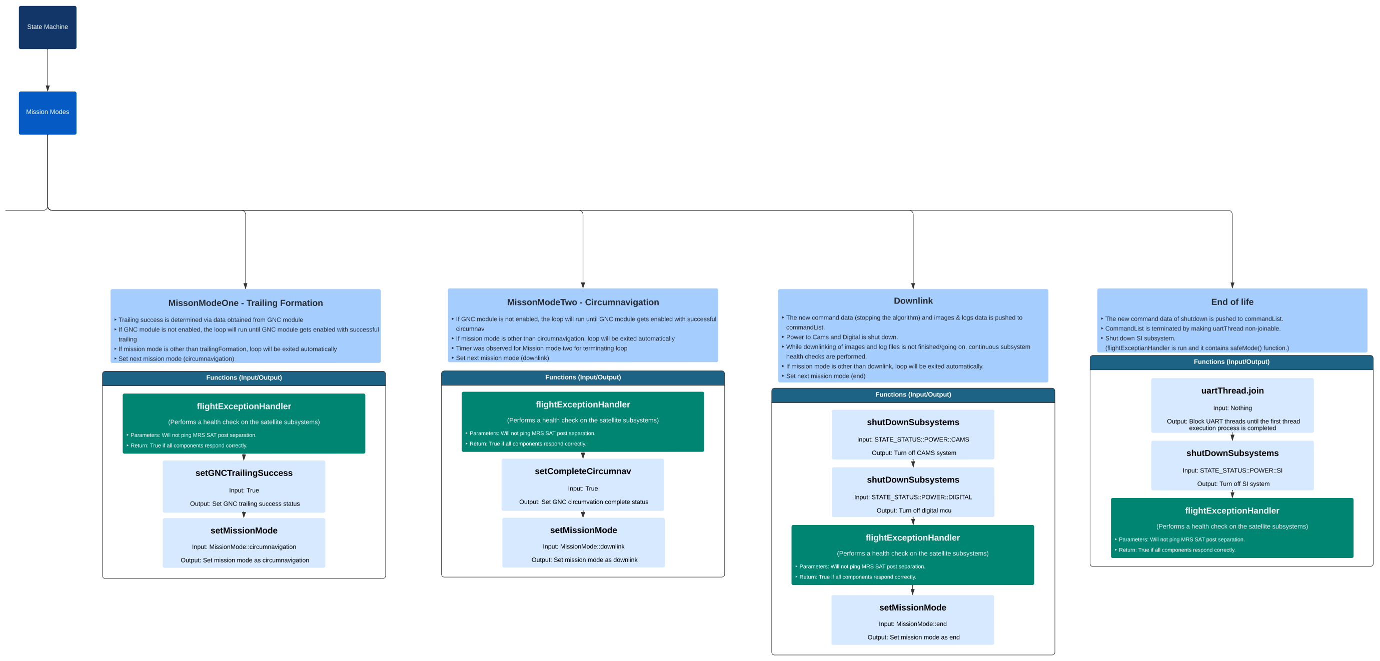

State Machine
Mission Modes
MissonModeOne - Trailing Formation
Trailing success is determined via data obtained from GNC module
If GNC module is not enabled, the loop will run until GNC module gets enabled with successful trailing
If mission mode is other than trailingFormation, loop will be exited automatically
Set next mission mode (circumnavigation)
Functions (Input/Output)
flightExceptionHandler
(Performs a health check on the satellite subsystems)
Parameters: Will not ping MRS SAT post separation.
Return: True if all components respond correctly.
setGNCTrailingSuccess
Input: True
Output: Set GNC trailing success status
setMissionMode
Input: MissionMode::circumnavigation
Output: Set mission mode as circumnavigation
MissonModeTwo - Circumnavigation
If GNC module is not enabled, the loop will run until GNC module gets enabled with successful circumnav
If mission mode is other than circumnavigation, loop will be exited automatically
Timer was observed for Mission mode two for terminating loop
Set next mission mode (downlink)
Functions (Input/Output)
flightExceptionHandler
(Performs a health check on the satellite subsystems)
Parameters: Will not ping MRS SAT post separation.
Return: True if all components respond correctly.
setCompleteCircumnav
Input: True
Output: Set GNC circumvation complete status
setMissionMode
Input: MissionMode::downlink
Output: Set mission mode as downlink
Downlink
The new command data (stopping the algorithm) and images & logs data is pushed to commandList.
Power to Cams and Digital is shut down.
While downlinking of images and log files is not finished/going on, continuous subsystem health checks are performed.
If mission mode is other than downlink, loop will be exited automatically.
Set next mission mode (end)
Functions (Input/Output)
shutDownSubsystems
Input: STATE_STATUS::POWER::CAMS
Output: Turn off CAMS system
shutDownSubsystems
Input: STATE_STATUS::POWER::DIGITAL
Output: Turn off digital mcu
flightExceptionHandler
(Performs a health check on the satellite subsystems)
Parameters: Will not ping MRS SAT post separation.
Return: True if all components respond correctly.
setMissionMode
Input: MissionMode::end
Output: Set mission mode as end
End of life
The new command data of shutdown is pushed to commandList.
CommandList is terminated by making uartThread non-joinable.
Shut down SI subsystem.
(flightExceptianHandler is run and it contains safeMode() function.)
Functions (Input/Output)
uartThread.join
Input: Nothing
Output: Block UART threads until the first thread execution process is completed
shutDownSubsystems
Input: STATE_STATUS::POWER::SI
Output: Turn off SI system
flightExceptionHandler
(Performs a health check on the satellite subsystems)
Parameters: Will not ping MRS SAT post separation.
Return: True if all components respond correctly.

Figure 4.3. State Machine flowchart 2

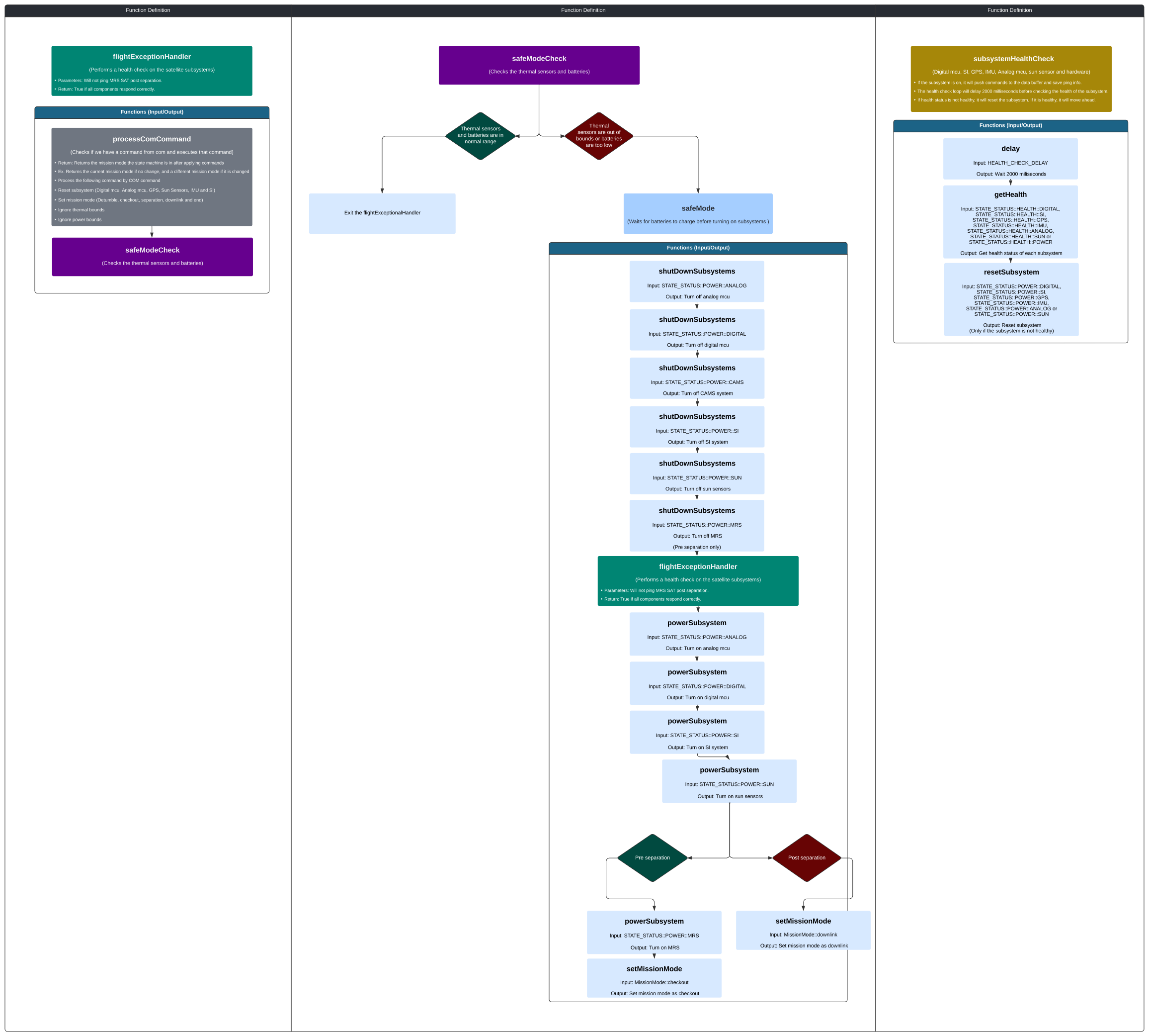

Function Definition
flightExceptionHandler
(Performs a health check on the satellite subsystems)
Parameters: Will not ping MRS SAT post separation.
Return: True if all components respond correctly.
Functions (Input/Output)
processComCommand
(Checks if we have a command from com and executes that command)
Return: Returns the mission mode the state machine is in after applying commands
Ex. Returns the current mission mode if no change, and a different mission mode if it is changed
Process the following command by COM command
Reset subsystem (Digital mcu, Analog mcu, GPS, Sun Sensors, IMU and SI)
Set mission mode (Detumble, checkout, separation, downlink and end)
Ignore thermal bounds
Ignore power bounds
safeModeCheck
(Checks the thermal sensors and batteries)
Function Definition
safeModeCheck
(Checks the thermal sensors and batteries)
Thermal sensors and batteries are in normal range
Thermal sensors are out of bounds or batteries are too low
Exit the flightExceptionalHandler
safeMode
(Waits for batteries to charge before turning on subsystems)
Functions (Input/Output)
shutDownSubsystems
Input: STATE_STATUS::POWER::ANALOG
Output: Turn off analog mcu
shutDownSubsystems
Input: STATE_STATUS::POWER::DIGITAL
Output: Turn off digital mcu
shutDownSubsystems
Input: STATE_STATUS::POWER::CAMS
Output: Turn off CAMS system
shutDownSubsystems
Input: STATE_STATUS::POWER::SI
Output: Turn off SI system
shutDownSubsystems
Input: STATE_STATUS::POWER::SUN
Output: Turn off sun sensors
shutDownSubsystems
Input: STATE_STATUS::POWER::MRS
Output: Turn off MRS
(Pre separation only)
flightExceptionHandler
(Performs a health check on the satellite subsystems)
Parameters: Will not ping MRS SAT post separation.
Return: True if all components respond correctly.
powerSubsystem
Input: STATE_STATUS::POWER::ANALOG
Output: Turn on analog mcu
powerSubsystem
Input: STATE_STATUS::POWER::DIGITAL
Output: Turn on digital mcu
powerSubsystem
Input: STATE_STATUS::POWER::SI
Output: Turn on SI system
powerSubsystem
Input: STATE_STATUS::POWER::SUN
Output: Turn on sun sensors
Pre separation
Post separation
powerSubsystem
Input: STATE_STATUS::POWER::MRS
Output: Turn on MRS
setMissionMode
Input: MissionMode::checkout
Output: Set mission mode as checkout
setMissionMode
Input: MissionMode::downlink
Output: Set mission mode as downlink
Function Definition
subsystemHealthCheck
(Digital mcu, SI, GPS, IMU, Analog mcu, sun sensor and hardware)
If the subsystem is on, it will push commands to the data buffer and save ping info.
The health check loop will delay 2000 milliseconds before checking the health of the subsystem.
If health status is not healthy, it will reset the subsystem. If it is healthy, it will move ahead.
Functions (Input/Output)
delay
Input: HEALTH_CHECK_DELAY
Output: Wait 2000 miliseconds
getHealth
Input: STATE_STATUS::HEALTH::DIGITAL, STATE_STATUS::HEALTH::SI, STATE_STATUS::HEALTH::GPS, STATE_STATUS::HEALTH::IMU, STATE_STATUS::HEALTH::ANALOG, STATE_STATUS::HEALTH::SUN or STATE_STATUS::HEALTH::POWER
Output: Get health status of each subsystem
resetSubsystem
Input: STATE_STATUS::POWER::DIGITAL, STATE_STATUS::POWER::SI, STATE_STATUS::POWER::GPS, STATE_STATUS::POWER::IMU, STATE_STATUS::POWER::ANALOG or STATE_STATUS::POWER::SUN
Output: Reset subsystem
(Only if the subsystem is not healthy)

Figure 4.4. State Machine flowchart 3

4.2.1. Initialization. The Initialization flow is shown in Figure 4.5. First, the startI2C_1Thread function is called to start the I^{2}C1 thread that manages the Sun sensors, magnetometer, and pressure transducer data with the analog board, digital board, and flight computer. Once the 45-minute radio silence timer has expired, the powerSubsystem function is called to turn on the radio and MRS SAT. Once MRS SAT's batteries are charged, the remaining threads (SPI, I^{2}C0, Data Manager, and COM) are started. The GNC subsystem is then initialized. A subsystem health check is performed, and the setMissionMode function is called to move to Detumble Mode if the health status is healthy.

4.2.2. Detumble. The detumble flow is shown in Figure 4.6. Once the GNC algorithm for detumbling is started, C&DH will wait for the detumble success status, or until the time limit has expired. During Detumble Mode, C&DH will continually check for commands, power, and temperature. If detumble is a success, the detumble status will be set as true. Once the nominal attitude is obtained or the time limit has expired, the setMissionMode function will be called to move to Checkout Mode.

4.2.3. Checkout. The Checkout flow is shown in Figure 4.7. During Checkout Mode, power and temperature status will be checked, and C&DH will process the command in the command data list. Once the command to move to Separation Mode is received and processed, the setMissionMode function will be called to move to Separation Mode.

4.2.4. Separation. The Separation flow is shown in Figure 4.8. Once Separation Mode is entered, the digital MCU, SI camera, and SI Pi will be powered on with the powerSubsystem function. The UART thread will then be turned on to manage imaging data, and a command to start the SI algorithm will be sent to the UART thread's command list. The spacecraft will then attempt the primary separation by commanding the power subsystem to fire the primary separation. If the primary separation fails, the secondary separation command will be executed. If the secondary separation fails, the digital MCU will be powered off, and the UART thread will be blocked until the first execution process is completed. In this case, the SI subsystem health status will be set to "warning." The SI

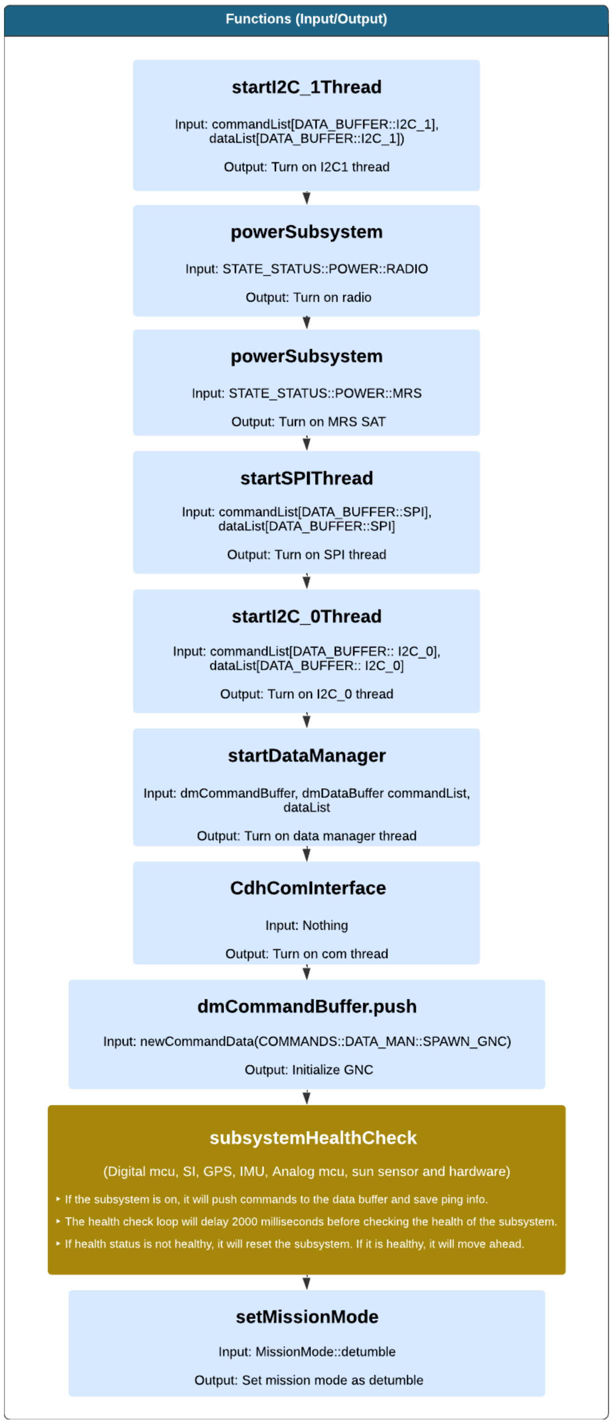

Functions (Input/Output)
startI2C_1Thread
Input: commandList[DATA_BUFFER::I2C_1], dataList[DATA_BUFFER::I2C_1])
Output: Turn on I2C1 thread
powerSubsystem
Input: STATE_STATUS::POWER::RADIO
Output: Turn on radio
powerSubsystem
Input: STATE_STATUS::POWER::MRS
Output: Turn on MRS SAT
startSPIThread
Input: commandList[DATA_BUFFER::SPI], dataList[DATA_BUFFER::SPI]
Output: Turn on SPI thread
startI2C_0Thread
Input: commandList[DATA_BUFFER:: I2C_0], dataList[DATA_BUFFER:: I2C_0]
Output: Turn on I2C_0 thread
startDataManager
Input: dmCommandBuffer, dmDataBuffer commandList, dataList
Output: Turn on data manager thread
CdhComInterface
Input: Nothing
Output: Turn on com thread
dmCommandBuffer.push
Input: newCommandData(COMMANDS::DATA_MAN::SPAWN_GNC)
Output: Initialize GNC
subsystemHealthCheck
(Digital mcu, SI, GPS, IMU, Analog mcu, sun sensor and hardware)
‣ If the subsystem is on, it will push commands to the data buffer and save ping info.
‣ The health check loop will delay 2000 milliseconds before checking the health of the subsystem.
‣ If health status is not healthy, it will reset the subsystem. If it is healthy, it will move ahead.
setMissionMode
Input: MissionMode::detumble
Output: Set mission mode as detumble

Figure 4.5. Initialization flow

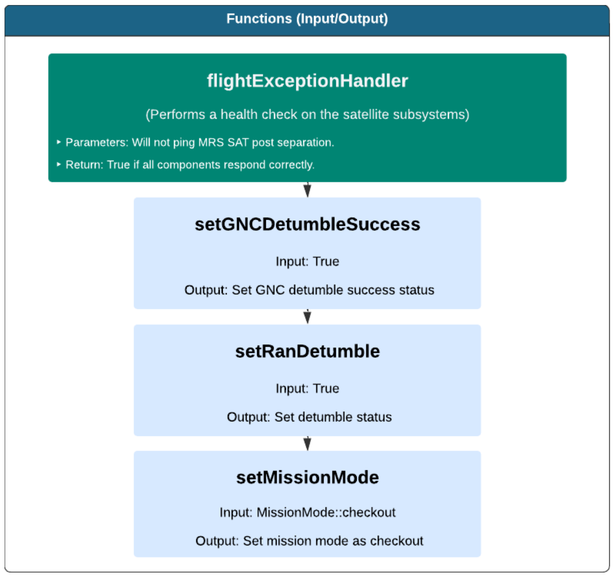

Functions (Input/Output)
flightExceptionHandler
(Performs a health check on the satellite subsystems)
Parameters: Will not ping MRS SAT post separation.
Return: True if all components respond correctly.
setGNCDetumbleSuccess
Input: True
Output: Set GNC detumble success status
setRanDetumble
Input: True
Output: Set detumble status
setMissionMode
Input: MissionMode::checkout
Output: Set mission mode as checkout

Figure 4.6. Detumble flow

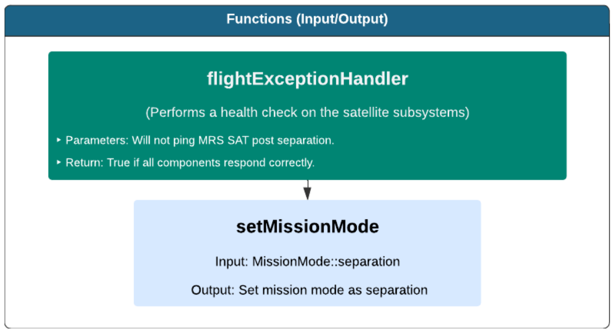

Functions (Input/Output)
flightExceptionHandler
(Performs a health check on the satellite subsystems)
Parameters: Will not ping MRS SAT post separation.
Return: True if all components respond correctly.
setMissionMode
Input: MissionMode::separation
Output: Set mission mode as separation

Figure 4.7. Checkout flow

camera, and SI Pi will then be turned off. The setMissionMode function will be called to move to Checkout Mode to wait for ground station commands. If either separation succeeds, the setMissionMode function will be called to move to Trailing Formation Mode.

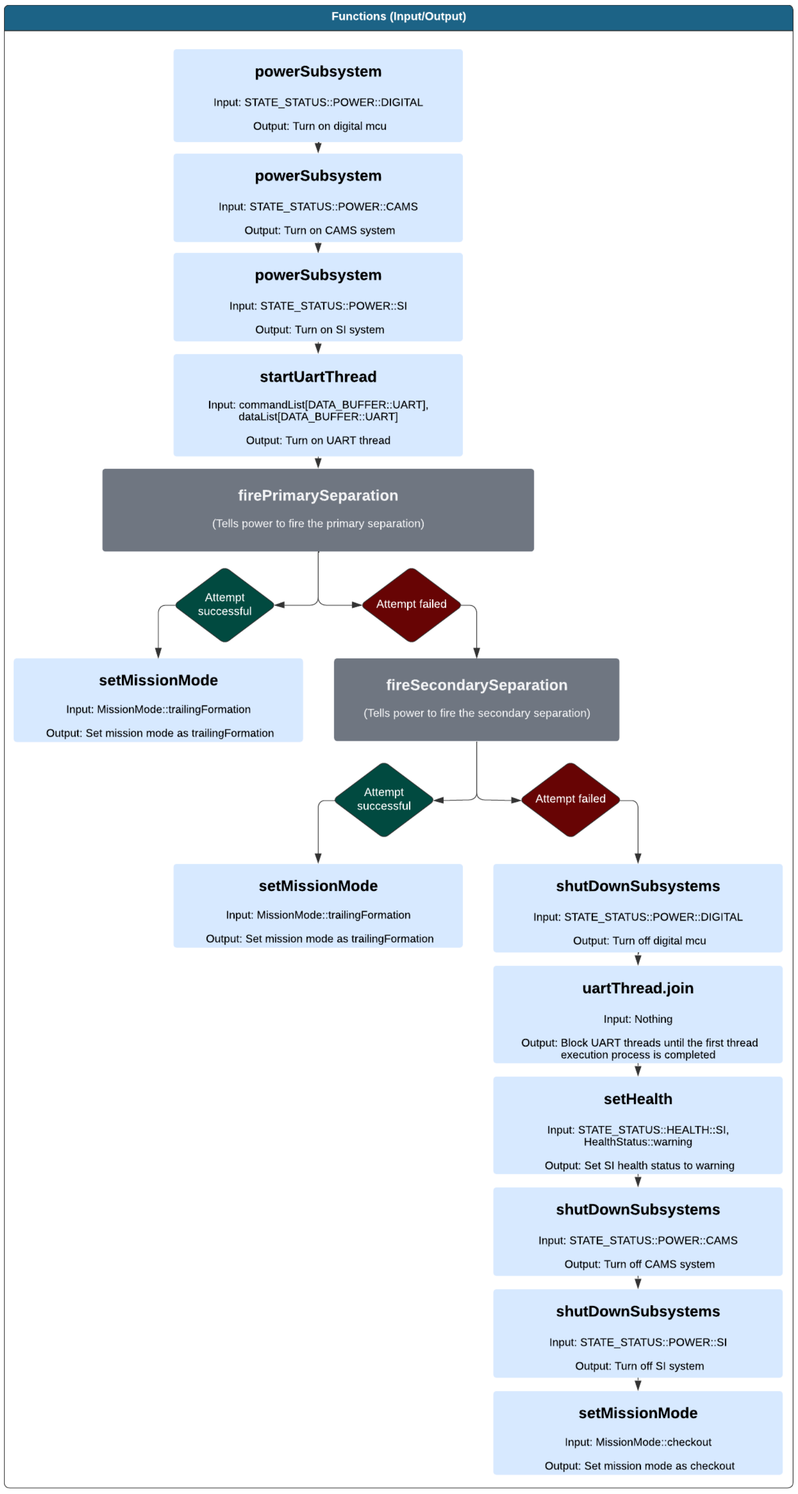

Functions (Input/Output)
powerSubsystem
Input: STATE_STATUS::POWER::DIGITAL
Output: Turn on digital mcu
powerSubsystem
Input: STATE_STATUS::POWER::CAMS
Output: Turn on CAMS system
powerSubsystem
Input: STATE_STATUS::POWER::SI
Output: Turn on SI system
startUartThread
Input: commandList[DATA_BUFFER::UART], dataList[DATA_BUFFER::UART]
Output: Turn on UART thread
firePrimarySeparation
(Tells power to fire the primary separation)
Attempt successful
Attempt failed
setMissionMode
Input: MissionMode::trailingFormation
Output: Set mission mode as trailingFormation
fireSecondarySeparation
(Tells power to fire the secondary separation)
Attempt successful
Attempt failed
setMissionMode
Input: MissionMode::trailingFormation
Output: Set mission mode as trailingFormation
shutDownSubsystems
Input: STATE_STATUS::POWER::DIGITAL
Output: Turn off digital mcu
uartThread.join
Input: Nothing
Output: Block UART threads until the first thread execution process is completed
setHealth
Input: STATE_STATUS::HEALTH::SI, HealthStatus::warning
Output: Set SI health status to warning
shutDownSubsystems
Input: STATE_STATUS::POWER::CAMS
Output: Turn off CAMS system
shutDownSubsystems
Input: STATE_STATUS::POWER::SI
Output: Turn off SI system
setMissionMode
Input: MissionMode::checkout
Output: Set mission mode as checkout

Figure 4.8. Separation flow

4.2.5. Trailing Formation. The Trailing Formation flow is shown in Figure 4.9. This mission mode program within the State Machine is similar to Detumble Mode. Once the GNC algorithm for Trailing Formation is started, C&DH will wait for "success" status. During Trailing Formation Mode, C&DH will continually check for commands, power, and temperature. If the mission is successful, the setMissionMode function will be called to move to Circumnavigation Mode. Unlike the Detumble Mode, the current Trailing Formation code does not yet have a time limit to exit this mode, but the 45-minute limit described in Section 3.5.6 will be added.

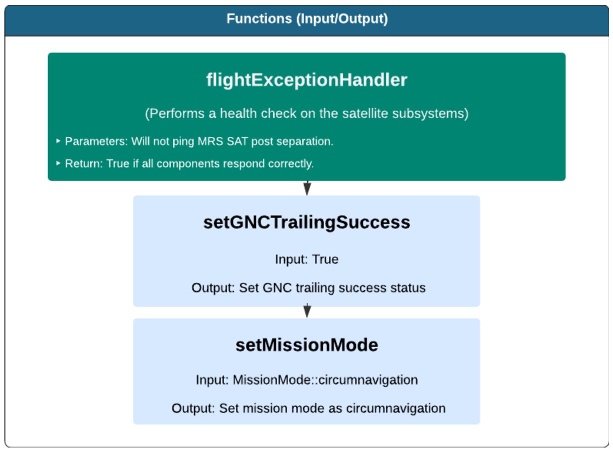

Functions (Input/Output)
flightExceptionHandler
(Performs a health check on the satellite subsystems)
Parameters: Will not ping MRS SAT post separation.
Return: True if all components respond correctly.
setGNCTrailingSuccess
Input: True
Output: Set GNC trailing success status
setMissionMode
Input: MissionMode::circumnavigation
Output: Set mission mode as circumnavigation

Figure 4.9. Trailing Formation flow

4.2.6. Circumnavigation. The Circumnavigation flow is shown in Figure 4.10. This mission mode program within the State Machine is similar to the Trailing Formation Mode. Once the GNC algorithm for Circumnavigation is started, C&DH will wait for "complete" status. During Circumnavigation Mode, C&DH will continually check for commands, power, and temperature. A timer will be observed for this mission mode to terminate the loop. Once the timer has expired or the propellant is fully expended, the setMissionMode function will be called to move to Downlink Mode.

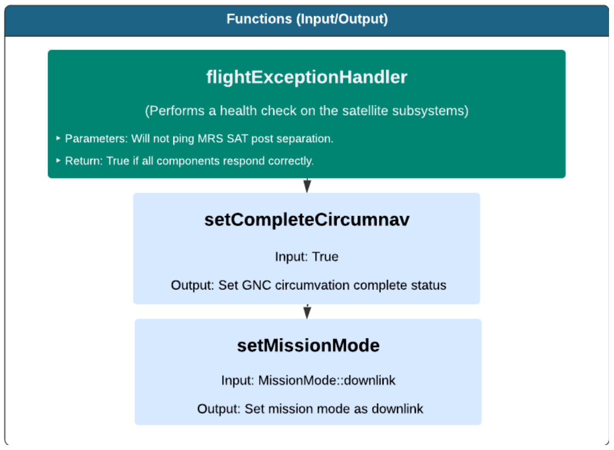

Functions (Input/Output)
flightExceptionHandler
(Performs a health check on the satellite subsystems)
‣ Parameters: Will not ping MRS SAT post separation.
‣ Return: True if all components respond correctly.
setCompleteCircumnav
Input: True
Output: Set GNC circumvation complete status
setMissionMode
Input: MissionMode::downlink
Output: Set mission mode as downlink

Figure 4.10. Circumnavigation flow

4.2.7. Downlink. The Downlink flow is shown in Figure 4.11. Once new command data for terminating the SI algorithm is pushed to the command list, the camera, and digital MCU will be powered off (because MRS SAT will have now drifted away from MR SAT). New command data for sending images and logging data from the SI Pi to the flight Pi will then be pushed to the command list of SI. While the downlinking of images and engineering data is ongoing, continuous power and temperature checks will be performed, and C&DH will process commands in the command data list. Once all data are successfully downlinked, the setMissionMode function will be called to move to End of Life Mode. All logs and engineering data will be stored and formatted in each buffer for respective data types as queue styles. As of the time of writing this thesis, coding to format and packetize all the data remains to be completed.

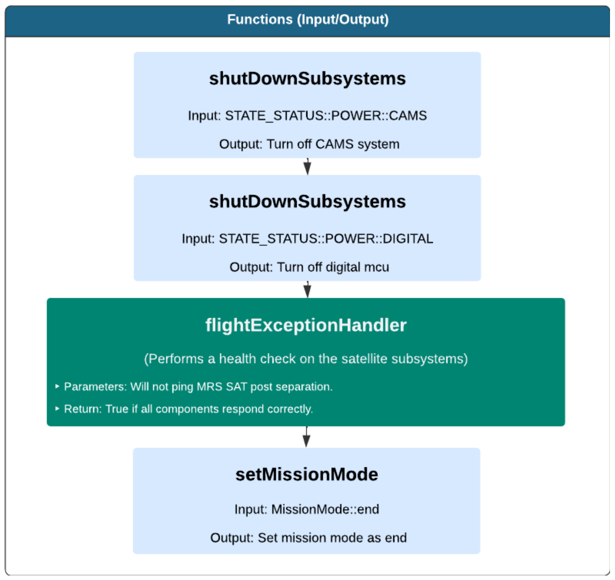

Functions (Input/Output)
shutDownSubsystems
Input: STATE_STATUS::POWER::CAMS
Output: Turn off CAMS system
shutDownSubsystems
Input: STATE_STATUS::POWER::DIGITAL
Output: Turn off digital mcu
flightExceptionHandler
(Performs a health check on the satellite subsystems)
Parameters: Will not ping MRS SAT post separation.
Return: True if all components respond correctly.
setMissionMode
Input: MissionMode::end
Output: Set mission mode as end

Figure 4.11. Downlink flow

4.2.8. End of Life. The End of Life flow is shown in Figure 4.12. First, new command data for thread shutdown will be pushed to the command list. The UART thread will then be blocked until the first thread execution process is completed and can be destroyed safely. Once the SI subsystem is powered off, the flightExceptionHandler function will process commands and check the power and temperature.

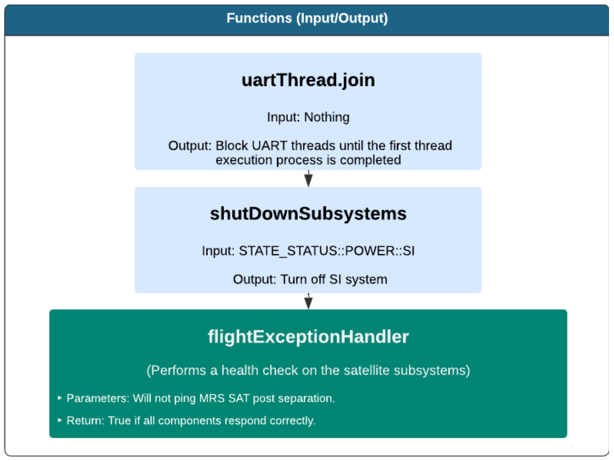

Functions (Input/Output)
uartThread.join
Input: Nothing
Output: Block UART threads until the first thread execution process is completed
shutDownSubsystems
Input: STATE_STATUS::POWER::SI
Output: Turn off SI system
flightExceptionHandler
(Performs a health check on the satellite subsystems)
Parameters: Will not ping MRS SAT post separation.
Return: True if all components respond correctly.

Figure 4.12. End of Life flow

4.2.9. Flight Exception Handler. The Flight Exception Handler flow is shown in Figure 4.13. This function may be called during the Detumble, Checkout, Trailing Formation, Circumnavigation, Downlink, and End of Life Modes. This function consists of two main functions: (1) processComCommand, and (2) safeModeCheck. The processComCommand function will check if a command from the COM subsystem is pending and, if so, execute the command that will have been previously uplinked from the ground station. Essentially, there are three types of commands that can be processed during this mission mode, including (1) to reset subsystems (digital MCU, analog MCU, GPS, Sun sensors, IMU, and SI Pi), (2) set mission mode (Detumble, Checkout, Separation, Downlink, and End of Life), and (3) ignore thermal and power bounds. The safeModeCheck function will check if the thermal sensors and the batteries are in bounds.

4.2.10. Safe Mode Check. The Safe Mode Check flow is shown in Figure 4.14. This function will be called from the flightExceptionHandler function. Once this function is called, it will check the thermal sensors and batteries. If they are in the normal range, it will exit the flightExceptionHandler. Otherwise, the safeMode function will be called to wait for the batteries to recharge by powering off non-essential subsystems. First, the

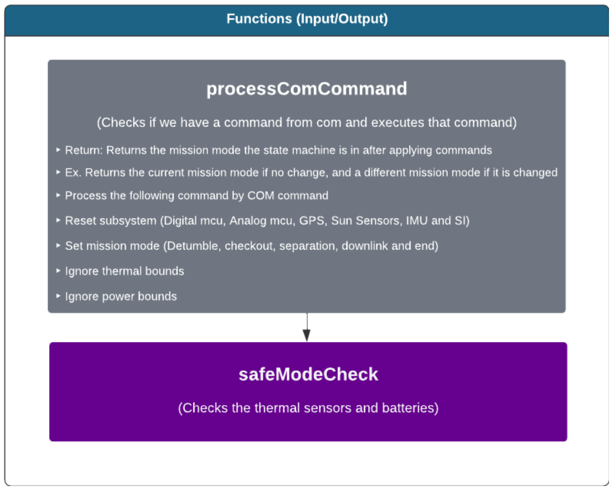

Functions (Input/Output)
processComCommand
(Checks if we have a command from com and executes that command)
Return: Returns the mission mode the state machine is in after applying commands
Ex. Returns the current mission mode if no change, and a different mission mode if it is changed
Process the following command by COM command
Reset subsystem (Digital mcu, Analog mcu, GPS, Sun Sensors, IMU and SI)
Set mission mode (Detumble, checkout, separation, downlink and end)
Ignore thermal bounds
Ignore power bounds
safeModeCheck
(Checks the thermal sensors and batteries)

Figure 4.13. Flight Exception Handler flow

analog MCU, digital MCU, CAMS, SI Pi, Sun sensors, and, if pre-separation, MRS SAT will be powered off. The flightExceptionHandler function will be called repeatedly until the batteries are sufficiently charged or a command has changed the mission mode. Once the batteries are fully charged, the analog MCU, digital MCU, SI Pi, and Sun sensors will be powered. If the Safe Mode is called before the separation, MRS SAT will be turned on, and the setMissionMode function will be called to move to Checkout Mode. If the safe mode is called after the separation, the setMissionMode function will be called to move to Downlink Mode.

4.2.11. Subsystem Health Check. The Subsystem Health Check flow is shown in Figure 4.15. The subsystemHealthCheck function will be called from Initialization Mode to check the subsystem health status. First, it will check if each subsystem (digital MCU, SI Pi, GPS, IMU, analog MCU, Sun sensors, and hardware) is in the on-state. If the subsystem is on, it will push commands to the data buffer and save the ping data. Next, each subsystem's health state will be checked. The subsystem will be reset if it is not healthy (i.e., warning, critical, or failure).

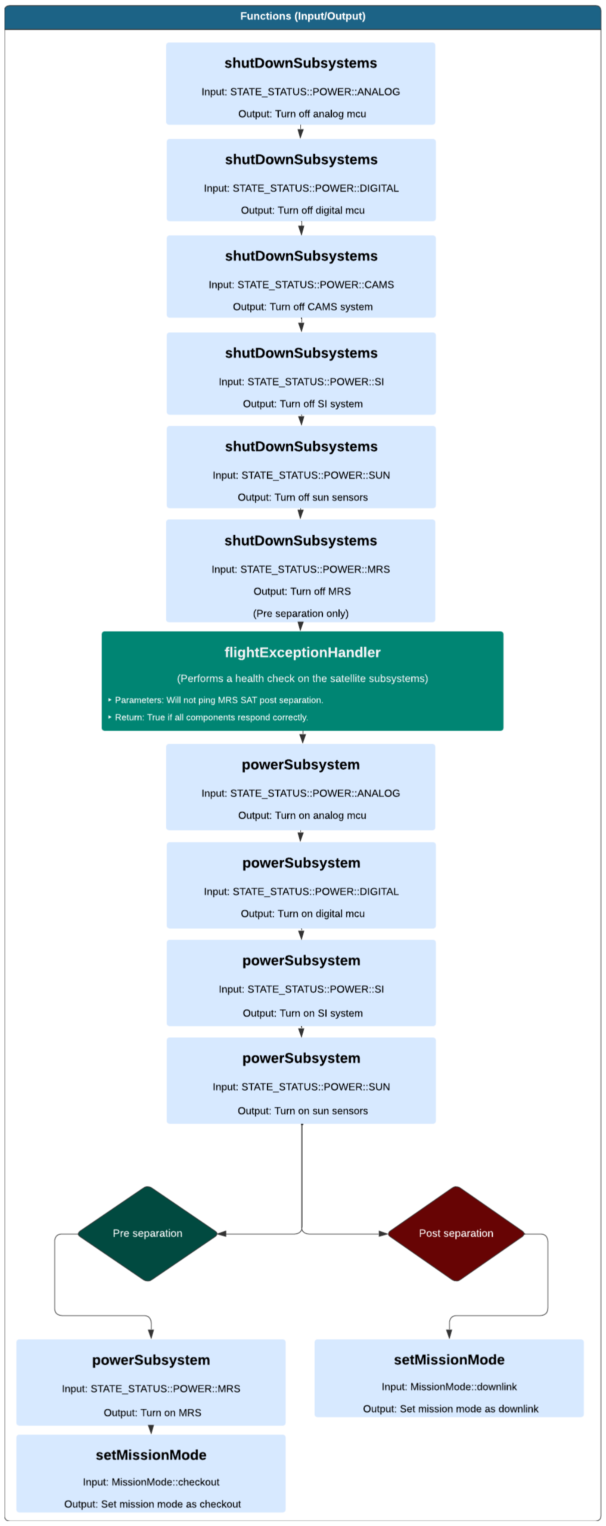

Functions (Input/Output)
shutDownSubsystems
Input: STATE_STATUS::POWER::ANALOG
Output: Turn off analog mcu
shutDownSubsystems
Input: STATE_STATUS::POWER::DIGITAL
Output: Turn off digital mcu
shutDownSubsystems
Input: STATE_STATUS::POWER::CAMS
Output: Turn off CAMS system
shutDownSubsystems
Input: STATE_STATUS::POWER::SI
Output: Turn off SI system
shutDownSubsystems
Input: STATE_STATUS::POWER::SUN
Output: Turn off sun sensors
shutDownSubsystems
Input: STATE_STATUS::POWER::MRS
Output: Turn off MRS
(Pre separation only)
flightExceptionHandler
(Performs a health check on the satellite subsystems)
Parameters: Will not ping MRS SAT post separation.
Return: True if all components respond correctly.
powerSubsystem
Input: STATE_STATUS::POWER::ANALOG
Output: Turn on analog mcu
powerSubsystem
Input: STATE_STATUS::POWER::DIGITAL
Output: Turn on digital mcu
powerSubsystem
Input: STATE_STATUS::POWER::SI
Output: Turn on SI system
powerSubsystem
Input: STATE_STATUS::POWER::SUN
Output: Turn on sun sensors
Pre separation
Post separation
powerSubsystem
Input: STATE_STATUS::POWER::MRS
Output: Turn on MRS
setMissionMode
Input: MissionMode::checkout
Output: Set mission mode as checkout
setMissionMode
Input: MissionMode::downlink
Output: Set mission mode as downlink

Figure 4.14. Safe Mode Check flow

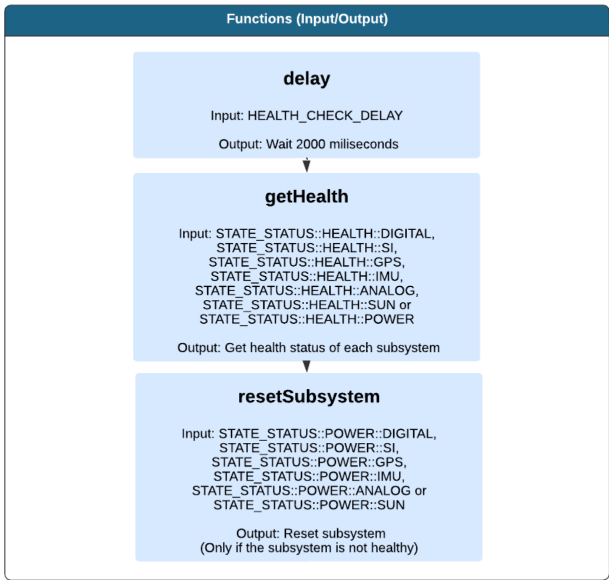

Functions (Input/Output)
delay
Input: HEALTH_CHECK_DELAY
Output: Wait 2000 miliseconds
getHealth
Input: STATE_STATUS::HEALTH::DIGITAL, STATE_STATUS::HEALTH::SI, STATE_STATUS::HEALTH::GPS, STATE_STATUS::HEALTH::IMU, STATE_STATUS::HEALTH::ANALOG, STATE_STATUS::HEALTH::SUN or STATE_STATUS::HEALTH::POWER
Output: Get health status of each subsystem
resetSubsystem
Input: STATE_STATUS::POWER::DIGITAL, STATE_STATUS::POWER::SI, STATE_STATUS::POWER::GPS, STATE_STATUS::POWER::IMU, STATE_STATUS::POWER::ANALOG or STATE_STATUS::POWER::SUN
Output: Reset subsystem
(Only if the subsystem is not healthy)

Figure 4.15. Subsystem Health Check flow

4.3. DATA MANAGER

The primary purpose of the Data Manager is to communicate with the GNC algorithm over a C socket. The Data Manager is a separate thread from the State Machine. Once the Data Manager thread is successfully created, the State Machine will send the command to spawn GNC to start the GNC algorithm. Then, the sensor data buffers will be cleared as the Interface Subthreads are started before the Data Manager. The Data Manager's primary loop will then gather sensor data, format it for GNC, check to see if any thermal sensors are out of boundaries, transmit data to GNC, receive data from GNC, update the State Info Class with data from GNC, and, if required, send commands to the torque coils or thrusters. The startDataManager function will be passed in the command and sensor data buffer vectors for the Interface Subthreads, and in separate command and sensor data buffers for communication with the State Machine. Before the Data Manager thread is terminated, the command to terminate GNC will be sent to avoid unnecessary resource usage.

4.4. INTERFACE SUBTHREADS

The Interface Subthreads are what the State Machine and Data Manager use to communicate with the hardware subsystems. Each Interface Subthread lives in its own thread, and there is a thread for each hardware communication bus, 1-Wire, UART, I^2C0, I^2C1, and SPI. Each thread has essentially the same structure: (1) connection to the hardware will be established, (2) a loop will be entered where a command from the command buffer is processed if there is one, (3) sensor data will be read when enough time has passed and sent to the Data Manager, and (4) the thread will sleep for 20 ms to ensure that the subthreads will not use all of the CPU's processing power.

4.5. STATE INFO CLASS

The State Info Class is designed to serve as the inter-thread communication of the state of C&DH, including the mission mode and each subsystem's health status. All the class members are static, so only one instance will be stored for each state, and an instance will be created before the main function is called. After the configuration file is read, all the mutexes needed will be created. The class's constructor is set to private, preventing the creation of additional class instances. Each member includes a corresponding getter and setter function, and the State Info Class only has these two functions. A mutex is assigned to each member and is locked and unlocked in the corresponding getter and setter functions. Therefore, only one thread can access a member at a time. If two threads access the same member, one that accesses the member earlier will lock the state, and the other thread will be blocked until the member is unlocked. Each function's parameters and variables are set to const. For maximum memory efficiency and user safety, the State Info Class utilizes mutexes and makes as many variables const as possible.

4.6. THREAD BUFFERS

The Thread Buffers are designed for inter-thread communication of command and sensor data. The Thread Buffer class uses a template so that any data types can be passed as input for the functions in the class. The type of Thread Buffers are queues where new data can be inserted into one end of the queue, and only data on the other side can be extracted. Therefore, the command and sensor data order will not change inside the buffer.

4.7. HARDWARE INTERFACES

The Hardware Interfaces have actual codes corresponding to each command from each subthread. The interfaces include the analog board, digital board, IMU, power, SI Pi, Sun sensors, thermal sensors, GPS, and COM. These interface programs have functions to control or read data from each subsystem. A detailed description of the commands and how to communicate each interface is discussed in Section 5.

5. DATA HANDLING AND INTERFACE CONTROL

The C&DH subsystem is responsible for handling sensor data and communicating with the other subsystems. The C&DH subsystem uses multiple data buses to communicate with all of the various systems on MR & MRS SAT. On MRS SAT, the flight computer uses I^2C to communicate with a thermal sensor and two UART lines for the Eyestar Radio and GPS. On MR SAT, the flight computer conducts communication through I^2C, SPI, UART, 1-Wire, TCP, and USB.

5.1. COMMUNICATION PROTOCOL

MR SAT needs to be able to communicate with multiple systems simultaneously, which requires an appropriate method of communication. All communication protocols used between the flight Pi and different systems are shown in Figure 5.1. Some sensors have predefined communication protocols that cannot be altered. The IMU sensor, Sun sensor, thermal sensor, and Pluto radio require an SPI bus, an I^2C bus, a 1-Wire bus, and a USB bus, respectively. A TCP connection is the simplest way to communicate with the two Raspberry Pis, so TCP is used as a communication protocol between the flight computer and the SI computer. For the rest of the systems, the selected data bus must be able to transmit data and commands. I^2C, UART, and SPI are possible bus solutions for MR SAT's main communication protocol to maximize communication effectiveness.

Considering these options, UART is the least appealing for complicated systems. Although UART can transfer data faster than SPI and I^2C, each board would require at least two individual data lines, requiring many UART lines to and from the flight computer. UART would require each system to have a hard-coded baud rate, and thus would prove to be too complicated and disorganized to communicate with the many systems on the satellite.

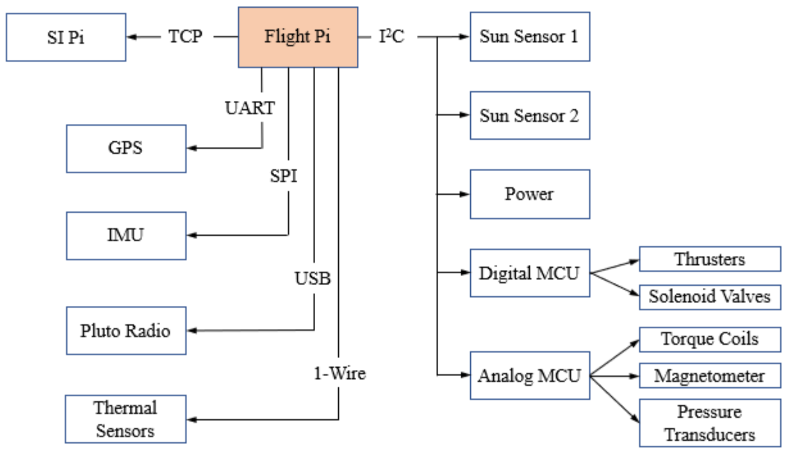


Figure 5.1. Communication protocol

SPI is a strong alternative to UART, but it still has disadvantages. Connecting all systems to the flight computer via an SPI bus would require fewer lines to and from the flight computer than UART, but more lines going into each board. This is because three lines are shared between each system with an SPI: a clock line, a transmit line, and a receive line. In addition to those three, each system would also have a select line running to it. This method of communicating would be better than UART, but the SPI is not ideal either.

The simplest and “cleanest” solution is the I^{2}C data bus. An I^{2}C bus allows the flight computer to communicate with every system with just two lines: a data line and a clock line. I^{2}C is also useful because it supplies the data sampling speed to every system by communicating over the clock line. This method of communication requires each board and sensor to have an address hard-coded onto it to allow the flight computer to select which board to communicate with. Although I^{2}C has a distance limitation, the problem can be solved by combining all C&DH functions on a single board. For these reasons, I^{2}C was chosen as the default communication protocol for MR SAT. I^{2}C is used for the following systems: the digital board that controls the thrusters and isolation valves, the analog board that controls the magnetometer, pressure transducer, and torque coils, and the Sun sensors.

5.2. COMMUNICATING WITH GNC

The GNC program runs in Simulink and is a different executable file from C&DH on the same Raspberry Pi, hence a different method is required to communicate with it. A function was written to allow GNC and C&DH to communicate with each other. The communication is handled over a C socket to allow the fast and rapid transmission of data between GNC and C&DH.

5.3. UPLINKING AND DOWNLINKING

Communication between the flight computer and the ground station is over the Pluto SDR radio. The flight computer is required to use USB to connect the device. Commands are sent from the ground station to the flight Pi, called "uplinking," and various data are sent from the flight Pi to the ground station, called "downlinking."

There are several types of commands from the ground station: (1) reset a specific subsystem, (2) enter a specific mission mode, (3) ignore any values from power or thermal sensors, and (4) determine if the satellite is ready to communicate with the ground station. Resetting the subsystem command can reset the digital MCU, analog controller, GPS, Sun sensors, IMU, or SI. Mission modes that can be entered by commands are Detumble, Checkout, Separation, Downlink, and End of Life Mode.

During each mission mode, GNC sensor data will be saved corresponding to the current mission mode. Once the mission mode is ready to be completed, the file will be compressed. During Detumble Mode or Checkout Mode, only the health and sensor data will be downlinked. No data will be downlinked during Trailing Formation Mode and Circumnavigation Mode because the nadir attitude needed for downlinking can not be maintained while performing the required maneuvers. When the spacecraft enters Downlink Mode, the data can be transmitted to the ground station by order of importance: the SI images, engineering data, and log files, respectively.

The maximum transmission capacity for the health and sensor data after Detumble or Checkout Modes is limited, and the data field within the frame structure will consist of 203 bytes of the frame. Therefore, two separate frames will be sent: health data (Table 5.1) and sensor data (Table 5.2). The data of the first six thermal sensors are in the health data frame, and the last nine thermal sensors are in the sensor data frame because of the limitation.

During Downlink Mode, after sending the image data, the engineering data and log data will be sent. The files for sensor data and event logs are separated by mission mode to allow the downlinking of data from the higher-priority mission modes during Downlink Mode. Before being downlinked, each log file will be compressed with 7-zip to accommodate the downlink budget. The engineering data include all IMU, GPS, Sun sensor, magnetometer, and SI data, along with any control data that needs to be sent, including thruster and torque coil data. Each sensor has a separate CSV file that stores its readings. The sensor readings will be added to its CSV whenever the sensor is read, with the exception of the IMU, where every tenth reading is saved to conserve space. C&DH will instruct the radio what file to downlink. Any action by the satellite which is not often repeated, such as reading sensors, is recorded in the log file. For example, a transition between mission mode or pinging a sensor would be logged. Events are logged at different levels depending on the seriousness of the event: (1) “debug 1” is for major actions (e.g., mission mode transit or starting a subsystem); (2) “debug 2” is for minor mission actions (e.g., waiting for the radio silence to be over or pinging a subsystem); (3) “error” is for an event where a minor action has failed (e.g., not detumbling in the allotted time or sensor not responding to a ping); (4) “critical” is for an event where a major anomaly has occurred (e.g., failing a separation attempt or a sensor not responding to multiple pings); and (5) “fatal” is for an event that cannot be recovered from (e.g., completely failing to separate or a sensor being declared as “dead”).

Table 5.1. Health data; Source: adapted from [40]

Field Name	Description	Value Type	Number of Entries	Total Byte Size	Starting Index
gps_time	The most up to date GPS time	uin32_t	2	8	0
gnc_filter_health	Has GNC Converged	bool	1	1	8
mrsat_position	Mr Sat Position	double	3	24	9
mrsat_vel	Mr Sat Velocity	float	3	12	33
mrsat_quat	Mr Sat Quaternion	float	4	16	45
mrsat_pos	Mrs Sat Relative Position	float	3	12	61
mrssat_vel	Mrs Sat Relative Velocity	float	3	12	73
mrsat_imu_bias	Mr Sat IMU Bias	double	3	24	85
gps_clock_bias	GPS Clock Bias	double	1	8	109
mission_mode	The Current Mission Mode	uint8_t	1	1	117
health_bool	Are we healthy	bool	1	1	118
ran_detumble_bool	Have we ran detumble	bool	1	1	119
battery_bool	Are the batteries charged	bool	1	1	120
detumbled_bool	Are we detumbled	bool	1	1	121
link_bool	Do we have ground link	bool	1	1	122
sep_command_bool	Do we have separation command	bool	1	1	123
separated_bool	Are we separated	bool	1	1	124
trail_bool	Have we trailed successfully	bool	1	1	125
circumnav_bool	Have we completed circumnav	bool	1	1	126
downlinked_bool	Have we downlinked everything	bool	1	1	127
all_image_data_bool	Has all image data been sent	bool	1	1	128
battery_min_bool	Battery above min threshold	bool	1	1	129
ignore_thermal_bool	Are we ignoring thermal	bool	1	1	130
ignore_power_bool	Are we ignoring power	bool	1	1	131
battery_voltage	Battery Voltage	float	1	4	132
battery_current	Battery Current	float	1	4	136
sensor_health_status	IMU Health	uint8_t	1	1	140
	GPS Health	uint8_t	1	1	141
	Sun Sensor Health	uint8_t	2	2	142
	Alalog Health	uint8_t	1	1	144
	Digital Health	uint8_t	1	1	145
	SI Health	uint8_t	1	1	146
torque_coil_data	Torque Coil Direction Bitfield	uint8_t	1	1	147
	Torque Coil Fire Percentage	uint8_t	3	3	148
thruster_data	Thruster Fire Percentage	uint8_t	12	12	151
thermal_data_id	Single byte identifying what thermal sensor is read	uint8_t	15	15	163
thermal_data	First 6 thermal sensors data	float	6	24	178

Table 5.2. Engineering data; Source: adapted from [40]

Name	Description	Value Type	Quantity	Byte Length	Total Field Length	Starting Index
imu_data	IMU Data				32	0
	XYZ Rotational Velocity	float	3	12		
	XYZ Acceleration	float	3	12		
	GPS Time	uint32_t	2	8		
gps_data	GPS Data				56	32
	XYZ Position	double	3	24		
	XYZ Velocity	double	3	24		
	GPS Time	uint32_t	2	8		
sun_data	Sun Sensor Data				24	88
	Alpha Angle 1	float	1	4		
	Beta Angle 1	float	1	4		
	Alpha Angle 2	float	1	4		
	Beta Angle 2	float	1	4		
	GPS Time	uint32_t	2	8		
mag_data	Magnetometer Data				20	112
	XYZ Magnetic Reading	float	3	12		
	GPS Time	uint32_t	2	8		
pressure_trans	Pressure Transducer				4	132
	Pressure 1	uint16_t	1	2		
	Pressure 2	uint16_t	1	2		
si_data	SI Data				24	136
	Alpha Angle 1	float	1	4		
	Beta Angle 1	float	1	4		
	Alpha Angle 2	float	1	4		
	Beta Angle 2	float	1	4		
	GPS Time	uint32_t	2	8		
thermal_data	Final 9 thermal sensors data	float	9	36	36	160

As of the time of writing this thesis, the downlink code has not yet been written, so it is currently a high priority in the software development effort. Still to be determined and programmed are the order of sensor data to be downlinked, the method of formatting and packetizing the sensor data, and how to transmit and receive the data over the Pluto radio.

6. VERIFICATION AND VALIDATION

6.1. UNIT TESTING

The first type of testing for the C&DH software is unit testing, the objective of which is to check if each function performs as expected. Once proper functionality is confirmed, the command execution testing (CET) will then be performed.

6.2. COMMAND EXECUTION TESTING

The objective of the CET test is to verify that all possible commands to the integrated satellite are executed correctly. The software will be tested with actual devices to ensure both the hardware and software can communicate and control each interface. First, the connection between the hardware and the flight computer has to be established and ensure the robustness of connection. All possible mission commands are then sent one at a time, and the return from each command on the interface is used to determine if all commands function as expected. Tests should be repeated, and “edge cases” should be considered and retested.

Each interface testing code has essentially the same structure: (1) show a menu of all commands on the screen, (2) read input typed from the computer corresponding to the menu, (3) call the respective function by a switch case statement, (4) send command to each interface, and (5) wait for a return from the interfaces. The menu must have an exit option to exit from the switch case statement and finish the testing. Each function should have a return message to indicate if the command succeeded. All commands that should be tested are shown in Table 6.1.

Table 6.1. Command list [41]

System	Command	Description
Digital Board	Turn On	Power on the digital board
	Turn Off	Power off the digital board
	Ping	Check if the flight computer can currently communicate with the digital MCU
	Update Thrusters	Indicate how long each thruster should remain on
	Open Iso Valves	Open the three isolation valves
	Close Iso Valves	Close the three isolation valves
	Test Thrusters	Test all the thrusters individually
	Run Thruster Sets	Test all the thrusters simultaneously
Analog Board	Turn On	Power on the analog board
	Turn Off	Power off the analog board
	Ping	Check if the flight computer can currently communicate with the analog MCU
	Read Mag	Read the X, Y, and Z strength of the magnetic field from the magnetometer
	Set Torque Coil	Turn off the magnetometers, set the direction for each of the torque coils, and set the burst timer for each of them
	Read Pressure	Read pressure from the pressure transducers
	Power Mag	Activate or deactivate the magnetometer
Power	Ping	Check if the flight computer can currently communicate with the power board
	Get Status	Get the power board status, battery current and voltage, and solar current and voltage
	Parse Status	Get an 11-byte array with data stored on the power board
	Execute Command	Run a user input command: activate or deactivate MRS SAT's inhibits, enable Separation Mode, or Activate Separation 1 or 2
Sun sensor	Turn On	Power on both Sun sensors
	Turn Off	Power off both Sun sensors
	Ping	Check if the flight computer can currently communicate with the Sun sensors
	Serial	Return the 6-byte serial number of the Sun sensors
	Read Voltages	Return the 3.3V, 5.0V, SRAM1, and SRAM2 current used by the Sun sensors
	Read Angles	Return the Alpha angle, Beta angle, and error codes
	Read CAM Settings	Return camera settings: the exposure, gain, blue gain, and red gain
	Set CAM Settings	Set camera settings: the exposure, gain, blue gain, and red gain
	Set Auto Adjust	Enable or disable the auto-adjust of the camera
	Reset	Reset communication interface, cameras, or MCU of the Sun sensor
IMU	Turn On	Power on the IMU
	Turn Off	Power off the IMU
	Reset IMU	Reset the IMU and go through power up and configuration
	Product ID	Return the IMU's eight character (ASCII) product ID, which is G364PDC0
	Firmware Version	Return the IMU's two-byte firmware version number, which is 2510
	Serial Num	Return the IMU's four-byte serial number, which is 00000026
	Read Vals	Get the X, Y, and Z gyroscopic values and accelerometer values
	Test GNC	Read the IMU values and write them into CSV file until the test has completed

Table 6.1. Command list [41] (cont.)

Thermal Sensor	Turn On	Power on the thermal sensors
	Turn Off	Power off the thermal sensors
	Detect All	Detect all the thermal sensors and display the IDs
	Display All	Display all thermal sensors with their ID, name, and temperature reading
	Name	Allow user to assign a name to each thermal sensor
Radio	Ping	Check if the flight computer can currently communicate with GPS
	Temperature	Get the temperature value from the radio device
	Transmit	Transmit data from the radio device of MR SAT to the ground station radio device
	Receive	Receive data from the ground station over the radio device
SI	Ping	Check if the flight computer can currently communicate with the SI Pi
	Start Algo	Tell the SI Manager to create a new thread and start the SI Algorithm
	Stop Algo	Tell the SI Manager to stop the SI Algorithm and join the tread it was running on
	Send Pics	Tell the SI Manager to start sending pictures
	Send Log Files	Tell the SI manager to start sending log files from the manager and the SI Algorithm
	Send Angles	Send a reading of angles to the Flight Raspberry Pi 3
	Si Die	End execution of the SI manager
GPS	Ping MCU	Check if the flight computer can currently communicate with GPS
	Power On	Power on the GPS receiver
	Power Off	Power off the GPS receiver
	Power Status	Check if the GPS is on
	Setup Connection	Initialize the GPS file descriptor and test sending functionality for UART
	Reset GPS	Send a command to the GPS MCU to reset
	Get Version	Get the GPS hardware and software version
	Get BESTXYZ	Display the BESTXYZ values
	Get RANGE	Display the RANGE values
	Get SATXYZ	Display the SATXYZ values
	Start BESTXYZ Logs	Start the BESTXYZ recurring logs
	Start RANGE Logs	Start the RANGE recurring logs
	Start SATXYZ Logs	Start the SATXYZ recurring logs
	Read Logs	Read the RANGE and SATXYZ recurring logs
	Settings	Set the baud rate to 9600 or 57600, clear logs, or save the current configuration

6.3. DAY-IN-THE-LIFE TESTING

Day-in-the-life (DITL) testing is the next test performed on the integrated satellite following the completion of the interface testing. The objective of this test is to verify that the integrated system performs as expected in scenarios that closely resemble on-orbit operations. The following lists all the possible scenarios in the actual run as:

- MR SAT fully nominal: a full run of mission from deployment to end of life
- MRS SAT fully nominal: a full run of MRS SAT's operations from separation to the sending of a kill command to MRS SAT's radio
- Detumble failure: spacecraft fails to detumble within seven days
- Mission Mode 1 battery depletion: the spacecraft loses power partway through Mission Mode 1
- Mission Mode 2 battery depletion: the spacecraft loses power partway through Mission Mode 2
- Post-detumble battery depletion: batteries deplete while in Checkout Mode
- SI failure: SI system fails to find MRS SAT
- Battery overheating
- Battery too cold
- Mission Mode 1 propellant depletion: the spacecraft consumes all propellant partway through Mission Mode 1
- Mission Mode 2 propellant depletion: the spacecraft consumes all propellant partway through Mission Mode 2
- Kill command: MR SAT is sent a kill command from the ground station to shut it off

All settings for each scenario are defined in the configure file. The configure file is read in the flight main code before the State Machine starts. The possible configurations are shown in Table 6.2, and the default option will be used if it is not defined in the configuration file. "Fake" data for GNC and COM are created because actual data will not be able to be obtained during the testing, but the functionality must still be verified. The appropriate settings for each scenario must be chosen. For example, the detumble timer has to be set for a much shorter period than the actual mission value to reduce the testing time.

Table 6.2. Configuration list

Name	Default	Option
Mission Mode	Initialization	Flight Code Start, Initialization, Detumble, Checkout, Separation, Trailing Formation, Circumnavigation, Downlink, End of Life, Safe, Navigation Only, and Reset Detumble
Start Time	0	
Last Time	0	
Digital Power	false	true or false
SI Power	false	true or false
GPS Power	false	true or false
IMU Power	false	true or false
Analog Power	false	true or false
Sun Sensor Power	false	true or false
MRS Power	false	true or false
First Run Status	true	true or false
Reboot Status	true	true or false
Separated Status	false	true or false
Detumble Status	false	true or false
Down Data Status	false	true or false
Down Logs Status	false	true or false
Down Pics Status	false	true or false
Display Status	false	true or false
CET Enable	false	true or false
GNC Enable	true	true or false
Fake GNC Enable	true	true or false
COM Enable	true	true or false
Fake COM Enable	true	true or false
IMU Enable	true	true or false
GPS Enable	true	true or false
Magnetometer Enable	true	true or false
Pressure Enable	true	true or false
Sun Sensor Enable	true	true or false
Thermal Enable	true	true or false
Power Enable	true	true or false
Radio Silence Timer	45 mins	
MRS Charge Timer	5 mins	
Detumble Timer	7 days	
Circumnav Timer	180 mins	

7. CONCLUSION AND FUTURE WORKS

This thesis presents the software development and current flight code description of MR & MRS SAT by the M-SAT Team. The description is to establish a reference for the software development of the C&DH subsystem and to serve as an example of a case study that can assist similar satellite programs. The software is designed to use the Raspberry Pi 3 Model B as the flight computer running a Linux environment written in the C++ language. The State Machine is implemented to control all mission flow for the spacecraft. Multithreading allows the concurrent execution of multiple threads: the State Machine, Data Manager, and Subthreads. The State Info Class and Thread Buffers are used to communicate between the separate threads. The communication method for each interface has been chosen to optimize the communication between the flight computer and various systems.

Because the radio device was recently changed and the communication protocol for each interface was modified to maximize communication efficiency, the interface codes have to be reconstructed, including the radio, SI, GPS, and analog interface. As the downlink code has not yet been written, it is a high-priority project for C&DH. There are also minor bugs to be fixed in the State Machine code. In order to complete these codes, unit tests need to be repeated until all software units have been successfully tested. CET will then be performed to validate all commands, and DITL will be conducted to validate all functionality on the integrated system with all possible scenarios on orbit.

As a part of the DITL testing, should Linux crash, it must be verified that the code execution can automatically restart from when it crashes. Furthermore, if the flight code's run time is found to be too long in the DITL testing, the flight code might have to be optimized to reduce the time. At the time of writing this thesis, the flight code can be run only when a user executes the file. However, the flight code must be executed automatically once the flight Pi is powered on, and this can be done by creating a service file.

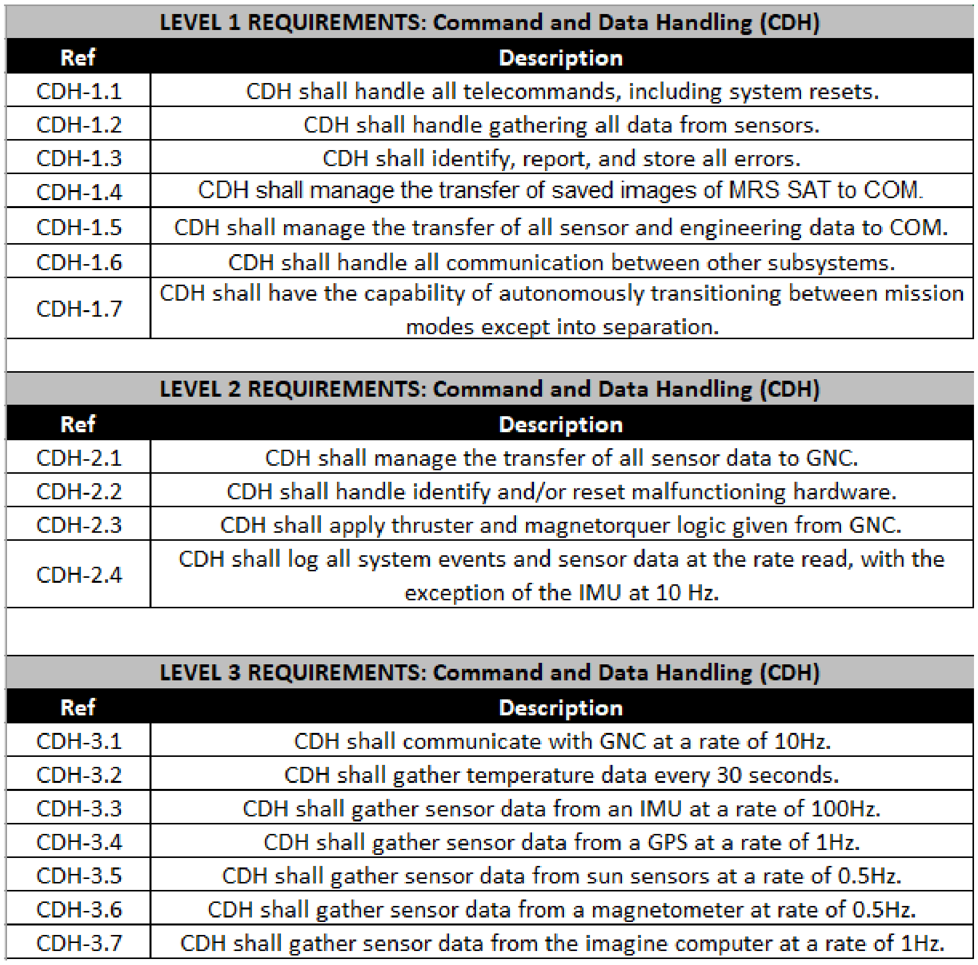
Triple modular redundancy (TMR) has to be implemented to increase the reliability of the system. With the use of TMR, a process is carried out by three systems, and the output is processed by a majority-voting system to generate one output. The two matching systems can reveal any flaws in any one of the three systems.

Finally, proper software development documentation allows for more efficient and reliable flight software and a higher likelihood of mission success. This is an ongoing process and will continue through integration and test of the spacecraft.

APPENDIX

COMMAND AND DATA HANDLING SUBSYSTEM REQUIREMENTS

Table 1. C&DH requirements; Source: adapted from [42]



LEVEL 1 REQUIREMENTS: Command and Data Handling (CDH)	
Ref	**Description**
CDH-1.1	CDH shall handle all telecommands, including system resets.
CDH-1.2	CDH shall handle gathering all data from sensors.
CDH-1.3	CDH shall identify, report, and store all errors.
CDH-1.4	CDH shall manage the transfer of saved images of MRS SAT to COM.
CDH-1.5	CDH shall manage the transfer of all sensor and engineering data to COM.
CDH-1.6	CDH shall handle all communication between other subsystems.
CDH-1.7	CDH shall have the capability of autonomously transitioning between mission modes except into separation.

LEVEL 2 REQUIREMENTS: Command and Data Handling (CDH)	
Ref	**Description**
CDH-2.1	CDH shall manage the transfer of all sensor data to GNC.
CDH-2.2	CDH shall handle identify and/or reset malfunctioning hardware.
CDH-2.3	CDH shall apply thruster and magnetorquer logic given from GNC.
CDH-2.4	CDH shall log all system events and sensor data at the rate read, with the exception of the IMU at 10 Hz.

LEVEL 3 REQUIREMENTS: Command and Data Handling (CDH)	
Ref	**Description**
CDH-3.1	CDH shall communicate with GNC at a rate of 10Hz.
CDH-3.2	CDH shall gather temperature data every 30 seconds.
CDH-3.3	CDH shall gather sensor data from an IMU at a rate of 100Hz.
CDH-3.4	CDH shall gather sensor data from a GPS at a rate of 1Hz.
CDH-3.5	CDH shall gather sensor data from sun sensors at a rate of 0.5Hz.
CDH-3.6	CDH shall gather sensor data from a magnetometer at rate of 0.5Hz.
CDH-3.7	CDH shall gather sensor data from the imagine computer at a rate of 1Hz.

REFERENCES

[1] C. Williams, "Nano-Microsatellite Market Forecast 10th Edition 2020." https://www.spaceworks.aero/wp-content/uploads/Nano-Microsatellite-Market-Forecast-10th-Edition-2020.pdf, 2020. Accessed: 2023-3-14.

[2] "Smallsats by the Numbers 2022." https://brycetech.com/reports/report-documents/Bryce_Smallsats_2022.pdf, 2022. Accessed: 2023-3-14.

[3] "University Nanosatellite Program." https://universitynanosat.org/. Accessed: 2022-12-28.

[4] B. Yost, S. Weston, G. Benavides, F. Krage, J. Hines, S. Mauro, S. Etchey, K. O'Neill, and B. Braun, "State-of-the-Art Small Spacecraft Technology," 2021.

[5] D. Miranda, "2020 NASA Technology Taxonomy," 2020.

[6] "Open Source Hardware Definition." https://www.oshwa.org/definition/. Accessed: 2023-1-18.

[7] R. Heradio, J. Chacon, H. Vargas, D. Galan, J. Saenz, L. De La Torre, and S. Dormido, "Open-Source Hardware in Education: A Systematic Mapping Study," *IEEE Access*, vol. 6, pp. 72094–72103, 2018.

[8] A. Cudmore, "Pi-Sat: A Low Cost Small Satellite and Distributed Spacecraft Mission System Test Platform," in *GSFC Fall IS&T Colloquium Series*, no. GSFC-E-DAA-TN27347, 2015.

[9] M. P. Del Rosso, A. Sebastianelli, D. Spiller, P. P. Mathieu, and S. L. Ullo, "On-Board Volcanic Eruption Detection through CNNs and Satellite Multispectral Imagery," *Remote Sensing*, vol. 13, no. 17, 2021.

[10] "What is Arduino?." https://www.arduino.cc/en/Guide/Introduction. Accessed: 2023-1-18.

[11] D. Geeroms, S. Bertho, M. De Roeve, R. Lempens, M. Ordies, and J. Prooth, "ARDUSAT, an Arduino-Based Cubesat Providing Students with the Opportunity to Create their own Satellite Experiment and Collect Real-World Space Data," ESA Publications Division C/O ESTEC, 2015.

[12] "Programming Language." https://www.techopedia.com/definition/24815/programming-language. Accessed: 2023-2-1.

[13] "Getting Started – About Version Control." https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control. Accessed: 2023-2-12.

[14] H. Leppinen, P. Niemelä, N. Silva, H. Sanmark, H. Forstén, A. Yanes, R. Modrzewski, A. Kestilä, and J. Praks, "Developing a Linux-Based Nanosatellite On-Board Computer: Flight Results from the Aalto-1 Mission," *IEEE Aerospace and Electronic Systems Magazine*, vol. 34, no. 1, pp. 4–14, 2019.

[15] H. Leppinen, "Current Use of Linux in Spacecraft Flight Software," *IEEE Aerospace and Electronic Systems Magazine*, vol. 32, no. 10, pp. 4–13, 2017.

[16] "FreeRTOS –Real-Time Operating System for Microcontrollers." https://www.freertos.org/. Accessed: 2023-2-12.

[17] M. Alam, A. Khamees, T. Aboelnaga, A. Amer, A. Harbi, M. Alamir, H. Alarwsh, and O. A. Elsayed, "Design and Implementation of an Onboard Computer and payload for Nano Satellite (CubeSat)," in *The International Undergraduate Research Conference*, vol. 5, pp. 361–364, The Military Technical College, 2021.

[18] B. Rajulu, S. Dasiga, and N. R. Iyer, "Open Source RTOS Implementation for On-Board Computer (OBC) in STUDSAT-2," in *2014 IEEE Aerospace Conference*, pp. 1–13, IEEE, 2014.

[19] "CODE.NASA.GOV." https://code.nasa.gov/. Accessed: 2023-2-12.

[20] G. Holzmann, "The Power of 10: Rules for Developing Safety-Critical Code," *Computer*, vol. 39, no. 6, pp. 95–99, 2006.

[21] G. Reeves and J. Snyder, "An Overview of the Mars Exploration Rovers' Flight Software," in *2005 IEEE International Conference on Systems, Man and Cybernetics*, vol. 1, pp. 1–7 Vol. 1, 2005.

[22] S. A. Johl and E. G. Lightsey, "A Reusable Command and Data Handling System for University CubeSats," *Journal of Small Satellites*, vol. 4, no. 2, pp. 357–369, 2015.

[23] J. Wang, *Formal Methods in Computer Science*. Chapman and Hall/CRC, 2019.

[24] J. S. Fant, H. Gomaa, and R. G. Pettit, "Architectural Design Patterns for Flight Software," in *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pp. 97–101, 2011.

[25] "Basics of the I2C Communication Protocol." https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol. Accessed: 2023-2-12.

[26] "Understanding I2C and SPI Communication Protocols." https://www.totalphase.com/blog/2021/12/i2c-vs-spi-vs-uart-introduction-and-comparison-similarities-differences/\#:~:text=Unlike\%20communication\%20protocols\%20like\%20I2C,send\%20and\%20receive\%20the\%20data. Accessed: 2023-2-12.

[27] "Basics of the SPI Communication Protocol." https://www.circuitbasics.com/basics-of-the-spi-communication-protocol/. Accessed: 2023-2-12.

[28] “Basics of UART Communication.” https://www.circuitbasics.com/basics-uart-communication/. Accessed: 2023-2-12.

[29] “Reading and Writing 1-Wire Devices Through Serial Interfaces.” https://www.analog.com/en/app-notes/reading-and-writing-1wirereg-devices-through-serial-interfaces.html. Accessed: 2023-2-12.

[30] J. Mankar, C. Darode, K. Trivedi, M. Kanoje, and P. Shahare, “Review of I2C protocol,” *International Journal of Research in Advent Technology*, vol. 2, no. 1, 2014.

[31] G. Fairhurst, B. Trammell, and M. Kühlewind, “Services Provided by IETF Transport Protocols and Congestion Control Mechanisms.” RFC 8095, Mar. 2017. https://www.rfc-editor.org/info/rfc8095. Accessed: 2023-2-12.

[32] S. A. Jacklin, “Survey of Verification and Validation Techniques for Small Satellite Software Development,” *Space tech expo*, 2015.

[33] “Day-in-the-Life Testing.” https://s3vi.ndc.nasa.gov/ssri-kb/topics/36/. Accessed: 2023-2-12.

[34] C. Venturini, B. Braun, D. Hinkley, and G. Berg, “Improving Mission Success of CubeSats,” *Proc. 32nd Annu. AIAA/USU Conf. Small Satell.*, pp. 1–11, 2018.

[35] Missouri S&T Satellite Research Team, “CDH101 - Conceptual Design Document.” Internal Documentation, 2021.

[36] “5 Git Workflows and Branching Strategy you can Use to Improve your Development Process.” https://rovitpm.com/5-git-workflows-to-improve-development/. Accessed: 2023-2-12.

[37] Missouri S&T Satellite Research Team, “SYS201 - Concept of Operations.” Internal Documentation, 2020.

[38] Missouri S&T Satellite Research Team, “SYS203 - Mission Overview.” Internal Documentation, 2019.

[39] Missouri S&T Satellite Research Team, “CDH108 - Software Design.” Internal Documentation, 2019.

[40] Missouri S&T Satellite Research Team, “CDH107 - Communicatios Interface Control.” Internal Documentation, 2019.

[41] Missouri S&T Satellite Research Team, “CDH103 - Interface Control Document.” Internal Documentation, 2019.

[42] Missouri S&T Satellite Research Team, “SYS206 - M-SAT Requirements Verification Matrix.” Internal Documentation, 2019.

VITA

Yumeka Nagano was born in Chiba, Japan on February 7, 1999. She started attending Saitama University in Japan in April 2017 and received a Bachelor's degree in Mechanical Engineering in September 2020. She entered Missouri University of Science and Technology in August 2021 and became a Command and Data Handling subsystem member of the Missouri S&T Satellite Research Team. She received a Master of Science Degree in Aerospace Engineering from Missouri University of Science and Technology in May 2023.