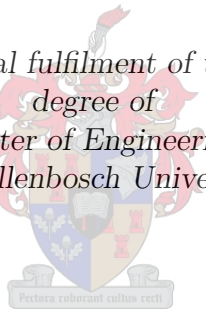
Design and Implementation of Generic Flight Software for a CubeSat

by

André Emile Heunis

Thesis presented in partial fulfilment of the requirements for the degree of
Master of Engineering
at Stellenbosch University



Supervisor:

Prof W. H. Steyn

Department Electrical and Electronic Engineering

December 2014

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

October 2014

Abstract

The main on-board computer in a satellite is responsible for ensuring the correct operation of the entire system. It performs this task using flight software. In order to reduce future development costs, it is desirable to develop generic software that can be re-used on subsequent missions. This thesis details the design and implementation of a generic flight software application for CubeSats.

A generic, modular framework is used in order to increase the re-usability of the flight software architecture. In order to simplify the management of the various on-board processes, the software is built upon the FreeRTOS real-time operating system. The Consultative Committee for Space Data Systems' telemetry and telecommand packet definitions are used to interface with ground stations. In addition, a number of services defined in the European Cooperation for Space Standardisation's Packet Utilisation Standard are used to perform the functions required from the flight software.

The final application contains all the command and data handling functionality required in a standard CubeSat mission. Mechanisms for the collection, storage and transmission of housekeeping data are included as well as the implementation of basic fault tolerance techniques. Through testing it is shown that the FreeRTOS scheduler can be used to ensure the software meets hard-real time requirements.

Opsomming

Die hoof aanboordrekenaar in 'n satelliet verseker die korrekte werking van die hele stelsel. Die rekenaar voer hierdie taak uit deur van vlugsagteware gebruik te maak. Om toekomstige ontwikkelingskostes te verminder, is dit noodsaaklik om generiese sagteware te ontwikkel wat hergebruik kan word op daaropvolgende missies. Hierdie tesis handel oor die besonderhede van die ontwerp en implementering van generiese vlugsagteware vir 'n CubeSat.

'n Generiese, modulêre raamwerk word gebruik om die hergebruik van die sagteware te verbeter. Ten einde die beheer van die verskillende aanboordprosesse te vereenvoudig, word die sagteware gebou op die FreeRTOS reëletyd bedryfstelsel. Die telemetrie- en telebevelpakket definisies van die "Consultative Committee for Space Data Systems" word gebruik om met grondstasies te kommunikeer. Daarby is 'n aantal dienste omskryf in die "Packet Utilisation Standard" van die "European Cooperation for Space Standardisation" gebruik om die vereiste funksies van die vlugsagteware uit te voer.

Die finale sagteware bevat al die bevel en data-hantering funksies soos wat vereis word van 'n standaard CubeSat missie. Meganismes vir die versameling, bewaring en oordrag van huishoudelike data is ingesluit sowel as die implementering van basiese fouttolerante tegnieke. Toetse het gewys dat die FreeRTOS skeduleerder gebruik kan word om te verseker dat die sagteware aan harde reëletyd vereistes voldoen.

Acknowledgements

The following people deserve more acknowledgement than they are receiving here.

- **Prof. W.H. Steyn** for accepting me as a student and showing me how to be a better engineer.
- **Pieter Botma, Christo Groenewald, Jako Gerber, Mike-Alex Kearney and Willem Jordaan** for their wisdom, knowledge and inexhaustible patience.
- **Gerhard Janse van Vuuren and Jan-Hielke le Roux** for being the best.
- **My parents, Riki Schutte and all my friends** for the unconditional love, support and motivation.

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations and Acronyms

ADCS	Attitude Determination and Control Subsystem
APID	Application Process Identifier
BSP	Board Support Package
CCF	Common Cause Fault
CCSDS	Consultative Committee for Space Data Systems
CMSIS	Cortex Microcontroller Software Interface Standard
CUC	CCSDS Unsegmented Code
DMA	Direct Memory Access
EBI	External Bus Interface
ECC	Error Correcting Codes
ECSS	European Cooperation for Space Standardisation
EDAC	Error Detection and Correction
EPS	Electrical Power System
ESL	Electronic Systems Laboratory
FATFS	File Allocation Table File System
ISR	Interrupt Service Routine
LEO	Low Earth Orbit
MCU	Micro-controller Unit
MPU	Memory Protection Unit
OBC	On-board Computer
PC	Personal Computer
PEC	Packet Error Control
PUS	Packet Utilisation Standard

RTC	Real Time Clock
RTOS	Real Time Operating System
SAA	South Atlantic Anomaly
SCS	Satellite Control Software
SD	Secure Digital
SDU	Service Data Unit
SEE	Single Event Effect
SEU	Single Event Upset
SID	Structure Identifier
SPI	Serial Peripheral Interface
SPOF	Single Point of Failure
SRAM	Static Random-Access Memory
SSID	Secondary Station Identifier
TID	Total Ionizing Dose
TMR	Triple Modular Redundancy
UART	Universal Asynchronous Receiver/Transmitter
UI	Unnumbered Information
WOD	Whole Orbit Data

Acknowledgements

The following people deserve more acknowledgement than they are receiving here.

- **Prof. W.H. Steyn** for accepting me as a student and showing me how to be a better engineer.
- **Pieter Botma, Christo Groenewald, Jako Gerber, Mike-Alex Kearney and Willem Jordaan** for their wisdom, knowledge and inexhaustible patience.
- **Gerhard Janse van Vuuren and Jan-Hielke le Roux** for being the best.
- **My parents, Riki Schutte and all my friends** for the unconditional love, support and motivation.

Chapter 1

Introduction and Problem Description

1.1 The CubeSat Standard

As has always been the case in most disciplines of engineering, there is a drive in satellite engineering to do more with fewer resources and a lower cost budget. This has led to the development of the CubeSat standard, initially presented in [11]. A 1 unit (1U) CubeSat has dimensions of 10 x 10 x 10 cm and weighs roughly 1 kg. CubeSat units can be added together to form larger satellites such as 2U (10 x 10 x 20 cm) or 3U (10 x 10 x 30 cm) satellites.

There are two main advantages to using the CubeSat standard. Firstly, adhering to a set of standards during development can significantly decrease the development time and cost of future projects that share the standards [11]. Space qualified components required to assemble a complete CubeSat can easily be purchased, removing the requirement to develop new components for each subsystem. Secondly, the implementation of the standard has enabled the development of a standard CubeSat deployer shown in Figure 1.1.

Figure 1.1 – The P-POD launcher [1].

The development of the Poly Picosatellite Orbital Deployer (P-POD) and the relatively light weight of CubeSats makes it easy to piggyback a CubeSat on the launch of a larger satellite or share the cost of a launch between many small satellites.

1.1.1 CubeSat in the ESL

The Electronic Systems Laboatory (ESL) at Stellenbosch University has already developed a number of subsystem components that conform to the CubeSat standard. The main components among these include:

- **CubeComputer:** An Attitude Determination and Control System (ADCS) On-Board Computer (OBC) that can also operate as the main OBC for a CubeSat;
- **CubeSense:** A sun and horizon sensor combination;
- **CubeControl:** An ADCS actuator system for magnetic control and reaction wheels.

Figure 1.2 shows a stack containing CubeComputer, CubeSense, and CubeControl.

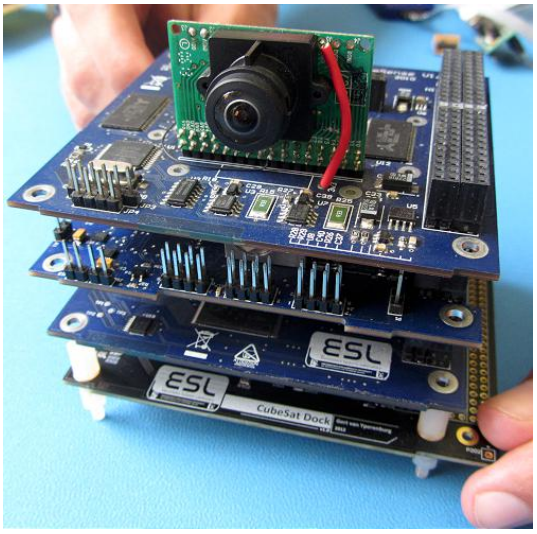


Figure 1.2 – A stack containing CubeSense, CubeControl, and CubeComputer

Although these components have flown on multiple satellite missions, the ESL has yet to launch a CubeSat assembled exclusively at Stellenbosch University. The QB50 project is an international effort to launch a network of 50 satellites into Low Earth Orbit (LEO). The main payloads of these satellites will be scientific sensors for measuring key parameters in the lower thermosphere. Parameters involved in the re-entry process will also be measured and compared with expected trajectories and orbital lifetimes[12]. The ESL is currently developing a CubeSat named ZA-AeroSat as its contribution to the QB50 project.

1.2 Satellite system overview

In order for a satellite to function effectively, a number of on-board subsystems have to work in unison. This section provides a brief overview of the subsystems generally found on a CubeSat.

On-Board Computer. The OBC is the centre of all the activity on board the satellite. It interfaces with all the other subsystems in order to provide services such as command and data handling, health monitoring and spacecraft timekeeping.

The command and data handling system serves two major purposes [13]. Firstly, it receives, validates, decodes and distributes commands to other spacecraft systems. Decoded commands received from a ground station may also need to be encoded into a format that can be used by subsystems on-board the satellite. For example, the format of command parameters for a particular subsystem or the transmission of data via an I2C bus. Secondly, it gathers, processes, and formats housekeeping and payload data for use by on-board housekeeping procedures or downlink to a ground station. Data may also need to be stored in persistent memory if no ground station is available to downlink data to.

The OBC is also used to monitor the overall health of the satellite. Health assessment can be achieved in a variety of ways depending on the subsystem being assessed. Telemetry sensors can be used to check that subsystem parameters such as temperature or current values are acceptable. The return values of certain processes can also be checked against expected values to see if a system is functioning correctly. Telemetry data collected from each subsystem is also stored for download to a ground station. A ground crew can then inspect the telemetry data to assess the state of the satellite. If error conditions are detected in any of the subsystems or processes, then corrective actions can be taken such as an OBC reset or disabling a part of the malfunctioning subsystem.

Electrical Power System. The Electrical Power System (EPS) on a satellite is responsible for providing, storing, distributing, and regulating power to the entire satellite system. It also supplies an interface through which it can communicate with the OBC. A conventional CubeSat EPS consists of solar cells, batteries, and power distribution and control circuitry [13].

Attitude Determination and Control System. The ADCS stabilises the satellite in the presence of disturbances and orientates it according to image capture or antenna pointing requirements. Sensors mounted on the satellite are used to determine the satellite's attitude and actuators are then used to control it. Passive control methods such as aerodynamic or gravity gradient control can also be used to place the satellite in a certain attitude without the use of actuators. The ADCS system is also responsible for managing the use of the propulsion system during orbital maintenance and manoeuvres.

Telecommunications System. The telecommunications subsystem provides the interface between the ground station and the satellite and is responsible for transmitting telemetry packets and receiving telecommand packets. It generally consists of antennas and transmitter and receiver circuitry. This transceiver hardware usually contains redundant components in order to ensure communication with the

satellite will be possible. Another responsibility of the communications subsystem is the encoding and decoding of packets using the chosen protocol for the mission. Studies done in [14] and [15] show that the most common protocol used in CubeSat mission between 2003 and 2012 was AX.25. This decoding and encoding of communication protocol packets is not to be confused with the decoding and encoding of telecommand and telemetry packets occurring in the command and data handling system.

Payload. The payload is the hardware and software on the satellite that performs the fundamental task for which the satellite was flown. In other words, it is the component of the spacecraft that interacts with the outside world to accomplish the mission's objectives [13]. As such, each mission usually carries a unique payload. Examples of payloads include observation instruments, communication systems and scientific experiments.

1.3 Challenges in the space environment

In most embedded systems, the system should ideally be able to run indefinitely without a reboot or maintenance after initialising. In a satellite system, this is more a requirement than an ideal due to a number of unique challenges that are posed by the remote space environment. Once a satellite is in orbit, it is impossible to physically interact with it. Inspecting and maintaining the integrity of the on board subsystems is therefore a task that needs to be performed by the OBC. Fault isolation and repair procedures also need to be selected and executed in order to ensure the satellite can continue operating in the presence of damage or faults within a subsystem.

A satellite is only in range of a ground station for a small percentage of its orbit. For the rest of its orbit, the satellite needs to be able to operate autonomously without assistance from a ground station. The OBC therefore needs to support the scheduling of processes for execution at a point when the satellite is not in range of the ground station. Telemetry and payload data collected during the orbit also need to be stored in persistent memory for download when a ground station comes into range.

Radiation in space can cause many faults within a satellite system. Most CubeSats operate in LEO where they are protected from the majority of high energy charged particles by the earth's magnetic field. However, the South Atlantic Anomaly (SAA) is a region where the Van Allen belts extend into the LEO range, allowing a large amount of charged particles into the atmosphere. Figure 1.3 shows the flux intensity map for regions experiencing energy levels bigger than 38 MeV.

The effects of this radiation on electronic equipment are separated into two areas namely Total Ionizing Dose (TID) and Single Event Effects (SEE)[16].

- **The Total Ionizing Dose** of electronic equipment is a measure of the amount of radiation a component can withstand before it degrades beyond a reliable state. TID radiation is built up from trapped electrons, trapped protons, and solar flare protons in the device.

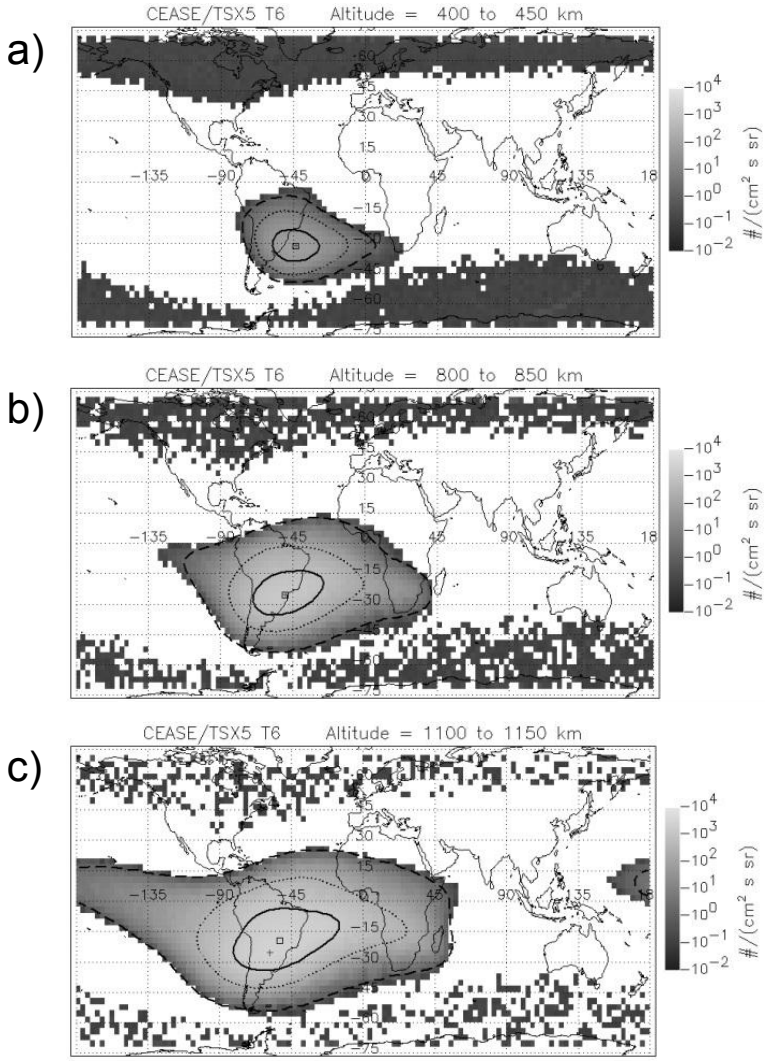


Figure 1.3 – Flux intensity map for energy levels > 38 MeV at a) 400 km, b) 800 km and c) 1100 km. Adapted from [2]

- A **Single Event Effect** (SEE) occurs when a charged particle deposits sufficient energy into the device. Common SEEs include the Single Event Upset (SEU) and the Single Event Latchup (SEL). A SEU generally manifests as a bit-flip in memory or a transient pulse in combinational logic. SEUs generally cause undesired effects in the software of a system. On the other hand, a SEL can permanently damage the hardware of a system. A SEL occurs when a charged particle creates a parasitic short circuit within a transistor resulting in excessive current flow. The only way to remove the latchup is to power cycle the device.

The occurrence of one of these faults has the potential to cripple a satellite system and end a mission prematurely, especially if they go undetected. Various hardware and software fault tolerance techniques are therefore required to increase the ro-

bustness of a satellite system. These techniques are further discussed in Chapter 5.

1.4 Software reuse

Beck *et al.* found that reusing as much software as possible is an important factor in increasing product quality while reducing costs and development times [17]. It is therefore logical to develop software to be easily applicable to different projects. Furthermore, Pasetti and Pree state in [3] that, in order for software reuse to be truly effective, the full architecture of the software system must be made reusable. Merely reusing pieces of code from an existing system does not transfer the initial intellectual investment to the new project.

Conventional methods used to achieve these reusable architectures include Domain-specific Architectures and, similarly, Software Frameworks [18]. Software Frameworks provide the skeleton for a set of applications or "domain". It should then be possible to develop any application within the specific domain as an extension of the architecture provided by the framework. This is achieved by constructing the framework out of components with a defined composition and interaction. Frameworks also make provision for the addition of application-specific code according to the requirements of the domain. Frameworks therefore reuse both code and design.

Figure 1.4 shows the concept of a framework and how it interacts with application specific components. In the figure, the shaded components are application specific while the unshaded areas represent the reusable architecture.

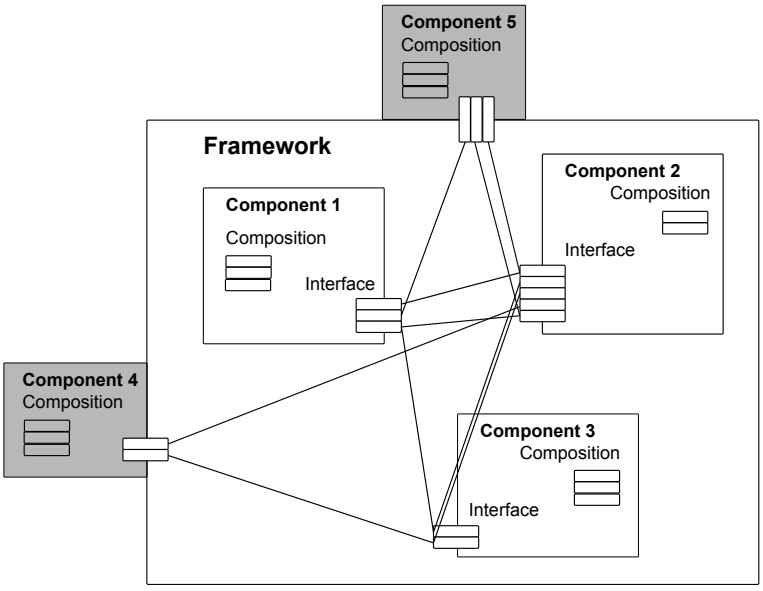


Figure 1.4 – Framework Concept. Adapted from [3]

1.5 Project goals

This chapter introduced the CubeSat standard and gave a summary of the ESL's involvement in CubeSat development. A brief description of each of the subsystems generally found in a CubeSat was given as well as an overview of the challenges faced when operating in a space environment.

From the above information, it is clear that even a small satellite is a complex system consisting of smaller, individual subsystems. In order for the satellite to be fully operational, each subsystem needs to operate correctly and in unison with the rest of the satellite. The satellite system should also be robust enough to enable continuous operation in the harsh environment of outer space. To satisfy all of these requirements, flight software is implemented on the main OBC of the satellite to manage and monitor the system and interface with the ground station. The main goal of this project is to develop reusable flight software for a CubeSat. As an additional requirement, the software will be developed using the FreeRTOS Real Time Operating System (RTOS).

The goals of the project outlined in this thesis are therefore as follows:

- To design and develop flight software for a CubeSat;
- To implement the flight software using FreeRTOS;
- To develop the flight software with a generic architecture so that it is easy to modify and reuse;
- To integrate and test the flight software on CubeSat OBC hardware.

1.6 Brief Chapter overview

This section provides an indication of what is covered in each chapter.

1. Chapter 2 provides an overview of the functionality required from flight software. Background information on the capabilities and tools provided by a RTOS is also presented followed by a quick summary of the OBC hardware.
2. Chapter 3 covers the initial design phase of the flight software. Firstly, the chosen RTOS is evaluated against similar products and it's configuration for the project is discussed. Secondly, the general structure of the flight software is developed. Most notably, the structure of the managers for each subsystem and their interfaces. Various initial design choices concerning aspects of the development (e.g software robustness, satellite software standards, etc.) are also discussed and motivated. Finally, the development and testing set-up used during the project is presented.
3. Chapter 4 presents all the components of the flight software involved in command and data handling processes. Initially the design of the I2C manager is presented as it is the pathway used for almost all commands and data.

Command management and scheduling are then discussed as part of command handling followed by an explanation of the data upload and download procedures and the file system interface.

4. Chapter 5 focuses on the housekeeping procedures and fault tolerance mechanisms. The processes involved in the collection, processing and storage of housekeeping data are presented as well as the mechanisms used to increase the robustness of the software.

5. Chapter 6 presents the various tests that were used to evaluate the functionality and performance of the flight software.

6. Chapter 7 concludes the thesis with a summary of the design, development and evaluation process that was followed in this project. Possible improvements to the final version of the flight software are also discussed.

Chapter 2

Flight Software Background Information

In Chapter 1, the need for flight software on satellites was established by examining the complexity inherent to even small satellite systems. It was also noted that the management of this complexity is further aggravated by the fact that satellites operate remotely (autonomously) and do so in the harsh environment of outer space. This chapter details the main background information required for the development of generic satellite flight software using a RTOS.

In the first section, the requirements for a generic flight software application are derived. Next, the main functions of a RTOS as well as tools generally included in a RTOS are discussed. An overview of embedded fault tolerance techniques that can be used to increase the robustness of the flight software are also presented. Finally an overview of the hardware and drivers that will act as a platform for the development of the flight software is given.

2.1 Requirements of flight software

Before the design of a system can begin, it is important to define its requirements. For this project, a balance had to be found between the project goals of producing functioning flight software and producing a generic application. In other words, enough complexity had to be included in the software to make it applicable to any mission while not over-developing so that it became difficult to adapt for new systems. The simplest way to do this was to look at the flight software design of a number of existing microsatellites and existing flight software frameworks. In this way, the most common and essential parts of flight software could be identified and included in the generic application. Components that would have to be modified extensively depending on the mission of the satellite were identified as "mission-specific". Although provision would be made for these components in the generic application, their implementation would be largely left to the user. A few of the cases that were studied to identify the requirements of flight software are listed below.

Generic Components	Mission-specific Components
Health and Housekeeping	
Housekeeping collection	
Housekeeping storage	
	Recovery procedures
	Error handling
Command handling	
Command Handling	
Command Scheduling	
	Command processing
Subsystems	
Subsystem management	
Payload management	
	Subsystem interfaces
	Satellite mode management
Data Handling	
Data Storage	
	Data Formats

Table 2.1 – Generic and Mission specific Flight Software Components

- The **UPMSat-2** microsatellite runs OBC software that controls telemetry and telecommand reception and transmission; measurement, processing and storage/transmission of sensor data; and payload command and data management. [19]
- A **Pattern-based framework** is presented by [20] to guide the development of an architecture for satellite flight software. The main packages defined in the framework include payload management; navigation and control; communications; subsystem management; and supervising and monitoring.
- The flight software of **SwissCube** contained modules for handling command Management and scheduling, data storage, modes of operation, and housekeeping [21].
- The **CKUTEX** microsatellite defined modules called Computer Software Components for ADCS, Command and Data Handling; EPS; Telemetry, Tracking and Command and Housekeeping and Error Handling [22].
- The **NTNU** test satellite runs OBC code that handles system control, housekeeping, data storage, payload processing and command management [23].

From the cases above, a number of generic software components could be identified. Table 2.1 shows the components identified as generic and those identified as mission-specific. The primary goal of the project would be to implement all the functionality listed in the “Generic Components” column.

Another responsibility of flight software is to ensure that on-board processes meet any real-time requirements they might have. The different types of real-time requirements and examples of subsystems containing these requirements is given in Section 2.2.

2.2 RTOS

One of the project goals was to implement the flight software using FreeRTOS. This section presents the basic functionality and tools included in most RTOSs. A RTOS can be used to greatly simplify the management of the various aspects of a real-time embedded application. Conventional operating systems provide an interface between an application and the hardware on which the application runs. A real time operating system is therefore an OS that is tailored to meet real time requirements in applications with time-critical functions [24]. According to [24], there are three types of real time requirements that can be identified namely:

- Hard requirements: Missing a deadline results in failure of the system.
- Firm requirements: Small tolerance for missed deadlines. System will continue to run but with a generally unacceptable reduction in performance.
- Soft requirements: Deadlines may be missed. Loss of performance due to missed deadlines can be recovered from.

A single system can consist of any combination of hard, firm, and soft requirements depending on its function. However, the more hard real-time requirements are placed on the system, the more deterministic it is considered to be. Determinism is a measure of how predictable a system is. The more predictable the execution timings in a system are, the less likely it is to miss a deadline.

The various components of a RTOS are briefly discussed below.

2.2.1 Multitasking

The main advantage of using an RTOS in an embedded system is the multitasking capability that it adds to the application. Although a single Central Processing Unit (CPU) can only process one thread of execution at a time, the required functions of the application can be divided into separate processes called tasks. The tasks in a system are managed by a scheduler that chooses which task to run according to a scheduling algorithm. Each task has its own entry point and is assigned its own stack from the heap. Its execution will normally be done in an infinite loop that continues running unless the task is removed from the scheduler managing the tasks. The CPU can then rapidly switch between processing the different tasks to achieve the effect of simultaneous execution. The process of saving the state of a task, and switching execution to another is know as a context switch [4]. During operation, a task will switch between various states of operation. The basic states a task can exist in are listed below.

- **Running**: A task in the Running state is the task currently being executed by the CPU.
- **Ready**: Tasks that are not currently being executed but are available to the scheduler are in the Ready state. If a task is preempted by the scheduler, it is transitioned into the Ready state to allow the CPU to execute the next task selected by the scheduler.
- **Blocked**: Tasks in the Blocked state wait for events to occur that transition them to the Ready State. For example, an interrupt or the expiration of a delay.
- **Suspended**: Tasks in the Suspended state are not available to the scheduler. They will never be selected by the scheduler unless a separate task or interrupt transfers them from the Suspended state to the Blocked state.

Figure 2.1 shows the possible transitions between these states.

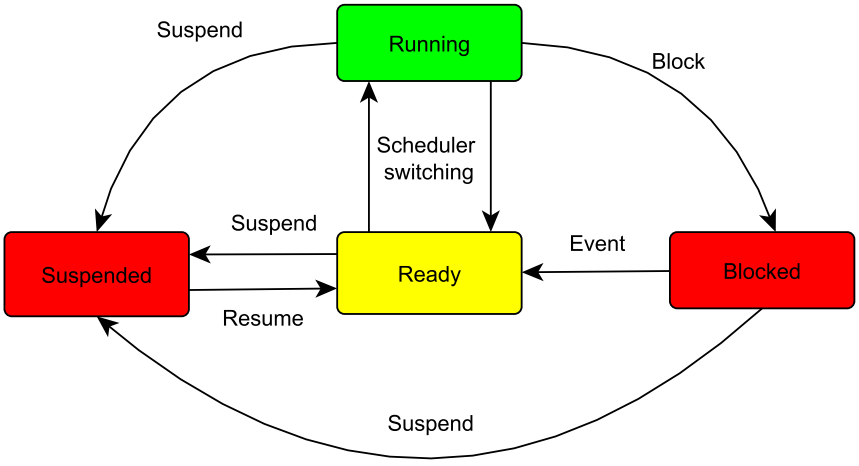


Figure 2.1 – Task state machine. Adapted from [4]

2.2.1.1 The Scheduler

In order for multiple processes to be running simultaneously, each task is added to a scheduler. The scheduler then switches between each task according to the scheduling algorithm being used. There are various methods of scheduling that can be used according to the needs of the application [25].

- **Cooperative Scheduling** allows tasks to run until they specifically yield to the next task awaiting execution. This ensures that a task will always be able to achieve a certain amount of execution before being blocked but also trusts tasks to execute in a timely manner. Any delay or lack of response in one task will result in the whole system slowing down.

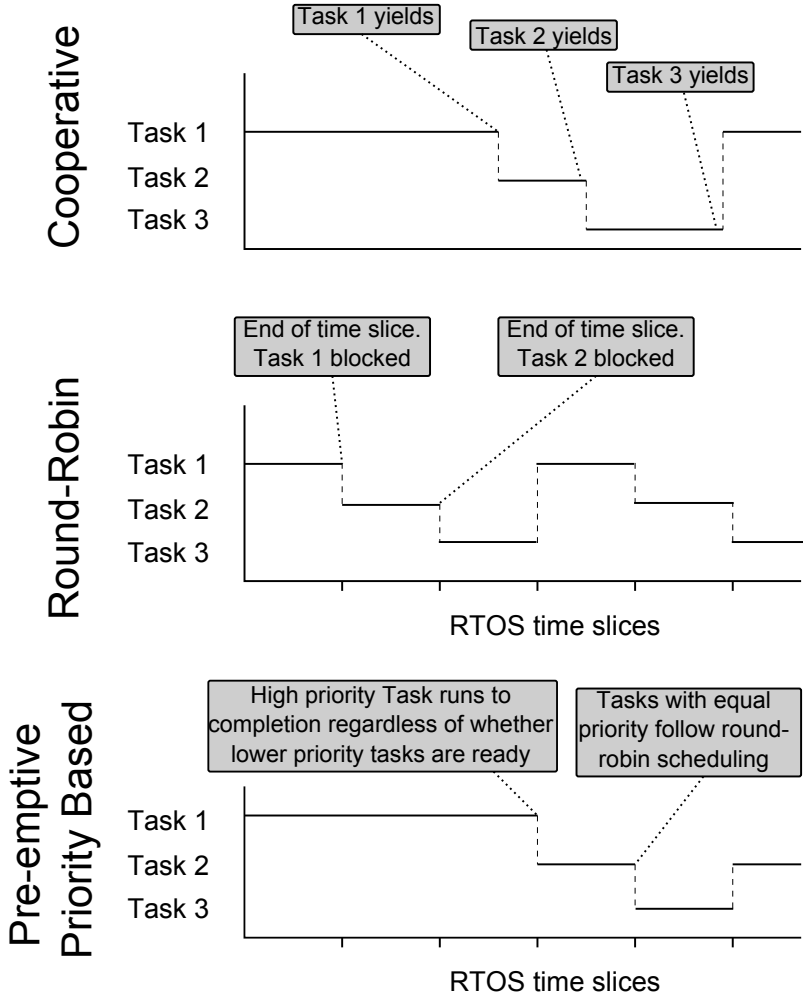


Figure 2.2 – Examples of common scheduling algorithms

- **Round-Robin Scheduling** assigns each task an equal time slice to execute before it is replaced with the next task awaiting execution. This ensures that each task gets CPU time but the absence of priorities means that important tasks with strict deadlines receive the same CPU time as less important tasks.
- **Pre-emptive Priority-based Scheduling** requires developers to assign priorities to tasks. Running tasks are immediately pre-empted when a task with a higher priority becomes available for execution. When two tasks of the same priority are scheduled to execute, a simpler algorithm (such as round-robin scheduling) can be used. This method is more deterministic than the above mentioned algorithms and ensures that tasks with hard real-time requirements are executed ahead of tasks with soft requirements. However, assigning priorities incorrectly could cause certain tasks to be completely starved of CPU time.

Figure 2.2 shows examples of the three procedures described above. Less common

forms of scheduling include time partition scheduling, deadline scheduling and priority decay scheduling. Further information on these can be found in [25].

2.2.1.2 Task Communication

In a multitasking environment, there are rarely tasks that operate independently of other processes. Certain tasks may require the results of a separate process in order to complete or need to notify other tasks of an event or failure. For this reason, a RTOS will include mechanisms for intertask communication such as queues, pipes, messages and mailboxes, and event flags [26].

2.2.1.3 Semaphores and Task Synchronisation

Task synchronisation can be achieved through the use of a semaphore. A semaphore can be thought of as an item that can be given or taken by a task. The most basic type of semaphore is the binary semaphore. A binary semaphore can only be taken once. If a task attempts to take a binary semaphore that is not available, that task will go into a blocked state until a separate process gives it the semaphore. In this way, it is possible to synchronise the execution of two tasks.

Tasks can be synchronised with interrupts and other tasks by first requiring a semaphore to be given from the interrupt or task before running. In other words, when synchronising with an interrupt, a task will run to a point and then remain blocked until an interrupt occurs which gives the semaphore to the task. A semaphore can also exist as a queue called a counting semaphore. This allows, for example, multiple instances of the same interrupt to occur while a handler task is running. When the handler task completes, a number of semaphores will be available to be received and the handler task will execute until the counting semaphore queue is empty [4].

2.2.2 Resource management

In any application with multiple threads of execution, access to a resource that is shared between tasks must be managed using mutual exclusion. Without proper management, a task using a certain resource could be replaced by a higher priority task before it is done with the resource. This would leave the resource in an unknown state possibly resulting in data corruption and errors in any task that uses the resource in the future.

A mutex is a type of binary semaphore that is used to control access to a shared resource [4]. When a shared resource is protected by a mutex, a task that wishes to make use of the resource must first acquire the mutex. If a different task has already acquired the mutex, the first task will go into a blocked state until the mutex becomes available again. Tasks must always make a mutex available for acquisition once they are done with its associated resource. If they do not, no other task will ever be able to access the resource. Figure 2.3 illustrates the main difference between a mutex and a semaphore. A “give” operation can be performed on a semaphore even if a task is still processing the event that previously caused the semaphore to be given. This makes semaphores ideal for task synchronisation. Conversely, a mutex must be returned before another task can obtain it making it ideal for resource management.

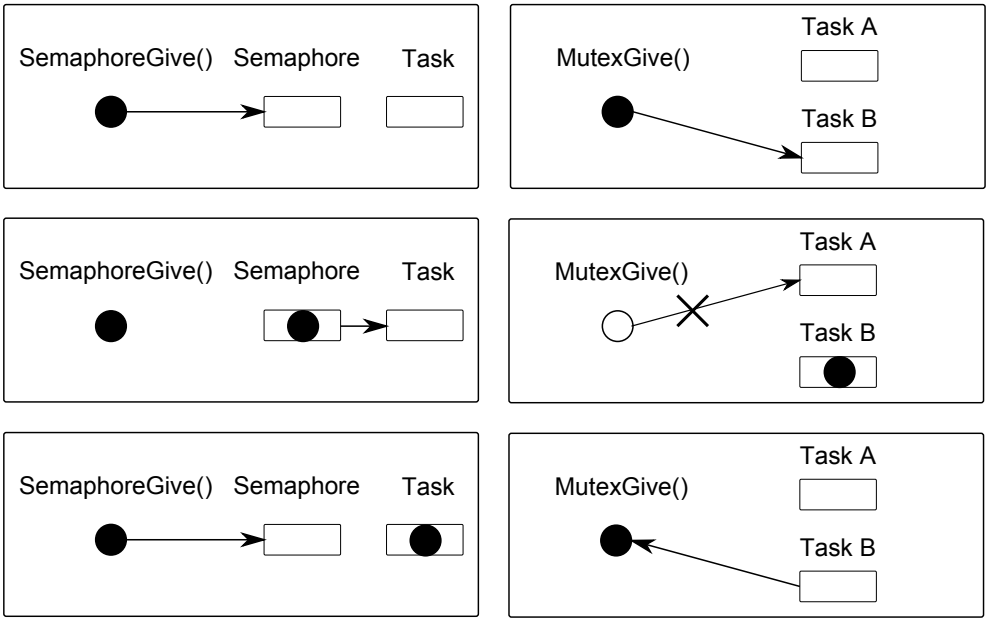


Figure 2.3 – Difference between the availability of a semaphore and a mutex. Adapted from [4].

There are two common problems associated with mutual exclusion [4]. The first is priority inversion. This occurs when a task with a high priority is waiting for a task with a lower priority to release a mutex. The second problem is known as a deadlock. A deadlock occurs when two tasks are blocked because they are each waiting for a resource that is held by the other. The best way to avoid both these problems is to consider their potential occurrence during development.

Critical sections, or critical regions, also require a form of mutual exclusion. A critical section is used to ensure that only one thread of execution can access shared data or a shared resource [27]. Critical sections can be implemented by disabling interrupts while the shared resource is being accessed. This will also prevent a preemptive context switch so the currently running thread will be the only running thread. However, disabling interrupts in this way can drastically decrease the performance of the system. Especially a system with hard real time requirements. This method of implementing critical sections can be improved upon by only disabling the scheduler and leaving interrupts enabled.

2.2.3 Memory management

Memory management is another function performed by an RTOS. It refers to the management of the allocation and deallocation of the memory required by kernel objects such as tasks and queues. This management can be simple or complex depending on the way the application functions. Applications that only make use of static memory allocation only require heap memory to be allocated before the application starts. This amount of memory will then stay constant for the entire lifetime of the application.

Applications that dynamically create and remove tasks and queues have extra complexities to consider such as memory fragmentation. Most applications allocate the memory they require before the scheduler is started and then keep that memory allocated for the lifetime of the application. In this case, or if the memory being freed and re-allocated is always the same size, fragmentation will not be a problem. However if tasks of different stack sizes are deleted and then reallocated, a sufficient memory management algorithm will need to be implemented. Without a memory management algorithm that meets the needs of the application, memory fragmentation will occur, slowing the system down and leading to a loss of determinism. With sufficient fragmentation, the system could also appear to run out of memory when in fact there are many small blocks of memory available that are too small to meet the system requirements.

2.3 Fault Tolerance

Fault tolerance is a vital component in any autonomous system. This is especially true for satellite systems due to the conditions discussed in Section 1.3. According to Torres-Pomales, fault tolerance generally refers to the use of techniques to increase the likelihood that the final design embodiment will produce the correct outputs [28]. Detecting and managing faults is a complex procedure that needs to be tailored specifically for the type of system and the environment it operates in. It is also inefficient to safeguard a system against faults that will never occur. This section provides an overview of faults and fault tolerant techniques relevant to satellite systems

2.3.1 Possible faults

In order to equip software with the necessary tools to remove and recover from faults, it is important to define exactly what can go wrong. Appropriate fault tolerance mechanisms can then be put in place according to the specifics of the application.

According to Butler, a fault is a defect in hardware or software that can lead to an incorrect state [29]. These defects can result from a deficiency in the structure of the system such as a logical design flaw, programming error, compiler error or manufacturing defect. Faults can also be generated by external factors such as SEUs or environmental damage to hardware. The frequency or period that a fault is present for will also influence the best way to handle it. Faults can be classified as:

- Transient faults that appear for a short period and then disappear;
- Permanent faults that remain in the system unless they are removed;
- Intermittent faults that periodically appear and disappear.

Certain faults, known as common cause faults (CCF), can cause errors in different areas of the system simultaneously. CCFs usually occur if there is a fault at a single point of failure (SPOF) in the system such as a single power source.

Errors can be defined as the impact of a fault on the system's state. An error is therefore the result of a fault upsetting an operation in the system. It is important to note that the duration of a fault does not necessarily determine the duration of the error it causes. A transient or intermittent fault can corrupt or damage parts of the system that are persistent between different states, thus causing a permanent error. Comparatively small faults should therefore be treated as having the potential to cause extensive damage. Failures occur when the service delivered by a system malfunctions or ceases due to the presence of errors.

Techniques used to prevent or mitigate faults can be implemented on an architectural or application level. The following sections discuss the difference between these two categories and the various techniques they include.

2.3.2 Architectural level fault tolerance

Architectural level fault tolerance refers to techniques that are applied during the development of the system architecture. As the topic of this thesis is the development of software, the various techniques to implement fault tolerance in the hardware architecture will not be covered. The hardware fault tolerance implemented on the CubeComputer OBC will be briefly discussed in Chapter 5 to motivate choices in the software fault tolerance design.

One method to enhance the fault tolerance of a system is to develop its architecture to be modular [28]. When developing a modular system, the software engineering concepts of coupling (the level of interdependencies between modules) and cohesion (the level of relation between functions in a single module) are very important. Developing a system with low coupling automatically raises the fault tolerance of an application by decreasing the likelihood that faults can spread beyond the modules they originate in. The clear interfaces and confinement of functionality that modular programming provides also simplifies the testing of software. This increases the likelihood that programming errors will be found during development.

Implementing the functionality of a module using atomic actions increases the fault tolerance of an application in a similar way to modular programming. An atomic action uses an exclusive set of components to perform its function. While the action is being performed, there is no interaction with any components not required to perform the function. The action therefore appears as a single, indivisible, instantaneous action to the rest of the application. Should an error be detected during the atomic action, then that error will be confined to the components taking part. Error recovery procedures for the specific error will then only have to take those components into account.

2.3.3 Application level fault tolerance

Application level fault tolerance refers to fault tolerance techniques that are implemented by the application code. At this level, a distinction can be made between single-version and multi-version software fault tolerance techniques. Single-version techniques improve the fault tolerance of an application within a single version of

the software. These techniques include mechanisms to handle the detection and containment of errors as well as any recovery processes.

Multi-version techniques make use of multiple versions of software, executed sequentially or in parallel, to ensure that failure in one version does not cause a system failure. In this way, the fault tolerance of an application is increased by using alternative methods of error detection in the different versions, performing replication checks or implementing majority voting [28]. However, the cost of implementing multi-version fault tolerance techniques is too high in terms of both system and financial resources to be able to implement on a CubeSat. The scope of this thesis therefore does not include multi-version techniques and only single version techniques will be expanded upon below.

2.3.3.1 Single-version software fault tolerance techniques

Regardless of the actual technique used, there are four steps to implementing single version fault tolerance techniques. These are Detection, Confinement and Assessment, Recovery and Fault Treatment [30].

Detection of errors in single version software requires the software to have knowledge of the intended state of the system. Different checks can then be performed do determine when the current state deviates from the intended state.

- Environmental checks such as checking for dangerous temperatures, overcurrents and low power levels.
- Timing checks such as a watchdog timer for detecting unresponsive applications or timeout checks for detecting unresponsive subsystems.
- Coding checks using redundant data such as a checksum.
- Reasonableness checks such as checking that outputs are within predefined threshold values.
- Structural checks such as checking data for null pointers.

When implementing fault tolerance in a modularised application, it is important that each module contains self-protection and self-checking. Implementing self-protection means that a module protects itself against inheriting faults from the rest of the system. This could happen during interaction with a separate module. Implementing self-checking means that a module checks its own outputs to ensure that faults originating in that module do not spread to the rest of the system [31].

Confinement and assessment procedures limit the extent to which an error can affect the system. Confinement is mainly achieved on an architectural level by developing software to be modular. By defining interfaces to modules and containing operations within modules it becomes easier to contain errors within the module that they occur. Once an error is detected, damage assessment procedures can be run on components associated with the process that produced the error to determine the extent of the damage to the system.

Recovery techniques can be separated into two categories: forward recovery and backward recovery. Forward recovery attempts to correct or suppress errors in order to allow the system to continue functioning. Even if it is functioning in a less optimal state. Redundant information can be added to data in the form of Error Correcting Codes (ECC) that can detect and correct bit flips. The process of "scrubbing" memory refers to the process of recovering corrupted memory by storing data with ECCs and then periodically evaluating the codes to determine if corruption has occurred [29]. If faults cannot be removed in this way, then redundant software can be enabled to replace the faulty section or outputs from the faulty software can be ignored. Backward error recovery attempts to return the system to a previous state in which it was functioning correctly. These techniques can be as simple as resetting the OBC or can grow very complex as with the checkpoint and restart method [28]. Once an error is detected, the system is returned to a previous state that is either predefined or dynamically saved as a "checkpoint" during operation. Backward error recovery carries the advantage of being able to recover from unanticipated, transient errors.

In order to ensure successful **continued service**, it is important that operators can determine the cause of the fault even after it is successfully removed. New versions of the application software may need to be developed in order to remove bugs in the code or provide tolerance against faults that were initially unanticipated. For this reason it is important that a system has the capability to store information about its operation and the detected errors.

2.3.3.2 Watchdog timers

One of the simplest and most common fault tolerance mechanisms is the watchdog timer. A watchdog timer is a piece of hardware usually found as a peripheral unit built into a microcontroller unit (MCU). It consists of a timer that counts towards a fixed, predetermined value. The system's software needs to reset the timer to its original value before it times out. If a time-out occurs, it means the system has become too unresponsive to continue functioning and a system reset is asserted [32]. The choice of the counter's value needs to strike a balance between being small enough to respond to errors quickly, but big enough to avoid unnecessary resets due to execution variation. Although the basic way to handle a watchdog time-out is to reset the system, other actions can also be taken before the reset such as recording system information for debugging or ensuring the system boots up in a safe mode or state.

2.3.4 Conclusion

This section presented some classes of faults that can occur in software applications as well as some common techniques used to increase the robustness of software against these faults. There are a great many more fault tolerance techniques used in embedded systems and other software applications than have been listed here. However, many of these require more resources that are generally available for CubeSat projects and are therefore not considered relevant. The design and implementation of the techniques used in the flight software is presented in Chapter 5.

2.4 Existing hardware and drivers

This section provides a brief overview of the existing hardware and software produced in the ESL that the flight software will be built upon. Firstly, the CubeComputer main OBC will be discussed to provide an idea of the typical hardware on which the flight software will run. Secondly, the various drivers developed for CubeComputer will be discussed as they provide access to the various features and functionalities of the board.

2.4.1 CubeComputer

CubeComputer uses the EFM32 Giant Gecko as its main Micro-controller Unit (MCU). The Giant Gecko has, as its processor, a 32-bit ARM Cortex-M3 that runs at up to 48 MHz. Up to 1024 kB Flash and 128 kB of RAM are available as well as a number of energy efficient, autonomous peripherals[33]. These include a Real Time Counter, Timers, ADCS, External Memory Interface, Universal Asynchronous Receiver/Transmitter (UART), Serial Peripheral Interface (SPI), and Inter-Integrated Circuit (I2C). A number of low energy modes are also available. Memory and data storage features that are included externally on CubeComputer include

- 256 KB of EEPROM.
- 4 MB of Flash for Code Storage.
- 2 x 1 MB of external Static Random-Access Memory (SRAM) for Data Storage. These include SEU protection via a Field Programmable Gate Array (FPGA)-based Error Detection and Correction (EDAC) and SEL protection by detecting and isolating latchup currents.
- MicroSD card socket supporting microSD card capacities up to 2 GB.

Figure 2.4 shows a block diagram of the CubeComputer feature layout.

More detail on the characteristics and capabilities of CubeComputer can be found in [5].

2.4.2 Board Support Package

The CubeComputer Board Support Package (BSP) is a collection of libraries containing drivers for the various features on the CubeComputer OBC. The following libraries were available at the time of writing.

- **bsp_acmp:** Initialises Analogue Comparator channels to detect latchup currents in the SRAM modules.
- **bsp_adc:** Controls the ADC module.
- **bsp_boot:** Interfaces with the boot table located in the external EEPROM and flash.

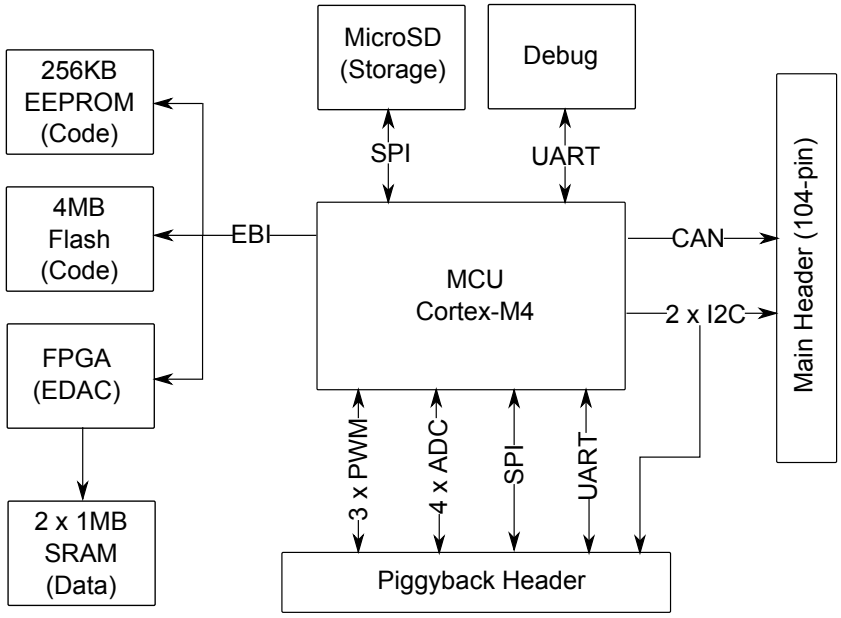


Figure 2.4 – CubeComputer Block Diagram. Adapted from [5]

- **bsp_dma:** Contains functions to initialise the Direct Memory Access (DMA) module.
- **bsp_ebi:** Contains functions for initialising the External Bus Interface (EBI), enabling and disabling the SRAM modules, and writing data to the EEPROM memory.
- **bsp_i2c:** Used to control and use the I2C bus.
- **bsp_rtc:** Used to control the Real Time Clock.
- **bsp_see:** Contains function to assist with memory scrubbing as well as latchup detection and removal.
- **bsp_uart:** Used to initialise and interface with the CubeComputer UART channels.
- **bsp_wdg:** Contains functions to control the Watchdog Timer peripheral.

Drivers to initialise and interface with the microSD card and FATFS file system are also included. These are discussed in further detail in Chapter 4. The BSP also includes an Eclipse project pre-programmed with test functions that can be used to test the features of CubeComputer.

2.5 Conclusion

This chapter derived the requirements for generic flight software by examining industry standards, existing flight software framework designs, and the flight software

designs for existing microsatellites. ADCS, EPS, communication subsystems, payload hardware and a main OBC were found to be the hardware subsystems most commonly found on microsatellites. It was also found that the OBC hardware is responsible for command and data handling, housekeeping procedures and software fault tolerance. The real-time needs of satellite flight software were also identified.

A discussion of the capabilities and tools generally provided by a RTOS was given as well as an overview of software fault tolerance techniques relevant to the flight software design. Existing hardware and software components that will be used to develop the software were also presented. These included the CubeComputer main OBC and the CubeComputer BSP.

The following chapters show how the information provided in this chapter was used to design and implement the flight software application. Chapter 3 covers the design of the structure of the flight software and the implementation of FreeRTOS. Chapters 4 and 5 then cover the design and implementation of command and data handling and housekeeping respectively.

Chapter 3

Flight Software Design Phase

Having established the requirements of the flight software as well as the tools that would be used during development, a number of design choices had to be made. This chapter starts by choosing particular software engineering techniques and principles to guide the design and development process. The configuration settings chosen to adapt FreeRTOS for the flight software application are also presented. With a base on which to build the software established, the structure of the software as well as the interface between the satellite and the ground are then defined.

3.1 Structural Design Choices

From the start of the project, two design choices were made that would influence the structure of the whole application. Both of these choices were made in the interest of keeping the software robust and re-usable.

3.1.1 Modular Programming

The first choice was to use modular programming techniques as described in [34]. Modular programming divides an application into modules with each module handling a single function within the application. Each module consists of an interface and an implementation. The interface allows other modules to interact with the implementation while all the details of the implementation are hidden from the rest of the application. As will be discussed in relevant sections of this report, the interfaces to most of the modules in the flight software consist mainly of command queues that conform to a specific format and functions that return data such as housekeeping parameters. Having separate modules with clearly defined interfaces makes the software easier to adapt for specific missions and therefore makes the software more re-usable.

In the C programming language, the **#include** directive and the **static** keyword can be used to increase the modularity of software. This is because they help to clearly define whether certain variables, functions and data types are part of a module's interface or part of it's implementation. Specifically, the **#include** directive is used to assign an interface to an implementation and clearly define its dependencies. The choice to use a modular approach therefore conforms to the project aim of making

the software easy to use and expand upon. As stated in Section 2.3.2, a modular design also increases the fault tolerance of software.

3.1.2 Memory Allocation

Another basic design choice was to completely avoid the use of dynamic memory allocation within the application source code. The main reason for this decision is to maintain the software's robustness through the avoidance of memory leaks. A memory leak is when memory that is dynamically allocated within the source code is not freed when it is not longer required. Memory leaks can cause a system to slow down and eventually run out of memory. The only way to get rid of a memory leak in this case would be to reset the OBC. The malloc() and free() library functions are also unsuitable for a flight software applications because they cause memory fragmentation. Sufficient fragmentation in memory may even cause a call to malloc to fail despite heap space being available [35]. While the FreeRTOS kernel does use dynamic memory allocation to create tasks, queues and semaphores, this will mostly only be done before the scheduler is started. The memory management performed by the FreeRTOS kernel is discussed in 3.2.2.

The choice to use only statically allocated memory also meant that memory could become a limiting factor during development. The growth of the application code size would have to be monitored due to the fact that the application developed would not include mission specific libraries and sub-system interfaces. There would therefore still need to be ample memory left to add in any such code when tailoring the software for a specific mission.

3.2 FreeRTOS

As mentioned in Chapter 1, the use of FreeRTOS was one of the requirements of the project. This section motivates the suitability of FreeRTOS for this project by evaluating its specifications and comparing it against similar products. The configuration of the RTOS for the flight software is also covered.

3.2.1 Choosing a RTOS

In recent years, the RTOS market has experienced radical growth. While this provides a large variety from which embedded developers can choose the most suitable RTOS, it has also complicated the process of distinguishing between products that will help or hurt a project. Traditionally, measures of performance, functionality and compatibility with development tools are the logical criteria by which a RTOS is selected. Interrupt response latency, latency jitter, worst case interrupt response time, context switch time overhead and software timer jitter can all be taken into account when measuring the performance of an RTOS [36] [37]. The middleware support in a RTOS is also commonly considered. In [38], Moore identifies four main areas of middleware namely:

- **Networking** for data transfer and remote diagnosis.

- **File Systems** for data storage and logging.
- **Universal Serial Bus (USB)** for connecting to a USB device or PC.
- **Graphical User Interface (GUI)** for receiving visual output and entering commands via a display.

However, there is currently no RTOS that has been shown to be far superior to the rest in a technical sense leading to a requirement for other criteria that are less quantitative. Krasner states that the best RTOS for a project is the simplest and most intuitive RTOS that can satisfy the requirements of the system being developed [39]. This statement is the result of identifying elements that affect the "time to market" of a project. The "ease of use" of a RTOS can be measured by evaluating factors outside of its technical capabilities including:

- Simple services.
- Consistent naming conventions.
- Good documentation.
- Good support.
- Availability of source code.
- Availability of demonstration software for target hardware.

Aside from these factors, choosing a RTOS that has an existing port for the hardware being used for the system will speed up development time. Porting a RTOS to a new processor architecture can be very time-consuming due to the in-depth knowledge that is required of both the operating system and target hardware. Choosing a RTOS that is overqualified for the project will also have a negative impact on development time. Aside from the technical considerations of increased system overhead and memory footprint, using a RTOS with too much complexity results in an increased learning curve and a greater chance of misuse during development [39]. Although this project is not aimed at a market as such, it also carries a time constraint and the elements discussed above are therefore also applicable.

In an embedded market study done in 2013, FreeRTOS was the most widely used RTOS that was commercially available [40]. It also ranked highest in RTOSs that companies planned to use in the next 12 months. From a technical point of view, FreeRTOS is suited to a flight software application. It is a lightweight RTOS that is highly configurable depending on the requirements of an application. The following aspects are relevant to this project. All the points listed in the remainder of this section were taken from the FreeRTOS website [41].

- A flash footprint of around 5 to 10 kB when using a minimal configuration.
- Includes all the basic RTOS tools such as a configurable task scheduler, resource management mechanisms and timers.

- A context switch time of 84 CPU cycles using a Cortex-M3 with the compiler set to optimise for speed.
- Very configurable according to the needs of an application.
- Designed to be small and simple and therefore does not include middleware features such as networking support, graphical environments and web servers.

From an "ease of use" point of view, the following aspects are relevant.

- Basic kernel contained in 3 source files.
- Existing port to the Cortex-M3 microcontroller used on CubeComputer. Demonstration projects available to speed up the learning curve.
- Free open source code.
- Online API documentation.
- Written in C using a strict naming convention.
- Free monitored support forum.

FreeRTOS therefore satisfies both the traditional, technical criteria and appears easy to learn and use. It is a lightweight OS designed for small embedded systems with real time requirements making it ideal for a CubeSat system.

3.2.2 FreeRTOS Implementation

As a FreeRTOS port for the ARM Cortex-M3 exists, "installing" FreeRTOS in the BSP Eclipse project is a simple process. The first step is to include the header and source files that contain the functionality required from the application. FreeRTOS handlers then need to be installed for the SysTick, PendSV and SVCCall interrupt vectors. The BSP project uses the handlers defined in the EFM32 startup file for the ARM embedded workbench. In C, defining a symbol as "weak" allows it to be overwritten by a "strong" symbol of the same name. As the vector symbols are Cortex Microcontroller Software Interface Standard (CMSIS) compliant and were defined as "weak" symbols, the default handlers can be replaced by adding

```
#define vPortSVCHandler SVC_Handler
#define xPortPendSVHandler PendSV_Handler
#define xPortSysTickHandler SysTick_Handler
```

to the configuration file FreeRTOSConfig.h. Here, vPortSVCHandler, xPortPendSVHandler and xPortSysTickHandler are the FreeRTOS defined handlers.

3.2.2.1 Configuration

A number of parameters also have to be set in the configuration file, FreeRTOSConfig.h. Some of the main parameters that required specification are listed below.

- configUSE_PREEMPTION: Selects either the preemptive or cooperative scheduler.
- configTICK_RATE_HZ: Sets the frequency of the FreeRTOS tick interrupt which the kernel uses to measure time. Higher values mean the kernel can measure time with a higher resolution but requires more CPU time to be given to the kernel. The tick rate also indicates the speed at which the scheduler can switch between tasks. For example, a tick rate of 100 Hz means a context switch can potentially occur every 10 ms allowing 100 context switches per second. With the Giant Gecko's clock speed of 48 MHz, a FreeRTOS tick rate of 100 Hz would mean that each task would be run for about 480,000 clock cycles before relinquishing CPU control back to the scheduler.
- configTOTAL_HEAP_SIZE: Sets the total size of the heap available to the kernel
- configMAX_PRIORITIES: The number of priorities that can be assumed by a FreeRTOS task. Each extra priority that is added increases the amount of RAM required by the application.

Many other configuration options are possible such as the inclusion or exclusion of various task control utilities, semaphores and mutexes, coroutines and software timers. As with the number of priorities, each extra feature that is included increases the memory footprint of the operating system.

3.2.2.2 Kernel Memory Management

As discussed in Section 2.2.3, memory management is important to avoid fragmentation and preserve determinism. FreeRTOS provides four different memory management schemes from which one is selected according to the requirements of the overall application.

- heap1.c: Does not permit allocated memory to be freed once it is allocated. This implementation is used if no tasks, queues or semaphores are ever deleted. It is fully deterministic and will not result in fragmentation.
- heap2.c: Allows the freeing of allocated memory but does not include a coalescence algorithm (i.e. it does not combine adjacent smaller blocks of memory into larger ones). It should therefore not be used if the memory being freed and reallocated is not a constant size and happens unpredictably. This scheme is efficient but not deterministic.
- heap3.c: Implements thread safe versions of the C library functions malloc() and free(). This scheme is not deterministic and considerably increases the kernel code size.

- heap4.c: This scheme uses a first fit algorithm and includes a coalescence algorithm. It prevents memory fragmentation even when memory blocks of random sizes are freed and allocated. It is not deterministic.

In Section 3.1, a design choice was made to avoid dynamic memory allocation during development. This refers to dynamic allocation from the system heap which is separate to the heap used by the FreeRTOS kernel. As shown in Figure 3.2, FreeRTOS tasks are implemented as infinite loops and are not allowed to execute past the end of their implementation functions. Should a task reach the end of its implementation function, a "HardFault" exception will occur. To prevent this, the *vTaskDelete()* function can be placed at the end of each task implementation function to remove the broken task. This allows for the implementation of error handling routines such as possibly reinitialising the task. In order to make this function available, heap2.c is currently used as a memory management scheme. As deleting a task in this way is expected to be a very rare occurrence, the determinism of the flight software should not be affected. As a side note, using any of the memory management schemes other than heap3.c results in FreeRTOS managing a heap separate from the main program heap. All the data structures used by FreeRTOS are then allocated from the FreeRTOS heap.

With the kernel added to the project and configuration options set, FreeRTOS functionality can now be included in the project. The structure of a typical main loop in a FreeRTOS application is shown in Figure 3.1. Once the scheduler is started, the tasks that have been added to the scheduler are executed according to the scheduling algorithm. Tasks are implemented as functions that are never allowed to return. Tasks can therefore run continuously, periodically, or be synchronised with an event as described in Section 2.2.1.3. The typical structure of a task is shown in Figure 3.2. Each task is assigned its own stack by the kernel when the task is created. The optimal size of the stack required by a task can only realistically be determined after the task has been fully implemented. This process will be more difficult if the task uses dynamic memory allocation.

3.3 Flight Software overview

For the initial design of the structure of the flight software, a top-down approach was used to break the overall system down into smaller components. As stated in Section 1.2, almost all nano-satellites will carry an instance of the following components:

- Main OBC
- ADCS
- Communications hardware
- Electrical power system
- Payload
- Housekeeping

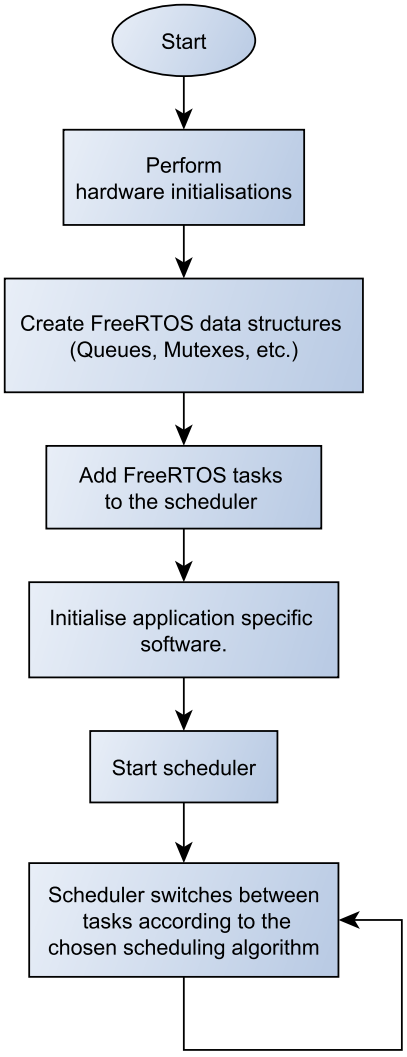


Figure 3.1 – Typical FreeRTOS application main function

- On-board data storage

It was therefore decided to develop software modules for each of these subsystems that could easily be adapted according to a mission's specific needs. Figure 3.3 shows a broad overview of the structure of the flight software. Except for the On-board Data Storage module, the structure for each of the subsystem modules is the same. Each module has a FreeRTOS task that acts as a manager for the module and is aware of each hardware and software component in the subsystem. The manager task reads commands and requests from a FreeRTOS message queue specific to that manager task. Depending on the command, these commands are then either directed to the interfaces to the specific hardware component or handled by mission specific procedures in the module itself. This manner of interaction between system components conforms to the choice to develop modular software as well as re-usable

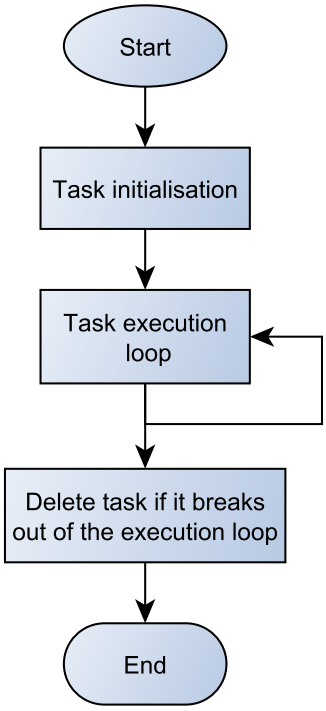


Figure 3.2 – Typical FreeRTOS task structure

framework structure presented in Section 1.4.

3.4 Flight Software Standards

Although one of the project goals was to develop the software as generic and reusable, it was necessary to choose a specific telemetry and telecommand interface between the flight software and the ground station control software. As the first mission in which it was planned to use the flight software was the QB50 mission, it was chosen to develop the interface to conform to the ground software that would potentially be used for QB50. For Stellenbosch University, one of the candidates for ground control software was the Satellite Control Software (SCS) program developed by the Swiss Space Centre. During the development phase of the flight software, SCS was the most developed and popular ground control suite available for teams contributing to the QB50 mission and it was therefore chosen to develop the flight software to conform to the SCS interface.

The interface used in SCS is based on standards set out by the Consultative Committee for Space Data Systems (CCSDS). Since its inception in 1982, the CCSDS has been developing standards and recommendations concerning the storage and transmission of data in space systems. This standardisation promotes collaboration and cost sharing between space agencies as well as simplifying the reuse and improvement of space data systems. The relevance and popularity of the standards are justified by [42], a list of 718 past and future missions that implement them.

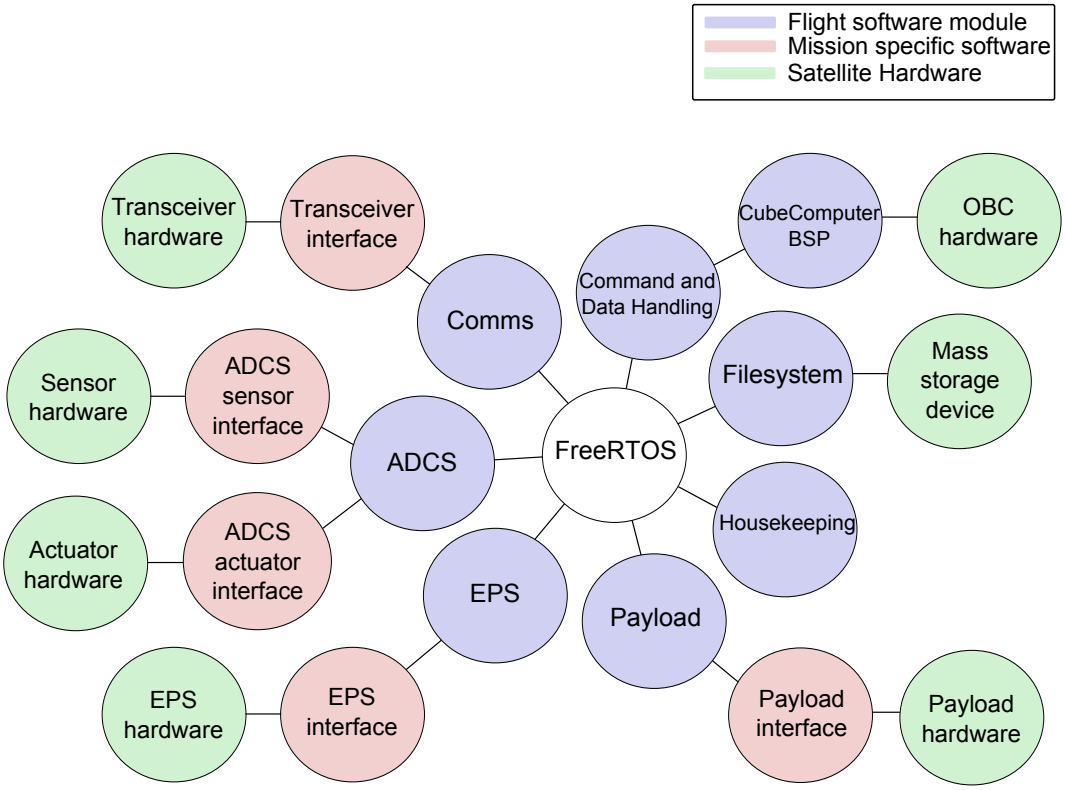


Figure 3.3 – Overview of Flight Software Module Structure

Developing the software to conform to these standards should therefore increase the re-usability of the software as existing ground software will be able to interface with the satellite and engineers will be familiar with the packet formats.

3.4.1 CCSDS Packet Standard

In order to interface with a ground station, the flight software would need to be able to decode and interpret the telecommand packets sent to it as well as format telemetry packets correctly for downlink to the ground station. The telecommand standard is specified in [43] and is shown in Figure 3.4.

1. The **Packet ID** fields identify the packet
    - The **Version Number** field indicates what variation (if any) of the telecommand packet structure is being used.
    - The **Type** field indicates whether the packet is a telecommand or telemetry source packet.
    - The **Data Field Header Flag** indicates whether a Telecommand or Telemetry Data Field Header field is included in the packet.
    - The **Application Process ID** holds a unique value which corresponds to an on-board process or subsystem. This is the final destination for

Packet Header (48 bits)							Packet Data Field (Variable)		
Packet ID				Packet Sequence Control		Packet Length	Telecommand Data Field Header	Application Data	Packet Error Control
Version Number	Type	Data Field Header Flag	APID	Sequence Flags	Sequence Count				
3	1	1	11	2	14				
16				16		16	24	Variable	16

Figure 3.4 – CCSDS Telecommand Structure [6]

this packet. The values that can be assigned to this field are therefore mission specific.

2. The **Packet Sequence Control** field identifies a packet in a series of telecommands.
   - The **Sequence Flags** field indicates where a packet belongs in a series of telecommands. A packet can also hold a "stand-alone" telecommand.
   - The **Sequence Count** field identifies a specific telecommand, enabling it to be traced in the telecommand system. A separate sequence count is maintained for each APID in the satellite system.
3. The **Packet Length** field indicates the length of the Packet Data field in octets (8 bit groups).
4. The Telecommand or Telemetry **Data Field Header** field contains any information necessary for the execution of the telecommand other than the application data.
5. The **Application Data** field contains data used during the execution of the destination process.
6. The **Packet Error Control** (PEC) field is used to transmit an error detection code with the packet in order to verify its integrity upon arriving at its destination.

Telemetry packets are used to transmit data from the satellite to the ground station. The CCSDS standard for a telemetry packet is defined in [44] and is shown in Figure 3.5.

For a telemetry packet, all the fields in the packet header are the same as for telecommand packets with the exception of the Grouping Flags field replacing the Sequence Flags field. The Grouping Flags field simply indicates where a telemetry packet belongs in a group of telemetry packets. In the telemetry packet data field, the Data Field Header and Packet Error Control fields perform the same function as in the telecommand packet definition. The Source Data field replaces the Application Data field and is used to hold the telemetry data the packet is carrying.

Packet Header (48 Bits)							Packet Data Field (Variable)		
Packet ID				Packet Sequence Control		Packet Length	Telemetry Data Field Header	Source Data	Packet Error Control
Version Number	Type	Data Field Header Flag	APID	Grouping Flags	Source Sequence Count				
3	1	1	11	2	14				
16				16		16	64	Variable	16

Figure 3.5 – CCSDS Telemetry Structure [6]

CCSDS Secondary Header Flag	TC Packet PUS Version Number	Ack	Service Type	Service Subtype
Boolean (1 bit)	Enumerated (3 bits)	Enumerated (4 bits)	Enumerated (8 bits)	Enumerated (8 bits)

Figure 3.6 – PUS Telecommand Data Field Header [7]

3.4.2 ECSS Packet Utilisation Standard

To complement the telemetry and telecommand packet standards developed by the CCSDS, the European Cooperation for Space Standardisation (ECSS) developed the Packet Utilisation Standard (PUS). The PUS defines an application level interface for interactions between the satellite flight software and the ground support software. It does this by defining a structure for the Data Field Header and Application or Source Data fields in telecommand and telemetry packets respectively. The structure of the Data Field Header field in a telecommand packet is given in Figure 3.6. Optional fields exist in the packet definition that are not included in the flight software. This is to keep the overhead of each telecommand packet small; an important factor considering the tight link budget of most CubeSat missions.

The PUS includes many optional services, packet fields and recommendations. These are included in the standard to make it applicable to as wide a range of missions as possible. The standard was not written with the expectation that it be applied in full to every mission. It therefore needs to be tailored for every mission, including more or less of the standard depending on what resources are available or what is required of the satellite. This is especially true for a CubeSat mission where a lot of the detail included for larger, more complex satellites is not required or cannot be supported.

The fields in the Data Field Header hold the following information:

- **CCSDS Secondary Header Flag:** This bit is set to zero to indicate a “non-CCSDS defined secondary header".

Spare	TM Source Packet PUS Version Number	Spare	Service Type	Service Subtype	Time
Fixed BitString (1 bit)	Enumerated (3 bits)	Fixed BitString (4 bits)	Enumerated (8 bits)	Enumerated (8 bits)	Absolute Time (40 bits)

Figure 3.7 – PUS Telemetry Data Field Header [7]

- **TC Packet PUS Version Number:** An indication of the PUS version number that the packet belongs to.
- **Ack:** This field holds an indication of what acknowledgements should be sent to the ground by the Telecommand Verification service (see Section 4.2) during the execution of this telecommand. Which stages of execution are verified depends on which bits in the field are set.

  - - - 1: Acknowledge acceptance
  - - 1 -: Acknowledge execution start
  - 1 - -: Acknowledge execution progress
  1 - - -: Acknowledge execution completion

- **Service Type:** Indicates which service type the packet belongs to.
- **Service Subtype:** Indicates which service subtype the packet belongs to.

The structure of the Data Field Header field for a telemetry packet is shown in Figure 3.7. As before, optional fields exist but are not implemented in SCS and are therefore not relevant.

The spare bits are included in order to maintain symmetry between the separate types of header. In order to include a time stamp with the telemetry packet, 5 bytes are reserved at the end of the header. These bytes represent time using the CCSDS Unsegmented Code (CUC) time format defined in [45]. In this case, 4 bytes are used to represent coarse time (counting seconds) with 1 byte counting fine time (counting subseconds). This allows for storage of time stamps up to roughly 136 years from the chosen epoch with a precision of roughly 4 milliseconds. Equation 3.4.1 shows how the bytes in the Time field are used to calculate a time value.

$$\text{Time} = \text{Epoch} + C1 * 2^{24} + C2 * 2^{16} + C3 * 2^{8} + C4 * 2^{0} + F1 * 2^{-8} \qquad (3.4.1)$$

In Equation 3.4.1, the epoch time is a reference from which time is measured and can be chosen according to the requirements of a specific mission. The variables $C1$, $C2$, $C3$ and $C4$ represent the values stored in the 4 coarse time bytes while $F1$ is the fine time byte.

The last fields that still need to be defined in order to completely specify the structure of the telemetry and telecommand packets are the Application Data and Source Data fields. The structures of these fields are defined by the PUS services.

3.4.3 PUS Services

Within the PUS, a set of standard PUS services are defined. These services act as the interface between the satellite and the ground segment and are designed to be independent of the satellite's mission and on-board architecture. Each service type relates to a specific function on the satellite. Within each service type, there is also a service subtype for each function that the service has to perform. Each service subtype defines the structure of the Source or Packet Data field for telemetry or telecommand packets belonging to that subtype. The minimum capability set of a service defines the minimum number of service subtypes that need to be included in the service implementation. A complete list of the 19 standard services defined by the ECSS can be found on page 50 of [7].

Implementing all of the standard PUS services for a CubeSat would be extremely time consuming not to mention superfluous. A subset of the standard services was therefore selected according to the basic functionality generally required on a CubeSat mission. The services and sub-services recommended for implementation in [9] were used as a starting point with additional sub-services being implemented as the need for them became apparent. Each of the services that were implemented are briefly discussed below. The details of their sub-services and implementation is discussed in Chapters 4 and 5.

Service 1: Telecommand Verification Service The minimum capability set of this service provides a means for reporting the success or failure of the delivery and execution of a telecommand. On the detection of one of these conditions, a telemetry packet is generated and transmitted to the ground station by the service. This packet uniquely identifies a telecommand and indicates the result of its processing. More detail is provided on the implementation of this service in Section 4.2.

Service 3: Housekeeping and Diagnostic Data Reporting Service This service handles the collection and transmission of housekeeping and diagnostic data reporting. Each predefined set of housekeeping parameters is given a Structure Identification (SID) field. The SID is used to identify which set of housekeeping parameters a telemetry packet contains when it is received by the ground station. More detail is provided on the implementation of this service in Section 5.1.

Service 8: Function Management Service This service provides control of subsystems or application processes to the ground station. In order to do this, each function within an application process or subsystem is assigned a Function ID. The combination of the APID contained in a telecommand packet's Packet ID, and the Function ID contained in the same packet's Application Data field, enable the service to direct the telecommand to its destination and execute the correct function. Also contained in the Application Data field will be any parameters required by the function. More detail is provided on the implementation of this service in Section 4.3.

Service 11: On-board Operations Scheduling Service Service 11 handles the scheduling of operations for future execution on-board the satellite. Telecommand packets are stored in an on-board schedule and sent to the relevant destination when the on-board time reaches the time stamp attached to the received telecommand. The service contains functions to control the operation of the schedule and as well as

report and manipulate its contents. The implementation and sub-services of service 11 is discussed in detail in section 4.4.

Service 13: Large Data Transfer Service Service 13 handles the transfer of large data units between the ground station and the satellite. In the context of this service, a large data unit is defined as any set of data that does not fit into a standard telecommand packet's Application Data field. For a typical mission, large data units that are uploaded could include new versions of the flight software and command scripts. Downloads could include command schedules, captured images and other payload data. More detail is provided on the implementation of this service in Section 4.7.

Service 15: On-board Storage and Retrieval Service Service 15 provides storage of telemetry packets that are generated for transmission when no ground station is in range for download. The contents of the storage managed by this service can then be examined and downloaded when the satellite passes over the ground station. The implementation of this Service is detailed in Section 5.2.

Service 131: File System Interface Service Service 131 is a custom service included to provide direct control over CubeComputer's microSD card and the file system running on it. Functions such as examining the contents of the file system, downloading files and resetting the file system are included in this service. The implementation of this service is detailed in Section 4.7.

The decision to implement the PUS and its services was an important one as it reduced the number of decisions that would have to be made concerning data formats and protocols. The extensive existing documentation for the PUS makes it easier for users of the software to learn its structure and adapt it for future missions. Achieving this easy re-usability is one the main project goals.

3.5 Conclusion

This chapter covered the design phase of the flight software development process. A number of design philosophies, such as the use of modular programming, were adopted to guide the development of the software. FreeRTOS was then evaluated to determine its suitability as a foundation for satellite flight software. Having been found to be adequate, the installation and initial configuration of FreeRTOS was then discussed.

With a foundation established, the flight software was broken down into components using a top-down approach and the research conducted in Chapter 2. Each component handles a specific function on-board the satellite and contains a manager task that acts as its interface to the rest of the satellite. The interfaces to the satellite subsystems are contained in their corresponding modules.

In order to establish a well defined interface between the satellite and ground services, the CCSDS packet standard was adopted. The PUS was included in the project along with its accompanying service definitions in order to make use of the well-defined functionality included in the services. The following chapters describe the implementation of the structure developed in this chapter.

Chapter 4

Command and Data Handling

The main OBC in a satellite acts as a central hub for communications between subsystems. Command and data handling activities therefore comprise the majority of the flight software's responsibilities.

The command handling specific components of the flight software are used whenever commands are retrieved from the transceiver by the OBC . A telecommand packet will either be destined for one of the services mentioned in Section 3.4.3 or one of the satellites subsystems. The command handling components ensure that telecommand packets reach their destinations in a timely manner. Command handling in the flight software comprises of the functionality provided by PUS services 1 (telecommand verification), 8 (function management) and 11 (on-board operations scheduling). Various types of data also need to be communicated between different processes on the satellite as well as downlinked to the ground. Data handling components include PUS service 13 (large data transfer), the interface to the file system and the interface to the I2C bus. In order for the functionality provided by the services to be used by the various subsystems, generic subsystem managers were implemented for each major subsystem.

This chapter presents the above-mentioned components starting with the management of the I2C bus. Once the communication between different hardware subsystems has been established, the different services involved in command and data handling are presented. Finally, the structure of the subsystem managers and the way they interact with the services and the I2C bus is discussed.

4.1 The I2C interface

In a CubeSat using the I2C bus, the main OBC is usually the only bus master and manages the other on-board subsystems. In a system where the bus master is running multiple tasks, it is important to ensure that only a single task is allowed access to the bus at a time. Attempting to send a new message over the bus before the previous transfer is complete could result in corrupted data being sent to or returned from one of the subsystems. If this corruption is not detected, the satellite system could malfunction and enter an unknown state. In order to ensure efficient

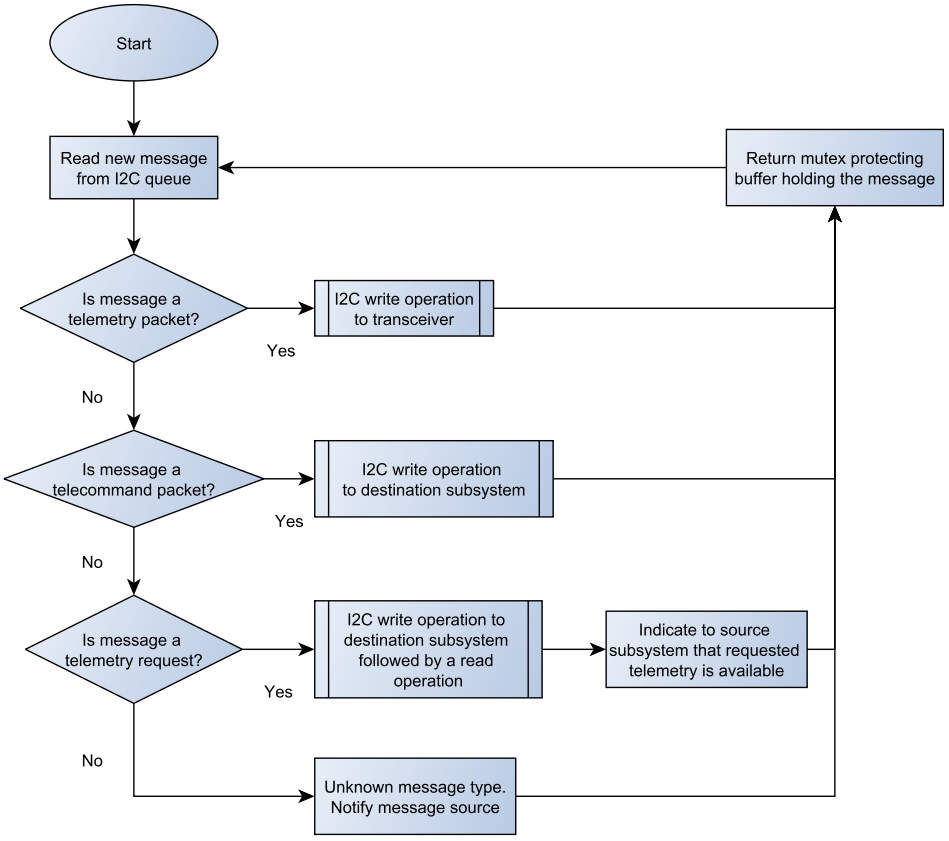


Figure 4.1 – Structure of the I2C manager task

communication on the satellite, various issues needed to be addressed when designing the I2C interface between the flight software application and the subsystems.

There are two main options for managing the interaction of the application with the bus. A FreeRTOS task dedicated to the management of the bus or a mutually exclusive global function. Initially, a manager task was used with success. However, it was later replaced by a global function after the complexities and disadvantages of using a manager task became apparent. The following section describes the implementation of the manager task as well as its shortcomings. Following that, the interface developed to replace the manager task is discussed.

4.1.1 The I2C manager

Before it was decided to use a global, mutually exclusive function to access the I2C bus, an I2C manager task was used. Any message that needed to be sent to a subsystem would have to be sent to an I2C message queue. The I2C manager would read messages from the queue and deliver them to their destinations. The basic structure of the manager is shown in Figure 4.1.

In this way, application tasks wouldn't have to worry about activity on the bus because only the manager task would have access to it. Application tasks would

also be able to continue after sending their message to the manager without having to wait for possible delays in communication such as transfers already in progress. However, although this approach worked for a time, a number of problems became apparent as development progressed.

In order to conserve memory, messages to the I2C queue contained references to data that needed to be sent over I2C rather than the data itself. As application tasks could not be sure how long it would take for the manager to send the message data, the buffers containing the message data would have to be protected by mutexes. This was to ensure that the messages in the buffers wouldn't be overwritten before they had been sent. Every message buffer in the flight software would therefore need to be protected by a mutex.

Although sending I2C requests to a manager task allowed application tasks to avoid having to wait for existing or higher priority transfers, it was found that few tasks could continue after sending a message as they first required a response to their message. This therefore invalidated the main advantage of application tasks not handling their own I2C communications. Returning responses to application tasks that had sent requests to subsystems was also a complex issue. The application tasks had to poll a flag that would be set by the I2C manager to indicate when a response had been received.

Another problem with this approach was that it would be difficult to prioritise messages. Simply adding messages to the I2C message queue meant that there was a chance that high priority messages would have to wait for low priority messages to be processed. Various methods could be used to add priorities to messages but this would add more complexity to a system that was already overly complex.

By examining the complexities and shortcomings presented above, it was clear that this was not the most efficient manner in which to manage bus interactions. It was therefore decided to use the alternative approach of using a mutually exclusive global function to access the I2C bus.

4.1.2 The I2C interface

The structure of the function used to access the I2C bus in the flight software is shown in Figure 4.2. This approach solves the issues described in Section 4.1.1. As the I2C transfer now takes place within the application task, no protection of shared memory or notification of transfer completion is required. The problem of prioritising messages is also simpler as I2C messages will inherit the priority of the task sending them. An example of this is shown in Figure 4.3.

In the figure, two tasks of different priorities try to send data over the I2C bus while a separate, lower-priority task is already busy with it. The lower priority task is allowed to finish it's transmission as it has possession of the mutex. Once the mutex is made available, the highest priority task trying to obtain the mutex becomes active. In this way, higher priority messages will get processed sooner than lower priority ones.

This method is, however, not perfect. Figure 4.3 illustrates a problem known as priority inversion in which a higher priority task is waiting for a lower priority task to

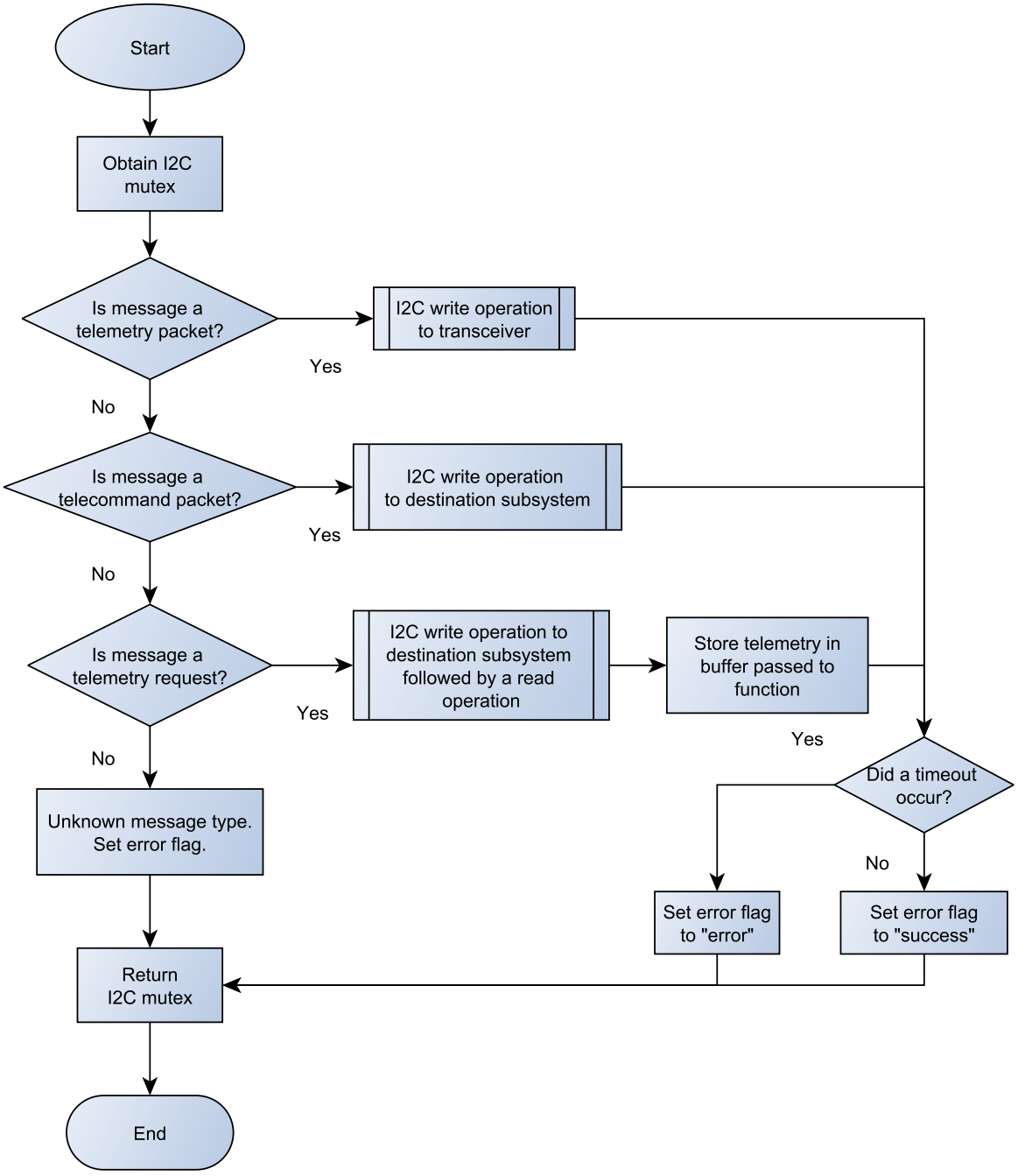


Figure 4.2 – Structure of the I2C access function

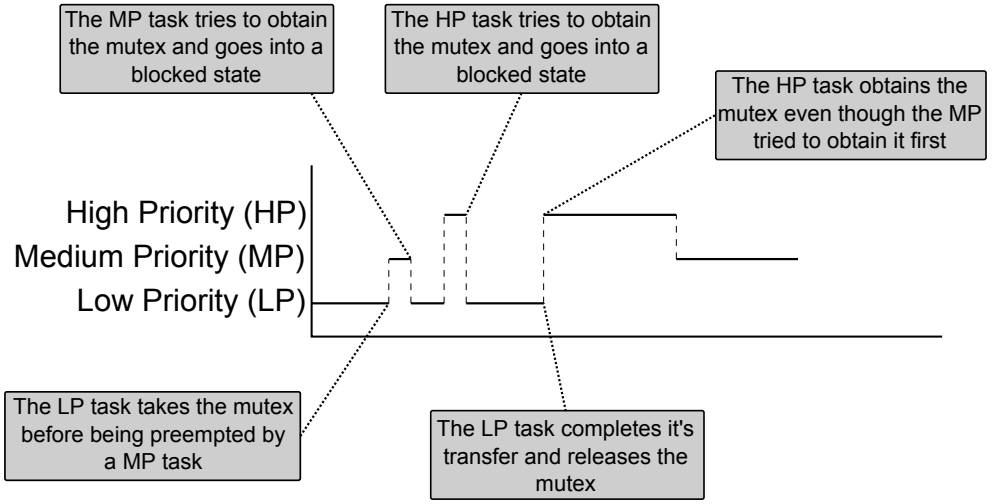


Figure 4.3 – Example of mutex transferral

finish executing. This situation can be further aggravated if a medium priority task that doesn't need the mutex starts executing. This would further delay the execution of the higher priority task as the medium priority will pre-empt the lower priority task. FreeRTOS mutexes include a priority inheritance mechanism to minimise the time for which priority inversion can occur. Priority inheritance temporarily raises the priority of a task holding a mutex to the priority of the highest priority task that is trying to obtain the mutex. In this way, high priority messages will be sent as soon as possible.

Depending on the result of the transfer, the interface function will return a code indicating if the transfer was successful and, if not, the reason for failure. Error codes include errors that occurred due to erroneous parameters being passed to the function or transfer timeouts.

4.2 Service 1 implementation

Service 1 provides the capability for the flight software to generate reports that verify the reception and execution of telecommands. The extent to which the different stages of execution are reported depends entirely on the information that the various application processes receive from the subsystems executing the commands. For most CubeSat systems, the data supplied in this way is minimal. It was therefore considered sufficient to implement the verification of only three stages of execution:

- **Acceptance:** This stage occurs prior to execution and a success report is sent if the telecommand's checksum is valid and the destination application process supports the telecommand.
- **Start:** Verification of this stage indicates that the parameters of the telecommand are valid and that execution can begin.

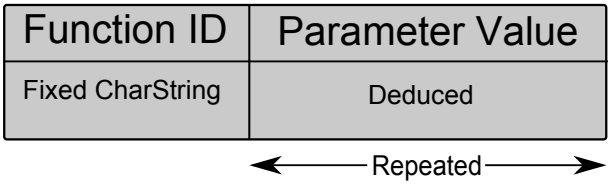


Figure 4.4 – Application Data field contents for a "perform function" telecommand

- **Completion:** Verification reports for this stage are generated according to the result of a telecommand's execution.

For each of these stages, a separate sub-service exists for reporting the success or failure of the stage. This means six sub-services are implemented in Service 1. Each of these sub-services consists of the generation and transmission of a telemetry packet containing information about the telecommand being executed. The Source Data field of a generated telemetry packet will contain copies of the Packet ID and Packet Sequence Control fields from the telecommand packet being verified. In the case of a failure report, an error code will also be included indicating the reason for the failure.

Not every telecommand needs to be verified in this way. The "Ack" field in a CCSDS telecommand packet indicates what acknowledgements are expected for a specific telecommand (see Section 3.4.2).

4.3 Service 8 implementation

Service 8 is the Function Management service. Telecommand packets containing commands for application processes not implemented as services are assigned to Service 8. These application processes can be any component that has functionality which may be controlled from the ground such as an imager or deployable element. The only service sub-type included in Service 8 is the "perform function" service. The Application Data field in a telecommand packet belonging to this service sub-type is shown in Figure 4.4.

The combination of the APID and Function ID fields in the packet can be used to uniquely identify the telecommand. If the telecommand needs to be sent to a subsystem, the destination application process will then format the parameters correctly for use by the subsystem.

4.4 Service 11 implementation

As briefly mentioned in section 3.4.3, the On-board Operations Scheduling service provides an on-board command schedule from which commands are released (executed) according to their accompanying time tags. The service also provides service users with functions to manipulate the schedule and the commands it holds. Depending on the extent to which the service is implemented, the times of various scheduling events are stored. An example of such an event is the time at which the schedule was enabled.

Of the 19 service subtypes included in this service, the following are implemented:

- Subtype 1: Enable the release of telecommands from the schedule.
- Subtype 2: Disable the release of telecommands from the schedule.
- Subtype 3: Reset the telecommand schedule .
- Subtype 4: Insert a telecommand into the schedule.
- Subtype 5: Delete telecommands within a specified range of sequence numbers.
- Subtype 6: Delete telecommands scheduled for execution over a specified time period.
- Subtype 7: Time-shift selected telecommands.
- Subtype 13: Report an existing command schedule summary.
- Subtype 15: Time-shift all telecommands.
- Subtype 17: Generate a summary of the command schedule contents.

From the list above, only a request to service subtype 13 will return a telemetry packet. All the other subtypes handle telecommands used to control the service. If desired, subtypes 13 and 17 could be implemented as a single subtype.

4.4.1 The Command Schedule

The structure of the command schedule itself is not defined in the PUS standard and a suitable structure therefore needed to be developed to hold telecommands awaiting execution. Two possible options for such a structure include a statically allocated fixed-size array and a dynamically allocated linked list. The fixed-size array is the safer option. The static memory allocation is less likely to result in an attempt to access a NULL pointer and the amount of heap memory used by the schedule is restricted. The most basic way of storing commands in this structure would be in chronological order with next telecommand to be executed stored in the first element of the array. However, the process of adding and removing telecommands from a schedule structured in this way would be very inefficient. In the worst case, each element of the array would have to be copied to the next array index in order to make space for an entry in the first index. The linked list would be more efficient in this regard. Adding and removing an element anywhere in a schedule constructed as a linked list would consist of finding the correct chronological position in the list, dynamically allocating memory for the new entry, and adding it to the list by assigning two pointers. The time taken to execute this process is far more predictable (and therefore, real-time) but the robustness of static memory allocation would be lost if this approach was followed. In order to utilise the advantages provided by the two approaches, a structure that incorporated elements of both was used.

The final command schedule structure is a statically allocated fixed size array of the custom data type TC_schedule_entry. This data type includes the following fields:

```
typedef struct tcScheduleEntry{
        uint8_t TC[MAX_TC_PKTSIZE];      // The schedule entry
        uint32_t exe_time;               // The time tag
        struct tcScheduleEntry* next_TC;// The next schedule entry
        bool free;                  // Flag indicating available storage
}TC_schedule_entry;
```

Using this structure, telecommand packets can be added to and removed from the static array without having to move the other entries around each time. This is due to each entry holding a reference to the next chronological entry in the schedule. A separate pointer to the first entry in the list is also kept to speed up schedule operations. When a telecommand is added to the schedule that has an earlier time tag than any other entry, the 'first entry' pointer is set to point to the memory address of the array element where the telecommand is stored. A software timer is then set to count down the time between the new entry's time tag and the current OBC time. When the timer expires, the schedule entry pointed to by the first entry pointer is removed from the list and executed. The schedule array element is then reinitialised to default values and marked as available to store a new telecommand packet.

Figure 4.5 gives an example of how the state of the schedule array changes as different telecommands move in and out of the schedule.

1. State A in Figure 4.5 shows a command schedule with space for five entries. Two telecommands (1 and 2) set to execute at times 1 and 3 respectively have been added to the array. The 'first entry' pointer is pointing to the next telecommand that needs to be executed.

2. In state B, telecommand 3, with execution time 2, has been added to the schedule in the first available memory slot. As its execution time falls between that of telecommands 1 and 2, it is placed between them in the 'linked list'.

3. In state C, telecommand 1 has been released from the queue due to some event such as deletion or execution. The remaining telecommands have retained their positions in memory but are now at the front of the linked list. The 'first entry' pointer now points to telecommand 3.

4. In state D, a new telecommand has been added to the schedule. Although its execution time is later than both of the telecommands added earlier, it is still placed in the first available memory slot at the beginning.

This method was considered the implementation that would provide the best balance between robustness and speed.

4.4.2 Scheduling service subtypes

With the structure of the command schedule defined, the implementation of the service subtypes described in 3.4.3 was relatively simple.

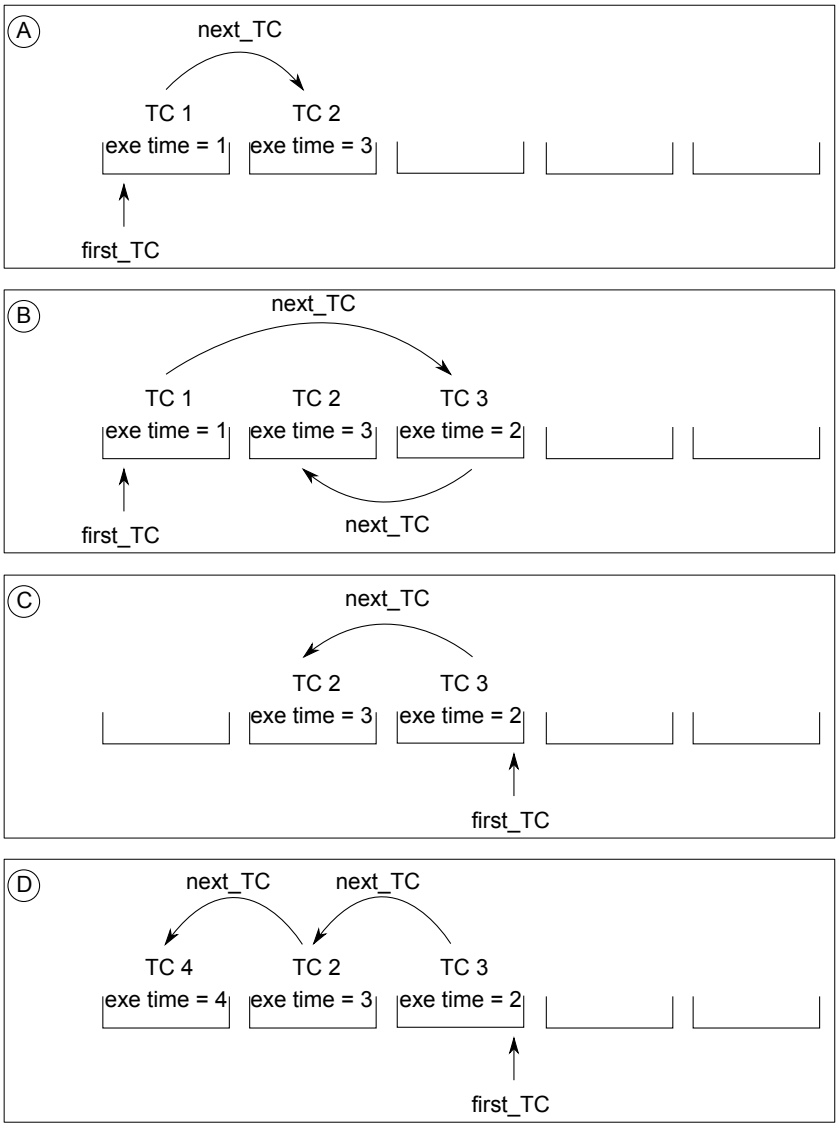


Figure 4.5 – Example of command schedule operation

- **Subtype 1 and 2: Enabling and disabling the release of telecommands.** A flag is kept indicating whether the release of commands from the schedule is enabled or disabled. When a command's execution time is reached, the flag is evaluated. If release is enabled, the telecommand packet is removed from the schedule and sent to service 8 for decoding and execution. If release is disabled, the command is removed from the schedule and an execution start failure report is sent via Service 1.

- **Subtype 3: Resetting the command schedule.** As defined in [7], a reset of the schedule entails clearing and disabling the schedule, and resetting all the scheduling event information. All the elements in the command schedule structure are set to default values and marked as available to store new commands.

- **Subtype 4: Inserting telecommands in the schedule.** The method of how commands are added to the schedule has already been discussed in Section 4.4.1. However, a number of errors can occur that may prevent the telecommand being added to the schedule. In this implementation of the service, these errors include the command schedule being full, the destination APID of the telecommand being invalid, and the time tag included with the telecommand referring to the past.

- **Subtype 5: Deleting telecommands from the schedule.** A command to delete telecommands from the schedule will contain 'APID', 'Sequence Count', and 'Number of Telecommands' fields as application data. The schedule will first be searched for the entry uniquely identified by the combination of the APID and sequence count. This entry will be deleted along with a number of successive entries belonging to the same APID. The number of successive entries that will be deleted is indicated by the 'Number of Telecommands' field

- **Subtype 6: Deleting telecommands over a time period.** This service subtype deletes telecommands from the schedule according to their execution time rather than their destination. A command of this type includes a 'Range' field that indicates the format of the time period. Depending on the range field, up to two time tags will also be included in the command. The time period is then set up as follows:

  - Range = 0. Time Period = "All". Commands are deleted from the beginning to the end of the schedule.
  - Range = 1. Time Period = "Between". Commands with execution time between time tag 1 and time tag 2 (inclusive) are deleted.
  - Range = 2. Time Period = "Before". Commands with execution time less than or equal to time tag 1 are deleted.
  - Range = 3. Time Period = "After". Commands with execution time greater than or equal to time tag 1 are deleted.

- **Subtype 7: Time-shift selected telecommands.** A command belonging to this subtype includes 'Time Offset', 'APID', 'Sequence Count', and 'Number

of Telecommands' fields as application data. The selection of telecommands to time-shift uses the latter 3 fields of the application data and is the same as the process used in Subtype 5. The 'Time Offset' field holds a positive or negative value by which to shift the execution times of the scheduled commands. A positive value shifts the execution time to a later stage while a negative value shifts it to an earlier stage. A command will not be shifted if the shift will result in it having a time earlier than the current OBC time.

- **Subtype 13: Summary schedule report.** A telecommand packet belonging to this service subtype is a request for a downlink of a summary schedule report of every telecommand currently in the command schedule. The schedule report itself is generated by service subtype 17.
- **Subtype 15: Time shift all telecommands.** This service is similar to subtype 7 except that it automatically applies to every command in the schedule. The application data for a telecommand packet belonging to this subtype therefore only consists of a positive or negative time offset which will be added to the release time of every command in the schedule.
- **Subtype 17: Report command schedule as summary.** This service constructs a report schedule of every telecommand in the schedule in summary form. The first element in the source data of a summary schedule report indicates the number of telecommands contained in the report. After that, the release time, APID, and sequence count of each telecommand are included.

4.5 Service 13 Implementation

The PUS defines Service 13 as the Large Data Transfer service. This service provides a protocol for the uplink and downlink of large sets of data referred to as Service Data Units (SDU). As mentioned in Section 3.4.3, a SDU is a set of data that does not fit in the application data field of a single telecommand packet. This service could therefore be required and utilised by the flight software in a number of scenarios. For example:

- The upload of a binary file containing a new version of the flight software.
- The downlink of image files.
- The downlink of large log files for processing on the ground.
- The transfer of mission specific files. For example, the scripts for the QB50 payloads.

A full list of the sub-services implemented in Service 13 is given below. Sub-services 1 to 8 apply to a downlink operation while 9 to 16 apply to uplink.

1. Downlink the first part of a SDU.
2. Downlink an intermediate part of a SDU.

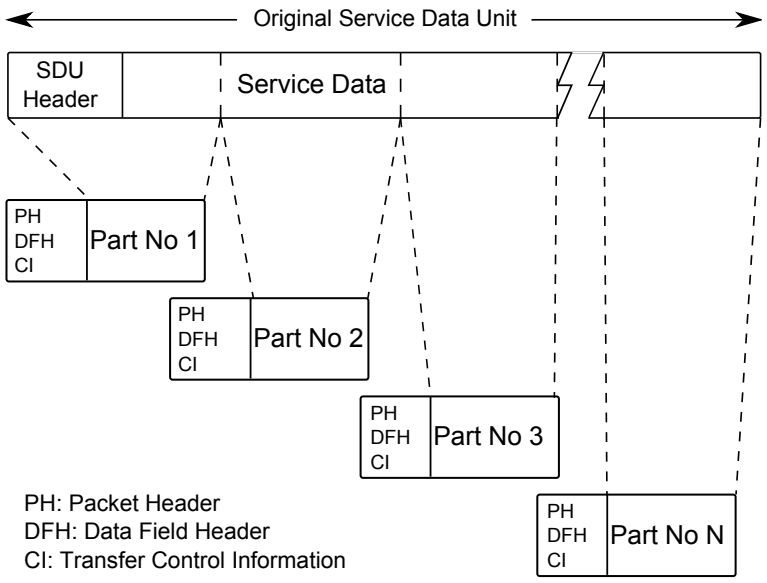


Figure 4.6 – The splitting of a service data unit into parts. Adapted from [7]

3. Downlink the last part of a SDU.
4. Downlink a report of a transfer abort initiated by the sending end.
5. Receive downlink reception acknowledgement.
6. Receive a report of unsuccessfully received parts.
7. Re-transfer part of a SDU.
8. Receive a transfer abort notification from the receiving end.
9. Accept the first part of a SDU.
10. Accept an intermediate part of a SDU.
11. Accept the last part of a SDU.
12. Accept a re-transferred part of a SDU.
13. Transfer abort initiated by the sending end.
14. Uplink reception acknowledgement report.
15. Downlink a report of unsuccessfully received parts.
16. Send a notification of a transfer abort initiated by the receiving end.

Figure 4.6 shows how a SDU is split into fixed size parts for transmission. The format of a standard SDU is shown in Figure 4.7. The fields of the SDU hold the following information:

Unit Type	Packet Header	Data Field Header	Packet Data
1 byte	6 bytes	Deduced	Any

Figure 4.7 – The format of a Service Data Unit. Adapted from [7]

Service Data Unit ID	Sequence Number	Service Data Unit Part
1 byte	2 bytes	Fixed Octet String

Figure 4.8 – The packet data format for a telecommand containing a Service Data Unit part. Adapted from [7]

- **Unit Type:** Indicates whether the packet that follows is a standard or extended packet.
- **Packet Header:** Standard CCSDS packet header as defined for telemetry and telecommand packets.
- **Data Field Header:** Standard data field header as defined for telemetry and telecommand packets.
- **Packet Data:** The large set of data being uploaded or downloaded.

The format of the packet data field is shown in Figure 4.8. The structure of each large set of data is defined by a SDU ID that is used to uniquely identify the SDU. This ID can be used to identify which SDU a packet belongs to when multiple large data transfers are being sent from or received by the same process.

To achieve the functionality listed above, the service is split into two operational segments for downlink and uplink data transfers. The protocols involved with downlink and uplink are identical with the distinction being that the uplink implementation is concerned with receiving, acknowledging, and assembling the data while the downlink implementation disassembles, packages, and transmits it. This section presents the implementation of the following functionality.

4.5.1 Large Data Upload

Initially, the service will receive the first part of a new SDU through service subtype 9. If the packet is error-free, a “transfer information” structure containing information about the current transfer is initialised. This structure holds information on the last successfully received packet in the transfer and whether missing packets have been detected. After the structure is updated with the Service Data Unit ID, a reception timer is started. If the timer reaches a specified timeout interval before the next SDU part is received, it will be assumed the transfer has failed and the

entire operation will be aborted. Sub-service 16 will then be used to notify the ground station of the abort. The same process of updating the context of the transfer information structure and resetting the reception timer applies to the reception of intermediate parts.

If, at any point during the transfer, an erroneous packet is received by the service or a discontinuity in packet sequence counts is detected, the sequence counts of the missing packets will be recorded and the transfer information structure will be updated. When the final part of a SDU is received, the service will either send an acknowledgement that the entire SDU was successfully received or a request will be sent for the re-transfer of the missing packets. The service will then send an acknowledge when all the remaining parts have been successfully received. If the last part of the SDU is not received, the transfer will be aborted. A flow diagram of the uplink process is shown in Figure 4.9

4.5.2 Large Data Download

The downlink procedure is almost identical to the uplink procedure. After receiving an indication from an on-board process that a large data unit is available for download, the service will split the data into parts of a predefined size. Except for the last part, this size is usually the maximum size of the source data field in a telecommand packet. Each part is then transmitted until the last part of the SDU has been sent. Upon sending the last part, an acknowledgement timer is started. If neither an acknowledgement, request for re-transfer or abort notification are received before the timer times out, the service will assume the transfer has failed. The download operation will be aborted and notifications will be sent to the data source as well as the receiving end of the transfer. If a request for re-transfer is received, the requested parts shall be collected from the data source and downlinked. This process will be repeated until the successful reception of the entire SDU has been acknowledged by the receiving end. A flow diagram of the downlink procedure is shown in Figure 4.10

4.5.3 Re-transferring missing packets

A number of components are required order to keep track of missed packets during an upload operation. Firstly, an array of 16 bit unsigned integers is initialised for storing the sequence numbers of the missed packets. The size of the memory reserved is determined by the largest number of parts expected to be transferred at once by the service. This is to ensure that no matter how many parts are unsuccessfully transmitted, the transfer will not have to be restarted. At the time of writing, the largest data unit expected to be uploaded is a binary file containing a new version of the flight software. A binary file of 150kB will require 633 parts to transfer and an array of 633 16-bit unsigned integers is therefore currently reserved in memory. Secondly, two counters are kept: one for recording the amount of packets that have been missed and one for the number of repeated packets that have been received.

As an upload operation progresses, the sequence numbers of any missing packets are recorded and the missing packets counter is incremented. When the final packet is received, the stored sequence numbers are communicated to the ground station

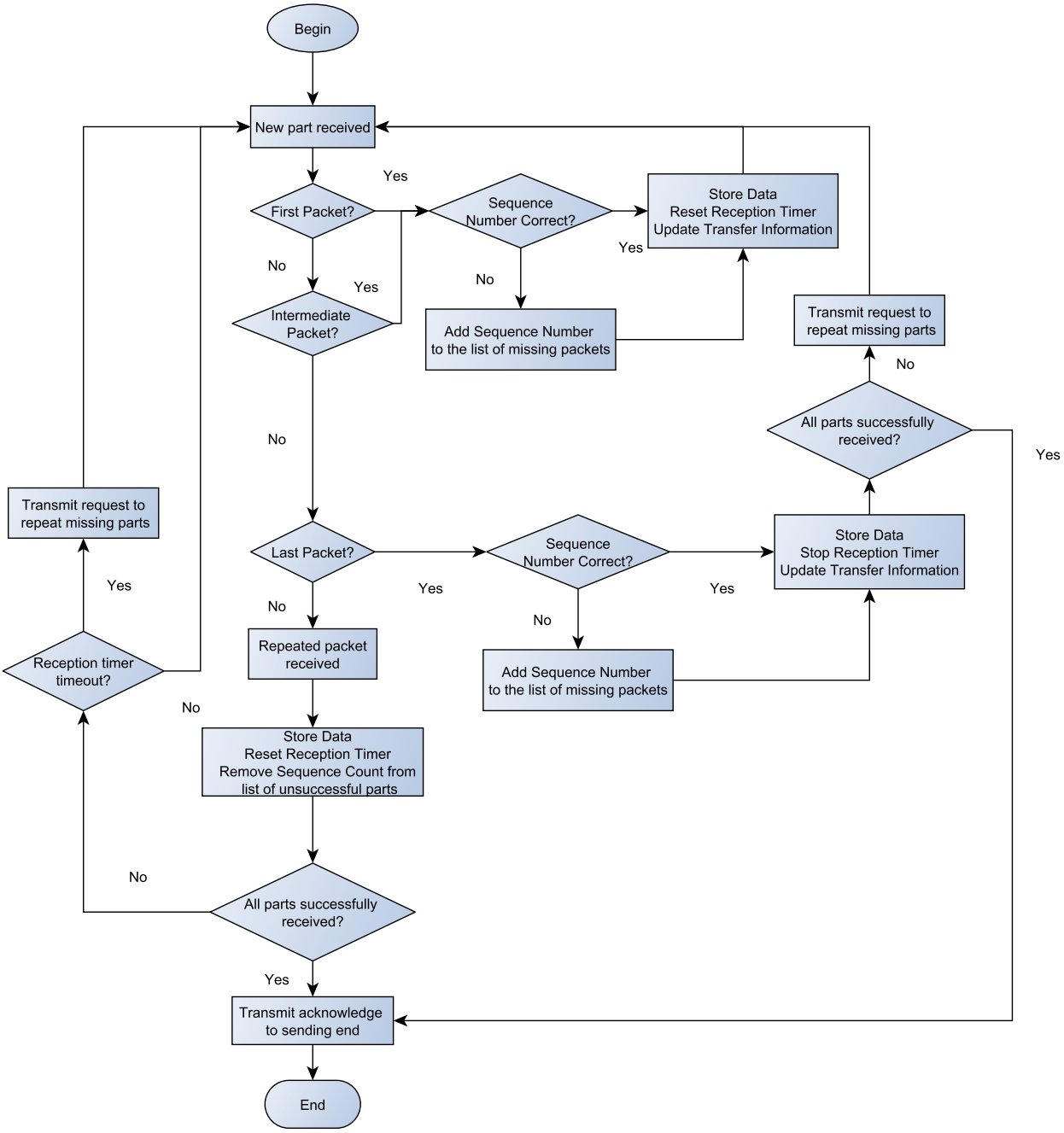


Figure 4.9 – Service 13 SDU upload protocol

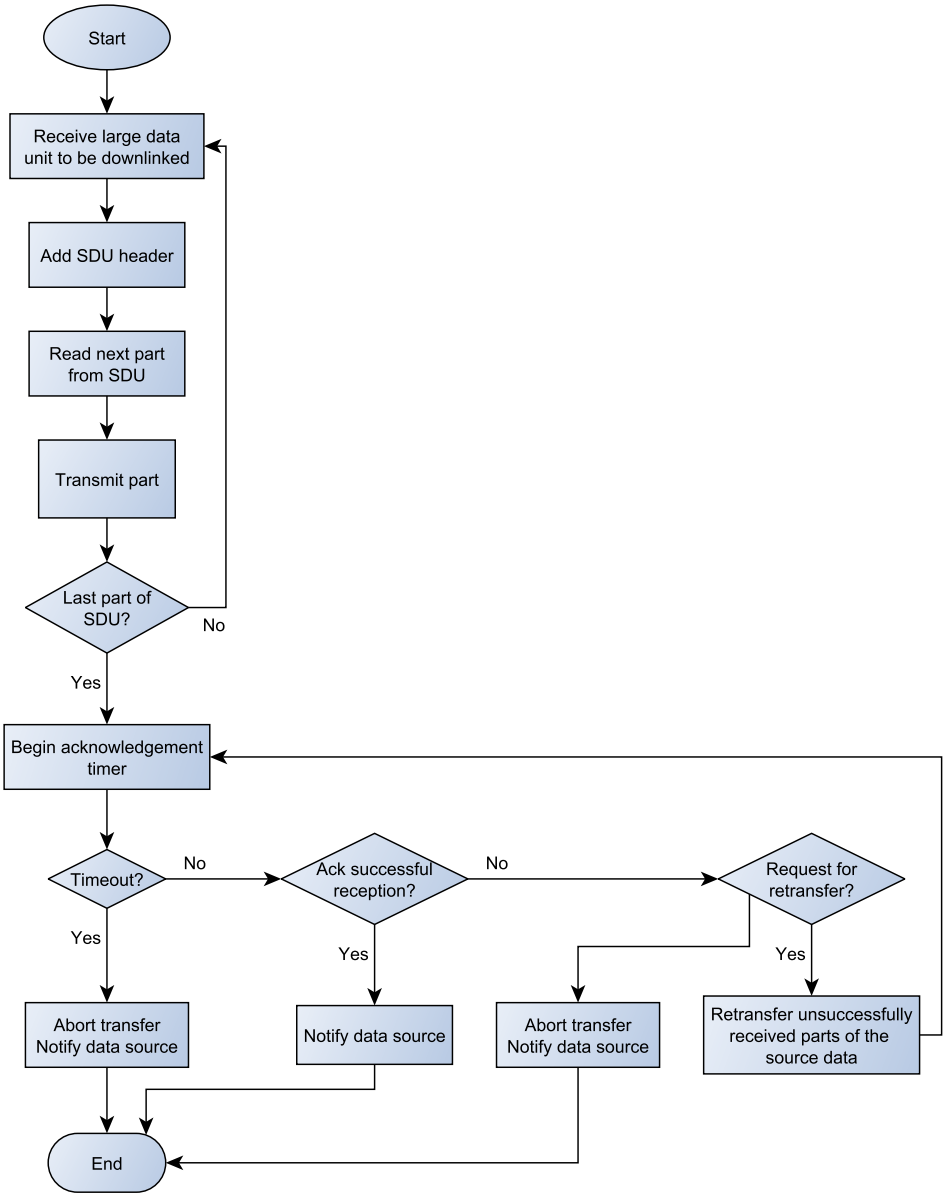


Figure 4.10 – Service 13 SDU download protocol

Figure 4.11 – Example of an image file before (left) and after (right) dropped packets are retransmitted.

which will start repeating the missing packets. As the satellite receives the repeated packets, the array holding the sequence numbers of the missing packets is updated. If the reception timer times out during the transfer of repeated packets, it is assumed that not all the packets were successfully transmitted and the process is repeated using the updated array. This process continues until an abort command is received from the ground station or the entire SDU is received.

Figure 4.11 shows an image file that was explicitly transferred with packets missing. The black areas on the left image are portions of data that were not received and therefore remain empty. After the first round of transmission, the sending end was automatically notified of the packets that were not received successfully. The missing data was extracted from the source a second time and retransmitted. The receiving end then received the re-transmitted packets and used the sequence numbers to place the new data in the correct location in the file.

4.5.4 Notes on the Transfer Protocol

During the development of this service as well as the file transfer service for Service 131 (see Section 4.7), a number of compromises were made to adapt the PUS defined service for the flight software. This section discusses which methods and modifications were used and the rationale behind them.

An optional functionality of the service that is not included in the flight software is the use of a sliding window. If a sliding window is used, a window size is defined. The window size defines the amount by which the sequence count of the SDU part currently being transmitted can exceed the sequence count of the last SDU part that was acknowledged. The sliding window is therefore defined by the sequence count of the last acknowledged part and the window size and only parts with sequence counts inside the sliding window may be sent. For example, if the window size is 50 and the last part for which an acknowledge was received had a sequence count of 100, then no parts past the 150^{th} part will be sent. The combination of a slow uplink speed and short ground station overpass times create a need for the fastest, most efficient communication possible. In this case lessons were learned by communicating with

engineers that were previously involved with the SUNSAT mission [46]. Engineers involved with this mission found it was far more efficient to transmit all the data in a large data set at once and then retransmit any packets that weren't successfully received rather than receive acknowledgements for each packet. As the service supports the functionality of retransmitting packets, it was decided to follow the approach recommended by the SUNSAT engineers and not make use of the sliding window.

A second compromise that had to be made was the choice to abort a transfer if the final packet is not received during an upload operation. This is because the service definition does not allow for the communication of how many parts a SDU consists of. Therefore, if the final part is not received, the service has no way of knowing how many packets are missing from the transfer.

The final decision that had to be made concerned the passing of large quantities of data to and from the service. The mechanism used to pass large data units from a source process to the large data transfer service can follow different methods according to the memory resources available. For a downlink operation, the service data unit can be passed to the transfer service in one block or already split into data blocks. For this implementation, it was decided that the sending process would provide the service with a reference to the complete data unit available for download. This approach was chosen in order to abstract away the details of the operation of the service from mission specific processes. The entire protocol of the downlink (splitting up the data unit, acknowledging packets, handling transmission errors) is therefore confined to the large data transfer service. When receiving a SDU via an uplink procedure, it is up to the user to either supply enough storage space to temporarily store the data or ensure the target subsystem can piece together the separate parts.

4.6 The Filesystem

On a remote system, recording events and whole orbit data is necessary in order for operators to assess the operation of the system and make decisions about how to proceed with the mission. Error logs give an indication of any faults that may have occurred on the satellite as well as whether any of the recovery procedures were successful. A whole orbit data log can give the operator of a ground station an overview of the status of all the subsystems at a glance. Payload outputs such as scientific data and images also need to be stored until the next pass over the ground station provides an opportunity for them to be downloaded.

Due to the issues inherent in operating an autonomous system in outer space, it is possible that the OBC could experience several resets over its lifetime. Persistent storage capabilities are therefore required to retain this data. On CubeComputer, the main OBC used in the ESL, a micro SD card is used over a SPI bus as a mass storage device. This microSD card slot provides a platform for the persistent storage required for the above mentioned data to survive these resets.

4.6.1 SD cards

SD cards are NAND flash memory devices designed for security, capacity and performance. The SanDisk SD Cards currently used with CubeComputer include certain features which are relevant in the context of this project.

Although NAND flash is less susceptible to radiation induced errors than volatile memory, errors could still occur on the SPI bus and while data is stored in the memory waiting for download. To mitigate these errors, memory field error correction is supported by including an error correction code when sectors are written to. If errors are present when the data is read, the defects are corrected before the data is transmitted[47].

Wear levelling is a technique used to extend the life of flash devices. A particular block of flash memory can only support a certain number of read/write cycles in it's lifetime. Always writing to the same locations in the memory can therefore result in the device malfunctioning when the majority of the memory on the device is still functional. Wear levelling algorithms attempt to maximise the lifetime of a device by distributing read/write operations evenly over every block on the device. Typical NAND flash devices (such as microSD cards) are rated for around 100,000 program/erase cycles [48]. For this reason, wear levelling is implemented by the micro-controllers on most SD cards.

4.6.2 The File System

When dealing with persistent memory, implementing a file system can simplify development, operation and re-use of the software. Any decent file system abstracts away low level details of interacting with the memory and provides an interface for the initialisation and use of its features. The FATFS (File Allocation Table File System) is included in the CubeComputer board support package and was the initial choice of file system during flight software design. However, not all SD cards are guaranteed to include the specialised features discussed in Section 4.6.1 and FATFS does not natively include these features. A number of other file systems were therefore looked into to see if a file system could be found that did support this functionality.

- **JFFS2, UBI:** Implement wear levelling but can only be implemented on raw flash devices.
- **Btrfs, Reliance Nitro:** These and multiple other similar systems implement fault tolerance but are commercial and not open-source. They also include a host of other features not required for the flight software such as support for distributed systems.

As no suitable alternative could be found that was both suitable and realistic, it was decided to implement FATFS and leave it up to the SD card itself and the flight software to handle wear levelling and fault tolerance.

The following specifications of FATFS are notable:

- **Memory usage** According to the FATFS application note, the memory used by FATFS with all its functionality enabled can be determined by

  $$\text{Memory usage (bytes)} = 10675 + V*4 + 2 + V*560 + F*550$$

  where V is the number of volumes holding the file system and F is the number of open files.

- **File name length** The maximum length of a file name can be set to up to 255 characters according to the available memory.

- **Re-entrancy** File operations to files on the same volume are not re-entrant.

- **Critical sections** The FAT structure could be broken should a critical section such as a write operation be interrupted. The $f\_sync()$ function can be used to prevent data corruption should an OBC reset occur during a critical section.

4.6.3 The File System Interface Library

The File System Interface Library is the gateway through which the services and subsystems in the flight software access the file system. The functions in this library mainly act as wrapper functions for the functions included in the FATFS API. For reasons that will be discussed below, any process that wishes to interact with the file system should do so through this library. The library includes the following functionality.

- Initialise the file system.
- Create a new file.
- Delete a file.
- Create a new directory.
- Delete a directory.
- Write to a file.
- Read from a file.
- List files/directories in a directory.
- Obtain the size of a file.

There are a few reasons for confining access to the file system to be through this library. The first and most important reason is mutual exclusion. In a system with multiple tasks running simultaneously, it is difficult to guarantee that multiple tasks won't attempt to open and perform operations on the same file. Particularly, FATFS does not allow duplicated opening of a file in write mode. Accessing the file system through the library provides a simple way to manage access and therefore prevent data corruption and memory shortages.

The implementation of the library also allows the flight software developer to abstract away the details of interaction with the file system. For example, obtaining and releasing semaphores used for mutual exclusion, checking if files have grown too large and checking the available memory on the volume. In its current state, this method only allows one task to access one file at a time. While this may seem like the slowest method of handling file access from multiple tasks, it is the most robust. It is also expected that there will not be any processes with hard real-time deadlines that will require access to the file-system. If multiple-file access is required, new file objects and mutexes will need to be defined to handle the extra open files.

The use of the wrapper functions also makes the code more modular. If it is decided in future work to use a different file system or a completely different method of storage, then the particulars of FATFS will not be difficult to remove from the applications source code.

The implementation of the functionality included in the library will now be discussed. For most of the functions other than file system initialisation, the directory and name of the file on which to perform the operation are passed as arguments. The "file system access" mutex is then taken and held while the function is executed. It is returned when the function is complete and the file has been closed. In order to leave error handling procedures up to users of the library, various error codes are defined that can be returned by the library. The following error codes are defined:

- **fs_success** is returned if a library function is successfully completed.
- **fs_FSnotEn** is returned if the file system is not enabled or functioning correctly when the function is called.
- **fs_noDir** is returned if the directory passed to the function does not exist.
- **fs_noFile** is returned if no file exists that corresponds to the file name passed to the function.
- **fs_fileTooBig** is returned if a function attempts to add data to a file that is already larger than the allowed file size.
- **fs_rEOF** is returned if a function attempts to read past the end of a file.
- **fs_FSfull** is returned if there is no free space left on the mass storage device.

It is up to the user to check for the appropriate errors and define procedures to follow if an error is detected. The functionality of the File System Interface is implemented as discussed below.

File system initialisation This function initialises the "file system access" mutex and creates the FATFS file system on the microSD card.

Creating and deleting files and directories The functions to create files and directories perform two checks before creating the new object. Firstly, a flag which indicates that the file system is currently enabled (i.e. correctly initialised and operating) is polled. If it is, then the file system access mutex is taken and the

existence of the directory path passed to the function is checked. If either of these checks fail the function will return an error code.

Writing and reading data The library functions to read and write data perform the same two checks as those performed when creating directories with the addition of checking if the specified file already exists. If it does not, the function returns with a corresponding error code. It is left to a file's data source to check that a file does not grow too large. This is because each file may have different specifications for how large it is allowed to grow. The function for writing data to a file also checks if there is enough storage space left on the volume to complete the write operation.

List contents of a directory This functions populates a buffer supplied as a function parameter with the names of any directories and files within a specified directory. The names are added to the buffer as C-style strings with directories identified by appending a '/' character to the start of the name. Error codes are returned if the file system is not enabled, if the specified directory does not exist or if the supplied buffer is too small to contain the entire content of the specified directory.

4.7 Service 131: Mass storage interface

Although the specific file systems and on-board storage mediums used will differ for various satellites, these elements will be present in the vast majority of missions. It was therefore decided to develop a custom service which would provide management and data retrieval sub-services for a storage medium. In other words, an interface to an on-board mass storage device. As the CubeComputer OBC uses a microSD card for mass storage, the service was developed to use the file system interface documented in Section 4.6.3. However, due to the modular nature of the flight software, the function calls to the file system interface can easily be replaced by an interface to a different storage medium.

The decision to implement the on-board storage interface as a service and not a library is mainly due to the readability and extensibility that the service structure provides. Adding the interface as a service is a better way to include it as part of the generic framework of the flight software. Adding it as a separate subsystem would mean communication to the interface would be through service 8 which would group it with mission specific subsystems.

The sub-services listed below are implemented to give the service it's functionality. All the sub-services with identifiers above 128 are involved in the process of downloading a file.

- Service 1: List the contents of a directory.
- Service 2: Download a file.
- Service 3: Delete a file.
- Service 4: Reset the file system.
- Service 5: Format the drive and reset the file system.

- Service 6: Report the contents of a directory.
- Service 7: Request the download of a file.
- Service 128: Transmit the first part of a file.
- Service 129: Transmit an intermediate part of a file.
- Service 130: Transmit the last part of a file.
- Service 131: Report a download abort initiated by the sending end.
- Service 132: Receive acknowledge of successful file download.
- Service 133: Receive request for re-transfer of unsuccessfully transmitted file parts.
- Service 134: Transmit a part of a file requested for re-transfer.
- Service 135: Receive request from the receiving end to abort a file download.

The protocol for downloading a file is exactly the same as described in Section 4.5.2 for a large data download operation. However, the download of a file is implemented separately to the download of data to maintain low coupling between the modules. Placing support for downloading files in Service 13 would require the service to have knowledge of file names, directories and file reading capabilities for finding data that is requested for re-transfer. As these functions are not required for downloading data not stored on the file system, adding these functions to Service 13 would decrease the modularity of the software.

As this service is not a standard PUS service, a full definition of the interface to the sub-services is given in Appendix A.

4.8 Subsystem Command Managers

With services 8 and 11 implemented, any telecommand can now be routed to a subsystem or scheduled for execution at a future time. Service 8 is used to direct commands to an application process not belonging to a particular service. The majority of these processes will belong to one of the modules identified in Section 3.3. In order for these modules to receive telecommands and relay them to the destination hardware subsystem, each module contains a manager task. These tasks perform three main functions:

- To act as an application level interface between the flight software and the software of the subsystems;
- To process/handle data received from subsystems;
- To analyse housekeeping data and execute confinement/recovery procedures.

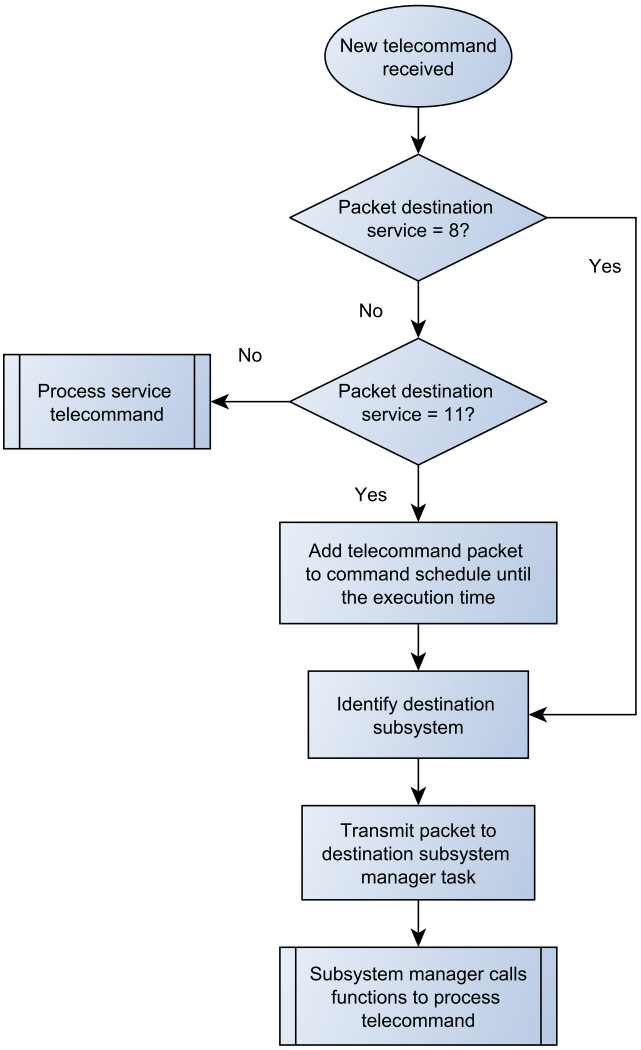


Figure 4.12 – Path of a telecommand from transceiver to subsystem manager

The manager tasks are one of the areas in the flight software where a line can be drawn between the generic application and the mission specific code. This is because the implementation of the functions listed above will potentially change for every subsystem and mission. The contents of a manager task will therefore be altered according to the subsystems belonging to a specific module.

Figure 4.12 shows the path of a telecommand destined for a subsystem manager from reception to execution.

The use of command queues provides a generic interface to subsystems by requiring commands to be formatted according to the following predefined structure.

```
typedef struct{
        uint8_t params[CDH_CMD_PARAMLEN]; // Command parameters
        uint16_t PacketID;                // Packet ID
        uint16_t PacketSC;                // Packet sequence count
```

```
        uint8_t function_id;            // Command ID
        uint8_t APID;                   // Destination module
        uint8_t param_len;              // Parameter count
}CDH_CMD_TypeDef;
```

The fields in this definition should satisfy the requirements of most commands but additional fields can easily be added if required. Adopting a generic interface for communicating commands around the flight software makes it easy to add and remove modules while requiring little modification to the source code.

4.9 Transceiver communication

In most satellites, the main OBC communicates with a ground station through a transceiver module. This hardware is responsible for handling the network protocol as well as the physical transmission and reception of data. As the main OBC will be the I2C bus master, it can push any data it wants transmitted to the satellite's transceiver board. However, when the transceiver receives new data from a ground station, it cannot push this data to the OBC as it is a slave device.

There are then two ways the OBC can be notified of the new data. Firstly, a flag can be set on the transceiver board indicating that new data is available. The OBC will then have to periodically poll this flag and execute a read operation when the new data is detected. A more efficient alternative to this can be used if the transceiver supplies the flag as a hard signal to the OBC. This signal can then be used to generate an interrupt on the OBC in which it executes a read operation to obtain the new data. The second, more efficient method is currently implemented in the flight software. The rest of this section describes the implementation of the interrupt.

In embedded systems, it is always desirable to keep an Interrupt Service Routine (ISR) as short as possible. This is to ensure that another event that would cause an interrupt is not missed while the initial interrupt is still being processed. FreeRTOS provides semaphores that can be used to defer processing to a handler task. As shown in Figure 4.13, the handler task initially tries to take the semaphore associated with the transceiver interrupt but is transitioned into a blocked state as the semaphore is not available. When the interrupt occurs, the ISR only clears the interrupt and performs a "give" operation to unblock the handler task. The handler task will then read the new data from the transceiver and be transitioned back into the blocked state to wait for another interrupt to occur.

In order to ensure that no interrupts are missed, the semaphore used to synchronise the handler task with the ISR is implemented as a counting semaphore. As briefly mentioned in 2.2.1.3, basic binary semaphores can be thought of as being able to queue up a single "give" operation at a time. If more than one interrupt occurs while the handler is busy, it will only detect that one interrupt occurred when it attempts to take the semaphore again. Counting semaphores can queue up multiple give operations ensuring that the handler task will process each interrupt that occurred. Using counting semaphores ensures that each received packet will be read from the transceiver module.

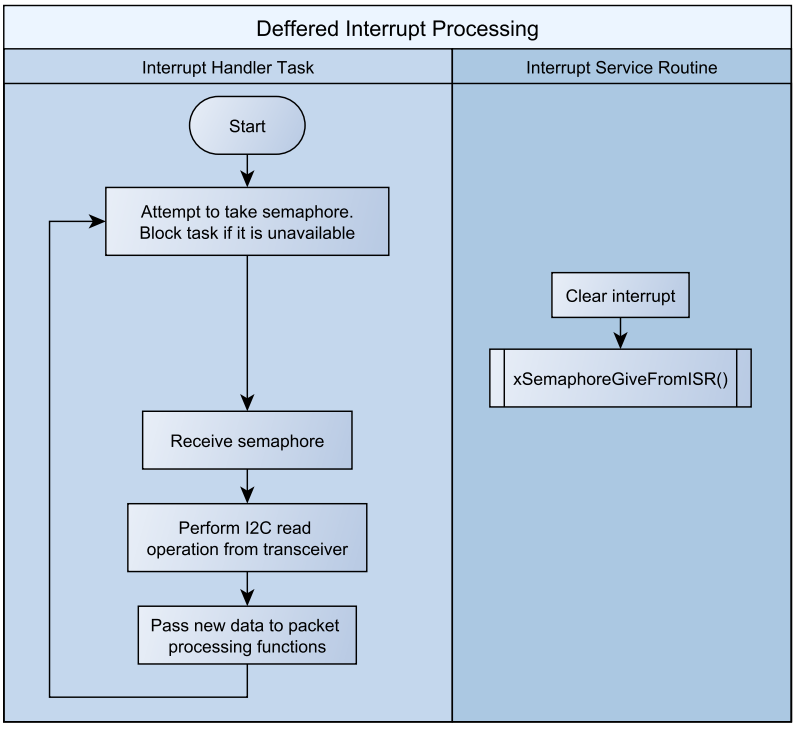


Figure 4.13 – Flow diagram showing how new data is detected and read from the transceiver

As described in Section 4.1.2, the priority inheritance mechanism implemented by FreeRTOS will prevent excessive delays in retrieving I2C data if other tasks are using the I2C bus. The system can also easily be changed to use a polling mechanism. By replacing the line that attempts to take the semaphore with a delay, and checking the flag before reading the data, the task will periodically check if the transceiver has received new data from the ground station.

4.10 Conclusion

This chapter presented the implementation of all the command and data handling components in the flight software. Starting from a low level, the way in which access to the I2C bus is controlled was presented. Next, the services involved in telecommand verification, management and scheduling were discussed followed by an explanation of the services and protocols used to upload and download large collections of data.The microSD card storage medium and the FATFS file system were also presented. To add features such as modularity and re-entrancy to the file system, a file system interface library was developed. The use of this interface makes it easier to use the file system within the RTOS environment as well as simplifying the process of replacing it if a different method of data storage is required.

With the command and data handling services defined, the subsystem managers were presented along with the way they relay commands and data between the flight

software services and the satellite's subsystems. The method with which commands are retrieved from the transceiver module and passed to the command handling services was also presented.

Chapter 5

The Housekeeping System

The challenges presented in Section 1.3 make housekeeping data collection and processing an essential task on any satellite. Housekeeping data collected from on-board sensors can be used to initiate autonomous recovery procedures on the satellite. The data can also be formatted, stored and downlinked for use by the ground station staff.

In the context of generic flight software, it is difficult to define what should be included in a housekeeping system that is easily configurable for any mission. Each satellite will have an unique set of sensors in each of its subsystems. Each sensor could then also have different requirements in terms of sampling frequencies, procedures to process the data, and threshold values at which error conditions are detected. It was therefore decided to develop the housekeeping system as a framework that allows users to specify the variables mentioned above as well as procedures to run on detection of an error condition. The two PUS services implemented to control this communication are Service 3 and Service 15.

Fault tolerance techniques are also important to prevent and mitigate errors. The sections in this chapter present the services used for collection and on-board storage of housekeeping data as well the fault tolerance techniques implemented.

5.1 Service 3: Housekeeping and diagnostic data reporting

PUS Service 3 provides the flight software with a method of reporting housekeeping data to the ground station. Housekeeping data is sampled and packaged according to a set of mission specific structures. Each of these structures is assigned a SID and holds a set of related data. For example, separate structures can be defined for ADCS parameters, on-board currents, event timings and temperatures. An example of a SID definition for payload data is shown in Figure 5.1.

Any number of structures can be defined depending on the requirements of the mission. Telemetry packets belonging to this service contain, as source data, both the housekeeping data and the SID of a specific structure. The SID is then used on the ground to link the data to a specific structure. The SIDs and the values they contain in their definitions are mission specific and will therefore be redefined for

Payload - SID 113 (0x71)

Description	Bits	Units
Detector temperature	8	°C
Microcontroller temperature	8	°C
Board temperature	8	°C
Current mode of the camera	1	
Read/write error of internal registers of the detector	1	
Image present in SRAM and ready to be transmitted	1	
Spare 1 bit (not used)	1	
Current program location being executed	4	
TOTAL	32	bits
	4	bytes

Figure 5.1 – Example of Payload data SID [8].

every mission. In order to avoid requiring the flight software to be heavily modified for each mission, the specific knowledge of the format of these structures is confined to the subsystems they belong to. Housekeeping diagnostic and recovery procedures are therefore also left to the developers of the specific subsystems.

The following sub-services were implemented in accordance with the recommendations made in [9]. Custom services (with subtype IDs of 128 and higher) were also implemented to extend the functionality of the service.

- **Subtype 5 and 6: Enabling and Disabling Housekeeping Parameter report generation.** Requests to these services either enable or disable parameter generation for a specific Structure ID. Housekeeping data sources belonging to disabled SIDs can still be sampled for on-board diagnostic purposes.
- **Subtype 25: Housekeeping Parameter Report** This sub-service generates and downloads telemetry packets containing the housekeeping data of each structure ID for which report generation is enabled. Telemetry packets generated in this way can also be sent to Service 15 for storage in the on-board packet stores (see Section 5.2).
- **Subtype 128: Changing reporting frequency** This sub-service can be used to change the frequency at which the different SIDs are sampled.
- **Subtype 129: Set SID mask** Requests to this service subtype change the value of the SID mask. As discussed later in this section, the SID mask indicates which of the values in each SID should be updated at each sampling interval.
- **Subtype 130: Request SID mask** This sub-service is used to request the current value of a specific SID mask. In order for a ground station to decode the data contained in a telemetry packet, it must have knowledge of the state of each SID mask. If the state of a SID mask is unknown at any point, this service can be used to verify it.

5.1.1 Housekeeping data collection

The generation of housekeeping data is achieved by implementing sub-service 25 as a FreeRTOS task. At every collection interval, the task passes empty structures specific to the relevant SIDs to the housekeeping collection functions located in each subsystem manager. These housekeeping collection functions populate the structures with the sensor readings belonging to the SID. When the populated structure is returned to the sub-service 25 task, a telemetry packet containing the housekeeping data as well as it's SID is constructed and downlinked. Figure 5.2 illustrates the operation of this task.

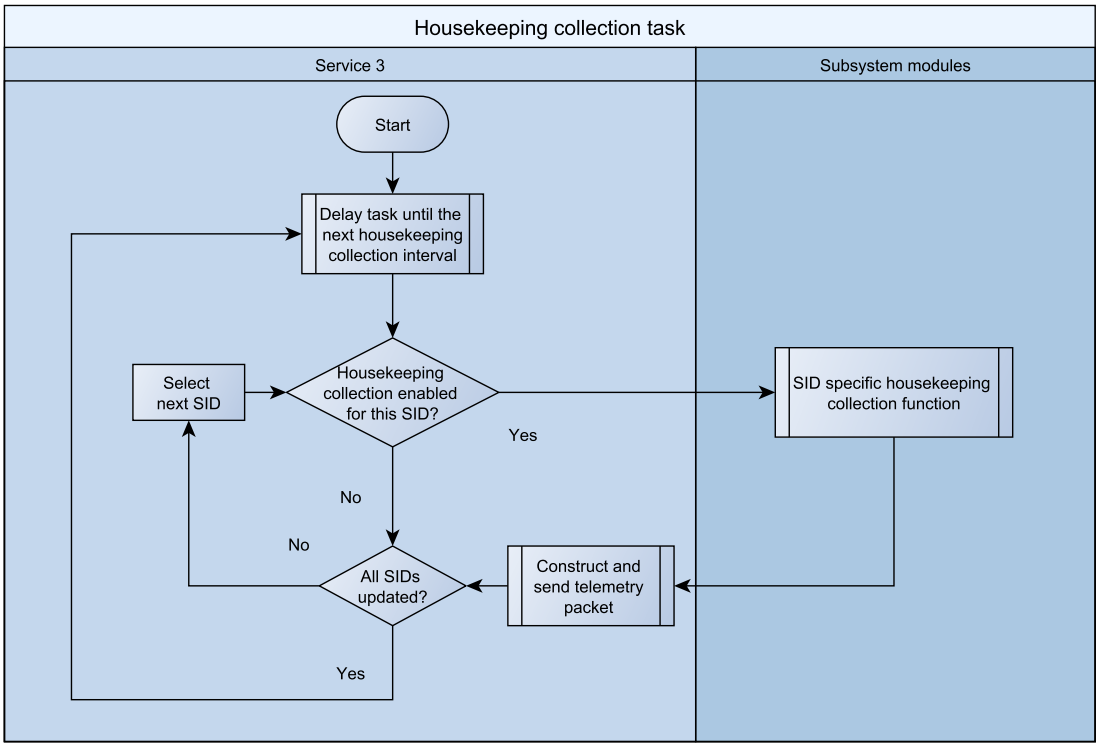


Figure 5.2 – Flow of the housekeeping collection task.

Two options exist for controlling the periodicity of the sub-service 25 task. FreeRTOS provides both the *vTaskDelay()* and *vTaskDelayUntil()* functions. Both functions allow the developer to specify a number of RTOS ticks for which the task should be delayed. The difference is that *vTaskDelay()* specifies a number of ticks relative to when it is called while *vTaskDelayUntil()* specifies an absolute time at which the task will be allowed to continue execution. This absolute time is calculated using a count of the number of ticks that have elapsed. This count is updated every time *vTaskDelayUntil()* is called allowing for accurate periodic execution. The delays introduced by *vTaskDelayUntil()* are therefore considered predictable and real-time.

5.1.2 SID masks

It may not be desirable to retrieve all the telemetry defined in a certain SID at each collection interval. Communication bandwidth and on-board storage space can be saved by only collecting those parameters that are of interest at the given time. For this reason, each SID has a mask of 4 bytes assigned to it. Each bit in the mask corresponds to a specific item in the SID. At each collection interval, if a specific telemetry item's mask bit is set to 0, it will not be sampled. Using 4 bytes, a collection of up to 32 sensors can be controlled for a single SID.

As only the telemetry values with corresponding SID mask values of 1 are included in the telemetry packet, the structure and length of a housekeeping telemetry packet depend on the state of the SID mask. The efficiency of the system could be increased if functions were added to construct telemetry packets to only be as large as needed. For example, using only a single bit instead of a full byte for a SID value that only indicates if a subsystem is enabled. This functionality is not currently in the system as its implementation depends heavily on the structures designed for a specific mission.

5.2 Service 15: On-board storage and retrieval

Service 15 provides a means to store telemetry packets generated on the satellite in so-called "packet stores". These packet stores are predefined locations in the main OBC's memory and are each assigned a Store ID that corresponds to a Structure ID defined in Service 3. Housekeeping packets generated by Service 3 can therefore be routed to Service 15 for storage in a specific packet store rather than being downlinked. Service 15 was included in the flight software for use in the following cases.

- If the satellite experiences low coverage from a ground station then the majority of Service 3 housekeeping packets will not be received. Storing these packets on-board during periods where no ground station is available will allow operators to retrieve information about the satellite for all stages of the orbit.
- If lowering power consumption is a concern for engineers, then packets can be stored rather than being constantly downlinked whenever they are generated. Specific packets can then be selected and downloaded as they are required.
- The service can be used to implement a "lost packet recovery" mechanism. If erroneous housekeeping packets are received by the ground station during an overpass, these packets can then be requested for download if they were stored in a packet store.

Figure 5.3 illustrates the concept of the service. In the flight software, the packet store is kept in the file system on the microSD card. Each SID from Service 3 has its own Store ID and each Store ID's packets are kept in a directory separate to the other Store IDs. The state of each packet store is recorded in a structure containing the elements shown below.

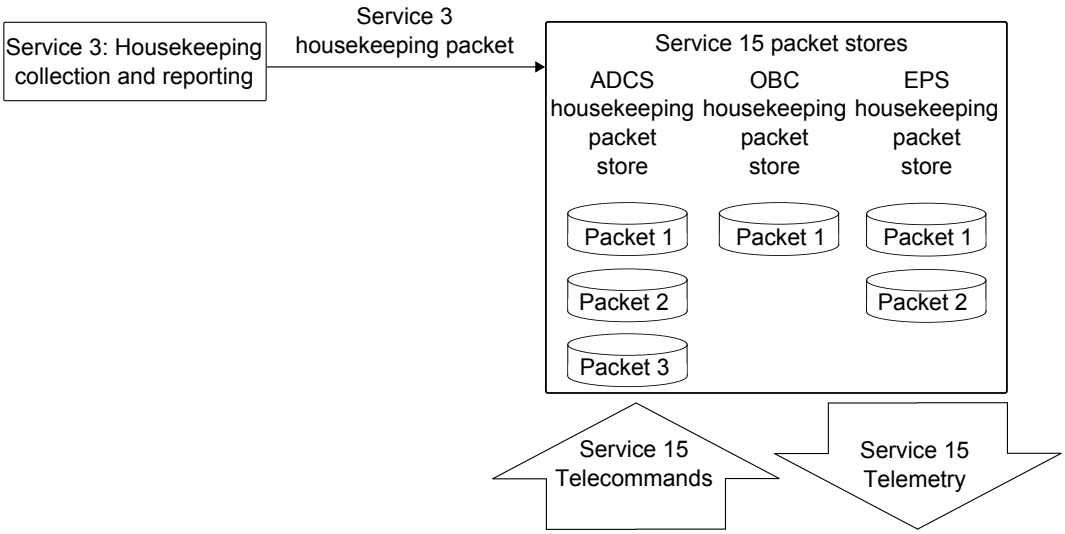


Figure 5.3 – Service 15 Concept. Adapted from [9]

```
/// Structure used to define a Store ID
typedef struct{
        uint8_t StoreID;          // Store ID
        bool enabled;             // Packet store enabled flag
        char store_dir[11];       // Store directory name
        char fName[21];           // Store file name
}STOREIDinfo_TypeDef;
```

An array of these structures is used to track the state of the service with each index of the array tracking a specific store. When the service receives a packet, the array index corresponding to the SID's store is determined. From then on, the relevant store and all its information is accessed with this index.

The sub-services implemented to provide an interface to this system are presented below.

- **Subtype 1 and 2: Enabling and Disabling storage in specified packet stores.** Requests to these services enable or disable storage in specific packet stores. If telemetry packet's corresponding packet store is enabled, it will be stored in the packet store rather than downloaded.
- **Subtype 9: Downlinking the contents of a packet store for a specified time period.** This sub-service handles the downloading of packets that were generated within a specified time period. In order to make this type of packet selection possible, packets are added to stores preceded with time stamps. As packets exist in the packet store as complete telemetry source packets, they are downloaded exactly as they are stored. The sequence number and time stamp contained in the generated packet will then serve to identify the packet as one being downloaded from a store.

- **Subtype 128: Abort packet store download.** When a request to this sub-service is received, a flag will be set indicating that a download abort was requested. The task responsible for downloading the contents of a packet store will check this flag after each packet is downloaded. If a request for abort is detected, the download operation will stop and be reset for the next download request.

Service 15 is implemented by using two FreeRTOS tasks. One to receive and store packets, and one to handle the download of packet store contents. The following sections will describe their implementation.

5.2.1 Packet Reception and Storage

The first task is used to receive redirected packets for storage in a specific packet store. Packets redirected for storage are received on message queue. The structure used to define each element of the queue is shown below.

```
typedef struct{
        uint8_t struct_packet[MAX_PACKETSIZE];
        uint8_t packet_sid;
        short length;
}S15_Packet_TypeDef;
```

Using this structure, each message holds the data for the entire telemetry packet it represents. If a pointer to data residing in Service 3 was used, one of two structural modifications would have to be made to the interface between Services 3 and 15. Either the memory being referenced would have to be protected with a mutex, or separate buffers would have to be defined for each Structure ID. If a mutex was used, situations could arise in which the housekeeping data collection processes in Service 3 end up waiting for some slow or faulty process in Service 15. This increased level of coupling between the services was considered unacceptable. Defining separate buffers for each Store ID was also deemed inappropriate as it would increase the number of source code modifications required each time the software was adapted for a mission. If many SIDs were defined, it could also result in higher memory usage than the method shown in the listing above. It was therefore chosen to pass the packets to Service 15 by value and not by reference.

Once packets are received by this service, they are appended as the newest entry to the packet store corresponding to the SID of the received packet. In the flight software, the packet stores have been implemented as directories on the microSD card. As packets are received, they are appended to the end of the newest file in the directory corresponding to the correct packet store. If a file grows too large, an error code is returned to indicate that a new file should be created. The reason for using the microSD card instead of volatile memory on the OBC is so that the packet stores will survive an OBC reset. The abundance of memory on the SD card also enables packet stores to maintain a large catalogue of packets.

As the PUS given in [7] assumes that packet stores are kept in RAM, it does not specify a storage format. A custom format for storing telemetry packets in files was

therefore implemented. The format of each packet store entry is shown in Figure 5.4.

Time of Storage	Packet Length	Packet
Enumerated (5 bytes)	Enumerated (2 bytes)	Enumerated (n bytes)

Figure 5.4 – Format of a packet store entry

The bytes containing the data shown above are written to a file as raw binary data. The time stamp indicating the time that the packet was added to the store is placed first. This is to make it easier to find specific packets requested for download. The time is constructed from 5 bytes in the same format used in CCSDS telemetry packets. The next two bytes indicate the length of the entire telemetry packet. After the packet length, the next bytes are the packet itself. It was decided that this structure would be the easiest to use when searching for and downloading specific packets using sub-service 9. This process is described in Section 5.2.2.

5.2.2 Sub-service Implementation

For sub-services 1 and 2, storage in the Store ID specified is enabled or disabled by setting a flag associated with the specified store. When the packet reception task receives a new packet to be stored, it checks if the store corresponding to the packet's SID is enabled. If it is, then the store entry header shown in Figure 5.4 is added to the front of the packet and the whole entry is added to the relevant store

When a "packet store contents download" request is received via sub-service 9, the first step is to find the first packet requested for download in the packet store. Four different time span options are available for this request.

- **All:** Every entry in the specified packet store is downloaded.
- **Between:** All the entries between two specified time instances are downloaded.
- **Before:** All the entries before a specified time instance are downloaded.
- **After:** All the entries after a specified time instance are downloaded.

Once the first packet corresponding to the request is found, housekeeping packets will be extracted and downloaded one by one exactly as they are stored. This carries on until the final packet specified by the request has been downloaded. Packets are read from the store in the following manner.

1. The time stamp and length of a packet are read from the packet store file.

2. If the time stamp of the packet does not fall within the specified time span, the file index is incremented an amount equal to the packet length. Thereby skipping the current packet and leaving the file index at the start of the next packet store entry.

3. If the time stamp does fall within the specified time span, the packet is read from the store and downloaded. The file index will then be at the start of the next packet store entry.

4. This process continues until a time stamp is read that falls after the specified time span or the end of the store is reached. The download process can also be aborted by a request to sub-service 128.

Once the contents of a store have been downloaded, there is usually little reason to keep the packets on-board. A way to remove packets from the store is therefore required. While the PUS for Service 15 specifies sub-services for this functionality, these sub-services deal with the removal of sets of packets. As the file system does not support the removal of data from a file, removing packets from a store would involve rewriting the file without the specified packets. As this would be very inefficient, it was decided implement the removal of packet store entries through the use of the "Delete file" sub-service included in Service 131 (see Section 4.7). The names of files in a store are taken from the time stamp of the first entry in the file, it is possible to choose files to delete by examining the contents of a directory.

Service 3 is separated from Service 15 as housekeeping collection should not be adversely affected by errors with the packet storage system or the file system. Service 15 is not as independent from the file system as it has to record the file and directory names used in the packet stores. The use of the File System Interface library in Service 15 makes the service easier to modify if a storage mechanism other than the microSD card is implemented. One way to make Service 15 more robust would be to implement functionality that creates a packet store in RAM should the SD card or SPI bus fail.

5.3 Fault Tolerance

As stated in Section 2.3, a transient or intermittent fault can corrupt or damage parts of the system that are persistent between different states, thus causing a permanent error. In a system as remote as a satellite system, even small errors with a low probability of occurrence should be treated seriously. In this section, the implementation of fault tolerant techniques in the flight software is discussed. As presented in Section 2.3, fault tolerant techniques can be implemented on an architectural level and an application level.

5.3.1 Hardware fault tolerance

Architectural fault tolerance techniques can be applied to both hardware and software architectures. Although this section concerns the implementation of software techniques, the selection and effectiveness of these techniques is influenced by the

hardware architecture. For this reason, a summary of fault tolerance techniques implemented on the CubeComputer is given. The information in this section is taken from [5].

Faults caused due to TID and SEL events are primarily hardware faults. CubeComputer can detect and disable an SRAM module in which a SEL has occurred within 20μs. The device can then be power cycled to remove the latchup. As SRAM is especially susceptible to SEUs, the stack and heap memory for an application running on CubeComputer will be kept in it's two external SRAM modules. When data is written to SRAM, it first goes through the EDAC module implemented in the FPGA. The EDAC module has the capability to detect up to 6 and correct up to two bit errors per byte. The same data is then written to both of the SRAM modules. In this way, if one of the modules fails or requires a power cycle, no data will be lost.

CubeComputer also contains both internal and external watchdog timers. Both of these timers need to be reset by the main application before timing out to prevent them resetting the OBC. The internal timer is used to detect lock-ups in the application code due to SEUs or programming errors. If an error causes the internal watchdog to malfunction, the external watchdog will power cycle the OBC in order to remove the fault.

5.3.2 Architectural level fault tolerance

As mentioned before, the flight software is developed in a modular structure. Each module performs its own application level checks to confine errors within a single module and prevent the spread of external errors.

Any shared memory was also assigned a mutex to prevent corruption. It was also insured that no two tasks required access to more than one of the same mutex. This lowered the possibility of a deadlock condition.

5.3.3 Application level fault tolerance

In the context of generic software, application level fault tolerance is difficult to implement as error detection generally requires knowledge of the intended state of the software or the intended output of a function. For example, the threshold value for the current drawn by a subsystem. Error handling and recovery procedures will also be implemented differently for every system. The flight software therefore only includes fault tolerance mechanisms applicable to every mission. This means that of the four fault tolerance stages discussed in Section 2.3.3, it is primarily only the detection step that is included in the flight software.

5.3.3.1 Error Detection

In a CubeSat system, implementation of fault tolerance is generally confined to single-version techniques. The use of multi-version redundancy techniques such as design diversity or TMR is undesirable to small CubeSat development budgets and the tight specifications of the CubeSat standard. Various combinations of the checks

documented in Chapter 2 are implemented at points in the flight software using *if* statements. Lower level checks include:

- Checking for **NULL pointers**.
- Checking if an **array index is out of bounds**. This includes indices that are negative or larger than the size of the array.
- Checking for a **stack overflow**. FreeRTOS can be set to perform run time stack checking although this increases the time taken to perform a context switch [4]. If it is enabled, a stack overflow will be caught and the hook function for an application stack overflow will be run. The handle and name of the task in which the overflow occurred will be passed to the hook function. At this point, two options are available. Either the OBC can be reset or the software can attempt to restart the task in which the overflow occurred. Restarting a specific task requires very careful design considerations and the task parameters may have been corrupted by the overflow. As the flight software does not use dynamic memory allocation, stack overflows are expected to be rare and a simple OBC reset is currently done in the hook function. Logging of the overflow can also be placed in the hook function.
- Checking for **numeric overflow**. Attempting to store a value in a data type not large enough to hold the value will result in data corruption. For example, the maximum value that an 8-bit unsigned integer can hold is 255. Attempting to store a value larger than 255 will therefore result in overflow. Using an 8-bit signed integer lowers the maximum value that can be stored to 127 due to the sign bit. Checks for numeric overflow are therefore specific to the data types they protect.
- Checking that **division by zero** does not occur.

Application level checks include:

- Checking for **invalid commands**. For example, checking that a command is being received by the correct subsystem.
- Checking for **invalid command parameters** such as parameters that are out of range and sequence numbers that are out of order.
- Checking for **invalid checksum** values in new commands and data.
- Checking that subsystems **return sensible values** and housekeeping data.
- Performing **time-out checks** using the watchdog timers.

The extent to which these checks are implemented is dependant on a number of factors including the probability of an error occurring in a section of code, the total code size and timing considerations. As these characteristics will change from mission to mission, these checks are only included in critical areas of the generic flight software application. The overhead and code size contributed by adding these

checks to the software is generally insignificant in comparison to the faults that can occur if these errors go unnoticed.

5.3.3.2 Fault treatment and continued service

The procedure followed to remove a fault or ensure it does not negatively influence the system depends largely on the state of the system and where it is detected. As discussed in Section 2.3.3.1, both forward and backward recovery techniques exist for removing faults from a system.

Forward error recovery techniques implemented in the flight software mainly include the use of the EDAC system in CubeComputer's FPGA and the use of the file system to facilitate data and error logging procedures. The EDAC system is used to execute "memory scrubbing" activities as described in 2.3.3.1. The CubeComputer BSP includes functions that will read a section of data from the external SRAM module through the FPGA and then write it back. This will correct up to two bit errors per byte of data read. The BSP functions automatically keep track of which addresses in the SRAM still need to be scrubbed. As the probability of more than two bit flips per byte occurring is very low, this process should not have a very high priority within the system. It is therefore placed in the FreeRTOS idle task. The idle task has the lowest priority in the system and therefore only runs when no other tasks are running. The scrubbing procedure therefore only runs when the flight software is idle. The use of Whole Orbit Data (WOD) and error log files can also be considered forward recovery techniques as they assist ground teams with deciding how to proceed with a mission when faults occur.

Backward error recovery is based on the idea of returning the system to a previous state in which a detected error was absent. It was decided not to implement a checkpoint method of backward recovery as these methods depend heavily on the full mission-specific satellite system and are complex to implement and maintain. OBC resets are the main method used to save the system from errors that cannot be removed by the forward recovery methods. These resets are mainly performed by the watchdog timer design presented in Section 5.3.3.3. For errors that can't be removed by a reset, a safe-mode flight software application is kept on CubeComputer's EEPROM. This application operates the satellite in the safest way possible while a ground station crew performs diagnostics.

5.3.3.3 Watchdog timers

A multitasking system presents more potential errors that will not be detected by a standard watchdog mechanism. As long as one task remains able to reset the watchdog counter, any number of other tasks could become unresponsive without the watchdog knowing about it. Two or more tasks could enter a deadlock situation as described in Section 2.2.2, causing them to remain at the same point in their execution indefinitely. Tasks with a higher priority could also starve lower priority tasks of CPU time, making them unable to execute at all. The watchdog timer therefore needs to be adapted to service each task independently of the others.

However, not all tasks can be serviced in the same manner. In [10], Murphy distinguishes between two types of tasks: regular tasks and waiting tasks. Regular tasks

run at periodic intervals as they complete a cycle of execution or react to a periodic timer. Waiting tasks react to events or inputs occurring at irregular intervals. As the flight software contains both regular and waiting tasks, a watchdog timer had to be implemented that accommodated both. Therefore, a watchdog manager task was implemented to add some intelligence to the timer. Two different methods of implementing the watchdog timer manager task were considered.

The first method was taken from [10]. In this scheme, the watchdog timer exists as a high priority task that checks on the operation of the other tasks. Each regular task has its own state flag that can assume a value of either *alive* or *unknown*. At some point in the regular task's execution, the flag is set to *alive*. If the watchdog manager task checks the flag and sees *alive*, it assumes the task is working correctly and sets the flag to unknown. If the watchdog task does not see *unknown* on any task's status flag, the watchdog timer is reset or "kicked". The regular task then needs to set the status flag back from *unknown* to *alive* before the next time the watchdog manager task reads the flag. These variables do not need to be mutually exclusive as atomic operations are performed on them.

As waiting tasks might not execute in every watchdog time-out period, an extra state is required to handle them. Before every blocking line of code in a waiting task, the state flag is set to *asleep*. Like *alive*, the *asleep* state is seen as a valid state by the watchdog manager. The flag is then set to *alive* as soon as the task receives input and begins executing. Figure 5.5 shows a diagram containing an example of how this method would work. In the figure, Task 0 and Task 1 are regular states and Task 2 is a waiting state.

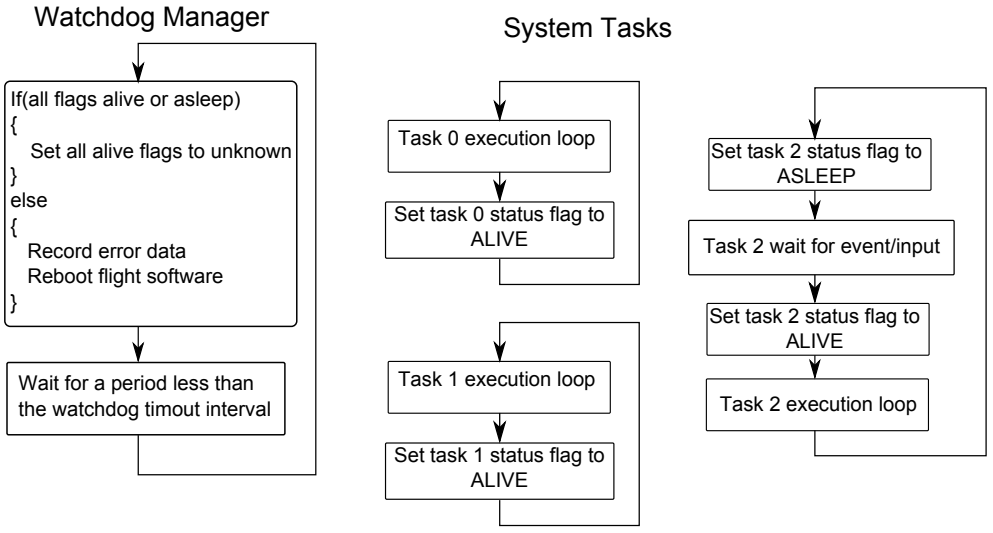


Figure 5.5 – Watchdog Manager Structure. Adapted from [10]

The second method was taken from [49]. As in the first method, this scheme consists of a single high priority watchdog manager task. However, instead of state flags, the watchdog manager task maintains a structure containing a counting integer for each other task in the system. Each time a task starts or loops, that task's counter

is incremented. If each task's counter is above zero when the manager task runs, the watchdog timer is reset. Advanced functionality can be added by including minimum and maximum values for each counter. These values can then be used to check if the tasks are running at the expected frequency. Of course, populating these fields accurately requires extensive profiling of the system and even then, separate timers may have to be set up for tasks that very rarely execute.

At the time of writing, the flight software consists of over 20 tasks of which nearly all can be classified as waiting tasks. The majority of these waiting tasks react mostly to telecommands from a ground station. As it is very difficult to predict how often certain telecommands will be used, profiling the software will most likely yield inaccurate results. The fact that the system has relatively few tasks also does not justify adding the complexity required by the second method. Therefore, it was decided to use the first method.

One exception to the mechanism shown in Figure 5.5 is tasks that are used to transfer large amounts of data. For example, the task used to manage large data downloads in Service 13. These tasks will receive commands to transfer data, set their state flags to *alive*, and begin the transfer procedure. In most cases, these transfers will last longer than the watchdog time-out period which would cause an unwanted reset. For this reason, resetting the state flags of the tasks performing the transfers needs to be done at intervals during the transfer. For example, after each packet store entry in a Service 15 download or each SDU part in a Service 13 download.

The frequency at which the watchdog manager task checks the state flags is dependant on the execution speed of the slowest task. Lower priority tasks may also take longer than expected to execute if high priority tasks are very active over a period. The watchdog time-out period can therefore only be accurately determined once the entire system has been assembled. Once the execution time of the slowest task has been determined, the time-out period can be set slightly longer to account for scheduling jitter and execution variation. If an unresponsive task is detected by the watchdog manager, debugging information about the malfunctioning task can be saved before the timer is allowed to expire.

5.4 Conclusion

This chapter presented the services and fault tolerance mechanisms that make up the housekeeping system. The two services implemented were Service 3: Housekeeping Reporting and Service 15: On-board Storage and Retrieval. These services control the collection, storage and transmission of housekeeping data. Users of the flight software are required to implement mission specific housekeeping structures as well as functions to populate them according to the requirements of the subsystems. The file system interface is used by Service 15 to store housekeeping packets.

Various fault tolerance mechanisms were implemented to increase the reliability of the satellite system. Due to the limited capacity of CubeSat systems, the implemented tolerance techniques consist mainly of single-version software techniques. Single-version techniques that were implemented include architectural and applica-

tion level checks, a memory scrubbing procedure and a watchdog timer tailored for a RTOS. The file system library discussed in Chapter 4 can also be used to maintain error logs or storage of data other than what is stored by Service 15. This data can then be downloaded and inspected by ground station teams to identify faults and initiate recovery procedures.

Chapter 6

Testing and Verification

The previous chapters of this thesis described how the various components of the flight software were designed and implemented. This chapter details the methods that were used to evaluate the operation of the software.

The first section of this chapter details all the components that were developed and included exclusively for testing purposes during the development of the flight software. Next, different components and characteristics of the flight software were identified that could be tested in order to verify that the application was working as desired. The rest of the chapter details the tests that were carried out as well as the results that were obtained.

6.1 Testing phase set-up

During the development of the flight software, the modular structure allowed each component to be tested as it was developed. For these tests, it was desirable to obtain a set-up as close to a full communication chain as possible. To achieve this, various hardware was used along with the main OBC and testing software was developed. This section presents these additional components and describes the testing set-up used during development.

For the tests conducted below, the flight software was loaded onto CubeComputer version 2B. Models of the CubeSense and CubeAim ADCS systems were available for integration with CubeComputer. The CubeSense model was added with a camera acting as a Nadir sensor. As discussed in Section 6.1.1, a CubeDock module was added in order to simulate the missing subsystems. The CubeDock module would primarily be used to simulate a transceiver module.

6.1.1 CubeDock

CubeDock is a general purpose platform developed in the ESL for testing and simulation purposes. The standard CubeDock setup consists of a Giant Gecko MCU, power regulation and control, UART and WIFI interfacing as well as lines configurable by means of link resistors. These link resistors allow the user to include or exclude functionality from the board on a hardware level. Having the same dimensions and header configurations as the other CubeSat hardware boards developed in

the ESL, it can easily be added to a stack of CubeSat hardware and programmed to perform the desired function. For this project, programming and integrating a CubeDock with CubeComputer was the only way to sufficiently test if the flight software would be able to effectively manage an entire satellite system.

The reason for this was that purchasing a copy of each subsystem's hardware could not be justified for the sole purpose of testing this project. In order to test if the flight software on CubeComputer could manage and maintain an entire satellite system, certain subsystems would need to be simulated. As CubeDock can be assigned multiple I2C addresses, the interfaces to multiple subsystems could be simulated by programming CubeDock to accept and reply to telecommands and telemetry requests. The following procedures and aspects of the flight software could then be effectively evaluated and stress tested.

- **Subsystem telemetry acquisition and logging:** In order to simulate a subsystem's telemetry and data capture capabilities, data structures containing fabricated sensor readings and image data were hard coded into the simulated subsystems. This enabled the data handling, housekeeping and logging procedures to be tested. The effectiveness and impact of recovery procedures could now also be tested by seeing if the system responded correctly to abnormal telemetry received from subsystems.
- **I2C management:** With the exception of the SD card and the filesystem it supports, every subsystem managed by the flight software communicates with the OBC over I2C. CubeComputer has a primary I2C bus used for general subsystems and payloads as well as a secondary I2C bus intended to be used exclusively for ADCS subsystems. This gives the bus the potential to become a major bottleneck in the command and data handling system. Therefore, it was important to ensure that the I2C bus was managed to allow effective and fair communication with each subsystem while still managing to meet hard real-time requirements.
- **Modular programming:** One of the goals of this project was to develop the software in a modular, reusable style. The ease with which the interfaces to the subsystems being simulated on CubeDock could be added to or removed from the flight software application would be a good measure of the extent of the software's modularity.

As one of the subsystems being simulated on CubeDock would be the transceiver board, the UART interface to the PC used for testing would be contained in CubeDock, which would then be connected to the PC via serial port. This accurately simulates the final implementation of the software in which the main flight software application is not exposed to the details of communication with the ground station. Data is simply sent in the correct format to the transceiver board.

6.1.2 Ground Software Simulation

For the majority of the project, the various functions of the satellite were tested by sending commands and data to it via UART from a PC. In order to do this, an ap-

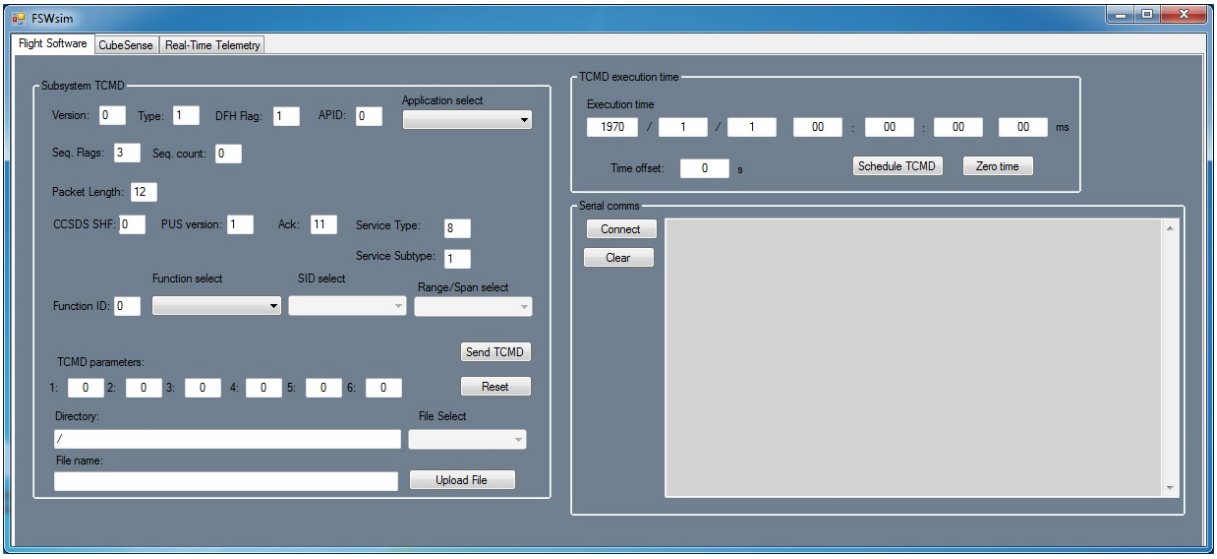


Figure 6.1 – The main tab of the ground software simulation application

plication was developed in C# using an express version of Visual Studio 2012. This application can be used to send commands and data to the satellite as well as receive and decode the various telemetry packets, file downloads and image downloads. The application was developed to conform to the CCSDS standards discussed in Section 3.4.1. The packetisation of the AX.25 protocol is not included in the communication between the application and the flight software as this protocol is handled by the transceiver on a satellite and the flight software therefore needs no knowledge of it. Image data that is downloaded from the flight software can be converted to a bitmap image file and displayed in the application. The main tab of the ground software simulation software is shown in Figure 6.1.

The final test setup that was used for the majority of the development is shown diagrammatically in Figure 6.2. The CubeDock module is connected to the PC through a UART to serial converter. Using the hardware and software described in this section, the transfer of data packets to and from the flight software could be tested as if it were part of a full satellite system. Each of the services, subsystem managers and interfaces could be tested as if the flight software was receiving telecommands through an actual transceiver. The packetisation and transmission of telemetry source packets could also be tested. Detection and handling of erroneous elements in packets such as incorrectly set flags or out-of-range parameters were tested by explicitly adding errors during the assembly of the packets in the ground station application.

6.2 System Evaluation

During development, the set-up described in Section 6.1 was used to test each component of the flight software individually as it was implemented. Having implemented all the functionality that would be included in the generic flight software,

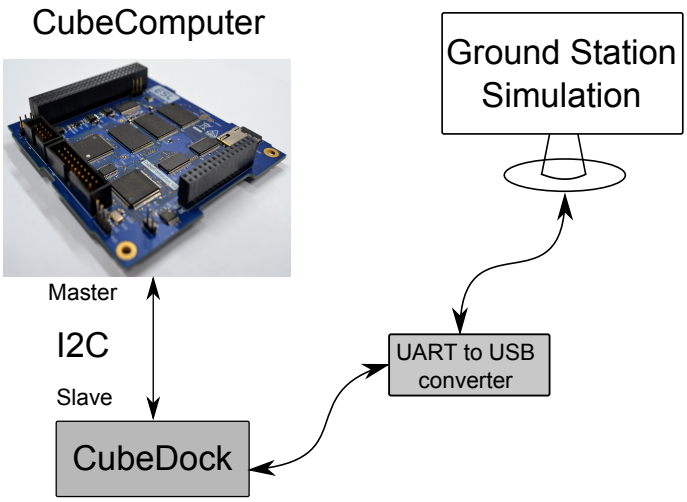


Figure 6.2 – Diagram of the test setup used during debugging and testing

the system could now be tested as a whole. A number of tests were designed and performed in order to determine whether the software functioned as desired.

6.2.1 System configuration

Before qualifying the system, a number of modifications were made to bring the software as close to a final version as possible. This was important to do before the tests were carried out as the results could affected by some of the changes.

- The priorities of different tasks in the software were simply set according to their real-time requirements. The task managing the ADCS algorithms was given the highest priority after the watchdog manager task. The priority of the interrupt handler task for the transceiver "Receive Ready" interrupt was set lower than the ADCS task, but higher than most other tasks to ensure the buffers in the transceiver would not overflow because data was being read too slowly.
- The stack usage of the tasks in the system was measured and the size of the memory assigned to each task was adjusted accordingly. The stack usage of each task in the application was measured using the

  `uxTaskGetStackHighWaterMark()`

  function from the FreeRTOS API. This function returns the minimum stack space that was unused since the task began executing. This information was used to identify tasks that were allocated more stack space than they would ever need. Reducing the memory assigned to these tasks would optimise the total memory required by the flight software application.
- A coding standard was implemented to make the software more understandable and easier to use.

With the final system modifications made, the evaluation could begin.

Test	System Aspects Tested	Test Method
1. Run multiple processes simultaneously	RTOS scheduler, resource and memory management.	Simultaneously download an image, send telecommands, and collect housekeeping data.
2. Conformance to CCSDS standard.	Correct packet formats and protocols	Test with SCS GS software and C# software
3. Stack usage and memory footprint.	Optimal memory use.	Test high watermarks
4. Real time requirements testing.	Real time execution and determinism, scheduling of commands	CubeControl synchronisation
5. Software robustness.	Robustness against erroneous packets, invalid TC params, etc	Send packets with errors in parameters, checksums, etc
6. Execution timings	Execution time of commonly executed areas of code	Hardware measurements using GPIO pins

Table 6.1 – Tests used to evaluate the flight software

6.2.2 System testing

The first step in the testing process was to determine which aspects of the system could be evaluated to determine if the system was functioning satisfactorily. Table 6.1 shows the different aspects that were tested and the methods used to evaluate their operation. In the remainder of this section, the tests and their results are presented. The test software presented in Section 6.1.2 was used for most of the tests discussed below.

Test 1: Simultaneous execution of processes. For this test, a number of commands were given to the software in addition to it performing various background tasks. The default background tasks of memory scrubbing and the watchdog manager were enabled as well as the periodic collection and logging of housekeeping data from various subsystems. In addition to these background tasks, the download of a large file from the file system was initiated. With the file download and background task in process, various commands were sent to subsystems from the test software. The timely execution of these commands was confirmation that the flight software could execute multiple processes simultaneously.

Test 2: Conformance to CCSDS standard. While the test software was developed to use the CCSDS packet format, it was still desirable to test the flight software with externally developed ground software to insure the CCSDS standard had been correctly interpreted. To do this, the recently released Satellite Control Software (SCS) ground station application was used. The SCS application also makes use of the PUS standard and could therefore be used to verify that the format used for telecommand and telemetry packets was correct. Scripts for sending Service 15 telecommands from the SCS application were included in the default installation and could be sent to the flight software using SCS's Single Script Client.

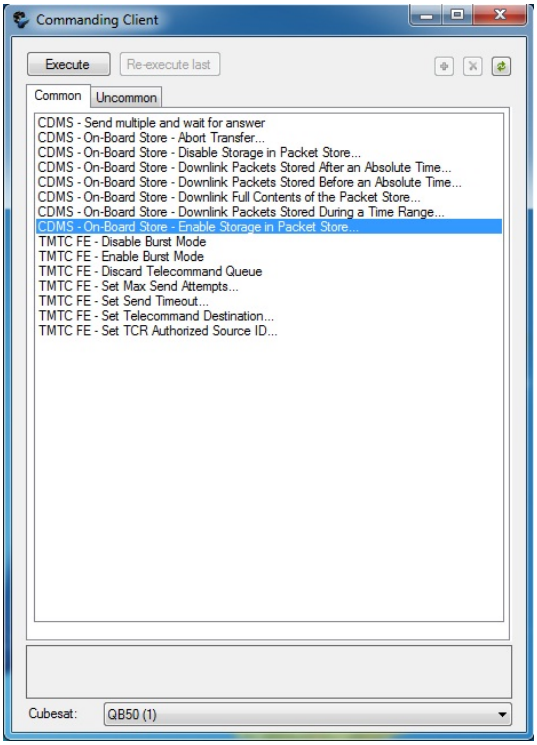


Figure 6.3 – Single Script Client

It was demonstrated that the flight software could successfully receive and decode these commands confirming that the packet structure had been implemented correctly. The protocol used to receive a file uploaded using Service 13 could also be checked using the Large Data Upload Client included in the installation.

This sub-application was successfully used to upload test files that were then stored on the microSD card and checked to verify that the upload was successful. The successful upload of files using the SCS application confirmed that the Large Data Transfer protocol had been successfully implemented in the flight software

Test 3: Stack usage and memory footprint. The binary size of the final application is 170 kB. This is excluding all test code and subsystem interfaces used for testing throughout the project. In order to optimise the size of the heap assigned to FreeRTOS, the stack size of each task was set to just above the maximum requirements of the task at the time of writing. The length of all the communication queues were also minimised. After this optimisation, the operating system could run with 35 kB of heap memory available to it.

Test 4: Real time requirements testing. During the implementation of the command scheduling service, it was verified that commands could be scheduled for execution at specific times. It was still desired to perform a practical test that would demonstrate the ability of the system to meet hard real time requirements. A suitable test was found in the ESL's CubeControl module. The execution loop in CubeControl contains an execution loop that contains a 5 ms window in which it must receive a synchronisation command from the main OBC. If it does not receive

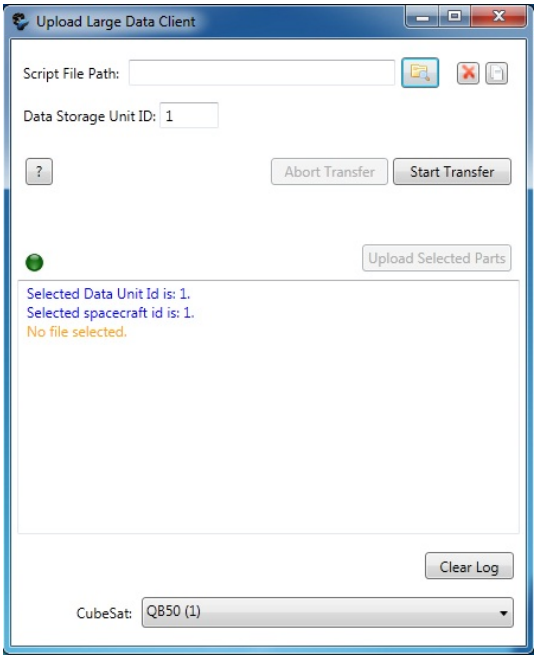


Figure 6.4 – Upload Large Data client

this command, it will return an error and malfunction. The CubeControl module could therefore be used to test whether the flight software could meet hard real time requirements. After integrating the CubeSense module with the flight software to a minimal extent, the application was run and the correct functioning of CubeSense was confirmed while the system performed various unrelated tasks. This validated the compliance of the flight software with the hard real time requirements.

Test 5: Software Robustness. A number of tests were performed to evaluate the robustness of the flight software. Certain tasks were hard-coded to intentionally become unresponsive at certain points. The watchdog manager task would then identify the offending task as being unresponsive and allow CubeComputer's watchdog timer to time out. Packets were also sent to the flight software with checksum errors and incorrect parameters. It was shown that the software could detect these errors and, in appropriate cases, notify the ground station of the failure.

Test 6: System timings. Although Test 4 had already verified that the system could comply with real-time requirements, it was still desirable to obtain some values for the execution timings of certain aspects of the flight software. This would give insight into the operation of the system and help to identify any bottlenecks in execution. The simplest, most accurate method that could be used to obtain these measurements was through the use of an available GPIO pin on the MCU. An easily accessible pin was found in the Data line of the secondary I2C bus. To use this pin for measuring timings, the initialisation of the secondary I2C bus was removed from the flight software and the pin previously used as the Data line was initialised as a standard output pin. The pin was then connected to an oscilloscope. By toggling the level of the pin at various points in the software, the execution time of sections of code could be measured.

Operation	Execution Time (ms)
1. Boot Time	34
2. Transfer of packet to subsystem manager	0.12
3. Reception of packet over I2C and transfer to subsystem manager	26
4. Generation and transmission of telecommand verification	3
5. Transfer of packet to subsystem manager while downloading a file from the file system	11.6
6. Transfer of packet to subsystem manager while downloading a file from the file system with the priority of telecommand reception task raised	3.6
7. Writing a 242 byte SDU part to the file system	33

Table 6.2 – Tests used to evaluate the flight software

The timings obtained are shown in Table 6.2. All of the timings in the table were obtained with the watchdog timer manager task and memory scrubbing procedure running in the background. Timing 1 is the time taken to initialise every peripheral, manager and service used by the basic flight software application. Timings 2 to 6 indicate the effect of various events on the transfer of a telecommand packet around the flight software. Timing 2 is the time taken for a telecommand packet to be transferred to its destination after it has been received from the transceiver. This can therefore be used as a base timing to which the other timings can be compared. Comparing timings 3 and 4 to timing 2 leads to the conclusion that the slow speed of the I2C bus is a bottleneck in the operation of the system.

To obtain timing 5, a file download task was run at the same priority as the telecommand handling tasks. A file download procedure was chosen as it a relatively lengthy and resource intensive operation. Timing 6 then shows the effect that raising the priority of one of the command handling tasks can have. Timing 7 simply shows the time required to write a buffer to a file in the file system; an operation that could happen over 1000 times during a file upload. The relatively long time required to do this is not considered to be a major bottleneck as data written to files will generally be received over the I2C bus at a similar rate.

The success of the tests described in this chapter indicated that the software was functioning as desired. Two of the desired characteristics of the software, namely its genericity and re-usability, were difficult to test. As stated throughout the thesis,

multiple measures were taken to strive towards these goals. The ease with which subsystem modules such CubeSense and CubeControl had been integrated with and removed from the flight software was considered a good indication that these goals had been achieved; at least to an elementary extent.

Chapter 7

Conclusion

The primary goal of this project was to develop a generic flight software application for a CubeSat using FreeRTOS. Initial phases of the project consisted of research to determine the requirements and specifications of flight software. This research identified the need for a command and data handling system, subsystem managers, a housekeeping system and the implementation of software robustness techniques. It was also decided to develop the software in a modular fashion to make the software easier to adapt for a specific mission.

During the design phase, it was decided to adopt the CCSDS standard and its extension, the PUS. The use of this standard simplified the design process, provided a definition for the interface between the satellite and the ground station, and increased the re-usability and maintainability of the software. The PUS contains definitions for services that perform the functions of each of the components identified during the research phase. These services were subsequently implemented giving the flight software the required functionality. Aside from the PUS services, other vital components were also implemented to complete the application. Interface libraries for the I2C bus and third-party file system were developed as well as subsystem managers to interface between the services and on-board subsystems.

Various fault tolerance techniques were also implemented to increase the robustness of the application. A watchdog timer manager was implemented in order to detect unresponsive tasks and return the system to an operational state. Application level coding checks were included in critical and frequently used areas of the software to protect against human errors, software bugs and other unexpected faults. To protect the system against excessive radiation-induced bit flips in the SRAM, existing memory scrubbing routines were included in low priority tasks.

Once all of the above-mentioned components had been implemented, various tests were carried out to evaluate the functionality and performance of the flight software. The first test was to verify whether the CCSDS packet structure and file transfer protocols had been correctly implemented. To do this, the SCS ground station application was used to send telecommands and files to the flight software, which could successfully decode these messages. In addition to testing each individual component, the ability of the software to simultaneously respond to various requests while performing separate background tasks was tested. The capability of the system

to meet hard real-time requirements was also tested along with its ability to detect, mitigate and recover from errors and faults.

Once the above-mentioned tests had been completed, each of the project goals had been achieved and the first version of the flight software application was complete. The goal of implementing the flight software as a generic application had been achieved by implementing all the core functionality required from OBC software.

7.1 Future Work

As the flight software has yet to be tested as part of a complete satellite system, there are undoubtedly hidden faults that are yet to be discovered. However, until such a time, there are a few improvements that can be made to the existing components of the software. Only existing components are considered here as adding new functionality to the software increases its complexity and therefore reduces its re-usability.

- Intelligence could be added to the file system initialisation procedure to provide backup storage should the microSD card or SPI bus fail. In these cases, a new file system could be created in flash memory. The file system could be created with less functionality and available storage space but would allow the storage of critical diagnostic data.
- The Cortex-M3 used on CubeComputer contains a Memory Protection Unit (MPU) that supports up to 8 protected memory regions. FreeRTOS contains support for the Cortex-M3 MPU and it can therefore be used to increase the robustness of the system by only giving tasks permission to use memory and peripherals that they require.
- The watchdog manager task can be improved to monitor tasks in more intelligent ways depending on the type of task. For example, distinctions can be made between tasks according to frequency of execution or their execution time. This would help to avoid unnecessary resets while improving the speed at which unresponsive tasks are detected. Efforts could also be made to only reset individual unresponsive tasks, returning the system to full operation without resetting the entire application.
- The memory management schemes provided by FreeRTOS could be extended to manage the heap memory used by the entire application. If this is done, it will be safer to use libraries that use dynamic memory allocation such as data compression libraries.

In conclusion, the implementation of tried and tested standards is highly recommended when developing flight software. Especially for inexperienced developers and software which is intended for multiple different missions. The use of a RTOS is also recommended as long as an appropriate RTOS is selected according to the needs of the mission. The tools provided by such a product can be of great assistance during development. As the complexity that can be included in a CubeSat

grows, the use of a RTOS in CubeSat flight software is expected to become standard practice.

Appendix A

Service 131: File System Interface

Each of the PUS services discussed in this thesis defines formats for the application data and source data fields of its telecommand and telemetry packets. The definitions used for elements of the standard PUS services can be found in [7]. During the course of this project, a custom service was to developed to provide an interface between the ground station and the file system in the flight software. As the packet formats for this service are not documented anywhere, this appendix is provided to illustrate the structure of requests and reports defined for Service 131.

A.1 List directory contents (131, 1)

This request shall generate a list containing the names of any files and directories within a directory. The structure of the resulting telemetry packet is defined by Subtype 6.

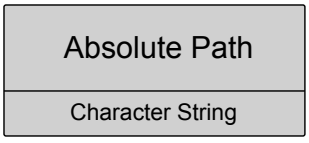


Figure A.1 – Service 131,1 telecommand packet application data

Absolute Path:

This field contains a null terminated string making up the absolute path to the directory for which the list shall be generated.

A.2 Downloading a file from the file system (131, 2)

This request will initiate the download of a file stored on the mass storage device. If the file is too large to be downloaded in a single telemetry packet, the file will be

downloaded via the large data transfer portion of this service. The service subtypes involved in the transfer protocol are subtypes 128 to 135.

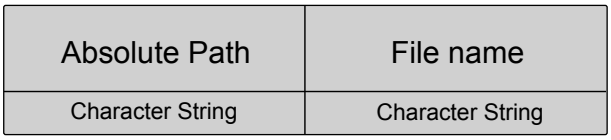


Figure A.2 – Service 131,2 telecommand packet application data

Absolute Path:

This field contains a null terminated string making up the absolute path to the directory where the file to be downloaded is situated.

File Name:

This field contains a null terminated string making up the name of the file to be downloaded.

Only one file can be downloaded at a time. The system needs to either receive an acknowledge of successful download or an abort command before it can initiate the transfer of a different file.

A.3 Deleting a file from the file system (131, 3)

This request will remove a file from the file system. The structure of a telecommand packet belonging to this sub-service is the same as a packet belonging to sub-service (131,2)

Absolute Path:

This field contains a null terminated string making up the absolute path to the directory where the file to be deleted is situated.

File Name:

This field contains a null terminated string making up the name of the file to be deleted.

A.4 Reset the file system (131, 4)

This request will reset the file system without clearing the data currently stored on the mass storage device. A telecommand packet belonging to this sub-service contains no application data.

A.5 Format the mass storage device and reset the file system (131, 5)

This request will clear all data from the mass storage device used by the file system. The file system will then be reinitialised. A telecommand packet belonging to this sub-service contains no application data.

A.6 Directory contents report (131, 6)

This report contains the list generated by Service (131, 1).

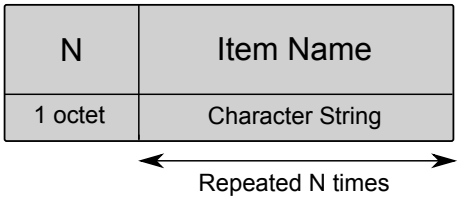


Figure A.3 – Service 131,7 telemetry source packet, source data

N:

The number of items contained in the list.

Item:

A null terminated string containing the name of a file or directory.

A.7 File requested for download (131, 7)

This report contains a file requested for download by Service (131, 2).

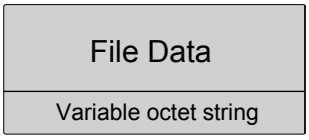


Figure A.4 – Service 131,8 telemetry source packet, source data

File Data:

The data making up the file requested for download.

A.8 File download service subtypes (131, 128) to (131, 135)

The list below contains the service subtypes involved in downloading a large file from the file system to the ground station.

- (131, 128): Downlink the first part of a file.
- (131, 129): Downlink an intermediate part of a file.
- (131, 130): Downlink the last part of a file.
- (131, 131): Downlink a report of transfer abort by the satellite.
- (131, 132): Receive downlink reception acknowledge.
- (131, 133): Receive report of unsuccessfully received parts.
- (131, 134): Downlink a part requested for re-transfer.
- (131, 135): Receive transfer abort from the receiving end.

The protocol for the file transfer matches that of service 13 except for the way in which the Large Data Unit ID is used. In service 13, the Large Data Unit ID field is populated with mission-specific values that uniquely identify the data that is being downloaded. As files will be created and deleted often with different names and contents, it does not make sense to try and assign an unique ID to each file. The ID will therefore simply act as a count for the number of files downloaded and will be incremented with each download. In this way, the ID field can still be used to ensure each received packet belongs to the correct file.

Appendix B

Mission specific elements of the flight software

The flight software in this project has been developed to provide a framework that is as generic and configurable as possible. However, every satellite mission is unique and it is unavoidable that a number of additions and configurations will have to be made to the software for every mission. This appendix provides a list of additions that will need to be made to the software when configuring it for a specific mission.

B.1 Addition of subsystems

The primary modification that will have to be made is the addition of the mission-specific subsystems. The following steps should be taken in order to integrate a subsystem with the flight software.

- Integrate the interface to the subsystem with the relevant subsystem manager task. After this has been done, the manager task should be able to receive a telecommand and format it into an I2C message that can be sent to the subsystem.
- Alter the FSW_COMM_receiveTC task to decode messages received from the transceiver module according to the transceiver's transfer format.
- In order for Service 3 (housekeeping collection) to be able to collect the required telemetry from the added subsystems, two additions are required. The structures that hold the housekeeping data as well as the functions that populate the structures. The structures will be located in Service 3 while the housekeeping data collection functions will reside in the relevant subsystem modules.
- Application level fault tolerance to verify the correct functioning of subsystems.
- Recovery procedures that can be executed should a fault be detected in one of the subsystems.

B.2 CCSDS modifications

Various modifications will also have to be made to the PUS services to configure them according to the needs of the mission.

- The APIDs for each process will need to be defined in order to identify the destination of a telecommand packet and the source of a telemetry packet.
- If any new application processes are defined, the tracking of sequence counts for received telecommands and released telemetry packets will need to be implemented in order to uniquely identify telecommands.
- Functions to configure the services during operation should be added to the relevant subsystem managers.
- Requests to Service 1 (telecommand verification) should be placed in areas of the code where confirmation of telecommand execution will be important. Mission specific error codes should also be defined for notifying ground station operators of the reason for a telecommand failure.
- Service 8 should be modified so that it can direct a telecommand from the ground to any application process.
- The length of the command queue managed by Service 11 should be set according to the needs of the mission.
- Each large set of data that is expected to be downloaded or uploaded using Service 13 (Large Data Transfer) should be assigned an ID in order to identify what sort of data a packet contains. Functions should also be implemented to transfer packets received during an upload operation to their destination on-board the satellite.
- The number of packets that can be missed in an upload or download procedure before the transfer is aborted should be decided upon. The array that stores the sequence numbers of missed packets should then be sized accordingly.
- Each structure defined for Service 3 should have a corresponding store in Service 15.

B.3 System modifications

A few components of the flight software will need to be configured regardless of whether changes are made to the subsystem definitions and PUS services.

- The priorities of all the tasks in the flight software should be set according to the needs of the mission. For most missions, it should be sufficient to operate most of the tasks on the same priority level. The watchdog manager task should always run at a higher priority than any other task.

- The start-up procedure of the flight software should be configured to ensure that the various components, subsystems and service are started in the correct order.
- The lengths of the various buffers, message queues and similar data types should be set according to the system requirements and the available memory in the system.
- Most satellites periodically transmit a beacon message to indicate that the satellite is still functional. The contents of this message should be defined and implemented in the BEACON_PULSE() task.

Once all of the above steps have been carried out, the flight software should be fully integrated and configured for a specific mission.

Bibliography

[1] VGNet: Lecture CubeSats Veron Breda. *Amateur Radio Station PE0Sat.* Available at: `www.pe0sat.vgnet.nl/tag/lecture/`

[2] Ginet, G., Madden, D., Dichter, B. and Brautigam, D.: Energetic proton maps for the south atlantic anomaly. In: *Radiation Effects Data Workshop, 2007 IEEE*, vol. 0, pp. 1–8. July 2007.

[3] Pasetti, A. and Pree, W.: A Reusable Architecture for Satellite Control Software. *IEE/AIAA 19th Digital Avionics Systems Conference*, 2001.

[4] Barry, R.: *Using the FreeRTOS Real Rime Kernel. Cortex-M3 Edition.* Real Time Engineers Ltd, 2010.

[5] Botma, P.: CubeComputer Version 3: General Purpose Onboard Computer. Datasheet Revision D. 2013.

[6] George, F. and Billeter, S.: *Satellite Control Software TM & TC Packets Definition Recommendation.* Swiss Space Centre EPFL, November 2013.

[7] ECSS: Space engineering (ECSS-E-70-41A). January 2003.

[8] Swiss Space Center EPFL: SwissCube housekeeping parameters. 2009.

[9] Masson, L., Voumard, Y. and Richard, M.: QB50 Recommendation for Flight Software Implementation. 2014.

[10] Murphy, N.: Watchdog timers. *EE Times-India*, November 2000.

[11] Heidt, H., Puig-suari, P.J., Moore, P.A.S., Nakasuka, P.S. and Twiggs, P.R.J.: CubeSat: A new Generation of Picosatellite for Education and Industry Low-Cost Space Experimentation. *14TH Annual/USU Conference on Small Satellites, North Logan, Utah, USA*, 2000.

[12] von Karman Institute for Fluid Dynamics: QB50. `https://www.qb50.eu`.

[13] Wertz, J.R. and Larson, W.J.: *Space Mission Analysis and Design. Third Edition.* Microcosm Press, 1999.

[14] Klofas, B. and Anderson, J.: A Survey of CubeSat Communication Systems. *CubeSat Developers' Conference*, 2008.

[15] Klofas, B. and Leveque, K.: A Survey of CubeSat Communication System. *CubeSat Developers' Workshop*, April 2013.

[16] LaBel, K.A., Gates, M.M., Moran, A.K., Marshall, P.W., Seidleck, C.M. and Dale, C.J.: Commercial microelectronics technologies for applications in the satellite radiation environment.
Available at: `radhome.gsfc.nasa.gov/radhome/papers/aspen.htm.`

[17] Beck, R.P., Desai, S.R., Radigan, R.P. and Vroom, D.Q.: Software Reuse: A Competitive Advantage. *AT&T Bell Laboratories*, 1991.

[18] Johnson, R.E.: Frameworks = (Components + Patterns). *Communications of the ACM*, vol. 40, no. 10, pp. 39–42, 1997.

[19] Alonso, A., Salazar, E. and de la Puente, J.A.: Design of on-board software for an experimental satellite. 2012.

[20] dos Santos, W. and da Cunha, A.M.: Towards a pattern-based framework for satellite flight software using a model-driven approach. *2009 Brazilian Symposium on Aerospace Engineering and Applications*, 2009.

[21] Cosandier, B., Florian, G. and Choueiri, T.: Swisscube flight software architecture. *Swiss Space Centre EPFL*, August 2007.

[22] Kuo, T., Juanj, J., Tsai, Y., Tsai, Y. and Sheu, J.: Flight Software Development for a University Microsatellite. *Journal of Aeronautics, Astronautics and Aviation*, 2012.

[23] Holmstrom, D.E.: Software and Software Architecture for a Student Satellite. *Norwegian University of Science and Technology*, 2012.

[24] Renesas Electronics Corporation: General RTOS concepts. 2010.

[25] Schaffer, J. and Reid, S.: The Joy of Scheduling. *QNX Software Systems*, 2011.

[26] Martin, S: Introduction to Real - Time Operating Systems. *Quadros Systems Inc.*, 2013.

[27] Suleman, M.A., Mutlu, O., Qureshi, M.K. and Patt, Y.N.: Accelerating critical section execution with asymmetric multi-core architectures. *ACM SIGPLAN Notices*, vol. 44, no. 3, p. 253, February 2009.
Available at: `http://portal.acm.org/citation.cfm?doid=1508284.1508274`

[28] Torres-Pomales, W.: Software Fault Tolerance : A Tutorial. *Langley Research Center*, October 2000.

[29] Butler, R.W.: A Primer on Architectural Level Fault Tolerance. *Langley Research Center*, February 2008.

[30] Burns, A. and Wellings, A.: *Real-Time Systems and their Programming Languages*. Addison Wesley, 1990.

[31] Abbott, R.J.: Resourceful systems for fault tolerance, reliability, and safety. *ACM Computing Surveys*, vol. 22, no. 1, pp. 35–68, 1990.

[32] Conrad, J.M.: *Advanced Embedded Systems Concepts using the Renesas RX63N Microcontroller*. Renesas Electronics Corporation, 2014. ISBN 9781935772958.

[33] Silicon Labs: EFM32GG Reference Manual: Giant Gecko Series. 2013.

[34] Pouly, J. and Jouanneau, S.: Model-based Specification of the Flight Software of an Autonomous Satellite. 2012.

[35] Ganssle, J.: *The Art of Designing Embedded Systems*. 2000. ISBN 0750698691.

[36] Aroca, R.V. and Caurin, G.: A Real Time Operating Systems (RTOS) Comparison. 2007.

[37] Otava, L.: Analysis of Selected RTOS Characteristics. 2010.

[38] Moore, R.: How to Pick an RTOS. *Micro Digital, Inc*, 2012.

[39] Krasner, J.: RTOS Selection and Its Impact on Enhancing Time-To-Market and On-Time Design Outcomes. *Embedded Market Forecasters*, March 2007.

[40] UBM Tech Electronics: 2013 Embedded Market Study. *DESIGN WEST Embedded Systems Conference*, 2013.

[41] FreeRTOS: Features overview.
Available at: `www.freertos.org/FreeRTOS_Features.html`

[42] Consultative Committee for Space Data Systems: CCSDS Missions. 2014.
Available at: `public.ccsds.org/implementations/missions.aspx`

[43] Consultative Committee for Space Data Systems: CCSDS Data System Standards: Telecommand. June 2001.

[44] Consultative Committee for Space Data Systems: CCSDS Data System Standards: Telemetry. November 2000.

[45] Consultative Committee for Space Data Systems: CCSDS 301.0-B-2: Time Code Formats. 1990.

[46] Milne, G.W., Schoonwinkel, A., du Plessis, J.J., Mostert, S., Steyn, W.H., vd Westhuizen, K., vd Merwe, D.A., Grobler, H., Koekemoer, J.A. and Steenkamp, N.: SUNSAT - Launch and First Six Month's Orbital Performance. *13th Annual AIAA/USU Conference on Small Satellites*, 1999.

[47] Sandisk Corporation: SanDisk SD Card Product Family Version 2.2. 2007.

[48] Thatcher, J., Coughlin, T., Handy, J. and Ekker, N.: NAND Flash Solid State Storage for the Enterprise: An In-depth Look at Reliability. 2009.

[49] Ganssle, J.: Great watchdog timers for embedded systems. 2011.
Available at: `www.ganssle.com/watchdogs.htm`