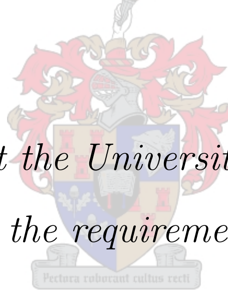
An eCos Based Flight Software for a Nanosatellite

by

Mr. Sifiso Selby Mthembu



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

Master of Science in Electrical and Electronics Engineering

Department of Electrical and Electronics Engineering
University of Stellenbosch
Private Bag X1, 7602 Matieland, South Africa

Supervisor: Mr. H. R. Gerber

March 2009

Declaration

By submitting this thesis electronically, I, Sifiso Selby Mthembu, 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.

Signed: ....

on this day: ....

Copyright © 2008 Stellenbosch University

All rights reserved.

Acknowlwdgement

I would like to thanks the following people and institutions.

- My God, for showing me the way.
- My Supervisor, for great support and motivation, and his wife for editing the thesis.
- Arno Barnard, you were always available when I am stuck in my project and needed a way forward.
- N. L. Steenkamp, your thesis laid a good background for my project.
- AM de Jager for translating the abstract into afrikaans, since it is the university requirement.
- My Family: from KwaZulu Natal (Kwamhlaba uyalingana) and also a home away from home - Western Cape (Stellenosch - Kayamandi). You were all good, and supportive.
- All my colleagues: Your company encouraged me, unity is what keeps us going.
- Financial support: National Research Foundation and Department of Science and Technology, I wouldn't have survived without your support.

Abstract

The nanosatellite is build-up of subsystems and payloads (defined as satellite nodes) connected together into the OBC using CAN bus as the main communication protocol. The flight software application is required to run within the eCos environment on the OBC to monitor and control satellite nodes.

The ground station must generate commands and send them to the satellite in space. The application is developed to validate, schedule and dispatch the commands to the satellite nodes at appropriate times. Each node manager, in the flight software, is required to execute the response messages from its respective satellite node. The housekeeping and error recovery data files are defined to convey useful information about satellite status to the user and can be downloaded to the ground station.

The flight software is developed using POSIX functions supported by eCos. Although it is not yet ready for real operation in space, the algorithm that can be used for full development is examined and approved.

Opsomming

’n Nanosatelliet bestaan uit ’n verskeidenheid substelsels en loonvrag (gedefinieer as satellietnodes) wat verbind is aan die aanboordrekenaar, deur van die CAN-bus as die hoof kommunikasieprotokol, gebruik te maak. Daar word van die vlugsagteware verwag om in die eCos omgewing uit te voer en die aanboordrekenaar te monitor en satellietnodes te beheer.

Die grondstasie moet opdragte genereer en na die satelliet in die ruimte stuur. Die programmatuur is ontwikkel om opdragte geldig te verklaar, te skeduleer en na die satellietnodes op die gepaste tyd te versend. Elke nodebestuurder in die vlugsagteware voer die reaksieboodskap van die onderskeidelike node uit. Die huishouding- en foutherstellings-leërs is verder van so ’n aard gedefinieer om bruikbare inligting oor die satelliet se toestand na die gebruiker oor te dra. Hierdie inligting kan na die grondstasie afgelaai word.

Die vlugsagteware is ontwikkel deur van POSIX funksies gebruik te maak wat deur eCos ondersteun word. Alhoewel die algoritme nog nie gereed is vir werklike gebruik in die ruimte nie, is dit ondersoek en goedgekeur.

Contents

Contents v

List of Figures x

List of Tables xii

List of Symbols xiii

1 Introduction 1

1.1 Satellite Overview 2

1.1.1 Subsystems And Payloads 3

1.2 Flight Software Objectives 4

1.3 Thesis Overview 6

2 Background Information 7

2.1 Satellite Software 7

2.2 Real-Time Systems 9

2.3 Real-Time Operating System 10
2.4 eCos 12
2.4.1 POSIX Compatibility Layer 13
2.5 Software Reliability 15
2.5.1 Fault Tolerance 16
2.5.2 Error Recovery Mechanism 17
2.6 CAN bus protocol 19
2.6.1 Identifier Field 19
2.6.2 Protocol Supported Message Types 21

3 Flight Software Design 23

3.1 Flight Software Description 23
3.1.0.1 Ground Station Interactions 23
3.1.0.2 Command And Node Managers 24
3.1.0.3 Autonomous Operation 25
3.1.0.4 Retain Data For Ground Station 26
3.2 Flight Software Structure 26
3.2.1 Thread Manager 28
3.3 Command Manager Design 29
3.3.1 Diary File Design 30

3.3.1.1 Command Design 30

3.3.1.2 Groups of Diary Files 32

3.3.2 Command Validating, Scheduling And Dispatching 34

3.3.2.1 Command Validation 34

3.3.2.2 Command Scheduling 35

3.3.2.3 Command Dispatching 38

3.4 Software Development Environment 40

4 Command Manager 41

4.1 Command Manager Processing 41

4.1.1 Processing A Message Received From Satellite Node 43

4.1.2 Processing After Node Manager Has Executed Message 44

4.1.2.1 Housekeeping Data File 45

4.1.2.2 Error Recovery Data File 46

5 Subsystem And Payload Manager 48

5.1 Node Manager Structure 49

5.2 Message Validation and Execution 52

5.2.1 Sample For Message Execution 55

5.3 Updating Flags 56

5.4 Monitoring The NSV 60

5.5 ERP 62

6 Flight Software Testing And Results 64

6.1 Implementation And Testing 64

6.1.1 Flight Software And Testing Modules 65

6.1.1.1 Flight Software 65

6.1.1.2 CAN Module (CAN bus emulator) 66

6.1.1.3 Simulated Satellite Nodes 67

6.1.2 Flight Software Testing 68

6.2 Results And Discussion 70

6.2.1 Command Execution 70

6.2.2 Handling Response Messages 71

6.3 Software Time Response 73

7 Conclusion And Recommendations 76

7.1 Conclusion 76

7.2 Recommendations 78

A Protocol Supported Message Types 82

B Linked List 85

B.1 Inserting A Node In A Linked List 86

B.2 Deleting A Node In A Linked List 87

C Generated Data Files 89

C.1 Sample for error log data file 89

C.2 Sample for Housekeeping data file 90

List of Figures

1.1 Satellite Overview 3

2.1 Shared Memory Model [7] 14

2.2 PThread Creation And Termination Against Time [7] 15

2.3 CAN Bus Data Frame [3] 19

2.4 29-Bit Identifier Allocation [1] 20

3.1 Flight Software Design 27

3.2 Commands Scheduling Using Linked List 36

4.1 Command Manager Process Flowchart 42

5.1 Node Manager Flowchart 50

5.2 Evaluate Received Message (Extract from Figure 5.1) 53

5.3 Node Status Vector (NSV) Indicating Voltage and Current Flags 54

5.4 Flags Updates Flowchart (Updates Flags refer to Figure 5.1) 57

5.5 NSV and ERP_NSV 58

5.6 Monitoring The NSV (Extracted From Figure 5.1) 61

6.1 Flight Software Testing 65

B.1 Inserting a node in order in a list [8] 86

B.2 Deleting a node from a list [8] 88

List of Tables

3.1 Command Fields 30

3.2 Data Fields For Invalid Message File 35

3.3 Command schedulers 37

3.4 Command Fields Sent To Satellite Nodes (CAN Message) 38

4.1 Data Fields For Housekeeping Data File 45

4.2 Data Fields for Error Recovery File 47

5.1 Telemetry Response Messages for Voltage Channel 55

5.2 Phases For Message Execution 56

6.1 Sample for Commands used to Generate Diary Files 68

6.2 Definitions of Message Identifier, DLC and Data Fields From Table 6.1 68

A.1 The Protocol Supported Message Types [1] 84

List of Symbols

ACK	Acknowledge
ADCS	Attitude Determination Control Systems
AO	Autonomous Operation
API	Application Programming Interface
CAN	Controller Area Network
DLC	Data Length Code
eCos	Embedded Configurable Operating System
ERP	Error Recovery Process
ERP_NSV	Error Recovery Process for Node Status Vector
ID	Identifier
IO or I/O	Input and Output
LEO	Low Earth Orbit
LSB	Least Significant Bit
MMQ	Manager Message Queue

MMU	Mass Memory Unit
MSB	Most Significant Bit
NSV	Node Status Vector
OBC	Onboard Computer
OS	Operating System
PC	Personal Computer
POSIX	Portable Operating System Interface
SPI	Serial Peripheral Interface
SUNSAT	Stellenboch University Satellite
RX	Receive
RTOS	Real-Time Operating System
TC_ACK	Telecommand Acknowledgement
TC_Req	Telemetry Request
TLM	Telemetry
TLM_ACK	Telemetry Acknowledgement
TLM_Req	Telemetry Request
TX	Transmit
VSOC	voltage state of charge

Chapter 1

Introduction

Since the development of the Stellenbosch University Satellite (SUNSAT), a satellite program has been provided for postgraduate students to develop subsystems and payloads for a nanosatellite. This is intended to introduce students into the satellite program while they are still at tertiary level, and also to constitute the university nanosatellite by integrating developed parts into a complete nanosatellite for a defined satellite mission statement.

The nanosatellite has a low design and launch cost compared to commercial satellites, since it is expected to weight less than 10 kg and requires less energy to launch it into Low Earth Orbit (LEO).

The aim of the satellite program is to develop the nanosatellite for earth observation, scientific research and other objectives that are yet to be defined. Flight software is among the projects defined to develop the nanosatellite, and this thesis describes the design, implementation and the outcomes during the testing of the flight software.

The flight software is required to communicate with every developed subsystem and payload. Each and every satellite node runs independently but they are all connected to the Onboard Computer (OBC) using a Control Area Network (CAN) bus. The embedded configurable operating system (eCos) is chosen as the operating system

(OS) for the OBC. Another student had the responsibility to configure eCos in order to run in the OBC.

The flight software application is developed using the C programming language since it is supported by eCos. The final structure of the flight software is made up of a single process with multiple threads, due to eCos constraints. The main thread is the command manager which is responsible for reading and sending the monitoring and control messages of the flight software.

1.1 Satellite Overview

It is very important to analyze the full integration of the nanosatellite in order to develop effective flight software. The satellite is made up of subsystems and payloads, defined as satellite nodes, connected together into the OBC using a CAN bus as the main communication protocol. Direct communication between certain satellites nodes is also supported, such as the Serial Peripheral Interface (SPI) used by the OBC to communicate with the Mass Memory Unit (MMU). This is shown in Figure 1.1.

The OBC is the core of the nanosatellite system and it is the host for the eCos which is required to execute the flight software. eCos provides the functions for the flight software to achieve real time requirements. Each satellite node is a combination of the hardware and software that interact with the outside world to execute the required satellite service. The satellite node software operates autonomously, but it requires the flight software to set control modes.

Payloads interact with the outside world to accomplish the satellite mission. Therefore they are designed according to the satellite mission, and they serve the primary objectives for which the satellite is placed into a desired orbit. The subsystems are designed to keep the payloads healthy, operating effectively and remain pointed to the right direction. The services that can be executed by the satellite nodes are discussed in the following section.

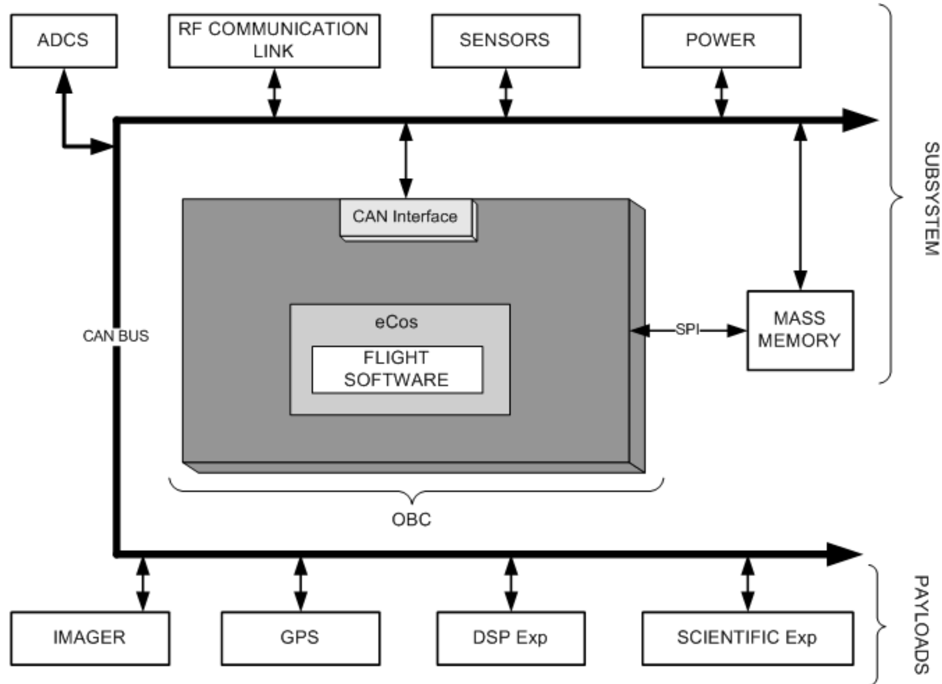


Figure 1.1: Satellite Overview

1.1.1 Subsystems And Payloads

The communication payload provides the means for transferring information, such as the broadcasting of television signals or transmission of navigation messages, between the satellite and the ground station(s). It also provides modulation and demodulation of digital signals which enables the transfer of data. This is used for sending the acquired satellite service data to the ground station or relaying signals from one ground station to another. The satellite also uses this payload for receiving control commands from the ground station to achieve the satellite's mission.

The remote sensing payload is for observing an object without directly contacting the object, such as imaging the earth's surface and stars. It uses various sensors to identify the target object and then uses cameras to take images that can be downloaded to the ground station for image processing and scientific experiments.

The ADCS subsystem is responsible for maintaining and controlling the orientation of the satellite in orbit. It uses sensors to get the orientation of the satellite and then uses

actuators to position the satellite in the right direction; such as antennas pointing to earth for communication with ground station and solar panels directed to the sun for recharging the batteries. Thrusters are used to keep the satellite in the correct orbital position, since it deviates slowly from its original orbit as it orbits the earth.

The power subsystem consists of solar panels and backup batteries that provide power when the satellite passes into the earth's shadow. This is used to supply the entire satellite, and it is crucial that satellite nodes are switched off when they are not used to conserve the power/energy.

The MMU provides the storage capacity for satellite data, such as payload data often produced by the imager payload. When the satellite passes over the ground station, the payload data can be transferred to the ground station, and then free the memory so that it can become available for new data.

1.2 Flight Software Objectives

The primary objective of this project is to develop flight software, a real time software application that will monitor and control a nanosatellite. Monitoring means to regularly evaluate the satellite's condition and ensuring that the satellite is in a safe operating mode. Controlling is to manage the satellite to deliver its required service to the ground station, by controlling the sequence of commands executed by satellite nodes. The flight software must be compatible with the eCos environment, as it has been chosen to be the operating system for the nanosatellite.

The satellite operates in space, hundreds of kilometres above the earth, therefore remote communication is used between the satellite and the ground station. The flight software must enable the ground station to control the satellite. The ground station must generate commands, and transmit them to the satellite so that the flight software can execute them. This means that the flight software must have the ability handle and distribute the commands to the respective satellite nodes.

The satellite is orbiting the earth at a very high speed, for instance at LEO a satellite's speed is approximately 8 km/s [5]. Therefore the real-time requirement is a major factor to consider for the satellite's performance. Suppose a satellite has to take an image of a chosen place or object, the camera should be accurate and quick enough to take a picture while the object is within the camera view area, otherwise the camera will miss the target view. This implies that the flight software must meet the real-time application requirements, by distributing commands at appropriate times.

Housekeeping means monitoring the status data of the satellites systems. The flight software must monitor the health of the satellite regularly while also checking for any system errors. Once an error is detected, the flight software must identify the cause of the error and then provide means to eradicate or minimise the effect of an error on the system. If the error is ignored, it may eventually disable the entire satellite system. The status data and system errors must be recorded by the flight software. The user at the ground station can analyse the recorded data and generate the commands for the flight software that will prevent (if possible) or minimise the chances for the system to experience the same errors again.

Since the satellite is not within a communication range of the ground station most of the time, the flight software should be able to control the satellite autonomously. This means that the satellite must be able to function on its own most of the time, especially to ensure that the satellite is in a safe operating mode. The flight software should always execute the commands that do not threaten the safety of the satellite. Every command must be evaluated to ensure that it is valid and safe to execute it. Otherwise it must not be executed, and this should also be recorded so that it can be evaluated at the ground station to avoid the system from experiencing the same problem.

1.3 Thesis Overview

The rest of the chapters provided in this thesis include a full description of the project, during the course of designing, implementing and testing the flight software. A brief overview of each chapter is given below:

- Chapter 2 - presents the subjects that were covered before designing the flight software. This describes the environment in which the flight software will be running, and also defines the principles that can be employed by the flight software in order to manage the satellite system effectively.
- Chapter 3 - describes the approach used to construct the final design, and also provides discussion about modules used in the design.
- Chapter 4 - provides the detail design and the implementation of the Command manager as the main module for the flight software.
- Chapter 5 - provides a general algorithm and principles that can be used to implement any node manager in the flight software. It has been noted that every satellite node is designed differently; hence the node manager should be implemented according to its respective satellite node.
- Chapter 6 - describes the performance observed during the course of developing and testing the flight software.
- Chapter 7 - provides the conclusion based on the performance of the flight software and recommendations that can be considered for upgrading the flight software.

Chapter 2

Background Information

For every project a candidate embarks on, the first approach is to enquire about information related to the project which is defined as background information. The purpose is to improve an existing project, and to follow the guidelines and specifications as the candidate goes into the designing and implementation phase of the project. Not all acquired background information is eventually used, and this chapter provides the information acquired and considered for designing and implementing the flight software application.

This chapter covers the following concepts: The general satellite software, the real-time requirements of the flight software and the OS of the nanosatellite. The mechanism required to implement a reliable software application and the message structure supported by CAN bus protocol is also discussed.

2.1 Satellite Software

The satellite is made up of various subsystems and payloads, and they are designed for various purposes. As a result every satellite node has unique software developed to support its hardware. Even though they are independent of one another, there is a need for general software that will monitor all the satellite nodes to ensure that

they remain in a proper working order while executing services. The general software functions required by most satellites are explained as follows [2]:

- **Navigation** - This function uses ADCS subsystems to monitor and control the orientation of the satellite. This is conducted by processing information from sensors and then controls the actuators to maintain the desired satellite orientation. This is conducted for various reasons as follows: when the satellite is drifting away from the desired orbit; exposing solar panels to sun rays for recharging batteries; and for controlling the satellite to point towards the earth for communication with the ground station.

- **Housekeeping and Health Monitoring** - This refers to the management and monitoring of the satellite, and this is essential for keeping the satellite functioning in a safe mode. This is conducted by analyzing the status data generated by a satellites nodes and then detect and correct the system from errors. The status data is recorded as housekeeping data, for the user at the ground station, once it is downloaded, to observe the satellite behaviour.

- **Subsystem and Payload Management** - These functions are required to manage a particular hardware resource referred to as satellite node. Since every satellite node is designed to serve a unique purpose, each should have a manager module as a part of the flight software. The node manager must have the ability to interpret the status of the respective satellite node and then control its operation mode.

- **Command Processing** - This is responsible for handling the commands generated by the ground station to control the satellite nodes. This validates commands based on their contents and according to the current state of the satellite. Valid commands are scheduled in order to control the execution of commands to the respective satellite nodes. An appropriate time for execution depends on a command and the satellite provides the various services at different times.

- **Communications** - This provides the means for the ground station to upload new commands to the satellites, and download housekeeping data and payload

data. These commands can be used for acquiring the new satellites services or for monitoring purposes using housekeeping data at the ground station.

The Navigation and Communications functions are partially covered by the flight software; the main software is installed in their respective satellite nodes, ADCS and communication subsystems. The rest of the functions should be covered by the flight software. The flight software must monitor and control the satellites nodes, by sending commands regularly for requesting status data while also sending control commands according to the current state of the satellite, which results in a real-time system.

2.2 Real-Time Systems

The satellite system orbits the earth and changes its behaviour (such as orientation and solar panels direction) as a function of time, and this is considered as a real-time system. The satellite computer continuously analyses the status data of the system and adjusts the satellites mode, if undesired behaviour is observed. The usefulness of this control mode does not only depend on the correctness of the resulting action, but also the time in which it was conducted. [2]

The real-time system defines the correctness of the data as it does not only depend on the computation but also on the time at which the results are produced. This means that the system should react to the respective situation within the defined time interval of that system environment, and this can be defined as a deadline. The usefulness of a result after its deadline has passed, classifies a real-time system into two categories: hard and soft real-time systems. [14]

The hard real-time system is defined in a system where the missing of a deadline results in system failure, whereas in a soft real-time system it degrades the system performance [2]. The soft real-time system indicates that the system may continue to function even though the overall system will not be highly effective. This may still be controlled (depending on the system environment) to achieve the desired system performance.

Such a system is defined to be fault tolerant and this is discussed in section 2.5.1. The hard real-time indicates that the system will never function or recover properly from that situation, and this may eventually disable the entire system.

The satellite is defined as an embedded system, since it is composed of satellite nodes which run independently but they are all connected to, and controlled by, the OBC. The flight software must conduct housekeeping while it is controlling the satellite to execute the required satellite service. Even though the software running on the OBC must meet real-time requirements, this application depends on the OS used by the satellite system.

2.3 Real-Time Operating System

An OS is the program that acts as an intermediary between a user of the computer and the computer hardware. The purpose of the OS is to provide an environment in which a user can execute programs in a convenient and efficient manner [9]. The real-time operating system (RTOS) is responsible for executing the real-time software, and enables concurrent execution [14].

The OS ensures correct and safe program execution of the computer systems. The program immediately ends its execution if the error is detected, meaning the OS must be aware of possible system errors. The OS also provides file system manipulation such as, create and delete file, read and write into a file.

There are various types of OS, and each has unique features which are suitable for different embedded systems. The basic functions required in the RTOS are: task dispatching, task scheduling and inter-task communication. [14]

- **Task Dispatching** - It is the ability to transfer execution from one process to another, in a multitasking environment. A process is given a fixed (at compile time) point where the execution initiates and terminates and task dispatching can be conducted at those fixed points, only if the process is still executing.

- **Task Scheduling** - This chooses the next task for execution based on the set of available tasks and once a task is chosen the dispatcher is called to execute it. A task is said to be available when it needs to be executed. Tasks are scheduled according to their priority level; therefore a task with a hard deadline must be given a higher priority. This can be controlled during the software development phase, and it can be fixed or changed dynamically while the application is running.

- **Intertask Communication** - This is the ability to successfully exchange data between tasks, since tasks are independent of each other. This is based on shared variables and message passing. Shared variables are accessed by more than one process; reading and writing into these variables is therefore allowed. Synchronisation is necessary for shared variables to avoid the case where one process is reading the information while another process is busy changing the same shared memory. This is defined as a critical section. Mutual exclusion is provided to allow one process at a time to access the shared variables, and this should be employed only during the critical section otherwise it will slow the execution of the processes.

  Message passing involves the exchange of data between two processes by means of a message, and this can be asynchronous or synchronous communication. For asynchronous communication, the process sends a message and immediately continues with its execution regardless of whether the receiving process has read the message or not. Whereas synchronous communication means the sending process waits until a reply is obtained from the receiving process.

These basic functions mentioned for RTOS are also necessary for flight software. The OS chosen for the OBC is eCos, as stated in section 1.1, and the following section provides more details about it.

2.4 eCos

eCos is an open source runtime system. This provides the platform to freely develop and distribute applications based on eCos. eCos supports applications with a single process and multiple threads, implemented either in C, C++ or Assembly programming language. It also supports the basic functions of the RTOS as discussed in the previous section, although it uses threads instead of tasks and it only supports a single process application with multiple threads. [4]

The eCos support two main packages, kernel and Portable Operating System Interface (POSIX), and each provides the major functionality needed for developing multi-threaded applications as stated below [4]:

- The ability to create new threads in the system, either during startup or when the system is already running.
- Control of the various threads execution in the system, such as dynamic scheduling during run time according to their priority levels.
- A range of synchronisation mechanisms, allowing threads to interact and share data safely.
- Integration with the system's support for interrupts and exceptions

The kernel functionality can be used in one of two ways. The kernel provides its own C Application Programming Interface (API), with functions like *cyg_thread_create*() and *cyg_mutex_lock*(). These can be called directly from application code or from other packages. Alternatively there are a number of packages which provide compatibility with existing API's, for example POSIX threads. This allows application codes to call standard functions such as *pthread_create*(), and those functions are implemented using the basic functionality provided by the eCos kernel. By using compatibility packages in an eCos application, this provides the means to develop an application that can run in various OS. The following section provides more details about POSIX packages supported by eCos. [4]

2.4.1 POSIX Compatibility Layer

eCos support divides POSIX into two packages called the POSIX Compatibility Layer and POSIX File IO Compatibility Layer, but not every function in those packages are supported by eCos. The POSIX Compatibility Layer package provides support for threads, signals, synchronisation, timers, and message queues. The POSIX File input and output (IO) Compatibility Layer package provides support for file manipulation and access to device input and output (I/O). Reference [4], indicates POSIX functions and data types that are supported by eCos.

A standardised C language threads programming interface has been specified by the POSIX package and implementations that adhere to this standard are referred to as POSIX threads, or Pthreads.

A thread is defined as an independent stream of instructions that can be scheduled to run independently from its main program, by the OS. It is also defined as the light-weight process [9]. A multi-threaded program is when the developed application consists of more than a single thread scheduled to run simultaneously or independently by the OS, and this is used to implement parallelism programming.

Communication between threads of the same process is achieved by a shared memory and message passing model. A message passing model is defined by POSIX package as a message queue. Message queue provides an asynchronous communications protocol, where messages can be placed onto the queue at any time and stored until the recipient retrieves them. The shared memory model is a section of the memory that can be accessed by more than one thread.

The Figure 2.1 illustrates a single process with multiple threads. Threads have their own private data and access to the shared memory allocated for the process. Programmers are responsible for synchronising (protecting) access to globally shared data. If the application code does not employ some sort of synchronisation constructs to prevent data corruption, then it is not a thread-safe program. Mutexes provide the synchronisation for controlling thread access to the shared memory.

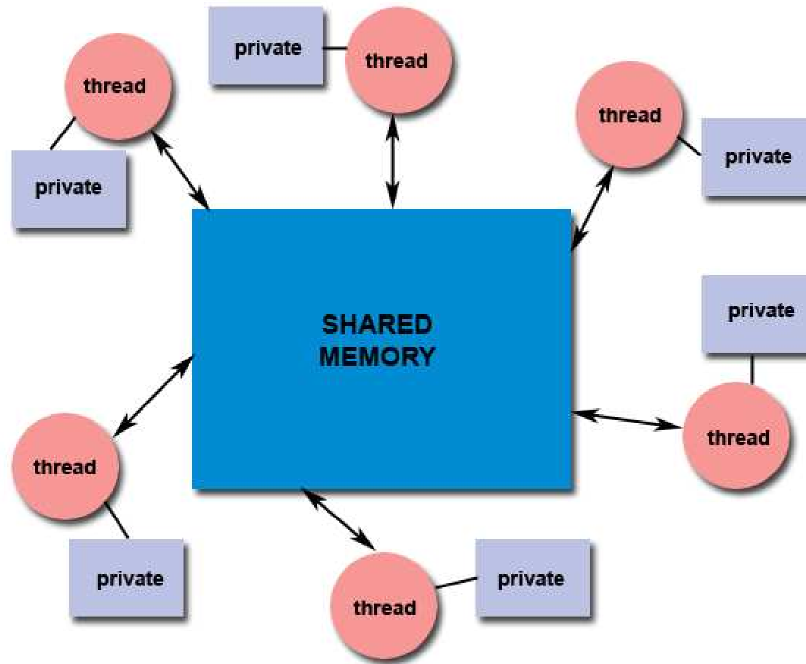


Figure 2.1: Shared Memory Model [7]

The Figure 2.2 illustrates a program that initially creates a single thread, and the rest of the threads are created at different times during process execution. This also indicates the parallelism where more than one thread is created and executed at the same time. The figure also implies that once a thread is created (parent thread), it can create another thread (child thread), and the parent thread can end its execution without interfering with the execution of the child thread. The maximum number of threads that may be created by a process is set to be 64 threads. This is only applicable to POSIX standards [4].

There are several ways in which a Pthread may be terminated. The thread can cancel itself by calling a *pthread_exit*() routine. The thread can also be canceled by another thread using the *pthread_cancel*() routine, and this uses thread identifier as a argument to identify exact threads since there can be multiples of them. Typically, the *pthread_exit*() routine is called after a thread has completed its work and is no longer required to exist. If the main process finishes before the threads it has created, and exits with *pthread_exit*(), the other threads will continue to execute [7].

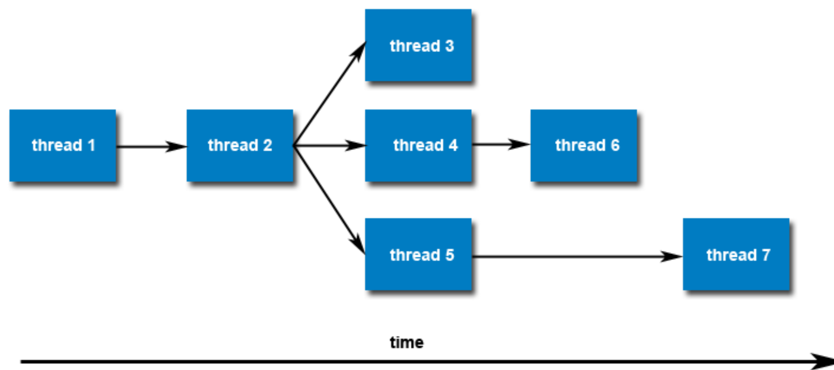


Figure 2.2: PThread Creation And Termination Against Time [7]

2.5 Software Reliability

The software does not deteriorate with use, unlike the hardware component; therefore the reliability of the software can be verified during the course of its development. If the software is defined as not reliable, program testing will indicate the location of faults which can then be corrected. Nevertheless, for embedded systems it is impossible to ensure that the software is fault free; since it is only the errors that were detected during testing that were corrected. Therefore the techniques for detecting and correcting errors, while the system is already operating, must be developed as part of the software application. [14]

Software reliability is defined, according to Okumoto, as the probability of failure-free operation of a computer program in a specified environment for a specified time [12]. This means that the software will operate without a failure for a specified period of time [13]. If the software experiences an error, then it should find the means to continue executing by going to the safe mode or to shut down and reboot, and this forms part of the fault tolerance as discussed in the following section.

2.5.1 Fault Tolerance

It is not possible for a human being to fly to the satellite while in space. This brings the possibility of satellite physical maintenance to zero, especially for a university nanosatellite. The common satellite system maintenance can be conducted through remote control by enabling the ground station to issue commands to the satellite. Hence the flight software should apply the fault tolerance mechanism since the satellite is not within communication range of the ground station the majority of the time.

The fault tolerance is, according to Steenkamp, when the autonomous operation of a satellite does not only imply the ability to automatically perform normal operating functions, but also the ability to manage the occurrence of anomalies [2]. This means that the system has to keep functioning even though it is experiencing unpredicted operations. This can be achieved either by trying to minimise the effect of such a behaviour to the system or return the system to the normal mode.

There are three different levels of fault tolerance, according to Burns, that can be provided by the system as follows [14]:

- **Full fault tolerance** - The system continues to operate normally in the presence of faults, over a certain period.
- **Graceful degradation (or fail soft)** - The system continues to operate in the presence of errors, while the overall system performance is degrading as it conducts error recovery operation. This is necessary in most applications, since full fault tolerance is achievable for a limited period.
- **Fail safe** - The system maintains its integrity while accepting a temporary halt in its normal operation. This implies to shutdown the system in a safe mode, and this mechanism limits the damage caused by the failure.

A fault can result from physical breakdown of a system component, error in the system specification or software programming mistake. Faults can either be transient or permanent. A transient fault is defined as a fault that exists for a certain period and

then disappears without any mechanism of repairing it. In the space environment, faults resulting from interference and radiation are defined as transient faults, since this effect will change as the satellite orbits the earth. A permanent fault exists until the system employs the repair mechanism and this is usually caused by the failure of the component or software programming mistake. [11]

Faults leads to an error, and is defined as when the system enters an invalid or unpredicted state. An error is also transient or permanent. If the states and data structures are initialised before the function performs its routine, then the fault can be transient since it will be experienced once, then the next time the function is called it will be initialised with default values. A transient fault can cause permanent fault, if the result from the function, where error is detected, is influencing other functions, even after the fault has disappeared from the originated function. [2]

Failure tolerance can be incorporated into the system design, by having the alternative hardware resource which will be used only if the primary hardware resource has a fault. This is employed in the nanosatelite where the CAN bus is implemented into two divisions. When the main CAN bus is not functioning properly then the secondary CAN bus is used automatically as the main bus for the system, and also the main CAN bus instead of the secondary CAN bus. This strengthens the communication between the satellite nodes and the OBC, otherwise the entire nanosatellite system might stop functioning if the CAN bus is no longer functioning. Fault tolerance can also be implemented by means of error recovery as discussed in the following section.

2.5.2 Error Recovery Mechanism

The system may experience a failure due to a detected error, hence it could deviate from its normal execution and then enter an unknown state. Once an error has been detected, error recovery must be initiated. This is intended to transform an erroneous system state into one in which the system can continue its normal operation, even though it may experience degraded performance. [14]

The error can be handled by exiting either the function routine or system software execution. Some errors can be predicted during the design of the system, and those errors can have a specified function to execute when they are detected, to minimise the effect of the error or remove the error from the system. This can only delay the normal execution of the application without terminating the software execution. There are two ways of error recovery namely, forward and backward recovery.

The forward error recovery means to enable the system to continue operating from erroneous states by making selective corrections to the system state. This is also making the system safe, which may be damaged because of failure. This is system specific and depends on accurate predictions of the location and cause of the errors. If more than one process is involved in providing the service when the error occurred; a shutdown of the system may also be necessary during error recovery because it is not easy to trace the source of the error since processes share results.

Backward error recovery is based on restoring the system to a safe state, previous to that in which the error occurred. This means that the system executes an alternative function instead of continuing with its normal operation, and this provides the same function with fault tolerance. The point to which the system is restored is called the recovery point, and this can be attained when the system state is recorded during the course of system operation.

State restoration is simple to implement since it does not require the cause of the fault to clear the erroneous state, but the existence of an error. Therefore this can be useful for unpredicted faults. The backward error recovery mechanism can be time consuming in terms of program execution, which may prevent its use in real-time applications. This can be experienced when one process using backward error recovery while it is also interacting with another process and this may result in both of them restoring their state, since the result from one process may be used by another. [14]

It is common for embedded systems to eventually employ both, forward and backward, recovery mechanisms since it has to recover from errors as quickly as possible in order to continue with normal system operation.

2.6 CAN bus protocol

The flight software application will be running within the eCos environment on the OBC as intended to communicate with satellite nodes. CAN bus protocol was designed for the nanosatellite to provide a reliable communication link between the OBC and satellite nodes. The flight software commands must be carried by the CAN bus into the desired satellite node(s). This section provides a brief summary of the CAN bus protocol chosen for the nanosatellite [1].

Every object connected to the CAN bus is defined as a node. Each node has the ability to send messages into, and read messages from, the CAN bus. The Figure 2.3 shows the full format of the frame that can be transmitted by the CAN bus. There are three main fields available in the CAN bus data frame (identifier, data length code (DLC) and data fields), that were configured during the design of the CAN bus protocol used for nanosatellite [1]. The other fields are necessary for improving network reliability.

Figure 2.3: CAN Bus Data Frame [3]

The message types transfer various sizes of data over the network, while the network can only support a maximum of 8 bytes per message. The DLC identifies the number of bytes used in the data field, and the data field contains data fragmented in number of packets (1 byte each) as identified by DLC. The following section provides more details about the identifier field.

2.6.1 Identifier Field

The identifier field carries the message identification information over the network, and every message of different purpose has a unique identifier. It also determines the

manner in which each message is handled by the protocol. CAN bus protocol can only carry one message at a time over the network. If there is more than one node requesting to transmit the message, only the message identifier of less numerical value will be transmitted first. The other message waits until the bus is free and then it is automatically re-transmitted in the next bus cycle, unless another higher priority message has requested to transmit as well. The protocol is based on CAN 2.0B using 29 bits identifier. The distribution of the identifier across the network is divided into four fields as shown in Figure 2.4. [1]

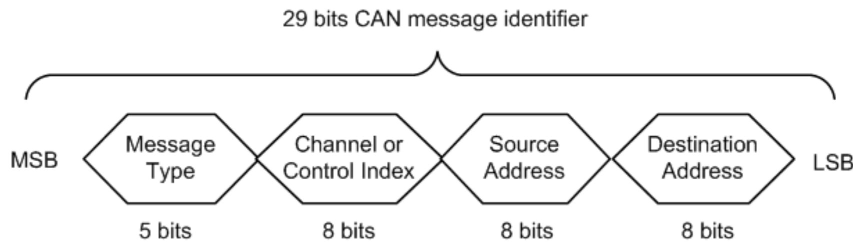


Figure 2.4: 29-Bit Identifier Allocation [1]

- **Message Type Field** - This field identifies the type of the message sent on the network. There are 32 different message types that network can handle, since the field size is 5 bits wide. Message type 0 is given the highest priority over the CAN bus network. There are already 14 defined message types, according to Khumalo, and they are stated in section 2.6.2. [1]

- **Channel or Control Field** - This is made according to the message type sent on the network. For telemetry and telecommand message types, this field identifies the channel in which the message will be categorised at the receiver node. File and data transfer use this field as the control or the command field that specifies what must be done with the data that is read or written to the specific memory address.

- **Source and Destination Address Fields** - Each and every node connected to the network will be allocated a unique address, and this address must be used as a source address when the node is transmitting a message and as a destination

address for receptions. The node will have an acceptance filter which is used to identify the frames that are addressed to it, and this can be adjusted to accept a message with a different address. This is used for broadcast messages so that every node can read the message even if the destination address does not correspond to the node address. Address 0 is reserved for broadcast messages.

2.6.2 Protocol Supported Message Types

This section states the message types that are already supported by the CAN bus protocol. All message types except for debug and the time synchronisation messages will be acknowledged and this acknowledgment will be a message type of its own. The following message types are already defined by the CAN bus protocol and they are given according to their priority level. Time synchronisation is given 0 in a message type field; hence it has the highest priority. Refer to Appendix A for more information.

1. Time Synchronization
2. Telecommand Request
3. Telecommand Response
4. Telecommand Not Acknowledgement
5. Telemetry Request
6. Telemetry Response
7. Telemetry Not Acknowledgement
8. Unsolicited Telemetry Request
9. File Header Transfer
10. File Header Transfer Acknowledgement
11. File Data Transfer

12. File Data Transfer Acknowledgement
13. File Data Transfer Not Acknowledgement
14. Debug Message

Chapter 3

Flight Software Design

This chapter provide details about the factors that were considered before designing the flight software, and discussion about the flight software design. This also describes the environment in which the flight software was developed.

3.1 Flight Software Description

These following sections provide the information considered during the design of the flight software.

3.1.0.1 Ground Station Interactions

A user at the ground station must have the ability to generate and upload commands, in a form of a diary file, onto the satellite in space. Commands should be simple enough for the flight software to interpret and execute them according to their description. They are necessary for monitoring and controlling the satellite to execute the required satellite service. These commands must be composed of the following fields as discussed below.

- The command fields (identifier, DLC and data fields) employed by CAN bus protocol, as defined in section 2.6, are considered.
- Time reference field to record the time in which the command should be executed by the flight software. The repeat value and time interval fields, are necessary for commands that need to be executed more than once, to record the repetition needed for executing the command and the next execution time after the command is executed.

The flight software must restructure commands, once they are read from diary files in order to be compatible with the frame format of the CAN bus which carries them to reach the desired satellite nodes.

3.1.0.2 Command And Node Managers

The command manager must validate, schedule and dispatch each command to the respective satellite node via the CAN bus. The command validation can be achieved by verifying that the command is composed of all the required fields, and ensures that the command is supported by the flight software as discussed in subsection 3.1.0.1. Scheduling must be conducted by grouping or linking the commands according to their priority levels, since some commands are time dependent while others are message type dependent as discussed in section 2.6. Dispatching is the ability to send commands to the satellite nodes in a defined sequence at an appropriate time. The command manager should also be able to read the response messages, from satellite nodes, and relay them to the destined node manager to evaluate the message.

Each and every satellite node is designed differently, and has it's supporting software application running on it, to serve a unique purpose. Therefore it is necessary for each satellite node to have a respective software manager module implemented in the flight software, and it is called a node manager. Every node manager should represent the status of the corresponding satellite node by recording information resulted from every command executed by the respective satellite node. The unique identifier or address

for the satellite node and node manager is necessary for the flight software and user at the ground station to differentiate them easily and control them accordingly.

3.1.0.3 Autonomous Operation

The communication time between the satellite and ground station is very limited, for instance SUNSAT was communicating for less than 10 minutes per over pass, with only 5 over pass per day [2]. The flight software application needs to keep the satellite stabilised and functioning according to the predefined specification while it is in a safe operation mode. Autonomous operation (AO) is the ability for the satellite to operate effectively without ground station intervention, especially when the satellite is not within the communication range of the ground station. Therefore AO application is needed in the flight software.

The AO can be implemented by regularly executing commands which request the status of each satellite node, such as telemetry commands described in Appendix A, in order for the flight software to evaluate the satellite status. When ever the satellite status data deviate from its nominal range, an error must be detected, and the node manager should be able to generate commands in response to the respective satellite node. Those commands can either remove or minimise the effect of the detected error, to keep the satellite node in a safe mode. This may result, if the error is ignored, in affecting the performance of other satellite nodes and eventually affect the entire satellite system.

Suppose the satellite power supply from the power subsystem batteries reduces to certain extent, therefore some of the satellite nodes need to be switched off until the batteries are recharged enough to keep the satellite nodes functioning at full capacity. If this is ignored, then the entire satellite system may run out of power and this can eventually end the satellite's life in space. Therefore the AO is necessary for such a situation, and quick recovery is necessary without waiting for a user at the ground station.

3.1.0.4 Retain Data For Ground Station

Satellites nodes are expected to generate the various types of data that can be used by the user at the ground station, mainly for processing data as the part of the satellite service and also analyzing the behaviour of the satellite. Payload and housekeeping data are the main data types that can be generated randomly [11]. The payload data file provide the service of the satellite, such as images and scientific experiments data, and this is expected to consume considerable amounts of memory. Therefore after the ground station has downloaded the file successfully, the satellite must delete the file immediately so that the system does not run out of memory.

The housekeeping data is the information about the satellite and its health and safety. This file should be updated regularly by the flight software using status data received from satellite nodes. The file size will always be growing and eventually consume a all the memory; therefore the maximum file size should be defined. It may happen that the ground station takes a long time before downloading housekeeping data and the defined maximum file size is eventually reached, then an exception should be made to record the current satellite status data. Therefore the data file must be deleted; either when the ground station has downloaded the file or else if a defined maximum file size has been reached, to free the memory for incoming data.

Users at the ground station may also use the housekeeping data to evaluate the behaviour of satellite nodes during the collection of the payload data [11]. Therefore it is necessary to download both of these data files, if possible, during a single pass.

3.2 Flight Software Structure

The flight software application was developed as shown in Figure 3.1. The figure also represents the satellite nodes (subsystems and payloads) as they are the main target of the flight software. This section provides a brief description about the role of each and every flight software module indicated in the figure.

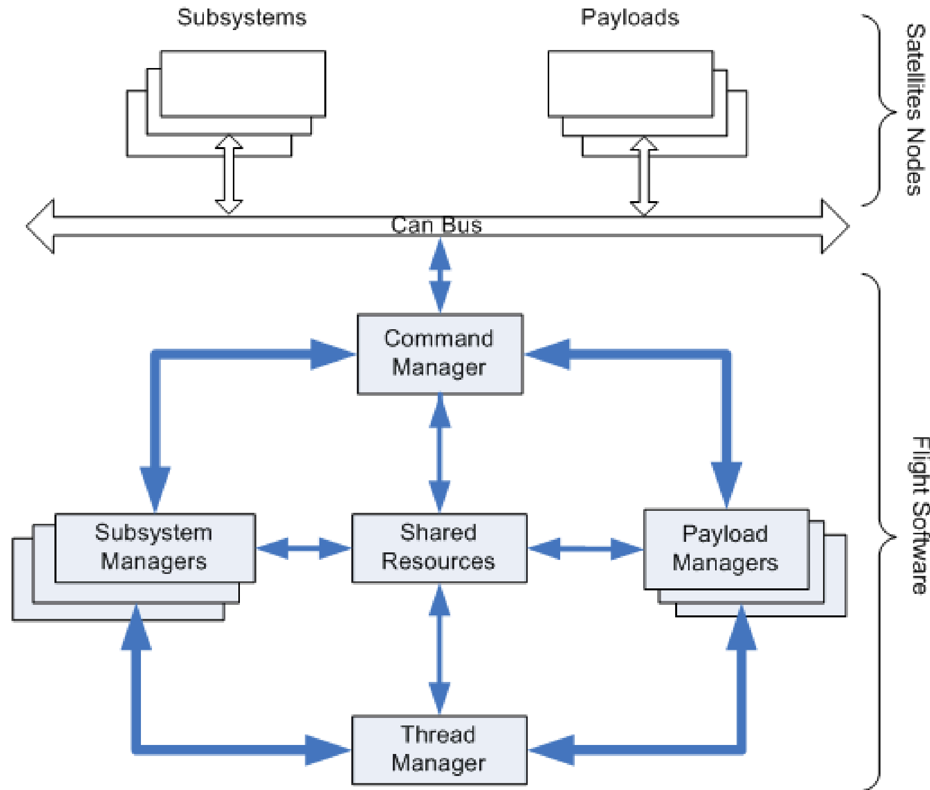


Figure 3.1: Flight Software Design

Command Manager - This is the main module for the flight software. This validates, schedule and execute commands for the satellite nodes. The command manager also identify the exact subsystem or payload manager, that can execute the response message from satellite nodes, which has the same address as the destination address of the message identifier. The detail design of this manager is discussed in section 3.3, and the implementation is defined in chapter 4.

Subsystem and Payload Managers - Each subsystem manager must consistently monitors its respective subsystem (satellite node) since it affects the health of the satellite, whereas each payload manager must control its respective payload (satellite node) to execute the required satellite service. There are many satellite nodes, and each node needs its own node manager in the flight software. Each node manager must accommodate all possible response messages that can result from the execution of the commands sent to the respective satellite node. Nevertheless, managers have common procedures such as execution of received messages from satellite nodes and

then generating response messages (for satellite nodes) if necessary. The flight software provide the standard procedure that can be employed to develop any subsystem and payload manager, and less work is needed to accommodate all possible control modes, and this is discussed in details in chapter 5.

Shared Resources - They indicate the shared memory and message queues available to all software modules. This is commonly used by the command manager to send messages received from satellite nodes to the managers in order to execute them. A message queue is used by node managers to send response messages back to the command manager to send them onto the satellite nodes. Global data structures are commonly used by the node managers to supply information to the command manager in order to record them into files such as the housekeeping data file.

AO is conducted by both node managers and command manager. The command manager regularly send commands to the satellites nodes and the node managers evaluate the response and generate the commands to control the satellites nodes to maintain its orientation and a healthy operating mode. The retaining of data files, until the ground station has downloaded them or a defined memory size is reached, is conducted by the command manager as the main flight software module.

Thread Manager - This was developed to create, monitor and terminate the flight software managers, since they are developed as threads, and this is described in the following section.

3.2.1 Thread Manager

Thread manager must be given a unique address, even though it is only responsible for node managers within the flight software other than the satellite node. The commands for switching on and off subsystem and payload managers are destined to this manager, and this should be considered as well by the user at the ground station. Every node manager is also given a thread identifier excluding the node manager identifier. The thread manager can create and terminate a node manager using its thread identifier, as

illustrated in Figure 2.2. A thread can be created and terminated at any time during the course of execution. This can be conducted from the commands read from diary files, and also for the safety of the satellite if the detrimental change is experienced by a node manager.

Timeout functions are implemented by the thread manager, and they are necessary to monitor the message execution period in the flight software managers. The general execution time for the message must be defined especially for messages executed by node managers. If the manager takes too long to execute the message, and the timeout eventually expires, the thread manager must terminate the execution of such a message. The thread manager can then terminate the manager, only if the termination of the message execution is not successful.

The thread manager also updates the satellite status flags, such as indicating the status of each node manager and satellite node whether it is switched on or off. This is necessary for the command manager to avoid forwarding messages to the node managers and satellite nodes which are switched off.

3.3 Command Manager Design

The main purpose of the command manager is providing a communication link between satellite nodes and node managers. Satellite nodes are the real subsystems and payloads connected to the OBC via the CAN bus, where as the node managers are the software application modules developed in the flight software. Each and every subsystem and payload hardware resource has its corresponding software manager module.

The command manager should provide reliable and safe communication, to keep the satellite in a normal operational mode. This can be achieved by ensuring that every command that is being sent to the satellite node is valid and timing is accurate. The scheduler module is employed by the command manager to control the flow of commands sent to the satellite nodes. The diary file is the groups of commands organised

together in order to serve the unique purpose when they are executed by the flight software, and this is discussed in the following section.

3.3.1 Diary File Design

The diary files can be created and included in the flight software package before the satellite is launched, or created and sent to the satellite while it is already functioning in space. The structure of the diary file should be defined, for the flight software to read and examine the command and execute it accordingly. The following section explains the design of the command structure that constitute a diary file.

3.3.1.1 Command Design

The command is composed of the fields shown in Table 3.1. The message identifier, DLC and data field have been used according to the CAN bus protocol as explained in section 2.6. The only difference is that flight software message identifier uses 32 bits instead of 29 bits. This is due to the 3 extra bits appended at most significant bit (MSB) in order to create a defined variable (*long int*) in C programming language, and this extra bits must be always set to zero.

Table 3.1: Command Fields

Field Name	**Field Type**	**Field Size in bytes**
Message Identifier	Int 32	4
DLC	Char	1
Data Field	Char array	8
Execution Time	Int 32	4
Repeat Value	Char	1
Repeat Time Interval	Int 16	2
Diary Identifier	Char	1

In the message identifier, address 0 is reserved for message broadcasting within the satellite system [1]. Therefore any existing node connected to CAN bus can broadcast

or read the message of destination address 0. The flight software is defined to be composed of node managers, each representing a satellite node. Each node manager must be given unique address of any even number. This makes it easy for the message filter in the OBC CAN interface to consider any message with an even number as a destination address, and forwarded it to the flight software. The satellite nodes must be given a unique address of any odd number, to avoid conflict with the messages destined for flight software node managers. For instance, the address for the power subsystem (satellite node) is 1 and power manager (flight software node manager) is 2.

Execution time field is chosen to be 4 bytes wide in order to represent any unix time in seconds when the satellite is in space. This field is only applicable for time dependent commands, used by the scheduler algorithm for prioritising commands in order to forward the message to the satellites nodes at the specified execution time. Execution time can be either absolute or relative. The absolute execution time specifies the exact time in which the command should be executed by the flight software. The relative execution time specifies the amount of time remaining, for the flight software to execute the command, this is determined from the time at which the command is scheduled by the flight software.

There are messages that needs to be executed more than once or that should repeat forever, such as telemetry messages which request the status of the satellite nodes for monitoring purposes. The CAN bus protocol does accommodate this type of message and they are defined as unsolicited telemetry [1]. Repeat value and repeat time interval fields are only applicable to the flight software, as an alternative to unsolicited telemetry messages. Those fields are only applicable for messages that need to be executed either indefinitely or not more than 254 times since the repeat value field is chosen to be 1 byte (maximum value is 255). The repeat value field stores the maximum number of repetitions needed for the messages, for indefinitely execution the repeat value should be set to 255. Repeat time interval is used to determine the next execution time after the message is forwarded to CAN bus.

The repeat time interval can also be used for the commands that are defined to execute once. This is necessary when the command execution fails while it is still possible to

execute the command again. The repeat value must be assigned with a value of 0, since they are defined to be executed once. This is designed to accommodate the soft real-time application, as discussed in section 2.2.

Every command in a diary file must have the same diary identifier, which is used to identify the source diary file of the command when it is executed by the flight software. This is necessary when some of the commands of the same diary file are not executed sucessfully. Then there might be no use to continue executing the rest of the commands of the same diary file since they are related to one another for accomplishing the projected service of the satellite. Therefore, this field can also be used for validating the commands that need to be sent to the CAN bus.

The satellites system performs various functions, and the flight software uses commands read from the diary files to control the satellite nodes and execute the satellite service. The diary files are classified into groups and the following section provide a description about types of diary files.

3.3.1.2 Groups of Diary Files

When the user at the ground station is designing the diary files, each must be given a unique diary identifier. Each diary file must be designed to provide a unique service when it is executed by the flight software. The diary files are classified according to the services they provide to the satellite or the ground station. They also classified according to the types of commands that the diary file can accommodate, and they are: autonomous, mission and error recovery diary files.

Autonomous diary file contains commands that are designed to execute from the on set of the flight software execution, or immediately when they are uploaded into the satellite in space by the ground station. The main execution purpose of autonomous diary files is to constantly monitor the satellite nodes, hence most of their commands are defined to execute many times and while others indefinitely. They are designed for AO, and they are composed of telemetry (including unsolicited) commands.

The autonomous file must be a single file, but it can be divided into multiple files if necessary, since their commands are independent from each other even though they serve the same purpose. The diary identifier should be set to 1, even if there are multiples of the autonomous diary files. The flight software can use diary identifier to recognise commands from this diary file, and then continue executing them even if the execution of a certain command fails as discussed in the previous section.

The mission diary files are defined to contain commands which fulfil the satellite mission when they are executed. Each and every file serves a unique portion of the satellite service, and this depends on the satellite objectives during design. These commands (in a diary file) are not executed regularly and they are highly depended to each other to provide the desired satellite service, mainly to the ground station. This diary file uses time dependent factor to control the sequence of the execution of commands.

Every mission diary file must have a unique diary identifier of greater than 1, since 1 is given for autonomous diary file and 0 is reserved for error recovery diary file. This will enable the flight software to avoid executing the remaining commands of the same diary file, if the command execution fails while the diary file is in the execution process. If the diary file is designed to execute once and it is efficient to execute it again only if the execution fails, the repeat time interval field must be given the next possible execution time and the repeat value must be set to 0, as discussed in section 3.3.1.1.

The error recovery diary files contain commands which keep the satellite in a safe operational mode, by removing the error or minimising the impact of an error to the satellite system, when they are executed. The error recovery commands are not necessary dependent on each other but on the existence of an error. If the error can be removed during the execution of error recovery diary file, then there will be no use to continue executing the remaining commands from this file and the flight software must stop executing commands from the file. Every node manager should have its own error recovery file, since the error is commonly detected by the node manager due to the response messages from respective satellite node. This can be simple if it is the same node manager that determines the commands to execute for the detected error.

The diary identifier for error recovery diary file is chosen to be 0 even if the node manager has more than a one type of this file, since they serve the same purpose. When designing the error recovery diary file, for any error there should be a start address and maximum number of commands to read from this file. This is possible since the node manager can detect various kind of errors and handles them differently. The error recovery commands, for each error, must start and end with the command(s) that examine the existence of the error. This may be necessary to ensure that the execution of error recovery commands is not going on for the error that no longer exist, and also to verify that the error does not remain valid if the error recovery commands were able to remove it during the execution process.

3.3.2 Command Validating, Scheduling And Dispatching

The command manager reads a command from the diary file and then verifies whether the command is valid or not. The valid command is scheduled and then dispatched to the satellite node at appropriate time defined in the command execution time field. Otherwise, if not valid, the command will not be executed, and this prevents the satellite from entering into unpredicted errors or states. The following section describes the means used by the command manager to validates command.

3.3.2.1 Command Validation

The command manager evaluates the command by examining each and every field in the message identifier, and the message type is evaluated according to the CAN bus protocol as stated in section 2.6.2. The channel, source and destination addresses are evaluated according to messages defined for the flight software application. The execution time (for absolute time only) for the command is also evaluated to verify whether it has not yet passed or not.

The messages that were detected as invalid are then recorded into invalid messages file, and the fields of this file are shown in Table 3.2. The message number records the

current number of messages logged into the file, and the log time records the current time at which the message is recorded. The invalid field records the field that was detected as invalid and the command fields record the entire command as read from the diary file.

Table 3.2: Data Fields For Invalid Message File

Field Name	Field Type	Field Size in bytes
Message Number	Char	1
Log Time	Long int	4
Invalid Field	Char	1
Command Fields	Refer to Table 3.1	21

This file can be useful once it is downloaded at ground station, for the user to investigate those commands and simply rectify an error and send them if it is still possible to execute the required service. If the user does not detect any error as indicated in this file then the user must find the means for the satellite to execute such commands, such as upgrading the flight software application.

The flight software should execute all diary files, with valid commands, including the files that can be arbitrary uploaded to the satellites while it is already in space. The messages types and execution time varies depending on each and every command in every diary files. The following section describes the scheduler algorithm employed by the flight software for determining the sequence in which commands can be executed.

3.3.2.2 Command Scheduling

The flight software is using a linked list for scheduling commands. It was chosen since it can dynamically insert a command at any point in the linked list, according to the order in which they are organised, and it is easy to access commands at any point in the linked list. The linked list is made up of linked list nodes connected together. Each linked list node is divided into two parts namely: command field and pointer links. The command field stores the command read from diary file and pointer links is used to link the node into the linked list. The linked list is referenced by the linked

list pointer, which always points to the beginning of the linked list. This is shown in Figure 3.2. The procedure for creating, inserting and deleting the node in the linked list is discussed in Appendix B.

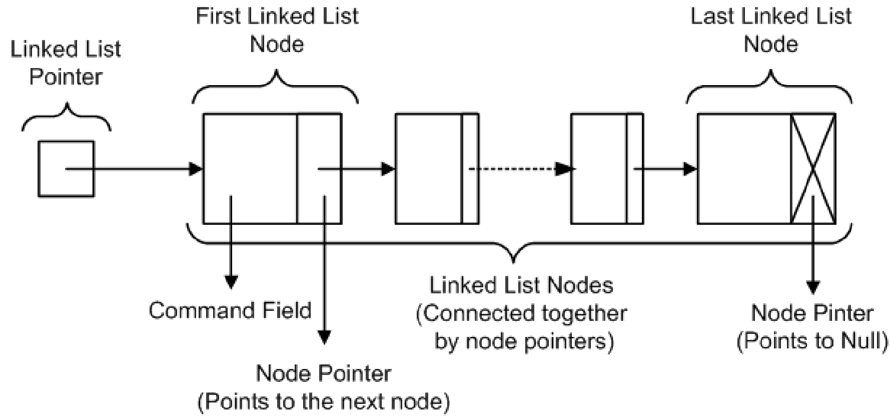


Figure 3.2: Commands Scheduling Using Linked List

The flight software is scheduling commands according to execution time field for time depended commands, other than the message type field used by CAN bus protocol as discussed in section 2.6.1. The execution time is used for commands that have absolute or relative execution time. During the scheduling of the relative execution time command, the current time at which the command is scheduled is added to the command execution time, and this convert the relative into an absolute execution time which is used for command scheduling. The execution time was chosen since it is very easy for the user at the ground station to control the sequence of command execution, when designing the diary files.

The command scheduling is conducted in an ascending order, where the command with less value of the execution time is positioned at the beginning of the scheduler. If some of the commands share the same execution time, then the sequence in which they were read from the diary files is considered, meaning the command read first is given a highest priority. There are three types of command schedulers defined by the flight software namely, error recovery, hard deadline and soft deadline schedulers. This is shown in Table 3.3, indicating the types of the diary files that are accommodated by the respective scheduler.

Table 3.3: Command schedulers

Scheduler Name	Command types	Diary type	Priority
Error recovery	Time dependent (relative time only)	Error recovery files	0 (Highest)
Hard deadline	Time dependent (relative and absolute time)	Mission files	1
Soft deadline	Time dependent (relative and absolute time)	Autonomous files	2 (Lowest)

The error recovery scheduler is designed for scheduling the error recovery commands when the error is detected. This scheduler is given a highest priority since it can only be loaded when the node manager has detected an error, and the error recovery commands should be executed as soon as possible in order to keep the satellite in a safe mode, as discussed in section 2.5.2.

The hard deadline scheduler is designed for scheduling commands with hard deadlines, refer to section 2.2 for the definition of soft and hard real-time systems. Even though the mission diary file does not only accommodate commands with hard deadlines, the mission diary files must be submitted into this scheduler since each of this diary file brings a unique service. This will enable the flight software to prioritise them, since they are given the highest priority in the absence of error recovery commands, to meet the real-time application.

The autonomous files are scheduled by the soft deadline scheduler. The autonomous diary file is also build up of commands with soft deadlines, since the satellite status information can be requested at any time. This scheduler is given the lowest priority since it is expected to always have the commands ready for execution or waiting for the execution time. And the autonomous diary files is build up of commands that are executed many times or indefinitely.

The command manager is designed to send only one message at a time. A provision is made for the case where more than one command scheduler has the command ready for execution. Table 3.3 also shows how the command manager prioritises the command

scheduler, the priority field given 0 is considered as the highest priority. The manner in which the commands are executed from the schedulers is described in the following section.

3.3.2.3 Command Dispatching

The command manager sends only three command fields into the satellite nodes and they are shown in Table 3.4. This fields were considered since they are also used by the CAN bus protocol for data transmission, as discussed in section 2.6. The other command fields (refer to Table 3.1 for the excluded command fields) are used by the flight software for command scheduling and determining the number of times in which the command should be executed.

Table 3.4: Command Fields Sent To Satellite Nodes (CAN Message)

Field Name	**Field Type**	**Field Size in bytes**
Message Identifier	Int 32	4
DLC	Char	1
Data Field	Char array	8

The command manager ensures that every command is scheduled before it is sent to the satellite nodes. A provision is made for commands from the manager message queue (MSQ), as they are uploaded (into the queue) by the node managers. The MSQ is a scheduler on its own, and it is defined to schedule using the message type which is also used by CAN bus protocol to schedule messages as described in section 2.6.1. The messages can be placed onto the queue at any time and stored until the command manager retrieves them, as described in section 2.4.1. Once a command manager read a command from the queue, the message is dispatched into the satellite nodes without scheduling it. Therefore the message from the queue has got the highest priority, compared to other schedulers. The MSQ is expected to be empty most of the time, since not every message executed by the node manager generate the response back to the satellite nodes, but it is common for messages with data fields greater than 8 bytes.

The execution delay is expected since the command manager sends only one command at a time, and the execution of commands is prioritised according to the scheduler as defined in the previous section. The scheduler must execute the command when the execution time is reached. If the repeat value field of the executed command is greater than 0, the command must be rescheduled. The rescheduling mechanism must update the executed command, by adding the repeat time interval to the execution time, and then inserting it into the command scheduler. The executed command must be deleted from the command scheduler, and the next command, if the scheduler is not empty, waits for a execution.

The command manager deletes the executed command by positioning the command scheduler pointer to the second node of the command scheduler linked list, and then free the memory allocated to the first linked list node which contains the executed command. This is done to ensure that the command scheduler pointer always points to the beginning of the command scheduler. The command manager can also delete every command of a certain mission diary file in the scheduler if the diary identifier is specified, with a value greater than 1, when calling the delete function. This is necessary if the execution fails during the process of executing mission diary file as discussed in section 3.3.1.2.

Although it is anticipated that each diary file is organised in a way that commands will be executed sequentially starting at the beginning of the file, the command schedulers must schedule them again. The main reason is that flight software should handle unlimited number of diary files at the same time in order to execute commands at predetermine time, and it is possible to have more than one commands with the same execution time in the same or different diary files. Therefore it is necessary for the flight software to schedule all diary files respectively into command schedulers, before executing them into satellites nodes.

The command scheduler is not limited in terms of the maximum number of commands it can handle but the only mitigating factor is the availability of memory space, since every command entering the linked list request a certain size from the memory in order to create the new linked list node. This is also influenced by the rate at which

commands are executed. It can also influenced by the types of the commands. If most of the commands are only executed once, as they should be since it is only telemetry messages that are expected to be executed more than once, the delete function frees the memory allocated to the executed command, and this frees the memory space for the new command entering command scheduler.

3.4 Software Development Environment

The target platform to execute flight software is eCos, in the OBC. The eCos only support a few programming languages as mentioned in section 2.4, but not every standard function of the supported language is applicable to eCos. There is also compatibility layers which provide their own function to allow the application code to interface with an eCos, as explain is section 2.4.1. Therefore a platform to develop and debug the flight software application is needed before installing it to the real system, OBC which runs eCos.

It has been mentioned in section 2.4, that eCos is for a single process application with multiple threads; hence the flight software application must produce a single output executable file. The flight software need to be developed in an environment that can output an executable file that is easy to port and run in the eCos environment. The Linux OS was used to run the flight software executable file, and the compiler used is GCC.

The flight software was developed using C programming language. POSIX functions were used since they provide an interface to the eCos. Every function implemented in flight software is supported by eCos, in order to easily run on the eCos in the OBC. In Linux OS, it is easy to install POSIX supported packages, and compile application with POSIX functions using GCC.

Chapter 4

Command Manager

This chapter provides full details about the implementation of the command manager to communicate properly and effectively with node managers and satellites nodes.

4.1 Command Manager Processing

The command manager is developed as a thread, and once it is initialised by the thread manager module, it runs forever unless the program is terminated either due to a detected error or by the user. The main purpose is to control the execution of commands from command schedulers defined in Table 3.3. The execution procedure for the command manager is shown in Figure 4.1 and this section explains the process in detail.

When the command manager has dispatched a command to the satellite node, the command is recorded in a command manager's transmit buffer and the transmit flag is also raised. Satellite nodes can initiate the communication with the node managers by responding to the unsolicited telemetry commands loaded to the CAN bus, since the CAN bus can keep on sending these messages to the respective satellites nodes regularly without interacting with the flight software. The execution of messages received from satellite nodes is discussed in the following section.

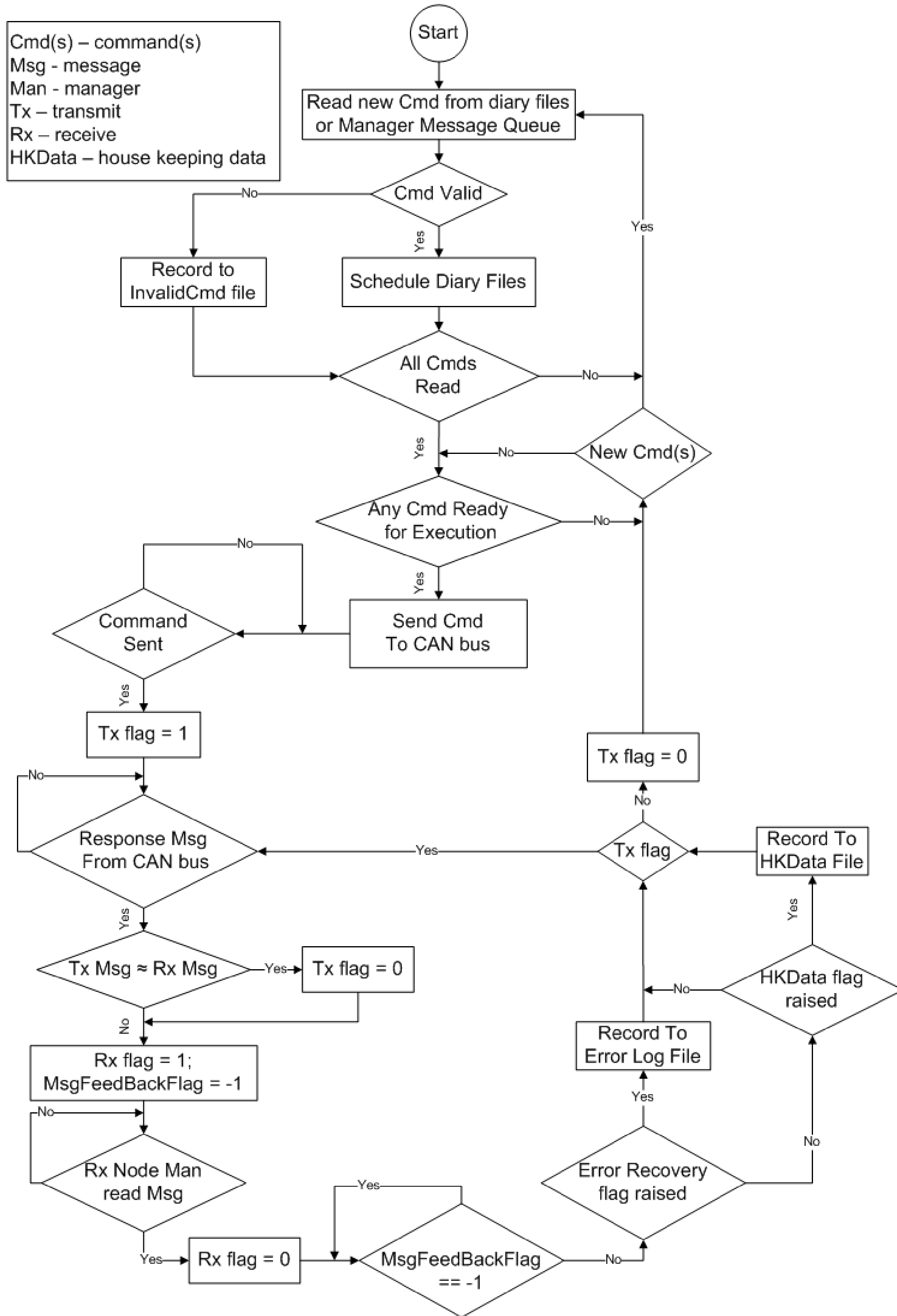


Figure 4.1: Command Manager Process Flowchart

4.1.1 Processing A Message Received From Satellite Node

It has been stated in section 2.6 that every node connected to the CAN bus has a unique address and a message filter which are also used for downloading the messages, that are sent to the node, from the CAN bus. The command manager passes the message receive buffer and address identifier as an argument to the function for requesting the message from the CAN bus. If the message destination address is an even number, the message receive buffer will be loaded with the new message. Otherwise the CAN bus will only allow the satellites node with the same address as the message destination address. Refer to section 3.3.1.1 for the type of addresses recommended for the node managers and satellite nodes.

When the command manager has received the message from satellite node via CAN bus, it uses its transmit buffer to determine whether the message received is due to the transmitted command or a certain satellite node has initialised communication to the flight software. For every command sent to the satellite node, there is a response message(s) defined as stated in section 2.6.2. If the received message from the satellite node is not the anticipated response message, then the command manager assumes that it has not yet received the response from the satellite node, but it handles the received message like any other message. The transmit flag is cleared when the received message from the satellite node is the expected response message.

Every message received by the command manager must be passed to the relevant node manager. The node manager is notified about the arrival of the new message, since the command manager raises the receive flag when it has received a new message from the CAN bus. The node manager passes it's receive buffer and manager identifier to the command manager. If the manager identifier is the same as the message destination address, the command manager loads the new message into the manager receive buffer and then clears the receive flag.

The command manager can only receive one message at a time. As a result, a single node manager receives one message at a time. The command manager remains in an idle state until the message is executed thoroughly by the node manager. It receives

a message feedback indicating the message execution status from the node manager, as indicated in Figure 4.1. The reasons for waiting until the node manager finishes executing the message are discussed in the following section.

4.1.2 Processing After Node Manager Has Executed Message

The node manager can raise the housekeeping data flag or error recovery flag when it is executing the received message. The housekeeping data flag is raised if the executed message contains information different from the last executed message of the same massage identifier. The error recovery flag is raised if the error recovery scheduler is loaded with commands, and this flag is cleared when this scheduler is empty. The housekeeping data file is loaded with data when the housekeeping data flag is raised, and the same protocol is followed for an error log file when the error recovery flag is raised. These files are mainly created for examination of the satellite performance at the ground station.

These files are chosen to be controlled by the command manager, although the process of writing to them is triggered by the node managers. This is due to the fact that every node manager can use these files, and then it becomes simple for command manager to control them since it is communicating with every node manager in the flight software. There can be only one node manager that initialises the process of writing into these files, because they can only be activated during the execution of the received message.

These files were chosen to be a random access files as they use binary data to store information, therefore they can accommodate a large amount of data in a maximum defined file size. Since it is not human readable, at the ground station there should be an application code defined for reading these files into text files.

The maximum size of each file must be defined to avoid having files with large file size, and this enables the ground station to download files easily. The next file (of the same purpose) is created if the current file has reached the maximum file size. The maximum number of files that can be generated must be defined as well to limit

the memory they can occupy if the ground station delayed downloading them. If the maximum number of files has been reached, then if the first generated file does exist then it will be overwritten and the only the new updated file is available for the ground station to download. The structure of the data loaded to each file is defined as well, and the following sections describe each files properties.

4.1.2.1 Housekeeping Data File

The Housekeeping Data File is intended to load mostly the telemetry (including unsolicited telemetry) response messages received from satellite nodes, because they indicate the condition of the satellite. During the design of the data fields for the housekeeping data, it was noted that this can become a huge file since telemetry messages are repeatedly executed. Therefore the housekeeping data file does not record every telemetry respond messages, but the messages that change the status of the receiving node manager as it represents the status of the respective satellite node. The data fields for the housekeeping data file are shown in Table 4.1, and they were chosen to minimise the data size but accommodating valuable information that is necessary at the ground station to examine the condition of the satellite.

Table 4.1: Data Fields For Housekeeping Data File

Field Name	Field Type	Field Size in bytes
Message Number	Char	1
Log Time	Int 32	4
Node Status Vector (NSV)	Int 16	2
Message Received	CAN Message	13

The message number is used to keep track of every loaded message in the file, as it increments every time the housekeeping data file is called to load a new message. This is also useful since there can be multiple files due to the defined maximum number of files, so this can also be used to track the sequence of files generated. Suppose the maximum number of files has been reached, then the message number is reset to zero and the first generated file is also overwritten to indicate the start of the new files.

The time at which the message is logged into the file is also recorded into the log time field, and this may be also be used for the same purpose as a message number. The node status vector (NSV), defined according to the node manager, is intended to verify if there is any flag(s) changed during the execution of the received message. This can be confirmed by comparing with the recently recorded NSV of the same node manager. The message received field records the exact message generated by the satellite node and executed by the node manager, and this is the same message that resulted in logging into this file. This message will be also evaluated at the ground station to verify whether the node manager has evaluated the message correctly or not.

4.1.2.2 Error Recovery Data File

The error recovery data file is defined to record messages that generated errors in the satellite and the information about the satellite behaviour during the execution of error recovery commands. Once this file is downloaded at the ground station, the user can analyze and then (if necessary) compile a diary file, especially when the error was not eventually cleared and it degrades the satellite operation. The type of data stored in a file is shown in Table 4.2.

The log time field is used for the same purpose as described for the housekeeping data file. The Transmit flag indicates whether the received message from the satellite node and executed by the node manager is the expected response message or not. If it is the expected response message then a transmit flag is cleared, otherwise it is raised, and this is also indicated in Figure 4.1. The receive flag indicates whether the node manager has read the message with a success (flag is cleared) or not (flag remains raised). Transmitted and received messages are recorded to indicate that a message sent to the satellite node was the correct message and relevant to the situation and also whether the response received was relevant to the transmitted message.

Error recovery flags, node and satellite status flags are recorded twice, which means that before and after the received messages are executed by the node manager. This is done to determine the exact effect that resulted during the execution of the received

message, by comparing the flags before the message is executed and after it has been executed as it updates them during the execution process. The node status flags contain information only about the receiver node manager, since every node manager has its own flags. Satellite status flags contain general information about the status of each and every satellite nodes and node manager, such as which nodes are switched on or off, including information about the error recovery flags of the satellite.

Table 4.2: Data Fields for Error Recovery File

Field Name	**Field Type**	**Field Size (bytes)**
Log time	int 32	4
Transmit flag and receive flag	2(char)	2
NSV (before executing rx msg)	int 16	2
node status flags (before executing rx msg)	3(char)	3
Satellite status flags (before executing rx msg)	4(char), int 16	6
NSV (after executing rx msg)	int 16	2
node status flags (after executing rx msg)	3(char)	3
Satellite status flags (after executing rx msg)	4(char), int 16	6
Transmitted message (as read from Dairy file)	Refer to Table 3.1	21
Received message	CAN message	13

Chapter 5

Subsystem And Payload Manager

The nanosatellite is a combination of subsystems and payloads, defined as satellites nodes, connected together into the OBC using CAN bus as a communication protocol, as already described in section 1.1. Payloads are responsible for implementing the satellite service and communicating with the ground station, whereas subsystems are responsible for keeping the satellite function in a safe mode. Even though they differ in terms of responsibility, each has its own developed software application running on the hardware that is built in accordance with its responsibility.

The flight software must monitor the health of the satellite nodes and also control them to remain in a safe operating mode while executing the satellites mission. The flight software contains manager modules, where each manager module represents a respective satellite node. Each manager module, in the software application, must evaluate and execute messages and commands that the respective satellite node could receive, execute and transmit as well. Therefore each manager module must be developed in accordance with its respective satellite node.

There are many satellite nodes that can be part of the satellite, as described in section 1.1. Some of them are not yet defined while others are in the development process. The candidate has developed an application that can be used to develop the manager module of any satellite node. The following section explains the algorithm used and the

principles that can be used as well as to develop an effective manager module according to the respective satellite node.

5.1 Node Manager Structure

The developed application for the manager module is defined as the node manager, and the execution flow is illustrated in Figure 5.1.

Every node manager must be developed as a thread, and once they are created they run concurrently and independently. Therefore, the node manager can be created and terminated during the course of flight software execution. The thread manager, as described in section 3.2.1, is developed for creating and terminating the node managers due to the commands read from diary files or if undesired behaviour is detected from node managers. The undesired behaviour can result during the message handling, especially if a node manager takes too long to finish message execution until the specified message execution timeout expires.

Every node manager must be given a unique identifier, as described in section 3.3.1.1, which can be used by the command manager to identify the recipient (node manager) of the new message it has received from satellites nodes.

The node managers keep on polling the command manager receive buffer using the function which specifies the node manager identifier and receive buffer as an argument. If the identifier corresponds to the message destination address, then the command manager uploads the message to the node manager receive buffer, otherwise the function returns a value of -1 to indicate that there is no message for the calling the node manager. Therefore only one node manager can execute a message at a time, and any received message (in the flight software) can be executed by a single node manager.

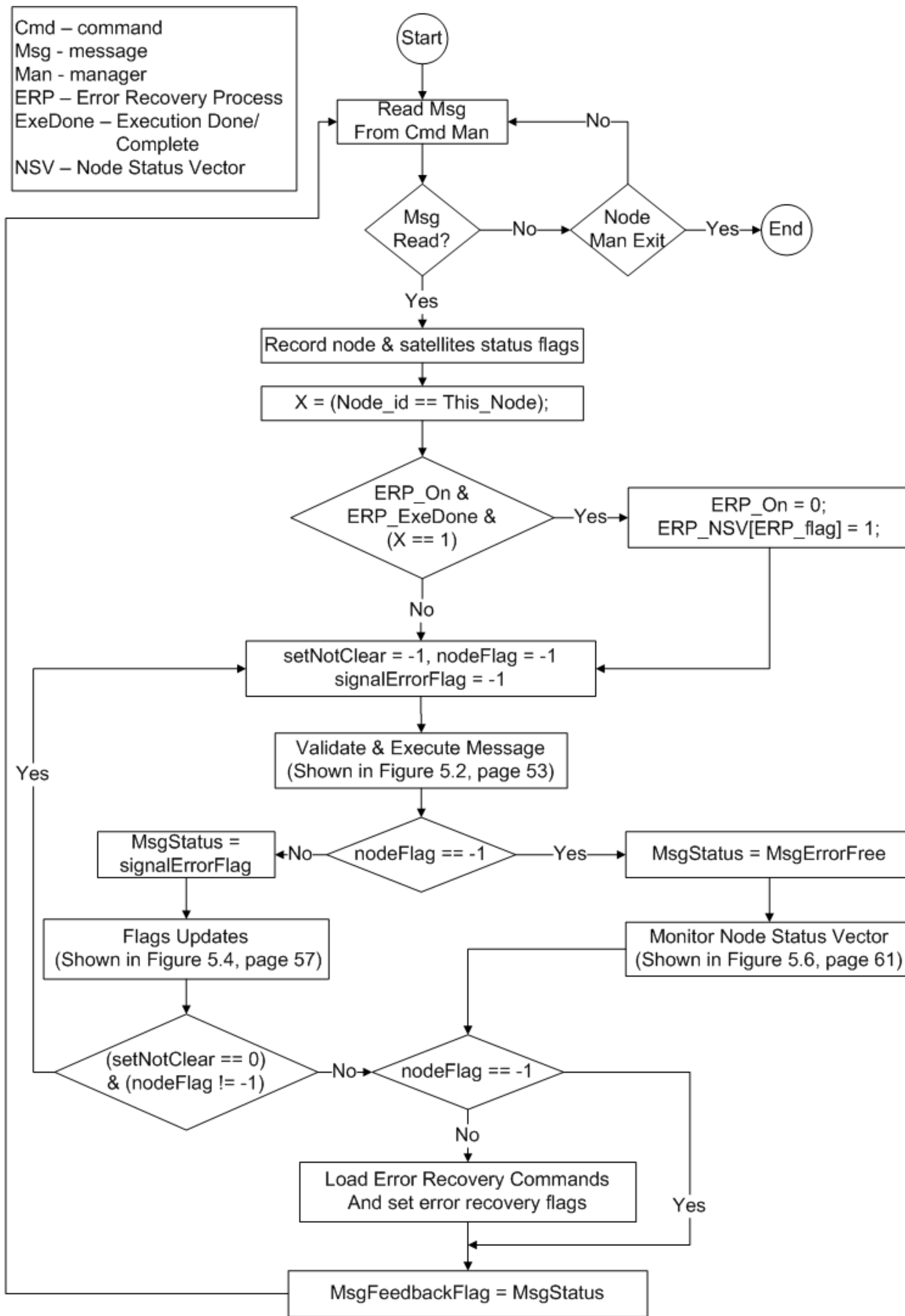


Figure 5.1: Node Manager Flowchart

Once the node manager has successfully downloaded a new message from the command manager, it starts by recording its current node status flags. Flags are recorded to indicate the status of the node manager before the received message is executed. These flags are updated during the course of message execution. After the node manager has finished executing the message, the recorded flags are used for comparison with the newly updated flags. If they are not the same, the housekeeping data flag is raised in order to enable the command manager to record received messages and node status flags in a file because they bring a conclusion about the behaviour of the satellite node as discussed in section 4.1.2.

If the flight software is in the state of error recovery process (ERP), and this is the last message response due to the execution of error recovery commands, the node manager has to clear the error recovery flags and raise a flag which will indicate that the ERP has been conducted for the error/flag. This is done in order to enable the node manager to initiate the ERP, during the course of the message execution, for other errors that already are or still yet to be detected. This is also explained in this chapter, especially in section 5.5.

The node manager initialises three main flags with a value of -1, and they are: $signalError$, $setNotClear$ and $nodeFlag$. The $signalError$ (or signal error) flag records the error signal of the received message, if the error is detected. The $nodeFlag$ (or node flag) records the selected flag of the NSV which represents the detected error signal of the respective message identifier, or an already raised flag in the NSV that needs to be cleared if the message is error free. The $setNotClear$ records the value that is used for updating the selected flag in the NSV. This flag can be updated during the process of validation and execution of the message. The value of -1 was chosen to initialise the flags since 0 represent an existing flag which is an least significant bit (LSB) in the NSV.

The node flag can also be updated during the flags updates and this is described in section 5.3, or during the monitoring of the NSV as described in section 5.4. If the node flag has been assigned with a flag other than -1 and remains uncleared, then the error recovery scheduler can be loaded with commands to initiate ERP. This is also shown in

Figure 5.1 where the ERP is the last process, before recording the execution status of the received message, that can be conducted if the node flag remains raised and this is explained in section 5.5. Nevertheless, the received message has to be validated before it is thoroughly executed, and this is described in the following section.

5.2 Message Validation and Execution

This section describes the validation process that can be employed by the node manager for every received message. Once a received message is validated, then it has to be executed. The execution procedure depends entirely on the type of message and the node manager that is executing the message. This section discusses the general algorithm that can be employed during message execution, and this is illustrated in Figure 5.2, whereby the role is to firstly determine the function handler and then call it to evaluate and finally execute the message.

The message type and channel of the message identifier are used for determining the function handler of the message. If the message type or channel is not supported by the node manager, the received message is considered as an invalid message. The detected error (invalid message type or channel) is recorded into a signal error flag. The node flag is assigned with the flag which represents the error signal in the NSV. The $setNotClear$ flag is raised in order to raise the flag identified by the node flag in the NSV. The execution returns from determining the function handler into the process of updating flags and this process is discussed in section 5.3. Otherwise, if they are both supported, the respective function handler is called to continue to evaluate and execute the messages.

Every function handler defines properties of the data length and data range of the message it receives. If the message received does not correspond to the function handler properties, then it is considered as invalid. The procedure is the same as for invalid message type or channel, except the value given to the node flag and signal error flag.

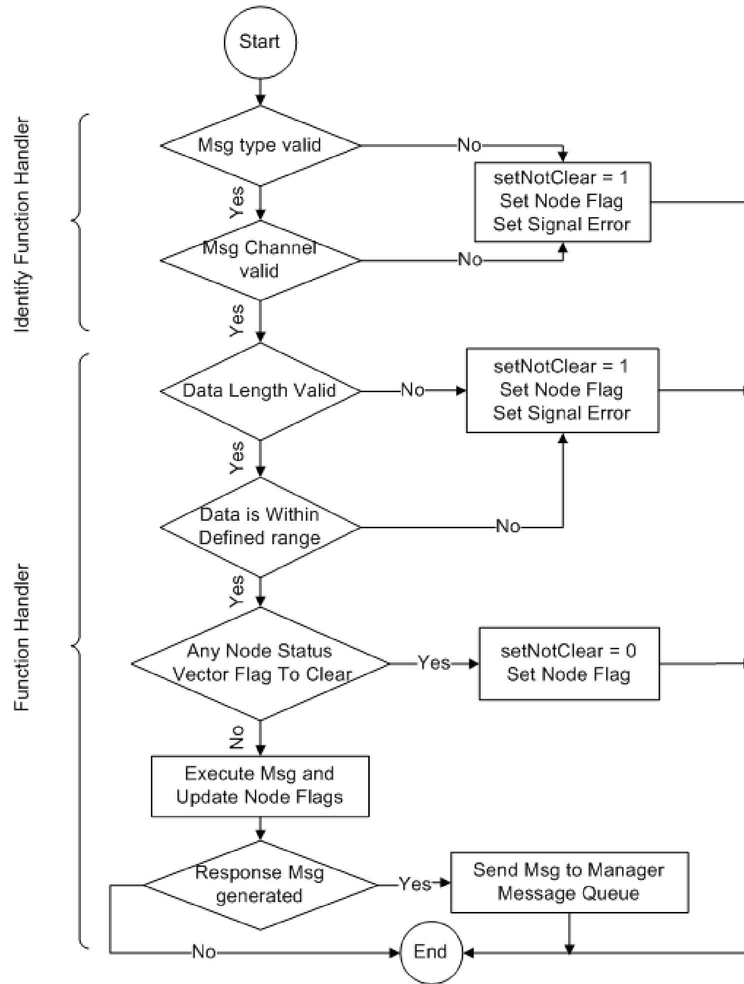


Figure 5.2: Evaluate Received Message (Extract from Figure 5.1)

The detected error (invalid data length or data range) is recorded to the signal error flag. The node flag is also assigned according to the flag in the NSV, which represents the detected error.

If the message is valid, meaning the function handler properties correspond to the received message, the NSV is examined first by searching any raised flag related to the type of the message received. Only one flag can be detected at a time, and if it is detected the node flag is set according to the detected flag and $setNotClear$ flag is set to 0 in order to clear the related in the NSV. This is done to clear any related flags to the message identifier other than in the NSV, since the current message has no error detected and also shows that the satellite has recovered from the error, if there was an error detected before and it remained raised.

The received message is thoroughly executed by the function handler when it is error free, and there are no remaining raised flags in the NSV that are related to the received message. There are various types of messages that the function handler can execute, some need a message respond or a command to be generated and sent back to the satellite node. The provision is made for those types of messages; the function handler can compose the message or command and send it to the MSQ. The message or command must be composed of the following fields, message identifier, data length code and data fields. Those are the only fields used when sending a message into CAN bus as described in section 3.3.2.3. Once the command is downloaded by the command manager from the MSQ, it will be sent into CAN bus immediately without being scheduled, even if the there were commands ready for execution.

The NSV can accommodate more than one flag for different errors of the same message identifier and this is illustrated in Figure 5.3. The voltage telemetry respond message executed by the node manager, defined as the power manager, shows the NSV flags that are active if the error is detected in the message. A single flag in the NSV is related to various node flags and error flags. Therefore an evaluation of the NSV can be repeatedly conducted to handle a single flag of the NSV and update other related flags at a time. This is conducted by returning from the function handler of the message and then updating the flags until there are no flags that need to be cleared and which are also related to the received message that is error free.

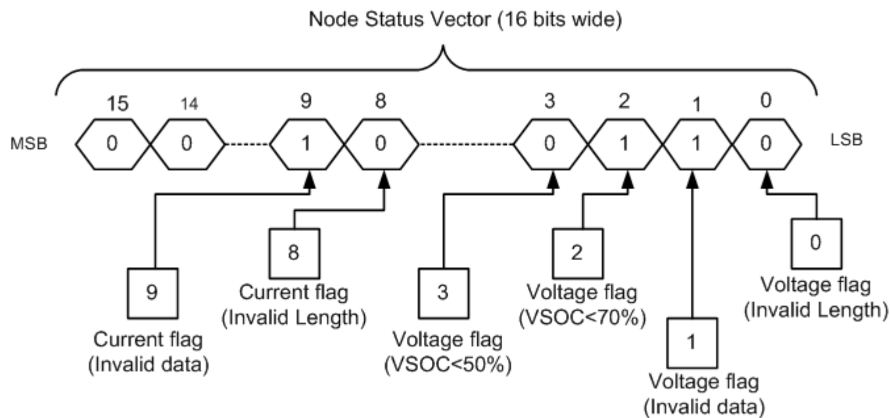


Figure 5.3: Node Status Vector (NSV) Indicating Voltage and Current Flags

There is a difference between the signal error flag and the node flag, although sometimes they can have the same flag. This is also illustrated in Figure 5.3, where by the signal error flag can be invalid data, but the node flag can either be invalid data for a voltage flag or a current flag. Therefore the values stored in these flags depend entirely on the type of the message received and the definition of flags within the NSV. The following section provides an example of how the flags are updated during the evaluation and execution of the message.

5.2.1 Sample For Message Execution

Suppose the received message type is the telemetry acknowledge (TLM_ACK) for voltage respond and the voltage state of charge (VSOC) is 90% as shown in Table 5.1 considering Msg_3. And also assume that there were already two flags raised in the NSV, which are voltage flags as shown in Figure 5.3, due to errors detected from execution of the previous received messages (Msg_1 and Msg_2) also shown in Table 5.1. Therefore during the execution of the Msg_3 all voltage error flags must be cleared since the VSOC is greater than 70% and it is within the limit of the voltage range (0% - 100%).

Table 5.1: Telemetry Response Messages for Voltage Channel

CAN Msg	**Msg Prperties**	**Msg_1**	**Msg_2**	**Msg_3**
Msg_type	TLM_ACK	TLM_ACK	TLM_ACK	TLM_ACK
Channel	VSOC	VSOC	VSOC	VSOC
Source	PowerSub	PowerSub	PowerSub	PowerSub
Dest	PowerMan	PowerMan	PowerMan	PowerMan
DLC	1	1	1	1
Data	0% - 100%	65	105	90

The invalid data flag can be detected first, since searching start from LSB of the NSV as shown in Figure 5.3. The node flag is set to 1 which identifies the invalid data for a voltage flag (or VSOC) in the NSV. The $setNotClear$ flag is set to 0, and this flag will be assigned in the flag of the NSV indicated by the node flag. The execution process

returns from the function handler, without executing the message thoroughly, and then updates the flags. The main purpose is to clear all raised flags, from NSV and error recovery flags, related to the received message.

When the message is executed for the second time (Phase_2), the invalid data flag will be cleared already. The VSOC less than 70% flag will then be detected and the same procedure as for invalid data will be conducted except that the node flag will be assigned by the value of 2, refer to Figure 5.3. The third time (Phase_3), there will be no error related to this message, and then the message execution is conducted thoroughly by the function handler. This is illustrated in Table 5.2, where Msg_1 and Msg_2 are executed once since the error was detected during their execution, but Msg_3 is executed more than once since it is error free and there were already errors related to this message therefore they need to be cleared.

Table 5.2: Phases For Message Execution

CAN Msg	**Phase_1**	**Phase_2**	**Phase_3**
Msg_1	signalError = V_less_70 nodeFlag = 2 setNotClear = 1		
Msg_2	signalError = invalid_data nodeFlag = 1 setNotClear = 1		
Msg_3	nodeFlag = 1 setNotClear = 0	nodeFlag = 2 setNotClear = 0	nodeFlag = -1 setNotClear = 0

5.3 Updating Flags

It is noted that the node status flags can only be updated if the node flag was raised during the process of message validation and execution; refer to the Figure 5.1 which also indicates an entry condition into the process of updating flags. The node flag can be raised when any of the two following conditions are met; when the error is detected

in the received message or there is a raised flag in the NSV which is related to the received message and the message is error free.

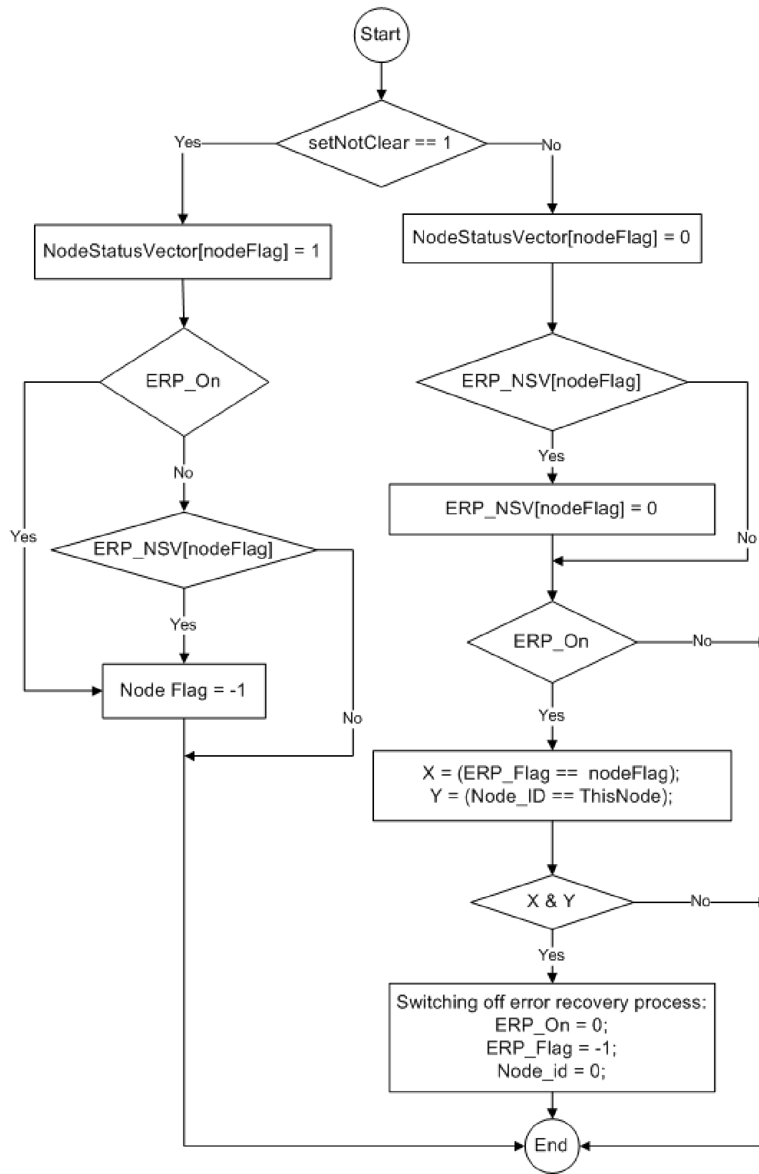


Figure 5.4: Flags Updates Flowchart (Updates Flags refer to Figure 5.1)

The NSV only changes a single flag at a time with a flag assigned to $setNotClear$ flag, in the position specified by the node flag. If the $setNotClear$ flag is set, this means the error was detected in the received message, otherwise the already existing error flag has to be cleared or removed from the NSV. This is shown in Figure 5.4. The number of errors that are still active in the node manager can be determined by counting the number of flags raised.

The NSV is defined as 16 bits, hence the node manager can represent up to 16 different errors. If the node manager is able to detect and handle more than 16 errors, then the NSV must be defined as a long integer (32 bits). Since the error log file and housekeeping data file are defined to record the 16 bits for the NSV, these files must also be defined to record 32 bits of NSV, if changes were made on the NSV.

There is a flag that can be triggered through NSV and ERP_Flag, and this belongs to the vector defined as ERP_NSV vector. The ERP_Flag records the flag of the NSV that has initiated the ERP, but only if the ERP is still running ($ERP_On = 1$), otherwise it is always set to -1. The ERP_NSV vector indicates the flags of the NSV that have already initiated the ERP and remained uncleared. This prevents the ERP to be executed more than once for the same flag in the NSV, even if the flag was not cleared during the ERP.

Every flag represented in the ERP_NSV corresponds to the flag defined in NSV. The Figure 5.5 illustrates the relationship between the two vectors, NSV and ERP_NSV. The second flag is raised in both vectors to indicate that the error signal exists even after the ERP has been conducted. The third flag is raised only in the NSV, and this indicates that the error has been detected, but the ERP is not yet complete (if $ERP_Flag = 2$) or is not yet initiated.

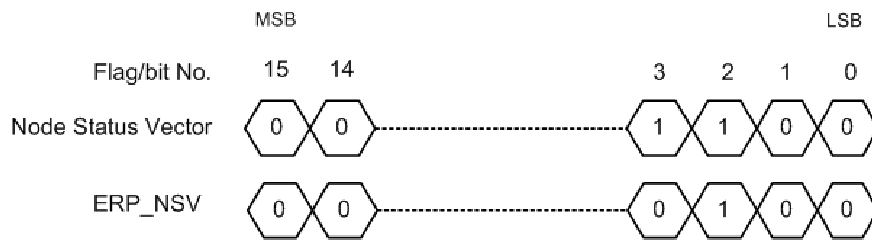


Figure 5.5: NSV and ERP_NSV

The ERP_On flag is defined as a global variable, which means every existing node manager can update the flag, but it can only be the one node manager at a time which is executing received messages. The address of the node manager is recorded when it is initialising the ERP and this address is used to allow the same node manager to switch

off the process and this is discussed in section 5.5. The rest of the flags are defined as private data and are only accessible within the node manager module, as indicated in section 2.1 that the thread can also have private data and global data as well.

When an error was detected ($setNotClear = 1$), the node manager can ignore the error only if the ERP is already on ($ERP_On = 1$) or it has already been conducted for the same flag ($ERP_NSV[nodeFlag] = 1$). The error can be ignored by assigning a value of -1 to the node flag, ($nodeFlag = -1$) as shown in the Figure 5.4. This is due to the next possible process after updating flags which is to load error recovery commands. The Figure 5.1 indicates that the process for loading error recovery commands can be initiated only if the node flag has an existing flag other than -1, otherwise the feedback of the message execution status is sent back to the command manager to signal the command manager that it (node manager) has thoroughly executed the received message.

When the flag in the NSV is cleared, the related error flags are also cleared if they were already raised. This is also shown in Figure 5.5, where the error was not detected in the received message, but there is a raised flag detected in the NSV. The $ERP_NSV[nodeFlag]$ is cleared to enable the ERP to be conducted again for the same flag if an error is detected. The ERP_On flag is based on the ERP and it is cleared if the two following conditions are valid. Firstly, the node manager that initiated the process is the one that is currently executing the received message, ($Node_ID = ThisNode$). Secondly, the flag that also initiated the process is the same as is cleared in the NSV, ($ERP_Flag = nodeFlag$). This is conducted to switch off the ERP because the error no longer exists.

When the received message is error free, all related flags will be cleared until there is no remaining raised flag in the NSV that is related to the received message. The Table 5.2 also indicates the last execution phase of the error free message where a node flag is assigned a value of -1 ($nodeFlag = -1$). This yields the execution process into the monitoring of the NSV, instead of updating flags, as indicated in Figure 5.1, and the following section provides more details about the necessity and execution procedure.

5.4 Monitoring The NSV

During the execution of every received message by the node manager, it has been noted that only one error signal can be detected at a time, but not every detected error signal can result in initialising ERP. This may happen when the error is detected while the ERP is already switched on for the same or different error flag. Monitoring of the NSV is necessary to identify any error flags since they can be harmful to the satellite system if they remain ignored or unattended. This section provides a brief overview of the monitoring process of the NSV.

The monitoring process is initiated when two of the following conditions are valid. Firstly, if there is no error detected during the execution of the received message. Secondly, when there is no flag remaining in the NSV that needs to be cleared. This is an entry condition to this process, as illustrated in Figure 5.1. This process can only be conducted thoroughly if two of the following conditions are met. Firstly, if the NSV has a raised flag so that the process can detect an existing flag. Secondly, if the ERP is not running, so that it can initiate the ERP once the flag is detected. This is shown in Figure 5.6.

During the monitoring of the NSV, one flag is examined at a time until the flag that has not yet initiated the ERP is detected or all the existing flags have been verified. If the flag which has not initiated the ERP is determined ($ERP_NSV[X] = 0$), then it is recorded to the node flag ($nodeFlag = X$). This will result in initialising the ERP. Refer to Figure 5.1, for the identified flag of the NSV and this process is explained in the following section.

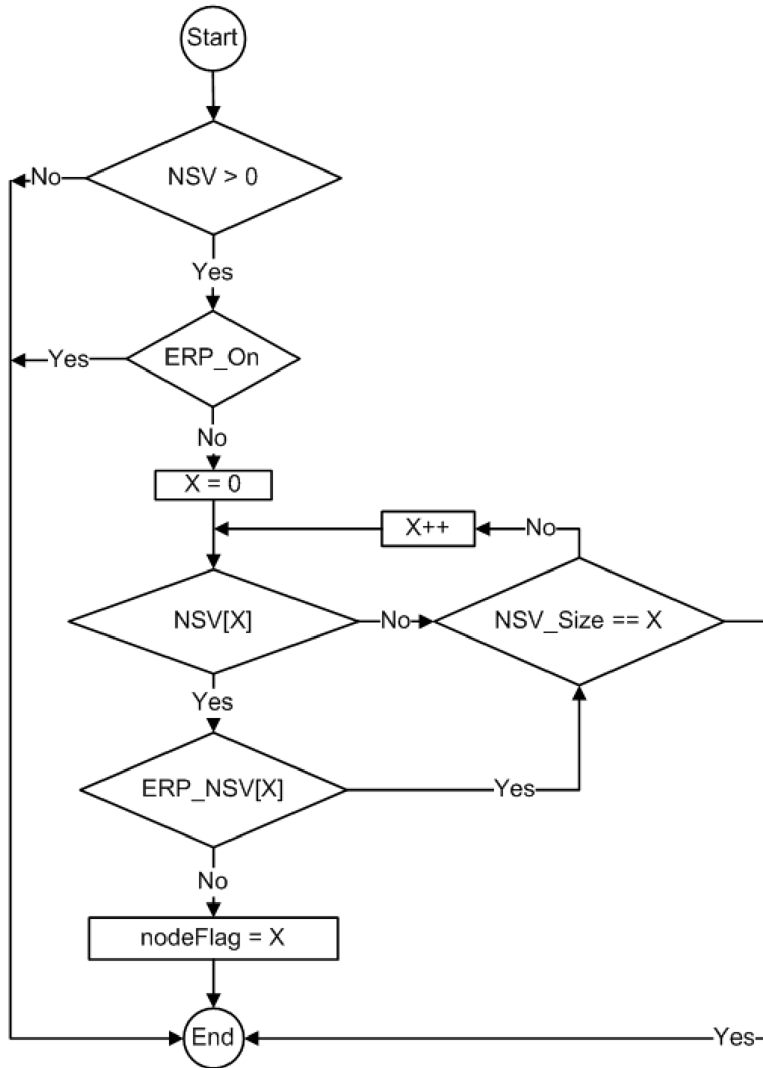


Figure 5.6: Monitoring The NSV (Extracted From Figure 5.1)

5.5 ERP

Section 2.5.2 describes the means of recovering the system from the detected error(s), and this can also be defined as ERP. There are error signals which are anticipated by the flight software and a set of commands which must be designed for ERP, whereas others can be conducted by simply resetting the respective node manager and satellite node which has experienced the error. The ERP is executed when the node manager has finished loading commands from an error recovery diary file into an error recovery scheduler, as described in section 3.3.2.2.

The error signal can be detected due to the execution of received message. It has been noted that the monitoring of the NSV is only conducted when there is no error detected in the received message, as discussed in the previous section, in order to identify a flag (if any) that has not yet initiated ERP. Therefore the error resulted from the current received message will get the highest priority. The ERP can be switched on for a single error at a time. The process is initiated if it is not running and the node flag is set according to an existing flag in the NSV ($nodeFlag$!= -1), as indicated in Figure 5.1.

This ERP can be terminated by the node manager, even though the error recovery scheduler is not empty, when the error is removed during the process. Otherwise the process will be terminated when the command manager has finished executing all the commands loaded in the error recovery scheduler. If the flag remains in the NSV ($NSV[ERP_Flag] == 1$), after the ERP has been executed for the same flag, then the $ERP_NSV[ERP_Flag]$ is raised. This prevents the flight software from executing the ERP for the same flag.

Once the flag in the NSV is cleared ($NSV[nodeFlag] == 0$), then the $ERP_NSV[nodeFlag]$ is also cleared. This enables the flight software to execute ERP for the same flag if the error is detected again. This can be observed in the case of transient faults, where the system goes back to normal state without recovery mechanism as discussed in section 2.5.1. The telemetry response messages to identify those types of changes, and then clears the flag in the NSV if the error is no longer detected.

For any error that the node manager can detect and the NSV represents, there is a defined start address and maximum number of commands to read from the error recovery diary file. Therefore each node manager must have its own NSV and error recovery diary file as stated in section 3.3.1.2. The node manager reads one command at a time and loads it into the error recovery scheduler, until the maximum number of error recovery commands has been reached. Thereafter the node manager sets the error recovery flags by recording the node flag and node manager identifier which represents the flag of the NSV and node manager that initiated the ERP, $(Node_ID = ThisNode)$ and $(ERP_Flag = nodeFlag)$.

The ERP is not anticipated to run regularly, unless the nanosatellite is not stable. Once the error recovery scheduler is loaded, the command manager should execute their commands immediately since the error recovery scheduler has the highest priority compared to other command schedulers, refer to Table 3.3. This is necessary since the error can be detrimental to the entire satellite system if it does not take immediate action to recover from the error, as described in section 2.5.1.

The ERP developed for the flight software is defined as a forward recovery mechanism, as explained in section 2.5.2, since it enables the system to continue operating from an erroneous state by making selective corrections to the system state. This was simple to develop, with the exception of making provision for returning to the previous state where the system was operating successfully as defined by backward error recovery.

The flight software application uses two levels of fault tolerance, graceful degradation and fail safe, as discussed in section 2.5.1. The execution of error recovery commands, when the error is detected, is degrading the system performance since the normal execution of commands from hard and soft deadline schedulers is halted until the system finishes the ERP. The application also fail safe when the error is detected and there is no commands reserved to recover from detected error. This is conducted by resetting the satellite node in which the error is detected. The node manager is reset by the thread manager, when the node manager takes too long to finish executing the received message, until timeout expires, as discussed in section 3.2.1.

Chapter 6

Flight Software Testing And Results

This chapter consists of three sections, firstly it explains the procedure used for implementing and testing the flight software application, and then it discusses the results observed during the course of testing. Lastly it provide the discussion about the speed at which the commands are executed including the response messages.

6.1 Implementation And Testing

The flight software application was not tested on eCos, in the OBC, as illustrated in Figure 1.1. This was due to the fact that the nanosatellite is still in a development phase, hence some of its parts are not yet completed (such as configuring eCos to run in the OBC) and some are not yet defined.

The flight software, according to this thesis description, is developed and tested in linux OS as shown in Figure 6.1. The figure also indicates the developed CAN module and satellite nodes which represent the CAN bus and satellite nodes of the nanosatellite, and PC is used instead of the OBC. The implementation of each module is discussed in the following section.

6.1.1 Flight Software And Testing Modules

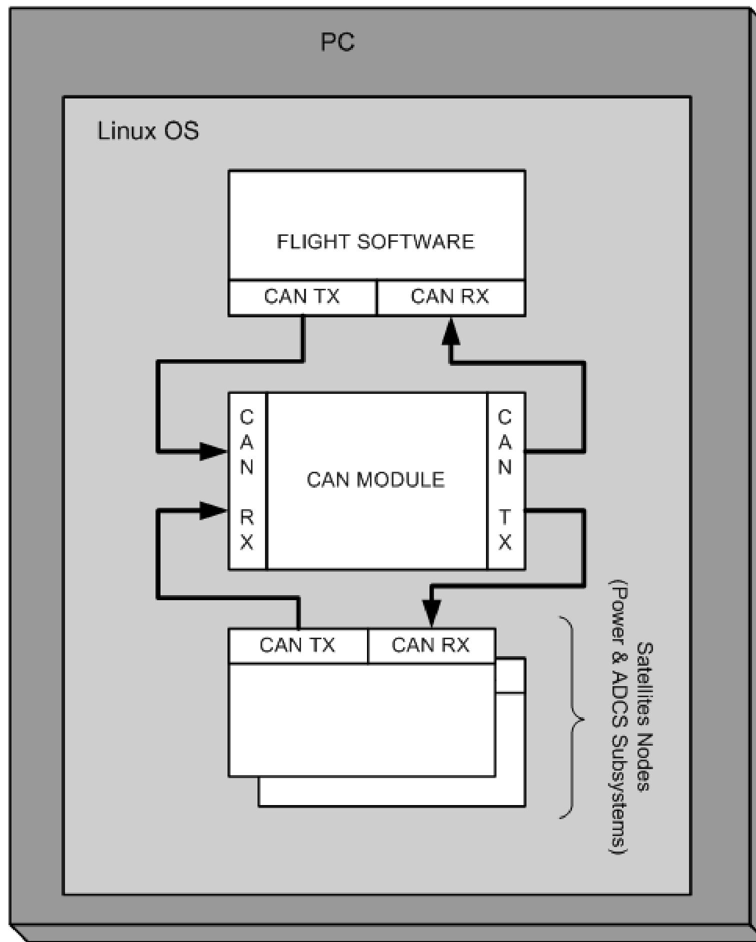


Figure 6.1: Flight Software Testing

6.1.1.1 Flight Software

This is the developed application on which this thesis is based. The command manager was developed and implemented, as discussed in chapter 4. It uses messages and commands to communicate with the satellites nodes via the CAN module. The power and ADCS managers were developed as part of the flight software to investigate and verify the algorithm for developing the node manager as discussed in chapter 5.

6.1.1.2 CAN Module (CAN bus emulator)

This was developed to provide the communication link between the flight software and the satellites nodes. Every module connected to the CAN module is commonly defined as a node, unless it is explicitly mentioned, and each has a unique address. The flight software is also considered as a node with an address of any even number. The CAN module allows messages with a destination address of an even number to be downloaded by the flight software, and this is discussed in section 3.3.1.1 page 31 where each node manager is recommended to have an address of an even number.

The CAN module accepts messages from the source nodes using a receive buffer (CAN RX), and then transfers them to a transmit buffer (CAN TX) so that they can be downloaded by the destination nodes. Each buffer can hold up to four messages, and both can conduct (receive or transmit) only one message at a time. Once the message is downloaded by the destination node, then the transmit buffer is free to load another message from the receive buffer if it is available.

It may sometimes happen that the destination node takes too long to download the message from the CAN module, for instance if the node is switched off. If the CAN module timeout expires before the destination node downloads the message, the CAN module changes the message identifier by interchanging the source and destination address and also updates the message type and data field to indicate that the original message was not delivered to the desired destination node. The source node of the message can then download the message from the transmit buffer because the destination address is changed into its original message source address.

The CAN module was developed to provide the same application of the CAN bus protocol designed for a nanosatellite as discussed in section 2.6. The only difference is that the CAN module can carry up to 4 messages at a time whereas the CAN bus protocol can only carry one message at a time. With the CAN bus protocol, if there is more than one node requesting to transmit the message, only the message identifier with a lower numerical value will be transmitted. The other message waits until the bus is free and then it is automatically re-transmitted in the next bus cycle, unless another

higher priority message has requested to be transmitted as well. The CAN module was developed to avoid holding an unlimited number of messages and also allowing more than one receiving node to download messages to free the transmit buffer.

6.1.1.3 Simulated Satellite Nodes

These are the simulated software modules developed to simulate the response messages that can be attained from subsystems and payloads of the nanosatellite, and they are power and ADCS subsystems. Each satellite node runs independently and communicates with a node manager in the flight software, power and ADCS managers respectively. Each satellite node is given a unique address, of an odd number to avoid conflict with the messages destined to the flight software node managers.

Once a satellite node receives a command from flight software via the CAN module, it executes the command and then sends a response back to the flight software via the CAN module. The execution and response of each command is conducted according to the command description, such as telemetry and telecommand request, as described in section 2.6 and 3.3.2.3.

For instance, once the power subsystem is switched on (or created since it is developed as a thread), the voltage regulates constantly after every 0.5sec by a difference of 5V from 110% to 0%, and 0% to 110%. Even though the defined valid range is between 100% to 0%, this is intended to be recorded by telemetry (voltage) response messages, and the power manger must identify and recover from such an error. This error is not removed by a recovery mechanism, since the power subsystem constantly regulates voltage and it eventually falls into a defined valid range. This can be defined as transient faults as discussed in section 2.5.1.

If the power subsystem receives a telemetry (voltage) request message, it will send a current VSOC to the power manager as a telemetry (voltage) response message, regardless of the VSOC. This symbolises the power supply that can be attained from the batteries of the nanosatellite as they recharge using solar panels exposed to sun

rays, and also discharged during the absence of sun rays as the power is also drawn by other satellite nodes that are switched on.

6.1.2 Flight Software Testing

The autonomous, mission and error recovery diary files were generated, according to section 3.3.1.2, for the flight software application to handle them appropriately as discussed in section 3.3.2. The samples of two commands extracted from each diary file are shown in Table 6.1, and the numerical values of the message identifier, DLC and data fields are also defined in Table 6.2.

Table 6.1: Sample for Commands used to Generate Diary Files

File Name	**Message Identifier**	**DLC**	**Data**	**Exe_time, repeatValue, timeInterval, diary ID**
Autonomous	4, 17, 2, 1	0	0	2, 255, 10, 1
Autonomous	4, 21, 4, 3	0	0	4, 22, 4, 1
Mission	1, 17, 2, 1	1	105	10, 0, 0, 2
Mission	1, 22, 4, 3	2	2, 0	9, 3, 10, 2
Error Recovery	4, 17, 2, 1	0	0	0, 0, 0, 0
Error Recovery	1, 19, 2, 1	1	1	0, 0, 0, 0

Table 6.2: Definitions of Message Identifier, DLC and Data Fields From Table 6.1

Message Identifier	**DLC**	**Data**
TLM_Req, VSOC, PowerMan, PowerSub	0	0
TLM_Req, SUN_RAYS, ADCS_Man, ADCS_Sub	0	0
TC_Req, VSOC, PowerMan, PowerSub	1	105%
TC_Req, Satellite_Direction, ADCS_Man, ADCS_Sub	2	SOUTH, 0
TLM_Req, VSOC, PowerMan, PowerSub	0	0
TC_Req, SWITCH_OFF, PowerMan, PowerSub	1	ADCS_Sub

The autonomous diary file is composed of telemetry request (TLM_Req) messages, designed to be executed several times and others indefinitely, such as voltage and current

requests for power subsystems (Power_Sub) and various sensors for ADCS subsystems (ADCS_Sub). The voltage, current and sensors are simply flags that constantly change their states according to the respective subsystem design. For instance, a voltage controlled by the power subsystem as discussed in the previous section. The telemetry messages keep on requesting the current state of flags, as they change over time. The telemetry respond messages must be executed by the respective node managers in the flight software.

The mission diary file is composed of telecommand requests (TC_Req) with different execution times, for changing various flags and sensors in the satellite node. This is anticipated to be detected by the execution of telemetry messages; the telemetry response messages should have the new status flags. The error recovery file is composed of the telecommand and telemetry request messages such as switch off the ADCS subsystem when the VSOC is less than 65%. These commands can be executed when the telemetry response messages deviate from their nominal value. It is the power manager (PowerMan) that has error recovery diary file, not ADCS manager (ADCS_Man), since this was sufficient for testing the ERP mechanism employed by the flight software. Nevertheless, in a real system each node manager should have its own error recovery diary file as recommended in section 3.3.1.2.

Every command or message introduced in the diary files, the properties of the message and a function handler to execute the message are defined to the respective node manager, as discussed in section 5.2, and also to the respective subsystem.

Some commands in diary files were designed sharing the same execution time, to test the scheduling and dispatching of commands employed by the flight software as discussed in section 3.3.2. The satellites nodes were sometimes controlled to respond with inconsistent data, invalid message identifier (message type or channel) or data (data range or data length). This was intended to verify whether the flight software is able to handle undefined messages and unpredicted errors.

The satellite nodes were also controlled to send telemetry response messages, without receiving a telemetry request messages from the flight software. This application is also

provided by CAN bus protocol for unsolicited telemetry messages, since this protocol can keep on sending those messages into the satellite nodes without interacting with the flight software. Refer to Appendix A for definition of unsolicited telemetry messages. This examines the reaction of the flight software towards the messages received unexpectedly, even though their properties are defined within the flight software.

The flight software application was executed for an hour, to examine if the application does not deteriorate as a function of time. The system setup was organised to ensure that the candidate can examine the performance of the flight software as described in the following section.

6.2 Results And Discussion

This section provides the results observed, and a discussion about the application behaviour, during the application testing. This is based on the manner in which the flight software dispatches commands, and means of handling various response messages.

6.2.1 Command Execution

The command manager validates and schedules commands according to the execution time and the manner in which they are read from the diary files. The execution time with the lowest value is given a highest priority and also the command that was read first from the diary file only if they share the same execution time. The command manager schedules one command at a time until the end of a diary file is reached. All diary files are scheduled at the beginning of the flight software execution, except for the error recovery file which is only scheduled by the node manager when the error is detected and ERP is not running. The command manager sends one command at a time, and the dispatching of commands from schedulers is prioritised as illustrated in Table 3.3, even if every scheduler has a command ready for execution.

When the subsystem is switched off, it is the commands addressed to that subsystem

that are not sent by the flight software, but they are rescheduled like any other command if the repeat value is greater than 0. If the repeat value is set to 0, the command is removed from the scheduler even if it was not sent successfully into the respective subsystem. This was observed for commands that have hard deadlines, since once they are not executed at specified time, this means that the application has failed and it cannot recover from an undesired state. The rest of the commands with the same diary identifier are also deleted from the hard deadline scheduler, since they might depend on each other to perform a required task.

All command schedulers are sometimes loaded with commands ready for execution, for instance when the ERP is on, while the soft or hard deadline scheduler also has a command ready for execution. The flight software has observed dispatching commands according to the priority of the command scheduler as shown in Table 3.3. It is noted that the commands that experienced a delay are from autonomous diary files, since they are scheduled to the soft deadline scheduler, which is given the lowest priority. This is still permissible since the autonomous diary file cannot fail the system because the satellite status can be requested at any time.

The flight software application was observed dispatching commands and also executing response messages accordingly for a long time without deteriorating, especially commands from autonomous diary file that were defined to execute indefinitely, until the user terminates the program.

6.2.2 Handling Response Messages

The flight software was observed handling the messages successfully including messages received unexpectedly from satellite nodes. Since the command manager sends one message and waits for a response, it always compares the received message with the recently transmitted message. There is a defined response message for every command sent to the satellite nodes as discussed in section 2.6.2. If the received message is not an expected response then it handles the message like any other received message and then waits for a response message, since the recently received message was not

considered as a response message for the transmitted message. The command manager waits, until the desired response message is received or timeout expires.

If the timeout expires, before the response message is received from satellite nodes, it is assumed that the execution error is in the satellite node. The CAN module is designed to send a response back to the source node of the message it has received, if the timeout expires within the CAN module before the addressed node has downloaded the message. The node manager resets the respective satellite node once it receives a response message from the CAN module indicating that the transmitted message was not delivered to the desired satellite node.

The node managers were evaluated and observed as they were executing the response messages from the satellites nodes. Node managers were able to respond to the satellite node by loading the message into the MSQ. Once the command manager retrieves a message from the MSQ, it immediately sends it to satellite nodes without scheduling it, even if there were commands ready for execution in the command schedulers. This is safe even during the ERP, since the error recovery commands do not generate commands or response messages to send back to the satellite nodes and they are designed to be executed for a short period to return the program to normal execution.

During the execution of an autonomous diary file, the flight software was observed recording variable data into the housekeeping data file. The node manager records the current node status flags, before executing the received messages. After the message is thoroughly executed, the new status flags are compared with the previous status flags. If there is a difference, then the housekeeping data flag is raised and the command manager logged into the housekeeping data file as discussed in section 4.1.2.1.

The command manager was able to create another file, the housekeeping data file or error recovery log file, only when the current file has reached a defined file size. The files were created but not downloaded as expected in real application; therefore the command manager clears the first created file if the current files have reached the defined maximum number of the file. The next files will be cleared if the current file is full. This is also necessary because the program will not run out of memory, even in

real application if the ground station takes too long to download them. The samples of these files are shown in appendix C.

The ERP was conducted after an error was detected, due to invalid data range or data length, by the node manager that is executing the message. During the ERP, the error log file was always updated by the command manager after the node manager had finished execution of the received message. If the error that initiated the ERP was removed during the course of its recovery process, the ERP was terminated and the remaining commands from the error recovery scheduler were deleted. For instance, when the voltage status flag is out of range and becomes stable once the ERP has been initiated for the error.

The ERP was conducted once for each detected error, even if the error remains after the ERP was completed, and then returns the program to normal execution. The errors were also removed due to telemetry response messages, as the satellite node status keeps on changing as a function of time. This means that the application implements fault tolerance, as described in section 2.5.1, since it returns the program to normal execution even though the error still remains after ERP.

6.3 Software Time Response

The flight software was observed sending commands to the satellite nodes through CAN module and then execute the response messages as discussed in the section 6.2. This section provides the discussion about the time delay, counting from the time at which the command manager sends the command to the CAN module until the command manager becomes ready to send next command.

This was tested within the Linux OS, as stated in section 6.1, which is running within a virtual machine installed on a Windows XP machine and this is loaded with many applications with some running in background. The Linux OS clock resolution is fixed at 1 MHz, and this was obtained by calling *clock_getres()*, which limits the time

resolution to 1 microseconds.

The command execution cycle is measured by recording the time (current time in seconds and nanoseconds using *gettime()*) immediately when the command is sent to the CAN module and also when the command manager becomes ready to send the next command, and the time difference is considered as the command execution cycle. This involves every step of the command execution such as: the execution of the command in the respective satellite node, the response message in the respective node manager and also the time delay between the threads as they relay the command/message into the relevant thread. The maximum number of threads that are required to execute the command and its response message are four. The respective satellite node and node manager execute once whereas the command manager and CAN module are called twice to send the command and then to receive the response message.

Initially the command execution cycle time was ranging between 2 to 3 seconds. This was due to a major delay of approximately 0.5 seconds when the execution of the command/message is transferred from one thread to another. The satellite nodes and command manager were polling the CAN TX (shown in Figure 6.1) whereas node managers were polling the receiver buffer of the command manager.

This was observed as time consuming especially for real-time application. The condition variables provide an effective way for threads to synchronise instead of polling the signal which keeps the threads running. The mutexes implement synchronisation by controlling thread access to data, whereas the condition variables allow threads to synchronise based upon the actual value of data. The condition variables block the calling thread(s) using *pthread_cond_wait()* until the specified condition is signalled. The *pthread_cond_broadcast()* should be used instead of *pthread_cond_signal()* if more than one thread is in a blocking wait state. This was used by the CAN module to signal satellite nodes and the command manager if there is a new command/message in the CAN TX, and command manager to signal node managers if there is a new message response read from CAN module.

After the condition variables were introduced, the command execution cycle was drastically reduced to ±1.5 milliseconds. The command execution time varies mainly due to the time (approximately 0.12 milliseconds) it takes for a thread to gain access of the mutex, after the condition is met, only when there is more than one thread waiting for the same condition. This may also result from logging to housekeeping or error log data files and initiating or terminating ERP.

This is believed to be sufficient performance while considering the fact that this was not measured in the required environment. It is anticipated to be much faster once it is executing on eCos environment in the OBC, since the hardware and software would be specially designed to function at a very high speed as the satellite orbits the earth. If further improvements need to be made, they should be done on the eCos environment as the platforms are so different. Optimisations will therefore not necessarily have desired effect on both platforms.

Chapter 7

Conclusion And Recommendations

The flight software application was developed and tested as discussed in chapter 6. This chapter provides the conclusion of the overall application behaviour and the recommendations that can be considered to upgrade the application for operation in the nanosatellite in space.

7.1 Conclusion

The flight software is developed using C and POSIX standard functions that are compatible with eCos.

The flight software has the ability to validate, schedule and dispatch commands read from diary files. The command fields and diary files are defined simply enough for the flight software to handle them effectively, by prioritising commands that can fail the system if they are not executed at appropriate time. This enables the ground station to efficiently control the satellite, by generating the diary files and send them to the satellite.

The autonomous diary file is composed of telemetry request messages which are repeatedly executed by the flight software, and some are indefinitely executed until the

application is terminated, at a defined time interval according to each command field. The telemetry response messages were executed by the node managers to keep track of the satellites nodes behaviour, and detect the change in the satellite status. If the change is not detrimental to the satellite system, messages received and status flags are recorded in the housekeeping data file. Otherwise the error recovery commands, for the detected error, are executed to minimise the effect of the error in the system, and this is recorded in the error log file. This also provides the means for the autonomous operation, to monitor and control the satellite even though there is no interaction with the ground station.

Housekeeping data files and error log files provide the means for the user at the ground station to observe the satellites behaviour. These files are defined simply enough for the user to identify the type and course of the error, and the change in satellite status which indicates the behaviour of the entire system. The user at ground station can then generate the diary files, in response to the content of these files, to keep the satellite in a safe condition and enhance the performance of the satellite.

The node manager is defined such that if it enters unknown or unpredicted states, that the flight software has not yet provided the error recovery commands, the flight software will then reset the respective satellite node. This prevents the system from errors such as transient faults, as they are eventually removed without any effort from the satellite system, and this means that the flight software is fault tolerant.

The flight software sometimes experiences a delay in executing commands into the satellite nodes, when there is more than one command scheduler having commands ready for execution or a some commands share the same execution time. This may prevent the system to meet real time requirements. The application has divided the schedulers and prioritises them according to the types of the commands each scheduler can possess. Error recovery commands are given highest priority to recover the system from errors. Where as the commands with hard deadlines (from mission diary files) are given highest priority in the absence of error recovery commands. Then the autonomous diary files are given the lowest priority, since their execution delay will not fail the system application as the satellite status can be requested at any time.

The flight software was developed and tested to ensure that the objectives as outlined in section 1.2 are met, but this was not enough to examine the efficiency of the application, since the flight software was not eventually tested on the eCos (target operating system) in the OBC (developed satellite computer) as discussed in section 6.1. The reason for this is that the nanosatellite is still in a development phase, hence some of its objects are not yet completed and some are not yet defined. The following section provides the recommendation for future work that can be considered to upgrade the flight software according to the mission that is still yet to be defined.

7.2 Recommendations

Since the flight software is not fully developed into a package that is ready to operate the nanosatellite in space, this section describes some of the tasks that need to be completed to in order to achieve full integration of the flight software application so that it can be ready for full operation.

When developing the node manager for the specific satellite node, the node manager needs to be defined according to the type of commands and messages that the respective satellite node could handle. For every message that can be received by the node manager, the function handler must be developed and message properties must be defined in order to validate and execute the message accordingly. This enables the node manager to identify errors due to telemetry response messages, and recover immediately before it affects the entire system. If a new command is introduced to the application, the candidate has to define the command properties and possible message responses in order to have a fully integrated satellite node and its respective node manager.

The housekeeping data file or error log file can be modified by adding more fields if necessary and the program for writing to these files should be updated according to the field's description. The program at the ground station should be updated as well to anticipate the new added field when reading the files.

It has been noted that the flight software can detect the errors that have been defined and then initiate the ERP according to the type of error. They may be errors that can be detrimental to the satellite system, if they are not defined and also if there is no provision made for the ERP. During the development of node managers, care should be taken to accommodate all possible errors that it can experience and also provide the ERP for each type of error. Then the resetting of the satellite node and node manager can be a solution for unpredicted errors.

If there is more than a one command in the mission diary files with the same execution time, then the flight software cannot meet the real-time application and this may fail the system since the mission diary files are defined to possess commands with hard deadlines. Therefore the flight software can be upgraded to send more than one message without waiting for a response of the message sent, but care should be taken to keep track of the messages in order to identify those that have received response messages. The other alternative is that the user at the ground station should avoid compiling commands with the same execution time, especially when designing mission diary files. The autonomous diary file cannot fail the system, unless the delay is too long for the flight software to miss the detrimental effects in the subsystems. The ERP is initiated due to telemetry response messages which are received due to the execution of commands from autonomous diary files.

Since the flight software was not fully tested, a full test setup would be good for future development and considering the testing procedure discussed in section 6.1.2.

Bibliography

[1] S. Khumalo, *A CAN Based Distributed Telemetry and Telecommand Network for a Nanosatellite*, Master's Thesis, Department of Electrical and Electronic, University of Stellenbosch, March 2008

[2] N. L. Steenkamp, *Development of the On Board Computer Flight Software for SUNSAT 1*, Master's Thesis, Department of Electrical and Electronic, University of Stellenbosch, December 1999

[3] Hamed Sanago, *Implementing The CAN Protocol in 3G Equipment Design*, *http://www.commsdesign.com*, Last modified: December 2002

[4] eCos Referrence Manual, *http://ecos.sourceware.org*

[5] J. R. Wertz and W. J. Larson, *SPACE MISSION ANALYSIS AND DESIGN*, Microcosm Press and Kluwer Academic Publishers, year 1999

[6] Anthony J. Massa, *EMBEDDED SOFTWARE DEVELOPMENT WITH ECOS*, Prentice Hall. Inc., year 2003

[7] B. Blaise, *Posix Threads Programming*, *https://computing.llnl.gov/tutorials/pthreads/*, Last Modified: 19 May 2008

[8] H. M. Deitel and P. J. Deitel, *C How To Program, Forth Edition*, Prentice Hall. Inc., year 2004

[9] A. Silberschatz and P. Galvin, *Operating System Concept, Fifth Edition*, John Wiley and Sons, Inc., year 1999

[10] Shem-Tov Levi and Ashok K. Agrawala, *REAL-TIME SYSTEM DESIGN*, McGraw-Hill, Inc., year 1990

[11] Meredith Gibb, *http://imagine.gsfc.nasa.gov/docs/sats_n_data/sat_to_grnd.html*, Last Updated: Monday, 27-Sep-2004 11:26:10 EDT

[12] Musa J.D, A. Iannino, and K. Okumoto, *Engineering and Managing Software with Reliability Measures*, McGraw-Hill, 1987

[13] J. H. du Toit, *COMPUTER SYSTEM FOR THE SUNSAT MICROSATELLITE*, Master's Thesis, Department of Electrical and Electronic, University of Stellenbosch, December 1994

[14] Alan Burns and Andy Wellings, *Real-Time Systems and Programming Languages*, Ada 95, Real-Time Java and Real-Time POSIX, *Third Edition*, ADDISON WESLEY 2001

[15] Kevin M. Obenland, *The Use of POSIX in Real-time Systems, Assessing its Effectiveness and Performance*, obenland@mitre.org, March 2001

Appendix A

Protocol Supported Message Types

This section covers the message types that are already supported by the CAN bus protocol, and this ensures flight software application that when it implements the same message types the protocol will be able to handle them appropriately.

There are 14 message types already defined by the protocol as shown by the Table A.1. All message types except for debug and the time synchronization messages will be acknowledged and this acknowledgment will be a message type on its own. The xx symbols indicate an unassigned channel, source address or a destination address and these are message packet dependent. The third column gives a brief idea of what each message type does. The following discussion is based on the message type used during the flight software development.

- **Time Synchronization** This message type has the highest priority on the bus because time accuracy is an important parameter for the satellite applications like the attitude determination and control system (ADCS). It is needed To communicate accurately and to have a stable system time for all nodes to synchronize their UNIX time to a master clock. The master clock on the OBC or the GPS will broadcast the system UNIX time at regular intervals.
- **Telecommand messages** In all cases these messages must be less or equal to 8 bytes in the data field and no data fragmentation is required. Each message will

be acknowledged positively or negatively. There are 256 possible telecommand channels and only 15 standard telecommand channels are currently reserved for the node and the application program can expand the list to include user specific telecommand channels.

- **Telemetry messages** They would be handled the same as the telecommand messages above. The difference is on the data field; while the telecommand sends the command data the telemetry message will contain no data and it requests data from the destination node. There will also be 256 possible telemetry channels with 15 of those already defined as the standard telemetry channels.

- **Unsolicited Telemetry** The difference from normal telemetry messages is that unsolicited telemetry responds periodically for each request. To setup an unsolicited telemetry request, the repeat value and the repeat period must be specified. The repeat period will take 4 bytes (32-bits from LSB) of the 8 bytes data field and a further 2 bytes will determine the repeat value. If there is a need to cancel an unsolicited telemetry; a request with the same channel and source address must be made with a period of 0 or a repeat value of 0. To setup a request that repeats indefinitely a repeat value of 0xFFFF must be specified.

- **File and data transfers** This message types are used when a large file is transferred across the CAN bus; a file transfer header is first sent and it would contain the memory start address and the total data size of the file to be transferred. The control field specify what data action will follow after the header transfer has been acknowledged by the receiver. When the full data file has been transferred across, the receiver of the file will reassemble the whole file and ultimately act upon it based on the command that was received in the header.

- **Debug Messages** These lowest priority messages will be broadcast by a node which wants to send anything for broadcast information on the bus to all the nodes. This debug data will be less or equal to 8 bytes. The typical information in the debug message can be the node status, a node warning or debug information after a certain operation and will be used by any application

Table A.1: The Protocol Supported Message Types [1]

Message Type	**Identifier Range**	**Comment/Description**
Time Synchronization	0x0000xx00	Broadcast Unix Time
Telecommand Request	0x01xxxxxx	Command/Request
Telecommand Response	0x02xxxxxx	Acknowledgement
Telecommand Not Acknowledgement	0x03xxxxxx	Command failure Reason
Telemetry Request	0x04xxxxxx	Request
Telemetry Response	0x05xxxxxx	Response
Telemetry Not Acknowledgement	0x06xxxxxx	Failure Reason
Unsolicited Telemetry Request	0x07xxxxxx	Request for periodic response
File Header Transfer	0x08xxxxxx	Start File transfer
File Header Transfer Acknowledgement	0x09xxxxxx	Response to initiate file transfer
File Data Transfer	0x0Axxxxxx	Data packets
File Data Transfer Acknowledgement	0x0Bxxxxxx	Each data packet acknowledged
File Data Transfer Not Acknowledgement	0x0Cxxxxxx	Data packet lost
Debug Messages	0x0D00xx00	Broadcast string

Appendix B

Linked List

Linked List are the collection of data items "lined up in a row"; insertions and deletions are made anywhere in a linked list. Stacks are important in compilers and operating systems; insertions and deletions are made only at one end of the stack, top stack. Queues represent waiting lines; insertions are made at the back of the queue(referred to as tail), whereas deletions are made at the from the front of the queue (referred to as head). [8]

Self referential structure contains a pointer member that points to a structure of the same structure type, and this is shown in the example below.

```
struct node {
        int data;
        struct node *nextPtr;
};
```

This is a structure of type *struct node* with two members; integer member *data* and pointer member *nextPtr*. Member *nextPtr* points to a structure of type *struct node*, a structure of the same type, and is referred to as a link which can be used to tie a structure to another of the same type and this formulate a linked list.

Linked list is a linear collection of self referential structures connected to each other by pointer links, and the linked list is accessed using a pointer links from the beginning until the end of the linked list. The end of the linked list is determined when the pointer links is pointing to NULL, meaning it does not points to another node. The linked list become full when the system has insufficient memory to satisfy dynamic storage allocation request, and it is empty when the pointer to the first node of the list is NULL. Insert and delete function are the primary function of the linked list, sine they alter the list. [8]

B.1 Inserting A Node In A Linked List

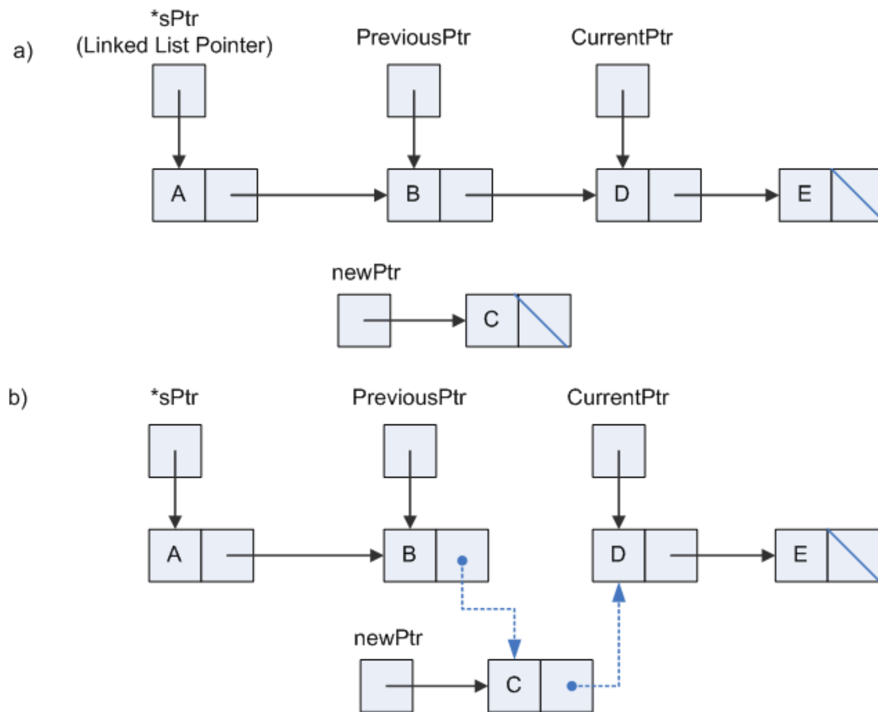


Figure B.1: Inserting a node in order in a list [8]

The Figure B.1 illustrate the linked list of characters inserted in an alphabetic order part *a*, and also shows how to insert a new node to the list part *b*. The insert address receives an address of the list and a character to be inserted. Since the list itself is a pointer, passing the address of the list create a pointer to a pointer, **stPtr*. The steps

for inserting a character in the linked list are as follows: [8]

- Create a node by calling *malloc*, assigning to *newPtr* the address of allocated memory, new character to the *newPtr->data* and NULL to *newPtr->nextPtr*.
- Initialise *previousPtr* to NUll and *currentPtr* to **sPtr* (the pointer to the start of the list). The *previousPtr* and *currentPtr* store the locations of the node preceding the insertion point and after the insertion point.
- While the *currentPtr* is not equal to NULL and the value to be inserted is greater than *currentPtr->data*, assign *currentPtr* to *previousPtr* and then advance *currentPtr* to the next node in the list. This will eventually locates the insertion point.
- Then the insertion is conducted as shown in Figure B.1 (*b*). If the *previousPtr* is NULL, insert the new node as the first node in the list by assigning the **sPtr* to *newPtr->nextPtr* (new node link points to the former first node) and *newPtr* to **sPtr* (**sPtr* points to the new node). Otherwise, if *previousPtr* is not NULL, the new node is inserted in between or at the end of the list by assigning *newPtr* to *previousPtr->nextPtr* (previous node points to the new node) and *currentPtr* to *newPtr->nextPtr* (the new node link points to the current node).

B.2 Deleting A Node In A Linked List

The delete function receives the address of the pointer to the start of the list and a character to be deleted, and this is also illustrated in Figure B.2, part *a* indicate the list before a node is deleted and part *b* indicate deletion of the node. The steps for deleting a character from the list are as follows: [8]

- If the charecter to be deleted matches the character in the first node of the list, assign **sPtr* to *tempPtr* and *(*sPtr)-nextPtr* to **sPtr* (**sPtr* now points to the

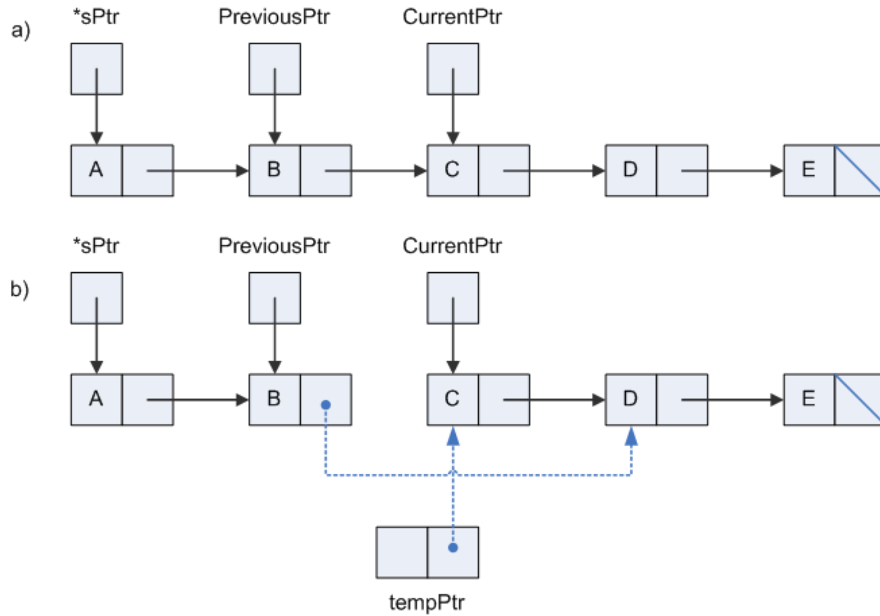


Figure B.2: Deleting a node from a list [8]

second node in the list). Then free the memory to by *tempPtr*, to avail the memory space to be requested again, and then return from the function.

- Otherwise, initialise *previousPtr* with **sPtr* and *currentPtr* with *(*sPtr)-nextPtr*.
- While *currentPtr* is not NULL and the value to be deleted is *currentPtr-data*, assign *currentPtr* to *previousPtr* and *currentPtr-nextPtr* to *currentPtr*. This locates the charecter to be deleted as shown in Figure 2.1 (*b*), only if it is on the list.
- If the *currentPtr* is not NULL, assign *currentPtr* to *tempPtr* and *currentPtr-nextPtr* to *previousPtr-nextPtr*, and then free the node pointed by *tempPtr* (to avail the memory space). If the *currentPtr* is NULL, signify that the character to be deleted was not found in the list.

Appendix C

Generated Data Files

The error log file is loaded with data when the application is in the ERP, where as the housekeeping data file is loaded with data when the satellite status data changes. This is recorded using a random access file, and it is intended to be downloaded into a ground station and converted into a human readable text such as text file used for in this application. The following sections show the information that was recorded and converted into text files.

It should be noted that this information (data/flags) are represented in a decimal format, and most of them provide meaningful information if they are in a binary mode. For instance, the log_time and msg_no are interpreted well in a decimal format whereas NSV and Flags are interpreted well in binary mode.

C.1 Sample for error log data file

This is a sample of the information recorded during a single log into error log data file. This indicates that every time the flight software logs into error log data file can record the information of this type/size.

msg_no = 1, tx_flags = 0, rx_flags = 0, log_time: 1229413639

TX command (all fields): msg_id: {4 17 2 1}, DLC= 0, buffer: {0},

exe_time = 1229413640, repeatValue = 0, TimeInterval = 0, diaryID = 0

Can Msg: rx_msg: msg_id:{5 17 1 2}, DLC = 1, buffer: {55}

previous RxNodeStausFlag 64 64 0

previous Flag_status_t: nodeManOrSub_onOff=2, node_id=2, errFlagNo=6, ErrRecExe=0, errRecMsgsSent=3

new RxNodeStausFlag {64 64 1}

new Flag_status_t: nodeManOrSub_onOff=2, node_id=2, errFlagNo=6, ErrRecExe=1, errRecMsgsSent=0

The execution time is always represented in absolute time once it is loaded into command schedulers, even if the command in a diary file is has a relative execution time. The log_time is in seconds and represent the UNIX time.

The RxNodeStausFlag represent three flags namely: NSV, NSV_ERP and ERP_on. The NSV indicates the errors that are still active in the system, whereas the NSV_ERP indicates the errors that have already initiated ERP and remain uncleared. The ERP_on indicates the status of the ERP, if it is raised this means that the ERP is running.

C.2 Sample for Housekeeping data file

The flight software can frequently logged into this file as the satellites changes its states as a function of time or orbiting the earth. Therefore the housekeeping data file is designed to record a small portion of data. This is a sample of the information recorded after the flight software has logged into housekeeping data file, msg_no 1 and 2 indicates the first and second logging into the file.

The change between this two information recorded is observed in the NSV. The first first flag indicates that the DLC is invalid, since it is anticipated to contain a value of 1

according to the voltage message properties. The second time the message is recorded the node manager has recovered from the invalid data length but detected an error which is the invalid data in the buffer field, the voltage range is within 0% - 100% according to the voltage message properties.

msg_no = 1, NSV = 1, log_time: 1229413587
can_msg: msg_id: {5 17 1 2}, DLC = 2, buffer: {105}

msg_no = 2, NSV = 2, log_time: 1229413590
can_msg: msg_id: {5 17 1 2}, DLC = 1, buffer: {105}