Faculty of Science and Engineering
School of Earth and Planetary Sciences

Pushing the Boundaries of Spacecraft Autonomy and Resilience with a Custom Software Framework and Onboard Digital Twin

Stuart Robert Gibson Buchan

This thesis is presented for the Degree of
Doctor of Philosophy
of
Curtin University

April 2023

Declaration of Authorship

To the best of my knowledge and belief this thesis contains no material previously published by any other person except where due acknowledgement has been made. This thesis contains no material which has been accepted for the award of any other degree or diploma in any university.

Stuart Buchan

Date: **April 18, 2023**

ABSTRACT

As the popularity of CubeSats as a means of conducting space research in universities continues to rise, a concerning trend has emerged. The small satellite missions these groups produce exhibit a high mission failure rate of approximately 51%. Coupled with an approximate failure rate of 32% from industry-led CubeSat missions, it highlights the lack of reliability in this class of spacecraft. A significant contributor to these failures is software error, accounting for approximately 35% of total failures, where university and industry groups fail to implement the complex software used in modern spacecraft safely. Moreover, the capabilities of the flight computers on CubeSats are often underestimated, resulting in inadequate fault prevention logic.

This research focuses on mitigating the risk of failure in space missions, with a particular emphasis on CubeSat missions. The issues arising from CubeSat missions provide an accessible database that quantifies critical areas that require attention to mitigate the risk of mission failure. These focus areas are equally applicable to larger-scale spacecraft, thus forming the central thesis of this work. A Learning From Incidents approach is used to identify recurrent and underlying factors contributing to software failures within historic space missions. Reliable software design principles are applied to satellite software development to prevent the reoccurrence of similar failures. Using Western Australia's first satellite, Binar-1, as a case study, the thesis demonstrates the efficacy of these principles in creating safe, reliable, and reusable software for the Binar platform. These principles can be applied to any mission, significantly increasing the probability of success.

Despite reliable programming efforts, operational failures during a space mission are still possible. The current approach to failure prevention is to transition the satellite into safe mode and await instructions from ground control. However, as spacecraft venture deeper into space, relying on ground communication to recover from a failure will become increasingly difficult to support, given overextended deep space network facilities and a rapidly increasing number of missions. To address this issue, a digital twin of the Binar-1 CubeSat, capable of running onboard the memory-constrained flight computer, is developed in the thesis. The digital twin platform has fault prediction capabilities, as demonstrated by a thermal model of the CubeSat battery heating system. Tasking the digital twin platform with identifying and preventing a power fault in advance shows that the digital twin can autonomously predict and prevent low-power faults without input from ground control.

The research presented in this thesis holds significant implications for satellite systems engineering by offering a solution to the high failure rate of spacecraft software. Software is critical in ensuring satellite mission success, regardless of size or class. By pushing the boundaries of satellite resilience and autonomy through software, this research paves the way for improved efficiency and effectiveness of mission operations, facilitating discoveries and advancements in satellite systems engineering. Following the successful deployment of the software framework on the Binar-1 mission, it is now being integrated into the Binar-2, 3, and 4 satellites, along with the novel onboard digital twin to undergo testing on orbit.

Though I'm past one hundred thousand miles

I'm feeling very still

And I think my spaceship knows which way to go...

- David Bowie

Acknowledgements

This research was supported by the Australian Government through the Research Training Program (RTP) scholarship, the Western Australian State Government through funding provided to the Binar Space Program, and institutional support from Curtin University.

I would like to gratefully acknowledge the guidance and support I have received throughout my research and thank all those without whom this thesis would not have been possible.

- **Fergus Downey** – You're an expert in building satellites and a master in turning complicated electronics into simple caveman-friendly instructions. Without your contribution to the hardware, Binar-1 would have remained just a dream floating in the stratosphere. I will fondly reflect on the memories of our late-night troubleshooting sessions and the afternoon beers we shared as we embarked on our journey to make history. Here's to you, my caveman friend; let's raise our clubs to finally finishing our PhDs!
- **Ben Hartig** – Ben, thank you for your contributions to the success of the Binar Space Program. Your tireless efforts behind the scenes were critical in ensuring that Binar-1 was successful. You handled the paperwork and administrative tasks that are often the least fun side of engineering, and for that, I am grateful. Without your leadership, we would have been lost in space. I wish you all the best in your future endeavours.
- **Jonathan Paxman** – Your encouragement and mentorship during my undergraduate years, particularly when you steered me towards working with the DFN, have forever changed the course of my academic and professional life. Working in the space industry has always been a dream of mine. Without your assistance

opening that door, I may never have pursued it as a career. For that, I am very grateful.

- **Phil Bland** – Phil, your never-ending stream of ideas made me feel like I was playing academic whack-a-mole - every time I thought I had a solid PhD topic, another one of your brilliant ideas popped up and made me change my mind (three times!). Your boundless passion for science is contagious, and I couldn't help but be swept up in your enthusiasm. Our regular chats were like a rollercoaster - thrilling, occasionally terrifying, but always worth the ride. Your tenacity in securing funding for science is nothing short of heroic. You fought the good fight to bring in the cash that made launching Binar-1 possible, and for that, I am grateful.

- **The Binar team** – We've grown a lot from the space program's early days, when our team was just a handful of brave pioneers dreaming of launching a satellite, to the present day, where our team has grown and thrived, bringing together some of the most passionate minds in the industry. You have been an integral part of this project. Thank you for taking the pressure off me by working on the Binar-2, 3, and 4 satellites, allowing me to focus on my PhD. Your contributions to the project are immeasurable, and your friendship is invaluable. I look forward to seeing all we will achieve together in the future.

- **My parents: Shona and George** – You've been my biggest supporters, cheerleaders, and champions. From our earliest nights in the back garden looking up at the stars, you've always nurtured my curiosity and encouraged me to pursue my dreams. When I was torn between pursuing acting or engineering, you didn't force me in any one direction, but you gently nudged me towards the path that would ultimately lead me here, to this moment. Your unwavering belief in me and your constant expressions of pride have guided this journey. I am forever grateful for your love and encouragement. You may not always understand the intricacies of my research, but your willingness to listen and to share in my enthusiasm means more to me than words could ever express. Without your sacrifices, love, and support, I would not be where I am today. Thank you from the bottom of my heart for everything you have done and continue to do. I love you both more than words could ever say.

- **My grandfather: Bob** – To my granddad, whose legacy of curiosity, ingenuity, and perseverance lives on in everything I do. You had to leave school at a young age to support your family, but you never lost your love of learning and never let your circumstances dim your thirst for knowledge. You were my first teacher, showing me how to use a screwdriver and sparking a love for engineering that would shape my life's path. Even though we were separated on either side of the Earth, you were always there for me, cheering me on from afar, asking about my research, and sharing my successes. Your memory will always be a source of inspiration for me, a shining light that guides me on my journey.

- **Emily** – I am immensely grateful for your unwavering support throughout my PhD journey. Your kindness, patience, and encouragement have been invaluable to me. Thank you for being there to listen to my complaints on bad days and for sharing in my excitement whenever I made progress or achieved a milestone. You have been a great sounding board for my ideas, even when they were bogged down in techy details. It will be a welcome change to finally have a conversation that has nothing to do with space or science! Thank you for being a constant source of comfort and support during this challenging journey.

CONTENTS

List of Publications Included as Part of this Thesis

First author Peer Reviewed Articles

Chapter 2 and Appendix B.1 – Stuart R. G. Buchan, Nathaniel D. Brough, Fergus W. Downey, Daniel C. Busan, Benjamin A. D. Hartig, Robert M. Howie, Phil A. Bland, Jonathan Paxman, and Martin Cupak, "Reliable Software Development for Small Satellite Missions: A Binar-1 Case Study," *Journal of Small Satellites*, under review.

Chapter 4 and Appendix B.2 – Stuart R. G. Buchan, Fergus W. Downey, Kyle McMullan, Benjamin A. D. Hartig, Eduardo Trifoni, Joice Mathew, Phil A. Bland, Jonathan Paxman, and Robert M. Howie, "Experimental Results and Modelling of the Binar-1 CubeSat On-Board Power Management in Thermal Vacuum Conditions," In preparation for journal submission.

First Author Conference Abstracts

Appendix C.1 – Stuart R. G. Buchan, Phil A. Bland, Jonathan Paxman, Benjamin A. D. Hartig, Nathaniel D. Brough, Fergus W. Downey, Daniel C. Busan, Martin Towner, Martin Cupak, Robert M. Howie, "The Binar CubeSat Program: Design and Development of a CubeSat Digital Twin," presented at the 19th Australian Space Research Conference (ASRC 2019) in Adelaide, Australia.

Appendix C.2 – Stuart R. G. Buchan, Fergus W. Downey, Benjamin A. D. Hartig, Daniel C. Busan, Jacob Cook, Chelse Tay, Kyle McMullan, Tristan Ward, Robert M. Howie, Phil A. Bland, Jonathan Paxman, "Binar CubeSat Program: Mission Two Payloads and Operation Plan," presented at the 36th Small Satellite Conference at the Utah State University in Logan, Utah, USA.

LIST OF COAUTHORED PUBLICATIONS

COAUTHORED CONFERENCE ABSTRACTS

Nathaniel D. Brough, **Stuart R. G. Buchan**, Fergus W. Downey, Benjamin A. D. Hartig, Martin Towner, Martin Cupak, Robert M. Howie, Jonathan Paxman, Phil A. Bland, ”The Binar CubeSat Program: Attitude Control for Small Satellites,” presented at the 19th Australian Space Research Conference (ASRC 2019) in Adelaide, Australia.

Benjamin A. D. Hartig, Phil A. Bland, Jonathan Paxman, Robert M. Howie, Martin Towner, Martin Cupak, Daniel C. Busan, Nathaniel D. Brough, Fergus W. Downey, **Stuart R. G. Buchan**, ”The Binar CubeSat Program: Past, Present and Beyond,” presented at the 19th Australian Space Research Conference (ASRC 2019) in Adelaide, Australia.

Fergus W. Downey, Benjamin A. D. Hartig, Nathaniel D. Brough, **Stuart R. G. Buchan**, Martin Cupak, Daniel C. Busan, Phil A. Bland, Jonathan Paxman, Robert M. Howie, “The Binar Space Program: Developing a reliable and efficient CubeSat Electronics Power System,” presented at the 19th Australian Space Research Conference (ASRC 2019) in Adelaide, Australia.

Appendix D – Fergus W. Downey, **Stuart R. G. Buchan**, Benjamin A. D. Hartig, Daniel C. Busan, Jacob Cook, Phil A. Bland, Jonathan Paxman, “Binar Space Program: Binar-1 Results and Lessons Learned,” presented at the 36th Small Satellite Conference at the Utah State University in Logan, Utah, USA.

Benjamin A. D. Hartig, **Stuart R. G. Buchan**, Fergus W. Downey, Phil A. Bland, Renae Sayers, Robert M. Howie, Rockwell McGellin, ”Binar Space Program: Launching CubeSats and Space Careers for Students of All Ages,” presented at the 36th Small Satellite Conference at the Utah State University in Logan, Utah, USA.

Fergus W. Downey, Benjamin A. D. Hartig, **Stuart R. G. Buchan**, Chelsea Tay, Jacob Cook, Daniel C. Busan, Phil A. Bland, Jonathan Paxman, Robert M. Howie, “Binar Space Program: Binar-1 Results and Lessons Learned,” presented at the 20th Australian Space Research Conference (ASRC 2022) in Sydney, Australia.

List of Abbreviations Used In This Thesis

ADC	**A**nalog to **D**igital **C**onverter
ADCS	**A**ttitude **D**etermination and **C**ontrol **S**ystem
AI	**A**rtificial **I**ntelligence
AMP	**A**synchronous **M**ulti**P**rocessing
ANU	**A**ustralian **N**ational **U**niversity
API	**A**pplication **P**rogramming **I**nterface
ASCII	**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
AWST	**A**ustralian **W**estern **S**tandard **T**ime
BCM	**B**inar **C**ubeSat **M**otherboard
BSF	**B**inar **S**oftware **F**ramework
BSP	**B**inar **S**pace **P**rogram
CI/CD	**C**ontinuous **I**ntegration/**C**ontinuous **D**eployment
COBS	**C**onsistent **O**verhead **B**yte **S**tuffing
COTS	**C**ommercial **O**ff **T**he **S**helf
CPU	**C**entral **P**rocessing **U**nit
CRUD	**C**reate **R**ead **U**pdate **D**elete
CSLI	**C**ube**S**at **L**aunch **I**nitiative
DBN	**D**ynamic **B**ayesian **N**etworks
DMA	**D**irect **M**emory **A**ccess
DRL	**D**eep **R**einforcement **L**earning
ECSS	**E**uropean **C**ommission for **S**pace **S**tandardisation
EDL	**E**ntry **D**escent and **L**anding
EKF	**E**xtended **K**alman **F**ilter
EPS	**E**lectrical **P**ower **S**ystem
FDIR	**F**ault **D**etection **I**solation and **R**ecovery
GNSS	**G**lobal **N**avigation **S**atellite **S**ystem

GPI	**G**eneral **P**urpose **I**nput
GPIO	**G**eneral **P**urpose **I**nput **O**utput
GPO	**G**eneral **P**urpose **O**utput
GPS	**G**lobal **P**ositioning **S**ystem
HAL	**H**ardware **A**bstraction **L**ayer
HITL	**H**ardware **I**n **T**he **L**oop
IC	**I**ntegrated **C**ircuit
IGRF	**I**nternational **G**eomagnetic **R**eference **F**ield
IMU	**I**nertial **M**easurement **U**nit
ISR	**I**nterrupt **S**ervice **R**outine
ISS	**I**nternational **S**pace **S**tation
IVT	**I**nterrupt **V**ector **T**able
JSSOD	**J**EM-**S**mall **S**atellite **O**rbital **D**eployer
JEM	**J**apanese **E**xperiment **M**odule
JPL	**J**et **P**ropulsion **L**aboratory
LEO	**L**ow **E**arth **O**rbit
LFI	**L**earning **F**rom **I**ncidents
LOC	**L**evels **O**f **C**ompliance
MIB	**M**ishap **I**nvestigation **B**oard
MISRA	**M**otor **I**ndustry **S**oftware **R**eliability **A**ssociation
ML	**M**achine **L**earning
MPC	**M**odel **P**redictive **C**ontrol
NASA	**N**ational **A**eronautics and **S**pace **A**dministration
NMEA	**N**ational **M**arine **E**lectronics **A**ssociation
NN	**N**eural **N**etwork
NSTF	**N**ational **S**pace **T**est **F**acility
OBC	**O**n **B**oard **C**omputer
OBS	**O**n **B**oard **S**torage
OS	**O**perating **S**ystem

PPOD	**P**oly-**P**icosat **O**rbital **D**eployer
PCB	**P**rinted **C**ircuit **B**oard
PPS	**P**ulse **P**er **S**econd
PV	**P**hoto**V**oltaic
PWM	**P**ulse **W**idth **M**odulation
RAM	**R**andom **A**ccess **M**emory
ReLU	**Re**ctified **L**inear **U**nit
RMS	**R**oot **M**ean **S**quare
RPC	**R**emote **P**rocedure **C**all
RTC	**R**eal-**T**ime **C**lock
RTOS	**R**eal-**T**ime **O**perating **S**ystem
SDK	**S**oftware **D**evelopment **K**it
SEE	**S**ingle **E**vent **E**ffect
SGP	**S**implified **G**eneral **P**erturbation
SPI	**S**erial **P**eripheral **I**nterface
SSDL	**S**pace **S**ystems **D**evelopment **L**aboratory
SatNOGS	**Sat**ellite **N**etwork **o**f **G**round **S**tations
TID	**T**otal **I**onising **D**ose
TCP	**T**ransmission **C**ontrol **P**rotocol
TDD	**T**est **D**riven **D**evelopment
TFLM	**T**ensor**F**low **L**ite for **M**icrocontrollers
UART	**U**niversal **A**synchronous **R**eceive **T**ransmit
UHF	**U**ltra **H**igh **F**requency
UML	**U**niversal **M**odelling **L**anguage
USART	**U**niversal **S**ynchronous/**A**synchronous **R**eceive **T**ransmit
USB	**U**niversal **S**erial **B**us
UTC	**C**oordinated **U**niversal **T**ime

CHAPTER 1

INTRODUCTION

1.1. THE CUBESAT

1.1.1. HISTORY OF THE PLATFORM

Since the launch of the first satellite by the Soviet Union in the late 1950s, the high cost associated with space research has made it a challenging endeavour for small organisations. The Cube Satellite, or CubeSat as it is more commonly known, provides a solution to this by drastically lowering the cost required for a space mission through miniaturisation and standardisation. The original purpose of the platform was to open up the space sector to education and research, and it has been achieving this goal for over twenty years. According to the global database for nano-satellites, as of January 2023, nearly two-thousand CubeSats have been launched into Earth orbit, with a further two-thousand expected to be launched in the next six years [1].

The modern-day CubeSat stems from a heritage of spacecraft design at Stanford University. Launched in 1994, the Space Systems Development Laboratory (SSDL) at Stanford University intended to provide project-based learning experiences in space systems engineering for engineering students [2]. The first satellite out of this program was named SAPPHIRE, the Stanford Audio Phonic Photographic Infrared Experiment. With a total budget of $50,000 USD and a resource pool of commercial off the shelf

(COTS) components, SAPPHIRE was a student-run project demonstrating that it was possible to conduct inexpensive space research in a small form factor [3]. As a successor to SAPPHIRE, SSDL began work on their second satellite, OPAL [2]. The Orbiting Automated Picosat Launcher (OPAL) was designed as a proof of concept mission to demonstrate remote deployment of many satellites from a mothership [2]. The payload satellites, dubbed 'picosatellites', were proposed to be roughly 80mm x 80mm x 50mm in dimension, possess independent onboard power and a means to collect and transmit data [4]. These picosatellites can be viewed as the original CubeSats.

Following the success of SAPPHIRE and OPAL, SSDL determined that picosatellites could aid in achieving the program's initial goal of opening up the space sector to education and research. An assessment of the picosatellite power requirements determined that a cube shape would be optimal for including body-mounted solar panels. Furthermore, from an analysis of the OPAL launch system, to provide room for rails and solar panels, it was decided that the optimal satellite width was to be 100 mm to store three satellites [2]. With these preliminary design requirements set, a collaboration between SSDL and California Polytechnic State University was constructed in 1999, which remains the foundation for every CubeSat launched today [5].

1.1.2. The CubeSat Design Specification

The CubeSat Design Specification is a document created by the California Polytechnic State University (Cal Poly) which dictates the constraints placed on satellites that intend to use the standardised launch system [5]. The document also stands to formalise the optimal design decisions mentioned in Section 1.1.1 to provide the standardised CubeSat measurement system of the Unit. For the remainder of this thesis, CubeSats will be referred to in multiples of 'U', which are cubes with side lengths of 100 mm.

P-POD

The Poly Picosatellite Orbital Deployer is the standardised successor to OPAL for CubeSat deployment [5]. One of the primary motivations for its development was to

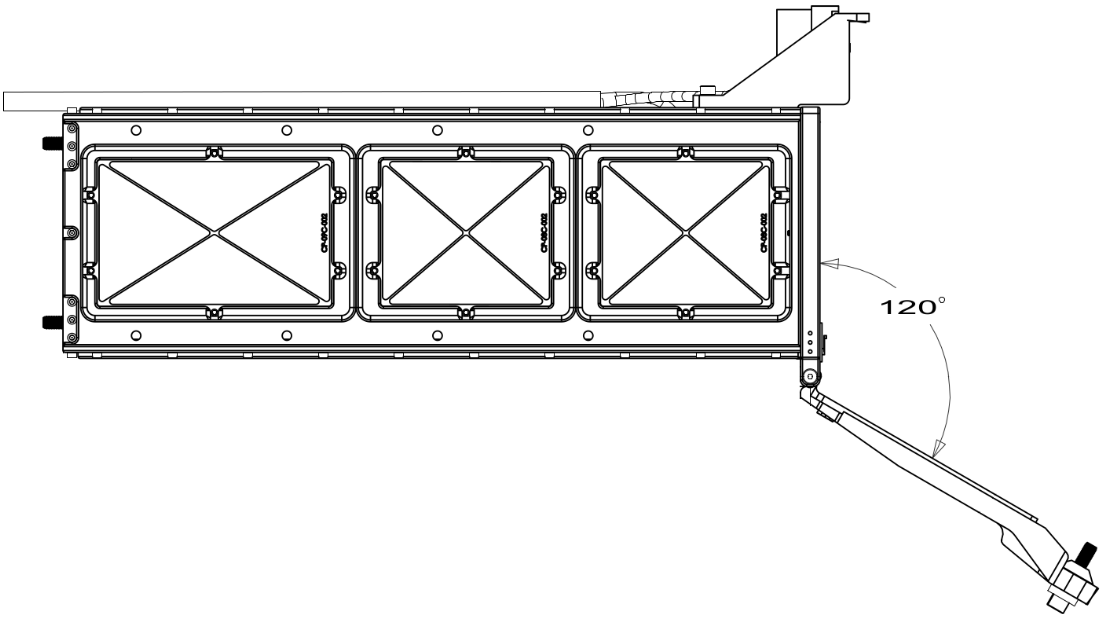


FIGURE 1.1: Side view of a P-POD system with an open exit hatch. Adapted from [7].

increase access to space for small satellites [6]. Unlike OPAL, the P-POD is designed to interface three 1U CubeSats with a launch vehicle rather than being a standalone orbital deployment system [7]. Consisting of an aluminium box with a spring for ejection, the P-POD allows for easy, safe deployment of satellites with the added benefit of reducing launch costs by sharing payload space [6].

Figure 1.1 depicts the current P-POD system used for CubeSat deployment. Before launch, an engineer will arm the satellites inside the container via the three access panels on the front-facing side of Figure 1.1. When ready for deployment, the front access panel will open, and the loaded spring will eject the CubeSat payloads. Integrating P-PODs into many of NASA's launch vehicles has directly contributed to CubeSat build teams' easy access to ridesharing capabilities, reaching space at a much lower cost [8].

The success of the P-POD has led to the development of the JEM Small Satellite Orbital Deployer (J-SSOD) as part of the commercialisation of the Japanese section of the International Space Station (ISS). The specifications for the launcher design are similar to the P-POD, with the added change of extending the container size to support up to 6U launches [9].

SATELLITE CONSTRAINTS

The CubeSat Design Specification specifies four main categories of requirements for safe integration with a P-POD: General, mechanical, electric, and operational requirements. Each will be addressed in this section, highlighting the requirements necessary for the reader to understand the CubeSat design process.

General requirements govern aspects of the satellite interacting with the environment, causing danger to other spacecraft. As space debris is becoming a serious issue with operation in the space sector [10], it is specified that all parts shall remain attached to the CubeSat during its lifecycle.

Mechanical requirements govern the size and mass of the spacecraft and the hardware it uses. All sizes of CubeSat shall be 100 mm in the X plane to fit in the launcher. For the P-POD, all sizes of CubeSat shall also be 100 mm in the Y plane. This restriction is reflected in the 1U, 2U and 3U spacecraft launched from the J-SSOD. However, the permitted size is extended to 226.3 mm for the 6U spacecraft that this launcher also accommodates. The Z plane for a 1U CubeSat shall be 113.5 mm long, and the satellite's total mass can be at most 1.33 kg. CubeSat mass limits are dictated for sizes larger than a 1U; however, as this thesis is concerned with Binar-1, a 1U CubeSat, they will not be included here. The centre of gravity for a 1U spacecraft is to lie within 20 mm of its geometric centre in both the XY and Z directions.

Electrical requirements dictate when power can be used on board the spacecraft. The power system must be deactivated at launch and only enabled once ejected from the launcher. Operational requirements describe how a CubeSat is to operate safely during its lifetime. Once ejected from the P-POD, the design specification states that the CubeSat is prohibited from actuating deployables and radio communications for up to 45 minutes post-ejection.

1.1.3. POPULARITY

A major factor that has led to recent considerable growth in small satellite missions is the increasing capability of electronics coupled with their size reduction. This combination allows payloads to be developed that fit within the CubeSat form factor and achieve the same calibre of research as traditional large-scale spacecraft. Alongside this, the advent of rideshare companies changing the paradigm of reaching orbit has made accessing space progressively cheaper. According to [11], at the time of publication in 2015, the only rideshare provider that publicly released launch prices was SpaceFlight Services, who quoted a 3U LEO launch cost of $295,000 USD. The same provider also quoted a 150kg LEO launch at $4.95 million USD. According to the CubeSat Standard, a 3U satellite has a maximum mass of 4kg [5]. This mass places the cost of a CubeSat rideshare launch during 2015 at roughly $74,000 USD per kilogram, whereas the small spacecraft launch would cost $33,000 USD per kilogram. Whilst more expensive per kilogram, the overall mission cost is significantly less than that of a full-scale mission and hence tended to be a platform mostly explored by universities. The high adoption by university groups aided in the perception that CubeSats were originally just a 'toy' platform in which students could learn about space systems engineering. However, with recent launch costs lowering substantially with rideshare options growing, coupled with the previously mentioned increasing capability of electronics, the CubeSat platform is being explored by non-university bodies at an increasing rate.

1.1.4. BEYOND LEO: THE NEXT STEP

The first CubeSats were launched in 2003, three years after the concept was created [12]. With now more than 1900 CubeSats launched, the platform has thoroughly demonstrated its applicability to research in a near-Earth environment and, as such, has recently been adapted to deep space missions. As a CubeSat technology demonstration, in 2018, NASA launched two CubeSats as part of the InSight Mars mission. These two 6U CubeSats, named MarCO-A and MarCO-B, were designed to relay data from the main lander as it descended to the Martian surface. The mission was successful, making the CubeSats the first of their satellite class to complete an interplanetary mission [13]. Following completing their primary objective, the CubeSat pair lost contact with Earth as their trajectory strayed further from the Sun. Failing to regain communication after their orbit reestablished line-of-sight, the mission was concluded [14].

The maiden launch of NASAs Artemis-1 in 2022 carried ten CubeSat secondary payloads beyond LEO. Of the ten launched, only four successfully established communications with Earth. The remaining six CubeSats failing to establish contact include the Near-Earth Asteroid Scout (NEA Scout) [15] and the CubeSat for Solar Particles (CuSP) [16] missions. The operational state of these CubeSats is still being determined. With the assumption that the spacecraft is still functional without communications, the NEA Scout development team have sent emergency communication requests to the spacecraft to prematurely deploy its solar sail in the hopes of optically sighting the satellite from Earth [17]. Similarly, the CuSP team is actively working to blindly regain contact with the spacecraft, assuming it is still functioning following a battery anomaly [18].

Whilst interplanetary CubeSats show promise in revolutionising space exploration, the commonality of mission termination and failure due to loss of communications is a major barrier to their ongoing success. The CubeSat platform relies heavily on communications with ground operators, an acceptable drawback for satellites operating in LEO. As the distance between the CubeSat and Earth increases, however, factors such as weak signals, communication delays, and ground control outages begin to have more of an impact on the mission. To address these challenges, the next boundary of CubeSat development is to incorporate onboard autonomy that can reduce the frequency of communications required with the ground. Heightened autonomy on CubeSats could enable the mission to continue to achieve objectives in a communications-denied environment, extending the mission's lifetime. By focusing on onboard autonomy, CubeSats will continue to push the boundaries of space exploration and pave the way for new scientific discoveries.

1.2. BINAR-1

The Binar Space Program began in 2018 as an undergraduate engineering honours project. In a small team of students, the project's initial scope was to purchase COTS systems from small satellite manufacturers and use them to assemble the 'Curtin University CubeSat', a common approach followed in the space sector. However, it

TABLE 1.1: Price in USD for the minimum required components for a functional satellite.

Part	Pumpkin	Endurosat	Tensor Tech
Flight Computer [19]	$1,575	-	-
Electrical Power System [20]	$10,500	-	-
Solar Panels [21]	-	$10,000	-
Location System [22]	$10,485	-	-
Communications System [23] [24]	-	$10,100	-
Structure [25]	-	$1,700	-
Attitude Control System (magnetorquers) [26]	-	-	$20.000
Total	**$22,560**	**$21,800**	**$20,000**

TABLE 1.2: Height and mass of the minimum internal components seen in Table 1.1.

Part	Height [mm]	Mass [g]
Flight Computer [27]	17.6	88
Electrical Power System [28]	48.4	710
Location System [29]	18.1	109
Communications System [30]	17.6	94
Attitude Control System (magnetorquers) [31]	13.5	140
Total	**115.2**	**1,141**

was evident from an early stage that this would result in three main problems: cost, size, and incompatibility.

The reasons for these issues can be seen when the spacecraft is broken down into its main components. For a spacecraft to operate at a minimum, it will require a flight computer, an electrical power system, a location system, a telecommunications system and a structure. Most small satellite component manufacturers operate on a quote-only basis, so getting a price estimate can be difficult. Some well-established small satellite component manufacturers, Pumpkin, Endurosat, and Tensor Tech, offer online pricing for some components that can be viewed as an average for the small satellite market. The cost of these components can be seen in Table 1.1.

Summing the total cost of the components seen in Table 1.1 gives a price of $64,360 USD, or approximately $100,000 AUD, to assemble the bare minimum for a spacecraft. This cost does not include a payload, meaning the spacecraft will have no scientific usefulness. This price does not include launch costs dependent on spacecraft dimensions. The size and mass of the selected internal components can be seen in Table 1.2.

As discussed in Section 1.1.2, the maximum height for a 1U CubeSat is 113.5 mm, with the maximum mass being 1.33 kg. It is evident from the summation of the size and mass of the bare minimum components in Table 1.2 that both of these thresholds have been exceeded even before the inclusion of a payload. This would require the satellite container to need to be increased to a 2U to handle the overflow. If scientific usefulness is desired, adding a payload could bring the size and mass of the satellite beyond the 2U limit, meaning that a further extension to a 3U satellite would be required to house a payload. While SpaceX advertises a relatively affordable rate of USD 280,000 for a 50kg spacecraft to reach LEO [32], it is important to acknowledge that this pricing model is not directly translatable to an extrapolation of USD 5,600 per kilogram. A 50kg payload represents the minimum mass a customer can send into orbit with SpaceX. Moreover, the advertised cost does not encompass the expenses associated with integration and testing, which can constitute a substantial portion of the overall expense. Drawing upon the Binar Space Program's experience launching Binar-1, the true cost of launching a 1U to 3U CubeSat into LEO is often within the range of USD 50,000 to USD 100,000 per kilogram.

Whilst the CubeSat standard does not impose restrictions on subsystem integration, commercial manufacturers tend to use the PC/104 standard connector, along with an inter-integrated circuit (I^2C) data bus. Conformity to this unofficial standard allows subsystems from different suppliers to be integrated [33]. It is evident, however, that different suppliers may implement the PC/104 connector and data transmission bus inconsistently, which can lead to potential failures [34]. Due to the elusive and somewhat unobtainable goal of building small, low-cost spacecraft using COTS components and the possibility of incompatibility of doing so from manufacturers, the scope of the Binar Space Program was shifted from an early point to focus on in-house development. Commencing February 2019, the Binar Space Program began development on Binar-1: the maiden mission of the Space Science and Technology Centre at Curtin University.

1.2.1. MISSION DESCRIPTION

The scope shift towards creating custom satellite circuitry meant that the risk involved with including an expensive payload on the first spacecraft was deemed too high. Therefore, from an early point in the program, the objective for Binar-1 was to be a product demonstrator for custom-developed spacecraft technology. Second to this, two inexpensive camera payloads were flown. The proof of concept hardware produced from the undergraduate engineering honours project mentioned in Section 1.2 was assembled into the first Binar CubeSat Motherboard (BCM) at the commencement of the Binar Space Program in 2019. This hardware included power, attitude determination and control, communication and flight computer systems. The BCM, along with all the other components required for the satellite, can be seen in the exploded view of Binar-1 in Figure 1.2.

Due to the highly integrated size, two BCMs were installed in Binar-1 to effectively collect the data of two missions without the extra launch costs. The primary BCM can be seen highlighted in teal in Figure 1.2, and the secondary in orange. Figure 1.3 shows a higher detail render of the BCM.

1.2.2. HARDWARE

This section will provide an overview of the key subsystems of Binar-1. While the information discussed in this section is not a contribution to this PhD thesis, it is included to provide the reader with a comprehensive understanding of the developed CubeSat platform. The developed platform serves as a foundation on which the contributions this thesis aims to make to the field of small satellites can be explored.

ELECTRICAL POWER SYSTEM

The spacecraft's electrical power system generates, stores, and distributes power. It features redundant battery pairs, as illustrated in Figure 1.4. Each redundant battery system is equipped with independent maximum peak power tracking circuitry, which

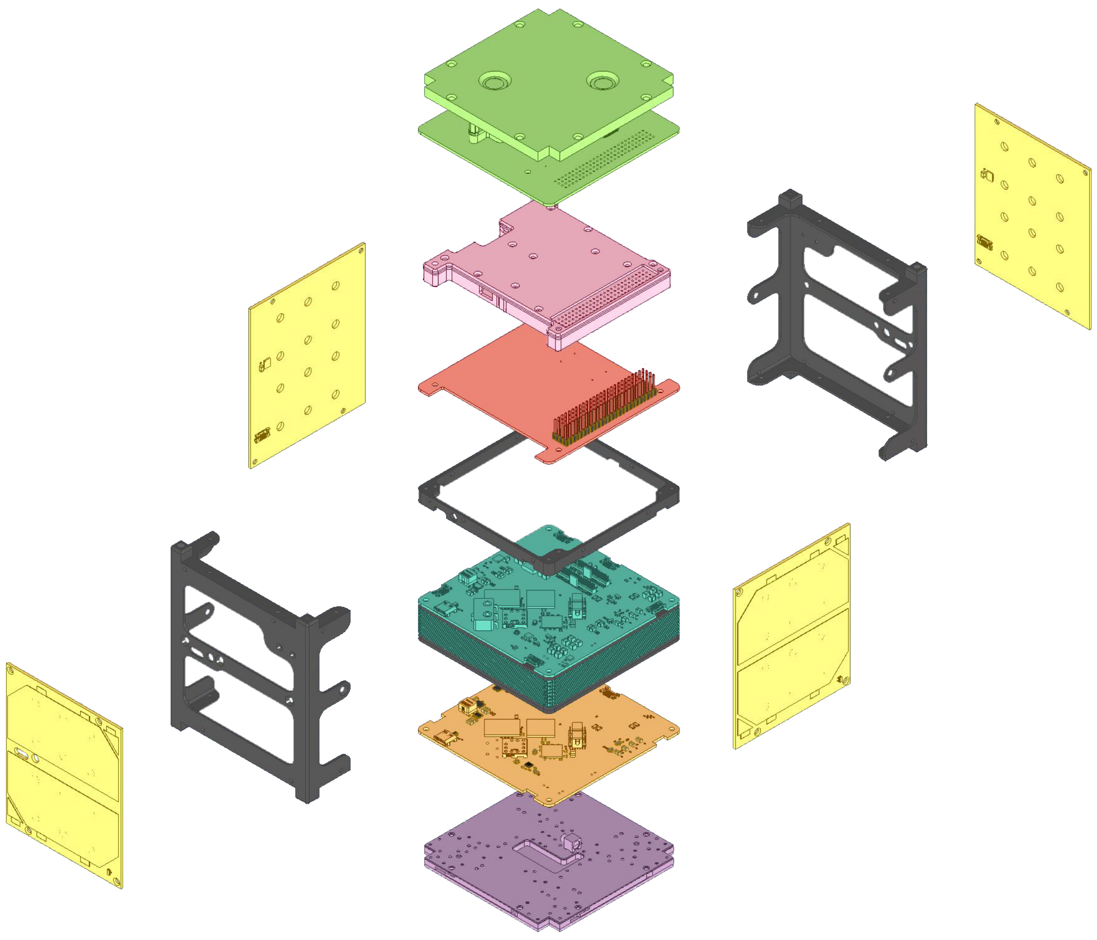


FIGURE 1.2: A colour coded exploded view of Binar-1. The base of the satellite with portholes for the payload cameras and the payload carrier board are shown in green. Pink identifies the COTS transceiver, connected to the primary BCM (Teal) through the red adaptor board. Orange shows the secondary BCM. Communications are facilitated through the purple antenna. The solar panels (yellow) are mounted on the outside of the structural frame (Black).

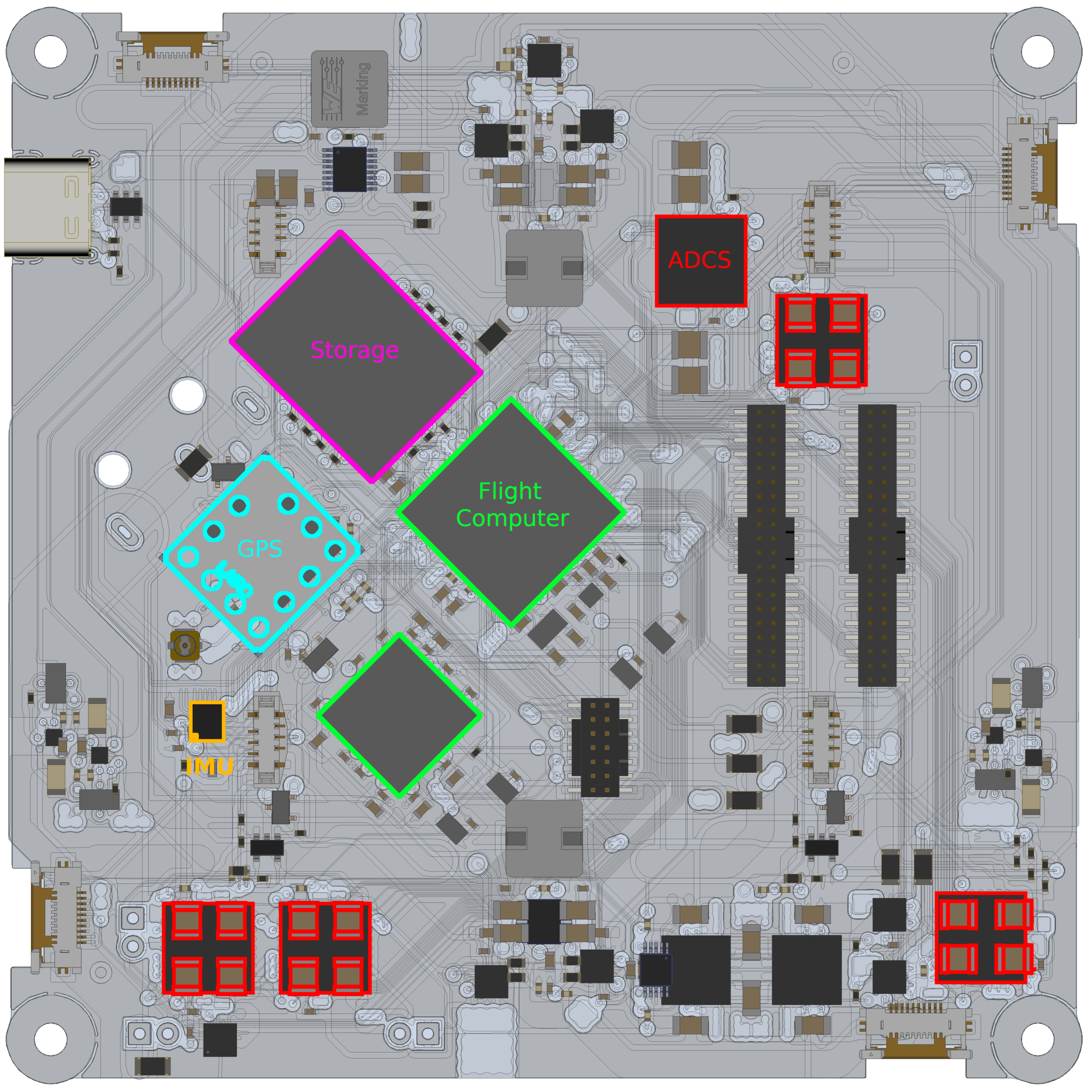


FIGURE 1.3: Render of the BCM that flew on Binar-1. Key sections have been highlighted; pink for storage, green for flight computers, orange for IMU, blue for GPS and red for the ADCS.

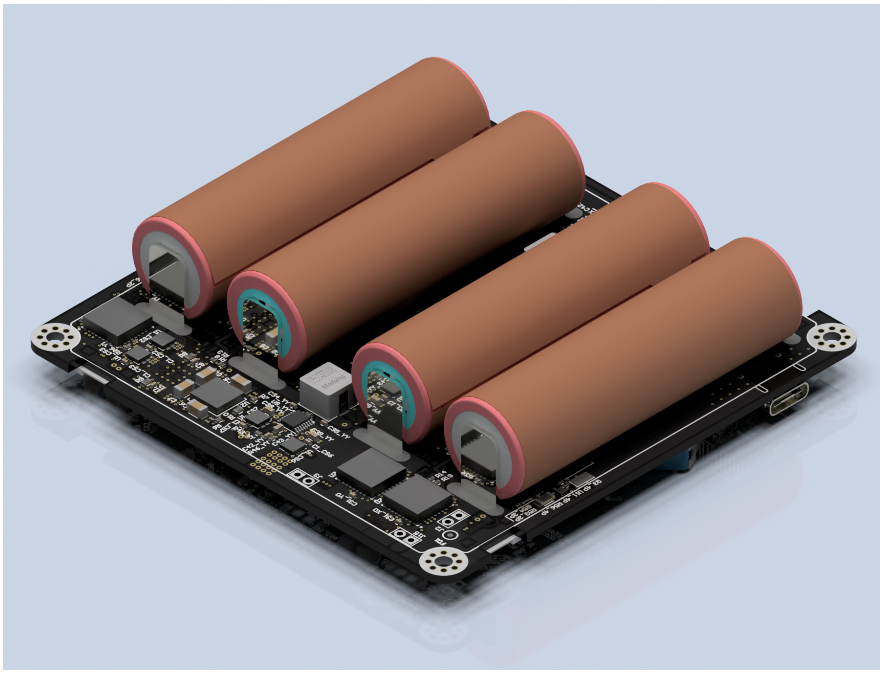


FIGURE 1.4: Render of the EPS on Binar-1. The four Lithium-Ion batteries are connected in a two-series, two-parallel configuration. They are charged through the solar panels using maximum peak power tracking circuitry.

can be seen adjacent to the batteries in Figure 1.4. This circuitry utilises four solar panels mounted outside the spacecraft, with an ideal maximum charging voltage of 5 V, to optimise power generation and storage in the batteries. The four Lithium-Ion batteries are connected in a two-series, two-parallel configuration. When fully charged, these batteries provide 8.3 V to the motherboard. The distribution network handles the necessary power adjustments for the onboard subsystems, regulating the battery voltage to 3.3 V and 5 V rails.

A battery management system is in place to protect both the subsystems and the batteries. This system offers protection against overvoltage, undervoltage, overcurrent, and short-circuit current. In parallel, the complete battery pack can store 25 Wh of energy, or 50 Wh when combined.

GPS

The GPS system, denoted in blue in Figure 1.3, is a high-performance, high-sensitivity, low-power receiver with an external antenna. It is used to determine the spacecraft's location over the Earth. The GPS communicates with the spacecraft flight computer using National Marine Electronics Association (NMEA) strings over universal asynchronous receiver/transmitter (UART) serial, which are decoded and parsed to receive the information.

STORAGE

The onboard storage, highlighted in pink in Figure 1.3, provides 32 gigabytes of solid-state memory for saving system logs and payload data. It can also be used to store backup copies of the spacecraft's flight software for restoration of the program memory in the event of a fault.

ATTITUDE DETERMINATION CONTROL SYSTEM

The ADCS is split into two core subsystems; Determination and control. The IMU highlighted in orange in Figure 1.3 encompasses the determination subsystem. With high accuracy and refresh rate, it measures the angular velocity of the spacecraft and the intensity of the magnetic field surrounding it. As seen from Figure 1.5, a metal frame is fitted around the batteries, housing three wire coils. These coils, known as magnetorquers, rely on electromagnetism to form the control subsystem.

To perform an attitude manoeuvre, the intensity of the Earth's magnetic field at a point in orbit is measured using the IMU. The magnetorquer drivers (highlighted red in Figure 1.3) control the current in the wire coils around the circuit board and in the two internal perpendicular rods (seen in Figure 1.5) to create independent magnetic dipoles. These dipoles move the satellite by pushing against the Earth's magnetic field with a configurable amount of counter-force. If the X or Y-axis magnetorquer fails, two redundant magnetorquers embedded into the solar panel PCBs can be enabled. Monitoring the change in angular velocity with the IMU completes a feedback loop and aids in tuning the current required through the magnetorquers to reach the orientation goal.

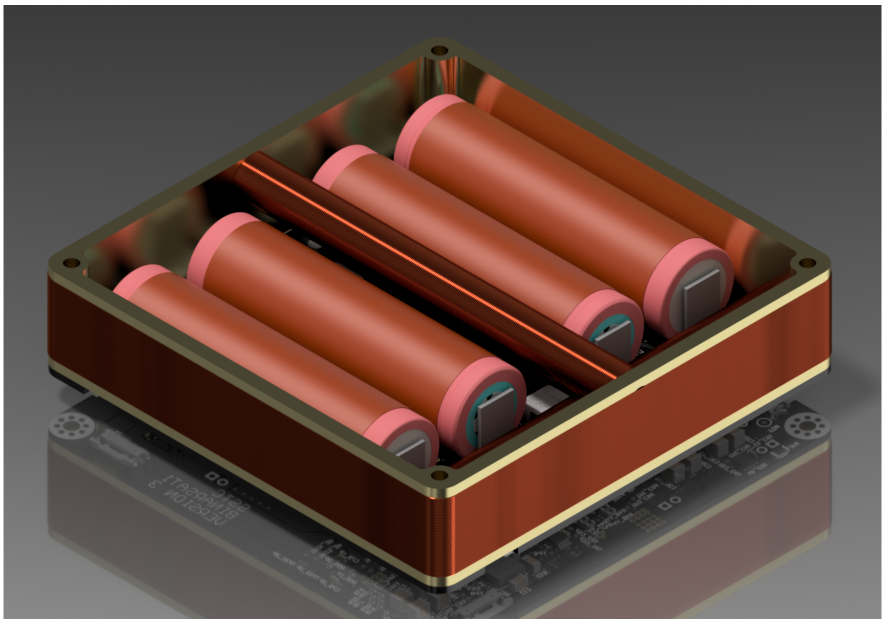


FIGURE 1.5: Render of the magnetorquers on Binar-1. The Z-axis coil surrounds the BCM. The X and Y-axis coils slot between and in front of the batteries.

COMMUNICATIONS SYSTEM

The communications system used on Binar-1 was a COTS solution developed by Endurosat. It is a UHF system comprised of three parts, shown in Figure 1.2. The communications adaptor board connecting the BCM and the transceiver is highlighted in red. The transceiver, highlighted in pink, processes the radio signal received from the antenna and generates signals to be sent from the antenna. The antenna, highlighted in purple, facilitates communications with the ground station. In conformance with the standardised launcher requirements, the antenna default state is stowed, meaning the elements are coiled and stored inside the antenna package. After deployment from the launcher, the module uses burn resistors to heat up and melt the nylon wire that holds the antenna doors closed, allowing the coiled elements to unravel. These burn wires can be seen highlighted in red in Figure 1.6.

FLIGHT COMPUTERS

As can be seen from the green highlighted sections of Figure 1.3, the BCM used on Binar-1 has two flight computers. The main flight computer, the larger of the two, is a

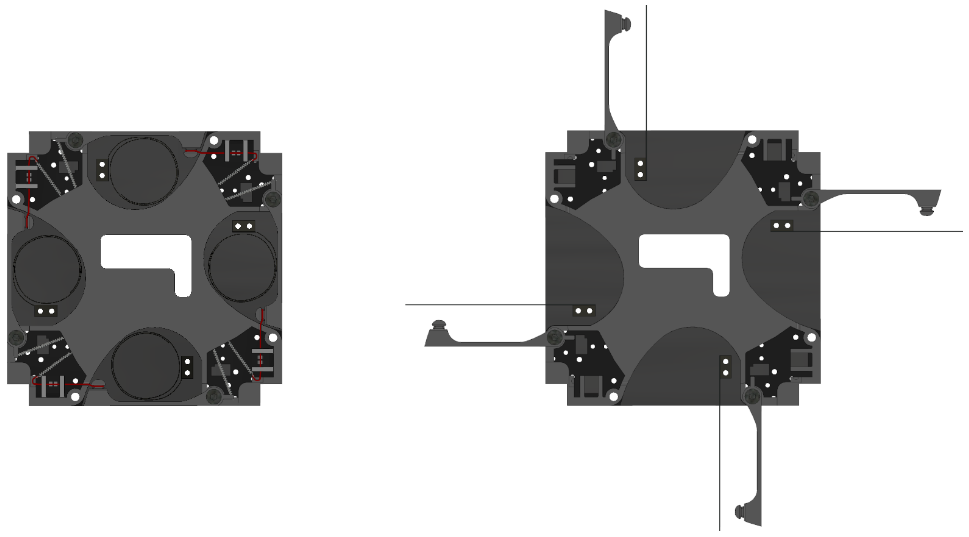


FIGURE 1.6: Stowed antenna configuration (left) and deployed antenna configuration (right).

dual-core microcontroller comprising a Cortex-M7 and Cortex-M4 core. The secondary flight computer, the smaller of the two, is a single-core Cortex-M4 microcontroller. Figure 1.7 provides an overview of the flight computers system on the BCM used on Binar-1.

The primary flight computer is purposed with controlling the spacecraft operations. With general purpose input output (GPIO), analog to digital converter (ADC), universal synchronous/asynchronous receiver/transmitter (USART), serial peripheral interface (SPI) and I^2C support, it can seamlessly integrate with the communications, attitude control, electrical power, onboard memory and payload subsystems to achieve mission goals. I^2C is used to interface with the communications system to action telecommands from the ground. The IMU and magnetorquer drivers are interfaced with the main flight computer over SPI, and the GPS is connected over UART. The primary flight computer integrates the data received from these components with the operational software for attitude control. GPIO and ADC inputs are used extensively with the electrical power system to monitor system health continuously. GPIO lines are implemented to turn subsystems on and off and to watch for system failures through interrupts asynchronously. Voltages, currents and temperatures are read using the ADC. SPI is used to interface with a payload and to store data in the onboard memory for downlink where possible.

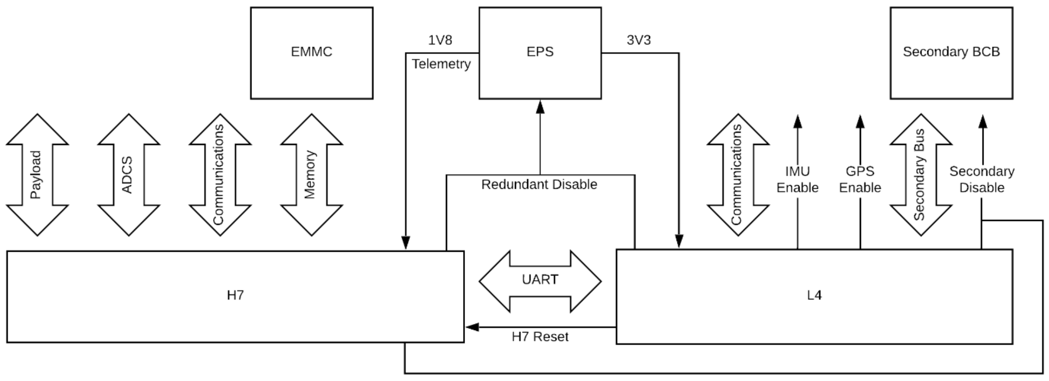


FIGURE 1.7: A system block diagram of the flight computers on the BCM used on Binar-1. The primary flight computer (H7) has access to the main system peripherals: Payload, ADCS, communications, and onboard storage. The secondary flight computer (L4) has access to the communications peripheral and the secondary BCM. It also has access to the enable line of the IMU and GPS devices to disable them if they prevent the primary flight computer from operating correctly. The H7 and L4 can communicate over a UART connection. The L4 has the added ability to reset the H7 in an emergency. Both computers can configure the EPS supplying 3.3 V to select between the primary and redundant converter.

The secondary flight computer takes up little room on the BCM yet is powerful enough to aid in regaining control of the spacecraft if an error occurs with the main flight computer. Communication between the two flight computers is continuously implemented by UART, in which the state of the primary flight computer is monitored by the secondary. A GPIO line is connected from the secondary to the primary flight computer, allowing the secondary computer to trigger a reboot in the event of a recoverable failure, in which case the primary flight computer will resume control. In the event of an unrecoverable primary computer failure, however, the secondary computer can interface with the communications module over I^2C to remain in contact with the ground and commence survival operations. As the computer's main goal is loss prevention, it has the authority to disable the payload using its GPIO connection. Moreover, similar to the primary flight computer, it can disable the redundant power system with the added capacity to also power down the IMU and GPS subsystems if an error with one of these systems causes the primary flight computer to fail. The secondary flight computer is capable of both programming the primary flight computer and being programmed by it in turn.

Both flight computers onboard the BCM used on Binar-1 were programmed using the Binar Software Framework (BSF), a custom in-house software framework developed by the candidate to maximise system reliability.

1.3. THE NEED FOR RELIABLE SOFTWARE IN SPACE MISSIONS

The success of space missions, regardless of their scale and complexity, hinges significantly on the reliability of their software systems. However, determining the exact impact of software on mission failures remains challenging due to the scarcity of publicly available data. Fortunately, due to their presence in the civil sector, recent data on CubeSat failure has shed some light on the impact of software flaws on their mission success rates. In this section, the pivotal role that reliable software systems play in the success of space missions will be discussed, with a particular emphasis on CubeSat missions, drawing insights from the impact of software on such missions.

As the prevalence of CubeSats in the space industry rockets higher, it is becoming increasingly apparent that a high rate of failure plagues these small satellite systems. This highlights the need to address the reliability issues that CubeSats face in the pursuit of developing more robust satellite systems that can operate safely in the harsh conditions of space.

Figure 1.8 demonstrates that educational university-built CubeSats have a failure rate of nearly 51%, while industry-built CubeSats have a failure rate of roughly 32%. This statistic has been substantiated through an independent analysis conducted by Dubos et al., revealing that approximately 50% of CubeSats launched were unable to fulfil their intended mission [35]. Furthermore, the National Academies of Sciences, Engineering, and Medicine have reported a failure rate of 55% for education-only CubeSats [36]. It is worth noting that NASA-funded CubeSats exhibit a considerably lower failure rate of 17%; however, they do not constitute the majority of CubeSats launched. This high failure rate among CubeSats is a cause for concern, as it not only results in a significant loss of time, money, and resources but also undermines the credibility of CubeSats as a viable, cost-effective solution for performing space research.

A comparison can be drawn between the high failure rates observed within CubeSat

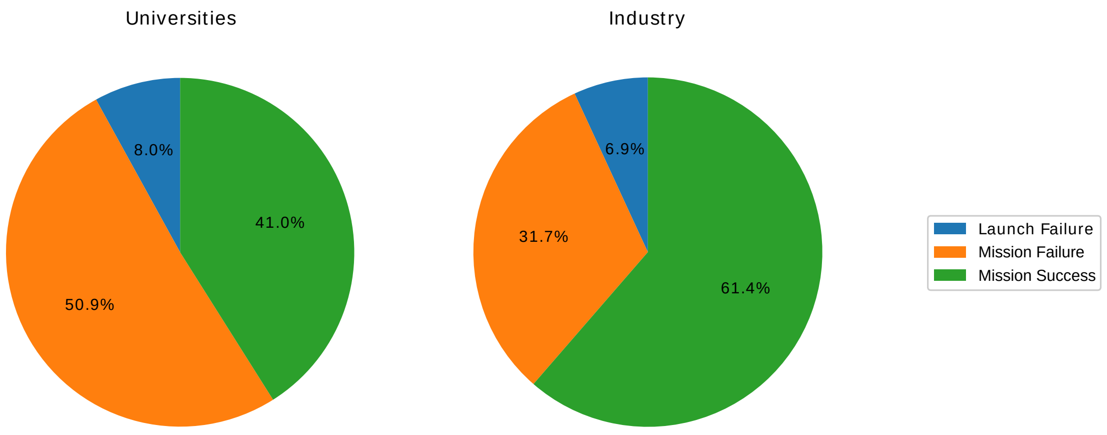


FIGURE 1.8: A comparison of mission success rates for first-time CubeSat launches between universities and industry. Educational CubeSats developed from university groups are likelier to fail than their industry counterparts. Adapted from [37].

missions and the success rates of NASA-built missions, considering NASA's reputation as a reputable organisation with extensive flight heritage. Internally to NASA, mission development closely references mission assurance guidelines. These guidelines categorise missions into four distinct risk classes: A, B, C, and D, each tailored to fulfil programmatic requirements [38]. Class A missions are highly critical, costly, complex, and long-lasting with strict launch constraints. They tend to be the most expensive missions developed by NASA. They require strict mission assurance processes to ensure success over a long lifespan, typically over ten years. Class B missions are critical with moderate risk tolerance, allowing minor compromises in risk in exchange for cost savings. They have high to medium complexity, long mission duration, and moderate launch constraints. Class C missions have a moderate risk acceptance. These missions are lower in cost and complexity and have a shorter lifespan than Class A and B. Class D missions are of low importance and have a high tolerance for risk. They are low-cost, simple, short-lived, and flexible. Failures in Classes C and D impact NASA's research objectives less than those in Classes A and B. Hence, unsurprisingly, these categories have the highest mission failure rate of roughly 20% [39]. When comparing the failure rate of the most risk-averse NASA missions to the previously discussed high mission failure rate of CubeSats, it becomes evident that CubeSat missions, despite their typically lower complexity and less critical objectives compared to NASA's more significant endeavours, still struggle to meet even the lower reliability expectations for Class C/D missions. This underscores the inherent challenges and difficulties associated with space projects.

Whilst the rate of failure remains high, the importance of the CubeSat form factor as an entry into the space industry needs to be stressed. The cost-effective access to space enabled by the CubeSat platform allows university groups to conduct research that would otherwise be unfeasible. Building and flying CubeSats provides students with hands-on experience in all aspects of space technology, from design to operations. This experience is valuable for students who aspire to work in the space industry and can help to bridge the gap between academia and industry. Therefore, the high failure rate must be addressed to ensure the success and longevity of the missions produced by university groups.

Although the high failure rate can be attributed to many complex systems across the platform, the scope of this thesis will explicitly address software faults, one of the major challenges faced in space missions. Spacecraft flight software is considered high risk as it directly controls all onboard systems with some level of automation [40]. These faults can originate both during software development and operations, often leading to mission failures. In a 2016 survey asking the small satellite community why CubeSats fail, a subset of respondents who identify as experts in the field identified software design error as the leading cause of failure within the first six months of operation [41]. The results from this subset are summarised in Figure 1.9.

The limited resources and harsh operating environments of space missions pose significant challenges for software development. Spacecraft often have limited computational power, storage, and communication capabilities and are subject to extreme temperatures, radiation, and vacuum conditions. These factors make it difficult to ensure the reliability and fault-tolerance of software systems in space missions. A single software fault in a mission can directly contribute to failure. For example, a software fault in the power management system can cause a power outage, rendering the entire system inoperable. A software fault in the communication system can prevent a spacecraft from sending or receiving data, compromising the mission's objectives. In addition to the potential for mission failure, software faults can lead to unexpected behaviour and data corruption, compromising the integrity and accuracy of collected data. This can significantly affect research throughput when using the platform. There-

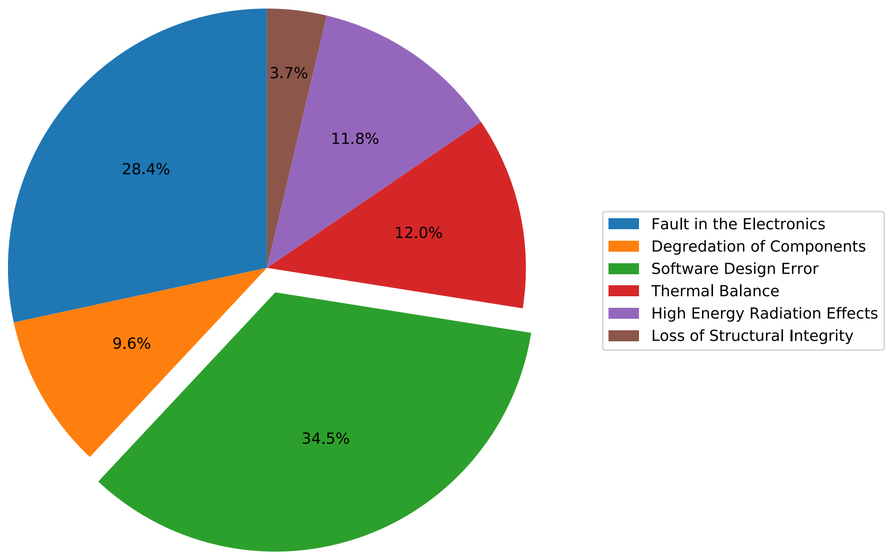


FIGURE 1.9: The result of a survey asking experts in the small satellite industry their suspected reasons for CubeSat failure [42]. The largest expected reason for failure is due to software design errors, highlighting the importance of robust software design to the success of a mission.

fore, it is essential to establish the reliability and fault-tolerance of software systems in space missions to ensure mission success. Pursuing this requires a comprehensive approach to software development, testing, and validation that considers the unique challenges and constraints of space mission systems.

Whilst the CubeSat form factor offers a platform to reduce the hardware complexity of a conventional large-scale space mission, the same reduction is not reflected in the complexity of the flight software. This complexity can create a barrier to entry for new entrants to the space field [43]. Hence, these entrants may seek literature to aid in the development process. Whilst software remains a significant factor in the success of a mission, published literature overlooks this importance. The NASA CubeSat Launch Initiative (CSLI) provides a document to first-time CubeSat Developers identifying key aspects of the importance of the mission. The document seldom mentions flight software design, lacking guidance for new entrants on how to approach this challenge [44]. Grubb proposes that CubeSat developers using this document may discover that flight software is a considerably more difficult component of a CubeSat than alluded to in the document, contributing to the high mission failure rate [45]. External to this document, the lack of specialised literature for flight software design further increases the magnitude of this challenge [46]. Moreover, specific CubeSat mission failure reasons are seldom published, perhaps due to a lack of telemetry preventing analysis from being conducted [47]. This scarcity of publications removes the possibility for flight software developers to learn from the mistakes of previous missions. While not directly replacing this lack of literature, coding standards can offer assistance in addressing this knowledge gap.

Software coding standards provide a time-tested approach to software development and serve as an invaluable reference that encapsulates the collective knowledge of experienced engineers. Following a coding standard determines how the software is designed, written, and maintained. As such, these standards play a pivotal role in minimising errors and enhancing code quality, ultimately contributing to a successful project. Software development in C and C++, a popular approach for embedded applications such as spacecraft, has many coding standards to choose from tailored for various domains. These standards address specific industry needs and ensure code quality, reliability, and safety. One prominent example used for these languages is the MISRA (Motor Industry Software Reliability Association) C and C++ coding standards. Initially targeting the automotive industry, these documents are now broadly used for safety-critical applications using these languages. However, These widely used standards do not address the risks related to multi-threading, a significant factor in space system software. Hence, the MISRA-C standard was adopted into the Jet Propulsion Laboratory (JPL) Institutional Coding Standard for the C Programming Language to mitigate these concerns [48]. Being developed and maintained by JPL, an institution renowned for its pioneering contributions to space exploration, this coding standard embodies decades of experience in the space industry. Missions such as the Ingenuity Mars Helicopter, where the JPL C coding standard was diligently followed, have exemplified its efficacy in safeguarding against software-related failures. The coding standard highlights six levels of compliance (LOC), ranging from general language compliance to specific MISRA-based *should* rules (Table 1.3), aligning with the exacting demands of space missions where a software fault could lead to mission failure.

Each LOC establishes progressively stricter rules for using the C programming language. At the base level, Language Compliance mandates fundamental practices to

TABLE 1.3: Levels of compliance for the JPL C coding standard. Adapted from [48].

Level of Compliance	**Rules Defined at Level**	**Cumulative Number of Rules Required for Full Compliance**
LOC-1 Language Compliance	2	2
LOC-2 Predictable Execution	10	12
LOC-3 Defensive Coding	7	19
LOC-4 Code Clarity	12	31
LOC-5 MISRA-shall rules	73	104
LOC-6 MISRA-should rules	16	120

ensure code safety, such as compiling software with all warning flags enabled. LOC-2 emphasises predictability in software execution, enforcing practices such as refraining from dynamic memory allocation. LOC-3, Defensive Coding, focuses on developing software capable of handling unexpected errors or inputs, enhancing overall reliability. An example of adherence to this level is using specific signed integer types instead of architecture-dependent ones for clarity and safety. LOC-4, Code Clarity, aims to improve code comprehensibility for future maintenance, employing techniques such as restricting the use of the language pre-processor. The final two levels, LOC-5 and LOC-6, reference the MISRA-C standard, which served as the basis for the JPL C standard. LOC-5's "Shall" rules are mandatory and cover essential standards, such as those prohibiting illogical variable type casting. In contrast, while optional, LOC-6's "Should" rules are crucial for full standard compliance, emphasising practices like minimising the use of the increment and decrement operators to enhance code clarity.

It is evident that ascending levels of compliance whilst following the JPL C coding standard require adhesion to an increasing number of rules. The benefit of this is that through compliance checking software, the resulting code base will have a quantifiable metric of standard compliance that provides insight to the software developers into the quality of the software being developed for their mission. When new software is being developed with adhesion to the coding standard, JPL proposes that there should not be any measurable impact on schedule or cost. This is not the case for application to code written without adhesion to the standard, with the amount of effort needed to achieve compliance increasing with each level depicted in Table 1.3 [48]. This adhesion complexity can become problematic for university-based CubeSat software developers, as it is common to see university-built CubeSats neglecting to follow well-established development methods, including project-wide adhesion to standards [49]. Hence, even if a decision is made in development teams to adopt a standard, the amount of work required to implement it retroactively may be too great, further limiting the literature applicable to university-based CubeSat software development teams.

Alongside a lack of software development experience, Alanazi et al. proposes some university groups may neglect a full requirements analysis of the CubeSat to be built, instead jumping straight to system and component level design [49]. Skipping past important design stages can result in the software failing to meet basic mission requirements. Software is often rushed through development with insufficient testing to support the rapid mission timeline [45]. Swartwout suggests that the resulting lack of time and resources allocated for systems-level testing contributes to the commonality of failure [50]. These factors make it difficult to ensure the reliability and fault-tolerance of CubeSat software systems.

One such area that may be overlooked is the consideration of radiation tolerance on satellite systems. Radiative particles originating from deep space and the Sun impose a significant challenge for satellites operating in space due to the sensitive electronics that power them [51]. Spacecraft in Earth's orbit are exposed to various radiative particles captured by the planet's magnetic field, including electrons, protons, and heavy ions [52]. As spacecraft venture beyond the Earth's magnetosphere, however, the intensity of radiation exposure increases. Exposing the sensitive electronics inside satellites to space radiation can result in long-lasting effects and immediate impacts. Long-lasting effects include those from Total Ionising Dose (TID), while immediate impacts are seen in Single Event Effects (SEE) [52]. TID refers to the cumulative damage that spacecraft components incur during prolonged exposure to a radioactive environment. This extended exposure detrimentally impacts the functionality of the integrated circuits (IC) that enable the spacecraft, degrading device voltage thresholds and increasing the probability of a component leakage current, potentially leading to irreparable faults.

SEE's relate to the transient damage experienced by ICs due to localised radiation events. In these situations, an impinging SEE-inducing particle can deposit enough energy at the storage node of some memory through ionisation to cause a change of charge state. This state change is commonly referred to as a bit-flip error, which can disrupt the normal operation of software [51]. When occurring in memory external to a flight computer, this can affect the correctness of data retrieval. When occurring internally, however, this can introduce program memory corruption, potentially resulting in processors executing erroneous instructions, posing a significant risk to overall system stability [53]. A common unit of measurement for predicting the occurrence of SEEs is the average number of SEEs that occur per bit of memory over a day. Data obtained from three NASA missions, Solar Anomalous Magnetospheric Particle Explorer (SAMPEX), Total Ozone Mapping Spectrometer (TOMS), and Cosmic Ray Upset Experiment (CRUX), provide perspective on the frequency of SEEs in various orbital altitudes close to Earth. At an orbital altitude of 580 km x 640 km, SAMPEX had an average of 4.21E-6 upsets per bit-day. In a circular orbit of 1200km, TOMS encountered an average of 3.12E-6 upsets per bit-day. Across all six independent memory units on CRUX, the mission encountered an average of 18.38E-6 upsets per bit-day at an altitude of 362 km x 2544 km [54].

In comparison, in low Earth orbit, the COTS components that usually power CubeSats generally encounter 10^{-3} to 10^{-7} upsets per bit-day [55]. The variance in average upsets per bit-day can be attributed to factors such as the size of transistors in various memory types [55], space weather events [56], as well as orbital time spent in areas of higher radiation density [54]. Despite variations, it is evident that the continuous exposure of satellite components to space radiation emphasises the necessity of taking proactive steps to avoid potentially mission-critical repercussions.

Considering the potential impact of radiation on spacecraft missions, it is common to implement design strategies focused on protecting sensitive electronics from radiation and incorporating recovery mechanisms in the event of exposure. Radiation shielding is commonly used in spacecraft structures and around sensitive components to reduce the impinging particle intensity [52]. Component selection can also alleviate radiation-induced issues through components that do not have physical bits adjacent to one another in memory to prevent multiple bit-flips per byte [54], and also through the selection of radiation-hardened devices [57]. These devices exhibit a SEE performance from 10^{-8} to 10^{-11} errors per bit-day [55].

While hardware solutions such as radiation shielding and radiation-hardened components play a crucial role in protecting space missions from radiation-induced errors, the importance of software in ensuring mission reliability cannot be overstated. Whilst significantly reducing the risk of SEE's, protective hardware measures may not eliminate them due to the high energy of the particles causing the event [55]. As a result, software mitigation techniques become indispensable for space missions. To address single-bit in-byte errors, memory scrubbing techniques employing error correction codes have recovered erroneous bit-flips [55]. In addressing multiple-bit errors within a single byte, Reed-Solomon coding techniques have been effectively employed [58]. These software-driven approaches supplement the hardware solutions to increase the overall mission reliability, mitigating the risks associated with radiation in the challenging space environment.

Although university-built CubeSats have been shown to have a failure rate of nearly 51%, it is perhaps even more crucial to note that 32% of industry-built CubeSats fail, highlighting the substantial risk for these groups. Whilst university-built CubeSats may have a high-risk appetite, this is often not reflected in industry applications that may be housing commercial payloads on board. In these situations, mission failure may bring a large monetary loss. It is difficult to quantify software's contribution to industry-built CubeSat failures as the survey results shown in Figure 1.9 are estimations regarding university-built missions. However, as the survey respondents comprised industry professionals, it could give insight into these individuals' prior experience. The high propensity for failure in CubeSats, in general, prompts the question: How much of the failures of large-scale space missions can also be attributed to software issues? Whilst CubeSats have been shown to have an astonishingly high failure rate, it is important to acknowledge that they operate under distinct constraints and conditions compared to larger, more extensively funded spacecraft. Larger spacecraft typically benefit from larger budgets, more rigorous engineering standards, and established technology and software with proven flight heritage. As a result, they tend to exhibit higher reliability when compared to CubeSats. Therefore, it is not accurate to directly extrapolate CubeSat failure rates to all types of spacecraft. However, it is crucial to recognize that rigorous software development and testing principles can benefit spacecraft of all sizes and classes. Regardless of their form factor, spacecraft require robust software systems to ensure mission success. With their quantifiable failure rates due to software, CubeSat missions serve as a valuable case study highlighting the significance of software reliability. Investing in developing reliable software can enhance the chances of success for all spacecraft, large and small. This approach can contribute to a more cost-effective, reliable future for space research and innovation, ultimately benefiting the space industry as a whole.

1.4. The Role of Autonomous Fault Prevention in Space Mission Reliability

1.4.1. State-Of-The-Art

Despite extensive efforts in programming and design, the unfortunate reality is that failures will inevitably occur during the operational phase of a space mission. By leveraging onboard autonomy for advanced fault prevention, space missions can better detect and recover from these failures, increasing their overall reliability and chance of mission success. This capability will become crucial for missions in which the time delay introduced due to the significant distance between a spacecraft and Earth can impede the recovery from safe mode, potentially leading to further complications or loss of the mission. Moreover, as the number of satellites being launched increases exponentially, mission operations for ground-in-the-loop spacecraft will become increasingly overburdened [59]. For satellites requiring assistance from an operations team when encountering a fault, this could increase the chances of faults propagating to a mission-critical failure.

The Deep Space One mission pioneered giving spacecraft more responsibility in operations. It was the first to use an onboard mission planner, which allowed the spacecraft to react quickly in suggesting an alternative operations solution if the original scheduled plan could not be carried out. The mission's success demonstrates that spacecraft operations can be autonomously reconfigurable without human intervention

[60]. A general-purpose onboard autonomy agent on the EO-1 spacecraft was partly responsible for ensuring the spacecraft's safety. An anomaly during the mission caused the spacecraft to think the camera payload cover was not closed. In response, the autonomous software prevented a hardware fault by reissuing a command to close the payload cover [61]. In the event of a major fault preventing commands from being received on the New Horizons spacecraft, engineers extensively developed an autonomy subsystem to de-spin the spacecraft. After achieving a favourable spin rate, the autonomy subsystem would orient the spacecraft into an Earth-pointing state and begin transmission of system health information [62]. However, these missions are part of a very small subset of missions that leverage onboard autonomy in this manner.

Where autonomy applications are discussed in the literature, it is commonly restricted to specific elements of mission operations, such as navigation and payload usage. Speretta et al. discuss the autonomous navigation capabilities of the LUMIO CubeSat and the autonomous methods of handling payload data [63]. Payload optimisation is leveraged on the IPEX CubeSat for data volume reduction and image analysis [64]. Beyond Earth's orbit, JAXA's Hayabusa spacecraft demonstrated successful autonomous guidance and navigation capabilities en route to the asteroid Itokawa [65]. Kohout et al. assess the feasibility of achieving a similar outcome on a CubeSat platform [66]. In the microsatellite scale, the Air Force Research Laboratory's XSS-10 mission was designed to demonstrate autonomous technologies for space situational awareness and proximity operations [67]. The literature is abundant with studies regarding autonomy used for mission operations, yet scarcely provides applications for fault prevention.

More commonly, fault preventative techniques in the literature are mainly restricted to basic Fault Detection, Isolation, and Recovery (FDIR) routines that transition the satellite into safe mode, waiting for intervention from a ground operator. Whilst being a common approach in the CubeSat industry [68][69][70][71], this pattern is also reflected in space missions in general. Often in these missions, the thresholds for entry into an FDIR routine that transitions into safe mode are set too conservatively. This can lead to high mission outage times due to unnecessary safe mode entries [72]. An example of this can be observed in missions such as ESA's Mars Express, having suffered 23 safe mode entries within the first three years of operations [73], during which the spacecraft could not complete its scientific objectives. Further continuous safe mode entries later in the mission lifecycle saw the ground team suspending science operations to conserve fuel [74]. Using an example from the Venus Express mission, Shaw et al. discuss that seeking alternative fault-preventative solutions to safe mode entries could assist in spacecraft uptime. Venus Express had 56 thermal monitoring parameters, each able to cause a safe mode entry if triggered. Instead, it was proposed that the sequence of operations with the ground following a safe mode entry could be replaced by an autonomous sequence preventing a fault occurring on the spacecraft without the risk of a safe mode transition [73].

In some situations, the overreliance on safe mode as a fault-preventative measure can sometimes introduce risk. It was discovered on the Mars Express mission that as the spacecraft reached its furthest point from the sun, the hard-coded safe mode would require more power than could be provided. Repairing this oversight was time-consuming for the mission operations team [73]. Similarly, safe mode functionality on NASA's CloudSat mission did not account for the degradation of the spacecraft's batteries, leading to more power being used than could be generated [75]. This situation was also reflected in ESA's Cluster mission [76].

Are the capabilities of autonomy in spacecraft being underestimated for fault prevention? It is apparent from the few missions that have implemented it that autonomy can improve both a spacecraft's resilience to failure and its overall uptime. It allows for the prediction and diagnosis of system failures and the ability to make decisions and take action in the event of a failure without relying on a communications link with the ground. However, many space missions minimise the scope of autonomy to mission operations, rather electing for a ground-in-the-loop approach to fault handling through safe mode transitions. The apprehension toward greater adoption of autonomy in the space industry could be motivated by the desire to minimise the costs and risks associated with development. Nevertheless, further advancements in space mission autonomy, transitioning the current ground-in-the-loop spacecraft design to a more self-sustainable system, will be paramount in overcoming the challenges imposed by deep-space missions [59]. With a relatively low mission cost and a high appetite for risk, CubeSats offer a medium to explore further development in this area, with scalable applications for larger missions.

By leveraging onboard autonomy for fault prevention, satellites can identify and address faults in real time, reducing the frequency of safe mode entries and removing the risks associated with distance-induced latency. This capability can increase the likelihood of success for the mission and extend the spacecraft's operational life. In pursuit of this, it is necessary to develop accurate models of the satellite system to implement effective algorithms for autonomous fault prevention and decision-making. Satellite developers need to refrain from settling for reactive measures when the power of autonomy can be harnessed to predict and prevent faults without interaction with the ground.

1.4.2. MODELLING

In the context of an engineering system, the term 'modelling' has historically taken on many meanings. From a physical scaled replica of a product to the development of governing equations that describe the interoperations of a system, all forms have existed to achieve the same goal - to attempt to understand the operation of a system and predict what it will do before it happens [77]. Dynamic modelling refers to the collection of equations that govern the operation of a system. The application of these equations and the subsequent analysis of their outcomes is known as a simulation [78]. As systems being developed increase in complexity, computers are often used to implement the developed models. Computational simulations allow for rapid testing of complex non-linear systems that would otherwise be tedious to complete by hand [79]. Furthermore, they allow parameters to be altered easily and have their results fed back into further development of the system [80], as seen in Figure 1.10.

A conceptual model is first developed using qualitative modelling based on the data gathered from a system being analysed. This conceptual model provides an abstract description of how the system functions on a subsystem level, focusing less on physical operation in favour of understanding and communication [81]. These subsystems can then be characterised by their governing equations to provide a quantitative representation to be fed into a computational model. However, as the real world

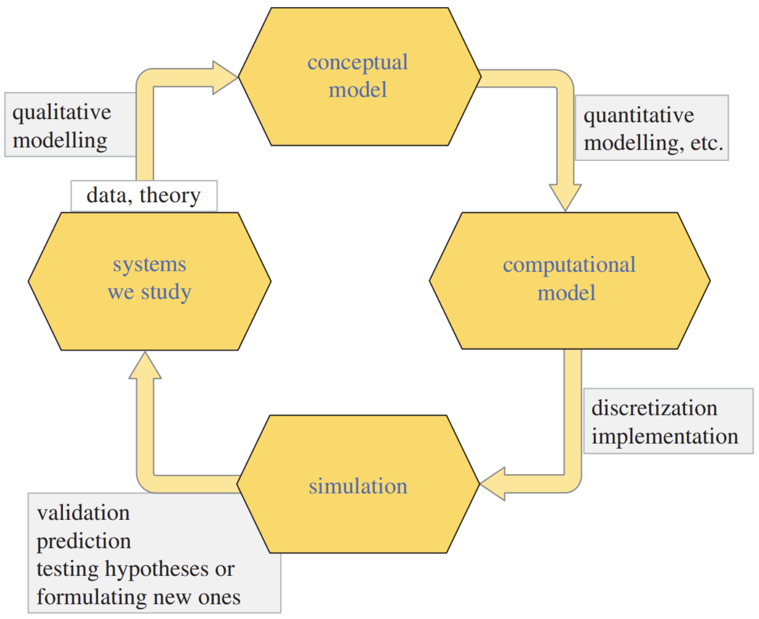


FIGURE 1.10: An overview of the modelling and simulation cycle in system development [77].

operates in a continuous-time system that cannot be truly replicated on a digital platform, the computational models are approximated as a discrete-time system that can be tested via a computer simulation with numerical integration methods. The output of these simulations offers verification of models, which can lead to predictions of the system's behaviour [77].

Traditionally, modelling and simulation were used during the research and development phases of the product lifecycle. As engineering systems continue to advance with additional functions, increased parts, and heightened interconnectivity, modelling and simulation are expanding beyond their traditional role as development tools and becoming an essential part of a system's functionality [82].

An example of this can be seen in the application of model predictive control (MPC). MPC leverages subsystem modelling in a feedback controller to address challenges concerned with nonlinear systems with multiple constraints. The control approach involves predicting future system behaviour, optimising control inputs, and introducing feedback over a finite time horizon [83]. After solving for the optimal sequence over the time horizon, only the initial control segment is applied, after which the process is repeated until a goal state is reached [59]. The utilisation of MPC onboard engineering systems with constrained system resources, such as spacecraft, introduces some limitations on the complexity of the optimisation model. Applications requiring fast resolvability are often restricted to convex optimisation routines with only one global minimum. In the spacecraft domain, Mixed-Integer Linear Programming is commonly used to handle optimisation problems with simple logical constraints [59]. Park et al. present an MPC-based approach for autonomous spacecraft rendezvous and docking with a tumbling platform [84]. In their approach, thrust, approach velocity, and spacecraft positioning within the line-of-sight cone from the docking port are treated as constraints. Whilst simulating approach trajectories, time-to-dock and fuel consumption are evaluated as cost function parameters within the MPC framework. Their simulation results demonstrate that MPC is a promising approach for spacecraft autonomy in rendezvous and docking scenarios. They also demonstrate their MPC-based approach to debris avoidance with similar success. Açikmeşe et al. introduce a robust approach to MPC in uncertain and nonlinear systems by including feedforward and feedback components in their control [85]. Their work is influenced by prior research into the autonomous guidance and control of spacecraft near small celestial bodies. The ability of the produced algorithm to handle uncertain and nonlinear systems with constraints makes it highly suitable for real-time autonomous control applications.

Moving forward, it is evident that modelling and simulation techniques are no longer confined to the developmental phase of an engineering project but have rather firmly integrated themselves into the operational core of a system driven by the increasing complexity of systems and the demand for improved performance. This evolution shapes the future of spacecraft autonomy by providing the means for operations in an environment where uncertainty and complexity are the norm. As the demands placed on engineering systems grow in complexity, however, conventional modelling and simulation methods may deviate from practical applicability. This divergence necessitates innovative approaches to bridge the gap between conceptual models and real-world systems.

1.4.3. DIGITAL TWIN

As the requirements for engineering systems become increasingly more complex, traditional modelling and simulation methods can begin to stray in similitude. Historically, this has been compensated with hardware approaches. An example of this can be seen in space systems engineering, where during NASA's Apollo program, ground-based replicas of space vehicles were built to allow for testing on the ground before launch. During the mission operation, the same ground-based spacecraft were also used to aid astronauts in orbit by simulating manoeuvres using real data from the orbiting craft. Prior testing of manoeuvres in a safe environment ensured the safety of the astronauts on orbit by analysing the outcomes of these simulations. The ground-based spacecraft was referred to as a twin [82]. As an extension of the twin concept, a virtual system model can be developed to support operations using a direct linkage to operational data to predict how a system will behave. [86]. This concept is known as a 'Digital Twin'.

The Digital Twin concept aims to address the shortcomings of the conventional simulation approach to create a virtual replica of an engineering system unparalleled in exactitude [87]. It extends the conventional modelling and simulation of an engineering system to include all interdependent systems; for a spacecraft, this includes everything from the mechanical structure through each subsystem [87]. At this scale, the spacecraft subsystems can understand their role in the operation of the spacecraft. Moving towards a large, heterogeneous, decentralised simulation model, a digital twin approaches the state of becoming "self-aware" in which subsystems possess knowledge about their state, the environment they operate in, and how they integrate with other submodules in the system [88].

Some examples in the literature regarding the application of digital twin technology to the aerospace industry demonstrate its capability in fault identification. Ye et al. demonstrate the advantages of using a digital twin approach to assess the structural health of a reusable spacecraft between flights [89]. Similarly, Tuegel et al. discuss the benefits of adopting a digital twin approach in reengineering aircraft's structural life validation process in the United States Air Force [90]. The proposed implementation would assist in identifying potential faults before they can occur, allowing for remediation before occurrence. On a smaller scale, Li et al. describe developing a digital twin model of an aircraft wing for crack growth prediction [91]. A digital twin-driven fault diagnosis and health monitoring approach for spacecraft, which tracks the on-orbit operation of a satellite by taking input from telemetry data, has been proposed by Shangguan et al. The developed prototype is designed to assist ground operators in diagnosing faults present in telemetry data [92]. Constraining their research to a single spacecraft subsystem, Peng et al. describe their approach to developing a digital twin model of spacecraft battery degradation to predict the lifetime of the cells used in a theoretical mission [93].

Whilst the digital twin examples showcase diverse uses, they all share one common trait: they operate on the ground. Currently, onboard digital twin applications are seldom mentioned in the literature. This may be attributed to the fact that high-fidelity, complex model inferencing may be incompatible with the system resources available. So, whilst deploying a digital twin onboard grants the advantage of faster data actionability, it necessitates working within the computational constraints of the device, leading to the necessity of making trade-offs in the twin's complexity. However, the latency introduced by offline inferencing may not be suitable for applications that require real-time feedback from a digital twin. Narrowing the scope of a digital twin to specific components can help alleviate the issue of computational resources. Recent work published by Qahmash et al. demonstrates the advantages of applying an onboard digital twin to solve the gripper-payload interaction challenge in an aerial robotics problem [94]. The developed onboard digital twin predicts the interactional forces between the gripper and the payload. If the measured forces during gripping deviate from the prediction from the digital twin, the system will pass to a fault handler. The onboard digital twin can immediately detect a problem as it occurs and take action on it, as opposed to requiring a ground communication loop. Digital twin systems can also be segregated into onboard and off-board modules. Huang et al. provide an innovative solution by combining onboard and off-board digital twinning for real-time health monitoring and anomaly detection in industrial systems [95]. Acknowledging that cloud platforms provide the necessary computing and storage capabilities for digital twins, ensuring accurate anomaly detection, they are leveraged for off-board digital twin tasks that require heavier computing resources, leaving edge computing onboard to focus on real-time data processing and executing anomaly detection models for real-time response. Onboard digital twins excel in real-time applications where low latency is essential. However, they fall short compared to the system resource availability of off-board digital twins. In the context of devices operating in extremely remote environments, such as deep space spacecraft, having the ability to process data locally and act swiftly without relying on external communication has enormous potential. As the desire for deep space missions continues to grow, the necessity for self-sustainable spacecraft is emphasised by requiring onboard models that mirror system modules with extreme accuracy to be integrated with physical sensors [87]. This capability onboard would allow for fault prediction and recovery, recommending changes to the mission plan whilst operating in a communications-denied environment [90].

Onboard software that can modify mission parameters and analyse their outcomes in real-time can be invaluable for deep space missions [87]; however, it can introduce risk. Whilst autonomous task generation and intelligent control are extensively used in the automotive [96], aeronautical [97], and manufacturing industries [98], the adoption of these algorithms in spacecraft is not at a similar level of maturity. With the high cost of a traditional spacecraft discussed in Section 1.1.3, the added risk of failure in implementing an onboard digital twin in a previously unexplored context has stagnated its development. The larger risk appetite of a CubeSat mission highlights the platform as the perfect test bed to develop onboard digital twin software and explore its potential for larger, more expensive missions. Demonstrating the ability to adapt, heal, and respond to changing mission tasks in an agile way onboard the CubeSat platform paves the path for all spacecraft to take advantage of fault handling, machine learning and in-orbit testing with a reduction in development risk.

Although there are similarities between an onboard digital twin and a conventional modelling approach like MPC, the two can also complement each other effectively. While both rely on a model-based approach to the decision-making process and require real-time updates to operate effectively, they target different aspects of a control strategy.

An onboard digital twin serves as a real-time, high-fidelity model of the spacecraft itself, as opposed to MPC, which is primarily a control strategy for optimising control inputs to achieve predefined objectives. The digital twin, a dynamic model that mimics a real spacecraft with continuous updates on the spacecraft's state, could be integrated into an MPC control strategy. Integrating the two approaches to autonomy could allow an MPC solution to leverage a digital twin to optimise control actions based on mission objectives and constraints.

Using onboard digital twin technology can be crucial in enabling heightened autonomy in space missions. Despite current programming and design efforts, failures will likely occur during satellite operations. The common approach for fault handling on a spacecraft is to transition into a safe mode from an FDIR routine, which introduces risk when operating in a communications-denied environment. The benefits of using autonomy for advanced fault prevention have previously been observed. However, some apprehension exists regarding a more ubiquitous adoption. The capabilities of autonomy on space missions are being underestimated. A digital twin modelling approach can revolutionise autonomous onboard fault prevention by breaking down the spacecraft systems into smaller components and thoroughly characterising each section to predict and prevent faults. This concept highlights the partial motivation for the research in this thesis: By leveraging digital twin technology for advanced fault prevention, satellite missions can better detect and recover from on-orbit failures, increasing reliability and the chances of mission success.

1.5. Overview of This Thesis

1.5.1. Reliable Software Development for Small Satellite Missions: A Binar-1 Case Study

Over the past two decades, the CubeSat platform has enabled access to research in space by driving down the costs associated with reaching orbit. As a cost-effective alternative to a traditional spacecraft, the platform has led to an influx of satellites built by universities populating LEO. The increased access to space for university groups is not without risk, however, as the skills available in the educational branch of the space industry may struggle to implement the complex software that enables spacecraft. This risk is demonstrated by the commonality of failures presented in missions developed by university groups, some of which are due to software.

Although the importance of robust software design in space missions is apparent, little published work exists regarding how to develop software for spacecraft reliably. Due to this, new entrants into the space industry may elect to use an established software solution. To fully benefit from an established software solution, the developed code base must conform to the existing structure, tightly coupling the code base to the solution used. This tight coupling can result in difficulty in breaking from the established solution if it no longer meets mission requirements or where ongoing licensing costs are undesirable.

In Chapter 2, an article under peer review in the *Journal of Small Satellites*, common pitfalls of software design leading to the failures of historical space missions are highlighted. The reoccurrence of these failures can endanger space mission success. Several software design methods and practices are demonstrated to address these pitfalls, using the Binar Software Framework written by the candidate for Binar-1 as a case study. Following these methods has aided in developing safe, reliable and reusable software for future Binar satellites. These same concepts can be applied to any spacecraft to increase the likelihood of mission success. The intended contribution of this paper is to aid in reducing common software-related failures presented in space missions.

1.5.2. BINAR-1 SOFTWARE DEVELOPMENT

An overview of the software developed for the BSF is presented in Chapter 3. This software framework was used onboard the flight computers on Binar-1 to meet mission objectives and is being integrated into Binar-2, 3 and 4. The chapter is broken into several areas that comprise the software framework; The hardware abstraction layer, general utilities, boot application modules, application modules, communications modules and operation modes. The chapter provides extensive detail on each of these libraries and, where applicable, describes the hardware implementations for the flight computers on the Binar-1 CubeSat.

The developed BSF aids in meeting regular satellite launches and provides a platform to support the development of the onboard digital twin. The framework has been developed with a core focus on preventing software faults from propagating from development through to mission operations.

1.5.3. Experimental Results and Modelling of the Binar-1 CubeSat On-Board Power Management in Thermal Vacuum Conditions

The health of the Lithium-Ion battery technology frequently used in satellite electrical power systems is known to degrade when exposed to the sub-zero environment of space. Due to this, battery heating circuitry is crucial for the longevity of a mission. On Binar-1, control over the battery heaters was implemented in analog circuitry to ensure the heaters could be enabled as soon as required without relying on software. However, this presents a challenge. As the technology responsible for heating the batteries constitutes one of the most significant demands on the CubeSat's electrical power system, its automation may result in a low-power state occurring if a high-power task is being performed simultaneously. The possibility of this highlights the necessity for a method of predicting in advance when these low power states are to occur.

In Chapter 4, an article in preparation for journal submission, computationally inexpensive models able to run onboard a CubeSat are created to predict when low-power states are to occur without requiring input from a ground operator. These models are created from data recorded whilst testing the Binar-1 CubeSat in simulated Low Earth Orbit conditions using a thermal vacuum chamber. The produced models are further verified against an additional thermal vacuum experiment and Ansys thermal model. The paper discusses the applications of this work for autonomous operations in a communications-limited environment.

1.5.4. Onboard CubeSat Digital Twin

A digital twin is a virtual representation of a system used for simulations. Using a digital twin is a common practice during the development phase of a system, yet it is frequently discarded upon completion of the system's development. However, having a high-fidelity model of a system offers a unique insight into the operations of that system during use. The developed digital twin platform extends this idea to provide a novel automation method for space missions. By deploying the digital twin onboard the spacecraft itself, information regarding the satellite's environment can be injected directly into high-fidelity simulations from the sensors on the satellite. The immediate simulation feedback allows for a heightened sense of autonomy on a spacecraft without relying on a communications link with the ground.

Chapter 5 proposes that this novel technology is crucial for spacecraft operating in a communications-limited environment that may not be able to rely on regular ground communications. It details the development of a digital twin platform capable of predicting a fault in advance and autonomously taking steps to avoid the fault from happening. It further explores the feasibility of using machine learning inferencing onboard the spacecraft as a novel approach to predicting faults in advance.

1.6. Conclusions

CubeSats have transformed the field of space research by providing a cost-effective approach to accessing space, opening up previously unavailable opportunities. In the past two decades, the platform has been observed to extend beyond the original goal of SSDL to now appeal to larger organisations interested in low-cost spacecraft. The increasing global adoption of the platform has led to the recent launch of deep-space CubeSats. As CubeSats continue to revolutionise space exploration, however, one question remains: How do we ensure their software systems are reliable and fault-tolerant? The primary focus of this thesis is to leverage the high failure rate of CubeSats to develop novel software-based techniques that enhance the reliability of spacecraft, irrespective of class. The Binar-1 platform was introduced, offering a medium to explore the contributions this research makes toward spacecraft software design.

One of the leading causes of CubeSat mission failure is software faults, resulting from a lack of development experience and neglect of established development methods. This thesis aims to address this issue by guiding spacecraft developers in implementing reliable software development practices. Onboard autonomy was introduced as a means for advanced prediction of system failures to compensate for faults occurring during mission operations. Through the development of a novel onboard spacecraft digital twin, a satellite can make decisions and take action in the event of a failure without relying on a ground link. The thesis will explore modelling to describe the behaviour of CubeSat systems to predict faults before they occur on the memory-constrained flight computer. By addressing these key challenges facing CubeSat missions, the thesis aims to contribute significantly to the field of satellite software to ensure the success and longevity of missions. With the development of a custom software framework and onboard digital twin, the boundaries of space mission autonomy and resilience can be expanded, paving the way for a new era of reliable and fault-tolerant satellite missions.

REFERENCES

[1] E. Kulu, *Nanosats Database*, en. [Online]. Available: `https://www.nanosats.eu/index.html`.

[2] H. Heidt, J. Puig-Suari, A. Moore, S. Nakasuka, and R. Twiggs, "CubeSat: A New Generation of Picosatellite for Education and Industry Low-Cost Space Experimentation," in *Proceedings of the 14th Small Satellite Conference*, Aug. 2000. [Online]. Available: `https://digitalcommons.usu.edu/smallsat/2000/All2000/32`.

[3] R. Twiggs and M. Swartwout, "SAPPHIRE – Stanford's First Amateur Satellite," Vicksburg, Mississippi, USA, Oct. 1998.

[4] B. Engberg, J. Ota, and J. Suchman, “The OPAL Satellite Project: Continuing the Next Generation of Small Satellite Development,” 1995.

[5] S. Lee, A. Hutputanasin, A. Tooriean, W. Lan, R. Munakata, J. Carnahan, D. Pignatelli, and A. Mehrparvar, *CubeSat Design Specification*, en, Feb. 2014.

[6] A. Chin, R. Coelho, R. Nugent, R. Munakata, and J. Puig-Suari, “Cubesat: The pico-satellite standard for research and education,” in *AIAA SPACE 2008 Conference*. Sep. 2008. DOI: 10.2514/6.2008-7734. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2008-7734.

[7] D. Pignatelli, R. Nugent, R. Munakata, and W. Lan, *Poly Picosatellite Orbital Deployer Mk. III Rev. E User Guide*, Mar. 2014. [Online]. Available: https://static1.squarespace.com/static/5418c831e4b0fa4ecac1bacd/t/5806854d6b8f5b8eb57b83bd/1476822350599/P-POD_MkIIIRevE_UserGuide_CP-PPODUG-1.0-1_Rev1.pdf.

[8] G. Skrobot and R. Coelho, “ELaNa – Educational Launch of Nanosatellite: Providing Routine RideShare Opportunities,” *Small Satellite Conference*, Aug. 2012. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2012/all2012/42.

[9] *JEM Payload Accommodation Handbook*, Nov. 2018. [Online]. Available: https://iss.jaxa.jp/kiboexp/equipment/ef/jssod/images/jpah_vol.8_b_en.pdf.

[10] M. Shan, J. Guo, and E. Gill, “Review and comparison of active space debris capturing and removal methods,” en, *Progress in Aerospace Sciences*, vol. 80, pp. 18–32, Jan. 2016, ISSN: 03760421. DOI: 10.1016/j.paerosci.2015.11.001. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0376042115300221.

[11] E. S. Nightingale, L. M. Pratt, and A. Balakrishnan, “The CubeSat Ecosystem: Examining the Launch Niche,” Jerusalem, Israel, Dec. 2015. [Online]. Available: https://www.ida.org/-/media/feature/publications/t/th/the-cubesat-ecosystem-examining-the-launch-niche-paper-and-presentation/d-5678.ashx?la=en&hash=320B4F2FAC75A8B30350A1471EEE7AB8.

[12] M. Swartwout, "The First One Hundred CubeSats: A Statistical Look," en, *Journal of Small Satellites*, p. 21, 2013.

[13] S. W. Asmar and S. Matousek, "Mars Cube One (MarCO) Shifting the Paradigm in Relay Deep Space Operation," en, in *SpaceOps 2016 Conference*, American Institute of Aeronautics and Astronautics, May 2016, ISBN: 978-1-62410-426-8. DOI: 10.2514/6.2016-2483. [Online]. Available: http://arc.aiaa.org/doi/10.2514/6.2016-2483.

[14] Andrew Good and JoAnna Wendel, *The MarCO Mission Comes to an End*, Feb. 2020. [Online]. Available: https://mars.nasa.gov/news/8408/the-marco-mission-comes-to-an-end.

[15] L. McNutt, L. Johnson, P. Kahn, J. Castillo-Rogez, and A. Frick, "Near-Earth Asteroid (NEA) Scout," en, in *AIAA SPACE 2014 Conference and Exposition*, San Diego, CA: American Institute of Aeronautics and Astronautics, Aug. 2014, ISBN: 978-1-62410-257-8. DOI: 10.2514/6.2014-4435. [Online]. Available: http://arc.aiaa.org/doi/10.2514/6.2014-4435.

[16] M. I. Desai, F. Allegrini, R. W. Ebert, K. Ogasawara, M. E. Epperly, D. E. George, E. R. Christian, S. G. Kanekal, N. Murphy, and B. Randol, "The CubeSat Mission to Study Solar Particles," *IEEE Aerospace and Electronic Systems Magazine*, vol. 34, no. 4, pp. 16–28, Apr. 2019, ISSN: 1557-959X. DOI: 10.1109/MAES.2019.2917802.

[17] L. Mohon, *NEA Scout Status Update*, Text, Dec. 2022. [Online]. Available: http://www.nasa.gov/centers/marshall/news/2022/nea-scout-status-update.html.

[18] Denise Hill, *Artemis I Payload CuSP CubeSat Mission Update*, Text, Dec. 2022. [Online]. Available: https://blogs.nasa.gov/sunspot/2022/12/08/artemis-i-payload-cusp-cubesat-mission-update/.

[19] Pumpkin, Inc., *Motherboard Module (MBM)*, en, Store, Feb. 2023. [Online]. Available: http://www.pumpkinspace.com/store/p49/Motherboard_Module_%28MBM%29.html.

[20] Pumpkin, Inc., *Intelligent Protected Lithium Battery Module with SoC Reporting (BM2)*, en, Store, Feb. 2023. [Online]. Available: `http://www.pumpkinspace.com/store/p198/Intelligent_Protected_Lithium_Battery_Module_with_SoC_Reporting_%28BM2%29.html`.

[21] Endurosat, *1U CubeSat Solar Panel X/Y*, en-US, Store, Feb. 2023. [Online]. Available: `https://www.endurosat.com/cubesat-store/cubesat-solar-panels/1u-solar-panel-x-y/`.

[22] Pumpkin, Inc., *GNSS Receiver Module (GPSRM 1) Kit*, en, Store, Feb. 2023. [Online]. Available: `http://www.pumpkinspace.com/store/p58/GNSS_Receiver_Module_%28GPSRM_1%29_Kit.html`.

[23] Endurosat, *UHF Transceiver II CubeSat Communication*, en-US, Store, Feb. 2023. [Online]. Available: `https://www.endurosat.com/cubesat-store/cubesat-communication-modules/uhf-transceiver-ii/`.

[24] Endurosat, *UHF CubeSat Antenna Module*, en-US, Feb. 2023. [Online]. Available: `https://www.endurosat.com/cubesat-store/cubesat-antennas/uhf-antenna/`.

[25] Endurosat, *1U CubeSat Structure*, en-US, Store, Feb. 2023. [Online]. Available: `https://www.endurosat.com/cubesat-store/cubesat-structures/1u-cubesat-structure/`.

[26] Tensor Tech, *Tensor Tech ADCS - MTQ Integrated Attitude Determination and Control System with Magnetorquers*, en-US, shop, Oct. 2023. [Online]. Available: `https://www.cubesatshop.com/product/tensor-tech-adcs-mtq-integrated-attitude-determination-and-control-system-with-magnetorquers/` (visited on 10/03/2023).

[27] A. E. Kalman, *CubeSat Kit Motherboard (MB)*, en, Mar. 2012. [Online]. Available: `http://www.pumpkininc.com/space/datasheet/710-00484-E_DS_MBM.pdf`.

[28] A. E. Kalman, *CubeSat Kit™ Battery Module 2 (BM 2)*, Sep. 2022. [Online]. Available: `http://www.pumpkininc.com/space/datasheet/710-01640-F_DS_BM_2.pdf`.

[29] A. E. Kalman, *CubeSat Kit™ GPSRM 1 GNSS Receiver Module*, Aug. 2022. [Online]. Available: `http://www.pumpkininc.com/space/datasheet/710-00908-D_DS_GPSRM_1.pdf`.

[30] *User Manual UHF Transceiver Type II*, en, Nov. 2018.

[31] *Tensor Tech Brochure*, 2022. [Online]. Available: `https://www.cubesatshop.com/wp-content/uploads/2023/01/Tensor-Tech-Brochure-2022-1-compressed.pdf` (visited on 10/03/2023).

[32] *SpaceX Smallsat Rideshare Program*, en. [Online]. Available: `http://www.spacex.com` (visited on 10/13/2023).

[33] J. Bouwmeester, M. Langer, and E. Gill, "Survey on the implementation and reliability of CubeSat electrical bus interfaces," en, *CEAS Space Journal*, vol. 9, no. 2, pp. 163–173, Jun. 2017, ISSN: 1868-2502, 1868-2510. DOI: `10.1007/s12567-016-0138-0`. [Online]. Available: `http://link.springer.com/10.1007/s12567-016-0138-0`.

[34] J. Bouwmeester and N. Santos, "Analysis of the Distribution of Electrical Power in Cubesats," en, in *Proceedings of The 4S Symposium*, Spain: ESA, CNES, 2014, p. 12.

[35] G. F. Dubos, J.-F. Castet, and J. H. Saleh, "Statistical reliability analysis of satellites by mass category: Does spacecraft size matter?" en, *Acta Astronautica*, vol. 67, no. 5, pp. 584–595, Sep. 2010, ISSN: 0094-5765. DOI: `10.1016/j.actaastro.2010.04.017`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0094576510001347`.

[36] E. National Academies of Sciences and Medicine, *Achieving Science with CubeSats: Thinking Inside the Box*. Washington, DC: The National Academies Press, 2016, ISBN: 978-0-309-44263-3. DOI: `10.17226/23503`. [Online]. Available: `https://nap.nationalacademies.org/catalog/23503/achieving-science-with-cubesats-thinking-inside-the-box`.

[37] M. Swartwout, *Mission Success for Universities and Professional Programs*, en, Jan. 2023. [Online]. Available: `https://sites.google.com/a/slu.edu/swartwout/cubesat-database/repeat-success`.

[38] G. Johnson-Roth, “Mission Assurance Guidelines for AD Mission Risk Classes,” *Aerospace Corporation, TOR-2011 (8591)-21*, 2011.

[39] Committee on Achieving Science Goals with CubeSats, Space Studies Board, Division on Engineering and Physical Sciences, and National Academies of Sciences, Engineering, and Medicine, *Achieving Science with CubeSats: Thinking Inside the Box*, en. Washington, D.C.: National Academies Press, Oct. 2016, ISBN: 978-0-309-44263-3. DOI: 10.17226/23503. [Online]. Available: https://www.nap.edu/catalog/23503 (visited on 10/03/2023).

[40] D. Dvorak, “NASA Study on Flight Software Complexity,” en, in *AIAA Infotech@Aerospace Conference*, Seattle, Washington: American Institute of Aeronautics and Astronautics, Apr. 2009, ISBN: 978-1-60086-979-2. DOI: 10.2514/6.2009-1882. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2009-1882.

[41] M. Langer and J. Bouwmeester, “Reliability of CubeSats - Statistical Data, Developers’ Beliefs and the Way Forward,” en, in *Proceedings of the AIAA/USU Conference on Small Satellites*, Aug. 2016, p. 12. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2016/TS10AdvTech2/4.

[42] J. Bouwmeester and M. Langer, *Results of CubeSat Survey on Electrical Interfaces & Reliability*, en, 2016. DOI: 10.4121/UUID:591FF8F8-B495-4D1C-9C5C-69F6F85ACE78. [Online]. Available: https://data.4tu.nl/articles/_/12694949/1.

[43] D. José Franzim Miranda, M. Ferreira, F. Kucinskis, and D. McComas, “A Comparative Survey on Flight Software Frameworks for ‘New Space’ Nanosatellite Missions,” en, *Journal of Aerospace Technology and Management*, e4619, Oct. 2019, ISSN: 2175-9146. DOI: 10.5028/jatm.v11.1081. [Online]. Available: http://www.scielo.br/scielo.php?script=sci_arttext&pid=S2175-91462019000100341&lng=en&nrm=iso&tlng=en.

[44] Jamie Chin, Roland Coelho, Justin Foley, Alicia Johnstone, Ryan Nugent, Dave Pignatelli, Savannah Pignatelli, Nikolaus Powell, Jordi Puig-Suari, William Atkinson, Jennifer Dorsey, Scott Higginbotham, Maile Krienke, Kristina Nelson, Bradley Poffenberger, Creg Raffington, Garrett Skrobot, Justin Treptow,

Anne Sweet, Jason Crusan, Carol Galica, William Horne, Charles Norton, and Alan Robinson, *CubeSat 101: Basic Concepts and Processes for First-Time CubeSat Developers*, en, Oct. 2017.

[45] M. D. Grubb, "Increasing the Reliability of Software Systems on Small Satellites Using Software-Based Simulation of the Embedded System," English, Master's thesis, West Virginia University, United States – West Virginia, 2021. [Online]. Available: `https://www.proquest.com/docview/2563685922/abstract/BD0ACAD670454C57PQ/1`.

[46] J. Eickhoff, *Onboard Computers, Onboard Software and Satellite Operations*, en, ser. Springer Aerospace Technology. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ISBN: 978-3-642-25170-2. DOI: `10.1007/978-3-642-25170-2`. [Online]. Available: `http://link.springer.com/10.1007/978-3-642-25170-2`.

[47] F. Stesina and S. Corpino, "Investigation of a CubeSat in Orbit Anomaly through Verification on Ground," en, *Aerospace*, vol. 7, no. 4, p. 38, Apr. 2020, ISSN: 2226-4310. DOI: `10.3390/aerospace7040038`. [Online]. Available: `https://www.mdpi.com/2226-4310/7/4/38`.

[48] *JPL Institutional Coding Standard for the C Programming Language*, en, Mar. 2009.

[49] A. Alanazi and J. Straub, "Engineering Methodology for Student-Driven CubeSats," en, *Aerospace*, vol. 6, no. 5, p. 54, May 2019, ISSN: 2226-4310. DOI: `10.3390/aerospace6050054`. [Online]. Available: `https://www.mdpi.com/2226-4310/6/5/54`.

[50] M. Swartwout and C. Jayne, "University-Class Spacecraft by the Numbers: Success, Failure, Debris. (But Mostly Success.)," *Small Satellite Conference*, Aug. 2016. [Online]. Available: `https://digitalcommons.usu.edu/smallsat/2016/TS13Education/1`.

[51] G. C. Messenger and M. S. Ash, *Single Event Phenomena*, en. Boston, MA: Springer US, 1997, ISBN: 978-1-4615-6043-2. DOI: `10.1007/978-1-4615-6043-2`. [Online]. Available: `http://link.springer.com/10.1007/978-1-4615-6043-2` (visited on 10/12/2023).

[52] E. Stassinopoulos and J. Raymond, "The space radiation environment for electronics," *Proceedings of the IEEE*, vol. 76, no. 11, pp. 1423–1442, Nov. 1988, ISSN: 1558-2256. DOI: 10.1109/5.90113.

[53] P. Madle, "STM32H7 Radiation Test Report," Open Source Satellite, Tech. Rep., May 2021. [Online]. Available: https://www.opensourcesatellite.org/downloads/KS-DOC-01251_STM32H7_Radiation_Test_Report.pdf (visited on 06/01/2022).

[54] C. Seidleck, K. LaBel, A. Moran, M. Gates, J. Barth, E. Stassinopoulos, and T. Gruner, "Single event effect flight data analysis of multiple NASA spacecraft and experiments; implications to spacecraft electrical designs," en, in *Proceedings of the Third European Conference on Radiation and its Effects on Components and Systems*, Arcachon, France: IEEE, 1996, pp. 581–588, ISBN: 978-0-7803-3093-1. DOI: 10.1109/RADECS.1995.509840. [Online]. Available: http://ieeexplore.ieee.org/document/509840/ (visited on 10/11/2023).

[55] R. C. Moore, "Satellite RF Communications and Onboard Processing," en, in *Encyclopedia of Physical Science and Technology*, Elsevier, 2003, pp. 439–455, ISBN: 978-0-12-227410-7. DOI: 10.1016/B0-12-227410-5/00884-X. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/B012227410500884X (visited on 10/12/2023).

[56] A. D. P. Hands, K. A. Ryden, N. P. Meredith, S. A. Glauert, and R. B. Horne, "Radiation Effects on Satellites During Extreme Space Weather Events," en, *Space Weather*, vol. 16, no. 9, pp. 1216–1226, 2018, ISSN: 1542-7390. DOI: 10.1029/2018SW001913. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1029/2018SW001913 (visited on 10/11/2023).

[57] K. E. Holbert and L. T. Clark, "Radiation Hardened Electronics Destined For Severe Nuclear Reactor Environments," en, Tech. Rep. DOE-ASU–NE00679, 1238384, Feb. 2016, DOE–ASU–NE00679, 1 238 384. DOI: 10.2172/1238384. [Online]. Available: http://www.osti.gov/servlets/purl/1238384/ (visited on 06/01/2022).

[58] C. Underwood, R. Ecoffet, S. Duzeffier, and D. Faguere, "Observations Of Single-event Upset And Multiple-bit Upset In Non-hardened High-density

SRAMs In The TOPEX/ Poseidon Orbit," in *1993 IEEE Radiation Effects Data Workshop*, Snowbird, UT, USA: IEEE, 1993, pp. 85–92, ISBN: 978-0-7803-1906-6. DOI: 10.1109/REDW.1993.700572. [Online]. Available: http://ieeexplore.ieee.org/document/700572/ (visited on 10/12/2023).

[59] J. A. Starek, B. Açıkmeşe, I. A. Nesnas, and M. Pavone, "Spacecraft Autonomy Challenges for Next-Generation Space Missions," en, in *Advances in Control System Technology for Aerospace Applications*, ser. Lecture Notes in Control and Information Sciences, E. Feron, Ed., Berlin, Heidelberg: Springer, 2016, pp. 1–48, ISBN: 978-3-662-47694-9. DOI: 10.1007/978-3-662-47694-9_1. [Online]. Available: https://doi.org/10.1007/978-3-662-47694-9_1.

[60] D. Bernard, E. Gamble, G. Man, G. Dorais, B. Kanefsky, W. Millar, K. Rajan, J. Kurien, N. Muscettola, and P. Nayak, "Spacecraft autonomy flight experience - The DS1 Remote Agent Experiment," en, in *Space Technology Conference and Exposition*, Albuquerque,NM,U.S.A.: American Institute of Aeronautics and Astronautics, Sep. 1999. DOI: 10.2514/6.1999-4512. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.1999-4512.

[61] D. Q. Tran, S. Chien, G. Rabideau, and B. Cichy, *Safe agents in space : Preventing and responding to Anomalies in the Autonomous Sciencecraft Experiment*, en, Jul. 2005. [Online]. Available: https://trs.jpl.nasa.gov/handle/2014/41168.

[62] B. A. Bauer and W. M. Reid, "Automating the Pluto Experience: An Examination of the New Horizons Autonomous Operations Subsystem," en, in *2007 IEEE Aerospace Conference*, Big Sky, MT, USA: IEEE, 2007, pp. 1–10, ISBN: 978-1-4244-0524-4. DOI: 10.1109/AERO.2007.352645. [Online]. Available: http://ieeexplore.ieee.org/document/4161523/.

[63] S. Speretta, F. Topputo, J. Biggs, P. Di Lizia, M. Massari, K. Mani, D. Dei Tos, S. Ceccherini, V. Franzese, A. Cervone, P. Sundaramoorthy, R. Noomen, S. Mestry, A. d. C. Cipriano, A. Ivanov, D. Labate, L. Tommasi, A. Jochemsen, J. Gailis, R. Furfaro, V. Reddy, J. Vennekens, and R. Walker, "LUMIO: Achieving autonomous operations for Lunar exploration with a CubeSat," en, in *2018 SpaceOps Conference*, Marseille, France: American Institute of Aeronautics and Astronautics, May 2018, ISBN: 978-1-62410-562-3. DOI: 10.2514/6.2018-259

9. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2018-2599.

[64] J. Doubleday, S. Chien, C. Norton, K. Wagstaff, D. R. Thompson, J. Bellardo, C. Francis, and E. Baumgarten, "Autonomy for remote sensing — Experiences from the IPEX CubeSat," in *2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, Jul. 2015, pp. 5308–5311. DOI: 10.1109/IGARSS.2015.7327033.

[65] M. Uo, K. Shirakawa, T. Hasimoto, T. Kubota, and J. Kawaguchi, "Hayabusa Touching-Down to Itokawa - Autonomous Guidance and Navigation," *The Journal of Space Technology and Science*, vol. 22, no. 1, 1_32–1_41, 2006. DOI: 10.11230/jsts.22.1_32.

[66] T. Kohout, A. Näsilä, T. Tikka, M. Granvik, A. Kestilä, A. Penttilä, J. Kuhno, K. Muinonen, K. Viherkanto, and E. Kallio, "Feasibility of asteroid exploration using CubeSats—ASPECT case study," en, *Advances in Space Research*, Past, Present and Future of Small Body Science and Exploration, vol. 62, no. 8, pp. 2239–2244, Oct. 2018, ISSN: 0273-1177. DOI: 10.1016/j.asr.2017.07.036. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S027311771730546X.

[67] T. M. Davis and D. Melanson, "XSS-10 microsatellite flight demonstration program results," in *Spacecraft Platforms and Infrastructure*, P. T. Jr. and M. Wright, Eds., International Society for Optics and Photonics, vol. 5419, SPIE, 2004, pp. 16–25. DOI: 10.1117/12.544316. [Online]. Available: https://doi.org/10.1117/12.544316.

[68] H. Leppinen, P. Niemelä, N. Silva, H. Sanmark, H. Forstén, A. Yanes, R. Modrzewski, A. Kestilä, and J. Praks, "Developing a Linux-based nanosatellite on-board computer: Flight results from the Aalto-1 mission," *IEEE Aerospace and Electronic Systems Magazine*, vol. 34, no. 1, pp. 4–14, Jan. 2019, ISSN: 1557-959X. DOI: 10.1109/MAES.2019.170217.

[69] I. Latachi, T. Rachidi, M. Karim, and A. Hanafi, "Reusable and Reliable Flight-Control Software for a Fail-Safe and Cost-Efficient Cubesat Mission: Design and Implementation," en, *Aerospace*, vol. 7, no. 10, p. 146, Oct. 2020, ISSN: 2226-4310.

DOI: 10.3390/aerospace7100146. [Online]. Available: https://www.mdpi.com/2226-4310/7/10/146.

[70] A. Almazrouei, A. Khan, A. Almesmari, A. Albuainain, A. Bushlaibi, A. Al Mahmood, A. Alqaraan, A. Alhammadi, A. AlBalooshi, A. Khater, A. Alharam, B. AlTawil, B. Alkhzaimi, E. Almansoori, F. Jarrar, H. Issa, H. Alblooshi, M. T. Ansari, N. Alzaabi, N. Braik, P. Dimitropoulos, P. Marpu, R. Alialali, R. Alhammadi, R. Al-haddad, S. Almazrouei, S. Bahumaish, V. Thu, and Y. Alqassab, "A Complete Mission Concept Design and Analysis of the Student-Led CubeSat Project: Light-1," en, *Aerospace*, vol. 8, no. 9, p. 247, Sep. 2021, ISSN: 2226-4310. DOI: 10.3390/aerospace8090247. [Online]. Available: https://www.mdpi.com/2226-4310/8/9/247.

[71] M. Smith, A. Donner, M. Knapp, C. Pong, C. Smith, J. Luu, P. Pasquale, and B. Campuzano, "On-Orbit Results and Lessons Learned from the ASTERIA Space Telescope Mission," *Small Satellite Conference*, Aug. 2018. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2018/all2018/255.

[72] A. Wander and R. Förstner, "Innovative Fault Detection, Isolation and Recovery Strategies On-Board Spacecraft: State of the Art and Research Challenges," en, *Conference on Control and Fault-Tolerant Systems, SysTol*, p. 9, 2012. DOI: 10.1109/SysTol.2013.6693950.

[73] M. Shaw, M. Denis, O. Camino, A. Accomazzo, and P. Jayaraman, "Lessons Learned from Safe Modes on ESA's Interplanetary Missions: Mars Express, Venus Express and Rosetta," in *SpaceOps 2008 Conference*, May 2008. DOI: 10.2514/6.2008-3257. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2008-3257.

[74] D. Lakey, "FAST: A new MEX Operations concept, quickly!" en, in *SpaceOps 2012 Conference*, Stockholm, Sweden: American Institute of Aeronautics and Astronautics, Jun. 2012. DOI: 10.2514/6.2012-1257123. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2012-1257123.

[75] M. Nayak, "CloudSat Anomaly Recovery and Operational Lessons Learned," en, in *SpaceOps 2012 Conference*, Stockholm, Sweden: American Institute of Aeronautics and Astronautics, Jun. 2012. DOI: 10.2514/6.2012-1295798.

[Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2012-1295798.

[76] I. Clerigo, S. Sangiorgi, and J. Volpp, “Avoiding Cluster Safe Modes,” en, in *SpaceOps 2012 Conference*, Stockholm, Sweden: American Institute of Aeronautics and Astronautics, Jun. 2012. DOI: 10.2514/6.2012-1262380. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2012-1262380.

[77] A. G. Hoekstra, B. Chopard, D. Coster, S. Portegies Zwart, and P. V. Coveney, “Multiscale computing for science and engineering in the era of exascale performance,” en, *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 377, no. 2142, p. 20 180 144, Apr. 2019, ISSN: 1364-503X, 1471-2962. DOI: 10.1098/rsta.2018.0144. [Online]. Available: https://royalsocietypublishing.org/doi/10.1098/rsta.2018.0144.

[78] P. E. Wellstead, *Introduction to physical system modelling*, en. London ; New York: Academic Press, 1979, ISBN: 978-0-12-744380-5.

[79] C. Taber and R. Timpone, *Computational Modeling*. Thousand Oaks, California, 1996. DOI: 10.4135/9781412983716. [Online]. Available: https://methods.sagepub.com/book/computational-modeling.

[80] S. J. Moss and P. Davidsson, Eds., *Multi-Agent-Based Simulation: second international workshop, MABS 2000, Boston, MA, USA, July: revised and additional papers*, en, ser. Lecture notes in computer science ; 1979. Lecture notes in artificial intelligence. Berlin ; New York: Springer, 2001, ISBN: 978-3-540-41522-0.

[81] J. Mylopoulos, *Conceptual Modelling and Telos*, en, 1992.

[82] P. Hehenberger and D. Bradley, *Mechatronic Futures: Challenges and Solutions for Mechatronic Systems and their Designers*. Springer International Publishing, 2016, ISBN: 978-3-319-32156-1. [Online]. Available: https://books.google.com.au/books?id=LAlkDAAAQBAJ.

[83] B. Kouvaritakis and M. Cannon, *Model Predictive Control*, en, ser. Advanced Textbooks in Control and Signal Processing. Cham: Springer International Publishing, 2016, ISBN: 978-3-319-24853-0. DOI: 10.1007/978-3-319-24853-0.

[Online]. Available: http://link.springer.com/10.1007/978-3-319-24853-0 (visited on 10/31/2023).

[84] H. Park, S. Di Cairano, and I. Kolmanovsky, "Model Predictive Control for spacecraft rendezvous and docking with a rotating/tumbling platform and for debris avoidance," Jun. 2011, pp. 1922–1927. DOI: 10.1109/ACC.2011.5991151.

[85] B. Açıkmeşe, J. M. Carson III, and D. S. Bayard, "A robust model predictive control algorithm for incrementally conic uncertain/nonlinear systems," en, *International Journal of Robust and Nonlinear Control*, vol. 21, no. 5, pp. 563–590, 2011, ISSN: 1099-1239. DOI: 10.1002/rnc.1613. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rnc.1613 (visited on 10/31/2023).

[86] R. Rosen, G. von Wichert, G. Lo, and K. D. Bettenhausen, "About The Importance of Autonomy and Digital Twins for the Future of Manufacturing," en, *IFAC-PapersOnLine*, vol. 48, no. 3, pp. 567–572, 2015, ISSN: 24058963. DOI: 10.1016/j.ifacol.2015.06.141. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2405896315003808.

[87] E. Glaessgen and D. Stargel, "The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles," in *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, American Institute of Aeronautics and Astronautics, Jun. 2012. DOI: 10.2514/6.2012-1818. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2012-1818 (visited on 11/02/2023).

[88] F. Faniyi, P. R. Lewis, R. Bahsoon, and X. Yao, "Architecting Self-Aware Software Systems," en, in *2014 IEEE/IFIP Conference on Software Architecture*, Sydney, Australia: IEEE, Apr. 2014, pp. 91–94, ISBN: 978-1-4799-3412-6. DOI: 10.1109/WICSA.2014.18. [Online]. Available: http://ieeexplore.ieee.org/document/6827105/.

[89] Y. Ye, Q. Yang, F. Yang, Y. Huo, and S. Meng, "Digital twin for the structural health management of reusable spacecraft: A case study," en, *Engineering Fracture Mechanics*, vol. 234, p. 107 076, Jul. 2020, ISSN: 0013-7944. DOI: 10.1016/j

.engfracmech.2020.107076. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0013794419315759.

[90] E. J. Tuegel, A. R. Ingraffea, T. G. Eason, and S. M. Spottswood, "Reengineering Aircraft Structural Life Prediction Using a Digital Twin," en, *International Journal of Aerospace Engineering*, vol. 2011, pp. 1–14, 2011, ISSN: 1687-5966, 1687-5974. DOI: 10.1155/2011/154798. [Online]. Available: http://www.hindawi.com/journals/ijae/2011/154798/.

[91] C. Li, S. Mahadevan, Y. Ling, S. Choze, and L. Wang, "Dynamic Bayesian Network for Aircraft Wing Health Monitoring Digital Twin," en, *AIAA Journal*, vol. 55, no. 3, pp. 930–941, Mar. 2017, ISSN: 0001-1452, 1533-385X. DOI: 10.2514/1.J055201. [Online]. Available: https://arc.aiaa.org/doi/10.2514/1.J055201.

[92] D. Shangguan, L. Chen, and J. Ding, "A Digital Twin-Based Approach for the Fault Diagnosis and Health Monitoring of a Complex Satellite System," en, *Symmetry*, vol. 12, no. 8, Aug. 2020, ISSN: 2073-8994. DOI: 10.3390/sym12081307. [Online]. Available: https://www.mdpi.com/2073-8994/12/8/1307 (visited on 11/02/2023).

[93] Y. Peng, X. Zhang, Y. Song, and D. Liu, "A Low Cost Flexible Digital Twin Platform for Spacecraft Lithium-ion Battery Pack Degradation Assessment," in *2019 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*, May 2019, pp. 1–6. DOI: 10.1109/I2MTC.2019.8827160.

[94] A. Qahmash, I. Al-Darraji, A. O. Khadidos, G. Tsaramirsis, A. O. Khadidos, and M. Alghamdi, "On-Board Digital Twin Based on Impedance and Model Predictive Control for Aerial Robot Grasping," en, *Journal of Sensors*, vol. 2023, May 2023, ISSN: 1687-725X. DOI: 10.1155/2023/9662100. [Online]. Available: https://www.hindawi.com/journals/js/2023/9662100/ (visited on 10/29/2023).

[95] H. Huang, L. Yang, Y. Wang, X. Xu, and Y. Lu, "Digital Twin-driven online anomaly detection for an automation system based on edge intelligence," *Journal of Manufacturing Systems*, vol. 59, pp. 138–150, Apr. 2021, ISSN: 0278-6125. DOI: 10.1016/j.jmsy.2021.02.010. [Online]. Available: https://www.scie

ncedirect.com/science/article/pii/S0278612521000467 (visited on 10/30/2023).

[96] R. Magargle, L. Johnson, P. Mandloi, P. Davoudabadi, O. Kesarkar, S. Krishnaswamy, J. Batteh, and A. Pitchaikani, “A Simulation-Based Digital Twin for Model-Driven Health Monitoring and Predictive Maintenance of an Automotive Braking System,” en, in *Proceedings of the 12th International Modelica Conference*, Jul. 2017, pp. 35–46. DOI: 10.3384/ecp1713235. [Online]. Available: http://www.ep.liu.se/ecp/article.asp?issue=132%26article=3.

[97] E. M. Kraft, “The Air Force Digital Thread/Digital Twin - Life Cycle Integration and Use of Computational and Experimental Knowledge,” en, in *54th AIAA Aerospace Sciences Meeting*, San Diego, California, USA: American Institute of Aeronautics and Astronautics, Jan. 2016, ISBN: 978-1-62410-393-3. DOI: 10.2514/6.2016-0897. [Online]. Available: http://arc.aiaa.org/doi/10.2514/6.2016-0897.

[98] F. Tao and M. Zhang, “Digital Twin Shop-Floor: A New Shop-Floor Paradigm Towards Smart Manufacturing,” en, *IEEE Access*, vol. 5, 2017, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2756069. [Online]. Available: http://ieeexplore.ieee.org/document/8049520/.

CHAPTER 2

Reliable Software Development for Small Satellite Missions: A Binar-1 Case Study

Stuart R. G. Buchan[1], Nathaniel D. Brough[1], Fergus W. Downey[1], Daniel C. Busan[1], Benjamin A. D. Hartig[1], Robert M. Howie[1], Phil A. Bland[1], Jonathan Paxman[2], and Martin Cupak[1]

[1]Space Science and Technology Centre, Curtin University, Perth, Australia

[2]Department of Mechanical Engineering, Curtin University, Perth, Australia

This article is undergoing peer review after submission to the Journal of Small Satellites. The formatting and referencing style have been modified for presentation in this thesis.

AUTHORSHIP DECLARATION

Article Title: Reliable Software Development for Small Satellite Missions: A Binar-1 Case Study
Journal: Journal of Small Satellites
Publication Status: Undergoing peer review

AUTHOR CONTRIBUTIONS

STUART BUCHAN

Investigated and analysed the reasons for spacecraft software failures, conceptualised a methodology for writing reliable software for small satellite missions, developed and validated the software for the Binar Software Framework, and wrote the manuscript.
Manuscript contribution: 81%

NATHANIEL BROUGH

Contributed to the conceptualisation of a methodology for writing reliable software for small satellite missions, contributed to the software development and validation, and reviewed the manuscript.
Manuscript contribution: 11%

FERGUS DOWNEY

Contributed to the conceptualisation of a methodology for writing reliable software for small satellite missions and reviewed the manuscript.
Manuscript contribution: 2%

DANIEL BUSAN

Contributed to the software development and reviewed the manuscript.
Manuscript contribution: 1%

BENJAMIN HARTIG

Provided project administration and reviewed the manuscript.

Manuscript contribution: 1%

ROBERT HOWIE

Provided research guidance and reviewed the manuscript.

Manuscript contribution: 1%

PHIL BLAND

Provided research guidance and reviewed the manuscript.

Manuscript contribution: 1%

JONATHAN PAXMAN

Provided research guidance and reviewed the manuscript.

Manuscript contribution: 1%

MARTIN CUPAK

Reviewed the manuscript.

Manuscript contribution: 1%

ABSTRACT

As the computers enabling small spacecraft increase in capability and decrease in cost, the role that flight software plays in mission potential is becoming increasingly important. Consequently, common pitfalls that are inherent in software design can endanger small spacecraft mission success. A review of space mission mishaps of the past two decades reveals that code reuse, software testing and poor code base maintenance were frequently responsible. This paper provides recommendations for software design for small spacecraft to increase the likelihood of success for small satellite missions, using Binar-1, Western Australia's first locally developed spacecraft, as a case study. In the first application of the SOLID principles of object-oriented programming (a set of principles intended to increase the maintainability and flexibility of a code base) to spacecraft software development, we demonstrate safe code portability between missions, and an increase in test coverage of the code base through test-driven development. Good code base maintenance is discussed with a focus on creating a development pipeline involving continuous integration and continuous deployment of spacecraft software. Implementing the recommendations for small spacecraft software design on the Binar platform has resulted in the development of safe, reliable and reusable software.

2.1. INTRODUCTION

In a 2005 study of 156 on-orbit spacecraft failures, Tafazoli found that six percent were attributed to incorrect software commands and flaws, demonstrating the importance of a robust software design methodology to a successful mission [99]. Seventeen years later, as more functionality is implemented in, or enabled by software running on smaller, cheaper, and more powerful computing equipment, this is becoming more true than ever [100]. However, the CubeSat form factor and onset of electronics miniaturisation has dramatically reduced the cost required to conduct research in space, enabling universities to explore this field with teams of students. This is not without risk, however, as the skills available in the educational branch of the space industry may struggle to safely implement the complex software powering future spacecraft.

This is reflected in the commonality of mission-critical risks presenting in the spacecraft being produced from these groups [69].

There is little published work documenting the role software plays in CubeSat failures [69]. Due to this, new developers may potentially suffer the same pitfalls as developers before them. As reliable software design is critical in risk mitigation [69], it is common to see university CubeSat groups seek out mature software frameworks for implementation into their projects. Open source software frameworks such as NASAs Core Flight Software aim to aid in cost and schedule savings for mission development, whilst improving the reusability and quality of a code base. The framework has proven flight heritage on missions such as the Lunar Reconnaissance Orbiter and the Lunar Atmosphere and Dust Environment Explorer, demonstrating its suitability in a space mission [101]. JPLs open source F Prime framework aims to achieve similar benefits for small satellites. Having been used on missions such as ASTERIA, Lunar Flashlight and NEA Scout, it has proven flight heritage in CubeSat applications [102]. If CubeSat developers are willing to select their on-board computer OBC for compatibility with a framework, KubOS provides a software stack that can aid in rapid development by supplying the API for interfacing with hardware [103]. It is also common for university CubeSat groups to purchase and use a commercial Software Development Kit (SDK) for their launches, attracted to the appeal of flight heritage with in-orbit validation and guaranteed long term support [104]. Whilst using a mature framework or an SDK can be useful for university groups to achieve the short term goal of a launch, it can become a detriment to a space program with the intention of flying multiple spacecraft. To fully benefit from the use of existing software solutions, the software written for the CubeSat needs to conform to the existing structure. This can couple the code base to the specific software solution being used, making it difficult to break from it in the future if it no longer meets mission requirements. In the instance of solutions with proprietary software, use on subsequent missions may carry with it extra licensing costs. Using an SDK for the development of EIRSAT-1, Doyle et al. mention that the knowledge base of the software development team regarding mission software design is encompassed within the confines of using the licensed SDK, further stressing the potential difficulty in breaking from the use of a kit. It was recommended that teams with the intention of flying multiple missions seek out full in-house development [104]. This recommendation is often heeded in the case where a University group wishes to produce reliable software that can be reused on future missions [105][106].

From an independent analysis however, Latachi et al. identified custom CubeSat flight software as a highly critical system in the success of a mission due to the lack of flight heritage. To decrease the risk of mission failure, four areas were identified to promote the development of a safe, generic software architecture for spacecraft: modularity, reliability, re-usability and extensibility [69]. These sentiments are seen to be reflected in other University groups that have written their own custom CubeSat software [49]. Modularity was mentioned to be a key driver for the development of ITASAT, where Carrara et al. mention that a major objective of their CubeSat project is to develop a multi-mission platform [107]. Coupled with modularity, extensibility is demonstrated in the software architecture for the Masat-1 and Kysat CubeSat missions, where a distinctive layered approach has been adopted. Each loosely-coupled layer consists of various modules, each encapsulating their own set of functionalities that do not interfere with one another [69][106]. Developing the code base in such a manner allows for hardware to be changed with only minor modifications to the software, as only the module encapsulating the driver for the hardware needs to be changed. Moreover, extra modules can be added to layers without breaking the software structure, permitting extensibility of the code base. Whitney et al. propose that a key element in the extensible software design of OpenOrbiter is the componentisation of modules, allowing for scalability [108]. Supplementary to extensibility, Hishmeh et al. discuss the importance of ensuring the modules developed have a simple interface. This increases readability of the code base, and therefore the likelihood of maintainability and reusability [106]. Both the Masat-1 and the Galassia CubeSat teams stress the importance of separating the functionality of the payload software from the flight logic. This was mentioned to enable reusability by allowing the majority of the code base to be reused between missions [69]. For Askari et al., implementing this level of modularity was key in achieving a program requirement of only needing to develop software for the new payloads being flown between missions [105]. De Souza et al. were able to achieve reusability through a layered software approach, where only the lower layers close to the hardware need to be changed for deployment on future hardware [109].

Whilst commonly recommending the development of in-house modular, reusable CubeSat software, it is often seen in published work that CubeSat software developers seldom offer advice to new developers on how to achieve this goal. The concepts of modularity, abstraction and layered software design may seem overwhelming to University teams embarking on the development of their first CubeSat, which in turn may increase the likelihood of the team electing to use a mature software framework or an SDK. As previously mentioned, whilst using an existing software solution decreases the time-to-orbit for an initial mission, the ongoing costs and tight coupling of the code base to the solution being used can make further development difficult. Furthermore, the University team may miss out on the potential learning experience of developing their own software framework.

Fortunately, independent bodies exist in an attempt to standardise the development of spacecraft systems, including software development. The European Cooperation for Space Standardisation (ECSS) is one such body. The design practices previously mentioned by the CubeSat software developers are in line with the topics discussed in publications for software engineering and software product assurance, ECSS-E-ST-40C [110] and ECSS-Q-ST-80C [111]. The software engineering and software product assurance documents are grounded in the creation of unambiguous, verifiable and traceable system requirements. The clauses therein encourage the developer to satisfy the produced requirements throughout the mission life cycle. Adherence to the standards promotes the creation of safe, reliable, maintainable code, increasing the likelihood of mission success. Whilst offering a high level overview of the necessary elements for reliable spacecraft software design, the documents however still do not offer in depth advice on how to implement modularity, reliability, re-usability and extensibility in space systems.

From the works discussed, software developed for CubeSats is often based around requirements, or in the case of Masat-1, influenced by FDIR routines [69]. Learning from incidents (LFI), in particular accident analysis, is a common technique used by the aviation industry to learn and prevent recurrence of mishaps [112]. However, it does not seem to be a common driver for CubeSat software development, perhaps as previously mentioned due to little published work on CubeSat failures existing. Yet, often described as a test-bed for larger spacecraft platforms, the development of CubeSat software can be influenced by the accidents that occur in larger spacecraft missions. Therefore, with the proven ability to aid in sustainable developments in system safety, the LFI concept was applied in the development of software for the Binar platform.

Binar-1 (named after the Noongar word for fireball) is Western Australia's first satellite, designed and built by the Space Science and Technology Centre at Curtin University in Perth. At its core, the 1U CubeSat is controlled by a motherboard that is designed entirely in-house, with a focus on subsystem design and integration. The completed core occupies just 0.25U, consisting of primary and redundant flight computers, a GPS, attitude determination and control system (ADCS), electrical power system (EPS) and memory storage, leaving significantly more room for payload than most similar platforms. Having this capability in-house enables prototype testing to destruction to stress the limits of the spacecraft, and facilitates hardware design reuse between missions. In addition to the hardware development, the author has developed the software stack powering Binar-1.

The custom software consists of flight application code, a hardware abstraction layer, operating system abstraction layer, utilities, system resource manager, and board support layer for low-level drivers for on-board peripherals. Writing custom software for the motherboard meant that on-board peripherals could be leveraged in the most optimal and efficient manner to function as expected. Being developed in parallel to the iterative hardware design process, software development saw adhesion to abstracted software design that enabled hardware to be changed without requiring much modification to the software. This ubiquitous abstraction has become a core feature of the Binar spacecraft that will allow software to be reused on future missions with minimal changes.

This paper analyses historical space mission failures due to inadequate software, using the findings to create recommendations for CubeSat software developers on writing reliable software for small spacecraft. As a case study, these recommendations are applied to the development of the Binar Software Framework, developed by the author, with the perceived benefits discussed. Unfortunately, a hardware fault on the adaptor board connecting the Binar CubeSat core to the transceiver on Binar-1 prevented communications between the motherboard and the transceiver. This meant that ground control was only able to communicate directly with the transceiver. Due to this, the on-orbit benefits of adopting the recommendations could not be validated. However, we have seen some benefits of adherence to the methods used in the paper as we are currently using the code base for development of our next three spacecraft. As the program progresses and the spacecraft produced become more complex, it is expected further benefits will become apparent.

Section 2.2 analyses the missions, finding that the areas of code reuse, software testing and poor code base maintenance are seen to be frequently responsible in mission failures. Following this, Section 2.3 then discusses how software can be written for small spacecraft to prevent these failure reasons from reoccurring, using Binar-1 as a case study. Where applicable, it mentions how the software aligns with the clauses specified in the ECSS documents. Finally, the benefits of adopting the proposed recommendations are discussed in Section 2.4 with a focus on safety, reliability and reusability, increasing the likelihood of small spacecraft mission success.

2.2. SPACECRAFT SOFTWARE FAILURES

Whilst the importance of software on spacecraft is well understood [113], failures are still common in the space industry specifically resulting from mishaps related to software, often due to insufficient testing prior to launch and poor coding practices [114] [100]. This trend is also observed in the small spacecraft community [41]. In a 2017 study, Venturini et al. [115] conducted 23 interviews of organisations with CubeSat industry experience. Respondents to the interviews identified flight software as a high-risk area, repeatedly stressing the importance of robust software design in the reduction of risk. Ideally, this paper would analyse small spacecraft mission failures due to software, however with a lack of definitive publications on this topic it will use examples from larger spacecraft missions instead. The scope of the missions selected for this paper was limited to only those that publicly disclose software to be a critical point of failure from the result of an accident investigation review. The spacecraft failures that met this criterion were the on-orbit collision involving DART in 2005, the potential atmospheric destruction of the Mars Climate Orbiter in 1998, and the self-destruction of Ariane-5 and Zenit-3SL shortly after lift-off in 1996 and 2000, respectively. Code reuse, software testing and code base maintenance were frequently mentioned in the investigation reports for these missions. The overlap of software failures in the missions selected can be seen in Table 2.1. It is hoped that this paper will serve to raise awareness of common spacecraft software failure issues, and showcase methods to prevent similar failures in the future. This section of this paper will examine the selected missions and discuss the reasoning behind their software failures. Following this, Section 2.3 will discuss how the Binar Software Framework addresses these common failure areas to produce safe, reusable spacecraft software to promote small satellite mission success.

TABLE 2.1: A summary of the cause of spacecraft software failures.

	Failure Reason			
Mission	Testing	Documentation	Code Review	Code Reuse
DART	*		*	
MCO	*	*		*
Ariane-5	*	*		*
Zenit-3SL	*		*	

2.2.1. DART

The Demonstration for Autonomous Rendezvous Technology (DART) project was a 2005 NASA mission to demonstrate the hardware and software required to perform an automated docking with another spacecraft in orbit [116]. The target spacecraft for docking, MUBLCOM (Multiple Paths, Beyond Line of Sight Communications), remained operational in orbit after having satisfied its primary goal. Eleven hours into the mission, DART collided with MUBLCOM and proceeded to its end of life sequence, failing to achieve any of the fourteen on-orbit mission objectives [117].

The Mishap Investigation Board (MIB) identified significant software faults that led to mission failure [118]. The first point of failure occurred as a result of an attempt at program error correcting logic. The spacecraft used a standard Kalman filter approach to estimate position and velocity vectors using GPS and inertial data. However, the spacecraft also introduced a software reset when the error between the estimated and measured data became too great. After the reset, the flight computer sampled the GPS and used those readings as its new estimate. This initial reading however was consistently 0.6 m/s inaccurate which would again cause the readings to drift and trigger a software reset, repeating every three minutes. The MIB discovered that the design requirements dictated an algorithm that should have been able to correct a discrepancy up to 2 m/s, well above the error range that forced the spacecraft into a software reset loop. The MIB proposed that the failure to detect the inadequate control algorithm stemmed from lack of testing across the full range as specified by the design requirements. Furthermore, in the calculation of its position and velocity vectors, DART weighted the estimated value far higher than it should have. This was caused by a late change of a gain constant in software that did not undergo sufficient testing. The new gain factor, attributed to changing mission requirements, meant that the algorithm could never actually converge. Ultimately, this caused excessive course correction manoeuvres during the short mission length, leading to the early mission retirement and contributing to the mishap with MUBLCOM. The failed docking and subsequent collision can also be attributed to an overly strict control sequence to initiate the docking manoeuvre. The waypoint that DART was to enter to begin the first on-orbit mission objective was too small, and as a result the spacecraft continued to drift toward MUBLCOM. The algorithm for the docking sequence did not account for the possibility of navigational errors, and hence there was no ability to recover after this waypoint was missed. An algorithm for collision avoidance was built into the spacecraft, however it depended on the same navigational data source that led to the early retirement fault and hence was ineffective at preventing the failure.

2.2.2. MARS CLIMATE ORBITER

The Mars Climate Orbiter (MCO) was a 1998 NASA mission intended to study the Martian climate from a low circular orbit. The intention was to use an initial elliptical orbit through the upper Martian atmosphere to lower the orbital apses to achieve a low circular orbit [100]. However, after beginning its orbital insertion burn, communication with the spacecraft was lost. The MIB determined that MCOs initial elliptical trajectory passed too deep into Mars's atmosphere, either destroying the spacecraft or causing it to skip out of the atmosphere and into a heliocentric orbit [119]. During the investigation, the failure was traced back to code reused from Mars Global Surveyor (MGS) (launched in 1996). The code in question assisted with the calculation of the small forces required to prevent saturation of the spacecraft's reaction wheels due to the build up of angular momentum. The MGS code required imperial units in its calculation of the impulse bit, which then was to be converted to metric units to be used by the propulsion control routines. However, the reaction control system thruster size on MCO was increased from MGS, requiring changes to the control algorithms. Whilst the equations used for MGS did take into consideration the conversion between pounds per second to Newtons per second, this value was buried in the code with a lack of sufficient documentation. Coupled with a failure to meet interface specification requirements through testing [120], when the new thruster equation was substituted into the reused software this conversion factor was missed and hence was mistakenly removed. As a result, the software interface specification for MCO received thruster parameters in imperial units rather than the required SI units, which ultimately led to the spacecraft dropping its periapsis to an estimated 57 km [120].

2.2.3. ARIANE 5

As a successor to Ariane 4, much of the already flight-proven software was reused to accelerate the development of Ariane 5. Unfortunately the first flight in 1996 resulted in a catastrophic failure 37 seconds after lift off when the self-destruct system triggered the destruction of the launch vehicle. The post incident investigation into the failure tracked the cause back to the inertial reference system failing after having encountered a software exception [121]. Incorrect data coming from the inertial reference system instructed the on-board computer to fully deflect the solid booster engines, causing the vehicle to veer off course. The flight computer could not swap to the redundant inertial reference system as it too had suffered the same failure. The software exception on both inertial reference systems was found to have been caused by an integer overflow in the attempt to convert a 64-bit float to a 16-bit signed integer. The conversion software did not handle the case where the float would be too large to fit into the integer. This error was located in the software that aligned the inertial reference system on the launch pad. After lift off, the function, which was designed for Ariane 4, continued for 40 seconds into flight. However, the designers had overlooked the need to update the flight logic to account for the higher acceleration from Ariane 5's more powerful Vulcain engines. The velocity that the vehicle reached early into flight resulted in a larger value than could be converted to the smaller integer. The Ariane 501 Inquiry Board mentioned that had the inertial reference system and complete flight control system been tested extensively, the software errors leading to the failure could have been detected [121].

2.2.4. Zenit-3SL

Zenit-3SL was a Ukrainian expendable launch vehicle primarily used to deliver communications satellites into geosynchronous orbits. It was a multinational collaboration through the Sea Launch project that used a mobile equatorial sea-based launch pad for easier access to equatorial orbits [122]. The third mission of the program in 2000 failed shortly after launch when the rocket self-destructed after a significant course deviation [123]. The failure analysis panel determined that the launch failure could be attributed to a software error prematurely disabling the second stage engine. The root cause was traced to software failing to close a pressurisation system valve, leading to an excess loss in produced thrust and early engine termination [122]. The error was traced to a late change in mission requirements resulting in a software engineer modifying control software. During this modification, the code concerned with valve operation was accidentally deleted [123]. Pre-launch checks of the software following this modification unfortunately failed to identify the missing command to close the valve [122].

2.2.5. Summary of Failure Reasons

The key areas leading to software failure shown from the missions discussed are insufficient software testing, inadequate documentation, lack of code review and code reuse. A summary of how these areas contributed to the failure of the discussed missions can be seen in Table 2.1, highlighting the overlap that is present. It is evident that the largest contribution can be attributed to lack of software testing, with inadequate documentation, insufficient code review, and unsafe code also causing failures. The remainder of this paper will discuss possible preventative measures for these failures in the development of reliable small spacecraft software using the development for Binar-1 as a case study.

2.3. RECOMMENDATIONS FOR SMALL SPACECRAFT

2.3.1. SAFE CODE REUSE

As the goal of the Binar platform is to facilitate rapid mission concept-to-orbit, hardware and software reusability is of high priority to reduce the development time of future spacecraft. Yet, as can be seen from the failures of Ariane 5 and MCO, code reuse is not without risk. The Binar Software Framework was designed to maximise code reusability whilst minimising risk by focusing more on platform portability rather than direct reusability. Software reusability refers to the development and maintenance of software modules that can be reused between compiled binaries. With a focus on meeting the needs of a broad range of software requirements, code developed with reusability in mind can be large, complex and difficult to maintain [124]. In the cross-platform context, source portability refers to adapting existing source code to a wide range of environments. With a reduction in generalisation, software developed this way can be easier to produce and maintain in future implementations whilst also benefiting from increased reliability [124].

Developing software in this manner can be achieved by following time-tested software principles, designed to streamline the process of writing good code. Two common collections of principles that are frequently used are GRASP and SOLID. GRASP (General Responsibility Assignment Software Patterns) is a collection of nine software principles, namely controller, creator, indirection, information expert, low coupling, high cohesion, polymorphism, protected variations, and pure fabrication [125]. The collection of principles aim to guide the developer into writing robust object

TABLE 2.2: The UML class diagram arrows used in this paper with their meanings.

Arrow	Meaning	Description
——→	Association	Child object exists within and is used by the parent
- - - -→	Dependency	Changes to the parent object will affect those that depend on it
——▷	Inheritance	Child object inherits (and can extend) the functionality of a parent
- - - -▷	Implementation	Child object implements the methods defined in the abstract parent
——◇	Aggregation	Objects are associated, but the child can exist without the parent object

oriented software through module responsibility assignment. Similarly, the SOLID principles of object-oriented programming, namely Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation and Dependency Inversion, aim to increase the maintainability and flexibility of a code base by resolving improper dependencies between software modules [126]. Whilst these collections of principles may differ in some aspects, the general ideology is that software written adhering to established principles should be of a higher standard than software written without. In this section, Unified Modelling Language (UML) is used to show applications of each of the SOLID principles to the development of the Binar Software Framework in the first use of SOLID for spacecraft software development. Whilst the GRASP principles do provide a beneficial reference on robust software design, the scope of this section is constrained to only the methodologies used during the development of the Binar Software Framework. For reference, the UML class diagram arrows used in this paper are defined in Table 2.2. The observed benefits are discussed in Section 2.4.

SINGLE RESPONSIBILITY PRINCIPLE

The Single Responsibility principle is concerned with the division of program logic amongst multiple objects encapsulating a single small responsibility each [127]. The concept promotes using multiple objects with well-defined interfaces over using a singular object to control multiple aspects of an application. As a general rule, an object conforming to this specification is said to have a single responsibility if the object only

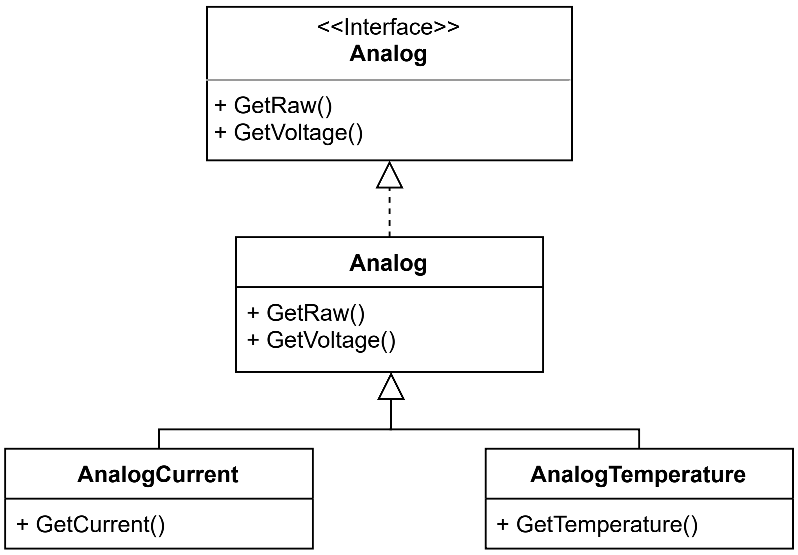


FIGURE 2.1: The application of the Single Responsibility Principle to the ADC module in the Binar Software Framework. The AnalogCurrent and AnalogTemperature modules are decoupled from each other by implementing their required functionality within the scope of their respective objects. A change to either of the AnalogCurrent or AnalogTemperature objects will not result in a change in the other.

has a singular reason to change. An example of this in the Binar Software Framework can be seen in the analog to digital conversion software module (Figure 2.1) where the current and temperature measuring objects are implemented as independent classes instead of being amalgamated into one complex monolithic object responsible with measuring both.

In our approach, a base class accounting all features required by the analog objects was implemented from an interface, and then separate specific analog classes inherited and extended this functionality. This prevented the non-cohesive current and temperature objects from being coupled together causing unnecessary rigidity in the module design. This is important in the context of code reuse as the prevention of coupling means that changes to subtypes of the Analog module will not invoke a change in other modules.

OPEN-CLOSED PRINCIPLE

The Open-Closed principle [127] refers to the development of software modules that are open for extension, but closed for modification. The functionality of modules under

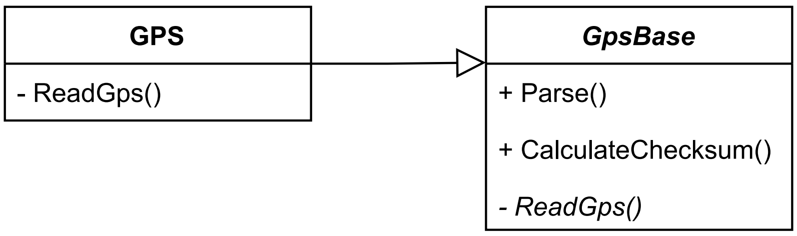


FIGURE 2.2: The application of the Open-Closed Principle to the GPS module in the Binar Software Framework. The functionality of the GpsBase class is extended through the addition of new code in the inheriting GPS object, without the need for modification of the original base class.

this principle can be extended upon when the requirements of an application change, but the functionality that has already been implemented cannot be modified. The principle recommends that changing project requirements should be addressed by adding new code rather than by rewriting code that already works. This is commonly implemented through the use of abstract objects in object-oriented programming design [127]. An example of the Open-Closed principle in the Binar Software Framework can be seen in the GPS library (Figure 2.2), where the design of the GPS module includes a base class with a set of concrete public methods which describe functionality in terms of the abstract method *ReadGps*. The abstract object *GpsBase* gets implemented in the inheriting module, GPS, thus extending the functionality of the module without modifying the source code.

The Open-Closed principle is helpful in this context because it explicitly highlights that not all code can be safely reused, favouring the development of modules that separate generic functionality from the detailed implementation. As GPS devices output data through a series of strings with a checksum, this functionality can be developed independent of the GPS, decoupling the module from the specific hardware chosen.

LISKOV SUBSTITUTION PRINCIPLE

The Liskov Substitution principle proposes that subtypes of modules must be substitutable for their base types in application code [128]. Analogous with run-time polymorphism, it is one of the enabling factors that makes the Open-Closed principle

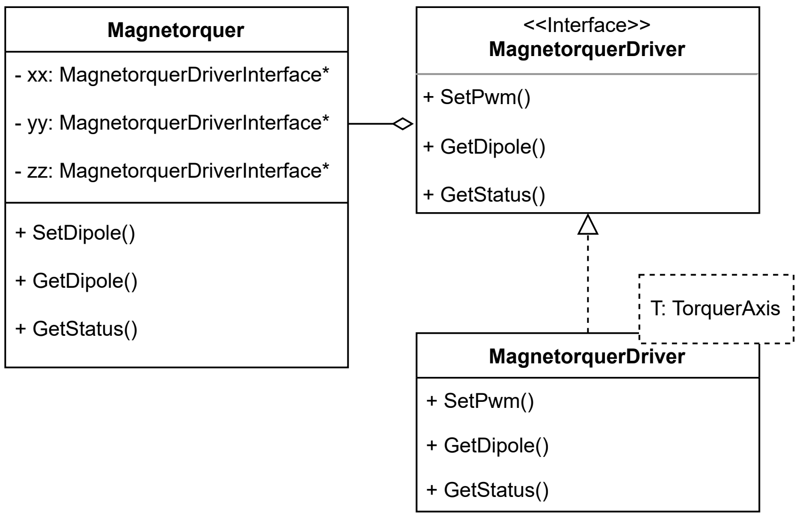


FIGURE 2.3: The application of the Liskov Substitution Principle to the magnetorquer module in the Binar Software Framework. The Magnetorquer object has three private pointers to axes that implement the MagnetorquerDriver interface object. Any of the template concretions of the MagnetorquerDriver class are eligible for substitution into these private pointers as they inherit from the MagnetorquerDriver interface object

discussed in Section 2.3.1 possible. The principle relates to code reusability through increasing portability by decoupling generic module software from the detailed implementation. Furthermore, this aids in future maintainability by providing a clear contract with the application code on how to use a specific module, rather than being concerned with how it is actually implemented. An application of the Liskov Substitution principle in the Binar Software Framework can be seen in the magnetorquers software module (Figure 2.3), where a templated MagnetorquerDriver object can be used by the Magnetorquer class as it inherits from the MagnetorquerDriver interface object.

In this module, the publicly visible magnetorquer object controls the MagnetorquerDriver objects through references to their abstract base class. The concrete MagnetorquerDriver subtype is templated on all available magnetorquer axes, making the addition of redundant axes compatible with the public module without modification. This is important as it reduces the coupling between objects in a module. Implementing functionality through pointers to abstract interfaces allows the higher level software to use the lower level objects without having to be concerned how they operate, thus permitting changes to interactions with spacecraft hardware without forcing a recompile of the full module.

INTERFACE SEGREGATION PRINCIPLE

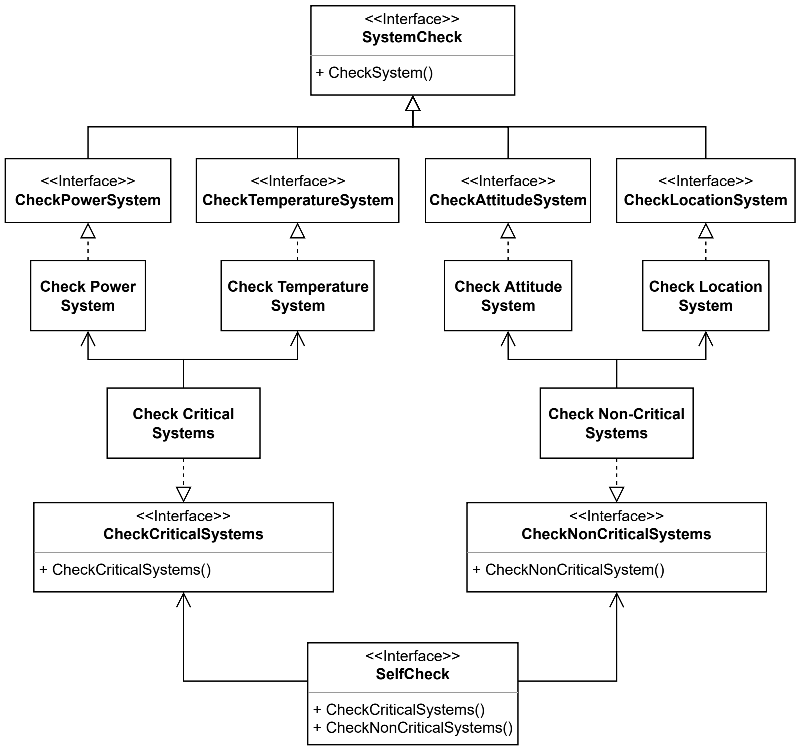


FIGURE 2.4: An application of the Interface Segregation Principle to the self check module in the Binar Software Framework. In this module, each subsystem that can be checked on-board the spacecraft implements a unique, decoupled interface. The critical and non-critical system check implementation objects then associate the implementations of the unique interfaces. This prevents a check done with the critical system object from becoming coupled to a non-critical system, allowing the systems being checked to be extended upon without forcing a recompile of unrelated modules.

The Interface Segregation principle states that software module subtypes should not be forced to depend on modules they do not use [127]. To achieve this, it recommends avoiding monolithic interfaces that contain many unrelated methods. Rather, the principle favours multiple smaller interfaces that get implemented where needed. This principle is closely related with the maintainability aspect of code reuse by removing code duplication as much as possible, whilst avoiding the pitfalls of non-cohesive software interfaces. An example of this principle implemented in the Binar Software Framework can be seen in Figure 2.4, where subsystems that are checked in the self test module are isolated from one another, and are implemented where required.

The alternative, more rigid approach to implementing the self check module would have been to have a singular self check interface that the subtypes of SystemCheck interface all depended on. Whilst simple to implement, this pattern would result in methods being added in the standard self check interface that would have only been used exclusively by specific subtypes. This would then couple all the clients of this interface together, and open up the possibility for a module to be affected by changes that other clients force upon the class. The resulting pollution of the interface could potentially lead to errors when code is reused. Due to the likelihood of on-board systems expanding as mission requirements become more complex, the Interface Segregation principle allows for these potential changes by decoupling objects from non-cohesive interfaces and therefore allows subtypes to not be forced to depend on methods they don't need to. Moreover, implementing this principle aids in high testability of software modules due to the granularity achieved in the responsibilities of the interfaces. In the example seen in Figure 2.4, rather than having the system classes that inherit from the abstract interface depend directly on one monolithic SelfCheck interface object, each subclass of the abstract class is given its own interface to depend on. Self check objects can then implement the individual interfaces that they will actually make use of for checking, rather than all systems on board. For testing the self check module on the host, the interfaces responsible for interacting with the hardware are able to be mocked and substituted into the module under test through polymorphic dependency injection.

DEPENDENCY INVERSION PRINCIPLE

Rather than the traditional application architecture approach of a direct dependency chain between high-level software and low-level implementations, the Dependency Inversion principle states that high-level software and low-level software are to both depend on abstractions [127]. An example of this can be seen in Figure 2.5, where the high-level public facing IMU object and the lower level private ImuImpl objects both depend on the abstracted IMU interface.

If the application code is interested in interacting with the IMU, it does so through a

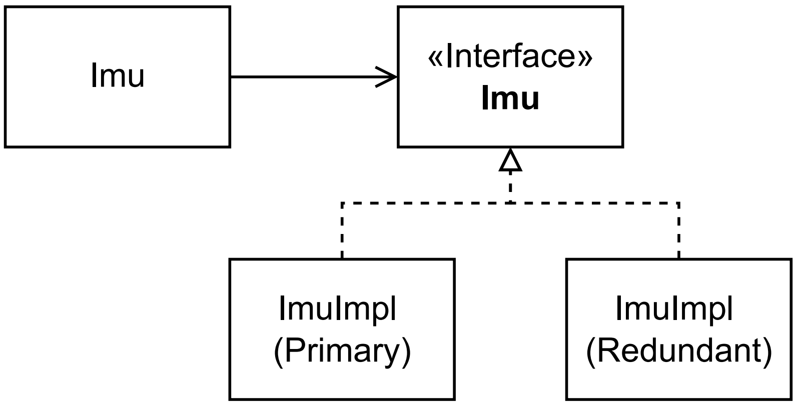


FIGURE 2.5: An example of the Dependency Inversion principle in the IMU software module. Application code wanting to access the IMU hardware will use the high-level object that depends on the same abstraction as the lower level implementations. This decouples the high-level IMU object from the hardware it is interacting with, allowing the IMU object to be reused even if hardware requirements change.

high-level object that implements the same IMU interface used by the implementation objects, rather than depending on the implementations directly. The software using the peripheral has then become abstracted from specifically how it receives the IMU data and instead focuses on relevant details that are independent of the platform. This principle is important for code reusability as it creates a modular interface where low-level implementations can be changed without affecting high-level code that interacts with them. As Binar-1 has a primary and a redundant flight computer with different architectures on board, they will need different software binaries to operate. Moreover, considering the redundant flight computer is to take over from the primary flight computer in the event of an emergency, it will need to be able to operate with the same peripherals. It would be time-consuming and error-prone to rewrite the software concerned with peripheral interaction simply because it is to be run on a different microcontroller. The application of the Dependency Inversion principle means that the high-level flight software can be reused between the computers, with the implementation simply swapped out.

The Binar Software Framework further leverages the Dependency Inversion Principle by making use of compile-time configurable implementation linking. Using custom command-line arguments, the compiler will link generalised high-level objects with their computer specific implementation to build binaries for the two flight computers on-board the Binar bus. As the hardware for Binar develops in the future, the application of this principle means that new drivers can be implemented without any risk of breaking high-level flight logic.

FAILURE PREVENTION

The usage of SOLID principles can help reduce the likelihood of errors appearing through the reuse of code. In the case of the Mars Climate Orbiter discussed in Section 2.2.2, when the small forces software from the Mars Global Surveyor mission was reused, it was tightly coupled to the thruster control equation being used. This was demonstrated by the fact that the software had to be modified, and also that the modification of the software broke its functionality. If it were possible to separate and encapsulate the section of the software concerned with calculating the thruster parameters, leaving the genuinely transferable code agnostic of the thruster being used to interact with a well-defined interface, then the design team could have possibly written a plugin for the new thruster to conform to the interface specification. Inserted into the small forces software in a modular approach, this could ensure successful implementation portability by removing the ambiguity between units by providing a documented interface specifying the inputs and outputs of the object.

Similarly, with the Ariane 5 rocket discussed in Section 2.2.3 care was not taken when reusing code from Ariane 4 to ensure that a narrowing conversion from a large data type would fit within the bounds of a smaller one used in application code. If the module concerned with using the inertial reference data was decoupled from reading it, instead interacting with an interface, then the remaining portable high-level code could have been transferred without the application-specific data conversion. Engineers could then write a new module for the low-level software to interact with the different hardware whilst conforming to the interface for it and have the high-level software interact with it seamlessly.

Being able to reuse code safely involves designing software to be understandable, flexible and maintainable. Adhering to the SOLID principles of object-oriented program design, the modules written for the Binar Software Framework ensure that a developer can safely reuse code on future missions. Modules having only a single responsibility mean they are decoupled from each other and won't be effected by changes to other modules. Changes are expected, mostly due to changing hardware on the spacecraft, and so high-level libraries controlling low-level drivers polymorphically allow for new drivers to be written that use the same interface. If the hardware requires a more feature-rich driver, the interface can be extended whilst still satisfying the polymorphic requirements of the high-level controlling object. When these drivers are added to support new hardware, new targets can be added to build files in the code base to selectively link the software framework against revisions at compile time. Moreover, drivers that do not rely on hardware can also be linked at compile time for testing the software independent of a spacecraft engineering model.

The usage of the SOLID programming principles contributes greatly in adherence to the clauses outlined in ECSS-Q-ST-80C. The standard mentions that the development of software should apply design methods that have performed successfully in a similar application to ensure dependability and safety [111]. The large volume of published work regarding SOLID proves the successfulness of the time-tested guidelines, and has even been observed in the scientific community to increase maintainability of a code base, whilst decreasing dependencies and complexities [129]. The ECSS-Q-ST-80C document also specifies that the development of software with the intention of future reuse is to be implemented with a minimum dependency on hardware. The modularity achieved from adherence to the SOLID principles aids in meeting this standard, promoting safe reuse on future missions.

2.3.2. SOFTWARE TESTING

DECOUPLING AS A TOOL FOR TESTING

A commonly reported reason for spacecraft failures in the missions discussed in Section 2.2 was the failure to adequately test modules of the software, to the extent that the Ariane-5 investigation board mentioned explicitly that testing could have identified fatal errors during development [121]. From a CubeSat perspective, the survey

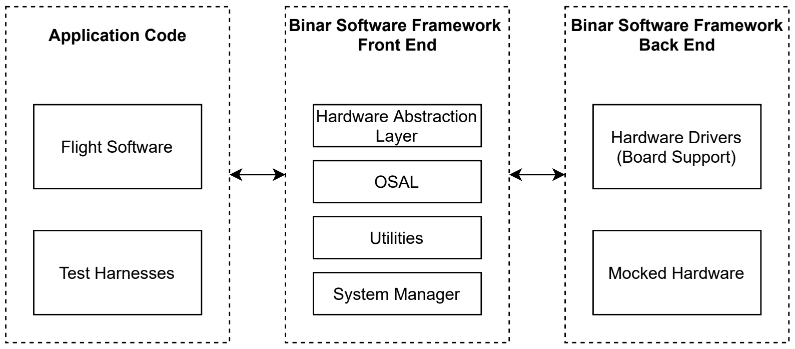

Application Code
Flight Software
Test Harnesses
Binar Software Framework Front End
Hardware Abstraction Layer
OSAL
Utilities
System Manager
Binar Software Framework Back End
Hardware Drivers (Board Support)
Mocked Hardware

FIGURE 2.6: Testability of the Binar Software Framework is increased through abstracting away low-level hardware dependencies.

conducted by Venturini et al. [115] saw an emphasis on the importance of software testing in the success of a mission. The failure to test is a common challenge faced in embedded software development due to hardware dependency. In these situations, the lack of ability to test is due to the tight coupling between high-level software and the low-level software that is concerned with interactions between embedded subsystems. Furthermore, barriers to testing that arise are related to commonly only having a single spacecraft engineering model to test on and the commonality for hardware revision. This scarcity of testing hardware can introduce issues during the testing process where software may cease to work on future hardware revisions [130].

Decoupling flight application software from the embedded hardware on the spacecraft allows testing on various satellite models, both physical and virtual. Tipaldi et al. [131] discuss the benefits of using black box testing verification in a high-level test procedure language to verify the flight software over the lifetime of Meteosat Third Generation. Taking advantage of the module decoupling that is achieved through the implementation of the SOLID principles discussed in Section 2.3.1 however, the white box testability of the Binar code base can be increased. In the Binar Software Framework, testing of libraries that interact with low-level hardware through an interface layer is done by polymorphically substituting mocked objects in place of the modules that require a hardware dependency, leveraging the Liskov Substitution Principle. Favouring many small interfaces over one monolithic interface, as recommended by the Interface Segregation Principle, adds granularity to the code base and hence only those objects concerned with hardware interaction need to be mocked when testing on the host. Tests can then be written based on how the libraries are to be used in application code, enabling testing of the software framework without a hardware dependency as can be seen in Figure 2.6. Having fine granularity in system interfaces also allows for fault injection into the module under test through faked implementations of interfaces, adhering to the software testing standards outlined in ECSS-E-ST-40C [110]. This allowed the software to be written in parallel with the hardware development of Binar-1, prior to performing integration testing. Furthermore, this module decoupling opens the possibility for extending upon the functionality of the software for future missions without breaking old tests. Moreover, polymorphically substituting mocked objects in place of hardware dependencies enabled libraries developed for the Binar Software Framework to adhere to test-driven development to meet the desired test coverage.

TEST-DRIVEN DEVELOPMENT

Test-driven development refers to a method of developing software in which requirements of software are converted to test cases as a precursor to the actual development of the software itself. The process involves firstly writing a test that won't succeed, followed by writing just enough code to make the test pass, and then lastly refactoring to remove duplication in the software written to pass the tests [132]. Adhesion to this method of development aids in meeting the software testing standards outlined in ECSS-E-ST-40C, which highlights that traceability between requirements and all software units is essential in order to produce meaningful test results [110]. Aside from having 100% test coverage, software developed through this method will be as lean and simple as possible. According to ECSS-Q-ST-80C, software can be assured to be dependable and safe through 100% code coverage at the unit testing level. Moreover, it is mentioned that this metric is critical during the feasibility review of reusing the software in the future [111]. The creation of the Binar Software Framework saw strict adherence to this design method during development. The benefits of using the TDD approach in the development of the software framework include the ease of performing positive and negative testing. These types of testing, enabled through the conversion of software requirements directly into test cases, ensure that the libraries developed will safely handle both correct and erroneous data to guarantee conformity of on-board software to system requirements. Positive testing ensures that the software is able to meet the requirements specified, and negative testing ensures stability through proving the software will behave as expected under the worst case scenarios, as recommended in ECSS-E-ST-40C [110]. High testability of libraries was assisted through propagation of errors from the low-level software to higher level code, providing transparency of the inner workings of a software module. This is important as the calling code generally has more information on how the error should be handled than what may be available in the scope of the library. This increases overall code portability and facilitates library reuse for future spacecraft.

An example of this in the Binar Software Framework can be seen from the development of the communications library. The operational requirements were determined prior to writing the library, outlining how the spacecraft should handle telecommands. Converting these operational requirements into test cases, negative testing was heavily employed to ensure safe execution pathways when receiving telecommands that either can't be decoded or were received in the wrong operational mode. Likewise, positive testing asserted that responses to correct telecommands were as expected. The test cases that enabled the development of libraries in the Binar Software Framework are executed frequently, ensuring that system requirements are always met.

FAILURE PREVENTION

One of the failure points for the DART mission discussed in Section 2.2.1 was an inadequate control algorithm that failed to correct for a velocity error of up to 2 m/s, even though this was specified as a design requirement. Considering the TDD methodology starts with the translation of software requirements into test cases, mission-critical requirements are factored into the software early, and any further development that could break this requirement would be highlighted when the test for the requirement begins to fail. Learning from this mishap, all requirements for the libraries written for the Binar Software Framework were defined and translated to test cases before the libraries themselves were written to prevent software requirements being missed.

Another area of failure for the DART mission occurred when an engineer changed a gain constant that resulted in an iterated calculation never converging. This error demonstrates the importance of continuous testing not just during library development, but consistently during the implementation of the library to ensure changes made to the software do not go out of sync with the tests written for it. From the adoption of TDD for the software framework enabling Binar-1, a change such as this would cause test cases to begin to fail, highlighting that the library is not safe for release.

It was evident through writing test cases for the Binar Software Framework that it became easy to test how the libraries would handle erroneous inputs, rather than relying on the assumption that libraries would always perform as expected. This was critical as the MIB mentioned during the DART mishap investigation that during pre-flight testing a software model assumed the GPS measured velocity perfectly [118]. Having the ability to stress-test all libraries used on Binar-1 provides further reassurance in the reliability of the Binar Software Framework.

Software testing also aids in the adherence to ECSS-E-ST-40C regarding the reuse of code. It is mentioned that for code to be reused, it needs to adhere to the functional requirement baseline of the project [110]. As TDD involves the conversion of software requirements to test cases, the reused software will need to verifiably adhere to the functional requirements to be included in the code base.

During the mishap investigation for Ariane 5 discussed in Section 2.2.3, the report mentioned that had the software for the inertial reference system and flight control system been tested extensively, the software errors leading to the failure could have been detected. By developing the libraries through the TDD approach, 100% test coverage is possible, potentially allowing for the identification of this error in the development stage of the mission.

2.3.3. OPTIMAL CODE BASE MAINTENANCE

DOCUMENTATION

In a 2012 review of the involvement of software in spacecraft accidents, Leveson [100] acknowledges that inadequate documentation is a factor in spacecraft mission failures. In the missions discussed, this is particularly relevant to the MCO and Ariane 5 mission failures which recommend improvements to code base documentation in their failure analysis reports. However, from a survey of 48 candidates with experience in the software engineering industry, Forward et al. [133] found that whilst participants thought positively of the usefulness of software documentation, it was seen to be seldom updated. To prevent issues arising as a result of poor software documentation, whilst acknowledging the difficulties of documentation maintenance, documentation generation is employed on the Binar platform to maintain up-to-date documentation. Having access to generated documentation pages can be very useful for future developers and maintainers to understand how to maintain and use the numerous libraries that comprise the Binar Software Framework. To fully utilise this tool, Binar developers are encouraged to use increased verbosity in variable and function names. Non-cryptic names are desired over their shorthand counterparts to reduce ambiguity and promote self-documenting code. To ensure that code generation is invoked often, it has been integrated into the Binar software development pipeline to automatically build when a new commit is pushed.

FROM LOCAL MACHINE TO SPACECRAFT

The code base for the Binar Software Framework is hosted on a remote version control repository that is always guaranteed to have the latest stable release of the software. This is done through the implementation of Continuous Integration and Continuous Deployment (CI/CD) online. The author introduced a development process where each developer in the Binar Space Program maintains their own private fork of the upstream repository, allowing for isolated development environments. When a developer pushes software changes to their fork and creates a merge request into the upstream repository, it executes the CI/CD platform remotely. It then notifies an assigned team member to start the code review process. The CI/CD platform automates the building and testing of all software libraries written in the repository. If any library fails to build or complete its tests, the merge request will be automatically blocked from merging into the upstream branch until these issues are resolved. This prevents potentially buggy or broken software from merging into the upstream repository, which is used for code deployment onto the spacecraft. The results of the test execution include the code coverage report, which is generated into an access-controlled web page providing a line-by-line breakdown of the coverage within the source code. This provides a quantifiable metric of test coverage, and assists the developer on adding tests to increase the coverage.

During pipeline execution, a team member is assigned to the merge request to conduct a code review by visually inspecting and assessing all changes in accordance with the design rules imposed by the team, and to identify any accidental incorrect program logic. Using an online version control platform for this, discussion threads can be added to specific sections of the software changes to clarify decisions made during development, to recommend further changes be made, and also provide a history of important software changes. Complementary to this, the CI/CD platform also runs a static analysis on all libraries in the code base, generating an access-controlled web page containing the analysis report to be referenced during code review.

Upon completion of the review process, the team member assigned to review the merge request is given the option to merge the changes into the upstream repository only if the CI/CD pipeline succeeds. This prevents any software that is written by a singular engineer from getting deployed to the spacecraft without explicit approval from at least one other team member. Once successfully merged, the last stage of the pipeline is executed to generate a third access-controlled web page containing the documentation scraped from the source code. A summary of the process from code development to deployment on the spacecraft can be seen in Figure 2.7.

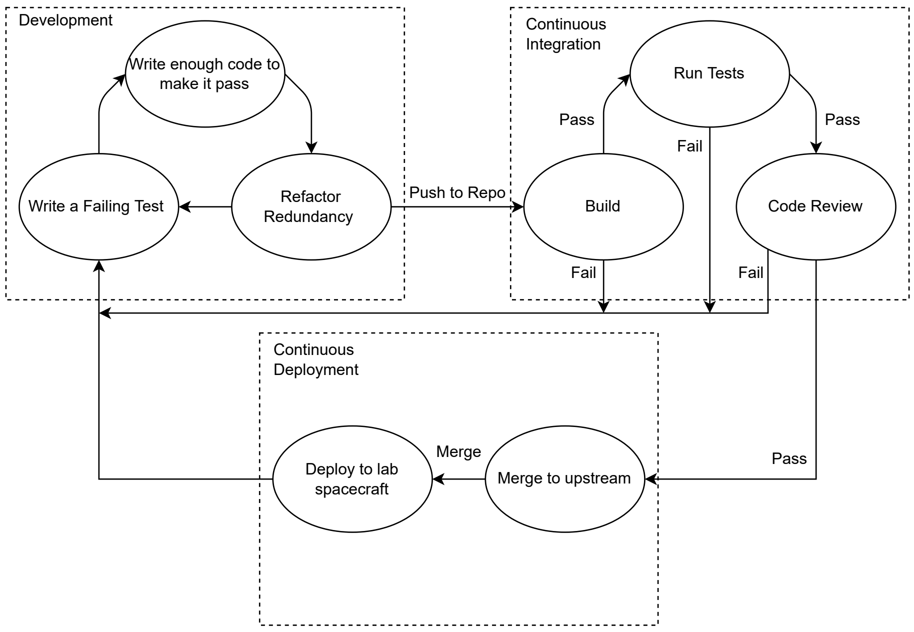


FIGURE 2.7: The Binar Software Framework development pipeline.

FAILURE PREVENTION

The importance of documentation being stressed early in the development of the Binar Software Framework was partly influenced by the mishap discussed in Section 2.2.2, where the Mars Climate Orbiter was lost during its orbital insertion manoeuvre. The mishap investigation report mentions that the equation used to calculate the small forces lacked sufficient documentation of the built-in conversion factor, hence being accidentally deleted by the developers reusing the software from MGS. Thorough documentation practices could potentially have identified the equation used in the software to have included the conversion factor, notifying the design team to adjust it accordingly. Also contributing to the mishap leading to the loss of Ariane 5, software documentation could be a solution to the investigation board's recommendation to verify the range of values taken by a variable in software. This has been achieved in the Binar Software Framework through extensive documentation of variables which are scrutinised during the code review process. The necessity of software documentation is stressed in ECSS-E-ST-40C, which insists that software documentation is to be constantly updated throughout the mission lifecycle [110]. Having software documentation automatically generated with every commit aids in meeting this standard, and ensures that errors related to insufficient documentation will not propagate into the spacecraft applications.

The push for implementing a pipeline getting software written on a local machine to the spacecraft stemmed from the mishap seen on the DART and Zenit-3SL missions discussed in Section 2.2, where a late change to control software led to the failure of the spacecraft. The ECSS-Q-ST-80C document mentions that software that is modified is to undergo regression testing [111]. If this standard was adhered to, it is expected that a change to the gain constant on these missions would trigger old tests to fail. Incorporating a CI/CD platform that runs tests online prior to allowing an update to be merged could potentially have identified these changes as faulty software, rejecting the commit and hence preventing the change propagating to release code. In the situation where tests seem to be passing on a surface level but in actuality lack sufficient depth, the generation of a code coverage report would notify the reviewer that the module is insufficient for inclusion into the upstream repository, and to hence reject the merge.

One of the recommendations by the investigation board following the failure of Ariane-5 mentions the importance of reviewing all software [121]. This was a strong influence for the code review and static analysis stage of the development pipeline imposed by the Binar software team, in which merges are rejected by a reviewer if the documentation is deemed insufficient, or software bugs are identified. The ECSS-Q-ST-80C document recommends that the evaluation of code is to be performed in parallel with the development process, aiding in the conformity to good coding standards and the reduction of software errors [111]. The code review stage of the pipeline aids in meeting this standard, preventing ambiguous and potentially buggy software from reaching release. Moreover, it is recommended that software is to be placed under configuration control after passing all unit tests successfully. Requiring the software being merged into the main repository to pass all unit tests aligns the developed software with this standard, providing assurance that the software propagating onto the spacecraft is dependable and safe.

Adopting a strict approach to code base maintenance can ensure code clarity, maintainability, portability and safety. A strong documentation standard can ensure a lack of ambiguity on how to implement software modules in application code, especially when internal software is reused, increasing the possible longevity of the framework for future Binar spacecraft. These attitudes toward software design can prevent mishaps associated with individual engineers altering software from occurring.

2.4. BENEFITS ON BINAR-1

Implementing the recommendations for small spacecraft software design on the Binar Software Framework has aided in the development of safe, reliable and reusable software.

Leveraging the SOLID principles of object-oriented program design, the software developed for the BSF is strongly positioned to be portable between hardware revisions. The example given for the application of the Single Responsibility principle on the Binar Software Framework was in regard to the ADC module. For future development on Binar spacecraft in the context of this module, this principle supports the intention of using a higher precision ADC reading on certain channels, requiring external ADC hardware. Interfacing with the external hardware will require a dependency on a library that will only be used for the higher precision subset of ADC measurements. Having a separate module concerned with the subset of ADC measurements prevents linking against unnecessary libraries that won't be needed for other ADC measurements.

The effectiveness of the Open-Closed principle for code reuse has already been demonstrated in the development of the Binar-2, 3 and 4 satellites. Due to the manufacturer deprecating the GPS used on Binar-1, the hardware needed to be swapped out in favour of their new module with a different pinout. If the GPS software module had been developed with a tight coupling to the hardware, any dependencies of this module would suffer from a cascading need to change the GPS object to a new implementation. Aside from slowing down development time and making code reuse difficult, this could also introduce unforeseen errors into the code base. The implementation of the Open-Closed principle supports code reuse by simply extending the already working GPS base object by implementing a new subtype for it.

The Binar-1 motherboard had circuitry for redundant magnetorquers on the x and y axes. However, the corresponding coils were not installed on the satellite. For the development of Binar-2, 3 and 4, these redundant drivers are intended to be used. From the adoption of the Liskov Substitution principle, support for these drivers can be added to the software module by adding extra axes to the templated magnetorquer driver object discussed in Section 2.3.1. The Binar Software Framework runs system checks on all peripherals during the spacecraft boot sequence. If the check determines a magnetorquer driver is faulty, the constructor for the publicly visible magnetorquer object can automatically switch to the redundant templated driver on-orbit, passing in a pointer to the driver polymorphically. This means the high-level magnetorquer module in application code can actuate agnostic of the actual magnetorquers being used as this is not necessary information to couple to the driver. This enables code reusability as it allows for any number of redundant coils to be controlled without modification to the high-level magnetorquer module, which can therefore be fearlessly depended on in application code.

As the primary purpose of Binar-1 was to be a technical demonstration of the hardware and software produced by the Binar Space Program, the software developed for system checking was constrained to systems embedded on the Binar bus. For future Binar missions, as the payloads flown become more complex, extending the self check module to support payload integrity checking is of high priority. From the adoption of the Interface Segregation principle, a corresponding base class for checking payloads simply needs to be added similarly to the systems shown in Figure 2.4. This payload system can then be implemented in a new systems interface class that will also implement checking the power system. This demonstrates the creation of cohesive groups of system checking objects within the module, showing how the library can be extended with future systems that don't depend on modules they won't use.

Through the application of the Dependency Inversion principle (Section 2.3.1), it was seen that much of the software developed for the Binar Software Framework has gone towards high-level libraries to be depended on in application code that are decoupled from the low-level hardware implementations. This currently allows application code to be cross-compiled between the differing architectures of the flight computers on board. As the Binar Space Program develops, a need has been highlighted for flight application code to be run natively on a desktop computer. This could allow for development of payload software without needing a Binar bus, and enable highly accurate simulations of on-orbit operations.

From strict adherence to the test-driven development methodology, the software running on board Binar-1 can be guaranteed to meet mission requirements. Moreover, due to the simplicity of setting up test cases, the libraries can be stress tested for the worst case scenario to increase reliability. As an extension to the portability achieved through the application of the SOLID principles, a large part of the software developed is decoupled from hardware and is hence easily testable. This aligns with the section regarding code reuse in ECSS-Q-ST-80C, which specifies that code to be reused should be assessed against requirements traceability, testing, and coverage [111]. The hardware decoupling achieved also allowed for software to be developed in parallel to the hardware revisions. Whilst this aided in preventing some bugs from occurring, it should be noted that testing on the hardware is also encouraged when possible. Testing on the host enables the behaviour of the modules to be verified, however issues may arise when running a hardware-in-the-loop test due to the disparities that exist between the platforms the software is running on. The biggest issues encountered during hardware integration for Binar-1 were related to data structure alignment, power domain access from microcontroller internal peripherals, and timing issues. These issues were not identified during unit testing as they were specifically hardware related. Coupling unit testing on the host with continuous deployment onto hardware can help with bug prevention internal to software modules, and identify any runtime issues that may occur.

Adopting good code base maintenance skills from the beginning of the Binar Space Program has led to a code base which is well documented, readable and maintainable. The benefits of this have been seen from the Binar Space Program team growing since its conception. As new team members begin to contribute to the code base, the plentiful documentation provides clarity on how to implement the many custom libraries developed for the framework. This decreases the time required to train new team members, and gets them contributing to the code base quicker. Clarity of software already in the code base has allowed for easy maintenance as hardware revisions are released and capabilities grow.

Enforcing a local machine to spacecraft cycle consisting of continuous integration and continuous deployment using cloud-based test running and code review means that no unexpected software will make it to space. Enforcing the use of testing over the full life cycle of the mission means that late changes made to the spacecraft libraries can be guaranteed not to break functionality.

2.5. CONCLUSION

With the complexity of software increasing in spacecraft missions, reliable software development is critical in the prevention of mission failure. The main contribution of this paper is demonstrating techniques for writing safe, reliable and maintainable software for small satellites through showcasing methods to prevent recurrence of common spacecraft software faults. As such, it is intended for small spacecraft developers planning on writing their own software who may not have prior experience. Where applicable, adherence to ECSS standards has been demonstrated. The missions showcasing common software failures were selected due to their transparency in disclosing software faults as reasons for mission failure. Being directed at small spacecraft software developers, this paper would have benefited from a discussion of CubeSat software faults. However, due to the lack of publications in this area, this could not be included. As software is shown in this paper to be a contributing factor in spacecraft mission mishaps, it is the hope that publications are more prevalent in the future relating to this topic. Although the benefits from recommendations requiring on-orbit validation were not able to be demonstrated in this paper due to a hardware fault on Binar-1, The Binar Space Program has already noticed benefits specifically regarding code base reuse as we progress into building our next satellites. As the program matures and we continue to develop the framework using the recommendations discussed in this paper, we expect further benefits to become more apparent.

Learning from the overlapping mishaps of previous mission failures, the Binar Software Framework has been developed to maximise mission success. The areas of code reuse, software testing and code base maintenance were addressed with examples of each from the core functionality of the Binar Software Framework. Safe code reuse was highlighted to be more complex than simple reimplementation, but was rather discussed in the context of developing software from the ground up with portability in mind. This was achieved in the Binar Software Framework through adherence to the SOLID design principles. Through these principles, the program goal of rapid mission concept-to-orbit can be achieved by reusing the platform independent hardware abstraction layer of the Binar Software Framework between missions. Although the SOLID principles were adhered to during development of the Binar Software Framework, other similar principle collections exist, such as GRASP. Regardless of the principles chosen for development, software written in accordance with time-tested principles will be of a higher standard than software written without.

Lack of software testing was seen to be the most common reason for failure in the missions discussed. It was demonstrated that developing software with decoupled interfaces aids with mocking modules to be substituted polymorphically for hardware implementations to break hardware dependencies and open up the possibility for test-driven development. To alleviate failures discussed in late changes to control software causing mission failures, the importance of tests written during the TDD process are stressed not only for use during development, but also during the full life cycle of the mission to highlight when software begins to break. Through the use of a CI/CD platform, when a developer made changes to software, all unit tests would be run remotely before software would be given the option for a merge request. Leveraging the pipeline to generate code coverage and static analysis reports assists the developer assigned to review the merge request, ensuring that software authorised to merge into the master branch has accountability. This was partly for the prevention of poorly designed code being added to the code base.

Code review also enforced strict adherence to writing documentation. This documentation, both through meaningful, clear and consistent variable names along with commenting, was done to ensure that future developers would be able to easily read, understand and maintain the code base. After giving approval to merge software upstream, the only software that could be deployed onto Binar-1 was pulled from the upstream master repository. This was to ensure that the software to be deployed onto the spacecraft was free of any errors, and gave a clear version history of software that was released.

Binar-1 was chosen as the case study for this paper as the software framework was being developed for the Binar Space Program. However, the concepts discussed in this paper hold broader implications for larger spacecraft, with the potential to improve the likelihood of mission success and promote code reusability for future missions.

2.6. FUTURE WORK

The mishaps discussed in this paper are a small subset of a large body of reference. It was observed from the majority of published failures that a common theme was lack of understanding how the mission would respond under the worst case scenarios. From the missions discussed, the Binar Software Framework has been built in a robust manner to prevent critical system errors from jeopardising the mission. However, it still does not allow for in-depth testing of the relationships between the spacecraft subsystems during large-scale failures. Taking advantage of the code base modularity achieved through the application of the SOLID principles, driver backends can be written in the Binar Software Framework to break dependency on hardware. This means that both offline spacecraft simulations and hardware-in-the-loop simulations are possible with minimal development effort to increase the quality of the software design and in-depth testability. This enables adherence to elements of software testing specified in ECSS-E-ST-40C, which describes the importance of demonstrating that the developed software can perform in a representative operational environment [110]. Therefore, moving forward the Binar team will build a low-level bounded simulation implementation of spacecraft subsystems for offline simulations. The backends for peripherals that give the spacecraft knowledge about its environment, such as the GPS, can be substituted with software that communicates with a simulation running on a host machine. This will allow the high level flight logic to remain unchanged, yet gain a deeper insight on how it would respond to a simulated space environment. A. da Silva et al. discuss the benefits of using fault injection into simulations of software developed for the Solar Orbiter mission, showing a strong improvement in software fault tolerance [134]. Applying a similar method to future Binar spacecraft will allow the satellite software to be tested on a much deeper level, providing earlier flight software verification. Developing the capability to replay researched failure scenarios into the spacecraft flight software will provide a deeper understanding of the causes for failure, and more importantly how to recover from them.

2.7. ACKNOWLEDGEMENTS

This research was supported by an Australian Government Research Training Program (RTP) Scholarship and received institutional support from Curtin University.

REFERENCES

[41] M. Langer and J. Bouwmeester, "Reliability of CubeSats - Statistical Data, Developers' Beliefs and the Way Forward," en, in *Proceedings of the AIAA/USU Conference on Small Satellites*, Aug. 2016, p. 12. [Online]. Available: `https://digitalcommons.usu.edu/smallsat/2016/TS10AdvTech2/4`.

[49] A. Alanazi and J. Straub, "Engineering Methodology for Student-Driven CubeSats," en, *Aerospace*, vol. 6, no. 5, p. 54, May 2019, ISSN: 2226-4310. DOI: `10.3390/aerospace6050054`. [Online]. Available: `https://www.mdpi.com/2226-4310/6/5/54`.

[69] I. Latachi, T. Rachidi, M. Karim, and A. Hanafi, "Reusable and Reliable Flight-Control Software for a Fail-Safe and Cost-Efficient Cubesat Mission: Design and Implementation," en, *Aerospace*, vol. 7, no. 10, p. 146, Oct. 2020, ISSN: 2226-4310. DOI: `10.3390/aerospace7100146`. [Online]. Available: `https://www.mdpi.com/2226-4310/7/10/146`.

[99] M. Tafazoli, "A study of on-orbit spacecraft failures," en, *Acta Astronautica*, vol. 64, no. 2-3, pp. 195–205, Jan. 2009, ISSN: 00945765. DOI: 10.1016/j.actaastro.2008.07.019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0094576508003019.

[100] N. G. Leveson, "Role of Software in Spacecraft Accidents," en, *Journal of Spacecraft and Rockets*, May 2012. DOI: 10.2514/1.11950.

[101] D. McComas, J. Wilmot, and A. Cudmore, "The Core Flight System (cFS) Community: Providing Low Cost Solutions for Small Spacecraft," in *30th Annual AIAA/USU Conference on Small Satellites*, Salt Lake City, UT, Aug. 2016. [Online]. Available: https://ntrs.nasa.gov/citations/20160010300.

[102] R. Bocchino, T. Canham, G. Watney, L. Reder, and J. Levison, "F Prime: An Open-Source Framework for Small-Scale Flight Software Systems," *Small Satellite Conference*, Aug. 2018. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2018/all2018/328.

[103] R. Plauche, "Building Modern Cross-Platform Flight Software for Small Satellites," *Small Satellite Conference*, Aug. 2017. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2017/all2017/204.

[104] M. Doyle, A. Gloster, C. O'Toole, J. Mangan, D. Murphy, R. Dunwoody, M. Emam, J. Erkal, J. Flanaghan, G. Fontanesi, F. Okosun, R. R. Nair, J. Reilly, L. Salmon, D. Sherwin, J. Thompson, S. Walsh, D. de Faoite, U. Javaid, S. McBreen, D. McKeown, D. O'Callaghan, W. O'Connor, K. Stanton, A. Ulyanov, R. Wall, and L. Hanlon, "Flight Software Development for the EIRSAT-1 Mission," in *Proceedings of the 3rd Symposium on Space Educational Activities*, 2020, pp. 157–161. DOI: 10.29311/2020.39. [Online]. Available: http://arxiv.org/abs/2008.09074.

[105] H. A. Askari, E. W. H. Eugene, A. N. Nikicio, G. C. Hiang, L. Sha, and L. H. Choo, "Software Development for Galassia CubeSat – Design, Implementation and In-Orbit Validation," en, in *31st International Symposium on Space Technology and Science (ISTS), 26th International Symposium on Space Flight Dynamics (ISSFD), and 8th Nano-Satellite Symposium (NSAT)*, Jun. 2017, p. 9.

[106] S. F. Hishmeh, T. J. Doering, and J. E. Lumpp, "Design of flight software for the KySat CubeSat bus," in *2009 IEEE Aerospace conference*, Mar. 2009, pp. 1–15. DOI: `10.1109/AERO.2009.4839646`.

[107] V. Carrara, R. B. Januzi, D. H. Makita, L. F. d. P. Santos, and L. S. Sato, "The ITASAT CubeSat Development and Design," en, *Journal of Aerospace Technology and Management*, vol. 9, no. 2, pp. 147–156, Apr. 2017, ISSN: 2175-9146. DOI: `10.5028/jatm.v9i2.614`. [Online]. Available: `http://www.jatm.com.br/ojs/index.php/jatm/article/view/614`.

[108] T. Whitney, J. Straub, and R. Marsh, "Design and Implementation of Satellite Software to Facilitate Future CubeSat Development," en, in *AIAA SPACE 2015 Conference and Exposition*, Pasadena, California: American Institute of Aeronautics and Astronautics, Aug. 2015, ISBN: 978-1-62410-334-6. DOI: `10.2514/6.2015-4500`. [Online]. Available: `https://arc.aiaa.org/doi/10.2514/6.2015-4500`.

[109] K. V. C. K. de Souza, Y. Bouslimani, and M. Ghribi, "Flight Software Development for a Cubesat Application," en, *IEEE Journal on Miniaturization for Air and Space Systems*, pp. 1–1, 2022, ISSN: 2576-3164. DOI: `10.1109/JMASS.2022.3206713`. [Online]. Available: `https://ieeexplore.ieee.org/document/9891727/`.

[110] Space Engineering Software, *ECSS-E-ST-40C: Space Engineering Software*, Mar. 2009.

[111] Space Product Assurance, *ECSS-Q-ST-80C: Space Software Product Assurance*, Feb. 2017.

[112] J. Clare and K. I. Kourousis, "Learning from Incidents: A Qualitative Study in the Continuing Airworthiness Sector," en, *Aerospace*, vol. 8, no. 2, p. 27, Feb. 2021, ISSN: 2226-4310. DOI: `10.3390/aerospace8020027`.

[113] R. Nilchiani, "Valuing software-based options for space systems flexibility," en, *Acta Astronautica*, vol. 65, no. 3-4, pp. 429–441, Aug. 2009, ISSN: 00945765. DOI: `10.1016/j.actaastro.2009.02.012`.

[114] X.-Y. Ji, Y.-Z. Li, G.-Q. Liu, J. Wang, S.-H. Xiang, X.-N. Yang, and Y.-Q. Bi, "A brief review of ground and flight failures of Chinese spacecraft," en, *Progress in Aerospace Sciences*, vol. 107, pp. 19–29, May 2019, ISSN: 0376-0421. DOI: 10.1016/j.paerosci.2019.04.002.

[115] C. Venturini, B. Braun, D. Hinkley, and G. Berg, "Improving Mission Success of CubeSats," in *Proceedings of the AIAA/USU Conference on Small Satellites*, Aug. 2018. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2018/all2018/273.

[116] T. E. Rumford, "Demonstration of autonomous rendezvous technology (DART) project summary," in *Space Systems Technology and Operations*, International Society for Optics and Photonics, vol. 5088, SPIE, 2003, pp. 10–19. DOI: 10.1117/12.498811. [Online]. Available: https://doi.org/10.1117/12.498811.

[117] S. Lilley, *Critical Software: Good Design Built Right*, Jan. 2012. [Online]. Available: https://nsc.nasa.gov/docs/default-source/system-failure-case-studies/good-design-built-right.pdf?sfvrsn=c6fecf8_4.

[118] S. Croomes, "Overview of the DART Mishap Investigation Results," en, National Aeronautics and Space Administration, Tech. Rep., 2006. [Online]. Available: https://www.nasa.gov/pdf/148072main_DART_mishap_overview.pdf.

[119] A. G. Stephenson, L. S. LaPiana, D. R. Mulville, P. J. Rutledge, F. H. Bauer, D. Folta, G. A. Dukeman, R. Sackheim, and P. Norvig, "Mars Climate Orbiter Mishap Investigation Board Phase 1 Report," NASA, Tech. Rep., Nov. 1999. [Online]. Available: https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf.

[120] Edward A. Euler, Steven D. Jolly, and H. H. Curtis, "The failures of the Mars Climate Orbiter and Mars Polar Lander: A perspective from the people involved," in *Proceedings of the 24th Annual AAS Guidance and Control Conference*, American Astronautical Society, Jan. 2001, p. 22. [Online]. Available: http://web.mit.edu/16.070/www/readings/Failures_MCO_MPL.pdf.

[121] J. L. Lions, "Ariane 5 Flight 501 Failure: Report by the Inquiry Board," European Space Agency, Tech. Rep., Jul. 1996. [Online]. Available: `https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf`.

[122] B. Harvey, *The rebirth of the Russian space program: 50 years after Sputnik, new frontiers*, en, 1st ed. New York: Springer, 2007, ISBN: 978-0-387-71354-0.

[123] A. Gorbenko, V. Kharchenko, O. Tarasyuk, and S. Zasukha, "A Study of Orbital Carrier Rocket and Spacecraft Failures: 2000-2009," *Information & Security: An International Journal*, vol. 28, pp. 179–198, Jan. 2012. DOI: `10.11610/isij.2815`.

[124] J. D. Mooney, "Portability and reusability: Common issues and differences," en, in *Proceedings of the 1995 ACM 23rd annual conference on Computer science - CSC '95*, Nashville, Tennessee, United States: ACM Press, 1995, pp. 150–156, ISBN: 978-0-89791-737-7. DOI: `10.1145/259526.259550`.

[125] C. Larman and P. Kruchten, *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development*. Prentice Hall PTR, 2005, ISBN: 978-0-13-148906-6. [Online]. Available: `https://books.google.com.au/books?id=tuxQAAAAMAAJ`.

[126] R. C. Martin, "Design Principles and Design Patterns," en, *Object Mentor*, p. 34, 2000. [Online]. Available: `http://staff.cs.utu.fi/staff/jouni.smed/doos_06/material/DesignPrinciplesAndPatterns.pdf`.

[127] R. C. Martin, *Agile software development, principles, patterns, and practices*, en, 1. ed., Pearson new international ed. Harlow: Pearson, 2014, ISBN: 978-1-292-02594-0.

[128] B. Liskov, "Data Abstraction and Hierarchy," *ACM SIGPLAN Notices*, vol. 23, no. 5, pp. 17–34, Jan. 1987, ISSN: 0362-1340. DOI: `https://doi.org/10.1145/62139.62141`.

[129] M. Källén, S. Holmgren, and E. P. Hvannberg, "Impact of Code Refactoring Using Object-Oriented Methodology on a Scientific Computing Application," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, Sep. 2014, pp. 125–134. DOI: `10.1109/SCAM.2014.21`.

[130] V. Garousi, M. Felderer, C. M. Karapıçak, and U. Yılmaz, "Testing embedded software: A survey of the literature," en, *Information and Software Technology*, vol. 104, pp. 14–45, Dec. 2018, ISSN: 0950-5849. DOI: `10.1016/j.infsof.2018.06.016`.

[131] M. Tipaldi, C. Legendre, O. Koopmann, M. Ferraguto, R. Wenker, and G. D'Angelo, "Development strategies for the satellite flight software on-board Meteosat Third Generation," en, *Acta Astronautica*, vol. 145, pp. 482–491, Apr. 2018, ISSN: 00945765. DOI: `10.1016/j.actaastro.2018.02.020`.

[132] K. Beck, *Test-driven Development: By Example*, ser. Addison-Wesley signature series. Addison-Wesley, 2003, ISBN: 978-0-321-14653-3.

[133] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: A survey," in *Proceedings of the 2002 ACM symposium on Document engineering*, ser. DocEng '02, New York, NY, USA: Association for Computing Machinery, Nov. 2002, pp. 26–33, ISBN: 978-1-58113-594-7. DOI: `10.1145/585058.585065`.

[134] A. da Silva, S. Sánchez, O. R. Polo, and P. Parra, "Injecting faults to succeed. Verification of the boot software on-board solar orbiter's energetic particle detector," en, *Acta Astronautica*, vol. 95, pp. 198–209, Feb. 2014, ISSN: 0094-5765. DOI: `10.1016/j.actaastro.2013.11.004`.

CHAPTER 3

BINAR-1 SOFTWARE DEVELOPMENT

3.1. INTRODUCTION

The quality of CubeSat software design directly impacts the resilience of a mission. A literature review in the previous chapter identified the benefits of developing CubeSat software in-house instead of implementing an established solution. Mature software frameworks and proprietary software development kits (SDKs) were discussed as being of interest to CubeSat software developers. Whilst benefitting from the flight heritage accelerating mission-concept-to-orbit in the short term, software written using this approach tightly couples the code base to the solution used. This tight coupling can make it difficult to break from the solution used if it no longer meets mission requirements or where it becomes difficult from a budgetary concern when proprietary software brings with it continuing licensing costs. With experience developing software for CubeSat missions, Doyle et al. recommend that space programs intending to launch multiple spacecraft should seek out full in-house development [104]. Reliable software design was highlighted as a critical element in risk mitigation for CubeSat missions [69]. The hardware decoupling achieved during the development of the custom BSF allowed for a significant effort to be put into testing. Extending on this, integration into the CI/CD pipeline enabled frequent automated builds, automated testing and static analysis, quantifying the reliability of the codebase. Moreover, in-house software framework development enables the software written to be modular, reusable, and extendable, reducing mission-concept-to-orbit time for subsequent missions. With a strong education goal, developing custom in-house software allows the Binar Space Program to offer staff and student engineers the opportunity to explore writing reliable software for a space system. Using established frameworks and proprietary SDKs can obfuscate solutions to unique problems inside a black box, such as how data is prepared for transmission to a spacecraft. The documentation available for the existing BSF, as well as the full availability of the source code, helps achieve the educational goal of the program by aiding in the training of new engineers progressing into the field of space engineering.

This chapter details the core elements of the Binar Software Framework (BSF). Due to the volume of information, the technical implementation of the BSF is provided in Appendix A. The primary focus of this chapter is to provide an overview of the design decisions behind the elements of the framework and their vital role in enhancing the reliability of Binar spacecraft. The BSF embodies a unique research-oriented approach to CubeSat software engineering, prioritising innovative design and practical functionality. The development is motivated by the need to surpass traditional software approaches to CubeSat software design and create innovative solutions to the unique challenges of CubeSat development.

Some of the unique challenges faced by software development for CubeSats include limited system resources, the operational environment, communication constraints, development speed and lack of standardisation. The specific challenges encountered and the innovative solutions devised are discussed, demonstrating the research contributions to CubeSat software engineering. This chapter is organised into sections that address the Hardware Abstraction Layer, Utilities, Boot Application Modules, Application Modules, Communications Modules, and Operation Modes, each of which plays a pivotal role in the BSF's capabilities. The Hardware Abstraction Layer section covers libraries designed for hardware interaction, while Utilities discusses libraries for general use within the BSF. Boot Application Modules offers insights into libraries used during the CubeSat's boot process, and Application Modules delves into libraries utilised in the application code. The Communications Modules section explores libraries facil-

TABLE 3.1: The UML class diagram arrows used in this chapter with their meanings.

Arrow	Meaning	Description
─▷	Inheritance	Child object inherits (and can extend) the functionality of a parent
─►	Template instantiation	Object implements a class through templating
─◇	Aggregation	Objects are associated, but the child can exist without the parent object

itating communication between the ground station and the spacecraft. Finally, the Operation Modes section sheds light on how these libraries are implemented in the satellite's firmware. The candidate wrote the BSF to enable regular satellite launches and reduce the risk of mission failure by promoting extensibility, adaptability, and enhanced performance in CubeSat missions. The framework's relevance extends to both current and future missions.

The previous chapter demonstrates the first application of SOLID principles in CubeSat software development. In addition to presenting innovative software solutions for addressing specific challenges in CubeSat missions, this chapter contributes to CubeSat software development by providing practical examples of how SOLID principles can be applied in the space domain. To illustrate the benefits achieved from adhering to SOLID principles in library development, the candidate presents examples of what library development would resemble without their application. This comparative analysis underscores the contribution that the application of the SOLID principles has in enhancing mission reliability within the CubeSat software domain.

Unified Modelling Language is used in this chapter to describe the structure of libraries. Table 3.1 summarises the arrows used and their corresponding meanings.

3.2. HARDWARE ABSTRACTION LAYER

A unique challenge of CubeSat missions is the rapid development cycle. CubeSat missions have shorter development cycles and lower budgets than larger space missions. Whilst larger missions may benefit from longer mission timeframes, which allow for the development of bespoke mission software, the timeframe available to CubeSat development may be short such that it precludes the mission from writing software tailored for a mission that safely satisfies the exact mission requirements. It therefore follows that software development faces the seemingly contradictory challenge of developing safe software at a rapid pace. Developing a CubeSat hardware abstraction layer (HAL) is a solution to this challenge, enabling agile, cost-effective software development for short mission lifecycles. A HAL is a software layer that serves as an intermediary between the system hardware and the high-level software that runs on it. It provides a standardised interface to use system peripherals consistently while abstracting away hardware-specific details. Abstraction of these details enables software to be written independently of the hardware it is intended to run on.

Development of the BSF HAL saw strict adherence to the SOLID principles of object-oriented design. Whilst the concept of a HAL itself is not novel, constructing a CubeSat HAL following the SOLID principles to address the dual challenge of ensuring software safety while meeting tight development schedules represents an innovative approach. The absence of prior published work in the literature on the construction of a SOLID-compliant HAL for expediting CubeSat missions accentuates the novelty of this approach. This section will provide an overview of the libraries written to abstract the hardware on Binar-1.

The hardware abstraction provided by the libraries in the BSF is essential in supporting the software to be developed in parallel to the spacecraft hardware. It enables the hardware to be changed without impacting the flight software. As the flight software interfaces with the peripheral through the HAL, a new implementation for the hardware to be supported can be added. Aside from external physical peripherals such as the GPS, IMU, and magnetorquers, this also includes peripherals internal to the flight computer, such as GPIO, ADC, and communications. Developing a HAL enables the Binar Space Program to overcome the challenges of a rapid development cycle.

3.2.1. FLIGHT COMPUTER BOARD SUPPORT

GPIO

The Binar CubeSat Motherboard is an example of an embedded system: a computer system designed to perform specific tasks within a larger system. It typically consists of a processor, memory and peripherals integrated to achieve specific tasks. In the context of the Binar-1 satellite, the BCM was designed to control the spacecraft and monitor its environment. It achieves this goal by controlling the onboard peripherals physically connected to it through a hardware trace on the PCB that houses them. The traces connecting to the microcontroller can be grouped into either inputs or outputs. The microcontroller and peripheral devices can then communicate by fluctuating the voltage on the IO lines between an off state and an on state, normally zero volts for the former and the internal reference voltage of the microcontroller for the latter. Controlling peripherals via IO pins can become more complex to support the transmission of messages. However, at the most fundamental level, simple state assertion is used. Using IO pins for interaction with a peripheral is not dependent on the peripheral. Hence, the pins are named *general purpose* input/output pins.

However, using software to instruct a computer to assert or read from these GPIO pins varies from computer to computer. It normally requires hardware register manipulation specific to the hardware architecture used by the chosen processor. Having to rewrite application code controlling spacecraft peripherals due to a change of flight computer contradicts the recommendation made in the previous chapter regarding code reuse. The BSF provides a solution to this through the GPIO library. The GPI and GPO objects in the library (Figures 3.1 and 3.2) are a crucial component of the flight computer system that simplifies interacting with hardware digital signals. This simplification makes it easier for developers to write reliable and efficient code for the flight computer. Table 3.2 summarises the GPIO pins assigned to the flight computer on Binar-1.

The manufacturer of the chosen microcontroller for the Binar-1 flight computer provides a custom abstraction layer over the register manipulation for controlling the

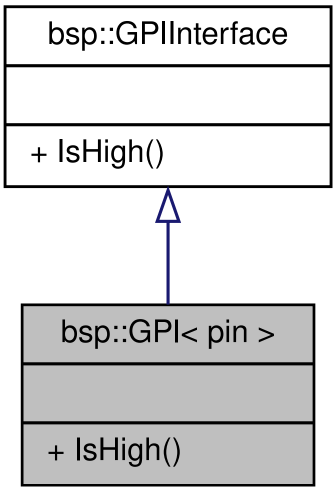


FIGURE 3.1: A UML representation of the GPI object on Binar-1. The GPI object inherits the GPIInterface and is further templated on all input pins. These templated objects are used to read digital signals.

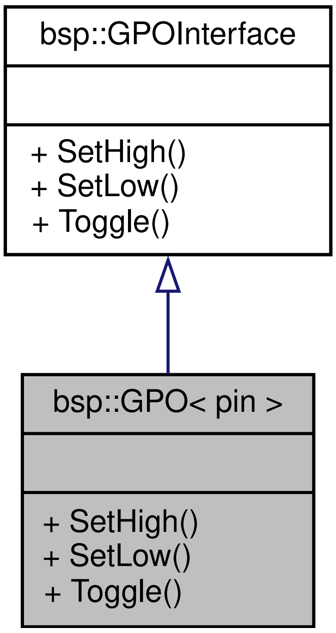


FIGURE 3.2: A UML representation of the GPO object on Binar-1. The GPO object inherits the GPOInterface and is further templated on all output pins. These templated objects are used to control digital output signals.

TABLE 3.2: A summary of the GPIO pins available on the Binar-1 flight computer.

Pin	Mode	Pin	Mode
Gyro/accelerometer chip select	Output	Z-axis torquer enable	Output
Magnetometer chip select	Output	Z-axis torquer fault	Input
Magnetometer data ready	Input	Redundant BCM disable	Output
Redundant X-axis torquer phase	Output	Supervisory computer disable	Output
Redundant X-axis torquer enable	Output	IMU enable	Output
Redundant X-axis torquer fault	Input	Payload chip select	Output
Redundant Y-axis torquer phase	Output	Battery voltage okay	Input
Redundant Y-axis torquer enable	Output	X-axis solar panel okay	Input
Redundant Y-axis torquer fault	Input	Y-axis solar panel okay	Input
X-axis torquer phase	Output	Payload enable	Output
X-axis torquer enable	Output	Burn wire 1	Output
X-axis torquer fault	Input	Burn wire 2	Output
Y-axis torquer phase	Output	GPS enable	Output
Y-axis torquer enable	Output	GPS low power mode	Output
Y-axis torquer fault	Input	Primary 5V enable line	Output
Z-axis torquer phase	Output	Secondary 5V enable line	Output

GPIO pins. However, wrapping this abstraction layer with the GPIO library allows the flight computer to be easily swapped out without impacting the application code. Appendix A.1 provides a detailed overview of the library.

The GPI and GPO objects responsible for interacting with hardware peripherals adhere to the Single Responsibility principle, as discussed in the previous chapter. Both objects are templated using an enumerated class that exhaustively lists all possible peripheral connections, ensuring that each template implementation controls only one peripheral. The alternative approach would be to have a singular class responsible for controlling every GPIO line on the flight computer. This approach would be difficult to maintain, as a change to any GPIO implementations would require the entire object to change. Moreover, it would not be scalable as the complexity of the object would grow with every new GPIO device support added. Adhering to the Single Responsibility principle promotes maintainability and flexibility, allowing for easier modifications and updates to the codebase without impacting application code.

ANALOG

Alongside digital signals in an embedded system, some peripheral devices also represent their state through analog signals. In the context of the BCM, this comprises fluctuating voltages sampled against the microcontroller's internal reference voltage. Like the GPIO library, sampling these fluctuating voltages varies from computer to computer. The ADC peripheral requires architecture-specific register access, and the internal reference voltage the ADC uses for comparison may also be hardware specific. Moreover, the resolution of the ADC will determine the precision by which it can sample the input voltage. Embedded systems may also favour external ADCs due to their higher resolution.

Contributing to the HAL's focus on addressing the demands of rapid mission timelines, the Analog library within the BSF effectively handles interactions with the ADC while promoting code reusability. The library samples voltages from all system sensors, including voltages, currents and temperatures. Figure 3.3 provides a high-level UML diagram demonstrating how analog voltages are read onboard the satellite. Appendix A.2 provides a detailed overview of the library.

The library follows the Single Responsibility principle through its use of templating. The Analog module within the library is templated using an enumerated class that exhaustively lists all available connections. This results in multiple objects that are each responsible for monitoring a single analog channel. Conversely, if the Single Responsibility principle were not followed, a single large class would be responsible for monitoring every available analog line. This approach would be difficult to maintain, as changes to any analog sampling method would reflect a change in the entire class. Moreover, this is not scalable, as adding further responsibilities to the large class would increase its complexity.

Following the Single Responsibility principle supports reusability by allowing updates to the analog sampling methods as new hardware revisions are introduced without impacting application code. It supports extensibility by allowing extra sampling channels to be added by appending to the enumerated class and supplying the required implementation. Moreover, as each templated object is responsible for its own

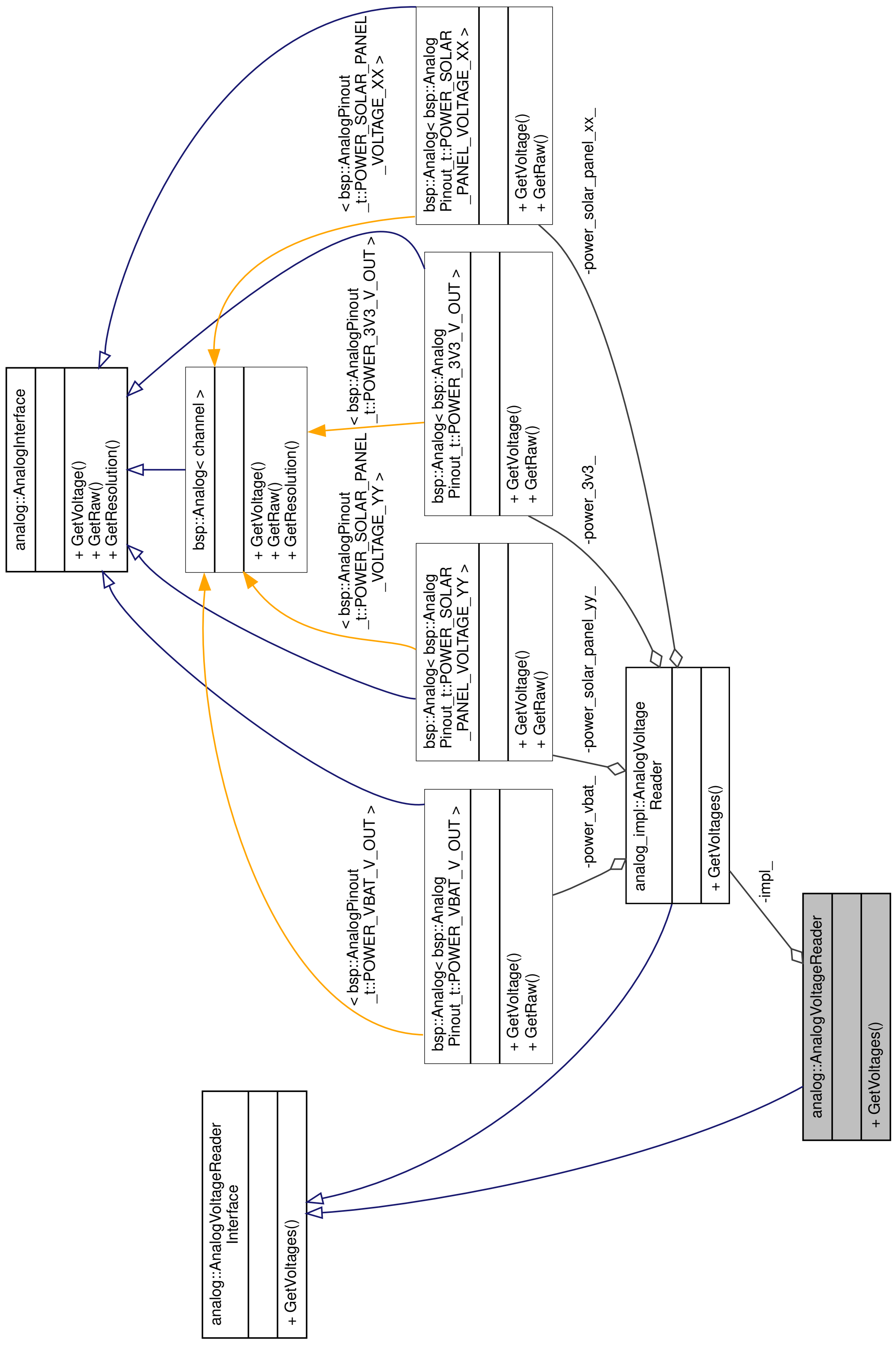

analog::AnalogInterface
+ GetVoltage()
+ GetRaw()
+ GetResolution()
bsp::Analog< channel >
+ GetVoltage()
+ GetRaw()
+ GetResolution()
bsp::Analog< bsp::Analog Pinout_t::POWER_SOLAR _PANEL_VOLTAGE_XX >
+ GetVoltage()
+ GetRaw()
< bsp::AnalogPinout _t::POWER_SOLAR_PANEL _VOLTAGE_XX >
bsp::Analog< bsp::Analog Pinout_t::POWER_3V3_V_OUT >
+ GetVoltage()
+ GetRaw()
< bsp::AnalogPinout _t::POWER_3V3_V_OUT >
bsp::Analog< bsp::Analog Pinout_t::POWER_SOLAR _PANEL_VOLTAGE_YY >
+ GetVoltage()
+ GetRaw()
< bsp::AnalogPinout _t::POWER_SOLAR_PANEL _VOLTAGE_YY >
bsp::Analog< bsp::Analog Pinout_t::POWER_VBAT_V_OUT >
+ GetVoltage()
+ GetRaw()
< bsp::AnalogPinout _t::POWER_VBAT_V_OUT >
-power_solar_panel_xx_
-power_3v3_
-power_solar_panel_yy_
-power_vbat_
analog_impl::AnalogVoltage Reader
+ GetVoltages()
-impl_
analog::AnalogVoltageReader
+ GetVoltages()
analog::AnalogVoltageReader Interface
+ GetVoltages()

FIGURE 3.3: A UML representation of the AnalogVoltageReader object on Binar-1. The temperature and current reader libraries follow the same structure.

analog device, each device can use a different post-processing method. Encapsulating this within the appropriate templated object provides a straightforward interface for developers to access analog data without worrying about the underlying hardware implementation. This design choice means that the library can adapt to the system's changing needs without requiring significant changes to the application code.

The Interface Segregation principle is adhered to in this library to increase modularity and promote reusability. Each analog reading object, comprising voltages, currents and temperatures, extends its own interface instead of using a common one. Following this approach, the library's clients and the analog signals' implementation details are decoupled. This aids in making the library more flexible and easy to maintain. Moreover, having a single interface responsible for a specific type of analog signal aids in clarifying the operation of the library. Lastly, this approach aids in testing. Tests can be written specific to each interface, allowing the behaviour of each analog reader to be verified independently of the other types.

The library uses the Open-Closed principle by having upstream classes closed for modification but open for extension. The AnalogInterface class is a common class that all types of Analog inputs extend to provide sampling functionality. This common class promotes reusability as it requires application-specific hardware details to be implemented by extending the class. The objects concerned with reading currents and temperatures on the spacecraft extend the analog object to add additional functionality for scaling the voltage value read into appropriate units.

The Liskov Substitution principle is leveraged in the library to remove code duplication and support scalability. The "GetScaledVoltage" method interacts with the analog device under examination through an abstracted pointer to the Analog class it derives. Hence, implementing this method only needs to be written once, reducing the likelihood of failure-inducing bugs being introduced into the codebase. Moreover, as the library is extended to add support for different analog devices in the future, this section of the library can remain unchanged, removing the need for regression testing.

INTERRUPT

During application runtime, the embedded processor may need to interact with a peripheral which may not be able to operate instantly. In these situations, the application code may continuously poll the peripheral, checking for information. However, this is a waste of clock cycles that could be used to process other instructions. The solution is to offload the peripheral monitoring to a dedicated internal controller, which can request the processor to stop executing code and action on information when it is ready. This controller process is referred to as an interrupt. The application code can register an interrupt handler which contains custom code to be executed on some defined event. When the processor receives the interrupt request, it can save its current stack pointer, execute the registered interrupt handler, and then return to the application code. Events that trigger interrupt service routines (ISR) in the Binar-1 flight computer comprise GPI and UART communications.

The interrupt library in the BSF is built according to the Liskov Substitution principle by heavily leveraging polymorphism. Application code creates an interrupt handler by inheriting and implementing the ExternalInterruptInterface object in the library, which contains the single abstract virtual method, ”Handler”. Any interrupt object that extends this interface can be used in place of the interface from which it is derived. This approach was used as the contents of the interrupt handler to be executed in the ISR may be application specific.

This section of the library adheres to the Open Closed principle. By closing the ExternalInterruptInterface object for modification, application-specific interrupt details must be provided by extending the interface. This approach promotes reusability as the ExternalInterruptInterface object is not coupled to any application specifics.

A custom static global interrupt vector table (IVT) exists within the library, providing a lookup between interrupt-enabled peripherals and the registered handler. The size of this IVT is determined at compile time using a custom enumerated class type. This enumerated class provides a list of interrupt-enabled peripherals. This enumerated type implicitly assigns unique integers, indexed from zero, to the entries in the type. Aside from offering clarity in accessing the desired interrupt entry through meaningful variable names, this enumerated list provides the number of interrupts required by the system. This allows the size of the IVT to be statically allocated at compile time. The data contained within the IVT is an array of pointers to ExternalInterruptInterface objects. Per the Liskov Substitution principle, the application-specific object derived from the ExternalInterruptInterface can be polymorphically substituted into the IVT for use during the ISR. Each interrupt-enabled peripheral on the BCM can have only one interrupt handler associated with it at any given time.

Technical details demonstrating the flow of operations during an interrupt trigger are provided in Appendix A.3.

SERIAL

In embedded systems, interacting with some peripherals requires transferring information that a simple state change cannot represent. For example, suppose the GPS peripheral wants to inform the flight computer of the satellite's location in orbit. In that case, it must communicate a series of values representing geodetic coordinates, substantially more complex than an on or off state available by GPIO or a variable voltage offered by the ADC. These situations require the use of serial communications. Serial communications refer to transmitting information one bit at a time as part of a structured packet. There are many types of serial communications. The serial library in the BSF provides support for the following:

- USART: Universal Synchronous/Asynchronous Receive/Transmit
- I2C: Inter-Integrated Circuit
- SPI: Serial Peripheral Interface

Configuration and usage of serial communication methods vary depending on the chosen processor. Hence, the manufacturer of the processor chosen as the flight computer provides an abstraction layer to make using these peripherals straightforward. The serial library, as part of the HAL, provides another abstraction layer on top of this, contributing to the solution to the challenges imposed by rapid mission timelines. The

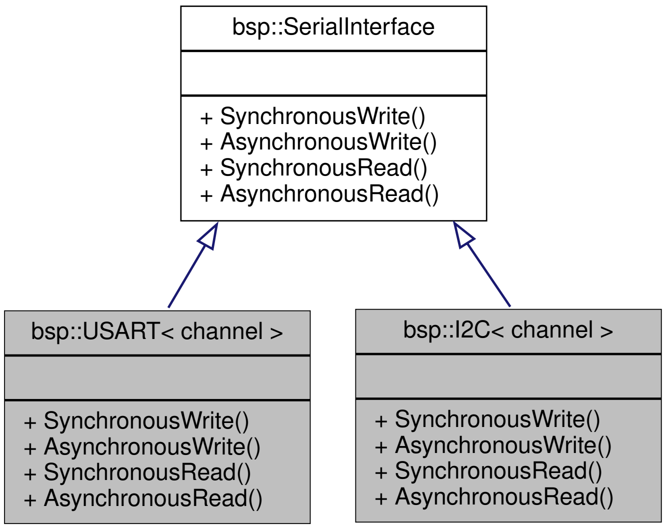


FIGURE 3.4: The USART and I2C communications objects in the BSF share a common interface for serial communications.

serial library is developed adhering to the Open Closed principle, in which the basic functionality for a serial device is defined in an interface class, which is then closed for modification. By extending the interface, support for the specific serial method used can be added. Of the three serial communications methods required for Binar-1, two communicate similarly. The peripherals using the USART and I2C communications methods only support half-duplex, meaning information can only flow in one direction. Hence, the UART and I2C objects extend a common interface, demonstrated in Figure 3.4.

Coupled with the Dependency Inversion principle, this library's Open Closed principle design approach promotes code reusability as high-level code that requires a serial implementation can interact with the serial device polymorphically through the abstracted interface. This approach means that the high-level code is not coupled to the specific communication method. Moreover, deriving from a common interface reduces the possibility of introducing failure-inducing bugs through code duplication. Lastly, the SerialInterface class can be extended to provide a mocked implementation to be used during integration testing to break the dependency on hardware.

Conversely, peripherals on the BCM requiring the SPI method use full-duplex communication, meaning data is transferred to and received from peripherals simultaneously. Hence, the SPI module in the Serial library inherits and implements its own interface class supporting full-duplex communications. This approach further supports reusability by allowing future full-duplex communication methods to extend this interface and provide the required implementations. This approach would see the same benefits as the half-duplex implementation.

The library uses the Single Responsibility principle by deriving a communication method-specific class from the appropriate interface class. These derived objects further follow the Single Responsibility principle through templating. The derived classes are templated using an enumerated class that lists all connected communications peripherals on Binar-1. Support for a new serial channel is added by extending the enumerated type with a new channel entry, demonstrating the scalability of the library.

3.2.2. OWNERSHIP

Using physical peripherals onboard a spacecraft can be problematic as hardware is a finite resource, unlike software. Considering Binar-1 used asynchronous execution, it was apparent that the conventional approach of constructing and using a peripheral when needed could cause the software to suffer from issues associated with concurrent programming, such as race conditions and deadlocks. The BSF, therefore, reverses the approach of user code constructing and owning peripherals to having a single global manager control access to all peripherals existing in a higher scope, allowing access to them when available.

The Manager object (Figure 3.5) follows the Single Responsibility and Dependency Inversion principles to interact with the spacecraft hardware. Although the global manager controls access to the peripherals, the memory for the peripheral objects does not exist within the scope of the manager class. The peripheral objects are constructed in a higher scope, not accessible by user code. These peripherals are then supplied to the Manager object through dependency injection. The result of this design choice is that the Manager object is not coupled to the specific peripherals it controls. The alternative approach would be a large God Object with all peripherals within it, thus having many dependencies and responsibilities for their operation. This antipattern reduces reusability as future missions may not have the same peripherals. Moreover,

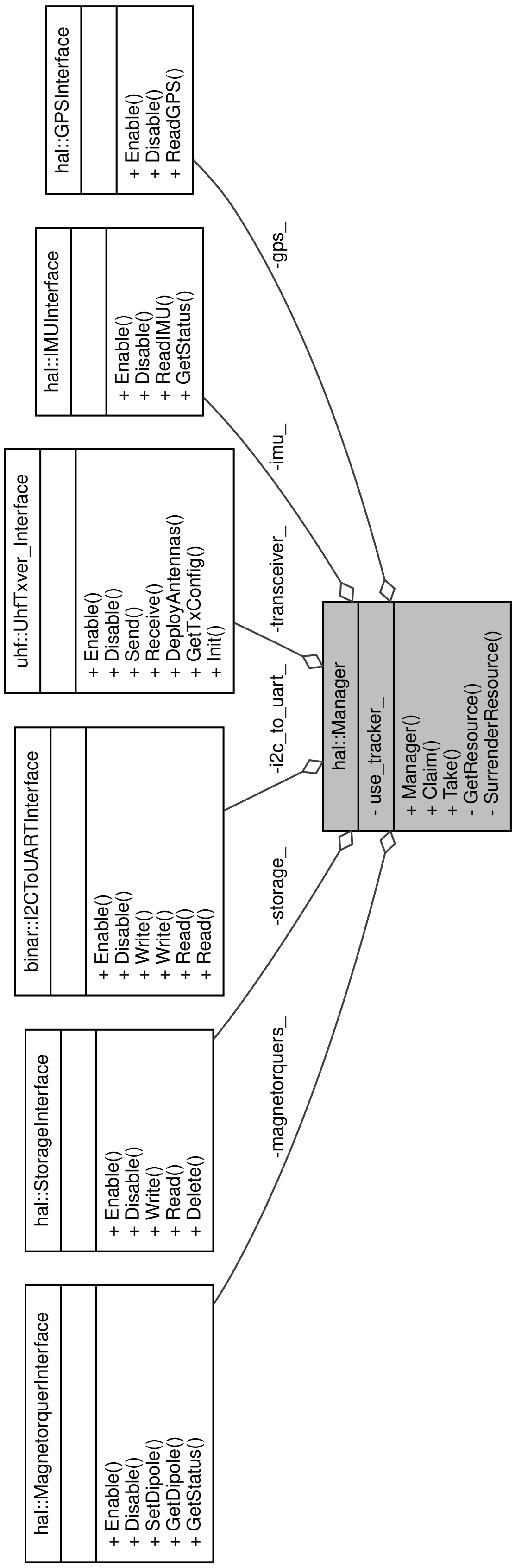


FIGURE 3.5: A UML representation of the Manager class from the Ownership library. The Manager object aggregates pointers to all system peripheral interfaces and controls access through the "Claim" and "Take" methods.

changing the functionality of a peripheral would require the whole Manager class and every other peripheral within it to also change, potentially introducing failure-inducing bugs.

Every system peripheral object in the BSF follows the Liskov Substitution principle for usage in the Ownership library, in which the derived peripheral implementation classes can be substituted in place of their abstract parent class. The Manager aggregates pointers to these interfaces of the system hardware peripherals to leverage polymorphism and control the peripheral implementation through an abstraction. Polymorphism in this application aids flight software to be tested and run on a host machine independent of the hardware by providing new implementations of peripherals that inherit from the same base class. Moreover, it supports the reusability of the library by allowing the implementation of the peripherals used to be application-specific without impacting the high-level code that uses the library. Lastly, it allows peripheral fakes to be dynamically substituted instead of faulty peripherals in the Manager at runtime without impacting the application code. This approach provides an innovative solution to tackle the challenge imposed on satellites of operations in a harsh environment. Exposure to radiation, extreme temperatures and vacuum conditions can damage system peripherals beyond repair. In this scenario, the system can continue functioning even when a hardware peripheral is faulty or unavailable by using fake peripherals. Leveraging polymorphism to seamlessly switch between real and fake peripherals to ensure the satellites's continued operation in the presence of hardware issues enhances the spacecraft's resilience and robustness. This level of flexibility and adaptability is valuable in space systems as it allows for fault tolerance and graceful degradation.

Before a peripheral is used through the Ownership object, a fail flag for that resource is set in the spacecraft error registers before the peripheral is powered. This flag is only reset by the "Disable" method, called either after the device has been used or the taken PeripheralHandler destructs. Suppose the spacecraft experiences an unexpected reset when the peripheral is used. In that case, the application code constructing the peripheral manager will identify the fail flag, shown in Figure 3.6.

The faked objects used in the manager's construction allow the flight logic to execute without requiring a firmware update to prevent a faulty peripheral from

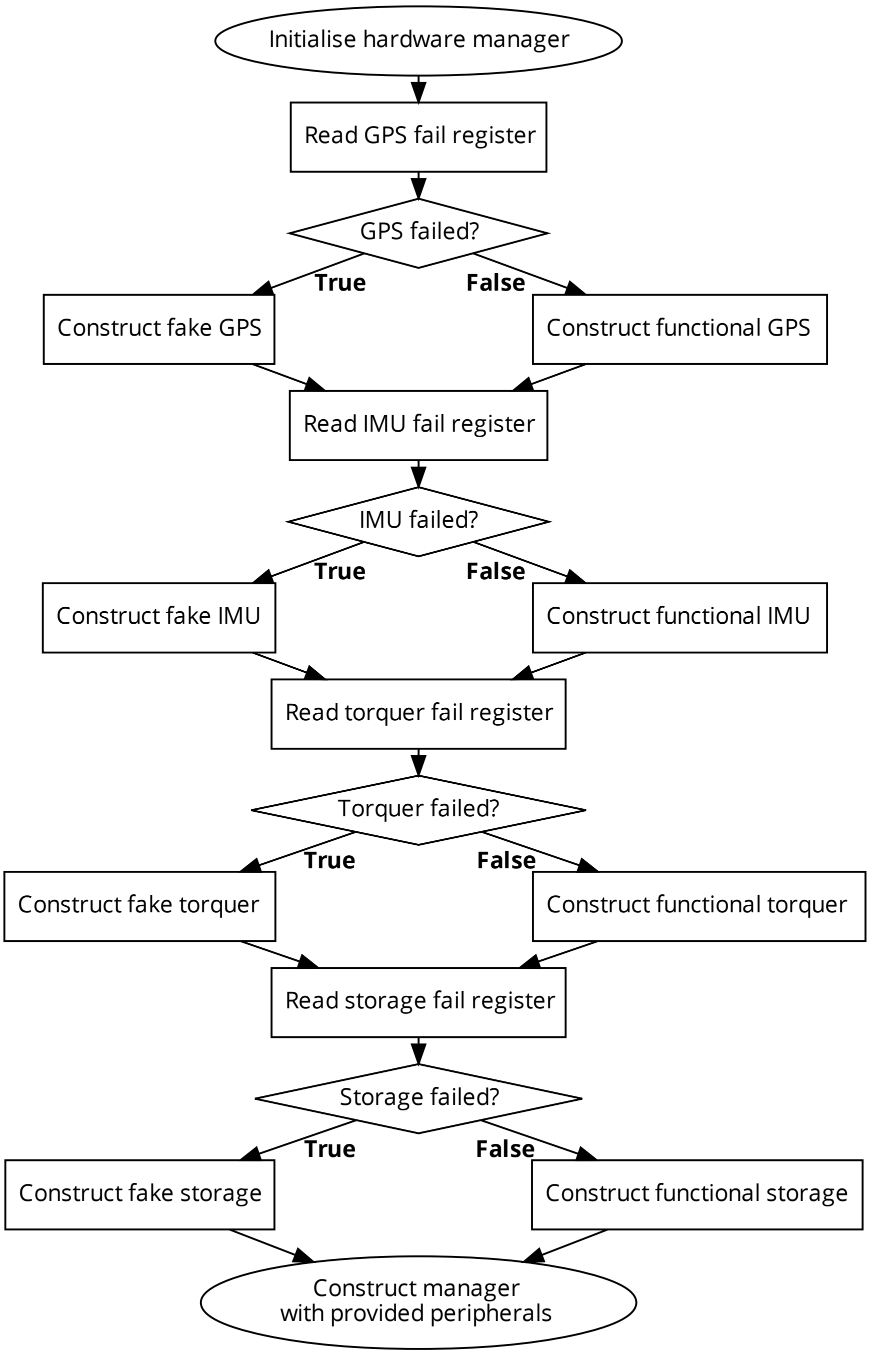


FIGURE 3.6: Constructing the ownership manager. As peripherals are stored within the manager as pointers to objects that derive the peripheral's interface, fakes can be substituted polymorphically to not impact application code if a peripheral device fails.

being used. Only a ground operator can reset these error flags as a failure prevention method. Technical details regarding how application code interacts with the underlying peripherals are provided in Appendix A.4.

3.2.3. IMU

An inertial measurement unit (IMU) is a common device used on spacecraft to discern information regarding the spacecraft's movement. As they are often external devices to the main processor, they employ serial communications to supply the flight computer with this information. IMUs vastly differ regarding the serial method used and the device configuration. Hence, the IMU library in the BSF is used to abstract the application code from IMU interactions to support the ability to change the selected hardware. The IMU peripheral chosen for Binar-1 also contains a magnetometer on the same die. The public-facing IMU object (Figure 3.7) provides access to the hardware peripheral through the private magnetometer_ and accel_gyro_ members. These are used with the driver supplied by the manufacturer to interact with the hardware.

The IMU library was developed adhering to the Open Closed principle. The IMUInterface is extended into the IMUBase class, where implementation-generic functionality is provided in the "ReadIMU" method. The hardware-specific implementation through the various getter methods, however, is not. The base class, being closed for modification, needs to be extended to provide this functionality. This approach promotes reusability by decoupling the IMU object from the hardware that it controls. Moreover, this design choice allows the behaviour of the "ReadIMU" method to be unit tested by extending on this base class to provide a testing implementation with mocked getter methods. Technical details of the functionality of the IMU hardware implementation on Binar-1 are provided in Appendix A.5.

3.2.4. MAGNETORQUER

To achieve on-orbit objectives and facilitate communications, spacecraft need to be able to adjust their attitude. Binar-1 used a magnetorquer-based attitude control method.

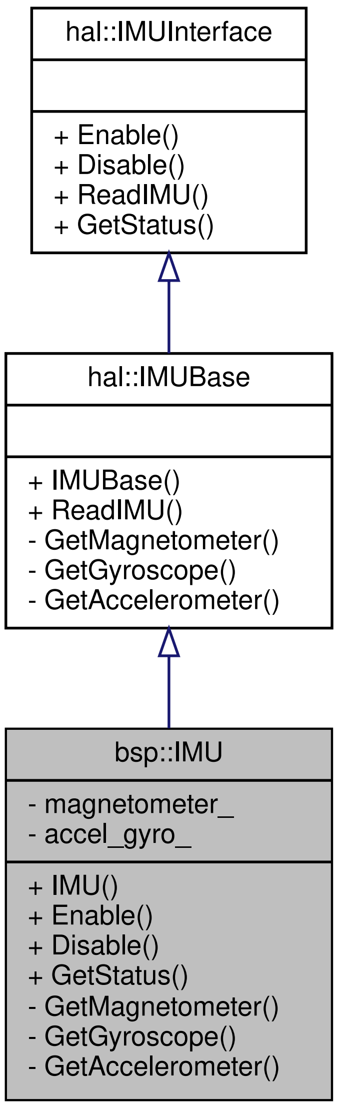


FIGURE 3.7: A UML representation of the IMU object on Binar-1.

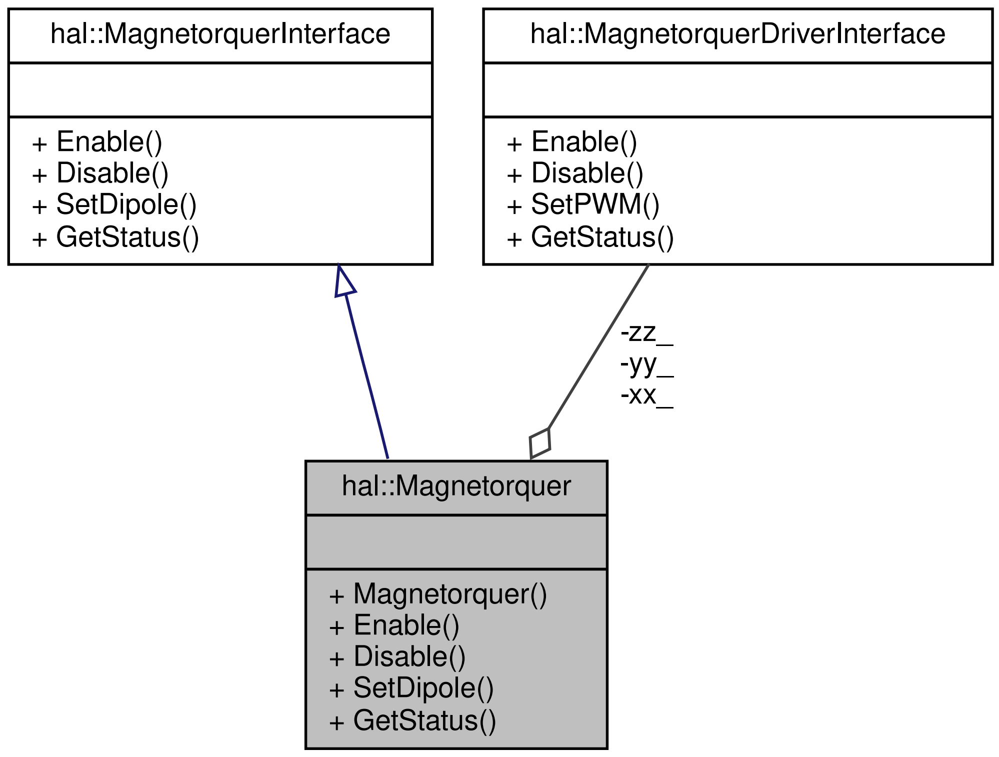


FIGURE 3.8: A UML representation of the magnetorquer object on Binar-1.

The magnetorquer library in the BSF facilitated interactions between the flight computer and the onboard attitude control device. The library provides an easy-to-use module called Magnetorquer (Figure 3.8) that enables the application code to control the three independent axes of the magnetorquer device.

The library uses the Dependency Inversion principle by having the high-level Magnetorquer object depend on abstractions of the MagnetorquerDriver (Figure 3.9) implementation objects. Moreover, it adheres to the Liskov Substitution principle by allowing the magnetorquer drivers under control to be substituted at runtime by any subclasses extending the interface. This innovative approach to magnetorquer control supports seamless switching to redundant onboard magnetorquers if required, which ensures the spacecraft's robustness to failure. Moreover, fake magnetorquers can be substituted at runtime to allow the code to disable an axis for safety. The alternative of having the high-level Magnetorquer object tightly coupled to the three axes it controls would mean that switching to redundant axes at run time would not be possible. To add support for them, the Magnetorquer class would need to be modified. This approach does not promote reusability, as not every mission will have the same number

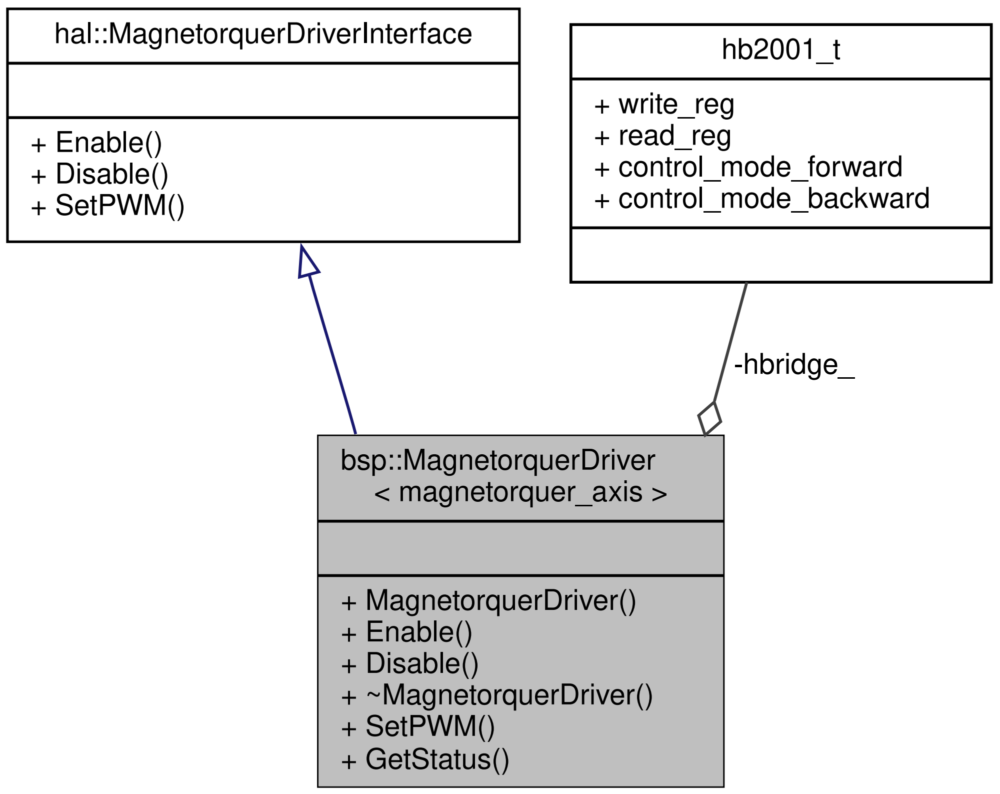


FIGURE 3.9: A UML representation of the magnetorquer driver object on Binar-1.

of redundant axes available.

The Magnetorquer library's design follows good software development principles, promoting reusability and decoupling from hardware. The Open Closed principle is leveraged to promote reusability by extending the MangetorquerDriverInterface to provide the hardware-specific implementations required for Binar-1 rather than coupling them directly to the interface. The hardware implementation of the magnetorquer axis object is templated, resulting in multiple axis-specific drivers being created. Templating this object instead of implementing unique classes for each axis forces all hardware-implemented axes to use the same API. This approach aims to prevent runtime bugs as changes to the API of the templated object must be reflected in the template definitions for the axes for compilation to succeed. Moreover, this allows for the support of extra axes by adding entries to the enumerated class.

The flight implementation of the MagnetorquerDriver object aggregates the custom driver for the H-bridge integrated circuit (IC) for interaction with the hardware. The write_reg and read_reg function pointers of the private hb2001_t aggregated driver, hbridge_, are assigned at compile time to provide the device-specific input/output. Assigning this at compile time allows the driver library to be decoupled from the hardware. Similarly, the control_mode_forward and control_mode_backward function pointers decouple how the driver configures the hardware device to adjust polarity. This approach promotes the reusability of the low-level driver on other platforms.

The design choices used during the development of this library facilitate extensive testing. The MagnetorquerDriverInterface can be extended to provide a mocked implementation of the magnetorquer drivers, which can be controlled polymorphically by the high-level Magnetorquer object. Examining how the high-level software interacts with the mocked methods can aid in verifying that the library functions as intended. Technical details of the Magnetorquer library are provided in Appendix A.6.

3.2.5. STORAGE

Persistent storage is required in space missions to retain data too large to fit into processor RAM and ensure its preservation during power cycling of the flight computer. Depending on mission requirements, many different types of storage technology can be used for a space mission. Hence, the storage object in the BSF is responsible for abstracting how the flight software interacts with the selected onboard storage peripheral. Abstracting interactions with the storage peripheral enables future hardware revisions without impacting application code. The UML representation of the library is given in Figure 3.10.

The Storage library follows the Open Closed principle by extending the StorageInterface object to provide the functionality required for interacting with the hardware storage peripheral on Binar-1. Following this principle promotes the reusability of the library by decoupling the interface from the application-specific implementation. On Binar-1, the storage implementation created a filesystem using an open-source library. If, conversely, the interface was modified to support using this filesystem, it would mean every future implementation of the Storage library would need to depend on the specific filesystem library used. This approach would make breaking from this file sys-

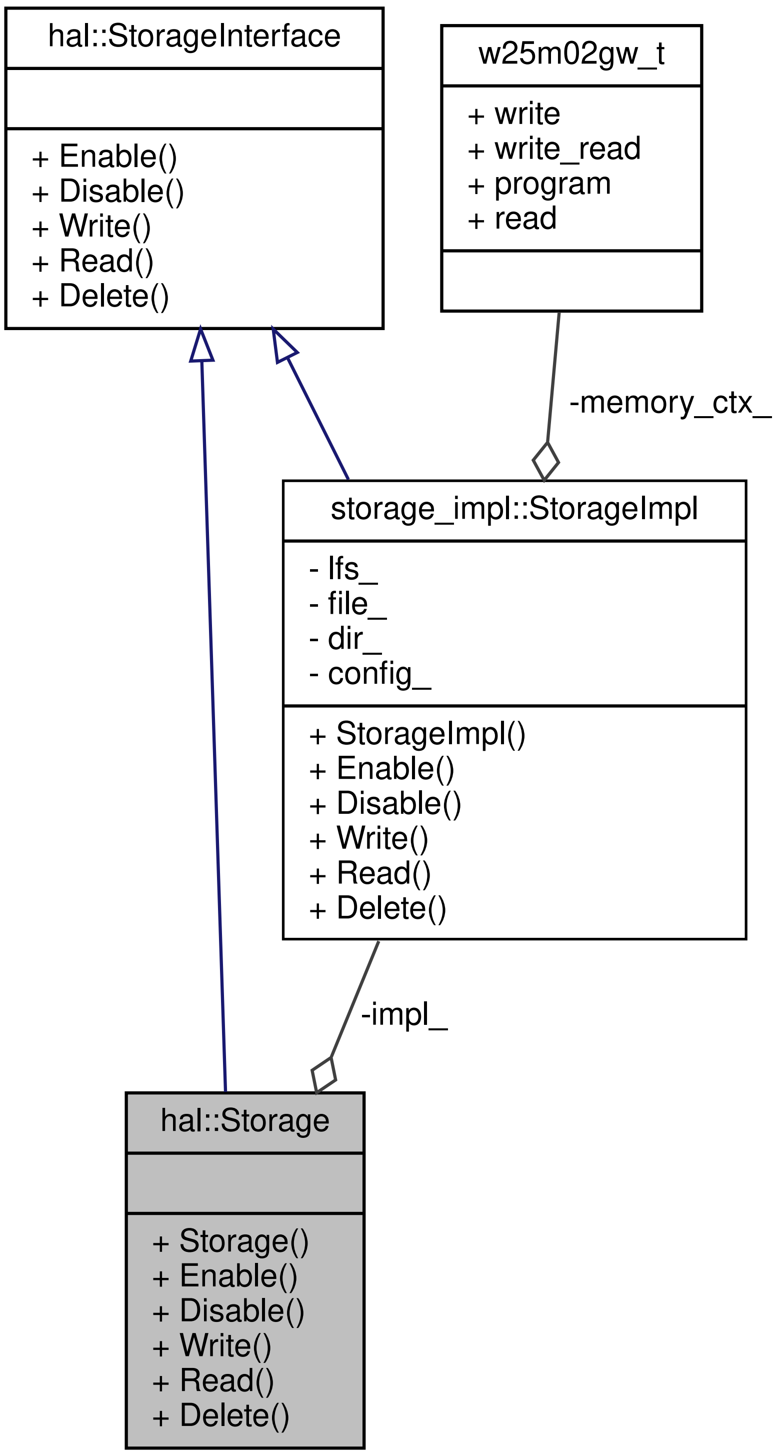


FIGURE 3.10: A UML depiction of the Storage object from the storage library. The Storage object aggregates an implementation object derived from the same interface. This object is concerned with persistent data storage external to the flight computer's flash memory.

tem difficult if it is no longer needed or fails to meet mission requirements. Following the Dependency Inversion principle, the high-level Storage object controls this interface extension through the abstracted member variable impl_. This approach allows for increased flexibility and maintainability, as the high-level storage object can interact with any object that extends the StorageInterface without requiring modification to the application code. The benefits of this approach have already been seen through reusing the library on Binar-2, 3 and 4. A new storage device was used on the newly revised BCM. A driver was written to support this storage device by extending the StorageInterface. As the high-level Storage object interacts with the implementation through an abstraction, the existing flight logic remains unaffected.

Alongside the design decision for the Storage library to facilitate the replacement of the storage IC used, the custom low-level device driver was also created to allow for a change in the flight computer. The custom driver implements most of the necessary functionality for device interaction but requires the application code to supply the features for serial interactions. The library provides these features via the public function pointer members in the w25m02gw_t object. This approach enhances library reusability by enabling developers to create serial interactions for a new flight computer while reusing most of the library code. Moreover, it removes the requirement for regression testing as the library remains unchanged.

Adherence to good software design practices in this library aids with software testing. The StorageInterface can be extended to provide a mocked storage device implementation during integration testing. Controlled through the same abstraction by the high-level Storage object, software storing data with the object can be verified to operate correctly by examining interactions with the mocked implementation. The low-level driver is unit tested by supplying fake implementations for serial interactions. The features of the driver can be verified by examining the interactions with the mocked serial device. Technical details of the operation of the Storage library are provided in Appendix A.7.

3.2.6. GPS

An onboard GPS is a crucial component of a space mission for orbit determination and time synchronisation. Like other external peripherals, however, the GPS modules integrated into a spacecraft design vary depending on the manufacturer. The GPS library in the BSF was written to abstract this interaction, adding support for the common challenge of rapid timelines faced in CubeSat design, allowing for the possibility of changing GPS hardware on future Binar missions without impacting the timeline. Figure 3.11 provides a UML representation of the module.

Although inheriting from the GPSInterface class, the GPSBase class does not override the methods specific to a hardware implementation. Instead, it follows the Open-Closed principle by extending the interface class with the methods required for interaction with a GPS that communicates using NMEA sentences. This design choice decouples the interface object from depending on an NMEA device, supporting module reusability by being more generic. The technical information regarding how the GPSBase object extracts the location and time information is provided in Appendix A.8. The GPS subclass then further exemplifies the Open Closed principle by deriving from the GPSBase and extending its functionality by implementing the virtual methods defined in the GPSInterface object to interact with hardware. This approach promotes reusability with new GPS hardware by extending the GPSBase superclass to provide an implementation for reading from the updated hardware. Secondly, this design choice aids in testability. The GPS library can be tested by extending the base class with a mocked hardware implementation. Keeping the GPSBase class closed for modification contributes to robust software design. It removes the possibility of introducing bugs when modifying existing working code and prevents breaking the existing functionality of modules that depend on it.

3.2.7. I2C TO UART

During Binar-1 development, the intention for communications with the transceiver was to use the USART serial. Two transceiver versions were purchased; one for the

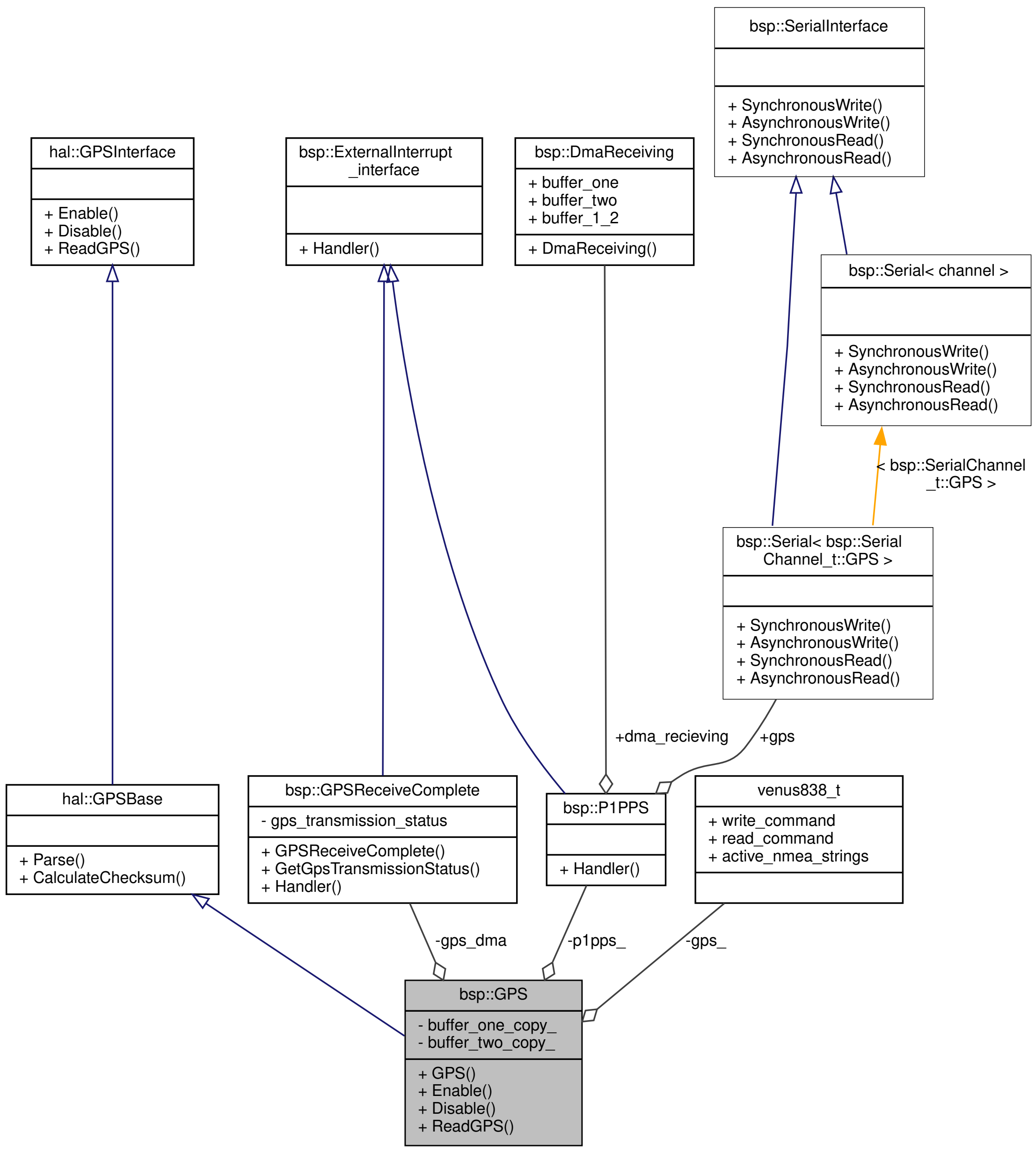


FIGURE 3.11: A UML representation of the GPS implementation in the Binar-1 flight software.

flight model and another for the engineering model. The flight model transceiver was reserved while assembly and testing of the engineering model took place. When the flight model transceiver was eventually opened, it became evident that whilst the engineering model transceiver spoke using UART, the flight model transceiver spoke using I2C. Because of this, a last-minute communication protocol adaptor board was assembled featuring an I2C to UART conversion IC. In support of this, the I2C to UART library (Figure 3.12) was created to facilitate communications with the transceiver.

The I2CToUART object follows the Open Closed principle by extending on the I2CToUARTInterface object through inheritance. Following the Dependency Inversion principle, the public-facing I2CToUART object interacts with the Binar-1 hardware through an implementation abstraction. Calls to the public-facing methods delegate to the implementation object, I2CToUARTImpl, using the private methods that extend the interface. The I2CToUARTImpl object likewise follows the Open Closed principle by extending the shared interface class, overriding the pure virtual methods and adding the required implementation-specific functionality.

Adherence to the Open Closed principle promotes reusability by decoupling the high-level application code from the hardware specifics. The extended implementation object adds device-specific implementations for configuring the selected I2C to UART hardware, for example, through the ”SetBaudRate” method. Moreover, similar to other external hardware peripherals, a custom low-level driver was written to support the functionality of the IC. This driver, developed following the device datasheet, is aggregated into the I2CToUARTImpl object through the private member driver_. As the high-level I2CToUART class depends on an abstraction, it is not coupled to this specific driver and hence supports substitution with new hardware if necessary. Moreover, it allows a mocked device to be substituted during software testing without impacting the high-level code.

Whilst the public-facing I2CToUART module was designed to support changing the specific IC used, the IC driver was also developed to support a change of flight computer. The custom driver provides most of the functionality required for interaction with the device but leaves features for serial interactions to be implemented in the application code and provided to the library through the public function pointer

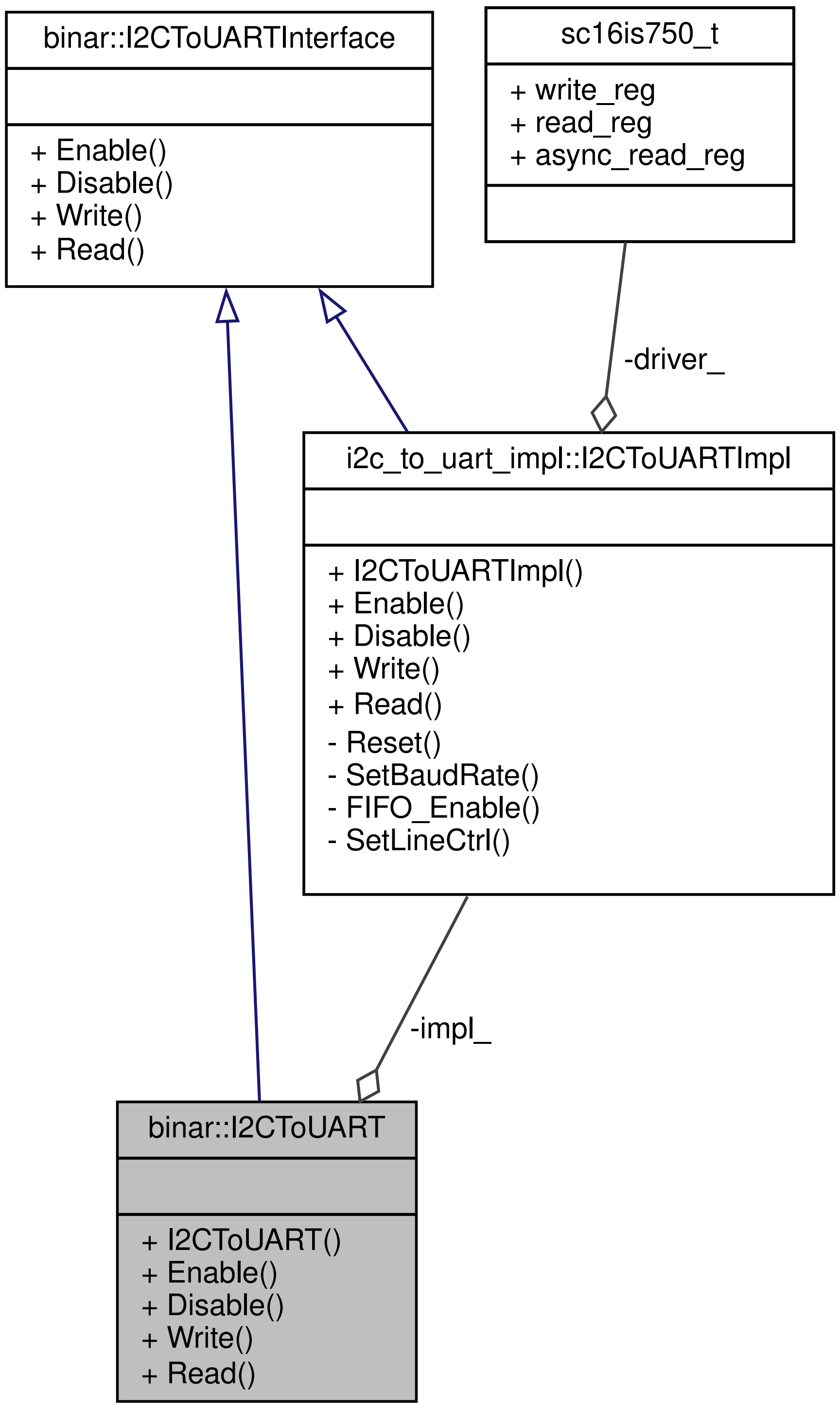


FIGURE 3.12: A UML representation of the I2C to UART converter object on Binar-1.

members in the sc16is750_t object. This approach promotes library reusability by allowing for serial interactions to be developed for a new flight computer whilst still porting most of the library code across to the new project with the relevant tests. Technical details of the I2C to UART library are provided in Appendix A.9.

3.2.8. REAL TIME OPERATING SYSTEM

Whilst offering a significant reduction in power, cost and complexity over conventional processors, microcontrollers often only have a single execution thread for applications. The lack of multiple threads can introduce complexity when multiple concurrent processes are required, such as listening for radio transmissions from a ground station whilst executing tasks, or when periodic functionality is required, such as a regular beacon transmission from the satellite. A common solution for these requirements is implementing a Real-Time Operating System (RTOS) kernel on the microcontroller to leverage the scheduler.

Implementing an RTOS in the BSF is developed according to the Open Closed principle, as seen in the UML diagram in Figure 3.13. The RTOS object extends the RTOSInterface class to provide implementations for the abstract methods in the interface. Moreover, it adds another public method, ”Initialise”, to benefit the application code.

The RTOSImpl implementation object further exemplifies the Open Closed principle, extending the interface class to support a hardware implementation. If the RTOSInterface was modified to add the implementation-specific features, it would couple the public-facing RTOS module to the FreeRTOS framework used on Binar-1. This approach would make it difficult to break from it if it no longer met mission requirements.

Adherence to the Dependency Inversion principle also provides support for changing the RTOS used. As the public-facing RTOS object interacts with the underlying RTOS implementation through the abstracted RTOSImpl object, the application code will be unaffected by a change in the RTOS used.

Therefore, for this module, adherence to the Open Closed principle means the

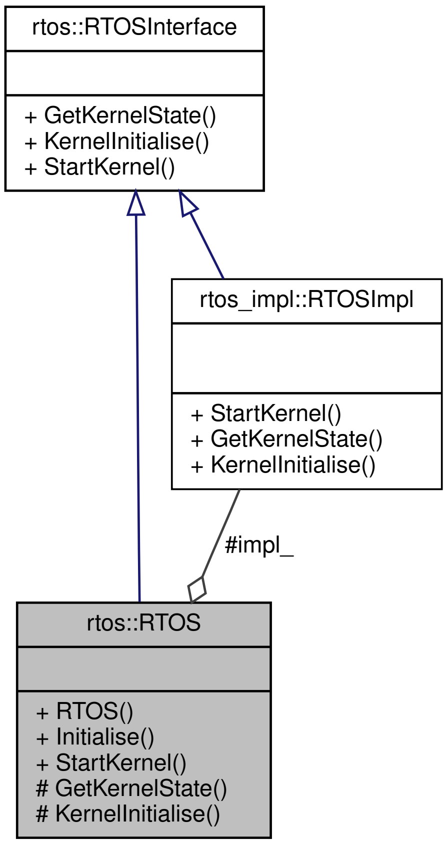


FIGURE 3.13: A UML representation of the RTOS class in the RTOS library. The RTOS object aggregates an implementation object derived from the same interface.

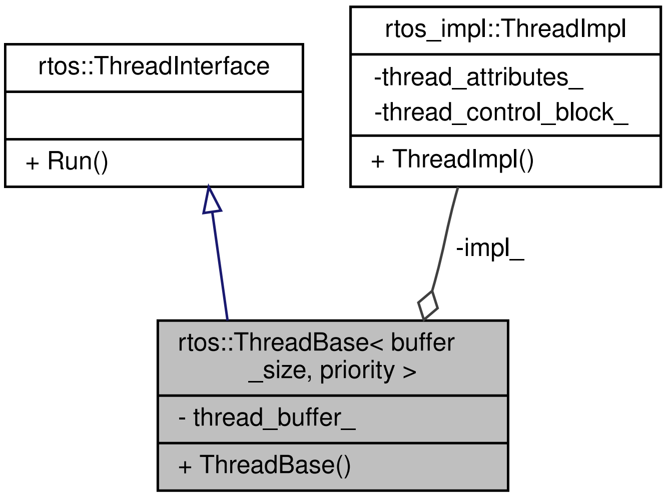


FIGURE 3.14: A UML depiction of the ThreadBase object in the RTOS library. The ThreadBase object obfuscates the FreeRTOS implementation, providing all the configuration required to initialise a thread with the RTOS. To use in application code, the user must derive a class from the ThreadBase, providing an implementation for the pure virtual "Run" method inherited from the ThreadInterface.

RTOSInterface can be extended to support any RTOS required. The Dependency Inversion principle then allows the public-facing RTOS object to interact with any supported subclass of RTOSInterface through an abstraction.

Threads can be created with their own runtime scope for the RTOS kernel to schedule when it is initialised. Threads are created using the ThreadBase object shown in Figure 3.14. This object follows the Open Closed principle by extending the ThreadInterface object to add generic functionality required by threads. With this generic functionality implemented, the module is closed for further modification. However, it does not override the "Run" method and is hence open for extension. In application code, the ThreadBase object must be extended by a thread implementation overriding the "Run" method. This approach promotes reusability by forcing thread implementations to be application specific, removing the possibility of coupling unnecessary dependencies into the Thread object. Moreover, being a critical component of application code runtime, removing the need to modify the existing well-tested code prevents failure-inducing bugs from being introduced into the code base.

The threads used in the Binar-1 application code were for communications with the ground, running scheduled tasks, and transmitting a periodic beacon. Technical information providing an overview of how threads are created and managed in the

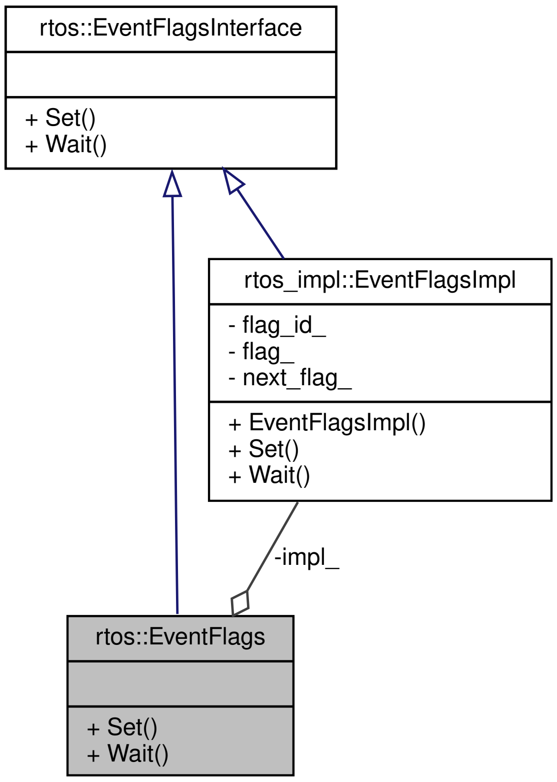


FIGURE 3.15: A UML depiction of the EventFlags object from the RTOS library. The EventFlags object aggregates an implementation object derived from the same interface. This object is used for synchronising threads.

library is provided in Appendix A.10.

The final main component of the RTOS library is the EventFlags object. Event flags are a method of controlling a thread's execution from a separate thread. The implementation of the EventFlag object in the BSF can be seen in Figure 3.15.

This library section also leverages the Open Closed and Dependency Inversion principles. The EventFlagsImpl object extends the EventFlagsInterface object to provide RTOS-specific event flags implementations. In the case of the software written for Binar-1, this decouples the EventFlags module from the underlying FreeRTOS. The public-facing EventFlags object then interacts with the underlying RTOS through the EventFlagsImpl abstraction, supporting changing the RTOS used by allowing for an alternative implementation of EventFlagsImpl to be controlled without affecting high-level code.

A further benefit of the extensive use of the Open Closed principle and the Depen-

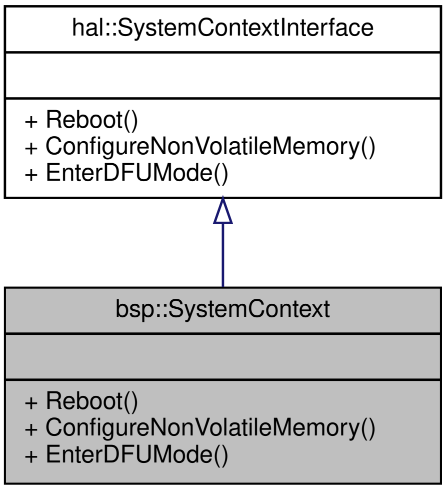

hal::SystemContextInterface
+ Reboot()
+ ConfigureNonVolatileMemory()
+ EnterDFUMode()
bsp::SystemContext
+ Reboot()
+ ConfigureNonVolatileMemory()
+ EnterDFUMode()

FIGURE 3.16: A UML diagram of the SystemContext object from the system context library. This object abstracts processor-specific functionality required by the application code.

dency Inversion principle in this library is in software testing. The interface objects can be extended to create fake RTOS backends controlled through the same abstraction without impacting the application code. This approach allows for granular control over the functionality of the RTOS fake to verify that the software interacts with it as expected, increasing reliability.

3.2.9. SYSTEM CONTEXT

Regardless of the processor used for a spacecraft flight computer, certain generic features, such as a system reboot, are always required. However, whilst these features are common across processors, the implementation is not. Hence, the System Context library is used in the BSF to abstract these features from the high-level software that requires its use. A UML depiction of the SystemContext object from this library is shown in Figure 3.16.

This module is built according to the Open Closed principle, requiring the specific implementation of the SystemContext module to extend the SystemContextInterface. Enforcing the module to be closed for modification but open for extension supports code reusability. Alongside the extending class providing device-specific implementations of the generic functionality, extending the interface also allows for adding processor-specific functionality if required. This means the library can be ported to a new processor without requiring unsupported features to be carried across.

Furthermore, this level of abstraction aids testing by allowing mock extensions of the library to be substituted in place of the hardware implementations when they are used polymorphically in the BSF. Technical specifications of the library's functionality are provided in Appendix A.11.

3.2.10. SYSTEM LOGGER

In orbit, the best-case scenario pass length over the Curtin University ground station for Binar-1 was less than ten minutes. This meant requesting housekeeping data from the spacecraft was limited to a small subset of the orbital period. Collecting housekeeping data over multiple orbits, however, was desired for the Binar Space Program to understand how the developed custom CubeSat platform operated in space. This data would directly influence the decisions made during the design of future spacecraft. Hence, data had to be collected autonomously when not in range of a ground station and saved in the onboard storage to be downlinked when the satellite passed overhead. The Logger object from the system logger library (Figure 3.17) is responsible for collecting this data and preparing its return.

The library follows the Single Responsibility principle by requiring a single object per system to be logged. This approach supports modularity, as changes to one system will not affect how another is logged. This design choice makes the library easily extensible as new system logger objects can be created, responsible for their own specific logging, and aggregated into the public-facing Logger object.

In the BSF, these logging implementation objects provide a means of abstraction over the system that is being logged. The Logger object follows the Dependency Inversion principle by interacting with the hardware systems logged through the aggregated abstractions. This approach supports the reusability of the library on other flight computers as the Logger object is decoupled from how information is collected and logged. Moreover, it enables testing by allowing mocked implementations to be

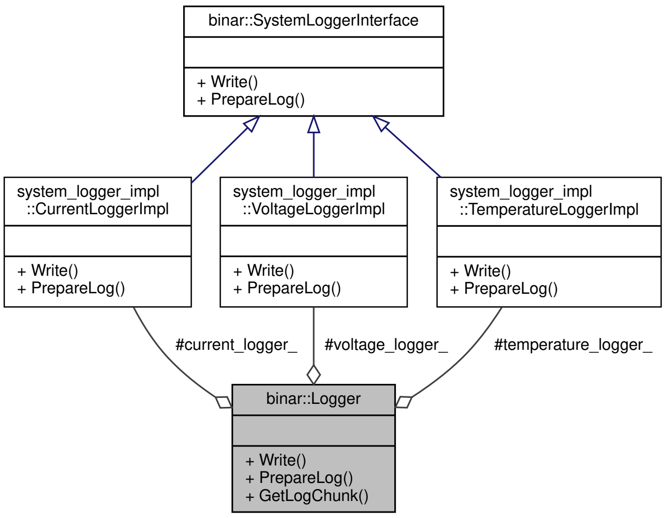


FIGURE 3.17: A UML diagram of the Logger object from the system logger library. This object periodically logs housekeeping data from the aggregated members to the onboard storage.

substituted in place of a hardware dependency without affecting high-level code.

Furthermore, the library adheres to the Open Closed principle by requiring the system logging objects to extend a closed common interface class for modification. The opposite approach of adding the support of new logging systems by modifying the SystemsLoggerInterface class could introduce failure-inducing bugs into all subclasses inherited from it. Furthermore, it would reduce the reusability of the library as the superclass would then be coupled to specific systems by which to log. These systems may not be required in future software that intends to reuse the library. Technical details of the System Logger library are provided in Appendix A.12.

3.2.11. MANIPULATOR

As part of the Binar Space Program's goal of collaborating with local industry, a simulated robotic manipulator payload was included on Binar-1. The payload aimed to provide a test bed for Fugro SpAARC to develop remote operation capabilities. The case study created for the robotic arm payload was to fix a simulated faulty solar panel deployment. With the simulated panel failing at a random deployment angle, the robotic arm simulation was to be controlled from Fugro SpAARC to correct it.

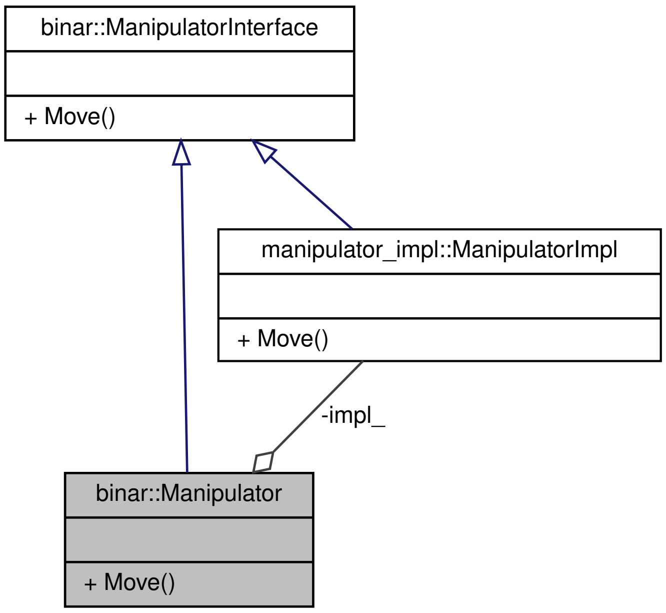

binar::ManipulatorInterface
+ Move()
manipulator_impl::ManipulatorImpl
+ Move()
-impl_
binar::Manipulator
+ Move()

FIGURE 3.18: A UML representation of the Manipulator object on Binar-1.

The Manipulator library (Figure 3.18) provides access to controlling the emulated payload. The library was developed in adherence to the Open Closed principle and the Dependency Inversion principle. The public-facing Manipulator object interacts with the payload through an abstraction that extends the ManipulatorInterface object. Having the ManipulatorInterface closed for modification but open for extension encourages implementations of the manipulator payload to provide the application-specific features without polluting the interface. This approach was made in preparation for the library to be reused on a future mission where physical hardware is to be controlled by the library. The interface class can be extended to support features specific to hardware. If the Open Closed principle were not adhered to in this library, these extra features would become coupled to the emulated implementation that would not require them. This approach would make the library more difficult to maintain and error-prone. As adherence to the Dependency Inversion principle abstracts the high-level manipulator object from the implementation it controls, the library can be reused to control the future hardware implementation of the manipulator without impacting application code.

For the implementation on Binar-1, tasks scheduled to use the manipulator payload provide a joint angle by which to move the manipulator. The ManipulatorImpl object

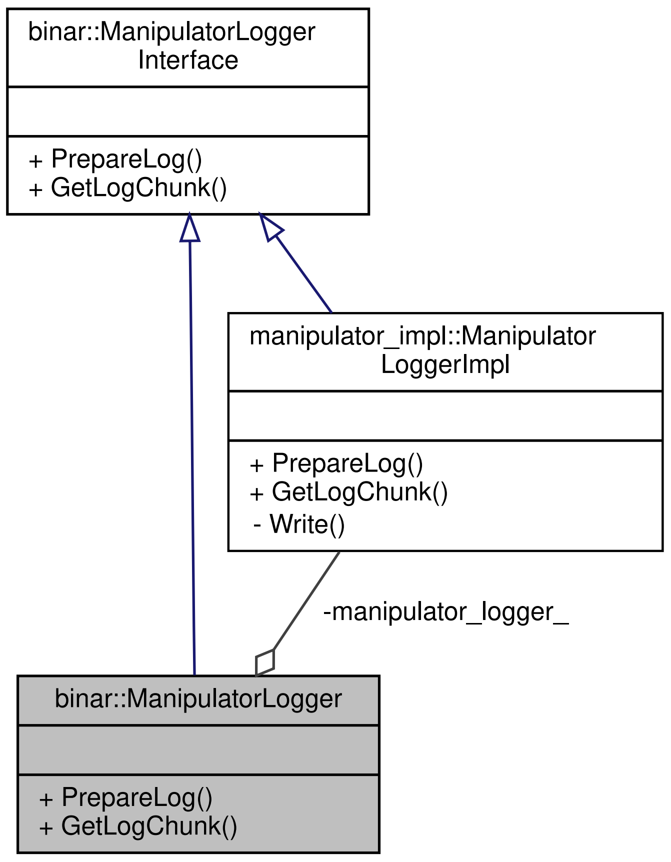


FIGURE 3.19: A UML representation of the ManipulatorLogger object on Binar-1.

then stores this angle for later retrieval by a ground operator through the Manipulator-Logger object (Figure 3.19). Appendix A.13 provides information regarding how the log is retrieved. This section of the library follows the Single Responsibility principle by decoupling the emulator implementation of the manipulator from how the angle information is logged. This means that the logging implementation can be updated without changing the emulator.

The ManipulatorLogger is also designed following the Open Closed principle and the Dependency Inversion principle. The ManipulatorLogger object interacts with the application-specific logging implementation through an abstraction that extends the ManipulatorLoggerInterface object. This approach provides similar benefits to the design of the Manipulator object, allowing for abstracted control over a logging implementation that is free of interface pollution. This approach enables the Manipula-torLogger to be reused with an alternative logging solution by extending the interface to provide the required functionality while allowing the high-level ManipulatorLogger object to control it through an abstraction.

Adherence to this library's various object-oriented design principles aids in software testing. During integration testing, the emulator implementation of Manipulator can be tested to work correctly with the ManipulatorLogger class. By extending the ManipulatorLoggerInterface to provide a mocked implementation of the logger and having the high-level ManipulatorLogger control it through an abstraction, the module's behaviour can be easily examined.

3.3. UTILITIES

Utility libraries are an essential component of any software framework, as they provide generic functionality that can be used across multiple modules in the codebase. Given the unique challenges and constraints of systems operating on orbit, ensuring that these modules were built on a foundation of robust and reliable software practices was crucial. This section details two of the most important utility libraries developed and how they improve the overall functionality and performance of the BSF.

3.3.1. RESULT AND VISIT

A good approach for writing software is acknowledging that errors can happen during runtime and providing a safe path for application code to progress down when they do. The Result and Visit library in the BSF represents an innovative solution to this in a reliable and safe manner. As a preventative measure to reduce runtime bugs as much as possible, the burden of enforcing safe error handling was shifted from the user's to the compiler's responsibility. This is implemented by having the return types to most methods and functions take the form of a variant from the standard C++ library. A variant is a class template representing a type-safe union, meaning the variant can only explicitly be one of the types of the template parameters. Creating a custom type Result object that masks an underlying variant of either a success or a fail type deduced from template parameters allows for a return type of either one of two custom values. This deduction is made statically as the variant requires the memory allocation of the largest of the two options. This approach enforces the application code to exhaustively extract the return type from the variant result type before it can be used. The type safety of C++ will prevent this step from being skipped in application code by causing compilation errors. This type safety enforces the return type to be either the type expected from a function call or a custom failure object that needs to be explicitly handled before the software can compile. Moving this error handling to compile time decreases the chance that software running on the satellite can encounter an unexpected event during runtime. C++ variants are a relatively recent addition to the C++ standard library. At the time of writing, there is no literature detailing their use for enforcing compile-time error handling. Hence, this approach represents a unique solution to error handling in space missions.

Extracting the result object from the custom Result variant is done through the non-member functor "Visit" associated with the C++ variant library. A custom wrapper over the "Visit" functor increases readability by accepting a template parameter for the variant type and then a parameter pack of lambdas that match individual variant types. The visit then behaves similarly to a switch statement and executes the code inside the lambda corresponding to the type encoded in the result variant. An added benefit of this approach is controlling scope. To use variables from a higher scope in a lambda, they must explicitly be captured into the lambda. Otherwise, the compiler will throw errors, preventing accidental reassignment of variables. An example is provided in Listing 3.1 attempting to read from the GPS peripheral.

LISTING 3.1: The visit syntax used throughout BSF applied to reading the GPS.

```
util::visit(
    gps.ReadGPS(), // Call the method
    [](hal::GPSReadings gps_readings) {
        // Use GPS readings object in this scope
    },
    [&error_logger](util::Fail) {
        // Handle fail in this scope
    });
```

In Listing 3.1, it can be seen that when the ”ReadGPS” method is called on the GPS object (not shown constructed in the example), the return type is either a GPSReadings object in the HAL namespace or the Fail object in the util namespace. Furthermore, in this example’s secondary lambda for the Fail variant, it can be seen that an object named error logger is being captured by reference and thus will be accessible within the failure scope. This approach requires an exhaustive list of all the types encoded within the variant to be addressed. Otherwise, an error will be thrown at compile time. This approach is especially useful when combined with creating custom libraries. It allows all libraries running on the spacecraft to propagate errors up to user code rather than being obfuscated and handled within the library. Handling errors this way is important as the calling code generally has more information on how the error should be handled than what may be available in the scope of the library. This increases code portability and facilitates library reuse for future spacecraft.

3.3.2. STATIC LINKED LIST

In application code, situations arise where elements need to be grouped as a linear sequence. A common way to implement this is using a container type such as an array. Whilst efficient for accessing data elements with contiguous memory, a challenge arises when attempting to sort the elements in the data sequence. Attempting to insert data is a slow operation that requires the elements at each memory address to be shifted along in the series to create space for the new element, as shown in Figure 3.20.

This operation requires free memory in the container to shift existing entries into to make space for the new addition. When developing for an architecture with no restrictions on heap memory usage, new memory can be created at application run time to resize the container to fit the new element. However, on an embedded system where memory is a finite resource, abstaining from using dynamic memory where possible can aid in preventing runtime bugs. Hence, this container can be alternatively implemented by preallocating memory at compile time for a maximum required size, allowing software interaction that simulates dynamic behaviour. The routine can gracefully handle errors if memory usage exceeds the predefined allocation. For

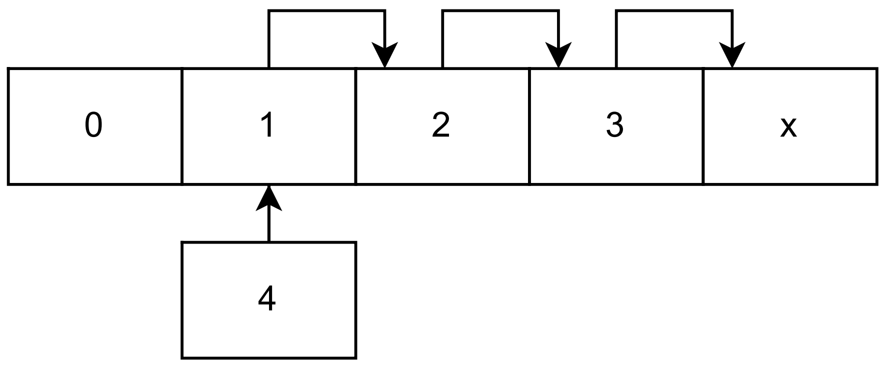


FIGURE 3.20: To insert the fourth element into an array at the first data entry, each element must be moved along to the subsequent memory address, giving this operation a complexity of O(n). For this operation to work, free space is required in the array to shift into, denoted by an X. In an application with plentiful heap memory, new memory can be created at run time to resize the container. However, on an embedded system, static memory allocation is preferred.

middle insertion, as all elements in the container need to be moved, the complexity for this operation is O(n), where n refers to the number of elements that need to be shifted. Hence, with a small array, the complexity is minimal. However, it is not a scalable solution as larger arrays can start to introduce significant complexity in operations. When random element access is less of a priority than sorting, a linked list structure provides an alternative to containers. A linked list is a method of storing a series of data elements, the order of which is not defined by the location of the element in memory. Instead, a pointer to the next node in the list is included alongside each entry in the structure. Due to this, the structure simplifies adding and removing elements at any point in the data series, as demonstrated in Figure 3.21.

The simplicity of sorting the entries in the structure highlights its applicability for a list of tasks to execute on the spacecraft, arranged in order of task execution time. When the ground station requests to schedule a task, the link between two tasks can be broken and the new task inserted. Whilst offering the advantage of simplified data restructuring, linked lists suffer from traditionally requiring the use of heap memory for runtime memory allocation to create new nodes to add to the list. A static memory linked list library was created for the BSF to access the benefits of element sorting in a linked list without the requirement of dynamic memory. This library demonstrates an innovative solution to the challenge of limited resources on

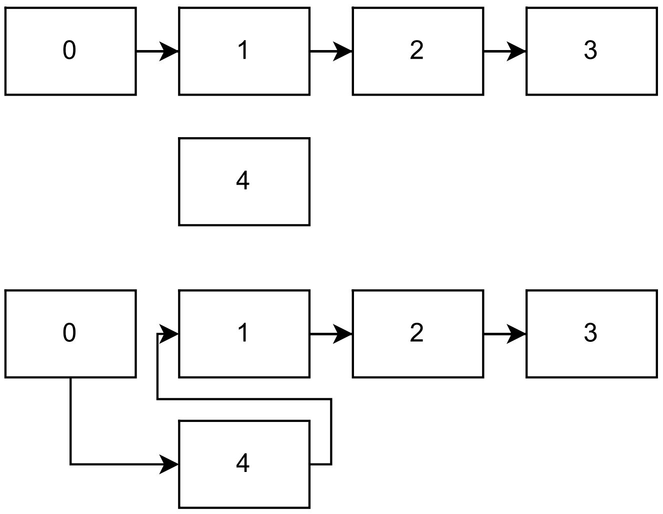


FIGURE 3.21: To insert the fourth element into a linked list at the first data entry, the pointer between the zero element and the first element must be updated to point to the fourth element and the fourth element to the first. This operation does not require any other entries to be updated and hence has a complexity of O(1).

CubeSat platforms. Due to the memory constraints and processing capability of the microprocessors powering the small spacecraft, efficient software design maximising resource utilisation is paramount. In this library, all the memory available to the linked list is allocated at runtime and is a private member aggregated by the StaticList object (Figure 3.22). The BSF implements this using C++ templating. When the user constructs an empty static list in application code, the datatype to be stored in the list and the maximum list size must be provided as template parameters. In this application, templating achieves compile-time polymorphism. Rather than hard-coding the datatype and list size, it is kept generic. This approach supports library reuse in the future.

Implementing the static list provides application code with an API reflective of interfacing with a traditional linked list without the requirement for dynamic memory allocation. The template parameters used during construction are shared with the private aggregated memory pool array, memory pool node pointer, and linked list objects. The memory pool array, memory_pool_, contains the memory addresses for each node in the list. The memory pool node pointer is used internally to the StaticList object for traversing the list. The linked list object, linked_list_, is used for pointing to the head and tail of the linked list. It aggregates two public memory pool node pointers,

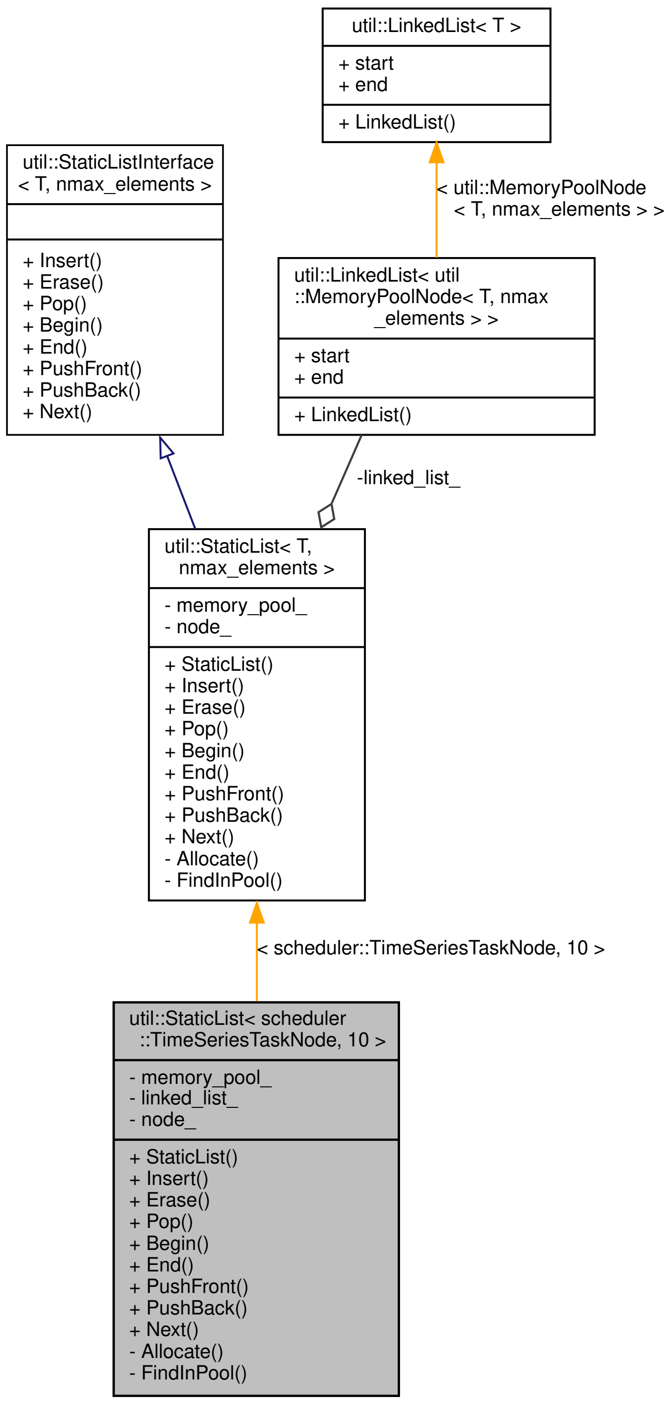


FIGURE 3.22: A UML representation of the StaticList object. The object provides the API of a traditional linked list without the dynamic memory requirements. Constructing the StaticList object requires storing the datatype in the list and the maximum number of elements as template parameters. These parameters are shared with the aggregated members. The memory for the list exists as a private member, memory_pool_, aggregated from an array of MemoryPoolNodes.

start and end, to provide list boundaries during traversal.

The StaticList object follows the Open-Closed principle by having an interface class closed for modification. An implementation of the interface class is provided by extending the interface object. The extended object adds specific functionality for the required linked list implementation, adding private methods such as "Allocate" and "FindInPool", as well as required internal variables. The benefit of this approach is that it makes the static list library extensible and maintainable over time. For code reuse in the future, a new implementation object that extends the same interface class can be added to support a new list implementation without having to modify the existing code. The alternative approach in this situation would see the StaticListInterface modified to supply the functionality required for the linked list, coupling the library to this implementation and making it less reusable. Furthermore, if future code reusing this library had to modify existing code to add implementation-specific details, it would affect everything that depended on it, potentially breaking code that had worked previously. By keeping the interface class stable and separating the implementation details into a separate object, the codebase can easily be extended as requirements change.

The LinkedList object used by the StaticList is instantiated from a generic LinkedList template, denoted by the yellow arrow in Figure 3.22. For the StaticList object, it is instantiated with the MemoryPoolNode object, represented in UML in Figure 3.23.

The MemoryPoolNode object follows the Open-Closed principle through inheritance. The MemoryPoolNodeInterface object provides an abstract interface that must be extended to provide an implementation. The MemoryPoolNodeBase class adds the generic functionality of tracking the usage of the node. This base class is then further extended into the MemoryPoolNode object, which adds the specific functionality for interfacing with underlying data without modifying the base object. Developing this library section in adherence to the Open-Closed principle increases its reusability. Although developed in this case for a linked list, the MemoryPoolNode should be able to be used with any other data structure. Hence, the MemoryPoolNodeBase is extended to provide the implementations for the linked list structure. The alternative, more rigid approach would be to modify the MemoryPoolNodeBase class to provide

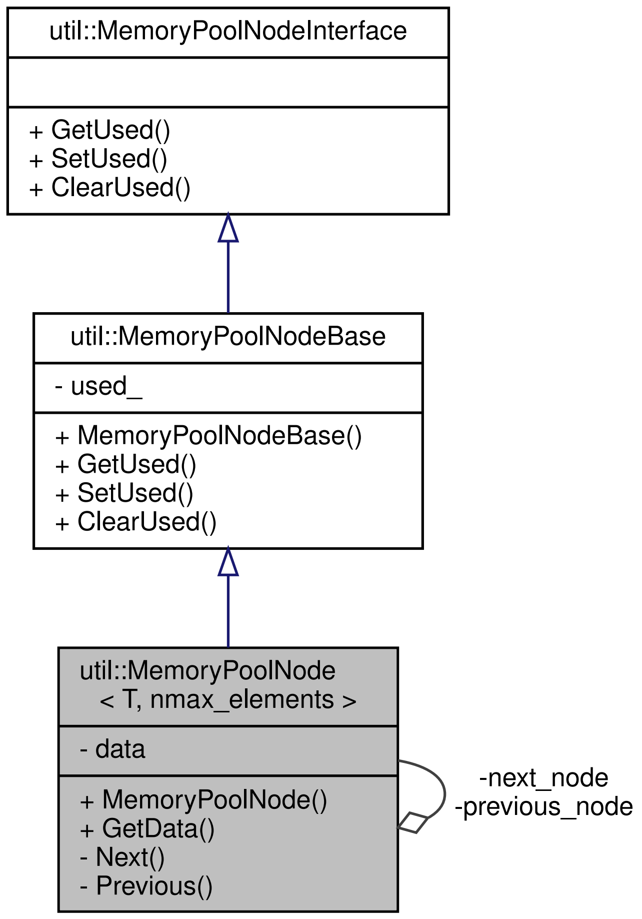


FIGURE 3.23: A UML depiction of the MemoryPoolNode object. This object derives and extends the MemoryPoolNodeBase object, adding template parameters for the datatype and the maximum number of elements.

the implementation-specific features, such as the next and previous pointers. However, this approach is not generic and would couple the MemoryPoolNode object to specific implementations.

The static list library follows the Single Responsibility principle by having the StaticList object not coupled to handling the datatype that is stored within the list. Instead, the responsibility for this is offloaded to the MemoryPoolNode object. Offloading the responsibility of handling the datatype stored within the list to the MemoryPoolNode object increases the maintainability and flexibility of the static list library by making it more modular and hence easy to modify. If the library was to be reused, the datatype stored within the list or how it is handled can be modified through the MemoryPoolNode object without needing to modify the entire static list library, reducing the likelihood of introducing software flaws. Moreover, the MemoryPoolNode object can be reused in other areas of the code base where a memory pool may be required. Technical details of the static list library are provided in Appendix A.14.

3.4. BOOT APPLICATION MODULES

The boot application modules are a collection of libraries used during the Start-Up/Bootloader application, entered during spacecraft boot. The Start-Up/Bootloader application is a critical component of a space mission, responsible for initialising the onboard systems during the boot process. Due to the importance of the success of this process, it is essential to ensure that the boot application modules are designed to be reliable. As such, the modules discussed in this section were custom-developed to address specific needs related to booting up the spacecraft in a reliable manner.

3.4.1. BOOTLOADER

A bootloader is a piece of firmware executed upon system boot or restart. The primary purpose of a bootloader is to load the application software to be executed into memory. In an embedded system, this application code commonly exists internally in the processor in a reserved flash region. As Binar-1 required dual-booting for robust

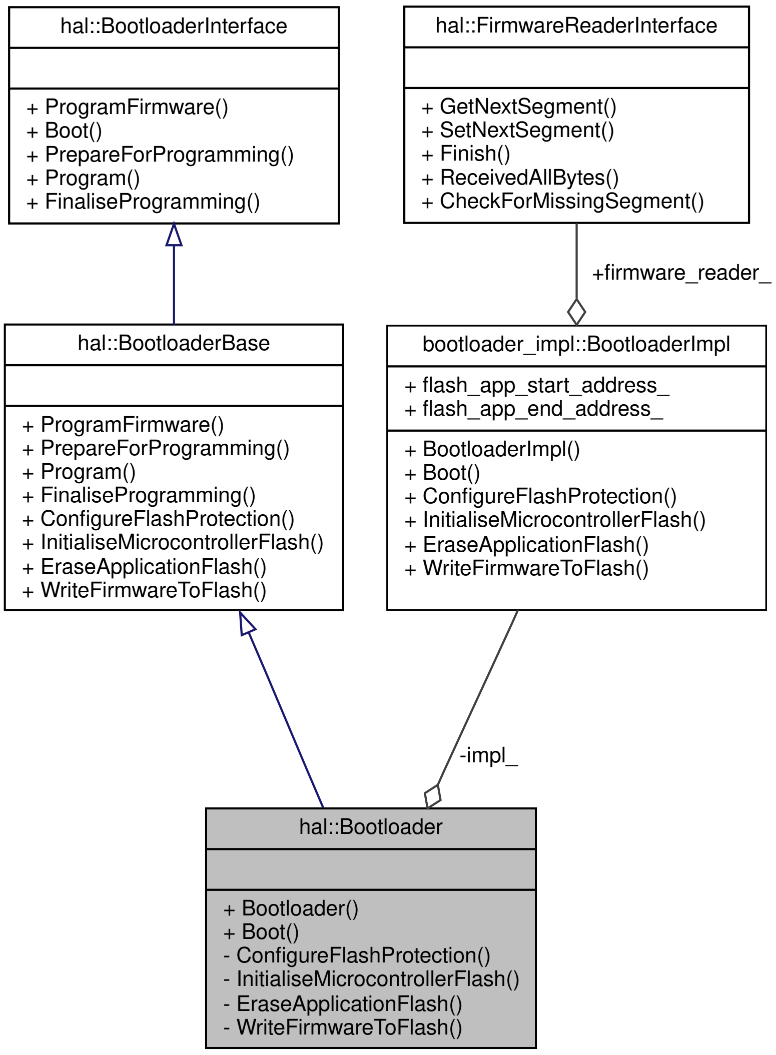

hal::BootloaderInterface
+ ProgramFirmware()
+ Boot()
+ PrepareForProgramming()
+ Program()
+ FinaliseProgramming()
hal::FirmwareReaderInterface
+ GetNextSegment()
+ SetNextSegment()
+ Finish()
+ ReceivedAllBytes()
+ CheckForMissingSegment()
+firmware_reader_
hal::BootloaderBase
+ ProgramFirmware()
+ PrepareForProgramming()
+ Program()
+ FinaliseProgramming()
+ ConfigureFlashProtection()
+ InitialiseMicrocontrollerFlash()
+ EraseApplicationFlash()
+ WriteFirmwareToFlash()
bootloader_impl::BootloaderImpl
+ flash_app_start_address_
+ flash_app_end_address_
+ BootloaderImpl()
+ Boot()
+ ConfigureFlashProtection()
+ InitialiseMicrocontrollerFlash()
+ EraseApplicationFlash()
+ WriteFirmwareToFlash()
-impl_
hal::Bootloader
+ Bootloader()
+ Boot()
- ConfigureFlashProtection()
- InitialiseMicrocontrollerFlash()
- EraseApplicationFlash()
- WriteFirmwareToFlash()

FIGURE 3.24: A UML representation of the bootloader object in the Binar-1 flight software.

firmware upgrades, the bootloader was purposed with selecting which application to boot. Alongside loading application software into memory, a bootloader is commonly used for upgrading system firmware by loading it from an external source. These features are implemented in the Binar-1 flight software using the Bootloader object (Figure 3.24).

The bootloader library adheres to the Open-Closed principle by requiring modules to be extended to provide the required functionality. The BootloaderBase class, extending the interface, provides a blueprint of the generic functionality any bootloader would share, refraining from any implementation specifics. Implementations are provided for ”PrepareForProgramming”, ”ProgramFirmware”, ”Program”, and ”FinaliseProgramming”. The device-specific ”Boot” method is left as pure virtual from the interface. Moreover, the BootloaderBase class extends the BootloaderInterface to add four virtual methods; ”ConfigureFlashProtection”, ”InitialiseMicrocontrollerFlash”, ”EraseApplicationFlash”, and ”WriteFirmwareToFlash”. The implementation of these methods is left to the deriving class. This base is closed for modification. The class is extended into the Bootloader object to provide the implementation-specific features required for Binar-1.

This approach increases reusability by making the bootloader library extensible and maintainable. Keeping the base class closed for modification, requiring objects to extend the base class to provide bootloader implementation specifics, ensures that any changes to the library will not break existing code that depends on it. This approach allows the same base class to be reused for different bootloader implementations while still providing the specific features required by each implementation. By identifying the generic bootloader functionality and implementing it in the base class, any module extending this class will also inherit the implementation. This helps limit redundant code, which could introduce fault-inducing bugs, increasing the reliability of the code base.

The public-facing Bootloader object adheres to the Dependency Inversion principle by interacting with the hardware-specific bootloader implementation through the abstracted object BootloaderImpl. This approach decouples the library from the low-level hardware it interacts with, supporting library reuse with changing mission requirements. This decoupling allows a new hardware-specific implementation to be controlled through the public-facing Bootloader object without requiring a change to the object itself. This capability will be useful if the flight computer is changed in the future or if the firmware is moved to an external location. Moreover, this approach aids with unit testing, as a mocked hardware backend can be substituted in place of the BootloaderImpl, allowing for behavioural testing of the Bootloader object.

The BootloaderImpl object also demonstrates the Dependency Inversion principle by aggregating a pointer to a derived class of the FirmwareReaderImpl from the firmware reader library (Section 3.4.2) for receiving the firmware to write to flash. This pointer is supplied a memory address through dependency injection during object construction. By interacting with the firmware reader implementation through an abstraction, the BootloaderImpl object becomes decoupled from how the firmware is received. This supports reusability as any object extending the FirmwareReaderInterface can be injected into the BootloaderImpl object and controlled without requiring the BootloaderImpl to change. Moreover, this approach enabled unit testing of the library. By extending the FirmwareReaderInterface into a mocked object, the behaviour of the Bootloader object could be tested with fine granularity, leading to a more robust library.

As performing a firmware update can be risky for a spacecraft, robust error handling is built into the library to prevent the spacecraft from trying to boot corrupted firmware. Before starting the firmware writing process, the failed firmware update flag in non-volatile memory (Section 3.2.9) is set. The flag is only reset upon successfully reprogramming the application's flash memory. This flag is checked on boot, meaning the spacecraft will enter Safe Mode if a firmware update previously failed and the failed firmware update flag is not reset.

If the firmware update process is successful, the spacecraft will attempt to boot into the updated firmware using the ”Boot” method on the Bootloader object. However, as a preventative measure against booting into corrupt firmware, the bad app check flag in non-volatile memory is set before booting. An assembly jump instruction is then performed to the base address of the application bank. The bad app check flag is reset in the application before continuing with the application execution. If the application has corrupted during the writing process, the flag remains set, and the watchdog timer triggers. On start-up, this boot bank will then be marked as unsafe to boot, requiring assistance from a ground operator to reprogram.

The bootloader library follows the Single Responsibility principle by separating the concern of the Bootloader object from how firmware is read into the application. The responsibility for reading firmware is delegated to the FirmwareReader object. This approach increases the maintainability, flexibility and reusability of the library. As the Bootloader object is not coupled to how the firmware to program is read into the module, changes to how the firmware is read will not reflect a change for the bootloader library. Moreover, by delegating the responsibility of reading firmware to the FirmwareReader object, this functionality can be reused in other areas of the code base where firmware needs to be read, removing the need for code duplication. This reduces code redundancy and makes the codebase more maintainable. Technical details of the Bootloader library are provided in Appendix A.15.

3.4.2. FIRMWARE READER

The firmware reader library abstracts how firmware binaries are read by the spacecraft. In Binar-1, this library was a core part of the firmware update process. A high-level UML diagram of the FirmwareReader object is provided in Figure 3.25. For the Binar-1 implementation, the FirmwareReader object within the library was used with the Bootloader module to receive firmware to be programmed into memory. This was implemented by receiving the firmware in sections from the ground station. Appendix A.16 provides a detailed overview of this process.

The firmware reader library benefits from modularity, ease of testing, and maintainability by adhering to the Single Responsibility principle. The FirmwareReaderImpl object, the hardware implementation for Binar-1, delegates the responsibilities of storing and reading the firmware segment to individual modules. Moreover, as the hardware implementation on Binar-1 required interaction with the ground station via the RPC network, this responsibility was implemented through the StartUpModeServer. The modular aspect of the firmware reader makes it easy to add support for a new implementation. Rather than requiring the whole firmware reader library to change, updates can be made specifically to how the firmware segment is stored, how it is read, and how it is communicated with the ground. This increases the reusability of the library. By isolating features of the library into various modules, each part can be unit tested independently, increasing the overall robustness of the library. This can make it easier to locate any software bugs that may exist. Coupled with the Dependency Inversion principle, this allows for granular control over testing by mocking some parts of the firmware reader library and substituting them in place of the hardware implementation through dependency injection. This approach improves the library's maintainability by increasing organisation and readability. Each module is responsible for a specific task,

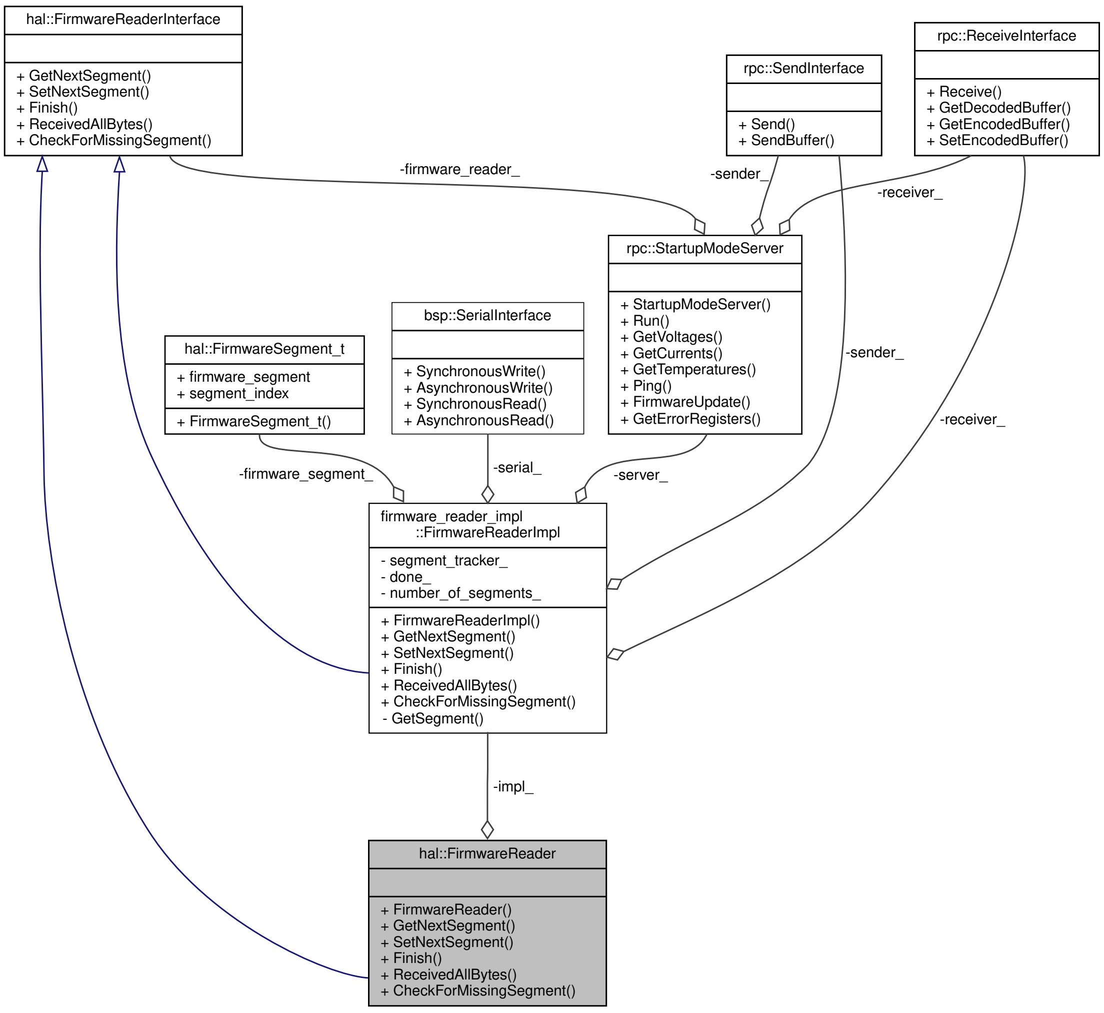


FIGURE 3.25: A UML depiction of the FirmwareReader object in the Firmware Reader library. The FirmwareReader object aggregates an implementation derived from the same interface object. This object is concerned with how firmware chunks are read into application code.

which makes it easier to understand the library's structure, and reduces the likelihood of confusion which could lead to errors being introduced.

The firmware reader library was further built adhering to the Open-Closed principle. The public-facing FirmwareReader object implements the same FirmwareReaderInterface as the privately aggregated FirmwareReaderImpl implementation object. This interface is closed for modification but is open for extension into the deriving objects to provide the implementation-specific functionality of the firmware reader library. Having a stable interface class allows this module to be reused without impacting any existing code that depends on it. Moreover, by decoupling the interface from a specific firmware reading implementation, the new implementations that are added will not be forced to depend on features that are not required. Benefits of this approach have already been seen in the development of Binar 2, 3 and 4. On Binar-1, the firmware reader object reads firmware directly from the ground. However, on the next Binar satellites, the firmware is being read from the onboard storage. By decoupling how the firmware is read from the high-level FirmwareReader module, the majority of the library could be safely reused.

Both interface extensions, FirmwareReader and FirmwareReaderImpl, demonstrate adherence to the Dependency Inversion principle by depending heavily on abstractions. The FirmwareReader interacts with the Binar-1 specific implementation object, FirmwareReaderImpl, through an abstraction set at compile time through the FirmwareReader object's constructor. This approach increases library portability by enabling the FirmwareReader to control any implementation object that extends the common interface. This capability, coupled with the library's adherence to the Open Closed principle, has enabled the reuse of the firmware reader library on the Binar-2, 3 and 4 satellites. The FirmwareReaderImplementation object abstracts how information enters and exits the module by aggregating pointers to the SerialInterface, SendInterface and ReceiveInterface objects. This modularity increases the testability of the module. By deriving mocked objects from the interfaces the FirmwareReaderImpl depends on and substituting them during construction of the FirmwareReaderImpl, it enables the module to be exhaustively tested, increasing the reliability of the firmware reading process.

As memory on the spacecraft flight computer is scarce, the FirmwareReaderImpl object providing the functionality for Binar-1 was set to read the firmware in segments of length up to one kilobyte at a time using the ”GetNextSegment” method. This maximum length is communicated with the ground to ensure the appropriate segment size is sent. The ground station notifies the spacecraft how many segments to expect using the requested segment size. As each segment is received by the spacecraft, the corresponding segment index is marked off in an internal boolean array. This approach is used so the spacecraft can ascertain if any firmware packets have been dropped when the ground station requests to finish the firmware update sequence. Missing segments are then requested individually by the spacecraft. This approach increases the robustness of the firmware update process, reducing the chance of the spacecraft attempting to boot into corrupt firmware.

3.4.3. START-UP

Depending on the launch provider used to get to orbit, a CubeSat may have requirements imposed on its initial operations. For example, in the case of Binar-1, the CubeSat was being deployed from the ISS. During an ISS release, a CubeSat must wait 30 minutes until deployments can occur and nominal operations can be assumed. The Start-up/Bootloader application allows for start-up requirements to be satisfied through the BSF’s Start-up module.

The StartUp module (Figure 3.26) was developed to decouple the start-up functionality for the spacecraft from the Start-Up/Bootloader firmware. It achieves this through adherence to the Single Responsibility and Dependency Inversion principles, promoting the customisability of the Start-up functionality. This approach was decided as missions may have varying requirements during start-up. Hence, the library is designed to be extended to implement start-up requirements for future missions.

Rather than internally containing all the functionality required for satellite initialisation on Binar-1, adherence to the Single Responsibility principle delegates the concerns to several objects, namely the aggregated BootTimer and BootCount implementations for determining when to run the separation timer and a SelfCheck implementation

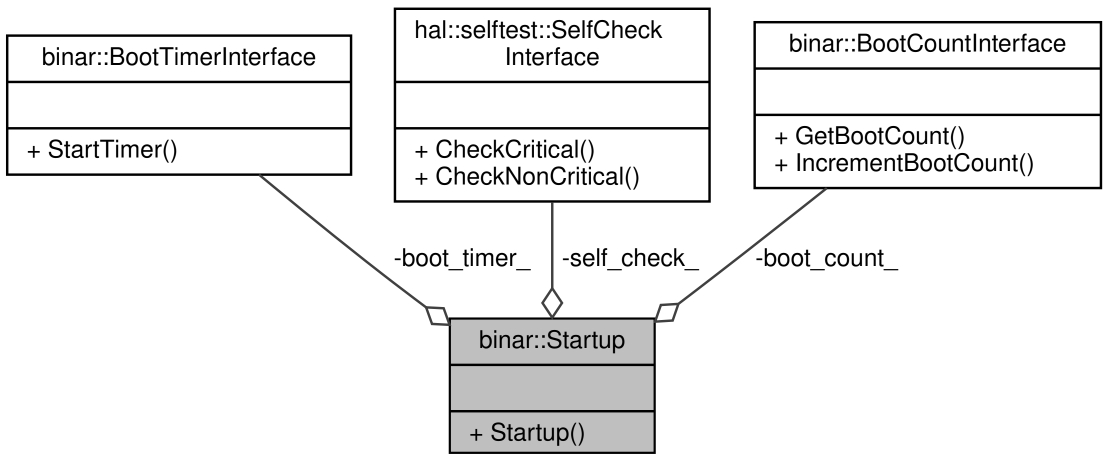


FIGURE 3.26: A UML depiction of the Start-up object from the Start-Up library. The Start-up object is designed to be extensible, aggregating any object required during system start-up.

(Section 3.5.3) to check the satellite's critical systems. This approach gives the library modularity. The benefit of this is that each responsibility can be developed, tested and maintained independently of the other modules required by the StartUp object. As the start-up software is the first section of code executed upon satellite deployment, the increased testability of the module is crucial to verify that the system can boot successfully. Furthermore, changes can be made to any responsibility encapsulating modules without requiring a change in the StartUp object that uses them.

The aggregated modules by the StartUp object are interacted with via pointers to their respective interfaces, demonstrating the Dependency Inversion principle. This approach promotes code reuse, as new mission-specific implementations of the boot count, boot timer and self-check objects can be derived from the respective interface and substituted in the StartUp object without impacting the application code that uses it. Moreover, this helps with integration testing, as hardware dependencies can be broken by substituting a mocked object that extends one of the aggregated interfaces. This allows the behaviour of the high-level StartUp module to be verified, leading to increased reliability of the software. The logic of the start-up software is provided in detail in Appendix A.17.

3.5. APPLICATION MODULES

The application modules are a collection of libraries used during the main Application Mode. This mode is the primary operational state of the spacecraft, responsible for carrying out the mission objectives. Due to this, the software libraries used during this mode must be designed to be reliable. This section provides an overview of the most important application modules developed and how they contribute to the reliability of a space mission.

3.5.1. DETUMBLE

When a satellite is released into orbit, the deployer may induce a spin on the spacecraft. This spin can introduce operational challenges, making establishing communications and using the payload difficult. Hence, the satellite must negate this spin or detumble at the mission's start. The detumble module in the BSF is used by the flight logic to slow the processional rate of the spacecraft on orbit. The implementation for the detumble object in the BSF is represented in UML in Figure 3.27. The Detumble object extends the DetumbleInterface class by adding a private member, stop_condition_dps_, which represents the normalised scalar rotational velocity in degrees per second below which the spacecraft precession is deemed satisfactory.

Having the implementation of detumbling provided by extending the interface class supports decoupling the flight logic from the specific detumble method used. On Binar-1, detumbling was performed by applying the common b-dot control algorithm with the onboard magnetorquers. However, future Binar missions may see more advanced algorithms or alternative attitude control devices. Future functionality can be implemented by extending the interface into a new implementation object without requiring modification to the application code that uses it. Technical details on using the Detumble library on Binar-1 are provided in Appendix A.18.

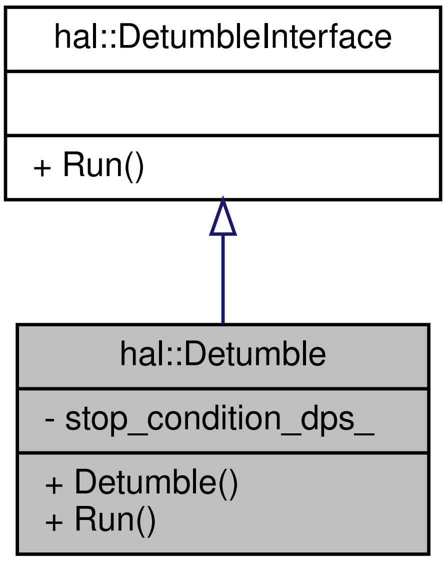


FIGURE 3.27: A UML depiction of the Detumble object in the detumble library. The Detumble object is used by the detumble task to slow the angular precession of the CubeSat.

3.5.2. SCHEDULING SERVICE

As Binar-1 could only be commanded during short pass windows within the line of sight of the Curtin University ground station, the scheduling service library was created to schedule time-based tasks for execution at a later stage on orbit. Figure 3.28 provides a high-level UML representation of the library.

The TimeSeriesTaskScheduler class was developed adhering to the Single Responsibility, Open-Closed and Dependency Inversion principles. During library development, the generic functionality of persistent storage was isolated from the task scheduler. This functionality is instead outlined in the CrudInterface class. CRUD, an acronym for Create, Read, Update, and Delete, provides the minimum functionality required for storing information [135]. The library exemplifies the Open-Closed principle by locking this interface for modification, instead requiring implementations to be provided through extending the class. The extending class passes template parameters through to the interface to determine the datatype stored and a key for data retrieval. This approach allows the interface to remain independent of specific datatypes and their implementations. For the Binar-1 implementation, the TimeSeriesTaskScheduler class inherits and implements the CrudInterface class using the TimeSeriesTaskNode

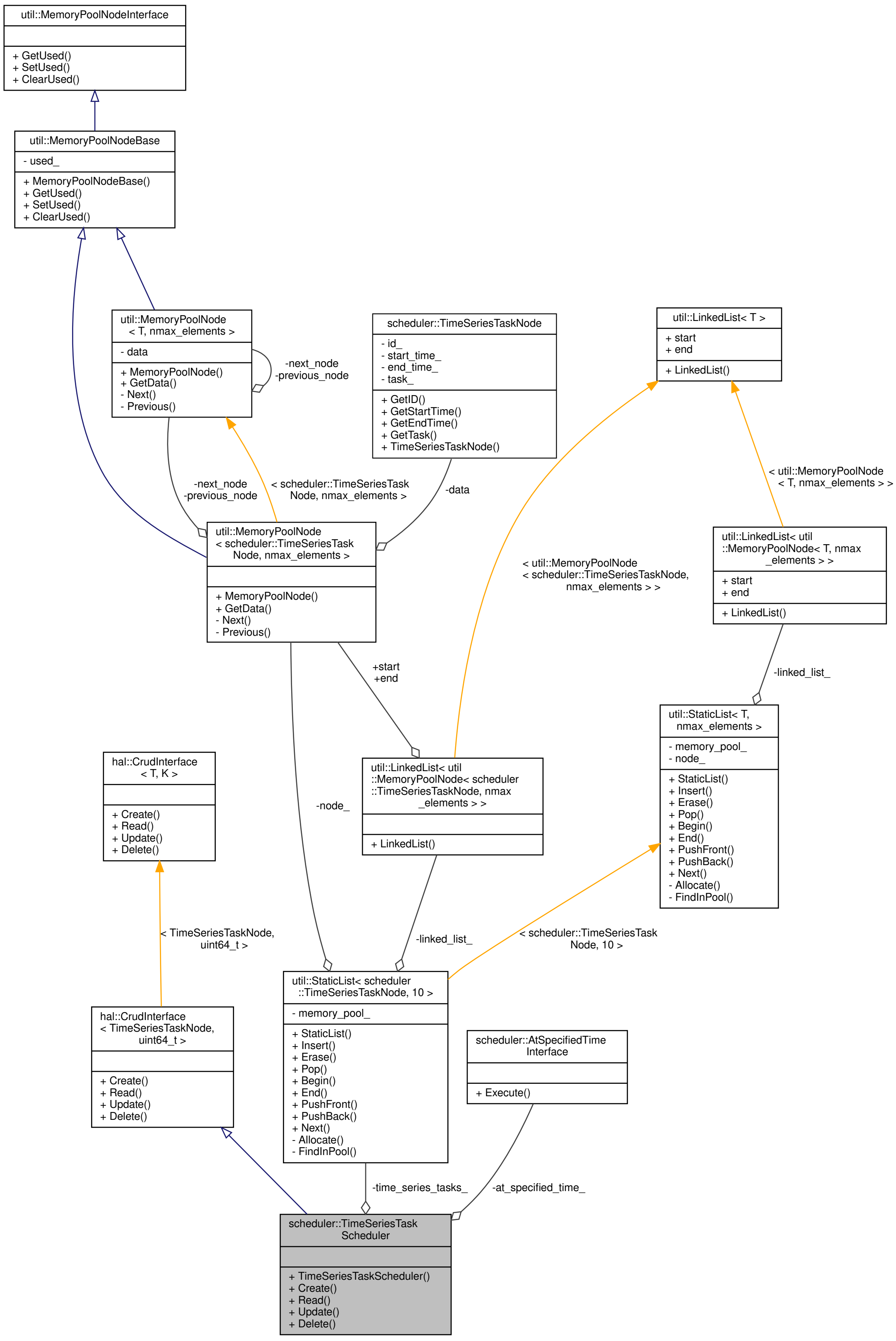


FIGURE 3.28: A UML depiction of the TimeSeriesTaskScheduler object from the Scheduling Service library. This object uses the Static List library to create a linked list of TimeSeriesTaskNodes, sorted by task start time. It is used in application code to store the time-based tasks executed on orbit.

and unsigned 16-bit integer types as template parameters. The first template parameter, TimeSeriesTaskNode, is the datatype to be handled by the interface. The second, the unsigned 16-bit integer, is the datatype used as a key to differentiate between the objects in the storage. The TimeSeriesTaskScheduler object aggregates two private members; a StaticList object, time_series_tasks_, templated with a TimeSeriesTaskNode type, and a pointer to an implementation of an AtSpecifiedTimeInterface, at_specified_time. As the structure of the static list library was discussed in Section 3.3.2, the design approach will not be replicated here. The private aggregated StaticList object holds the tasks a ground operator schedules for execution at the specified time. The implementation of the CRUD methods for Binar-1 is provided in Appendix A.19.

The Single Responsibility principle is demonstrated by separating concerns about managing and executing tasks. The library delegates the responsibility of task execution to the AtSpecifiedTime object. This approach promotes library reuse by offering flexibility and extensibility. When a task is read from the list, it is provided to the AtSpecifiedTime object to execute when appropriate. Calling "Execute" on the AtSpecifiedTime object used on Binar-1 traps the execution in a loop, waiting for the set start time for the provided task. As the application code uses the scheduler in an RTOS thread (Section 3.2.8), the thread is yielded until the start time is met. Once met, the "GetTask" method on the TimeSeriesTaskNode object is called to return the task, which is then executed through the "Run" method. However, this approach may not continue to be suitable as requirements change for future missions. For example, future requirements may prefer task execution on an independent processor or the flight computer executing the task in an independent thread. Adopting the Single Responsibility principle allows these implementations to be made in the future without changing the well-tested TimeSeriesTaskScheduler module.

Adherence to the Dependency Inversion principle further increases the reusability of the library. The TimeSeriesTaskScheduler aggregates a pointer to an implementation of an AtSpecifiedTimeInterface object and hence interacts with the implementation via an abstraction to its interface. When the library is reused on future missions that require alternate methods to execute tasks, any derivations of the AtSpecifiedTimeInterface object can be substituted into the TimeSeriesTaskScheduler and controlled polymorphically. Adherence to this principle also aids in library testing. Fake AtSpecifiedTime implementations are provided to the TimeSeriesTaskScheduler object during testing to verify the task scheduler's behaviour, increasing the module's reliability.

To refrain from using dynamic memory, the possible tasks to schedule on Binar-1 must be known during compile time. To statically allocate the memory required for any task the ground operator wanted to schedule in any order, the TimeSeriesTaskNode object used in the templated instantiation of the MemoryPoolNode object stores the scheduled task in a type-safe union of all available tasks. This union informs the compiler to allocate enough memory for the largest task to fit within the TimeSeriesTaskNode object. The tasks that were available for scheduling on Binar-1 were:

- Take Picture
- Get Image Names
- Generate Thumbnail
- Compress Image
- Split Image
- Get Image
- Self-Check
- Use Star Tracker
- Move Manipulator
- Detumble

A detailed description of the usage of these tasks is provided in Appendix A.19. This library benefits from extensibility by abstracting the tasks with a union, decoupling code that uses them from the specific task to be run. As no part of the task scheduler is coupled to the specific set of tasks available, the tasks can be altered and set at compile time without changing the TimeSeriesTaskScheduler that uses them. Each task

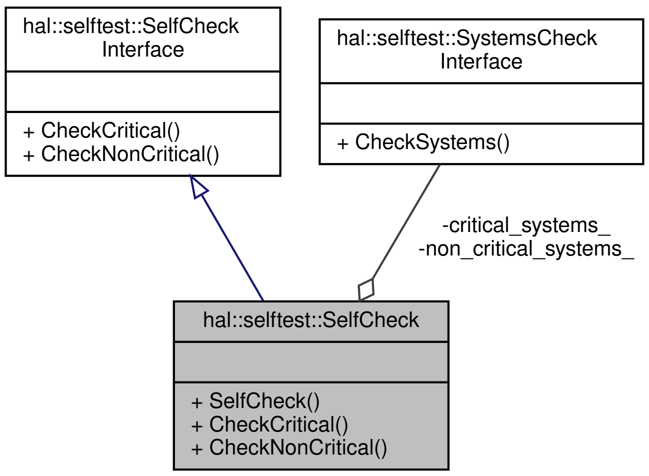


FIGURE 3.29: A UML depiction of the SelfCheck object from the Self Check library. The object aggregates pointers to objects derived from a SystemsCheckInterface, giving access to critical and non-critical systems.

in the variant list extends and overrides a TimeSeriesTaskInterface specifying a ”Run” method. When the AtSpecifiedTime implementation determines the task start time is met, the ”Run” method is executed on the task without the object requiring knowledge of what the underlying task is. The ease of reuse of this library has been observed in developing the task set for the Binar-2, 3 and 4 satellites, where only the tasks needed to be developed - the existing scheduling service and all the existing tests could be brought forward to the new project.

3.5.3. SELF CHECK

In pursuing CubeSat resilience and autonomy, the Self Check library was created for the spacecraft to perform checks of onboard systems to identify faults. The application software using the BSF achieves this by using the SelfCheck object (Figure 3.29).

The self-check library adheres to the Single Responsibility principle by delegating the responsibilities of system checks to individual modules. Instead of having one object concerned with checking all systems, it collates objects responsible for checking critical and non-critical systems. This separation of concerns supports reusability as the critical and non-critical systems on board a spacecraft are mission specific. If, for example, several non-critical systems are added in a future mission reusing the library, it will not also require a change in the critical systems module. Moreover, it aids in understandability as systems are segregated as critical or non-critical.

The collated objects responsible for checking the critical and non-critical systems are implemented following the Open-Closed principle by extending the SystemsCheck-Interface object to provide systems-specific implementations. This approach allows objects to depend on a stable interface not coupled to any particular systems on the spacecraft, promoting the reusability of the library.

The implementations are provided to the SelfCheck object through dependency injection, which interacts with them through an abstracted pointer to the interface they extend. Using interface pointers to abstract the implementation of which systems are being checked conforms to the Dependency Inversion principle. This approach aids in the extensibility of this library as the support for new systems to check can be developed and substituted into the public-facing SelfCheck object without requiring a change to the library. Moreover, it assists with breaking hardware dependencies, making the library easier to test and more reliable. The hardware implementation of the critical and non-critical systems check objects are represented by the UML diagrams in Figures 3.30 and 3.31.

The CheckCriticalSystems and CheckNonCriticalSystems further exemplify the Single Responsibility principle by delegating the responsibility of performing a check to classes responsible for their own system. The CheckCriticalSystems object aggregates the most important systems on the satellite, all of which are essential for nominal operations. The object also aggregates a pointer to a derived class of a system context interface, providing implementations for rebooting the spacecraft and reading and writing to non-volatile memory. All aggregated SystemCheckInterface derivations provide the implementations for checking critical systems. The critical systems on Binar-1 comprised the power, temperature, and error logging systems.

The CheckNonCriticalSystems object aggregates the systems that are not essential for the operation of the satellite. The three aggregated SystemCheckInterface derivations provide methods for checking the magnetorquers, GPS, and IMU systems. All the non-critical systems on Binar-1 were peripherals, so checking was done subject to the

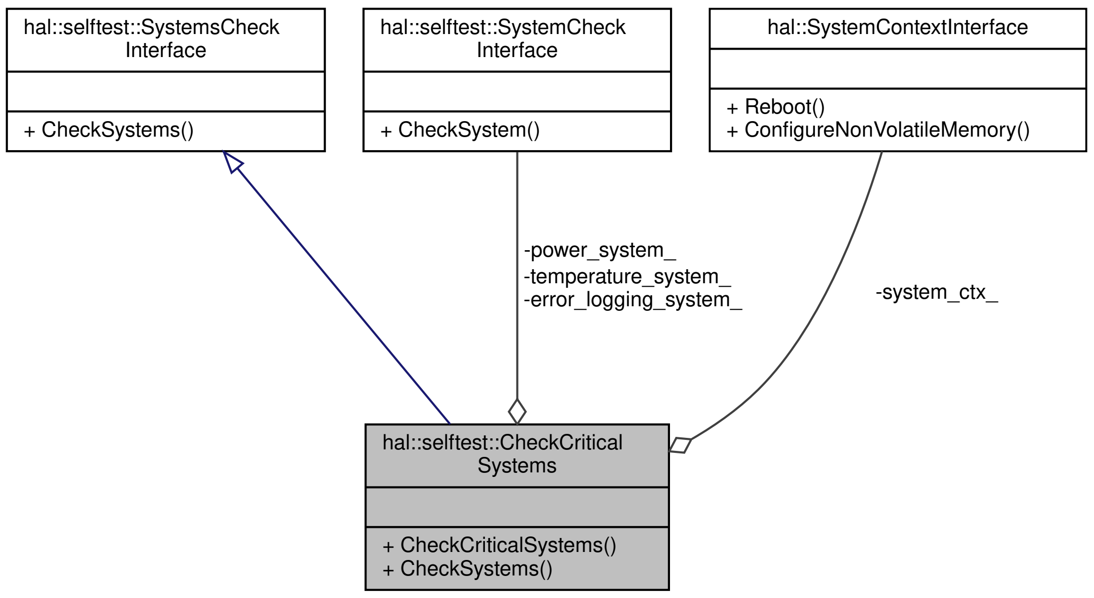

hal::selftest::SystemsCheck Interface
+ CheckSystems()
hal::selftest::SystemCheck Interface
+ CheckSystem()
hal::SystemContextInterface
+ Reboot()
+ ConfigureNonVolatileMemory()
-power_system_
-temperature_system_
-error_logging_system_
-system_ctx_
hal::selftest::CheckCritical Systems
+ CheckCriticalSystems()
+ CheckSystems()

FIGURE 3.30: A UML diagram of the CheckCriticalSystems object in the Self Check library. This object aggregates pointers to critical systems derived from the System-CheckInterface object for checking.

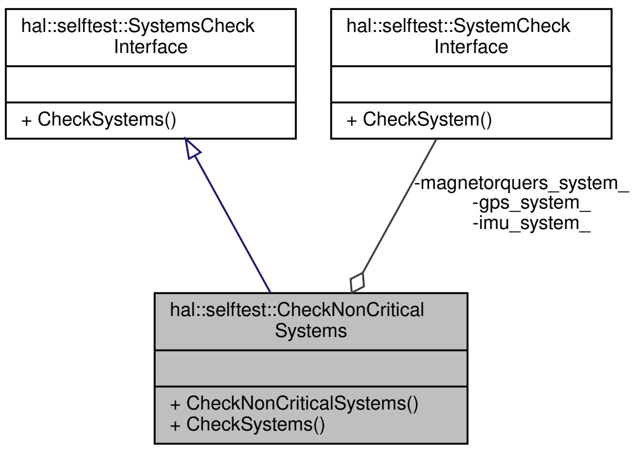

hal::selftest::SystemsCheck Interface
+ CheckSystems()
hal::selftest::SystemCheck Interface
+ CheckSystem()
-magnetorquers_system_
-gps_system_
-imu_system_
hal::selftest::CheckNonCritical Systems
+ CheckNonCriticalSystems()
+ CheckSystems()

FIGURE 3.31: A UML diagram of the CheckNonCriticalSystems object in the Self Check library. This object aggregates pointers to non-critical systems derived from the SystemCheckInterface object for checking.

peripheral availability through the ownership manager. Failing to access the peripheral would conservatively return a fail for the system check. Technical details on performing crtical and non-critical system checks are provided in Appendix A.20.

Both the CheckCriticalSystems and CheckNonCriticalSystems also adhere to the Open-Closed and Dependency Inversion principles. They both extend the same interface but provide alternative implementations. If the SystemsCheckInterface was rather open for modification, the critical and non-critical systems to test would be coupled together. This would remove granularity in system checks. In this application, adherence to the Open-Closed principle aids in achieving the separation of concerns. The implementation objects heavily leverage the Dependency Inversion principle, as each system is checked through an abstraction. This approach assists with reusing the library, as it allows for new implementations of systems to be controlled without requiring library modification.

These systems are controlled through an abstraction to the SystemCheck interface object. For implementing the library in the Binar-1 flight software, the Interface Segregation principle was followed by further deriving this common interface into multiple device-specific interfaces. This approach benefits from being loosely coupled. Objects can selectively depend on relevant interfaces rather than a bloated object implementation. This approach can make the library easier to maintain and reuse. Moreover, the library can be tested with high granularity, as mocked modules can be substituted for any interface allowing the behaviour of the module that interacts with them to be closely examined.

3.5.4. BEACON

As Binar-1 communicated in the amateur frequency spectrum, it was required to periodically output non-encrypted beacons that radio amateurs could receive and deserialise. For this purpose, the Beacon library was created. As a further benefit, regular beacons enable tracking when the satellite is not over the Curtin University ground station using the SatNOGS network.

TABLE 3.3: The serialised structure of the Binar-1 beacon.

Byte Index	Data	Byte Index	Data
[0, 1]	3.3 line voltage	[24, 25]	z axis magnetorquer current
[2, 3]	Battery voltage	[26, 27]	Redundant X axis magnetorquer current
[4, 5]	X axis solar panel voltage	[28, 29]	Redundant Y axis magnetorquer current
[6, 7]	Y axis solar panel voltage	[30, 31]	X- axis solar panel temperature
[8, 9]	3.3 line current	[32, 33]	X+ axis solar panel temperature
[10, 11]	Battery current	[34, 35]	Y- axis solar panel temperature
[12, 13]	X axis solar panel output current	[36, 37]	Y+ axis solar panel temperature
[14, 15]	X axis solar panel input current	[38, 39]	Battery heater 1 & 2 temperature
[16, 17]	Y axis solar panel output current	[40, 41]	Battery heater 3 & 4 temperature
[18, 19]	Y axis solar panel input current	[42, 45]	Latitude
[20, 21]	X axis magnetorquer current	[46, 49]	Longitude
[22, 23]	y axis magnetorquer current	[50, 85]	Custom message

The beacon object has one public-facing method, MakeBeacon, used by a dedicated RTOS thread (Section 3.2.8) to beacon to the ground. A restriction imposed on the beacon functionality by the COTS transceiver on Binar-1 limited the message to 85 bytes. The information contained in the beacon is summarised in Table 3.3.

When called, the "MakeBeacon" method collects the voltages, currents and temperature objects using the Analog object discussed in Section 3.2.1. The latitude and longitude are collected from the GPS object, discussed in Section 3.2.6. Using the byte boundaries specified in Table 3.3, the raw bytes from the collected readings are copied into the beacon buffer, following the specific contiguous order. Finally, the custom message is appended to the serialised beacon. On Binar-1, this message was *"VK6BUS BINAR1-Contact binarspace.com"*.

3.6. COMMUNICATIONS MODULES

Effective communication between a spacecraft and its ground station has an essential role in the success of a space mission. The communications libraries detailed in this section are designed to facilitate this, providing a reliable and efficient process for transmitting and receiving data. By detailing the development of these modules, this section aims to provide insight into the operation of the communications system in Binar spacecraft, highlighting its crucial role in mission success.

3.6.1. REMOTE PROCEDURE CALL

From a high-level perspective, a standard software application consists of a collection of procedures or functions, which get called by higher-level software to complete its purpose. One of the purposes of using functions in an application is that they allow for the abstraction of implementation. Calling code understands how to interact with a procedure regarding its parameters and returns but does not need to understand how they are correlated. When a procedure is called in a singly threaded application, it assumes control of the main program thread to execute the containing code. This synchronous implementation may be satisfactory for a simple application but may be an inefficient method of execution when the requirements of the software become more complicated. Moreover, depending on the module's complexity, the function may benefit from execution on a different computer entirely. Because function implementations are abstracted from calling code, parameters can be sent to a remote machine for execution rather than executing on the host machine. Variables can be returned to the calling code with the same syntax as if the procedure were executed locally. Unlike local execution, however, this frees up the CPU on a single-threaded application to execute other instructions. This approach is called a remote procedure call. As the software execution in an RPC network happens remotely, there is no requirement for the architectures of the processors or the language used to be the same. This introduces a challenge, however, in developing a standardised interface allowing for communications between the execution machines. One possible method of communication could be serialising a data structure for transmission. Sharing the same header file between the local and the remote machines would allow the received raw byte stream to be cast back into the initial data structure. However, implementing this method needs both computers to use the same memory alignment and hence does not provide portability. Moreover, this method requires the datatype being transferred to be known in advance to correctly cast back to a data structure, reducing the extensibility of the communications library. A common approach is to use ASCII-based communications to compensate for problems introduced by alignment. As ASCII characters can be represented using one byte, either computer on the RPC network can unambiguously know what is being transferred as it is received byte-by-byte. However, this method suffers from a severe overhead. For example, if the ground station were to request the current up-time of the spacecraft, it may return a value such as 259,200, signifying the number of seconds since the last reboot. Representing this number using ASCII characters would require six bytes. However, if the number were directly serialised, it could be represented using only four bytes as it fits within the scope of a 32-bit integer. In spacecraft communications, where the ground station pass-over time may be limited, reducing the time required to complete communications is essential. Therefore, the optimal serialising approach must be architecture and programming language-agnostic, enabling platform portability and offering minimal overhead. These key areas are addressed through Google's Protocol buffers. Traditionally, protocol buffers require dynamic memory for serialising and deserialising and hence did not apply to memory-constrained platforms. However, in early 2020, Google released the Pigweed project, an open-source collection of C++ libraries targetting memory-constrained embedded devices [136]. As part of this collection, a port of protocol buffers was created for embedded devices. Hence, the RPC library was created on top of the protocol buffers implementation to provide a minimal overhead, easily extensible means of communicating with Binar-1. Moreover, it supports backward compatibility, meaning future development for a ground station client can control older satellite server versions.

This library demonstrates a novel approach to spacecraft communications. Conventional approaches to data serialisation in satellite communications often rely on industry standards such as CCSDS, AX.25, and ECSS protocols [137]. While these standards provide reliability, they can be platform and language-specific, limiting flexibility and portability. Moreover, there is a lack of standardisation between the various serialisation techniques in the literature. This lack of standardisation can limit the reusability of ground station software between missions, further intensifying the challenges CubeSat teams face regarding rapid development cycles. The approach to spacecraft communications implemented by the candidate offers many benefits over the conventional approach to spacecraft communications. The architecture and language agnosticism achieved through an RPC approach offers flexibility for communication integration between satellites and ground stations. Satellite missions may involve a variety of platforms and hardware. Ensuring platform portability enables the same communication protocols to be used across different satellite systems, reducing the need to develop custom communication protocols for each mission, hence reducing development and integration efforts. This contributes an innovative solution to the challenge of short time frames for CubeSat missions. The ability to maintain compatibility with legacy systems is a valuable feature, especially in space programs that have multiple spacecraft in orbit at once. A space program iteratively building upon its RPC command suite from the most common telemetry requests through mission-specific requests as their technology evolves can use the same ground software to speak to all assets on orbit. This approach demonstrates an innovative approach to the solution of the lack of standardisation seen in satellite communications in the CubeSat domain. Moreover, this approach provides a solution to the challenge of limited resources on CubeSat platforms, as an RPC/protobuf approach allows for data serialisation with minimal overhead.

The Binar-1 flight software could communicate on three RPC networks; Application, Safe, and Start-Up Mode. The library applies the Open-Closed principle by having each mode extend the same interface class, ServerInterface, exhaustively containing the methods possible for the spacecraft to execute. When inherited, the deriving classes override each method in the abstract virtual interface and implement mode-specific features. Table 3.4 lists the various methods available in the ServerInterface, and the modes that use them.

In the Binar-1 implementation of the three RPC networks, the responsibility of performing the functionality of the procedure call was delegated from the main server module to multiple aggregated objects per the Single Responsibility principle. This improves the reusability of the library by increasing modularity. It decreases coupling between irrelevant procedure calls, meaning a change to the implementation of one call will not also reflect a change in another. Moreover, being a complex library in the framework, a clear separation of concerns makes the code easier to understand and maintain. The alternative approach to having the RPC server objects directly responsible for implementing all the procedure calls would result in a large, monolithic object that could be very difficult to maintain. This would increase the likelihood of bugs being introduced into flight software.

TABLE 3.4: The three RPC servers available in the various operation modes on Binar-1 and the supported procedures.

RPC Method	Application Mode	Safe Mode	Startup Mode
GetVoltages	*	*	
GetCurrents	*	*	
GetTemperatures	*	*	
Ping	*	*	
FirmwareUpdate	*	*	*
ScheduleTask	*		
GetErrorRegisters	*	*	
DeleteImages	*		
GetImuData	*	*	
DownloadLog	*		
DownloadManipulatorLog	*		
DownloadImage	*		
DisableBeacon	*		
ChangeFlashBank	*	*	
ClearErrorRegisters	*	*	

Application mode is the nominal operation mode for the spacecraft, hence has the full suite of methods available for execution. The ApplicationModeServer object (Figure 3.32) aggregates eight private members during construction, responsible for implementing procedure calls: scheduler_, a TimeSeriesTaskScheduler object (Section 3.5.2), sender_, a pointer to a SendInterface, receiver_, a pointer to a ReceiveInterface, temperature_reader_, voltage_reader_, and current_reader_, pointers to analog reader interfaces (Section 3.2.1), router_, a pointer to a router interface, and system_ctx_, a pointer to a derivation of a SystemContextInterface (Section 3.2.9).

Safe mode is a feature-limited operational mode with a stripped-back set of available methods. The SafeModeServer (Figure 3.33) aggregates six private members responsible for implementing procedure calls; voltage_reader_, current_reader_, temperature_reader_, sender_, receiver_ and system_ctx_.

The Start-Up mode RPC network (Figure 3.34) is only used during the firmware update process. Hence, the StartUpModeServer only aggregates three private members for implementing the procedure call: firmware_reader_, sender_ and receiver_.

All three RPC servers implement the Dependency Inversion principle by interacting with the objects responsible for the procedure calls through abstractions. This facili-

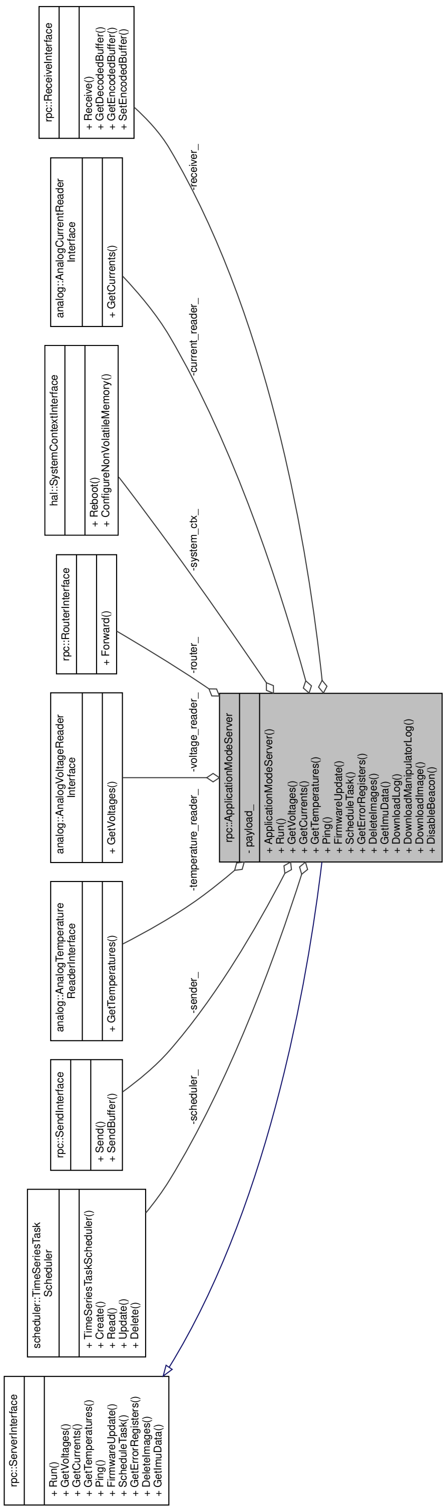


FIGURE 3.32: A UML depiction of the RPC Application Mode server in the Binar-1 flight software.

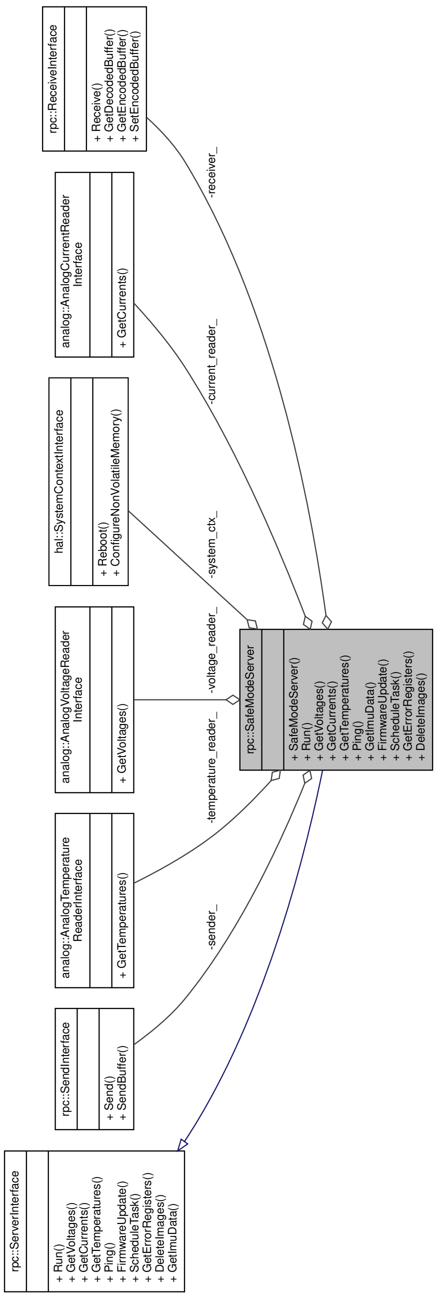


FIGURE 3.33: A UML depiction of the RPC Safe Mode server in the Binar-1 flight software.

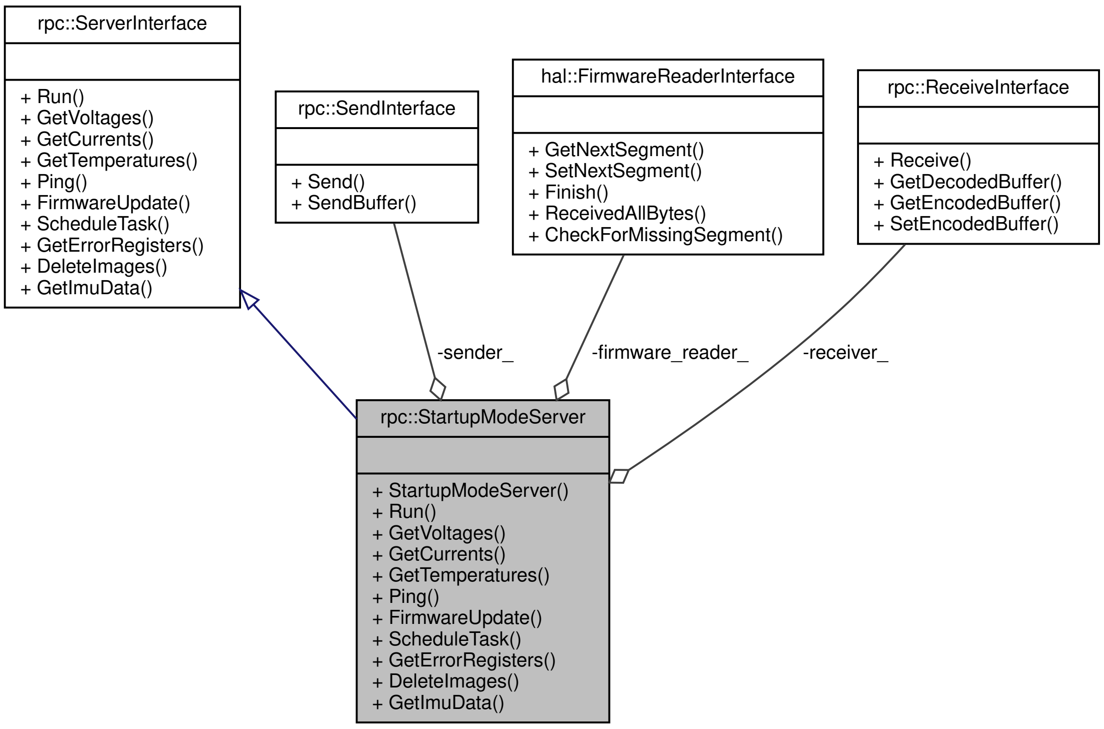


FIGURE 3.34: A UML depiction of the RPC Start-Up Mode server in the Binar-1 flight software.

tates library reuse by allowing new implementations of the modules to be developed and substituted into the RPC server if mission requirements dictate. It also enables testing of the RPC networks through mocked substitution. The mocked modules can be monitored to ensure the RPC network is interacting with the procedure implementation objects as expected, verifying the behaviour of the communications library and increasing reliability.

All three networks share the common trait of requiring data to enter and exit their respective modules. This is achieved through extending the SendInterface and ReceiveInterface objects, injected into the constructor for the modules to be used during operations, aligning with the Dependency Inversion principle. Decoupling the RPC networks from how data is received and sent is extremely useful for reusability and testing. It means that the RPC library can be reused with a different communications method, allowing the communications hardware to be changed. The benefit of this has been observed moving from Binar-1, which used a COTS communications solution, to Binar-2, 3 and 4, which use a custom-developed transceiver. It also allows the

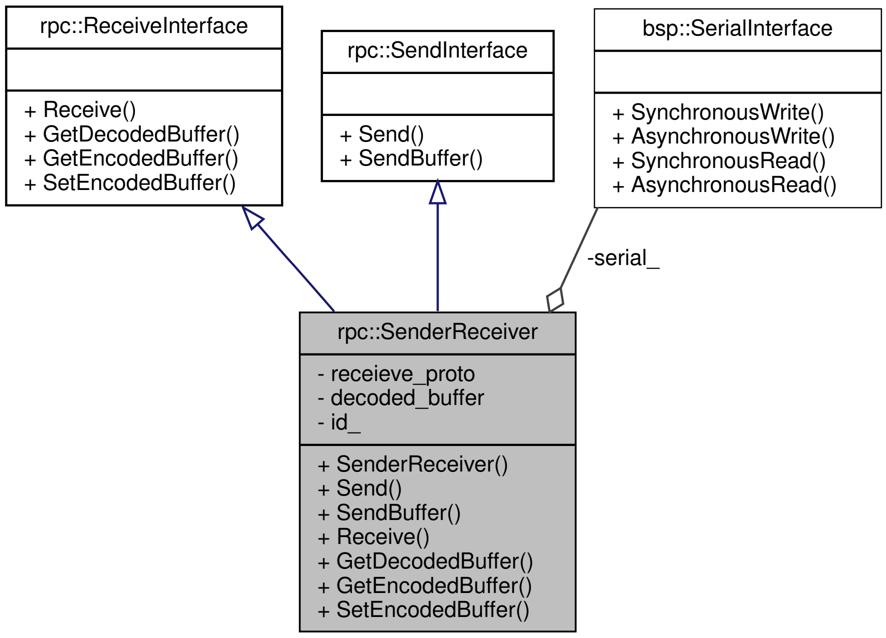


FIGURE 3.35: A UML depiction of the SenderReceiver object in the Binar-1 flight software.

communications network to be tested entirely on the host machine by providing faked implementations for sending and receiving. In the hardware implementation for Binar-1, the SendInterface and ReceiveInterface objects were extended into a SenderReceiver object, shown in Figure 3.35.

The sender_ and receiver_ members point to the same object in memory as it is derived from both SenderInterface and ReceiverInterface. However, as the members are pointers to their respective interface classes, they only have access to relevant methods; receiver_ can only receive, and sender_ can only send. This implementation made it easy to tag messages with a transaction ID without having to duplicate code whilst maintaining encapsulation.

The "Run" method on the RPC server first dereferences the receiver_ member and calls the "Receive" method. This method calls the "GetDecodedBuffer" method to decode a received buffer. The decoded buffer is then passed to the protobuf library to deserialise. The Pigweed protobuf library decoder returns the type of request contained within the serialised byte stream but does not provide means of extracting the data. Hence, each request type has a custom corresponding decoder handler object to extract the information contained within the request to use in the RPC method.

After decoding the message, the intended address is checked against the unique flight computer ID. Suppose the message is intended for a different computer. In that case, the encoded message is retrieved using the "GetEncodedBuffer" method on the receiver object. It is forwarded to the secondary flight computer using the "Forward" method on the privately aggregated router_ object (Figure 3.36) for a further check. This process repeats to the secondary bus until the intended recipient is found.

The Router object aggregates two private members: supervisor_computer_ and primary_computer_. These members are template instantiations of the Serial class. The benefits achieved from the design decisions of the Serial library were discussed in Section 3.2.1, and so will not be replicated here. Using templates in this application benefits from compile time polymorphism, supporting library reuse. Further template concretions can be added to the serial library to support routing data to other endpoints and can be used with the Router library. Following the Single Responsibility principle and having the router object separate from the main RPC network allows other libraries that require data routing to reuse the module without needing to depend on the entire RPC library. Technical details of the implementation of the RPC library on Binar-1 are provided in Appendix A.21.

3.6.2. Consistent Overhead Byte Stuffing

Data transmission in a non-master/slave configuration introduces issues regarding receiving full packets. This impacts the spacecraft's communications between the primary and secondary flight computers and the ground station. When variable-length packets are required, the receiving device must know when to start listening to a packet and when to stop. A commonly used method for solving this issue is reserving bytes to denote a transmission frame's start and end. This method, however, requires data to be scaled to not naturally include these reserved bytes so they can be used for delimiting.

Consistent Overhead Byte Stuffing (COBS) is a packet framing algorithm for encoding serialised data with minimal overhead [138]. It is a computationally inexpensive means of converting bytes from [0, 255] to [1, 225]. The encoding (Figure 3.37) is implemented by pushing an offset to the front of the serialised array indicating the position

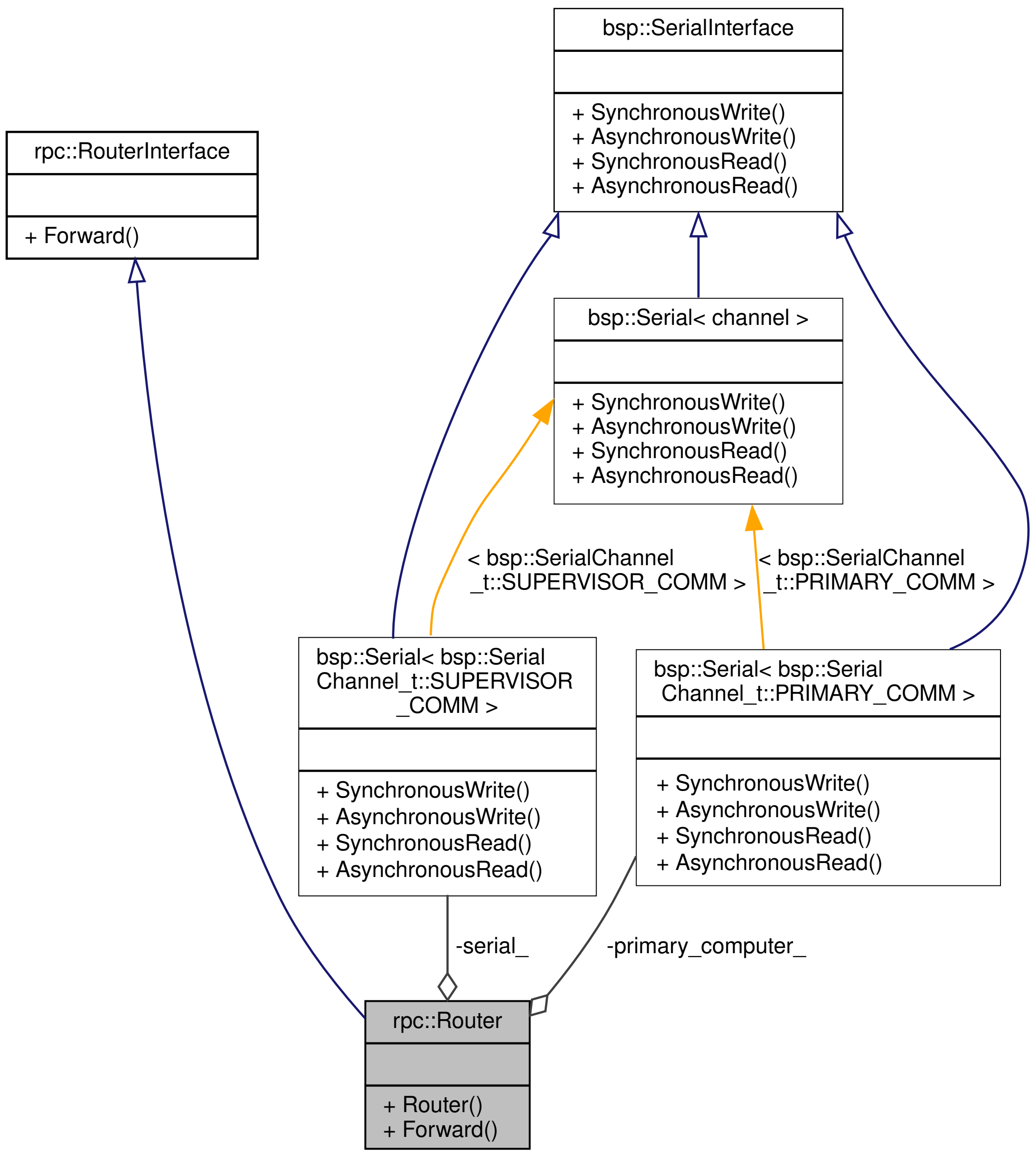


FIGURE 3.36: A UML depiction of the RPC router in the Binar-1 flight software.

TABLE 3.5: An example array of bytes encoded using the COBS algorithm. The differences with the source array are highlighted in red. The algorithm adds one byte of overhead at the start of the byte array and a delimiter at the end.

Raw Byte Array	**Encoded Byte Array**
29 00 12 19 00 95	02 29 03 12 19 02 95 00

of the first occurrence of a zero byte. This zero byte is then replaced by the offset to the next occurrence of a zero byte. This process is repeated until the last zero byte occurrence. The final zero byte is replaced by the offset to the end of the serialised array. If no zero bytes exist in a serialised message, one more than the size of the serialised buffer is pushed to the start of the array to denote the location of the delimiter. A zero byte can then be pushed to the end of the array, acting as a delimiter between messages. The algorithm adds one byte of overhead per 254 bytes. An example conversion can be seen in Table 3.5.

Using a COBS communication approach provides an efficient means of transferring serialised data reliably. The simplicity refrains from complex methods of data framing, which could introduce errors. Moreover, the simple encoding/decoding algorithm is optimal for use on an embedded system with limited memory.

3.7. OPERATION MODES

A series of operation modes had to be defined for the spacecraft to operate effectively on orbit. For Binar-1, these operation modes comprised the Start-Up/Bootloader, Safe Mode, and Application Mode. A high-level diagram describing the transitions between these modes can be seen in Figure 3.38. Red lines indicate automated transitions, and green lines indicate manual telecommand transitions. The roles of each mode will be examined, and how they were implemented on the Binar-1 spacecraft using the libraries discussed in the previous sections.

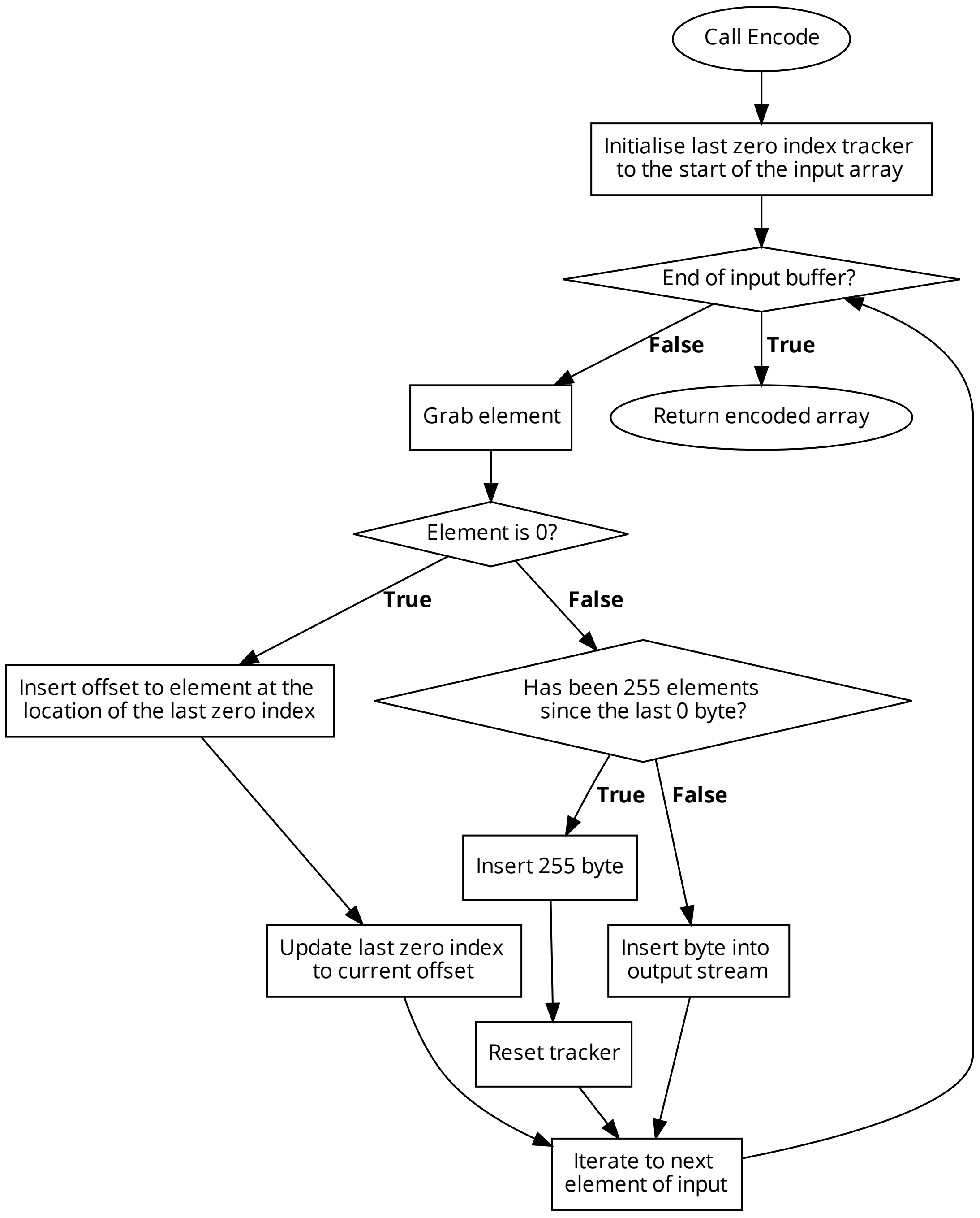

Call Encode
Initialise last zero index tracker to the start of the input array
End of input buffer?
False
True
Grab element
Return encoded array
Element is 0?
True
False
Insert offset to element at the location of the last zero index
Has been 255 elements since the last 0 byte?
True
False
Insert 255 byte
Update last zero index to current offset
Insert byte into output stream
Reset tracker
Iterate to next element of input

FIGURE 3.37: The high-level logic for performing a COBS encode.

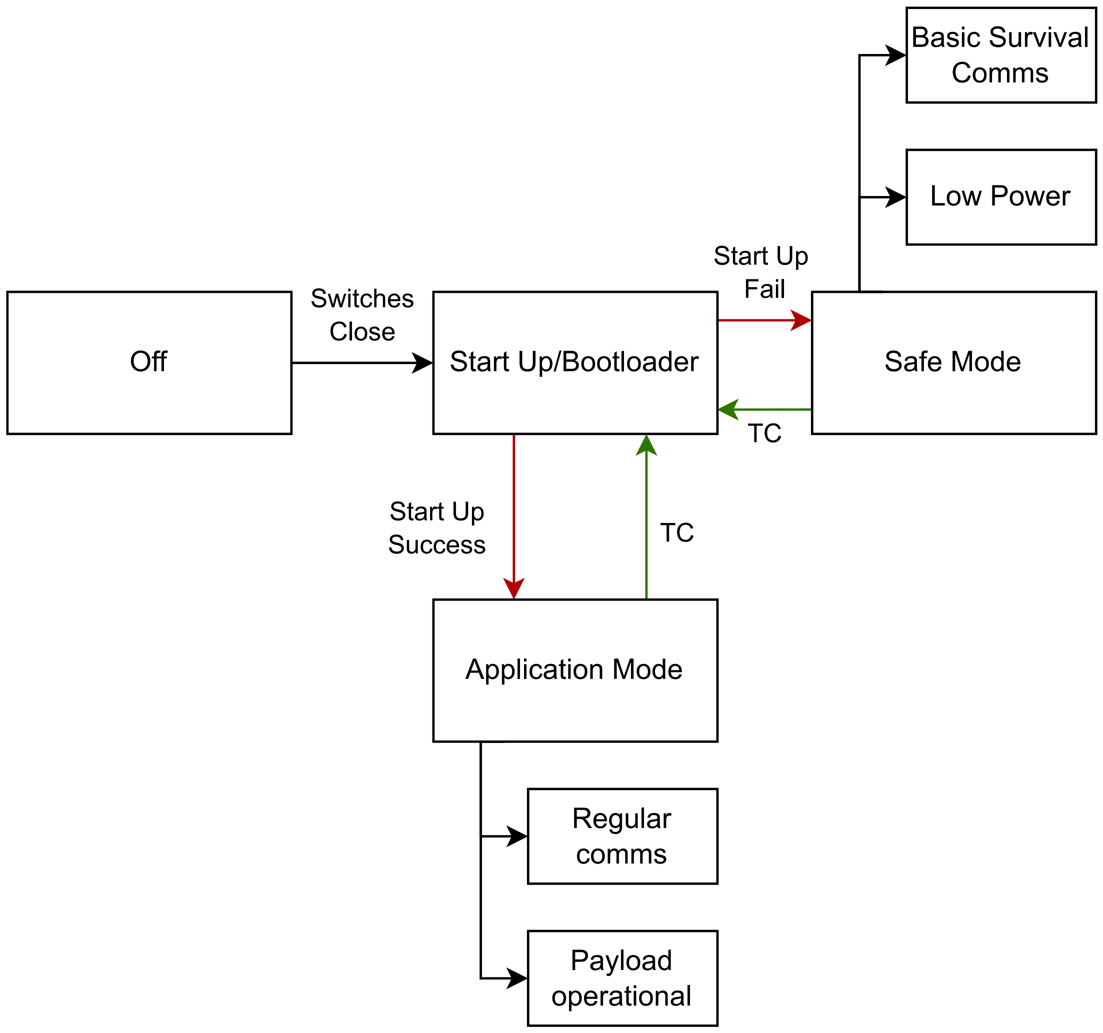


FIGURE 3.38: The operation modes on Binar-1 and how they are related. Red lines indicate automatic transitions, and green lines manual telecommand transitions.

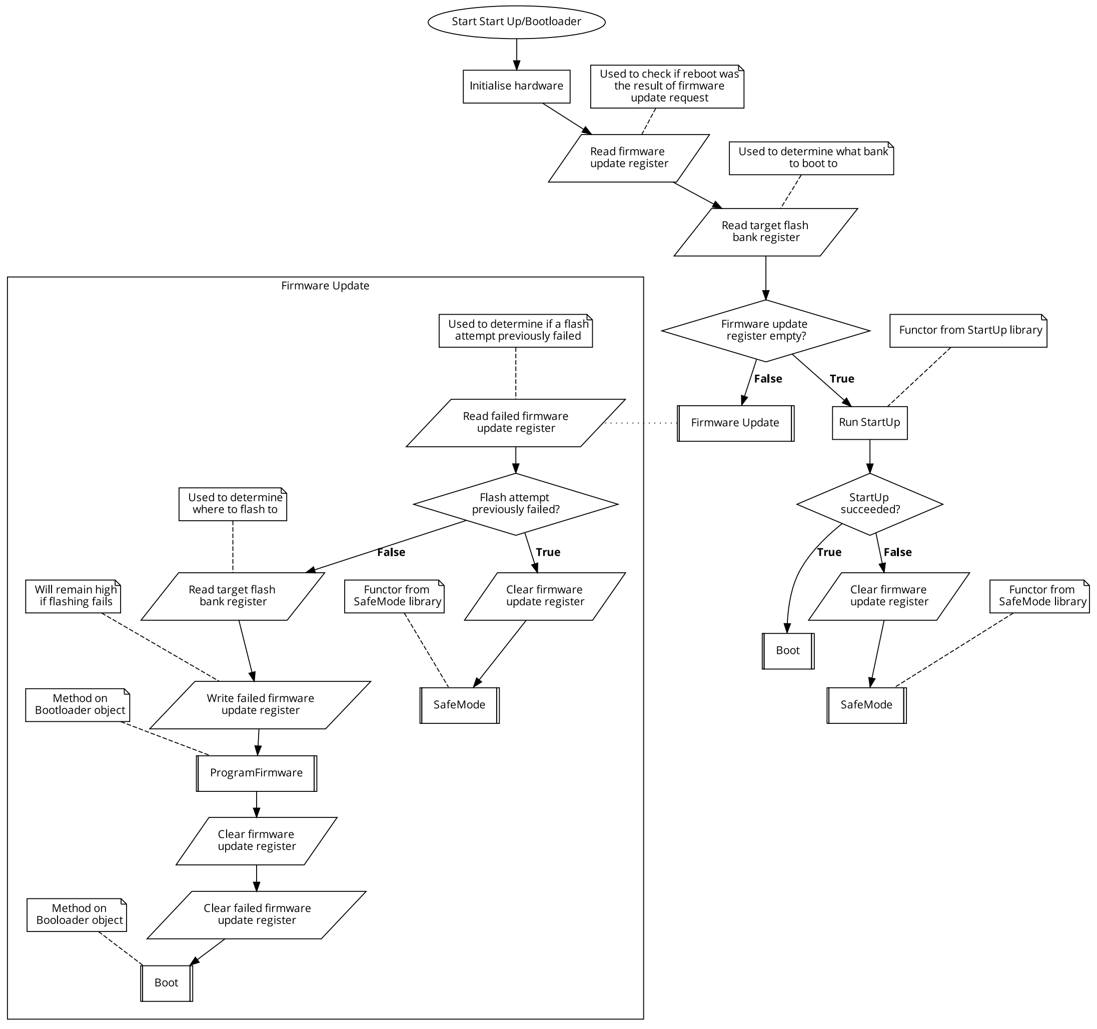

Start Start Up/Bootloader
Initialise hardware
Used to check if reboot was the result of firmware update request
Read firmware update register
Used to determine what bank to boot to
Read target flash bank register
Firmware update register empty?
Functor from StartUp library
False
True
Firmware Update
Run StartUp
StartUp succeeded?
True
False
Boot
Clear firmware update register
Functor from SafeMode library
SafeMode
Firmware Update
Used to determine if a flash attempt previously failed
Read failed firmware update register
Flash attempt previously failed?
Used to determine where to flash to
False
True
Will remain high if flashing fails
Read target flash bank register
Functor from SafeMode library
Clear firmware update register
Write failed firmware update register
SafeMode
Method on Bootloader object
ProgramFirmware
Clear firmware update register
Method on Booloader object
Clear failed firmware update register
Boot

FIGURE 3.39: A high-level overview of the logic flow for the Start-Up/Bootloader operation mode. This mode is entered during Binar-1 boot. It is concerned with executing any required start-up software, handling firmware updates, and booting to the application flash or Safe Mode.

3.7.1. START UP/BOOTLOADER

The start-up/bootloader software is the first operation mode the spacecraft enters when booting. Figure 3.39 provides a high-level overview of the operation mode. After initialising the hardware, the bootloader reads from the firmware update register (Section 3.2.9) to discern if the entry was due to a firmware update request. If a begin firmware update request is received, the application logic will write this register with the number of firmware segments to be received. The spacecraft will also read the target flash bank register to determine the flash bank targeted for boot by the ground station.

Control passes to the Firmware Update function if the firmware update register contains a non-zero value. This function reads the failed firmware update register to check if the currently serviced firmware update request had previously failed, causing a reboot. If this register is set, the firmware update register is cleared before passing control to the SafeMode object (Section 3.7.2). If the failed firmware update register is reset, the target bank to flash to is read from non-volatile memory to progress into programming firmware. The failed firmware update register is set, and control passes to the ″ProgramFirmware″ method of the bootloader object (Section 3.4.1). Failed attempts to program firmware will cause a system reboot. After programming is complete, the firmware update register and the failed firmware update register are cleared. Control is passed to the ″Boot″ method on the Bootloader object to jump to Application Mode.

If the firmware update register was empty from the initial check, control passes to the StartUp object (Section 3.4.3) to perform the boot count, boot timer and self-check. If the start-up process fails, control is passed to the SafeMode object to await input from the ground. If the start-up succeeds, however, control is passed to the ″Boot″ method of the Bootloader object to jump to the application code.

3.7.2. SAFE MODE

When the Start-Up/Bootloader firmware logic discerns that it is unsafe for the spacecraft to boot into Application Mode, it will transition into safe mode instead of booting to application code. Safe mode is a common state in space missions that aims to prevent damage to the satellite. It disables non-essential subsystems on the spacecraft and conserves power until regular operations can be resumed. The functionality of the safe mode state is provided in the BSF through the safe mode library. Figure 3.40 describes a high-level overview of the Safe Mode logic.

Upon entering Safe Mode, the SafeMode object constructs the Safe Mode RPC server, which has a stripped-back set of commands available from the ground station. The beacon object (Section 3.5.4) is initialised with a message indicating the spacecraft is operating in Safe Mode. Identifying this in advance enables operations to be planned for when communications are established above Curtin University. In this situation, operators can provide commands to discern the reason for entering Safe Mode and instruct an exit if safe to do so. For this reason, the beacon cannot be disabled when the CubeSat is operating in Safe Mode.

After entering safe mode, an asynchronous receive from the transceiver is initiated, and execution enters an infinite loop waiting for instruction from the ground. When a packet is received, it is passed to the safe-mode RPC server for processing. In between receipt of packets from the ground station, the logic checks to see if a beacon packet should be transmitted.

3.7.3. APPLICATION MODE

The Application Mode (Figure 3.41) is the main section of the spacecraft operation. It comprises three RTOS threads (Section 3.2.8); the beacon thread for transmitting periodic beacons, the communications thread for communicating with the ground station, and the application thread for executing scheduled tasks. After booting into the application space, the application first initialises the hardware. The initialisation stage involves configuring the redundant power lines (detailed in Appendix A.11) and

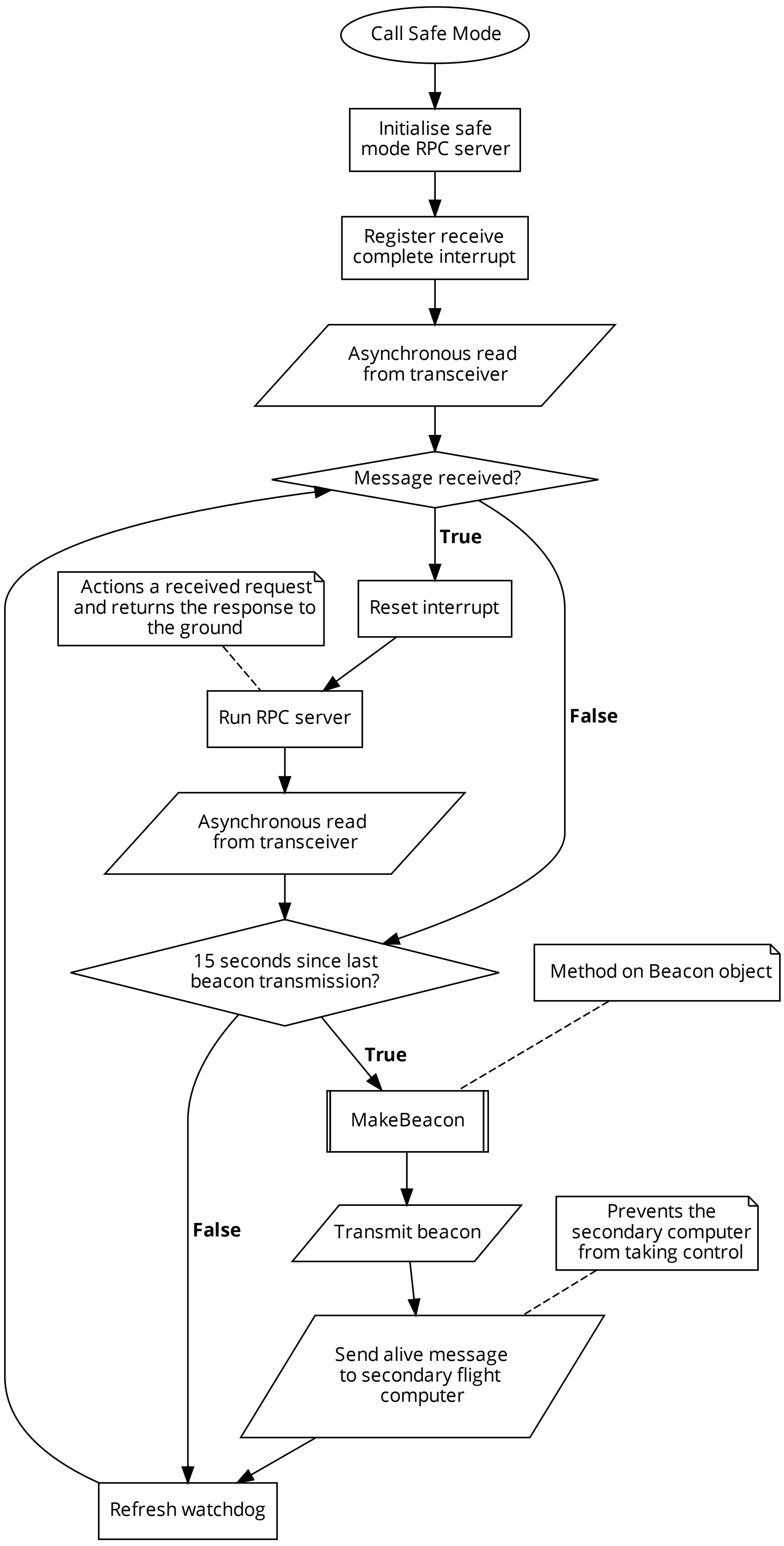

Call Safe Mode
Initialise safe mode RPC server
Register receive complete interrupt
Asynchronous read from transceiver
Message received?
True
False
Actions a received request and returns the response to the ground
Reset interrupt
Run RPC server
Asynchronous read from transceiver
15 seconds since last beacon transmission?
Method on Beacon object
True
MakeBeacon
False
Transmit beacon
Prevents the secondary computer from taking control
Send alive message to secondary flight computer
Refresh watchdog

FIGURE 3.40: A high-level overview of the logic for the SafeMode functor in the Safe Mode library. Control is passed to this mode when an error occurs on the spacecraft.

the redundant motherboard. If the boot process fails and the command to disable the redundant motherboard is not executed, the redundant motherboard will be permitted to boot and take control of the spacecraft. After system configuration, the ownership manager is initialised with the onboard peripherals, namely the GPS, IMU, magnetorquers, and external memory. As mentioned in Section 3.2.2, when peripherals are used during Application Mode, the ownership manager takes out a mutex stored in non-volatile memory. When the bootloader initialises the peripherals, it will first check the mutex section of the non-volatile memory to determine if a reboot was caused when using a certain peripheral. Suppose any mutexes are seen to be still claimed. In that case, the ownership manager will polymorphically construct faked objects instead of the potential faulty peripherals using an object derived from the same interface class. This approach allows the application code to progress as normal and, through polymorphism, gives the developer control over what the faked object should do. In the case of Binar-1, these faulty peripherals would return a failure response to notify the ground and log in the error registers of a faulty peripheral. After initialisation, the application clears the corrupt flash flag in the non-volatile error registers (Section 3.2.9) set prior to the boot instruction during the start-up/bootloader software. The RTOS is initialised, and the three threads are created. A detumble task is scheduled before the kernel is started.

During the thread creation process, the communications thread is given a high priority, and both the beacon and application are given the same normal priority. This higher priority means the communications thread is the first thread to be moved into the running state after starting the kernel. The communications thread initialises the Application Mode RPC server (Section 3.6.1) and registers a callback transaction receive complete interrupt with an event flag. An asynchronous receive is started using the direct memory access (DMA) peripheral, and the thread is then instructed to move to the blocked state whilst waiting for the event flag to be set in the callback interrupt. The thread will yield into the blocked state, and another thread can run. When the DMA receive completes, and the registered interrupt sets the event flag, the thread will be moved from blocked to ready. As it has the highest priority, the scheduler will shift it into running as soon as another thread yields.

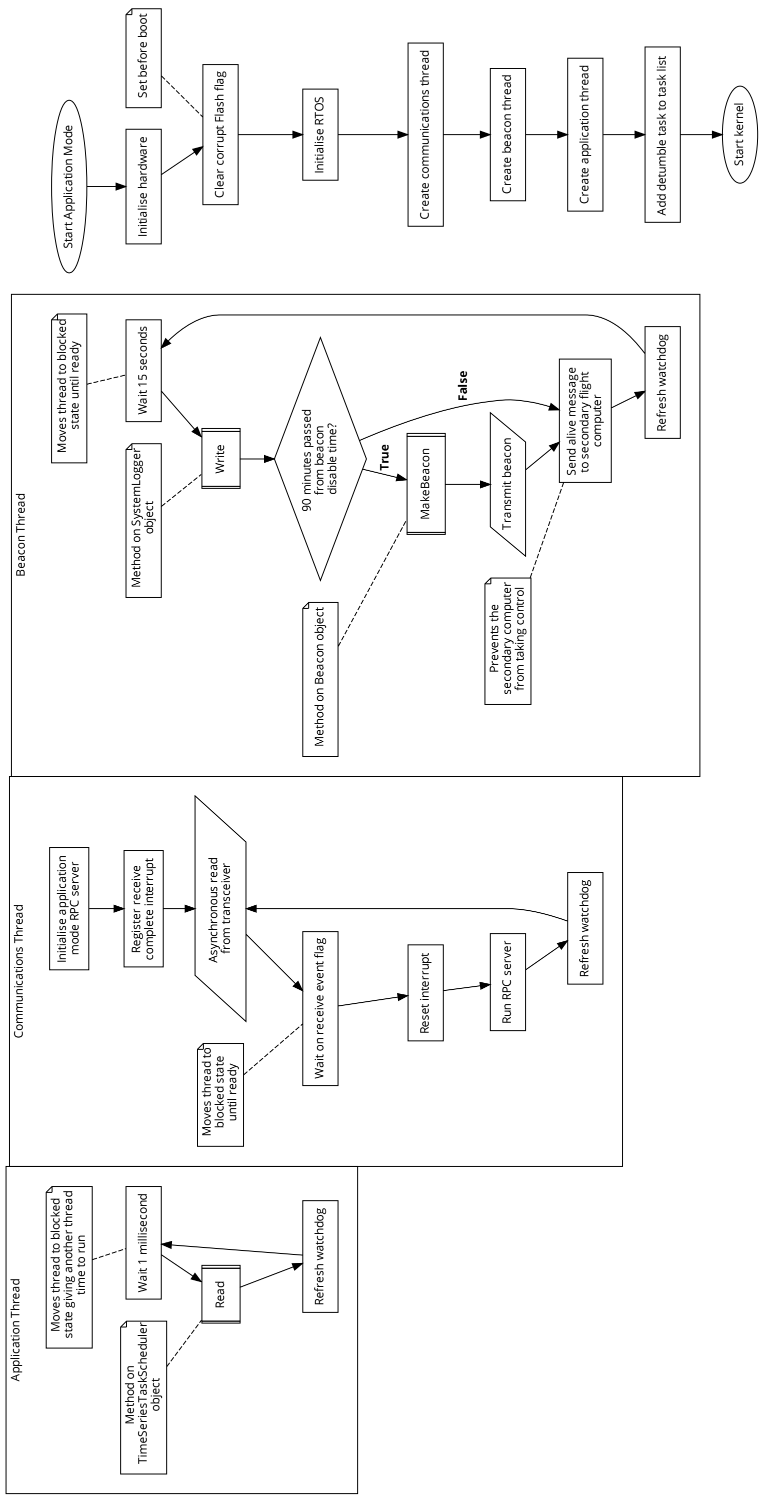

Start Application Mode
Initialise hardware
Set before boot
Clear corrupt Flash flag
Initialise RTOS
Create communications thread
Create beacon thread
Create application thread
Add detumble task to task list
Start kernel
Beacon Thread
Moves thread to blocked state until ready
Wait 15 seconds
Method on SystemLogger object
Write
90 minutes passed from beacon disable time?
True
False
MakeBeacon
Transmit beacon
Send alive message to secondary flight computer
Refresh watchdog
Method on Beacon object
Prevents the secondary computer from taking control
Communications Thread
Initialise application mode RPC server
Register receive complete interrupt
Asynchronous read from transceiver
Wait on receive event flag
Moves thread to blocked state until ready
Reset interrupt
Run RPC server
Refresh watchdog
Application Thread
Moves thread to blocked state giving another thread time to run
Wait 1 millisecond
Read
Method on TimeSeriesTaskScheduler object
Refresh watchdog

FIGURE 3.41: A high-level overview of the logic flow for the Application operation mode. This mode employs RTOS threads to handle communications with the ground, execute time-series tasks, and emit periodic beacons.

When the communications thread is blocked during the waiting process, either the beacon or RPC thread will run with equal priority. The beacon thread begins with a wait instruction for fifteen seconds. This is to limit the period in which beacons are transmitted. During these fifteen seconds, the thread is moved to the blocked state. When the timer elapses, the scheduler moves it into the ready state. When running, the first instruction is for the thread to write to the system logs using the SystemLogger library (Section 3.2.10). Following this, a check is performed to see if a disable beacon request has been sent from the ground station. If the beacons are not muted, a beacon is assembled using the ″MakeBeacon″ method on the Beacon object from the beacon library (Section 3.5.4). This beacon is then transmitted to the ground. As the BCM contains a primary and supervisory computer, the beacon thread is also responsible for communicating with the supervisory flight computer. A message is sent to the supervisory computer at the end of the beacon thread. If the supervisory computer does not receive this message for one hour, it deduces that a problem exists with the primary flight computer and assumes control of the spacecraft. Finally, the thread refreshes the system watchdog before looping to the wait at the top of the thread, yielding for another fifteen seconds.

The application thread starts with a one-millisecond wait instruction. This instruction yields the thread, checking to see if the communications thread or the beacon thread is in the ready state. If not, the application thread continues by calling the ″Read″ method on the TimeSeriesTaskScheduler object to get the task at the head of the list. Re-entrant yielding is supported within tasks to efficiently use clock cycles if a task needs to wait for information. After executing a task, the thread refreshes the system watchdog timer and loops to the wait at the top of the thread, yielding for another millisecond.

3.8. DISCUSSION AND CONCLUSIONS

Binar-1 was deployed from the International Space Station at 17:20 AWST on the 6th of October, 2021. The first opportunity for communications with the ground station at Curtin University was at 23:00 AWST. However, communications and beacons were not established from the CubeSat during this pass. After conducting a failure analysis, the COTS transceiver on Binar-1 was instructed to transmit its beacon separately to the BCM. This request was successful, and secondary beacons were received from the spacecraft. The ability to communicate with only the COTS transceiver rather than the BCM has led the team to believe that a hardware fault occurred on the adaptor board required to convert the UART communication from the flight computer into I2C for the transceiver. This idea is compounded by the fact that the flight logic configures the transceiver to communicate at a bit rate that differs from the transceiver default during boot. However, power cycling was observed from Binar-1 whilst on orbit, and when it occurred, the transceiver would revert to the default communications bit rate. For the transceiver to receive power, it can be discerned that the start-up/bootloader application executed and placed the spacecraft into Safe Mode or Application Mode.

Moreover, as the transceiver power enable line and the antenna deployment burn wires operated through a GPO, the HAL for the GPO peripheral is known to have functioned correctly. However, due to the partial success of the mission, other areas of the BSF could not be verified on orbit. Nevertheless, benefits have been observed regarding the code base reusability as the BSP progresses with the development of the Binar-2, 3 and 4 CubeSats, requiring significantly less development time than Binar-1.

This chapter provided an overview of a collection of libraries, the development of which comprised significant effort over the course of the PhD project. Context is provided through operation modes, showing where the custom-written libraries have been used in the software that enabled Binar-1. The extensive effort put into the HAL libraries decouples the application code from the hardware peripherals. Aside from assisting in the testability of the code base, this effort means that hardware can be changed without affecting application code, assisting in reusability. This benefit has been observed through required hardware changes for the Binar-2, 3 and 4 satellites, such as changing the microcontroller core the flight logic executes on and supporting a change of hardware for the GPS, IMU, magnetometer, and onboard storage peripherals. Controlling the peripherals available onboard was discussed using the Ownership library, providing safe peripheral access during threading to prevent race condition issues.

The utility modules provided reusable tools for other libraries in the BSF. The ubiquitous Result and Visit library provides a way for functions and methods to return varying custom types, statically allocating the required memory and enforcing all possible returns to be handled at compile time. The Static List library provides a templated static implementation of a linked list customisable with length and datatype at compile time.

The boot application modules provided the libraries used by the Start-Up/Bootloader operational mode. The Start-Up library provides an easily extensible interface to implement the functionality required on spacecraft boot. On Binar-1, this was used for the boot timer, boot counter and initial critical system self-check. The bootloader library controls the application bank the bootloader boots into and facilitates firmware updating using the firmware reader library. Where a boot is deemed unsafe, the Safe Mode library provides a safe execution space for the CubeSat until ground intervention is possible.

The application module libraries are used for firmware in the application code space. When initially deployed into orbit, the detumble library is used to slow the angular precession of the CubeSat. The Scheduling Service library, using the Static List utility library, provides a linked-list style approach to schedule time-specific tasks without dynamic memory. Binar-1 used the Self Check library to identify any faults that may exist onboard the spacecraft, checked during the boot process, and through task scheduling. Alongside Safe Mode, the application firmware beacon thread used the Beacon library to assemble unencrypted messages for transmission to the ground.

Communications with the ground were made possible using the communication module libraries. Extensive effort was made toward developing an extensible, platform-agnostic communications protocol by building an RPC framework on top of Pigweed protocol buffers. The serialised messages were transacted with minimal overhead using COBS encoding, adding a single overhead byte per every 254 bytes.

This chapter showcased an innovative approach to CubeSat software engineering to address the unique challenges associated with CubeSat missions. The benefits of enhanced mission reliability are showcased from the first application of the SOLID principles of object-oriented programming to CubeSat software development. By comparing library development with and without adherence to the principles, the contributions to CubeSat software development are clear. Specifically demonstrated through the development of the BSF HAL, adherence to the SOLID principles provides an innovative solution to address the dual challenge of ensuring software safety while meeting the tight development schedules common to CubeSat missions. As a solution to the challenge of operations in a harsh environment, the Ownership library demonstrates fault tolerance and graceful degradation by leveraging polymorphism in a manner that allows for the seamless switching between real and fake peripherals to ensure the satellite's continued operation in the presence of hardware failure. The Result and Visit library represents a unique solution to error handling in space missions by leveraging a relatively recent addition to the C++ standard library to enforce compile-time error handling with library interaction, eliminating the possibility of run-time errors related to error handling. Ensuring mission reliability for CubeSats hinges on efficiently using their limited resources. One innovative solution discussed to tackle this challenge is using a statically allocated linked list for task scheduling. A novel approach to satellite communications using a Protobuf-based RPC service was discussed. The innovative solution proposed provides some advantages over a conventional approach to satellite communications, including portability, standardisation, backward compatibility, and efficient resource usage. The approach was demonstrated to contribute a solution to the challenge of rapid development cycles inherent in CubeSat missions by enabling the same communication protocols to be used across different satellite systems, reducing the need to develop custom communication protocols for each mission. This approach contributes to CubeSat communication standardisation by allowing the same ground station software to control multiple generations of satellites by iteratively building on the RPC command suite and leveraging backward compatibility. Building the RPC library on top of protobuf enables efficient data serialisation with minimal overhead, showing promise for application on resource-constrained computers enabling small spacecraft. While certain individual components of this chapter may not have exhibited significant novelty in isolation, it is the cumulative effort invested in the creation of the fault-tolerant software framework that collectively represents a novel and impactful approach towards enhancing the reliability of CubeSat platforms.

The BSF was written by the candidate to facilitate rapid mission-concept-to-orbit and maximise code reliability. The developed framework provides an alternative to using existing software solutions or proprietary SDKs for the BSP. The development approach means that the framework can be easily adapted to support changing mission requirements without compromising its reliability or introducing bugs, the benefits of which have been seen during the development of the Binar-2, 3 and 4 satellites. The BSF can be used in subsequent missions without incurring ongoing licensing costs, and its modular design allows for easy updates and improvements. Moreover, the availability of documentation and source code means that new developers can easily learn how to develop software for space systems using the BSF, which promotes knowledge transfer and skills development within the BSP. Overall, the BSF exemplifies the advantages of developing software in-house while adhering to industry best practices in software design.

REFERENCES

[69] I. Latachi, T. Rachidi, M. Karim, and A. Hanafi, "Reusable and Reliable Flight-Control Software for a Fail-Safe and Cost-Efficient Cubesat Mission: Design and Implementation," en, *Aerospace*, vol. 7, no. 10, p. 146, Oct. 2020, ISSN: 2226-4310. DOI: 10.3390/aerospace7100146. [Online]. Available: https://www.mdpi.com/2226-4310/7/10/146.

[104] M. Doyle, A. Gloster, C. O'Toole, J. Mangan, D. Murphy, R. Dunwoody, M. Emam, J. Erkal, J. Flanaghan, G. Fontanesi, F. Okosun, R. R. Nair, J. Reilly, L. Salmon, D. Sherwin, J. Thompson, S. Walsh, D. de Faoite, U. Javaid, S. McBreen, D. McKeown, D. O'Callaghan, W. O'Connor, K. Stanton, A. Ulyanov, R. Wall, and L. Hanlon, "Flight Software Development for the EIRSAT-1 Mission," in *Proceedings of the 3rd Symposium on Space Educational Activities*, 2020, pp. 157–161. DOI: 10.29311/2020.39. [Online]. Available: http://arxiv.org/abs/2008.09074.

[135] J. Martin, "Data-Base Action Diagrams," in *Managing the Data-base Environment*, Prentice-Hall, 1983, pp. 375–411, ISBN: 978-0-13-550582-3.

[136] The Pigweed Authors, *Pigweed*, Mar. 2023. [Online]. Available: `https://github.com/google/pigweed`.

[137] A. Zeedan and T. Khattab, "CubeSat Communication Subsystems: A Review of On-Board Transceiver Architectures, Protocols, and Performance," *IEEE Access*, vol. 11, pp. 88 161–88 183, 2023, ISSN: 2169-3536. DOI: `10.1109/ACCESS.2023.3304419`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/10224067` (visited on 10/27/2023).

[138] S. Cheshire and M. Baker, "Consistent overhead byte stuffing," *IEEE/ACM Transactions on Networking*, vol. 7, no. 2, pp. 159–172, Apr. 1999, ISSN: 1558-2566. DOI: `10.1109/90.769765`.

CHAPTER 4

Experimental Results and Modelling of the Binar-1 CubeSat On-Board Power Management in Thermal Vacuum Conditions

Stuart R. G. Buchan[1], Fergus W. Downey[1], Kyle McMullan[1], Benjamin A. D. Hartig[1], Eduardo Trifoni[2], Joice Mathew[2], Phil A. Bland[1], Jonathan Paxman[3], and Robert M. Howie[1]

[1]Space Science and Technology Centre, Curtin University, Perth, Australia

[2]Advanced Instrumentation and Technology Centre, Research School of Astronomy and Astrophysics, Australian National University, Cotter Road, 2611 ACT, Canberra, Australia

[3]Department of Mechanical Engineering, Curtin University, Perth, Australia

This article is in preparation for journal submission. The formatting and referencing style have been modified for presentation in this thesis.

AUTHORSHIP DECLARATION

Article Title: Experimental Results and Modelling of the Binar-1 CubeSat On-Board Power Management in Thermal Vacuum Conditions

AUTHOR CONTRIBUTIONS

STUART BUCHAN

Conceived and developed the software used for testing Binar-1 and contributed to conducting the experiment. Processed the experimental data to create the CubeSat onboard power system models and verified the models produced. Wrote the manuscript.
Manuscript contribution: 80%

FERGUS W. DOWNEY

Conceived and developed the Binar-1 hardware, including the electrical power system under test for the experiment. Contributed to conducting the experiment. Reviewed and provided critical feedback on the manuscript.
Manuscript contribution: 5%

KYLE MCMULLAN

Created a thermal model of Binar-1 in Ansys, which was used to verify the models. Assisted in analysing the thermal data. Reviewed and provided critical feedback on the manuscript.
Manuscript contribution: 5%

BENJAMIN A. D. HARTIG

Provided project administration. Reviewed and provided critical feedback on the manuscript.
Manuscript contribution: 1%

EDUARDO TRIFONI

Operated the Wombat vacuum chamber during the experiment. Encouraged the candidate to investigate thermal modelling of the satellite. Reviewed and provided critical feedback on the manuscript.

Manuscript contribution: 3%

JOICE MATHEW

Assisted in operating the Wombat vacuum chamber during the experiment. Reviewed and provided critical feedback on the manuscript.

Manuscript contribution: 3%

PHIL A. BLAND

Provided research guidance. Reviewed and provided critical feedback on the manuscript.

Manuscript contribution: 1%

JONATHAN PAXMAN

Provided research guidance. Reviewed and provided critical feedback on the manuscript.

Manuscript contribution: 1%

ROBERT HOWIE

Provided research guidance. Reviewed and provided critical feedback on the manuscript.

Manuscript contribution: 1%

ABSTRACT

The electrical power system on-board a spacecraft has a crucial role in the success of a mission. Central to this however, the health of the lithium-ion cells frequently used in CubeSat applications degrades when exposed to sub-zero temperatures. With the external faces of a spacecraft in the Low Earth Orbit environment expected to reach -40°C, the necessity of battery heating in the success of a mission is clear when using this battery technology. It therefore became a key element in the design choices for Binar-1, Western Australia's first locally made satellite. The decision was made for Binar-1 that this functionality be delegated from the flight computer to analog circuitry. This is not without risk as the battery heating circuitry composes one of the largest loads on the electrical power system, increasing the possibility of a low-power state occurring on the spacecraft. Thermal simulation can offer insight into battery heater usage, but is a computationally expensive process reserved for high-end desktop computers. This highlights a gap for computationally inexpensive models able to run on-board a spacecraft, predicting in advance when these low power states are to occur. Coupled with software capable of suggesting alterations to a mission operations plan, CubeSats can demonstrate a proof-of-concept approach for safe operations in a communications limited environment when encountered with an emergency state that cannot wait for input from a ground operator. Using data collected from a thermal vacuum experiment in the National Space Test Facility at the Australian National University, the response of the electrical power system and battery heaters in Binar-1 was characterised from being subject to the conditions of Low Earth Orbit. This information is used in the creation of two predictive models of when the battery heaters are to be enabled, influenced by the thermal state inside and external to the spacecraft. From a simulated orbital analysis and a further vacuum chamber test, a model is selected to be used on the next Binar spacecraft to allow the satellite to optimally postpone tasks when the usage of the battery heaters are to be prioritised.

4.1. INTRODUCTION

Operating from the Space Science and Technology Centre at Curtin University, the Binar Space Program is enabling a new Western Australian space capability. The primary output from the program, the BCM, is a highly integrated custom CubeSat motherboard that comprises a primary and redundant flight computer, a Global Positioning System (GPS), ADCS, on-board storage and EPS. The BCM gained flight heritage for the first time onboard Binar-1 in 2021, and is now being integrated into the next missions from the program.

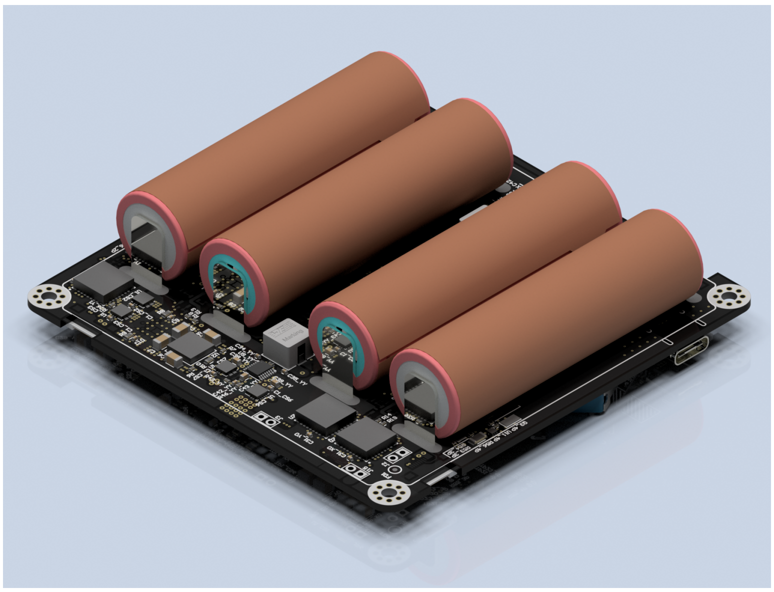


FIGURE 4.1: The top side of the Binar CubeSat Motherboard, detailing the EPS. The four batteries that power the EPS are wrapped by flexible printed circuit board (PCB) battery heaters, triggered by analog circuitry to enable when the battery temperature drops below 5°C.

As the maiden flight of the custom BCM was on Binar-1, the mission was intended to be a technology demonstrator to show the suitability of the Binar Space Program's hardware in a Low Earth Orbit (LEO) environment. Downey et al. [139] provides an overview of the Binar-1 mission and its objectives. The EPS powering the BCM comprises four lithium-ion batteries, seen in Figure 4.1. Each of the batteries has a dedicated flexible PCB heater wrapped over the external casing for thermal maintenance. This is a requirement as the number of discharge cycles and capacity of lithium-ion cells, frequently used in CubeSat applications, have been shown to be dramatically reduced

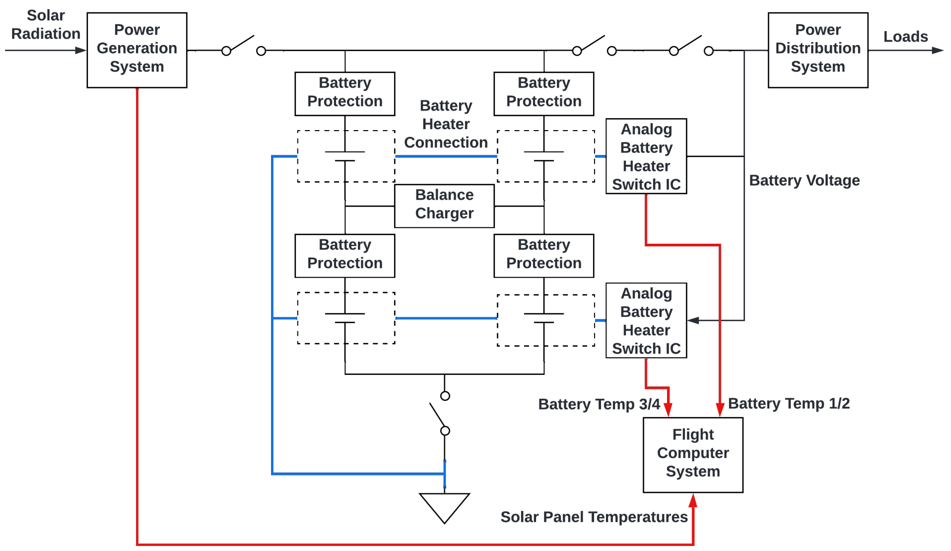


FIGURE 4.2: A high level block diagram of the EPS on Binar-1. Highlighted in blue is the electrical connections between the parallel heating circuits, represented by the dashed boxes around the power sources. Each circuit has an independent analog battery heater integrated circuit for operation. The temperature feedback into the flight computer from the solar cells and the battery heating circuitry is shown in red.

from exposure to conditions below 0°C [140][141]. Rather than activating the thermal maintenance circuitry at 0°C, a safety factor was added on the BCM for the two groups of two heaters to be enabled when the temperature in the battery compartment drops below 5°C. The resistor-programmable temperature switches are set to disable when the temperature set point reaches 10°C to conserve energy. Figure 4.2 provides a high level block diagram of the EPS on the BCM installed in Binar-1. The battery heating system is separated from the flight logic, being instead implemented through analog circuitry. This was done to ensure the heaters can be enabled as soon as required. As the circuitry warms the batteries through power dissipation, the battery heating circuitry composes one of the largest loads on the EPS. This poses a risk as without control over this load the spacecraft could potentially encounter an electrical low-power state if the heaters are enabled during a period of high current draw on-orbit. This highlights the necessity for on-board autonomy for the spacecraft to predict in advance when these low power states are to occur, without relying on input from an operator.

4.2. ON-BOARD AUTONOMY

On-board autonomy for spacecraft has aided in the success of numerous historical space missions [142]. It has been successfully demonstrated to be an invaluable resource in a communications denied or limited environment. Straub et al. summarise that in these situations, missions may need to hand control over to autonomous software to handle an emergency response that cannot wait for input from ground operators [143].

The Curiosity Mars rover relied extensively on autonomy during the EDL due to a communications link delay that extended beyond the predicted entry time [144]. Even with a consistent bent pipe link with Earth during EDL, the subsequent Perseverance rover likewise had to rely on autonomy due to the extensive link delay [145]. Both spacecraft autonomously responding to inputs from on-board sensors allowed the rovers to land safely in a situation where ground operators could not provide the real-time input that was needed.

Even without the extensive time delay induced by deep space missions, the importance of autonomy is paramount in a mission in the case where communications are lost. During the landing phase of the Beresheet Lunar mission, a hardware fault requiring a component reset occurred. However, a loss of communications with the ground station prevented reset requests from reaching the spacecraft. By the time communications were restored, the altitude of the lander was too low to slow the descent, and the spacecraft was lost [146][147].

On-board autonomy has also been shown historically to successfully suggest changes to an operations plan. The first Mars rover, Sojourner, relied on this capability as communication could only be made with ground operators twice per Sol. Using on-board autonomy, the spacecraft could infer which commands from the previously sent operations plan could safely be executed when encountering terrain difficult to traverse [148]. Extending on the responsibilities for autonomy on spacecraft, the Deep Space One mission became the first spacecraft to fly an on-board mission planner. The successful mission was able to demonstrate that if the pre-set operations plan is not able to be executed, the spacecraft could respond by autonomously suggesting an alternate solution in an agile manner [60].

Although historical missions have demonstrated that autonomous capabilities on spacecraft can provide a higher level of return than those requiring a ground-based link [142], its extensive use has been met with some scepticism. A 2013 survey of the planetary science community found that there is apprehension toward the acceptance of autonomous control of spacecraft [143]. This is understandable, yet necessary in the context of expensive planetary missions. This highlights a possible application for CubeSats. CubeSats have demonstrated operation in environments extending beyond LEO, whilst maintaining a fast development cycle and low cost [149]. Prior to more widespread adoption on future interplanetary missions however, Feruglio et al. stress that CubeSat mission autonomy needs to be improved [150]. Often described as a 'test bed' for larger missions, the CubeSat form factor can also be leveraged to trial mission autonomy techniques with a significant reduction in risk over larger missions. As the computers powering small spacecraft become smaller, cheaper and more powerful, applications requiring significant processing power are becoming possible in a small spacecraft form factor [143].

To this aim, the author used data obtained from a space environment experiment to develop a model to be used for on-orbit power management predictions. Featuring as a software payload on the upcoming Binar 2, 3 and 4 missions, the inference from the model will allow the spacecraft to autonomously suggest changes to the operations plan if the possibility of a low power state is deemed to occur. This paper details the creation of the model, and the results of preliminary testing.

4.3. TESTING PROCEDURE

The Binar-1 engineering model was placed in the centre of the Wombat XL thermal vacuum chamber at the National Space Test Facility (NSTF) located at the Mount Stromlo campus of the Australian National University. The Wombat XL is a 5.76 cubic meter thermal vacuum chamber used for the testing of space-based instruments and small satellites. It features a liquid nitrogen fed platen and shroud system for thermal

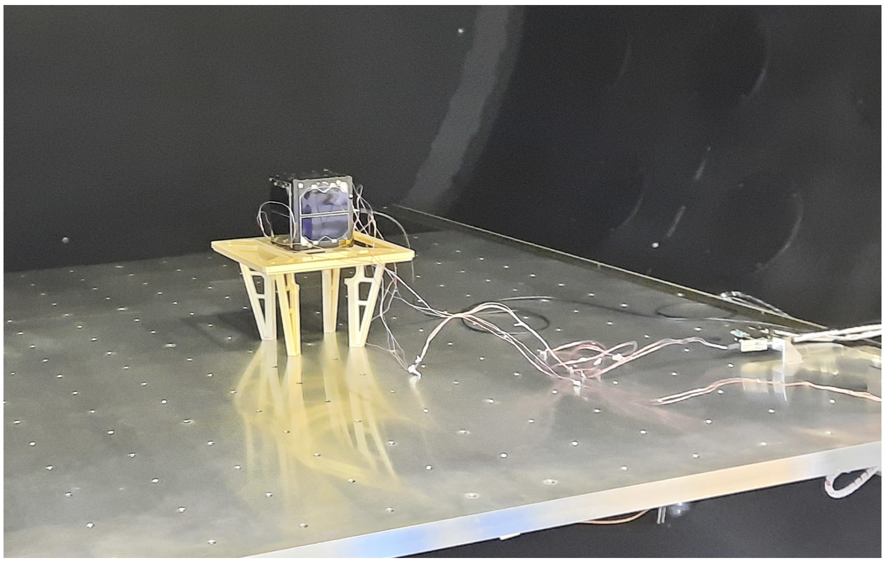


FIGURE 4.3: The Binar-1 engineering model placed in the Wombat XL vacuum chamber with thermal probes attached.

cycling from -170°C to +150°C at a rate of 3°C per minute. The support stand for the satellite was made of 3D printed Ultem to limit thermal conduction as much as possible from the platen shown in Figure 4.3. Seven chamber supplied thermal sensors were placed on the exposed faces; one on each of the solar panels in the bottom right corner (next to the surface mounted thermal sensors on the panels), and three on the top face of the satellite (shown circled in Figure 4.4). These sensors were logging with a one second period, coinciding with the internal logging frequency of the sensors inside Binar-1. Internal to the satellite, battery and regulated voltage and current, X and Y axes solar panel voltage and current, individual solar panel temperature, and battery compartment temperature were recorded.

One of the primary objectives for testing Binar-1 at the NSTF was to understand and characterise the functionality of the analog battery heating circuitry in LEO conditions. From calculations done prior to the experiment, Binar-1 was expected to undergo a worst-case temperature swing between -40°C and +65°C on the external faces during an orbit. Initially it was decided to adopt a schedule for the Wombat XL to dwell at -40°C and +90°C in the attempt to have the surface of the solar panels reach -30°C and +80°C respectively to stress test the satellite. The inner chamber temperature set points were set to be -50°C and +100°C with a thermal gradient of 3°C/min, before stepping to the desired dwell temperature as the platen and shroud approached the set points, as

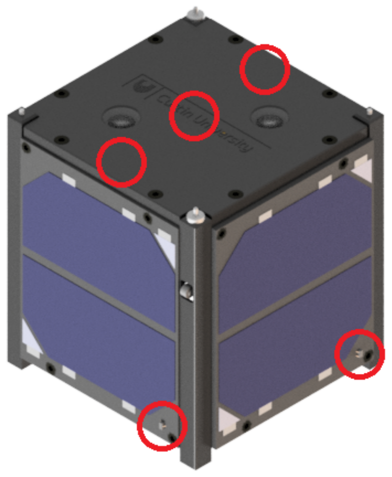
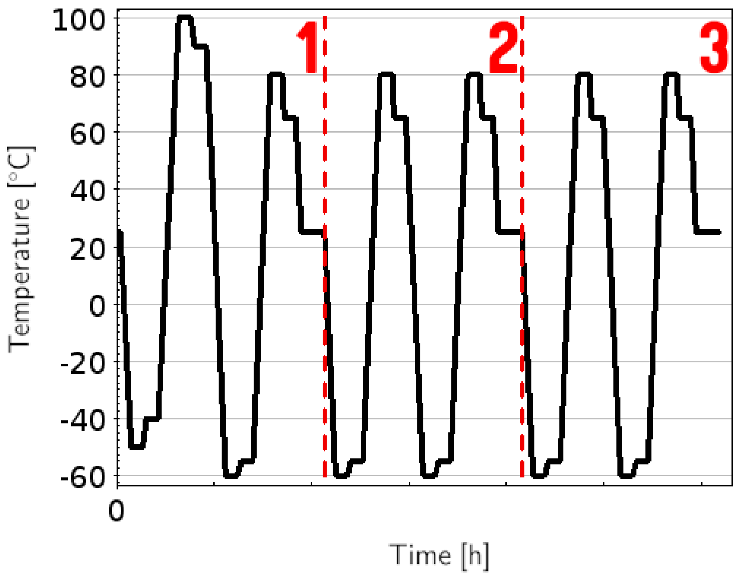


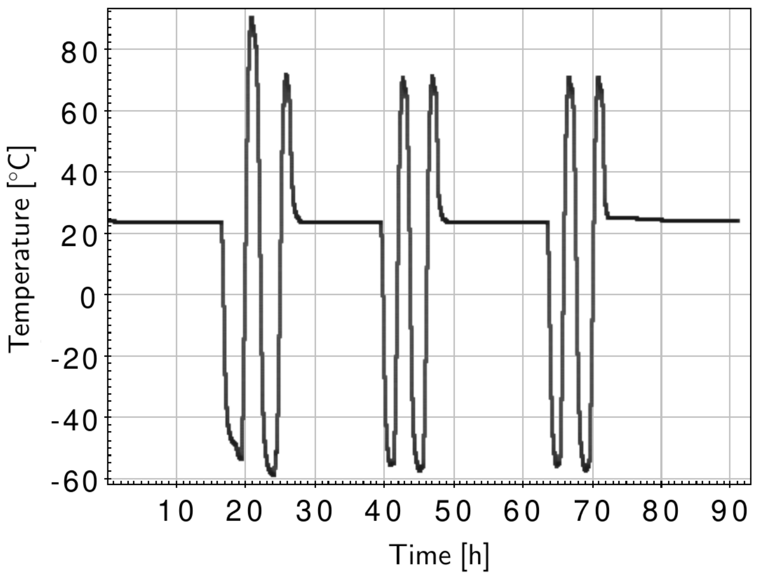
FIGURE 4.4: A render of Binar-1 showing the location of the thermal sensors during the experiment. Three sensors were placed on the bottom face of the satellite facing the platen (shown on the top of the satellite in this figure), and one was placed in the corners of all four sides of the solar panels next to the surface mounted panel thermal sensors (only two shown in this figure).

seen in Figure 4.5(a). Figure 4.5(b) shows the experimentally recorded average platen and shroud temperature during the experiment.

The chamber pressure was brought to 6E-7 Torr after an initial overnight pump-down. During the first cold swing, the platen and shroud temperatures were varied, using the feedback from the thermal sensors on the spacecraft to achieve the desired temperature of -30°C on the surface of the panels. Following the first cold dwell, reaching the desired +80°C set point on the surface of the cells took longer than anticipated. This caused the batteries in the spacecraft to reach the temperature limit. To maintain optimal battery health, the maximum internal temperature was intended not to exceed +65°C. However, the maximum recorded temperature of the batteries reached +70°C during the first hot swing, exceeding this range. After completing the first thermal cycle, it was decided to amend the temperature profile to change over -55°C to +65°C, with the intention of having the surface of the cells reach -45°C to +55°C respectively. The battery heating circuitry could then be stress tested further by allowing a colder external temperature to be reached, whilst protecting the integrity of the battery health on the hot swings. The remainder of the testing schedule continued with these updated values, as seen from the profile in Figure 4.5.



(A) The proposed temperature profile for the experiment, comprising six thermal cycles segmented into three days. The platen and shroud temperatures were varied to achieve a thermal gradient of 3°C/min, before stepping to the desired dwell temperature as the platen and shroud approached the set points. The proposed temperature profile remains at each set point for 30 minutes.



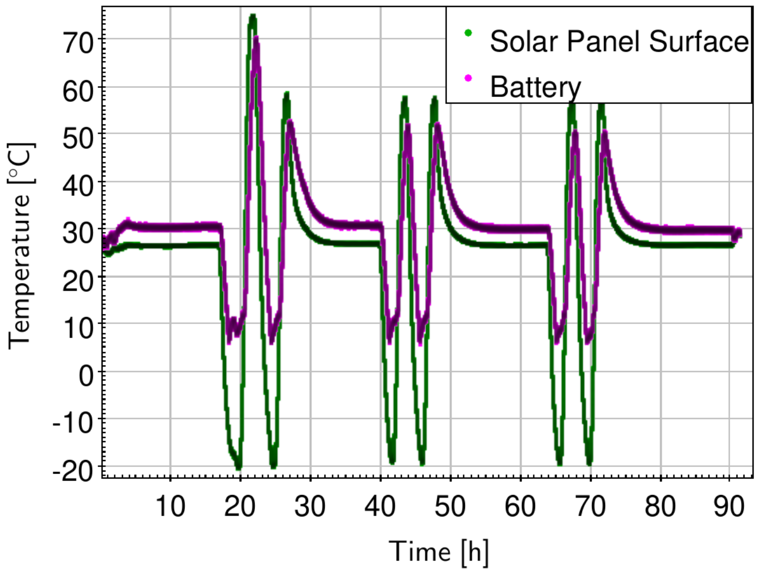
(B) The experimentally recorded average platen and shroud temperature during the experiment.

FIGURE 4.5: The proposed temperature profile for the experiment and the experimentally recorded average platen and shroud temperatures. Initially the set points for the vacuum chamber were to change over -40°C to +90°C, attempting to get the surface of the panels to -30°C and +80°C respectively. On the first hot swing however, the batteries reached +70°C, far exceeding the safe battery temperature of +65°C. Due to this, the thermal profile was amended to change over -55°C to +65°C, allowing for stress testing the battery heating circuitry whilst maintaining safe battery temperatures. Between test days, the chamber was brought to an ambient temperature.

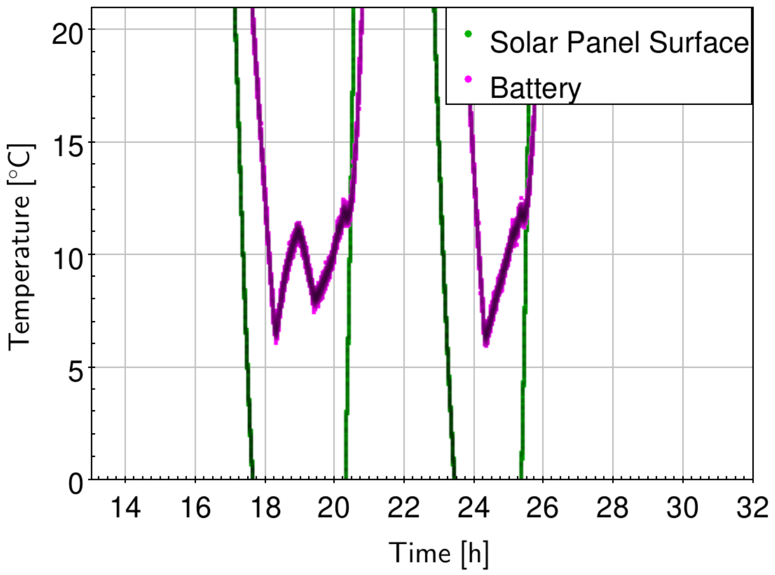
4.4. RESULTS

4.4.1. BATTERY HEATING SYSTEM

The recorded temperature of the batteries internal to the spacecraft is plotted against the solar panel surface temperature in Figure 4.6(a), showing that the battery temperature did not drop below 6°C during the six thermal cycles performed in the chamber. This demonstrates that the analog battery heating circuitry functioned successfully. Figure 4.6(b) shows a magnified view of the first two cold swings, showing the response of the spacecraft internal battery temperature to the temperature at the surface of the solar cells over a roughly ten hour period. The spacecraft internal battery temperature lags the solar panel surface temperature, dropping until it reaches 6°C. At this point the battery heaters activate, warming the batteries until they disable at 12°C. The enable and disable temperature differing slightly from the desired set points can be attributed to inaccuracies in the resistors used in the analog logic, the temperature sensors, and the analog to digital converter on-board the flight computer. As the inaccuracy has caused the usage of the heaters to shift in a more conservative direction however, this is deemed acceptable. As the chamber is dwelling at -50°C, the spacecraft eventually cools down enough to re-enable the battery heaters. The heating process again disables at the cut-off temperature, coinciding with the chamber temperature increasing to the hot dwell. After the temperature lag ceases, it is observed that the temperature begins to increase again just after the 20 hour mark. The operation of the battery heaters is shown (Figure 4.6(a)) to consistently repeat during the remaining thermal cycles in the vacuum chamber. This invokes high confidence in the heaters to operate as expected in a LEO environment, keeping the batteries in the optimal temperature range and extending the on-orbit lifetime. As the heaters maintain the battery temperature through power dissipation, their frequent usage, though required, poses a significant load on the EPS.



(A) The temperature of the batteries on Binar-1 plotted against the temperature of the solar panel surface over the experiment duration.



(B) Magnified view of the temperature profile during the first two cold swings, detailing the temperature maintenance of the batteries from the operation of the heaters.

FIGURE 4.6: The temperatures of the batteries on-board Binar-1 during the experiment plotted against the temperature of the solar panel surface. The battery heating circuitry prevents the temperature of the batteries from dropping below 6°C. The heaters are enabled twice in the first cold swing as the chamber was held at the cold set point for a longer duration than the second swing.

4.4.2. ELECTRICAL POWER SYSTEM

The electrical power system on Binar-1 consisted of four 18650 lithium-ion batteries in a two series and two parallel (2S2P) configuration providing approximately 50 Wh of power with a nominal voltage of 7.4 V. Fully charged, the batteries supply 8.3 V to the motherboard. During the initial overnight pump-down in the Wombat XL, the battery voltage was brought to full charge via a power harness to start the experiment. Binar-1 remained connected to the power harness until the third day of testing.

Figure 4.7 depicts the battery voltage sharply decreasing each time the battery heaters are enabled. The connected power harness brought the batteries back to full charge for the first two days.

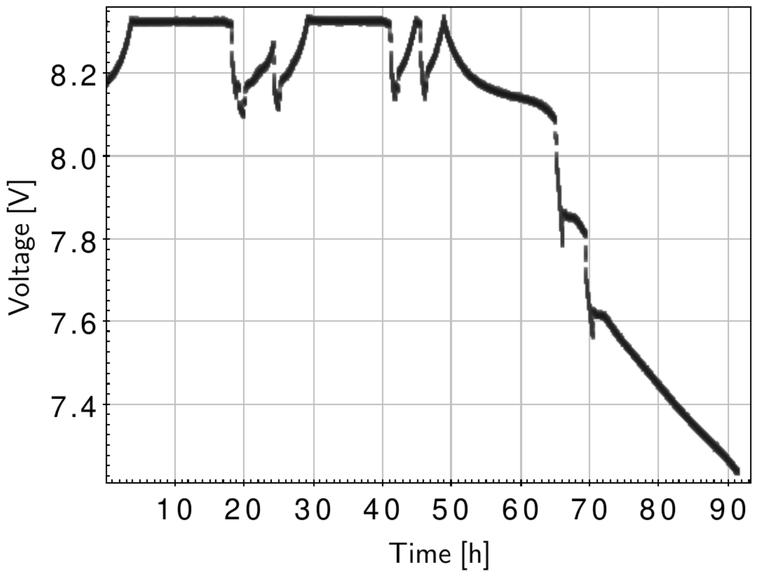
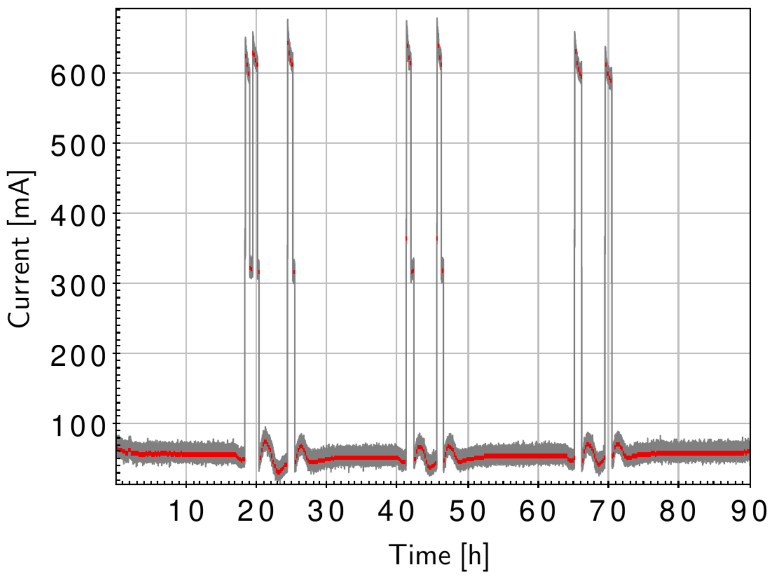
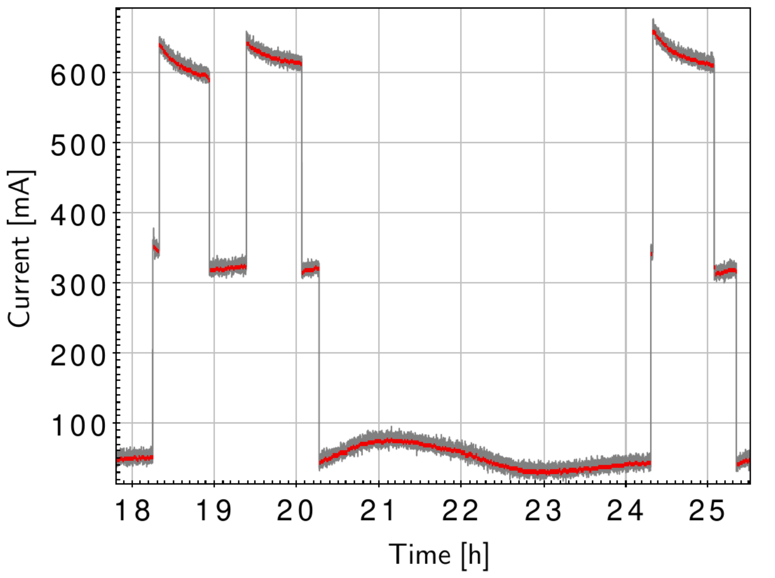


FIGURE 4.7: The change in spacecraft battery voltage during the experimentation in the Wombat XL. Initially plugged into the power harness, the batteries charged completely. During sections of high power usage, the voltage profile is seen to drop before charging again. Roughly 50 hours into the experiment, the power harness was removed, allowing the batteries to discharge.

The batteries are shown to have a consistent current draw at approximately 60 mA in Figure 8(a) prior to the battery heating circuitry being enabled. During usage, the current spikes to over 600 mA, explaining the drop in battery voltage seen in Figure 4.7. The total current draw from the batteries is seen to slope downward during heater usage, attributed to the generated heat increasing the resistivity of the circuit. It should



(A) The change in battery current over the experiment duration. The current draw is shown to be consistent for all instances of heater usage.



(B) Magnified view of the battery current profile during the first two cold swings, detailing the magnitude of the power increase during battery heater operation. When enabled, the current spikes roughly ten times higher than the regular operation. Two plateaus are present in the current response, one at roughly 600 mA, and one at roughly 300 mA. This can be attributed to the two groups of battery heaters being enabled and disabled independently. The battery heaters previously mentioned being enabled twice during the first cold swing is further shown here with two corresponding current peaks.

FIGURE 4.8: The change in the battery current over the experiment duration. A rolling average has been displayed in red over the data to reduce the noise.

be noted that the current drop is minimal as the battery voltage is also dropping during heater usage. Figure 4.8(a) shows that the first thermal cycle involves three current peaks, as opposed to the two peaks seen during the remaining thermal cycles. Figure 4.8(b) shows a magnified view of the current draw during this thermal cycle. The extra peak can be attributed to the first cold dwell lasting for a longer duration than the remaining dwells, resulting in the battery heating circuitry being used twice. As the thermal management of the EPS is done through two groups of two heaters, it is possible for the heaters to be enabled independently. This is clear in Figure 4.8(b) where a plateau at approximately 300 mA is present on multiple occasions. The variance in the time the current draw remains at these plateaus shows that the heat generated from one of the independent battery heating circuits has an influence on the operation of the other. From the data collected in the chamber however, it is apparent that a single battery heating circuit is not sufficient to keep the entire battery compartment at an optimal temperature. It has been observed that the operation of the heaters imposes a significant power draw on the EPS, and as such limits the power available for other systems. On average, the budget for the power storage system estimates a power usage of 0.370 Whr/orbit. With the satellite estimated to use a total of 1.493 Whr/orbit, this equates to roughly 25% of the power used by the satellite per orbit. Whilst in the Wombat XL, Binar-1 was running low power tasks. In conjunction with the increased power draw, it is expected that a power intensive task such as payload operation could potentially cause a low power state on-orbit. Therefore, it is of crucial importance that usage of the battery heaters can be predicted so that power utilisation can be managed.

4.5. THERMAL MODELLING

Alongside physical testing, characterising the stresses induced on a spacecraft from the thermal environment of LEO is possible through the use of thermal simulation software. This is often an integral part of the design phase of a satellite as information obtained regarding the thermal properties of the spacecraft can help influence decisions made when finalising the design. A thermal model of Binar-1 was implemented in Ansys 2021 Thermal Analysis Software to assess the applicability of the Binar CubeSat

TABLE 4.1: A summary of the values of emissivity used in the Binar-1 thermal modelling, adapted from [151].

	Emissivity		
Material	**Infrared**	**Visible**	**Total Hemispheric**
Wombat XL Shroud			0.85
Wombat XL Platen			0.15
Aluminium	0.86	0.86	
Solar Panels	0.8	0.86	
PCBs	0.6	Not Exposed	
PV Cell	0.85	0.8	
Polyimide	0.84	Not Exposed	

TABLE 4.2: The simplifications made in the creation of the Binar-1 thermal model.

Simplification	**Impact**	**Justification**
No energy is lost to free space; Radiation transfer for exposed faces in the simulation is limited to surface-to-surface.	Minor	Reduces simulation time.
Specific heat capacity of aluminium material was reduced by 10% to account for the holes in the frame of the spacecraft not being modelled.	Moderate	Reduces nodes in the mesh.
As the thermal properties of all structural materials are similar to aluminium alloy, they are all given the properties of the Ansys defined material 'Aluminium Alloy'.	Minor	Reduces setup time between thermal simulations.

TABLE 4.2: The simplifications made in the creation of the Binar-1 thermal model (Continued).

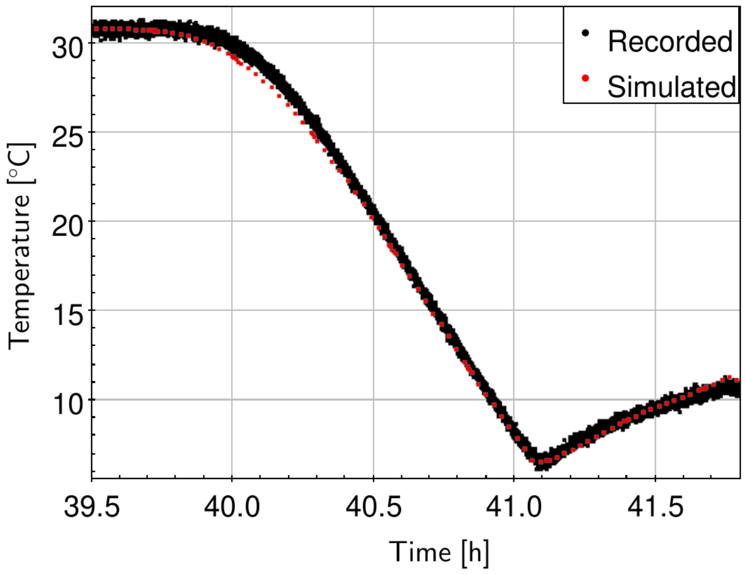
Simplification	Impact	Justification
As the radiative properties of the external non-structural materials are similar to black anodised aluminium, they are all given the properties of black anodised aluminium as measured by [151].	Minor	Reduces setup time between thermal simulations.
Battery cells assumed to be thermal properties of lithium-ion as measured by [152].	Moderate	Compensating for inhomogeneity in cells significantly increases simulation complexity, and was deemed beyond the scope of this model.
Battery heaters are given the thermal properties of the most abundant material, Polyamide.	Minor	Compensating for copper traces in the heaters significantly increases simulation complexity, and was deemed beyond the scope of this model.
PCBs given the thermal properties of the most abundant material, FR4 as measured by [153].	Moderate	Compensating for copper traces and integrated circuits on the PCBs significantly increases simulation complexity, and was deemed beyond the scope of this model.

TABLE 4.2: The simplifications made in the creation of the Binar-1 thermal model (Continued).

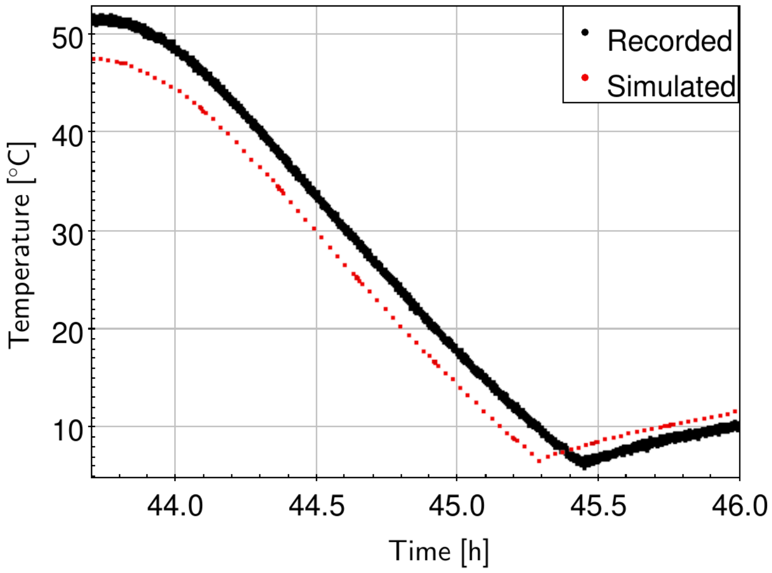
Simplification	Impact	Justification
Solar cells are given the thermal properties of the most abundant material, gallium arsenide as measured by [154].	Minor	Compensating for cover glass and metal tabs on solar cells significantly increases simulation complexity, and was deemed beyond the scope of this model.
General approximate value for contact thermal conductance found through trial and error and set for all surface-to-surface contacts.	Significant localised errors	Getting true values for all surface-to-surface contacts involves disassembly of the satellite, followed by significant testing of each material. Deemed beyond the scope of this model.

bus in a LEO-like environment. Ansys 2021 was selected due to its prevalence in thermal modelling for spacecraft design and its availability to the Binar Space Program. The thermal model was generated after the launch of Binar-1 to enable improvement of future launches reusing the majority of the bus design. The thermal profile used during the testing at the NSTF is useful for the verification of the Binar-1 thermal model. A summary of the values of emissivity used for the thermal radiative properties of the Binar-1 thermal model is outlined in Table 4.1. Furthermore, Table 4.2 shows a summary of the assumptions made in the creation of the Binar-1 thermal model. The third and fourth cold swings (second day) of the experimentation done at the NSTF were simulated to verify the accuracy of the Binar-1 thermal model. These swings were selected as the input conditions were identical to the fifth and sixth swings.

The thermal response of the battery temperature of the Binar-1 thermal model to the simulated Wombat XL chamber temperature is plotted in Figure 4.9 against the recorded



(A) The recorded temperature of the third cold swing plotted against the output of the Binar-1 thermal model. The initial thermal condition for the simulation was an ambient steady state. The thermal simulation closely matches the data recorded in the Wombat XL.



(B) The recorded temperature of the fourth cold swing plotted against the output of the Binar-1 thermal model. The thermal simulation of Binar-1 leads the trend recorded in the Wombat XL, attributed to the initial thermal condition for the simulation starting at the coldest point of the preceding cold dwell.

FIGURE 4.9: A comparison between the experimental recordings of the third and fourth cold swings in the Wombat XL and the thermal response from simulating the Binar-1 thermal model. The discrepancy in accuracy can be attributed to the initial thermal conditions. The simulation of the third cold swing used an ambient steady state starting condition, whereas the simulation of the fourth cold swing used the coldest point of the preceding cold swing. This was done to achieve a realistic initial thermal gradient for the hot-to-cold swing.

values from the experiment in the chamber. Figure 4.9(a) shows that the temperature of the batteries in the Binar-1 thermal model closely match the experimental data when entering a cold swing from ambient conditions. The response seen in Figure 4.9(b) shows less accuracy in the thermal modelling when entering a cold swing from a hot environment, however shows a similar gradient to the recorded data. This inaccuracy can partly be attributed to the assumptions outlined in Table 4.2, introducing errors in the simulated satellite. Furthermore, the temperature condition preceding the response seen in Figure 4.9(a) was at a steady state, as this corresponded to the first cold swing after an overnight lull between testing days. The initial temperature condition for the response shown in Figure 4.9(b) however was the coldest point of the preceding cold dwell. This was done to achieve a realistic initial thermal gradient for the hot-to-cold swing, but resulted in a discrepancy between the simulated and experimental cold swing starting temperatures. Showing a simulated thermal response with high similitude to the experimental data, the Binar-1 thermal model can be subject to varying thermal inputs with confidence that the response seen will be similar to that of the physical satellite. This can aid in characterising the spacecraft in potential mission orbits, and will influence the design decisions for subsequent Binar spacecraft.

4.6. POWER MANAGEMENT MODELLING

As the battery heating system is hardware controlled, the BCM is not able to control its operation through software. This can be problematic as the total battery power draw during heating is approximately ten times higher than regular operation (Figure 4.8), capable of rapidly depleting the batteries. With limited power available on-board, this stresses the need for a prediction of when the heaters are to be operational. Thermal simulation, however, is a computationally expensive process, even when performed on a desktop computer. The drastically reduced capabilities of the primary flight computer on the BCM would prove infeasible to effectively utilise the simulation on orbit. Hence, simplified linear modelling is employed in the prediction of battery heater usage instead. This section details the creation of this a model, and discusses its applicability for autonomously suggesting changes to the mission plan on-orbit

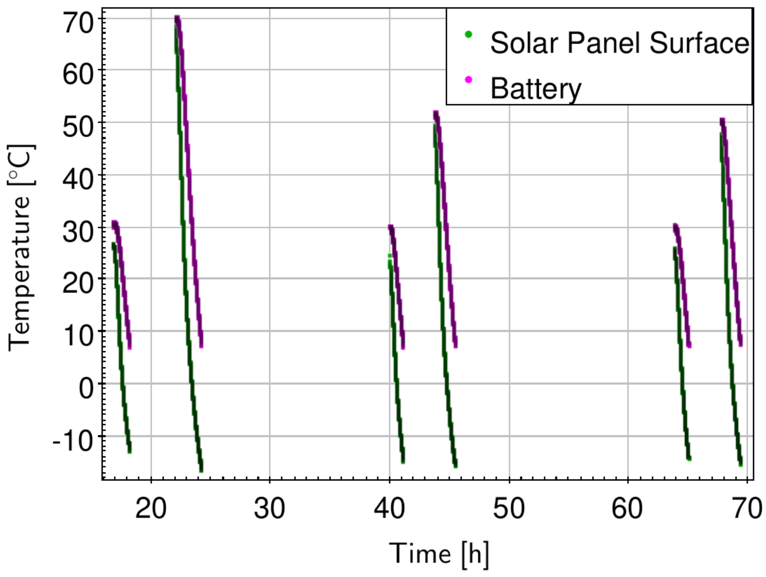


FIGURE 4.10: Isolated view of the temperature response of the spacecraft to entering a cold swing in the Wombat XL. In each of the six groups of trends, the right (pink) trend shows the thermal response inside the battery compartment at the core of the satellite. The left (green) trend shows the thermal response of the surface of the solar cells. The segments have been taken from when the surface of the cells begin to respond to a cold cycle and stop when the battery heaters are enabled.

without needing to communicate with the ground.

From the data collected on the internal and external spacecraft temperature sensors during testing at the NSTF, the sections leading into a cold swing can be isolated to assess the relationship between the readings prior to the battery heaters being enabled. Figure 4.10 shows the temperature on the surface of the solar panels and at the spacecraft batteries for each of the experiment cold swings. As the temperature inside of the spacecraft is influenced by the temperature measured at the surface of the solar cells, a model can be generated that relates the difference in the internal and external temperatures to the rate of change of temperature internal to the spacecraft at the batteries. The internal temperature is taken to be the average between the two thermal sensors in the spacecraft battery compartment. Likewise, the external temperature from the average of the four solar panel thermal sensors. A ten minute rolling average of these readings was used for smoothing and a linear approximation was trialled due to its low complexity to assess the fit (Figure 4.11):

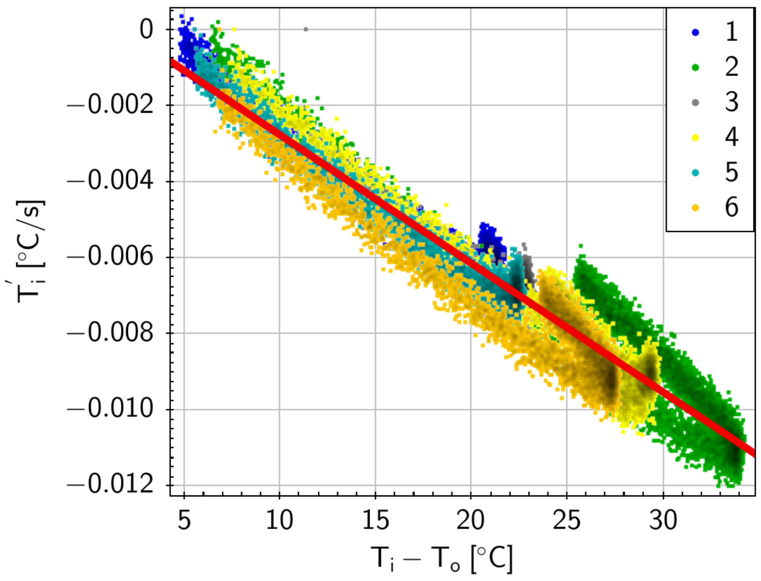


FIGURE 4.11: The relationship between the difference in the internal and external temperatures of the spacecraft and the rate of change of the temperature internal to the spacecraft at the batteries for each of the cold swings. A linear fit has been applied to the data. All six series shown in Figure 4.10 are superimposed here, showing a consistent trend.

$$T_i' = -3 * 10^{-4}(T_i - T_o) + 6 * 10^{-4} \qquad [^\circ\mathrm{C\,s^{-1}}] \quad (4.1)$$

Where T_i and T_i' refer to the spacecraft battery temperature and the corresponding rate of change, and T_o refers to the temperature at the surface of the solar cells. The model given in Equation 4.1 has a coefficient of determination of 0.8833. This shows a relatively strong linear fit in the region between entering a cold swing and enabling the battery heaters. The present error can be partially attributed to noise in the sensors recording the temperature, and from the data processing to obtain the relationship. Moreover, the lower right of Figure 4.11 shows a particularly noisy region of the data present in all of the superimposed series. This can be attributed to the difference between the internal and external temperatures starting to re-converge after they separate, showing that the relationship is not bijective. However, the corresponding rate of change to the points that re-converge seem to fit with the linear approximation with minor error. Knowing that the analog battery heaters are programmed to turn on at 6°C, Equation 4.1 can be used to give a reasonable prediction of the time until the battery heaters are enabled.

$$T_{enable} - T_i = T_i^{'} * t_{heaters\_on} \qquad [^\circ\text{C}] \quad (4.2)$$

$$\therefore t_{heaters\_on} = \frac{T_{enable} - T_i}{-3 * 10^{-4}(T_i - T_o) + 6 * 10^{-4}} \qquad [\text{s}] \quad (4.3)$$

Where T_{enable} refers to the battery heaters enable condition, 6°C. The model given in Equation 4.3 is propagating the instantaneous rate of change to predict when the battery heaters will be enabled. Due to this, at the start of the cold swing when the delta between the internal and external temperatures is the smallest, the model will be unstable when the corresponding rate of change of the internal temperature is minimal. This is evident in Figure 4.12, where the time taken between the surface of the cells responding to a cold cycle and the battery heaters enabling for the fifth cold swing is plotted against the predictive model. As the delta between the internal and external temperatures increases and the rate of change becomes more constant, the accuracy begins to increase. Below a 20°C temperature difference between the battery temperature and the heater enable temperature, the model can reliably predict the time remaining before the heaters were enabled in the Wombat XL chamber.

In an attempt to compensate for the inaccuracies introduced from propagation of small rates of change, a second linear approximation was produced relating the difference between the temperature at the surface of the solar cells and the internal batteries, and the second derivative of the internal temperature response with respect to time (Figure 4.13):

$$T_i^{"} = 2 * 10^{-7}(T_i - T_o) - 7 * 10^{-6} \qquad [^\circ\text{C}\,\text{s}^{-2}] \quad (4.4)$$

Where T_i" refers to the second derivative of the temperature of the batteries. The model given in Equation 4.4 has a coefficient of determination of 0.3988 showing a poor linear fit in comparison to the first model. Approximating the second derivative of the internal temperature to be constant with respect to time (due to only a very minor change), an estimation for the time required to reach the battery heater enable

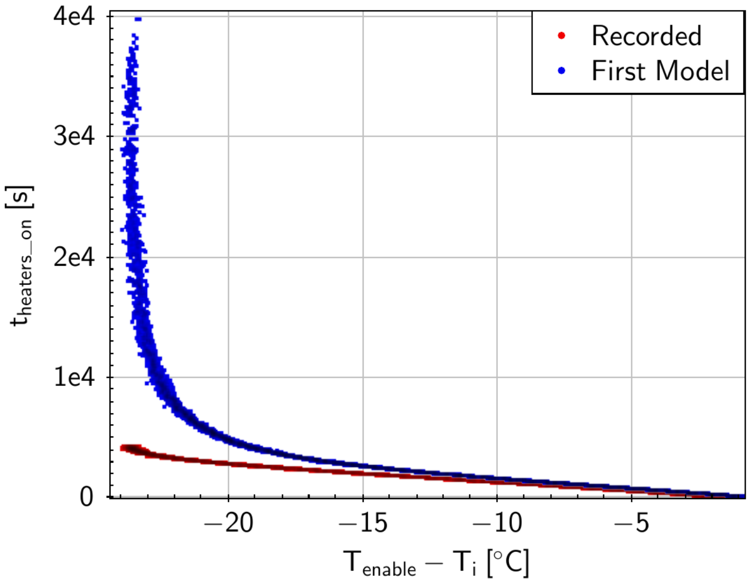


FIGURE 4.12: The predicted amount of time in seconds until the spacecraft battery heaters are enabled plotted against the difference between the battery temperature and the battery heater enable temperature. The actual recorded time remaining is also plotted for reference. The model is unstable above a 20°C separation between the internal temperature and the battery heater enable temperature as it relies on propagating the instantaneous rate of change of the battery temperature. When the difference between the internal and external temperatures is minimal, the model tends towards an infinite amount of time required. As the temperature gradient stabilises, the prediction increases in accuracy.

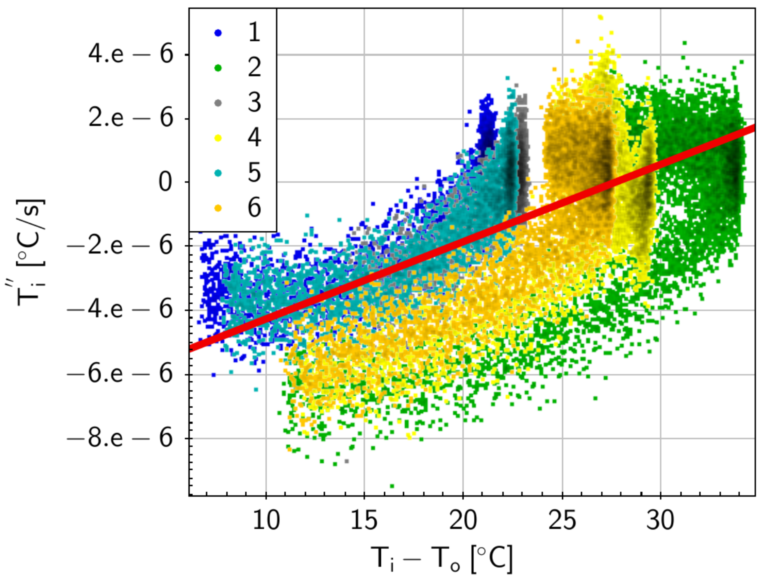


FIGURE 4.13: The relationship between the difference in the internal and external temperatures of the spacecraft and the second derivative of the temperature internal to the spacecraft at the batteries for each of the cold swings. All six series shown in Figure 4.10 are superimposed here. A linear fit has been applied to the data.

temperature can be given by:

$$T_{enable} - T_i = T_i' t_{heaters\_on} + \frac{1}{2} T_i'' * t_{heaters\_on}^2 \qquad [^\circ\mathrm{C}] \quad (4.5)$$

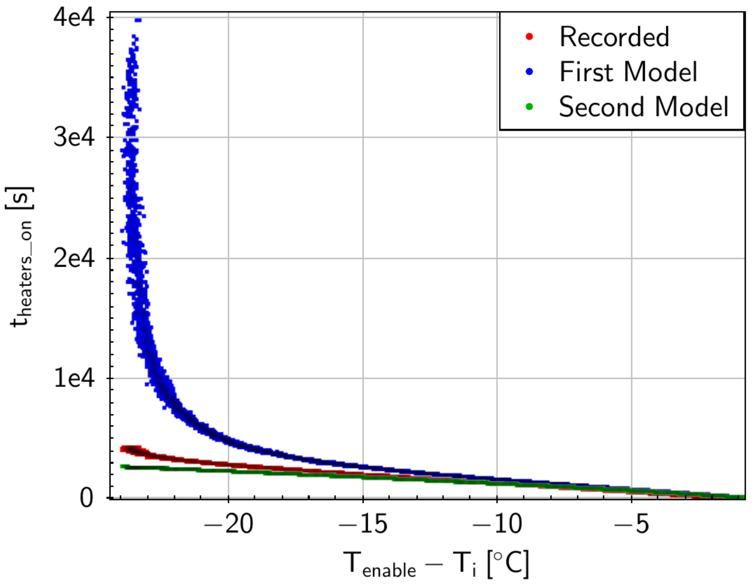
$$\therefore t_{heaters\_on} = \frac{-T_i' \pm \sqrt{T_i'^2 + 2T_i''(T_{enable} - T_i)}}{T_i''} \qquad [\mathrm{s}] \quad (4.6)$$

Plotting the predicted time for the battery heaters to be enabled using the second model against the recorded data and the first model (Figure 4.14(a)) shows that the inaccuracies due to propagation of the instantaneous rate of change have been addressed. Figure 4.14(b) provides a magnified view of the stable range of the first model, showing the predictions based on the second model seem to track the recorded data with higher accuracy. The first model, whilst maintaining a similar gradient to the recorded data, consistently predicts a longer time till enable than the recorded data. Applying both predictive models to the sixth cold swing (Figure 4.15) however shows a higher accuracy with the first model, with both predictive models under-predicting the time remaining until the battery heaters are enabled. The prediction inaccuracy between different cold swings can be attributed to the errors present in fitting the experimental data to a linear approximation. As these models were produced from limited data collected during one experiment with a single spacecraft, the authors acknowledge the possibility of over-fitting the models to the data. The next section therefore applies the generated models on-board the Binar-1 engineering model in the Curtin University vacuum chamber, and to the Binar-1 thermal model in a simulated LEO environment to assess their efficacy in heater usage prediction.

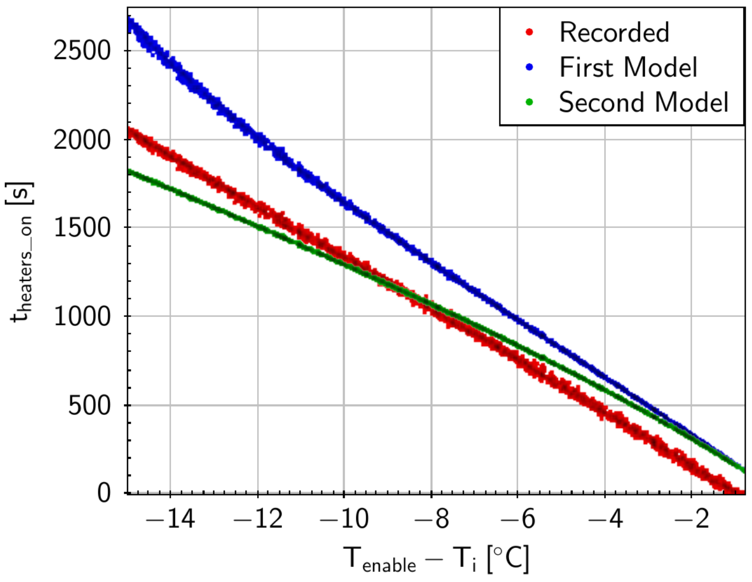
4.7. MODEL VERIFICATION

4.7.1. THERMAL VACUUM

The engineering model of Binar-1 was placed in the thermal vacuum chamber at Curtin University to test the derived models on-board the satellite during thermal cooling. A summary of the differences to the Wombat XL can be seen in Table 4.3. After pumping



(A) The predicted amount of time in seconds until the spacecraft battery heaters are enabled plotted against the difference in the battery temperature and the enable temperature (6°C). Both predictive models and the recorded data from the fifth cold swing are shown. The second model compensates for the inaccuracies introduced due to the non linearity seen in the first model, giving a higher prediction accuracy in the higher temperature difference ranges.



(B) Magnified view of the predictive models. The first model produced consistently predicts a higher time estimation than the recorded data until the heaters are enabled. The second model is seen to more accurately track the recorded data.

FIGURE 4.14: Predictions from both derived models of the time remaining until the spacecraft battery heaters are enabled, plotted against the recorded data from the fifth cold swing in the Wombat XL.

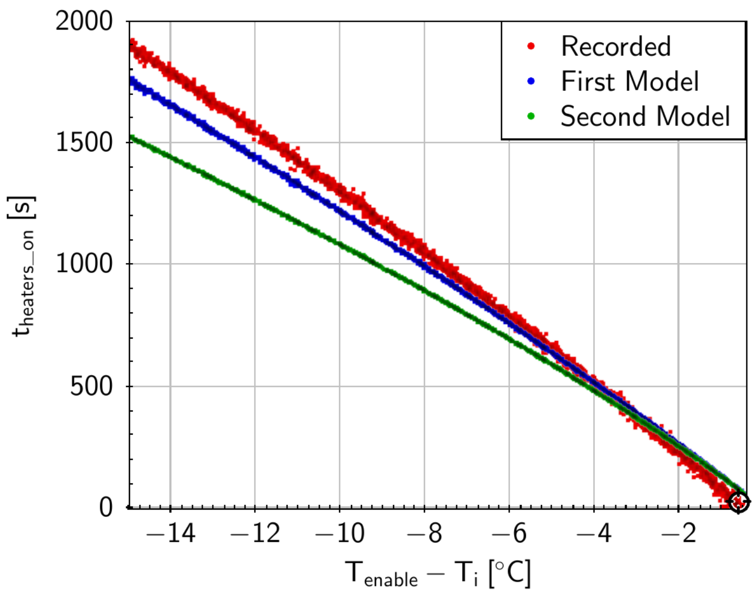


FIGURE 4.15: Magnified view of both predictive models of the time remaining until the spacecraft battery heaters are enabled, plotted against the recorded data from the sixth cold swing in the Wombat XL.

TABLE 4.3: Thermal vacuum chamber differences

	Curtin University	**Wombat XL**
Volume (m^3)	0.034	5.76
Vacuum Capabilities (Torr)	7.5e-4	1e-5
Temperature Range (°C)	-100 to +100	-170 to +150

down to an average chamber pressure of 7.5E-4 Torr, ten hours of thermal vacuum testing was performed (Figure 4.16(a)) before the chamber was permitted to return to ambient conditions. During this period, the battery heating circuitry was enabled twice. The subsets of the solar panel surface and battery temperatures pertaining to the regions preceding the battery heaters are shown isolated in Figure 4.16(b).

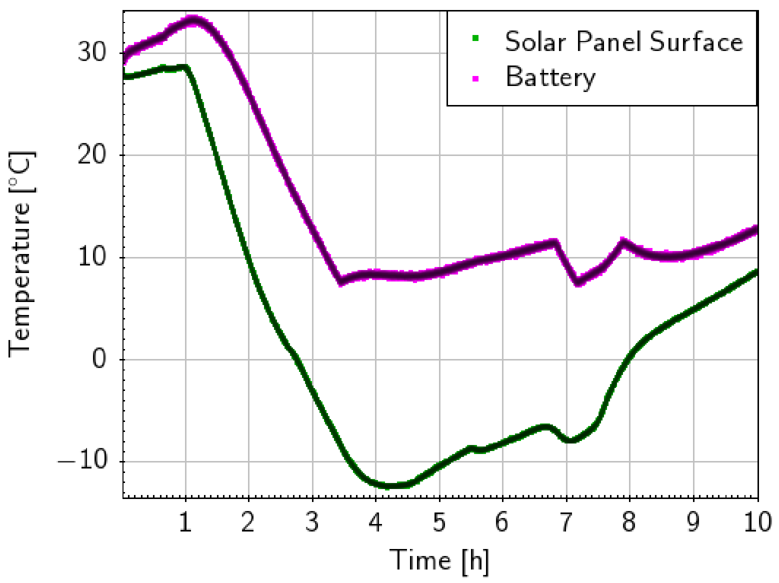
Applying the predictive models to the first and second instances preceding heater usage (Figures 4.17 and 4.18 respectively), the first predictive model is seen to more closely track the time remaining until the heating circuitry is enabled than the second. Both predictive models in this instance consistently under-predict the remaining time until the heaters are enabled. The predictions are seen to converge with the actual time remaining as the enable temperature set point is neared.

4.7.2. LEO SIMULATION

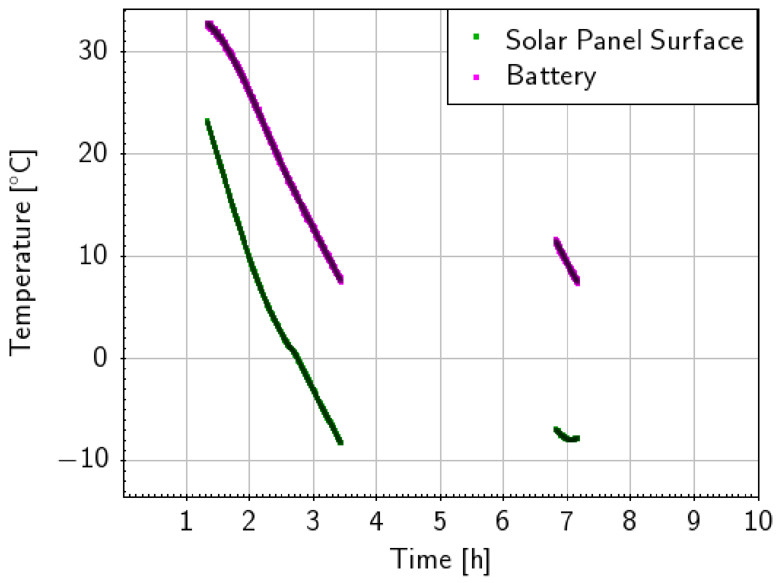
With the Binar-1 thermal model showing high similitude to the engineering model tested at the NSTF, the thermal input to the simulation was updated to emulate the temperatures expected over the course of an orbit.

The on-orbit thermal simulation of the Binar-1 thermal model used an incident radiation on the satellite from direct sunlight of 1400 W/m^2, an Earth albedo of 150 W/m^2, and IR radiation from Earth of 200 W/m^2 [155]. The spacecraft was assumed to be nadir pointing, with the exposure time of thermal radiation from the sun being distributed across the exposed faces evenly. An excerpt of the thermal response of the internal spacecraft temperature is plotted against the two prediction models in Figure 4.19, focusing on the section of the orbit where Binar-1 is eclipsed by Earth and the internal battery temperatures are starting to drop.

It is evident that the predictive models closely match the simulated response of the battery temperatures of the Binar-1 thermal model, giving high confidence of the efficacy of the model in LEO-like environments.



(A) The change in the battery temperature and solar panel surface temperature during the thermal vacuum test at Curtin University.



(B) Isolated view of the temperature response of the spacecraft to entering a cold swing in the Curtin University vacuum chamber. These segments have been taken from when the surface of the cells begin to respond to a cold cycle and stop when the battery heaters are enabled.

FIGURE 4.16: The change in the battery temperature and solar panel surface temperature during the thermal vacuum test at Curtin University. The heating circuitry was enabled twice during the test. Completion of the first usage of the models saw only one heater activate. As such, the battery warming portion following the first usage of the heating circuitry takes longer than the second as the chamber shroud was continued to be cooled in an attempt to trigger the second heater. The completion of the second use of the predictive models features both heaters activated.

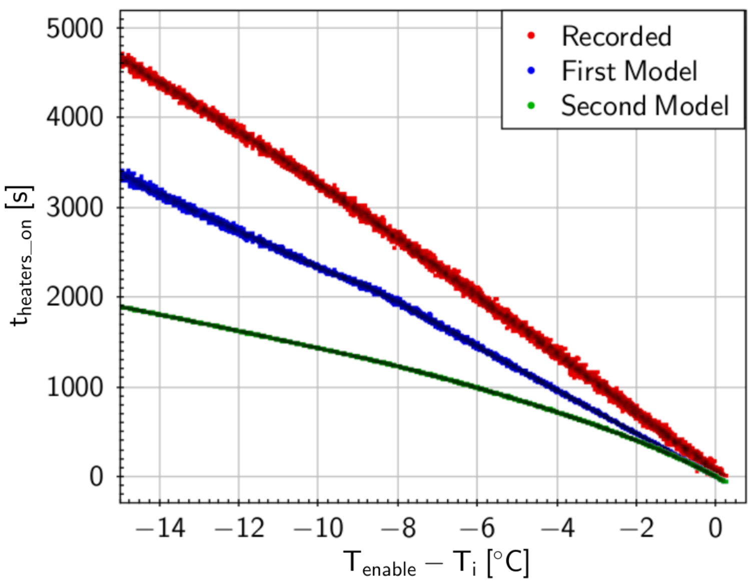


FIGURE 4.17: The predicted amount of time in seconds until the spacecraft battery heaters are enabled plotted against the difference in the battery temperature and the enable temperature during the experiment in the Curtin University vacuum chamber. Also present is the actual recorded time until the heaters were enabled. This section is taken from the temperature profile preceding the first usage of the battery heating circuitry in Figure 4.16(b). The first predictive model tracks the actual remaining time more accurately than the second model.

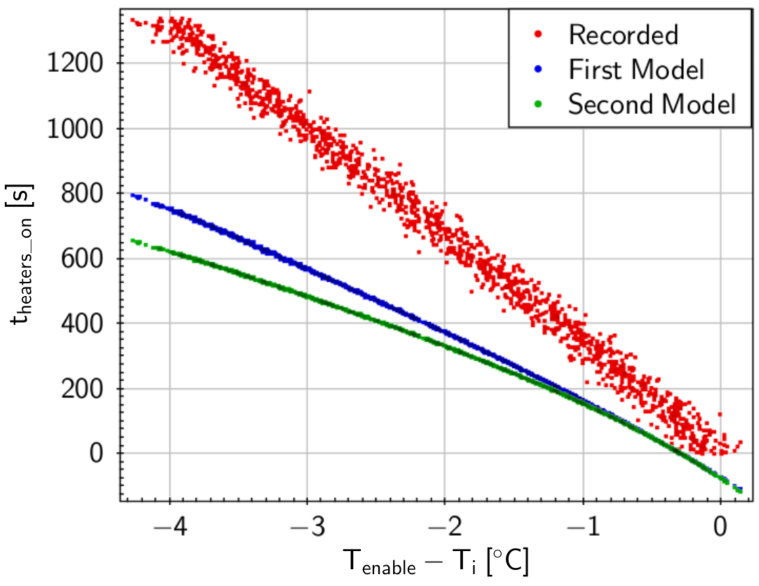


FIGURE 4.18: The predicted amount of time in seconds until the spacecraft battery heaters are enabled plotted against the difference in the battery temperature and the enable temperature. Also present is the actual recorded time until the heaters were enabled. This section is taken from the temperature profile preceding the second usage of the battery heating circuitry in Figure 4.16(b). The first predictive model tracks the actual remaining time slightly more accurately than the second model

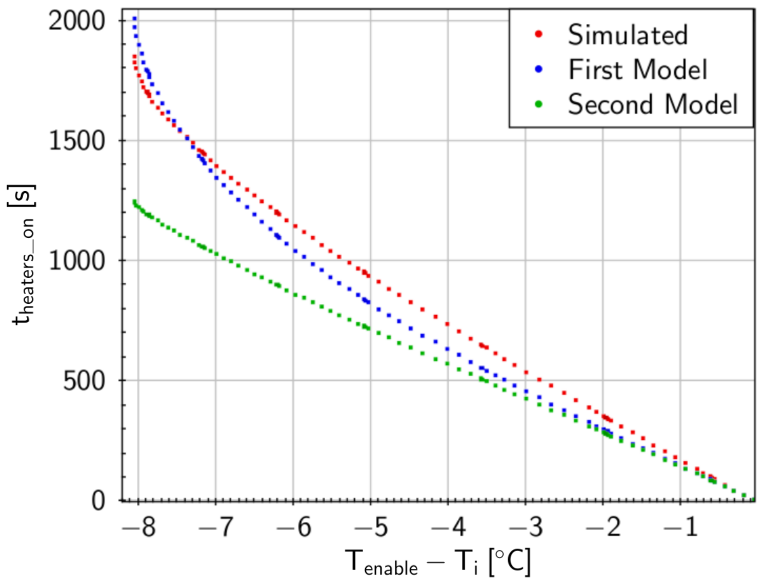


FIGURE 4.19: The predicted amount of time in seconds until the simulated spacecraft battery heaters are enabled plotted against the two produced models. The simulated spacecraft temperature was started at the thermal equilibrium (6°C). The data range is focused on the section of the orbit where Binar-1 is eclipsed by Earth, and the internal battery temperatures are starting to drop. The first model has a higher accuracy than the second in the prediction of the recorded data.

4.8. DISCUSSION

The relative low complexity of the models allows them to be included in the spacecraft flight logic to predict the usage of the heaters. The application mode of the Binar software framework contains a thread that executes scheduled tasks from a queue. Each task in the queue contains the software to execute and the start and end times of the task. In the software developed for Binar-1, when a task start time arrives, the task would be executed regardless of the status of the system vitals. From the experimentation done with the battery heaters, this is shown to be problematic as a high power task coupled with battery heater usage could potentially see the spacecraft encounter a low-power state. Therefore, the modelling derived in Section 4.6 can be used prior to task execution to assess if the spacecraft can safely progress with doing so. Figure 4.20 provides a proposed integration into the flight logic. After a task is popped from the queue, the battery capacity is to be checked to determine if the predictive modelling is required. If a sufficient charge is present, the flight logic can progress with regular task execution. However, if the spacecraft has insufficient charge, the

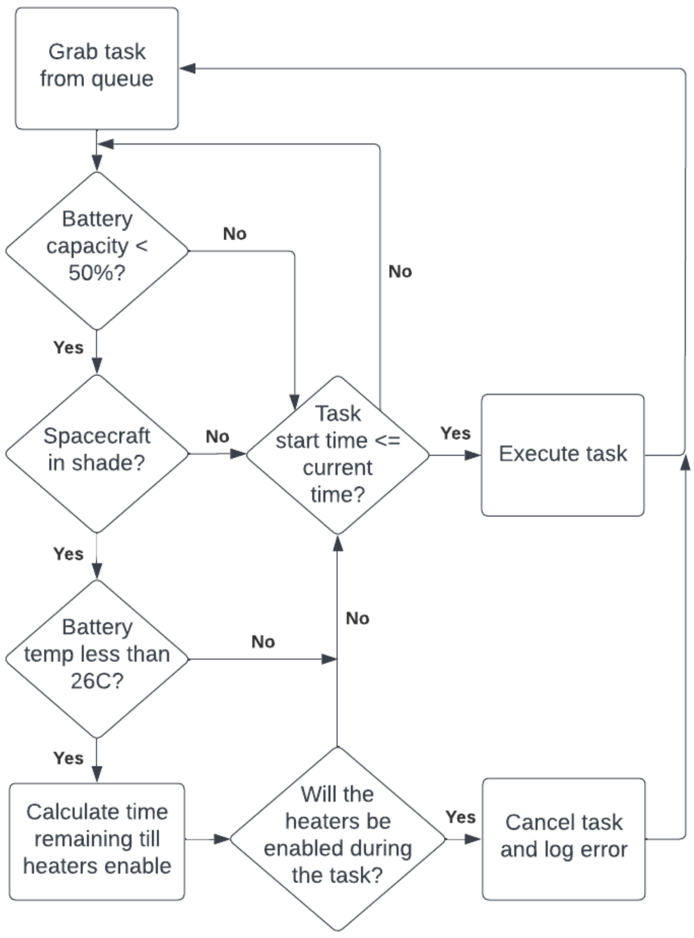

Grab task from queue
Battery capacity < 50%?
No
Yes
No
Spacecraft in shade?
No
Task start time <= current time?
Yes
Execute task
Yes
Battery temp less than 26C?
No
No
Yes
Calculate time remaining till heaters enable
Will the heaters be enabled during the task?
Yes
Cancel task and log error

FIGURE 4.20: The proposed implementation of the predictive modelling in the Binar flight logic. If there is insufficient battery capacity to execute a task and run the battery heaters, a series of checks are employed before calculating the expected time remaining before the heating circuitry is operational. This can be used to cancel the execution of a task that could otherwise cause the spacecraft to enter a low-power state.

predictive modelling is to be used to determine if the spacecraft is safe to progress with task execution. The average solar panel voltage is to be checked to determine if the Earth eclipses the sun from the spacecraft. Following this, a further check is employed to verify that the internal battery temperature is less than the heater enable temperature summed with the maximum usable range of the model as discussed in Section 4.6, 26°C. In the case which this check succeeds, a prediction of the time remaining until the heaters are enabled will be calculated. Summed with the current time, this predictive value can be referenced against the task end time to estimate if the heaters are to be operational during the task. This condition will see the task cancelled, and an error logged on board to be communicated to the ground station.

The responses seen from applying the predictive models to further thermal vacuum testing at Curtin University and the Binar-1 thermal model show a higher predictive accuracy in the time remaining until heater usage using the first model over the second. This relationship is also present in the application of the predictive models to the sixth cold swing recorded at the Wombat XL. Moving forward therefore, the first model is to be implemented in the spacecraft flight logic for the prediction of the usage of the battery heating circuity. The error in the prediction of the battery heater usage varies between the application to the data collected at the Wombat XL, the data collected at Curtin University, and a simulation of the Binar-1 thermal model in a LEO environment. This is likely due to unmodelled factors being present in the model, as it simplifies a relationship that is inherently non-linear. However, this is expected as the intention of the model is to provide a simplified computationally inexpensive means of prediction which by nature will deviate from a high accuracy model. Due to this, it is deemed sufficient to progress to an on-orbit test of the predictive model as a precursor to further development.

4.9. CONCLUSIONS AND FUTURE WORK

Being the first spacecraft built by the Binar Space Program, Binar-1 comprised many custom components that lacked flight heritage. Due to this, testing in a simulated space environment prior to launch was crucial in understanding the performance of the spacecraft in LEO conditions. With this objective in mind, verification of the battery heating system electrical characteristics was of high priority.

The testing procedure in the Wombat XL thermal vacuum chamber demonstrated that the battery heating system functioned as expected, keeping the internal temperature of the spacecraft from dropping below 6°C, demonstrating its reliability in LEO like environments. It was seen however that when the battery heaters were enabled, the power requirement from the electrical power system increased dramatically. As the battery heating system cannot be controlled by the flight logic, it stressed the necessity for on-board autonomy to predict when the heaters are to be used. With this, the spacecraft can make changes to the operations plan to prevent power intensive tasks running on the spacecraft during periods of high current draw from the heaters, possibly resulting in a low-power state.

From the data recorded during the testing in the thermal vacuum chamber, two models were generated relating the difference between the internal and external temperatures of the spacecraft to the first and second derivatives of the internal battery temperature with respect to time. These models were shown to predict the enabling of the battery heaters when the spacecraft entered a cold swing in the thermal vacuum chamber with sufficient accuracy. The engineering model of Binar-1 underwent further thermal vacuum testing at Curtin University with the two predictive models derived in Section 4.6 running on-board. Both models were found to track the estimated time until the battery heaters were enabled. To simulate the efficacy of the models on-orbit, a thermal model of Binar-1 was used. Verified against the recorded data in the Wombat XL, the Binar-1 thermal model was subject to a thermal simulation of a LEO environment. Applying the predictive models on the thermal data produced by the simulation of the Binar-1 thermal model showed that the first predictive model estimated the time remaining for the usage of the battery heaters with a higher accuracy than the second predictive model. This was likewise demonstrated from the results of the thermal vacuum testing at Curtin University, and the sixth cold swing in the Wombat XL. Due to this, moving forward the first model will be tested in the flight logic on future Binar spacecraft.

It was evident from the application of the models that the presence of unmodelled factors caused an inconsistent error during experimental predictions. Further testing on-orbit will contribute to refinement of the predictive model, potentially compensating for unmodelled factors and increasing the accuracy of the prediction of battery heater usage. Feruglio et al. demonstrate the benefits of using artificial neural networks on a CubeSat to improve mission autonomy when applied to a payload optimisation problem. Having been proven feasible, they propose that their research may be extended to encompass autonomous failure detection and isolation [150]. Acknowledging the predictive modelling proposed in this paper simplifies an inherently non-linear relationship, the predictive modelling may also be improved by using orbital data in training of a machine learning model of battery heater usage, with on-board inference.

Some apprehension exists in the community for relinquishing control over the mission operations plan to autonomous software. On-board autonomy on spacecraft however is crucial for safe operations in a communications limited environment to handle emergencies that cannot wait for input from ground operators. CubeSats offer a platform to trial on-board autonomy with a significant reduction in risk over larger, more expensive missions. The derived model will thus be used on Binar-2, 3 and 4 to infer when a task is not safe to execute due to power usage concerns without requiring input from a ground operator.

4.10. ACKNOWLEDGEMENTS

This research was supported by an Australian Government Research Training Program (RTP) Scholarship, and benefited from the research services provided by the National Space Test Facility at the Australian National University. It received Strategic Research Investment support from Curtin University and assistance from Australian National University.

REFERENCES

[60] D. Bernard, E. Gamble, G. Man, G. Dorais, B. Kanefsky, W. Millar, K. Rajan, J. Kurien, N. Muscettola, and P. Nayak, "Spacecraft autonomy flight experience - The DS1 Remote Agent Experiment," en, in *Space Technology Conference and Exposition*, Albuquerque,NM,U.S.A.: American Institute of Aeronautics and Astronautics, Sep. 1999. DOI: 10.2514/6.1999-4512. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.1999-4512.

[139] F. Downey, S. Buchan, B. Hartig, D. Busan, J. Cook, R. Howie, P. Bland, and J. Paxman, "Binar Space Program: Binar-1 Results and Lessons Learned," in *Proceedings of the Small Satellite Conference*, Aug. 2022. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2022/all2022/56.

[140] S. Ma, M. Jiang, P. Tao, C. Song, J. Wu, J. Wang, T. Deng, and W. Shang, "Temperature effect and thermal impact in lithium-ion batteries: A review," en, *Progress in Natural Science: Materials International*, vol. 28, no. 6, pp. 653–666, Dec. 2018, ISSN: 1002-0071. DOI: 10.1016/j.pnsc.2018.11.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1002007118307536.

[141] O. Erdinc, B. Vural, and M. Uzunoglu, "A dynamic lithium-ion battery model considering the effects of temperature and capacity fading," in *2009 International Conference on Clean Electrical Power*, Jun. 2009, pp. 383–386. DOI: 10.1109/ICCEP.2009.5212025.

[142] J. Straub, "A Review of Spacecraft AI Control Systems," *WMSCI 2011 - The 15th World Multi-Conference on Systemics, Cybernetics and Informatics, Proceedings*, vol. 2, pp. 20–25, Jan. 2011.

[143] J. Straub, "Attitudes towards Autonomous Data Collection and Analysis in the Planetary Science Community," en, *Galaxies*, vol. 1, no. 1, pp. 44–64, Jun. 2013, ISSN: 2075-4434. DOI: 10.3390/galaxies1010044. [Online]. Available: https://www.mdpi.com/2075-4434/1/1/44.

[144] D. W. Way, "Preliminary assessment of the Mars Science Laboratory entry, descent, and landing simulation," in *2013 IEEE Aerospace Conference*, Mar. 2013, pp. 1–16. DOI: 10.1109/AERO.2013.6497404.

[145] F. Abilleira, G. Kruizinga, S. Aaron, K. Angkasa, T. Barber, P. Brugarolas, D. Burkhart, A. Chen, S. Demcak, J. Essmiller, J. Gilbert, E. Gustafson, J. Kangas, M. Jesick, S. E. McCandless, S. Mohan, N. Mottinger, C. O'Farrell, R. Otero, C. Pong, M. Ryne, J. Seubert, P. Thompson, S. Wagner, and M. Wong, *Mars 2020 Perseverance trajectory reconstruction and performance from launch through landing*, en, Aug. 2021. [Online]. Available: https://trs.jpl.nasa.gov/handle/2014/52267.

[146] H. Shyldkrot, E. Shmidt, D. Geron, J. Kronenfeld, M. Loucks, J. Carrico, L. Policastri, and J. Taylor, "The First Commercial Lunar Lander Mission: Beresheet," in *Proceedings of the 2019 AAS/AIAA Astrodynamics Specialist Conference*, Aug. 2019.

[147] Y.-B. Kim, H.-J. Jeong, S.-M. Park, J. H. Lim, and H.-H. Lee, "Prediction and Validation of Landing Stability of a Lunar Lander by a Classification Map Based on Touchdown Landing Dynamics' Simulation Considering Soft Ground," en, *Aerospace*, vol. 8, no. 12, p. 380, Dec. 2021, ISSN: 2226-4310. DOI: 10.3390/aerospace8120380. [Online]. Available: https://www.mdpi.com/2226-4310/8/12/380.

[148] H. W. Stone, *Mars Pathfinder Microrover: A Small, Low-Cost, Low-Power Spacecraft*, en, 1996. [Online]. Available: https://trs.jpl.nasa.gov/handle/2014/25424.

[149] T. J. Martin-Mur and B. Young, *Navigating MarCO, the first interplanetary CubeSats*, en, Feb. 2019. [Online]. Available: https://trs.jpl.nasa.gov/handle/2014/49242.

[150] L. Feruglio and S. Corpino, "Neural networks to increase the autonomy of interplanetary nanosatellite missions," en, *Robotics and Autonomous Systems*, vol. 93, pp. 52–60, Jul. 2017, ISSN: 0921-8890. DOI: 10.1016/j.robot.2017.04.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889016304419.

[151] J. H. Henninger, *Solar absorptance and thermal emittance of some common spacecraft thermal-control coatings*, Apr. 1984. [Online]. Available: `https://ntrs.nasa.gov/citations/19840015630`.

[152] H. Maleki, S. Al Hallaj, J. R. Selman, R. B. Dinwiddie, and H. Wang, “Thermal properties of lithium-ion battery and components,” *Journal of the Electrochemical Society*, vol. 146, no. 3, p. 947, 1999.

[153] Z. Peterson, *FR4 Thermal Properties to Consider During Design*, en, Nov. 2020. [Online]. Available: `https://www.nwengineeringllc.com/article/fr4-thermal-properties-to-consider-during-design.php`.

[154] Z. Tian, S. Lee, and G. Chen, “A Comprehensive Review of Heat Transfer in Thermoelectric Materials and Devices,” *Annual Review of Heat Transfer*, vol. 17, Jan. 2014. DOI: `10.1615/AnnualRevHeatTransfer.2014006932`.

[155] A. Raslan, G. Michna, and M. Ciarcià, “Thermal Simulation of a CubeSat,” in *2019 IEEE International Conference on Electro Information Technology (EIT)*, May 2019, pp. 453–459. DOI: `10.1109/EIT.2019.8833736`.

CHAPTER 5

On-Board CubeSat Digital Twin

5.1. Introduction

When spacecraft operate in environments with limited communication, it is essential to have onboard systems capable of preventing failures without human intervention. Onboard simulation inference is one way to achieve this capability autonomously. To this aim, the previous chapter detailed the development of a linear model to be implemented in the flight logic for the Binar 2, 3, and 4 satellites. The inferences from this model were proposed to autonomously cancel a scheduled task on orbit when the battery heaters are simultaneously required and insufficient power is available. This chapter details the development of an onboard digital twin platform and, as an extension of the previous chapter, implements the onboard failure prevention method proposed in the future work section. The onboard digital twin platform detailed in this chapter has the capability to prevent failures not only in the battery heating system, but also in other critical systems onboard the spacecraft. This approach enables autonomous identification and mitigation of a wide range of potential failure scenarios.

As seen from the subset of mission failures analysed in Chapter 2, unfortunately, failures occur on spacecraft while in orbit. In the interest of mission success, safe code execution pathways should exist to handle faults when they occur. A common method of implementing this in spacecraft design is through FDIR routines. The elements that make up these routines can be broken down as follows:

- **Fault Detection** - Identify a fault.
- **Isolation** - Determine the location of the fault in the system.
- **Recovery** - Provide a way to remedy the fault.

FDIR routines are implemented following an investigation of all possible faults on board, using tools such as fault tree analysis [156]. These routines can encompass straightforward errors, such as erroneous voltages coming from a sensor, to more complex problems, such as boot looping. After detecting a fault, a lookup table is often consulted to execute a process to remedy the situation. As the highest priority is maintaining the mission [157], often the result of the recovery procedure transitions the spacecraft into a safe mode that requires intervention from a ground operator to exit [158]. An exception to this is when hardware redundancy is available onboard. When a faulty sensor is detected, the FDIR routine can swap to a redundant sensor without requiring an operating mode transition. However, on a low-budget, resource-constrained platform such as a CubeSat, this may not be a viable option [72]. Aside from this, traditionally, more complex FDIR routines often involve processing on the ground using satellite telemetry [157].

A limited subset of operations is possible for system safety when the spacecraft is transitioned into safe mode. The limited operations may mean that the spacecraft cannot perform mission objectives. Concerningly, false positives from sensors often transition the spacecraft into safe mode due to conservative thresholds on sensor nominal readings triggering FDIR routines [72]. The latency added from relying on a ground operator to transition the spacecraft back into nominal operation mode can result in significant scientific loss over the full mission duration. Furthermore, the latency introduced requiring interaction with a ground operator for the processing of complex FDIR routines may be detrimental for a deep space mission [72].

With increasing system complexity, the volume of sensor datasets obtained from spacecraft is enormous. Expecting a ground operations team to identify faults in the data throughput, especially with potentially incomplete or erroneous data, is not possible [72]. On-board the spacecraft, high data throughput volumes have resulted in conventional FDIR methods sometimes failing to isolate faults [157]. Due to this, research into advanced FDIR concepts utilising machine learning (ML) techniques for fault detection is being pursued. Kato et al. propose the computational analysis of mined data on a spacecraft to narrow down the scope of the data presented to an operator during a failure in an attempt to reduce human error [159]. Holsti et al. extend the conventional fault detection approach by using a Bayesian Network in fault identification to automate fault analysis. However, these methods still do not address the latency issue when spacecraft operate in a communications-denied environment.

Moreover, spacecraft failures do not always happen immediately. The root cause analysis of critical mission failures may present a chain of contributing faults [159]. However, the current conventional FDIR approach in the literature is largely a reactive process for detecting instantaneous faults. This approach may not be able to identify contributory faults higher up the fault tree, and offer recovery methods for them [158].

An approach in the literature for addressing the negative impact of relying on ground operators as part of the FDIR loop is exploring the feasibility of performing advanced FDIR routines entirely onboard the spacecraft. Shifting the FDIR functionality closer to the edge has been made possible by advances in computational technologies, reducing the frequency of intervention required by ground operators [72]. Adapting advanced FDIR concepts to run onboard a spacecraft also introduces the possibility of dealing with uncertain dynamic system evolution, allowing for identification and actioning on potential failures rather than relying on reaction alone [158].

Significant effort has been made in the literature to develop the robustness and reliability of FDIR routines to increase spacecraft operational time by reducing the required amount of safe mode entries. However, little work has been done in the literature towards using the same advanced FDIR routines to identify faults in advance. As advanced as FDIR routines can be, some fault routines cannot avoid entering a safe mode. Detection of potential failures in advance in the system should be a key component for failure prevention on spacecraft [159], increasing the operational time on orbit. To this end, in a feasibility demonstration Portinale et al. apply Dynamic Bayesian

TABLE 5.1: Mission execution autonomy levels. Taken from [160].

Level	Description	Functions
E1	Mission execution under ground control; limited onboard capability for safety issues.	Real-time control from ground for nominal operations. Execution of time-tagged commands for safety issues.
E2	Execution of pre-planned, ground-defined, mission operations onboard.	Capability to store time-based commands in an onboard scheduler.
E3	Execution of adaptive mission operations onboard.	Event-based autonomous operations. Execution of onboard operations control procedures.
E4	Execution of goal-oriented mission operations onboard.	Goal-oriented mission re-planning.

Networks (DBN) to identify an anomaly in advance when applied to a simulation of the power supply system of a Mars rover [158]. The network identified the anomaly and recommended preventative measures at an early depth in the fault tree.

Techniques such as this are crucial for high mission autonomy. The ECSS has provided a qualitative means of describing mission autonomy on spacecraft, replicated in Table 5.1. According to Olive [157], most spacecraft operate at autonomy level E2, executing pre-planned ground-defined mission operations onboard. Advanced FDIR routines capable of detecting potential faults before they happen are required onboard to progress towards autonomy level E4.

However, the DBN approach by Portinale et al., as well as most other studies of advanced onboard FDIR in the literature, are simulation-based. Implementation on spacecraft hardware is still a novel topic that needs exploring [72]. CubeSats are viewed as not an option for trialling advanced onboard autonomy due to their relatively low system complexity [72]. On the contrary, it is the judgement of the candidate that the low-cost aspect of CubeSat missions provides a higher accommodation for risk than larger-scale expensive missions, and they, therefore, can provide an ideal platform for testing novel approaches to FDIR. The effort put into developing the Binar CubeSat platform and the Binar Software Framework (BSF) that powers it enables the capability to trial advanced FDIR concepts on orbit to address the gap in the literature using onboard intelligence for preventative fault detection in advance. This chapter, therefore, leverages the existing framework to develop an onboard digital twin platform capable of performing preventative fault detection at the edge without relying on ground operations. The specific fault detection capabilities useful to a spacecraft are hugely scalable and customisable to the specific platform. Hence, the scope of the onboard digital twin platform covered in this chapter will be limited to two examples, both generic areas that most on-orbit spacecraft can benefit from - a prediction of the usage of battery heaters and an estimation of future power usage. The inferences performed by the flight logic are used to detect a future power fault issue and recover by preemptively removing unnecessary power usage that would cause the future fault.

The chapter first documents the development of the digital twin platform abstracted from the spacecraft to speed up the development process and increase the testability of the platform. The platform is then ported onto the BCM, and a hardware-in-the-loop test (HITL) is performed to assess the efficacy of the flight software using the onboard digital twin to suggest changes to the mission plan in respect of safety. The HITL testing is then repeated, swapping out the inferencing implementation in favour of a trained ML model, demonstrating the modularity of the developed platform.

5.2. Digital Twin Development

5.2.1. Hardware Decoupling

The development approach desired for the digital twin library was to break hardware dependency for development, support portability across various platforms, and maximise the testability. From the benefits seen from the implementation of the RPC platform implemented for ground-to-satellite communications, a digital twin RPC approach was investigated. Breaking hardware dependencies was raised as a requirement as developing software libraries for embedded targets is a slow process. Compared to the development cycle of software developed on a host machine that can be compiled and executed quickly, software developed for an embedded target needs to be cross-

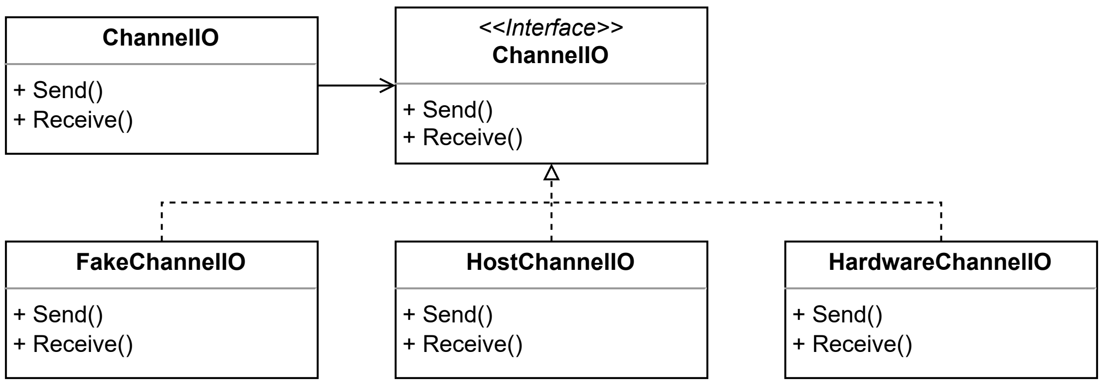


FIGURE 5.1: The digital twin client and server objects wanting to access the input and output channel for the network will use the high-level ChannelIO object that depends on the same abstraction as the lower-level implementations. A variety of IO backends can be provided as an implementation, abstracting the client and server objects from the channel they communicate over.

compiled and then deployed onto the target for execution. This process of deploying onto a target for execution can take a few seconds, eventually accumulating to a significant portion of time over a complex development process. More impactful, however, is that as software is developed parallel to the spacecraft hardware, new board revisions require new board support packages to be developed to interface with the hardware. Even with a complete board support package, hardware faults that may exist on the spacecraft motherboard could hinder the correct operation of the library. This can cause lengthy debugging cycles hindering the speed of development.

Furthermore, hardware interaction time becomes a resource for a team of engineers with a finite amount of hardware targets to develop on. Fortunately, most software development can be done independently of the hardware by abstracting the logic from hardware dependencies, as discussed in Chapters 3 and 4. This is especially true for the digital twin development, the majority of which does not require a hardware dependency to operate. The only aspect of the digital twin library that requires a hardware dependency is how the client and server communicate. This was abstracted by requiring the client and server objects to communicate through a ChannelIO module built following the Dependency Inversion Principle discussed in Chapter 3. A UML depiction of the module is provided in Figure 5.1.

In the module, the high-level public-facing ChannelIO object and the lower-level private application-specific implementations depend on the abstracted ChannelIO interface. When the digital twin client and server attempt to send or receive serialised messages, they must go through the high-level object that implements the same interface used by the implementation objects rather than depending on the implementation objects directly. This allows the IO backend to be swapped without impacting the digital twin client and server application code. Coupled with the language-neutral and platform-neutral aspects of the RPC approach, the digital twin can be developed to support operation on various hardware using various programming languages. To this aim, the digital twin framework could be developed independently from the spacecraft on a host machine before porting to spacecraft hardware.

The RPC library used for ground-to-satellite communications on Binar-1 was custom written as no suitable open-source alternatives were available. Since the Binar-1 software development, however, Google has released an open-source RPC module as part of the Pigweed project, a collection of libraries for use on embedded systems [136]. The decision was made to adopt this library instead of using the original Binar RPC framework as it has open-source community support, providing frequent bug fixes and feature additions. With the scope of the digital twin implementation narrowed to a demonstration of the prediction of the usage of battery heaters and an estimation of future power usage, the digital twin service could be written using the high-level protocol buffers description language.

LISTING 5.1: High level description of the digital twin service.

```
service DigitalTwinService {
      rpc BatteryHeatersUsagePrediction(
         ↪ BatteryHeatersUsagePredictionRequest) returns
         ↪ (BatteryHeatersUsagePredictionResponse);
      rpc FutureBatteryCapacityPrediction(
         ↪ FutureBatteryCapacityRequest) returns(
         ↪ FutureBatteryCapacityResponse);
}
```

The service described in Listing 5.1 informs the protocol buffers compiler of the expected request and response messages for the RPC communication, which can be specified independently of the RPC service. The implementation of the request and response messages is provided in Listing 5.2.

LISTING 5.2: Message types available for the digital twin client and server.

```
message Temperature {
      float internal_average = 1;
      float external_average = 2;
}

message BatteryHeatersUsagePredictionRequest {
      Temperature current_temperature = 1;
}

message BatteryHeatersUsagePredictionResponse {
      uint64 seconds_till_enable = 1;
}

message FutureBatteryCapacityPredictionRequest {
      float battery_voltage = 1;
      float battery_current = 2;
      uint64_t time_delta = 3;
}

message FutureBatteryCapacityPredictionResponse {
      float battery_capacity = 1;
}
```

The benefit of segmenting request and response messages is the modularity and extensibility achieved in the RPC service. The contents of the messages being transferred by the RPC service can be modified easily in the future as the simulation requirements become more complex. Moreover, the unique identifiers given to each field within

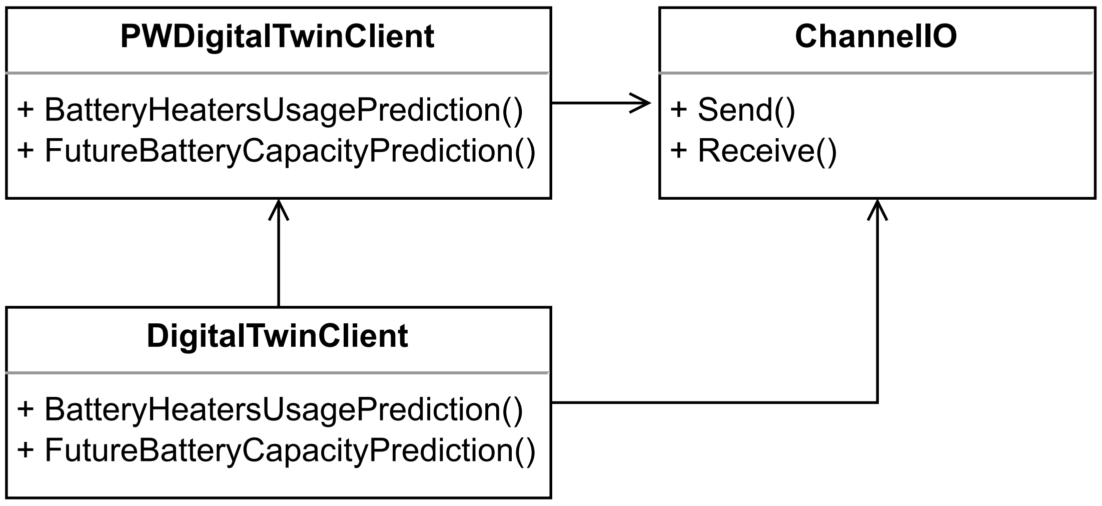

PWDigitalTwinClient
+ BatteryHeatersUsagePrediction()
+ FutureBatteryCapacityPrediction()
ChannelIO
+ Send()
+ Receive()
DigitalTwinClient
+ BatteryHeatersUsagePrediction()
+ FutureBatteryCapacityPrediction()

FIGURE 5.2: The high-level DigitalTwinClient module associates both the Pigweed generated client, PWDigitalTwinClient, and the ChannelIO object. The PWDigitalTwinClient is responsible for transmitting messages, and the DigitalTwinClient is responsible for receiving the responses.

the message allow for backward compatibility between the digital twin client and the server communicated with. Converting the high-level RPC descriptions into software provides independent client and server libraries to be compiled into their respective applications.

Following the drive for reusability in the BSF, the client and server libraries were wrapped with high-level objects to be used in application code. This allows the entire RPC framework implementation to be exchanged in the future without impacting the application code. The final digital twin client object (Figure 5.2) is used by the flight logic and the server object (Figure 5.3) by the digital twin processor.

The DigitalTwinClient and DigitalTwinServer objects associate the ChannelIO object, meaning the module exists within and is used by the digital twin. The DigitalTwinClient object also associates the Pigweed-generated client, PWDigitalTwinClient. When the client application calls a method on the DigitalTwinClient object, it will access the member PWDigitalTwinClient object to perform the RPC call. This is why the PWDigitalTwinClient also needs to associate the ChannelIO object - The high-level DigitalTwinClient object is responsible for receiving messages, and the PWDigitalTwinClient object is responsible for transmission. Conversely, the high-level DigitalTwinServer object inherits and overrides the methods provided in the Pigweed-generated

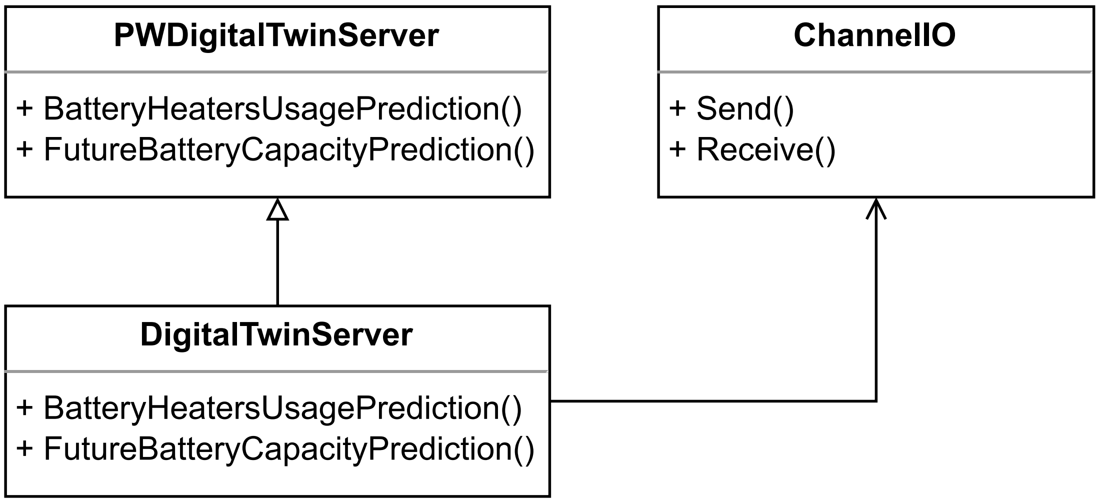

PWDigitalTwinServer
+ BatteryHeatersUsagePrediction()
+ FutureBatteryCapacityPrediction()
ChannelIO
+ Send()
+ Receive()
DigitalTwinServer
+ BatteryHeatersUsagePrediction()
+ FutureBatteryCapacityPrediction()

FIGURE 5.3: The high-level DigitalTwinServer object associates the ChannelIO object and inherits the PWDigitalTwinServer. It is responsible for both receiving messages from the client and transmitting responses.

server, PWDigitalTwinServer. This is because the protocol buffers generated description only generates method stubs that need to be implemented. Due to this, only the high-level module must associate the ChannelIO object, handling both receiving and transmitting messages. Figure 5.4 provides an overview of how the client and server libraries integrate to perform an RPC cycle.

A further benefit of decoupling the I/O communications channel from the RPC implementation is that it enables the server and client logic to be developed in adherence to Test-Driven Development (TDD). Unit testing the client and server independently is achieved by creating a fake input channel for the ChannelIO object that gives full control over the communications within each test case. This is especially useful for the DigitalTwinServer object, which provides the implementation of the remote procedure call. The contents of the buffer that will get returned when the module invokes the Receive method on ChannelIO can be set to return undecodable messages, valid decodable messages, and error-inducing decodable messages containing erroneous values. This aids in ensuring robustness and safety in the procedure implementations, which is crucial when the result of the digital twin inferencing will be used to influence mission outcomes without consultation from the ground. With automated unit testing achieved, the digital twin client and server objects can be incorporated into the automated CI/CD pipeline, demonstrating safety on each commit. After providing an implementation

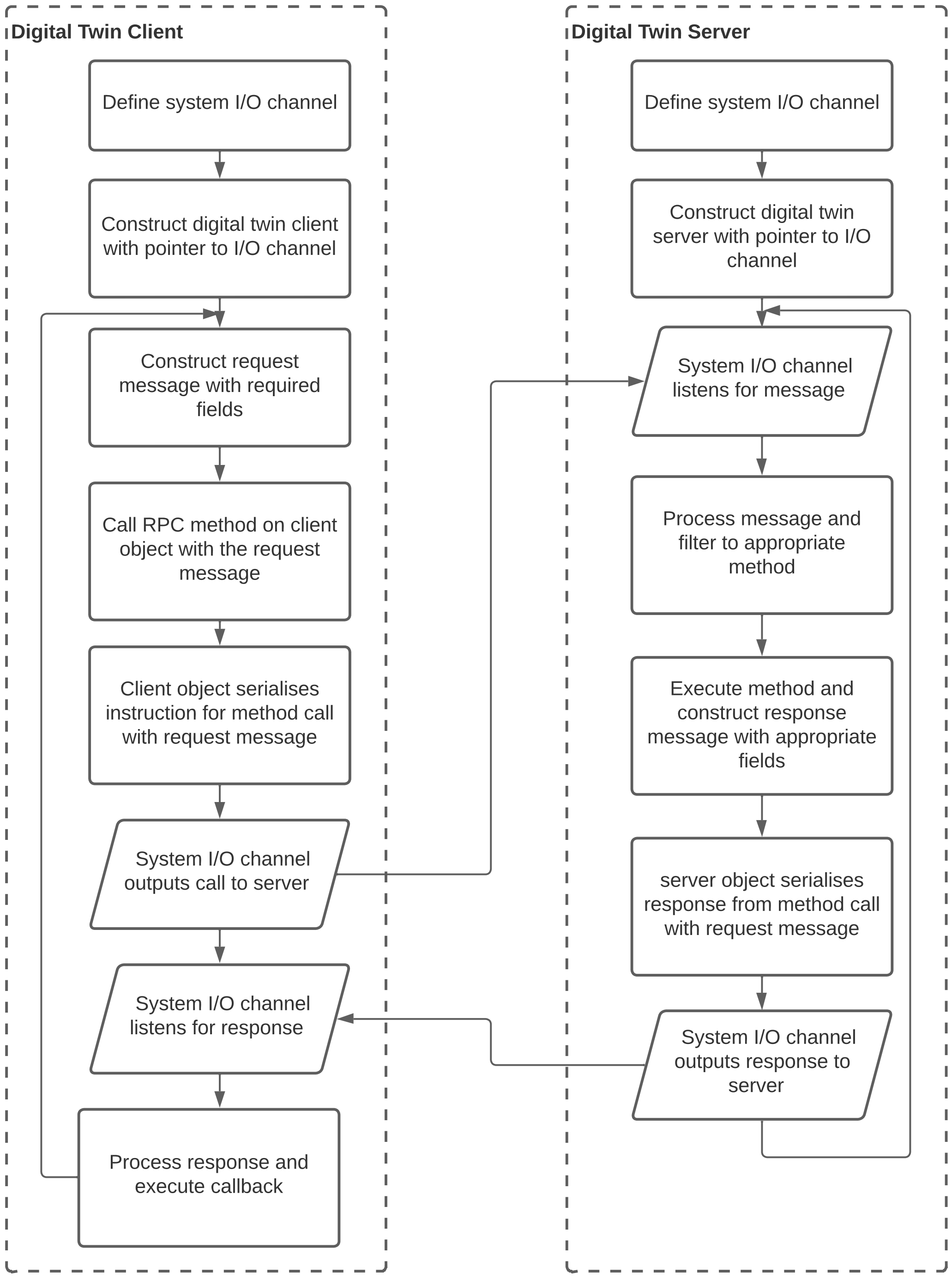

Digital Twin Client
Define system I/O channel
Construct digital twin client with pointer to I/O channel
Construct request message with required fields
Call RPC method on client object with the request message
Client object serialises instruction for method call with request message
System I/O channel outputs call to server
System I/O channel listens for response
Process response and execute callback
Digital Twin Server
Define system I/O channel
Construct digital twin server with pointer to I/O channel
System I/O channel listens for message
Process message and filter to appropriate method
Execute method and construct response message with appropriate fields
server object serialises response from method call with request message
System I/O channel outputs response to server

FIGURE 5.4: A depiction of how the digital twin client and server operate together in the RPC implementation.

for the digital twin client and server using a TDD approach, integration testing of the libraries was performed on the host machine. A transmission control protocol (TCP) ChannelIO backend was implemented to enable the client and server applications to communicate on the host, verifying compatibility prior to attempting execution on the target. Leveraging the portability of the codebase achieved with the BSF, the client and server applications could then be ported onto the microcontroller cores to demonstrate the feasibility of simulations on a computationally constrained platform.

5.2.2. PORTING TO A CONSTRAINED PLATFORM

To minimise the potential performance impact on flight logic runtime, a requirement was made for the onboard implementation of the digital twin server to run independently of the flight logic processor. The original flight computer selected for Binar-1 was the STM32H743, offering a single high-performance Cortex-M7 core. Due to the impact of the global semiconductor chip shortage from the COVID-19 pandemic, the flight computer was swapped to the pin-compatible dual-core STM32H747xi during the final board revision before launch. Although the global semiconductor chip shortage eventually impacted the procurement of this new flight computer, it was decided it would remain unchanged to support the digital twin on the Binar 2, 3, and 4 satellites. The STM32H747xi offers a high-performance Cortex-M7 core and a power-efficient Cortex-M4 core, with 1 megabyte of Flash memory each. During Binar-1 development, it was realised that the features of the Cortex-M7 core exceeded what was required for the flight logic. A lower-performance core would be satisfactory to meet the mission requirements whilst reducing the overall power usage. Thus, leveraging the software portability achieved from the development of the BSF, the flight logic was ported to the Cortex-M4 core on the STM32H747xi flight computer, allowing the higher-performance Cortex-M7 core to be reserved for the onboard digital twin server.

A key challenge with software execution on a multiprocessor system is the synchronisation of the cores. The approach for the communications channel for the onboard digital twin employs the use of Open Asymmetric Multiprocessing (OpenAMP) [161]. OpenAMP is a software framework that facilitates asynchronous core-to-core communication through a shared read/write memory region with cross-core interrupt triggering. For the digital twin implementation, the client and server cores are set up in a master/slave configuration to reduce power consumption when the digital twin server core is not required. The master flight logic core permits the slave server core to boot, waiting for an acknowledgement. The server core completes the endpoint connection and signals the flight logic core that it is ready to receive RPC requests. Either core writing to the shared memory region will trigger an interrupt for the recipient core, notifying there is a message to process. A sequence diagram of the communications is shown in Figure 5.5.

5.3. HARDWARE-IN-THE-LOOP DIGITAL TWIN TEST

5.3.1. METHODOLOGY

Performing inferencing onboard a spacecraft allows for information from the environmental state to be read directly from the sensors as input parameters, bringing data computation closer to the edge. Hence, the onboard simulations will operate regardless of the source of the sensor data. Therefore, rather than requiring a thermal vacuum experimental setup to demonstrate the power management predictions from the onboard digital twin, HITL testing can instead be conducted using pre-recorded data. Leveraging the modularity of the developed BSF, the hardware backend for the onboard temperature sensors is replaced with a replay of recorded temperature values to provide fault injection to the target under test. The data used in the demonstration was taken from the sixth cold swing done in the Wombat thermal vacuum chamber at the Australian National University (ANU), as discussed in Chapter 4. The replayed data is the 2400 seconds before enabling the battery heating circuitry (Figure 5.6).

The timestamp, internal and external temperature data collected was converted into a lookup table compiled into the application code. When the respective temperature sensor objects call to read the temperature data, the onboard time is used as an input to the lookup table to retrieve the corresponding temperature data. As proposed in the future work section from Chapter 4, the model derived to predict the usage of the

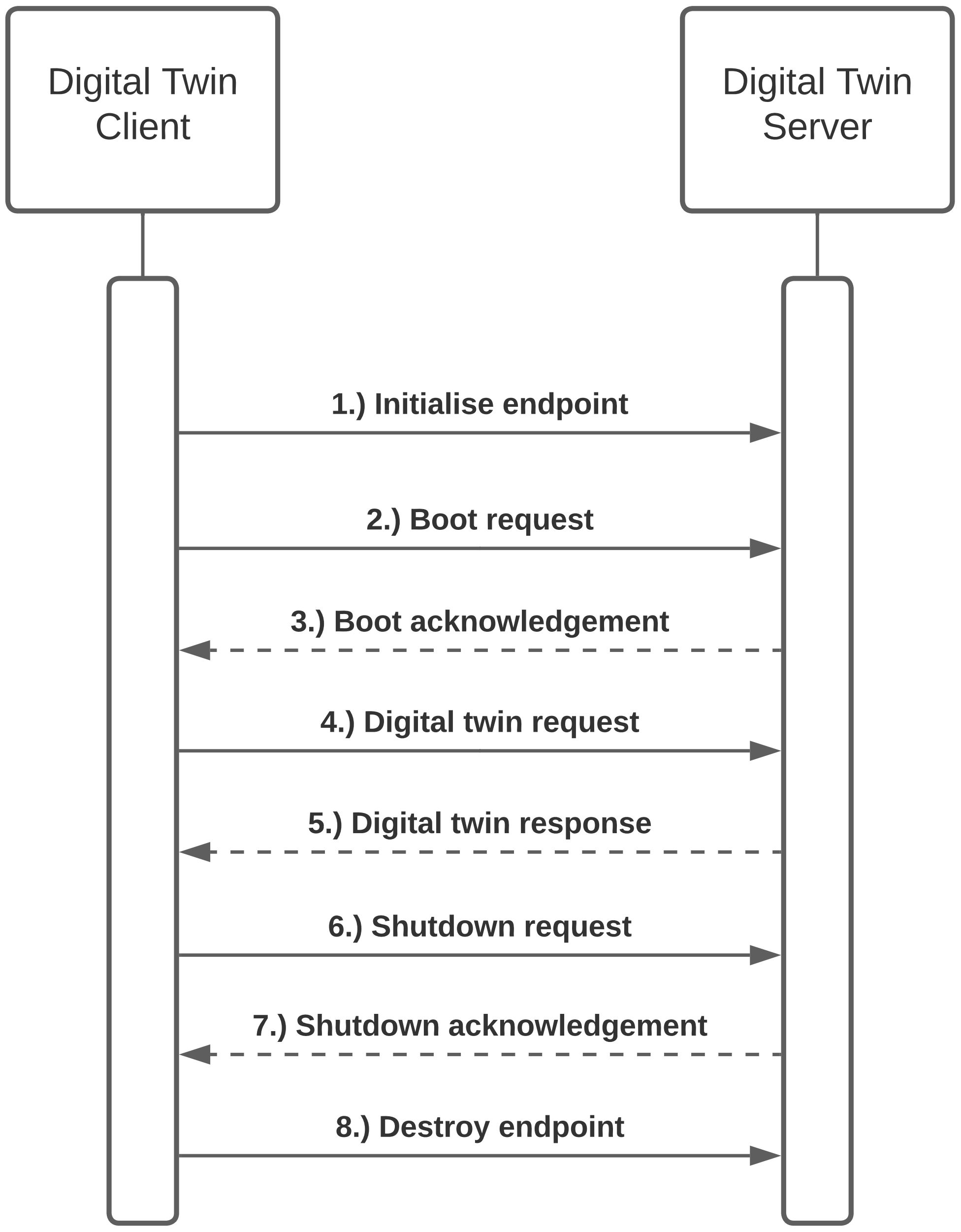


FIGURE 5.5: Sequence diagram showing the synchronisation of the dual-core digital twin implementation. The flight logic core (using the digital twin client library) operates as the master and the digital twin server core as the slave device.

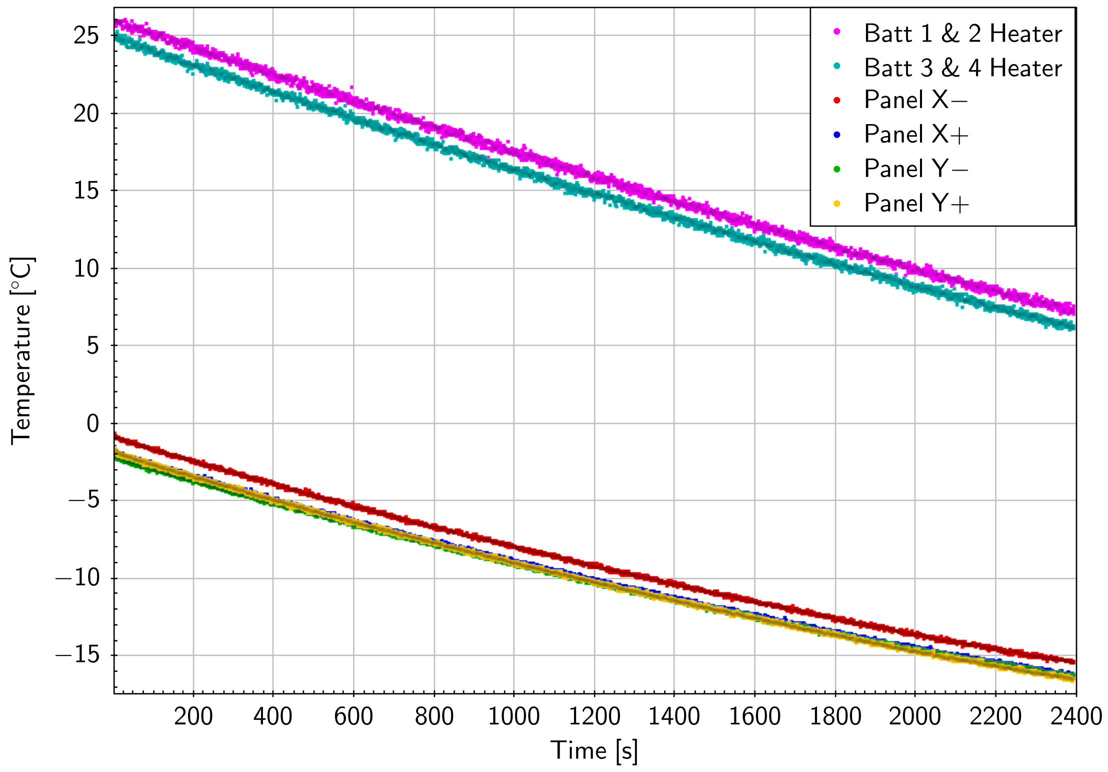


FIGURE 5.6: An excerpt of the change in the battery and solar panel surface temperature during a thermal vacuum test of the Binar-1 engineering model at ANU. The battery heaters drop below the enable temperature 40 minutes into the dataset.

battery heaters can be used onboard the spacecraft to optimally postpone tasks if the spacecraft is in danger of encountering a low power state. The successful demonstration will show the digital twin logic able to prevent a low power state from occurring by cancelling a scheduled task. As an improvement to the logic flow proposed to be implemented in the flight software from the future work section of Chapter 4, the logic will now also account for the future predicted battery capacity in the decision to postpone a task, as shown in Figure 5.7.

This change was made to the proposed logic implementation as the previous suggestion required the battery capacity to drop below 50% charge before the digital twin would be used. The previously proposed implementation would be satisfactory for tasks being executed in the near future when the spacecraft is still occulted from the Sun by the Earth. However, with an orbital period of roughly 90 minutes, tasks scheduled for execution a few orbits in the future could gain a sufficient charge that would remove the spacecraft from the danger of a low power state occurring. The proposed improvement, therefore, after identifying that the battery heaters are to be enabled during task execution, will predict what the battery capacity will be at the start of the task time and will only postpone a task if the future capacity is less than 50%.

Prediction of the future battery capacity is implemented by assuming constant battery current usage between the moment the digital twin is enabled and the start time of the task under the decision of postponing. This estimated capacity usage is then subtracted from the battery capacity when the digital twin was enabled. Battery capacity is not measurable, however, and needs to be inferred from the battery voltage reading at the ADC. The spacecraft batteries were drained from full charge at a 1 A constant current rate to characterise the relationship between the battery voltage and the battery capacity. The batteries in the EPS are arranged in a two-series and two parallel configuration. The configuration can nominally supply 6.7 Ah according to the battery's rated capacity from the datasheet. The battery capacity could be inferred at each voltage step using the time taken to drain the batteries during the applied load. The relationship between the battery voltage and the battery capacity for the four lithium-ion cells powering the spacecraft is shown in Figure 5.8.

As the proposed digital twin logic only postpones a task when the battery capacity is

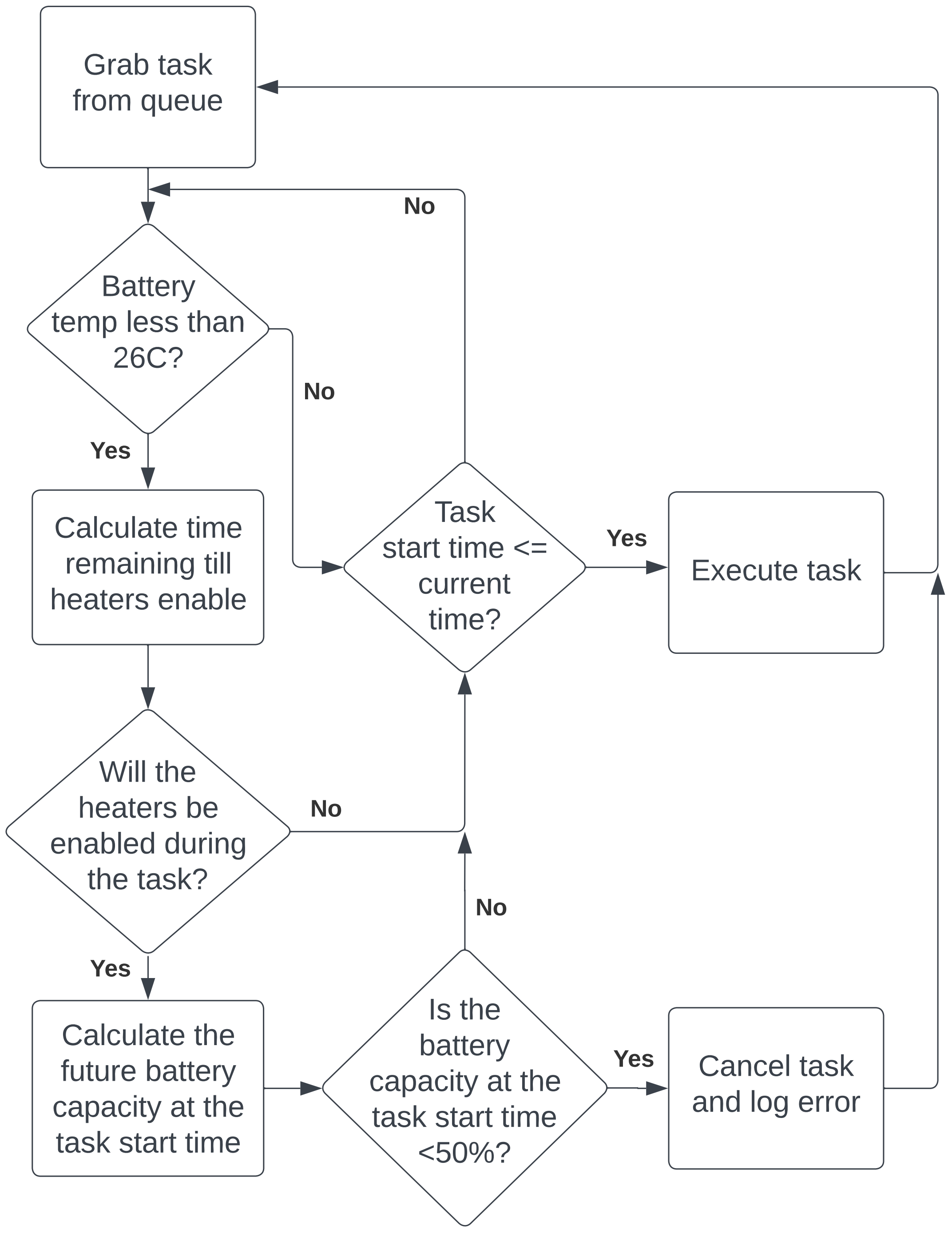

Grab task from queue
No
Battery temp less than 26C?
No
Yes
Calculate time remaining till heaters enable
Task start time <= current time?
Yes
Execute task
Will the heaters be enabled during the task?
No
No
Yes
Calculate the future battery capacity at the task start time
Is the battery capacity at the task start time <50%?
Yes
Cancel task and log error

FIGURE 5.7: The proposed flight logic to demonstrate the capability for the onboard digital twin to optimally postpone tasks without intervention from the ground if executing them would result in a low power state.

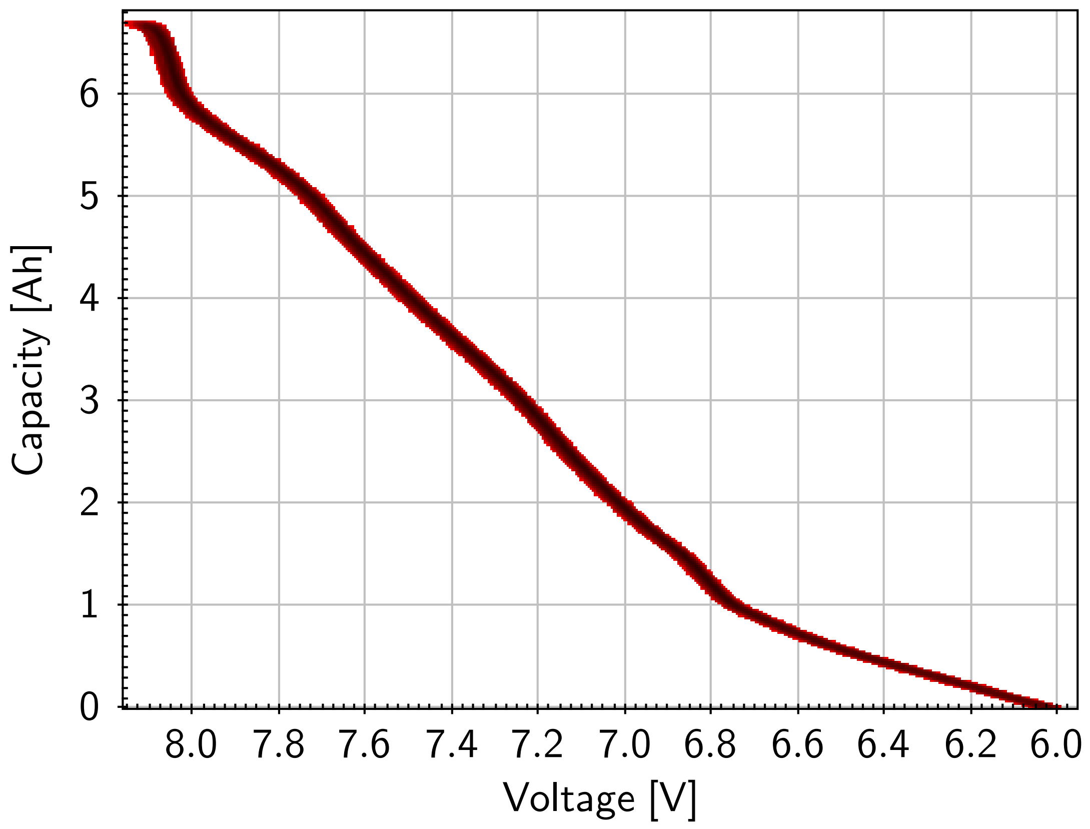


FIGURE 5.8: The relationship between battery capacity and battery voltage for the four lithium-ion cells powering the spacecraft. The profile was collected by discharging the spacecraft batteries from full at a 1 A constant current rate.

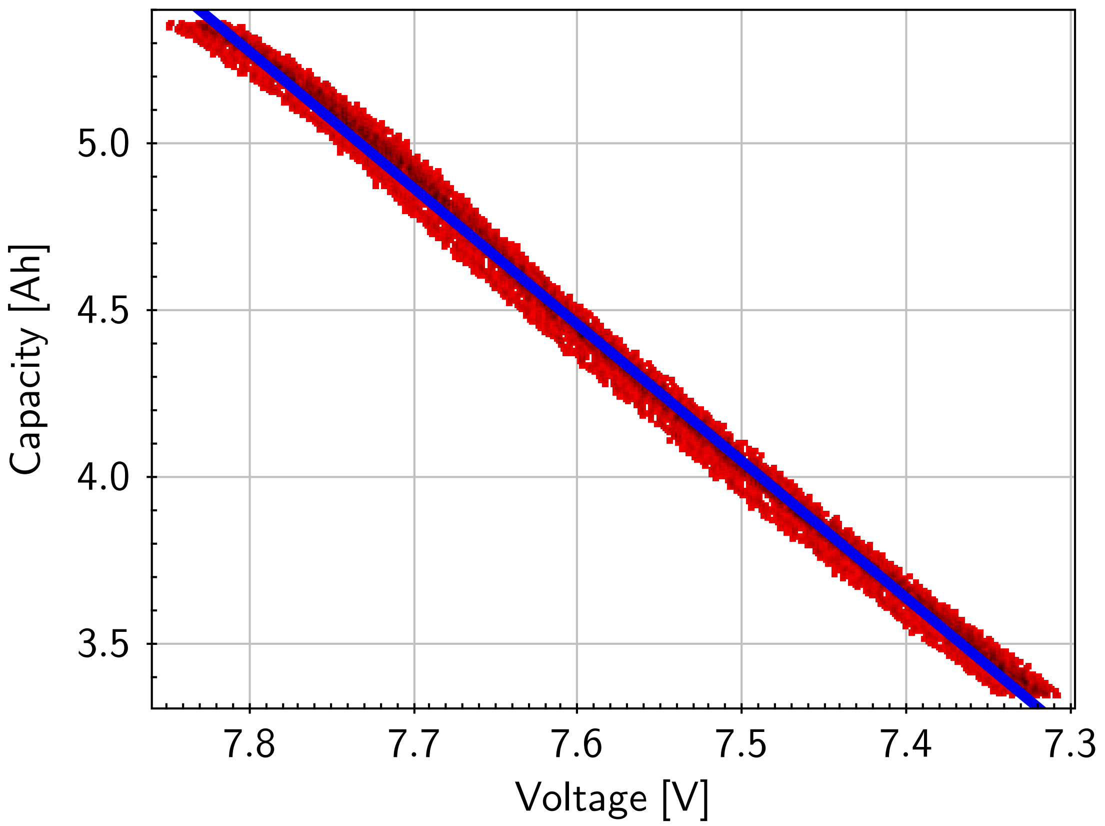


FIGURE 5.9: A line of best fit on the battery characterisation data within the 50% to 80% capacity range.

less than 50%, the region of interest for the model could be constrained above this value. Furthermore, it was deemed that if the battery capacity is above 80%, the spacecraft would not be in danger of encountering a low power state. Figure 5.9 shows the region of interest for battery capacity inference with a linear approximation applied.

From this, the battery capacity could be inferred from the battery voltage within the 50% to 80% capacity range using Equation 5.1:

$$capacity = 4.088568 * voltage - 26.616879 \qquad [\mathrm{A\,h}] \tag{5.1}$$

Knowing when the heaters are to be enabled from the pre-recorded temperature sensor data, the experimental set-up consists of scheduling a task for the spacecraft to operate during the usage of the heaters. Noticing that the battery temperature sensor objects are reporting the internal temperature below 26°C, the digital twin client will

TABLE 5.2: A summary of the three trials proposed to test the reliability of the onboard digital twin in failure prevention. In all three trials, the replayed data will have the heaters enabled 2400 seconds after the program is executed.

Trial	**Time Until Task Start (s)**	**Time Until Task End (s)**	**Battery Capacity at Task Start (%)**
1	2350	2500	45
2	1000	1200	90
3	2350	2500	80

start to communicate over the RPC network to the digital twin server. With this input condition, the digital twin server identifies that the battery heaters will be enabled during the task execution. The flight logic will then request the digital twin server to perform a further inference to predict the battery capacity at the task start time. Again leveraging the modularity of the BSF, the hardware backend for reading the battery voltage and instantaneous current usage onboard can be faked to provide low-capacity readings. By providing these values to the digital twin to predict the future battery capacity to be less than 50%, the server should recommend the flight logic cancel the task execution and log an error.

Two more trials will be executed to ensure the reliability of the digital twin server's decision to cancel a task. Firstly, using a task start time before the battery heaters enable according to the recorded data. This should see the digital twin server allow the task to execute at the original time. Secondly, a task will be scheduled to execute when the heaters are to be used, but ensuring that the future battery capacity at the task start time is greater than 50%. This should see the digital twin server acknowledge sufficient charge to execute the task. Table 5.2 summarises the experimental trials.

The task to be scheduled for all three trials is to emit a beacon at a five-second period. This task provides a visual element to the task execution, as the output channel from the spacecraft hardware can be monitored for incoming messages. A further check of the onboard system logs will show an entry signifying that the task was not executed.

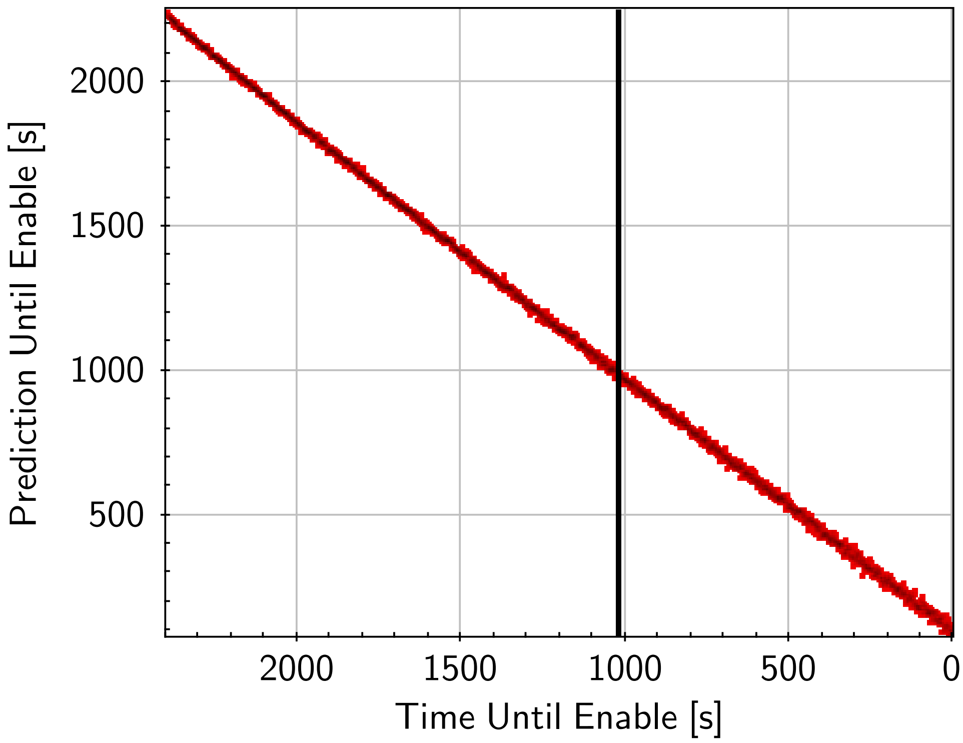


FIGURE 5.10: The predicted time remaining until the battery heaters are enabled is plotted against the actual time from the dataset used for the onboard inference. The onboard digital twin will predict the battery heaters to be used during the scheduled task roughly 23 minutes into the dataset, depicted by the black separator.

5.3.2. RESULTS AND DISCUSSION

With the task in the first trial scheduled 2350 seconds after the start of the application, the digital twin starts to predict the battery heaters to be used during the task execution roughly 23 minutes into the dataset, depicted by the black separator in Figure 5.10.

The runtime error logs were checked 10 minutes into the trial, carrying no entry regarding task cancellation. However, 25 minutes into the trial, after the point in which the digital twin starts to predict the heaters to be used during the upcoming task execution, an entry in the error logs denotes that the digital twin cancelled the task. The cancellation was further verified when the flight computer failed to emit beacons when the task start time arrived.

The second trial saw a task executed before the battery heaters were used, shown in Figure 5.11 bounded by black separators. Checking the runtime error logs 10 minutes

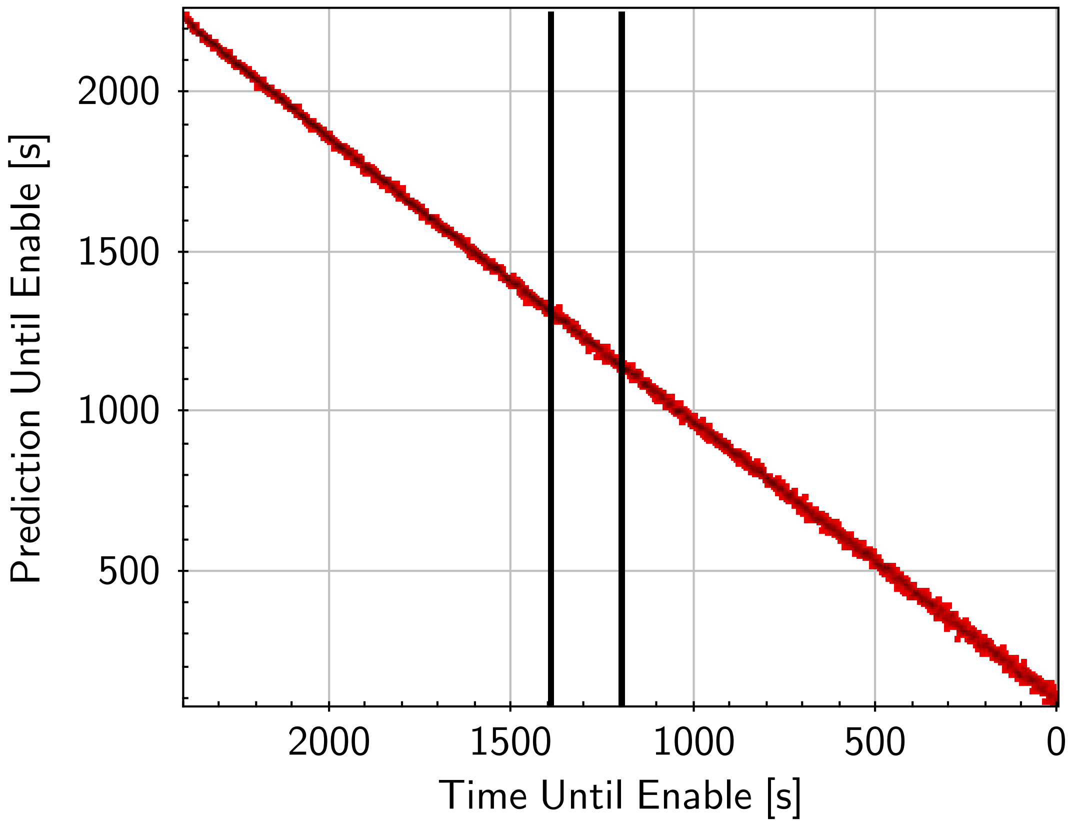


FIGURE 5.11: The predicted time remaining until the battery heaters are enabled is plotted against the actual time from the dataset used for the onboard inference. The scheduled task executes in the region bounded by the black separators well before the heaters are activated at the end of the dataset.

into the trial returned no entry regarding task execution. As expected, when the task start time arrived, it was verified to have been successfully executed by sighting the beacons emitted from the flight computer.

The same task execution time used for the first trial was repeated with the third trial, with an alteration to the battery capacity remaining. The runtime error logs were checked 10 minutes into the trial, showing no entry regarding task cancellation. Similar to trial one, the error logs were again checked 25 minutes into the trial. Unlike trial one, the error logs still contained no entry regarding task cancellation. When the task start time arrived, it was verified to have been successfully executed by sighting the beacons emitted from the flight computer.

From the three trials performed, the onboard digital twin successfully identified when limited power resources required prioritisation of the mission-critical battery heaters. As expected, the first trial showed a cancelled task when the heaters were to be enabled during task execution and the battery capacity was lower than 50%. Trials two and three allowed the task to be run, in the former due to the heaters not being required simultaneously and the latter when the digital twin was informed there was sufficient battery capacity to handle the increased load. Having demonstrated robustness in autonomous operations on the spacecraft hardware, the developed digital twin will be deployed on the Binar-2, 3 and 4 satellites to demonstrate on-orbit operations.

This experiment also demonstrated the portability of the developed digital twin platform. The host-verified platform was successfully ported to hardware by developing an OpenAMP backend for the abstracted ChannelIO object, allowing the onboard digital twin client and server to run on independent cores within the flight computer package. A platform to facilitate data replay from a lookup table was developed to simplify the onboard digital twin demonstration. The immediate observed benefit of this platform is the granular control over what system peripherals are reading at any given time. While allowing repeatable constrained testing of the digital twin logic under varying scenarios, it also highlights the ability to inject faults into the flight logic to test the response. During the experimental set-up, the proposed implementation of the battery heaters usage prediction in the flight logic was amended from the future work section in Chapter 4. The amendment added a requirement for the onboard digital twin to check the future battery capacity when the task is to be started. The extra check refines the suggestions proposed by the onboard digital twin concerning task cancellation and demonstrated the extensibility of the developed digital twin platform. New inferences can be added with minimal effort and collated in custom inference chains. As the server and client are decoupled, procedure implementations on the server can be changed independently of the client. This means that the digital twin server can be updated without affecting the digital twin client as long as the server implementation accepts the same inputs and returns the same outputs. Consequently, the application software hosting the digital twin client can interact with the updated digital twin server, regardless of the server's version. When the digital twin server and client exist in separate firmware, the client firmware can interact with the updated server firmware without also requiring a firmware update. This provides flexibility in updating and using the digital twin system. In the context of a spacecraft on orbit, the digital twin server firmware can be upgraded and interacted with without risky firmware upgrades to the flight software that hosts the digital twin client.

5.4. MACHINE LEARNING ON A CONSTRAINED PLATFORM

5.4.1. METHODOLOGY

As the RPC network implemented for communication with the digital twin abstracts the actual implementation of a procedure from the calling code, the backend can be modularly swapped out without impacting the rest of the software. This enables the fidelity of simulations to be updated easily. An ML approach can offer higher accuracy in predictions over the conventional modelling used in Section 5.3.1. This is due to ML excelling in detecting hidden insights in large amounts of data, learning from them, and performing reliable predictions on unseen data [162]. However, an area where the ML approach falls short of conventional modelling is commonly due to its large computational footprint and, as an extension, ability to run closer to the sensor at the edge. Computational requirements for trained model inference conflict with the limited resources available on edge devices, primarily power, CPU cycles and storage space [163]. Due to this, traditionally, data from edge devices would need to be transmitted from the device to be processed elsewhere where computational constraints are not an issue [163]. The requirement for remote processing severely worsens latency on performing inference [164]. When operating in a communications-limited environment, such as in a space application, the latency introduced by processing being done remotely from the spacecraft may be detrimental to a mission. The exponentially increasing global adoption of the Internet of Things (IoT) has led to huge amounts of data being collected that could benefit from computation close to the sensor when performing inference [165][164][163]. This is because IoT devices, while capable of generating huge amounts of data, typically have small memory banks and limited computing power [165]. As more data-generating devices are added to the IoT, requiring offloading of inference can worsen latency and cost issues [164], impacting the device's functionality.

Hence, efforts have been made in recent years to bring ML to the edge to perform inferencing closer to the sensor [164]. The ability to perform onboard inference close to the data sources on edge devices can remove the latency issues traditionally associated with data transmission for remote ML inference. However, it is important to note that the microprocessors often enabling edge devices are limited by a severely reduced clock speed compared to desktop computers and servers that traditionally perform model inferencing. Hence, with the ML at the edge approach, the model's complexity and the dataset's size can noticeably impact performance due to the instruction throughput of the edge device. Therefore, it is essential to evaluate the applicability of the edge device for applications with real-time requirements that may need to process data quickly. A simplified ML or standard computational model may be more appropriate in these situations.

TensorFlow, a popular open-source software library for training ML models, recently added support to run ML models on highly memory-constrained platforms. TensorFlow Lite for Microcontrollers (TFLM) does not require operating system support, standard libraries, or dynamic memory allocation [166], showing high applicability for inclusion on the spacecraft flight computer. With this technology, the digital twin backend for predicting the time remaining until the battery heaters are enabled was swapped out in favour of an ML approach. This enables a demonstration of the digital twin modularity and explores if the computationally constrained spacecraft flight computer can perform onboard ML inference.

The first stage in developing an ML model is procuring a training dataset. The same dataset used in Section 5.3 from the thermal vacuum experimentation at ANU was again used. The temperature data internal and external to the spacecraft during the six cold swings were divided into training, validation and testing groups. The training and validation dataset was shuffled, and a random selection of 15% of the data points contributing to the first five cold swings was reserved to verify the model. The remaining 75% were used for training. The data points from the sixth cold swing were reserved for testing the trained model. A correlation table (Figure 5.12) shows the relationships between features in the training dataset.

The candidate acknowledges the potential limitations in the validation dataset's

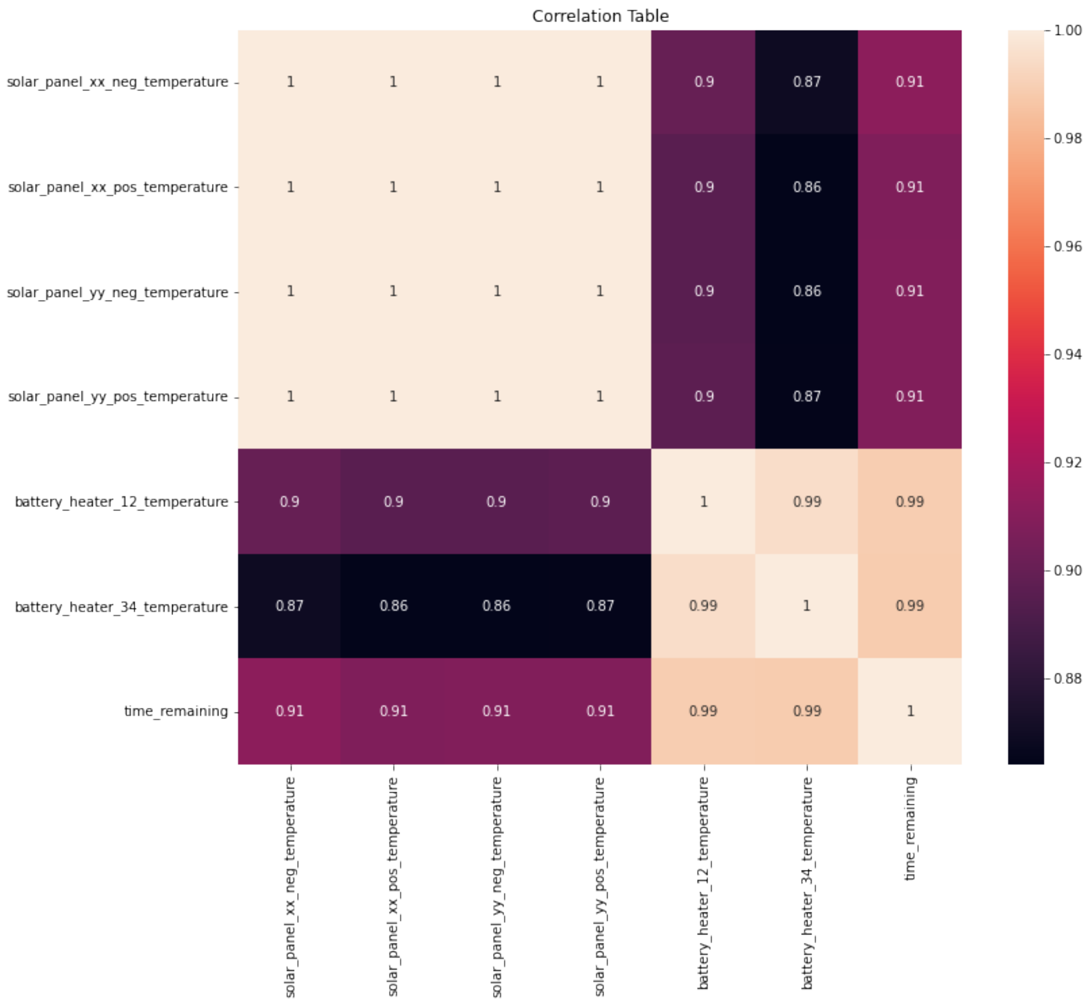


FIGURE 5.12: A correlation matrix showing relationships between all sensor recordings in the training dataset.

robustness. The thermal cycles observed in this dataset may exhibit similar behaviours to the calibration dataset, which could result in the validation dataset not being entirely independent of the calibration dataset. It is important to stress, however, that the primary objective of this experiment is not to achieve the highest possible accuracy in training the machine learning model using this dataset. Instead, the main goal is to demonstrate a proof of concept of performing machine learning model inference on memory-constrained computers under challenging conditions. It is worth noting that more reliable and diverse data will be collected during the Binar 2, 3, and 4 missions. This future data will provide an opportunity to train a model with a more extensive and representative dataset, enhancing the model's accuracy and performance. Therefore, the results obtained from this initial experiment, while not perfect, serve as a valuable starting point and a proof of concept for the broader mission to improve model performance in the future.

The four external solar panel surface temperatures are highly correlated, as expected, as they received the same thermal input. The correlation between all solar panel surface temperatures and the temperature of the heaters for batteries one and two are consistently 90%. However, the correlation between all solar panel surface temperatures and the temperature of the heaters for batteries three and four have the lowest correlation, fluctuating between 86% and 87%. Identifying this lower correlation shows strength in the ML approach, where conventional modelling may fail to appropriately adjust the weighting assigned to the data being contributed from these sensors. The reduced correlation between the external temperatures on the solar cells and the temperatures recorded at the third and fourth batteries could be due to the thermal insulation not being consistent in the battery compartment of the spacecraft. Thus, an ML approach can help compensate for inhomogeneity in the spacecraft's construction. As mentioned in Section 5.2.1, the contents of the messages being transferred by the RPC service can be modified easily to support increased complexity. To make use of variable weightings for all sensor axes, the temperature message transferred to the digital twin server as part of the battery heaters usage prediction request was updated from internal and external averages to all six onboard temperature sensors.

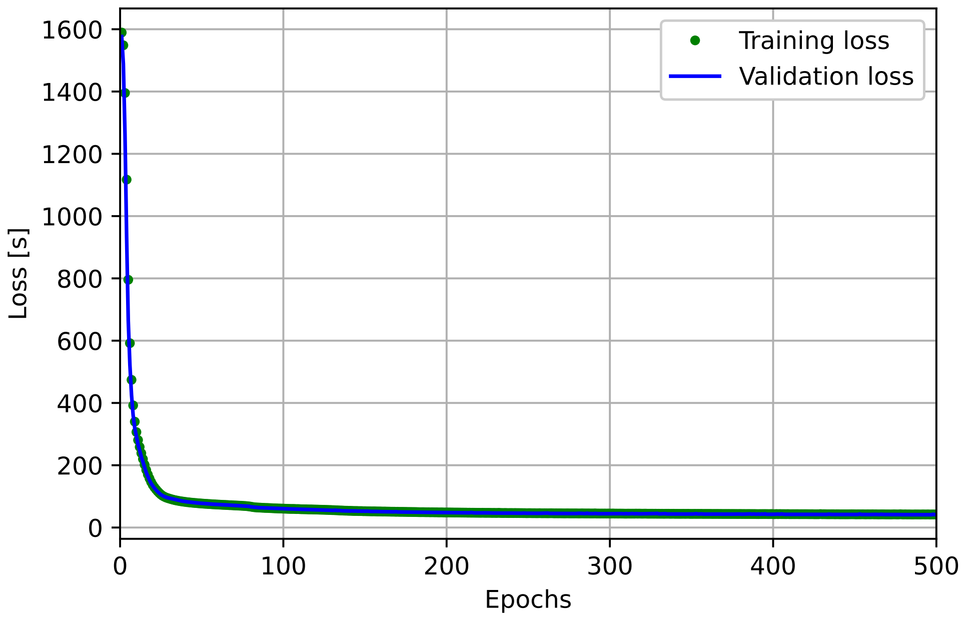
To design TFLM to be lightweight enough to run on memory-constrained devices, only a subset of the full Tensorflow framework is supported. This limitation is a trade-off between model complexity and runtime efficiency. Whilst a variety of common model types are supported, such as Neural Networks (NN), features required by more specialised models may not be supported. If the original model to be converted for TFLM support requires features that are not available, model rework will be required to find equivalent operations that are supported. In some situations, this may require replacing one non-supported operation with a combination of supported operations. These substitutions may add complexity to the model, increasing the time required to perform an inference, and increasing the overall size, limiting the remaining space on the memory-constrained device for other models. Hence, the supported operations currently limit this approach to ML inferencing at the edge.

With the limitations of both the computational power of the edge device and the operation support of the TFLM framework, simplifications were made for model development to reduce complexity and increase the likelihood of performing useful inferencing at the edge. Hence, a relatively simple feedforward neural network with four dense layers was chosen for the ML application. The input layer is a normalisation layer, which takes in the six temperature sensor features and preprocesses them before they enter the NN. This layer was added to the model to standardise the input features for stable and efficient training. The first hidden layer takes in six normalised values. The layer comprises 16 neurons and uses the Rectified Linear Unit (ReLU) activation function. This activation function was chosen due to its commonality in NN, support in TFLM, and ability to learn complex non-linear patterns in the data. It outputs 16 values. The second hidden layer also comprises 16 neurons and uses the ReLU activation function. This layer accepts 16 values from the previous layer and also outputs 16. The second layer was added to better account for the non-linearity of the relationship being modelled. The output layer consists of a single neuron and no activation function. The layer accepts the 16 values from the second hidden layer and uses them to produce a single output value representing the estimated time remaining until the battery heaters are enabled. An Adam optimiser with a mean squared error loss function was used for model training. Each additional layer and the number of neurons added to the model increases the model's complexity, correlating to increased inference time and resource usage on the edge device. The chosen simplified model architecture allows for some compensation for the non-linearity of the relationship being modelled whilst remaining modest with system resource usage. The model's accuracy would benefit from increased model depth or using more neurons on each layer. However, the corresponding model complexity may have resulted in the model being unfeasible to run onboard a memory-constrained platform. Training the resulting model using TensorFlow with a batch size of 64 over 500 epochs, the model reached a loss error of roughly 41 seconds on both the training and validation datasets, as shown in Figure 5.13.

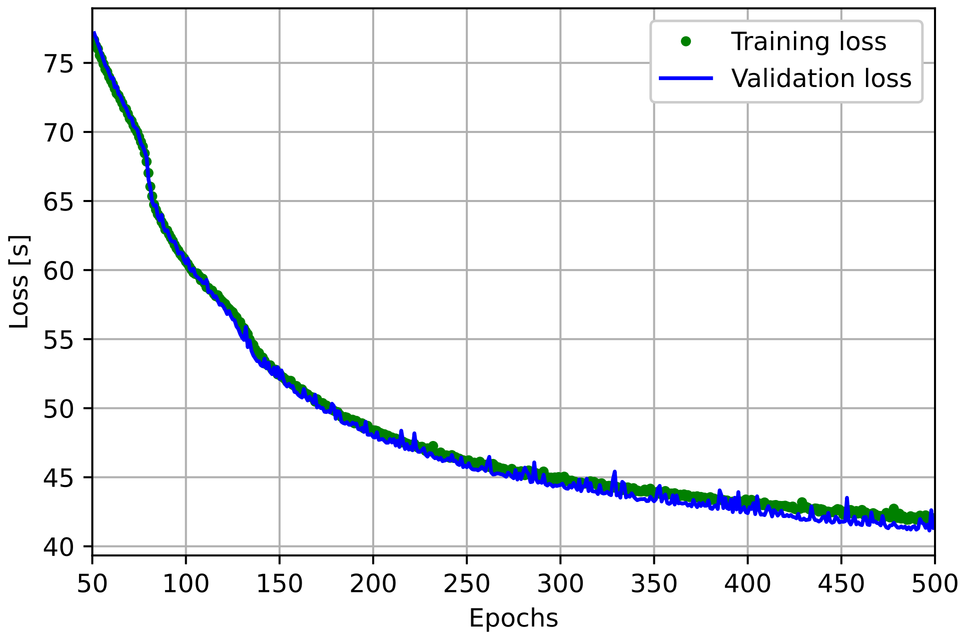
5.4.2. Results and Discussion

The trained model was tasked with making predictions of the time remaining until the battery heaters are enabled using the reserved testing dataset from the sixth cold swing recorded in the Wombat thermal vacuum chamber. Figure 5.14 shows the accuracy of the model inference plotted with the actual time remaining until heaters were enabled, where $t_{heaters\_on}$ refers to the time until the heaters are enabled, T_{enable} refers to the enable temperature of the battery heaters, and T_i refers to the average internal temperature of the spacecraft measured at the batteries. The linear model produced from Chapter 4 is also included for comparison.

The trained model has a much higher prediction accuracy than the developed linear model. With the TensorFlow model showing a big improvement in accuracy over the initially developed linear model, it was converted to a TFLM model for validation on the flight computer. X-CUBE-AI, a software package developed by STMicroelectronics, was used to convert the TFLM model into C code to be integrated into the BSF. The microcontroller-ready library requires 17.46 KiB of Flash memory, equating to roughly 1.7% of the available 1024 KiB reserved for the digital twin. This is a significant increase over the 0.71 KiB required for the previous conventional modelling approach. During runtime, the library requires 2.59 KiB of RAM, equating to roughly 0.37% of the available 704 KiB reserved for the digital twin. The modest requirements for system resources demonstrate that the learned model can be deployed on the computationally



(A) A comparison between the training and validation loss during model learning.



(B) A zoom of the loss function, showing the rate of change of loss decreasing with increasing epochs. The loss curve does not show clear signs of a plateau, meaning a slight decrease in loss may be achieved with more epochs.

FIGURE 5.13: A comparison between the training and validation loss during model learning. The loss from the training and validation sets is shown to initially be consistent before beginning to diverge as the epochs increase. A slight decrease in loss may be achieved with more training time; however, the achieved sub-one-minute prediction error is deemed acceptable for this demonstration.

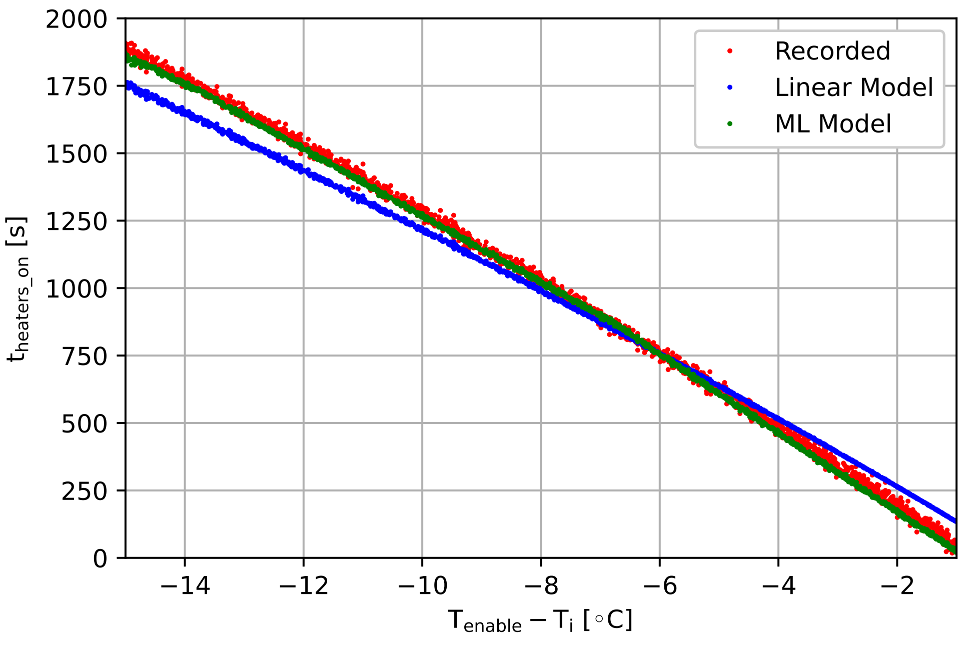


FIGURE 5.14: Inference from the trained ML model of the time remaining until the spacecraft battery heaters are enabled plotted against the recorded data from the sixth cold swing in the Wombat XL. The linear predictive model developed in Chapter 4 is included to compare accuracy.

constrained platform without encroaching on the system peripherals required for flight logic operation. The testing dataset comprising the sixth cold swing done at the Wombat thermal vacuum chamber was again used for validation on the spacecraft flight computer. The average time required to perform an inference using the trained model was 0.012 ms in comparison to 0.00535 ms for the linear model. The conversion process introduces negligible discrepancy in inference compared to the TensorFlow model, shown in Figure 5.15.

During the conversion stage, trained models can be compressed to reduce Flash usage. Compressed, the generated library requires 16.38 KiB, reducing 1.08 KiB in Flash usage. RAM requirements and average inference time are unaffected. Compressing the model, however, introduces a higher error margin, shown in Figure 5.16. Inferences on the dataset using the desktop TensorFlow model and both compressed and uncompressed onboard models are plotted together in Figure 5.17 as a comparison. The error between the uncompressed onboard model inference and the desktop TensorFlow inference is negligible. In contrast, the compressed model adds a noticeable error

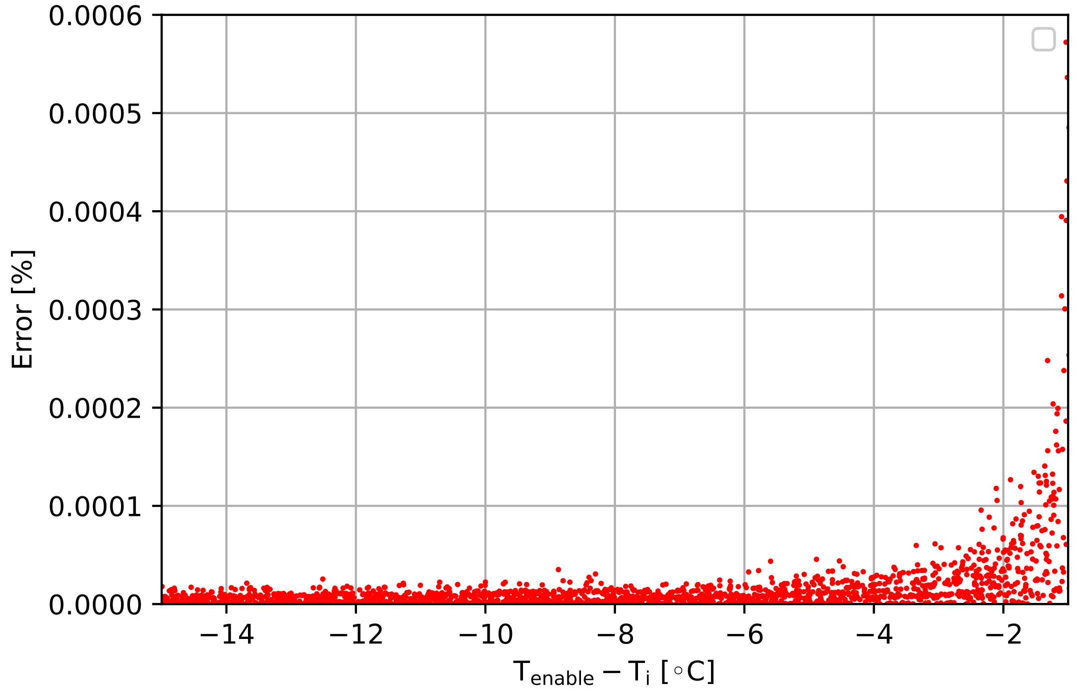


FIGURE 5.15: Percentage error discrepancy between inferences performed on desktop and the spacecraft hardware. The error is negligible throughout the useful range of the model, with an increase in error as the enable temperature is approached.

margin, underpredicting the time remaining till the heaters enable for the majority of the test dataset. Table 5.3 provides a comparison of the system resource requirements for the linear, compressed, and uncompressed ML models.

The developed onboard digital twin platform has been shown to support modular inference upgrades by implementing an ML backend for the battery heaters usage prediction without requiring any modifications to the flight software hosting the digital twin client. Using TensorFlow, training the regression model using the same dataset

TABLE 5.3: A comparison of the system resource requirements for performing battery heater usage prediction onboard the spacecraft against the reserved testing dataset.

Model	**Flash Size (KiB)**	**Average Inference Time (ms)**	**Average RMS Error (s)**
Linear	0.71	0.00535	70.15
ML Uncompressed	17.46	0.012	21.39
ML Compressed	16.38	0.012	56.96

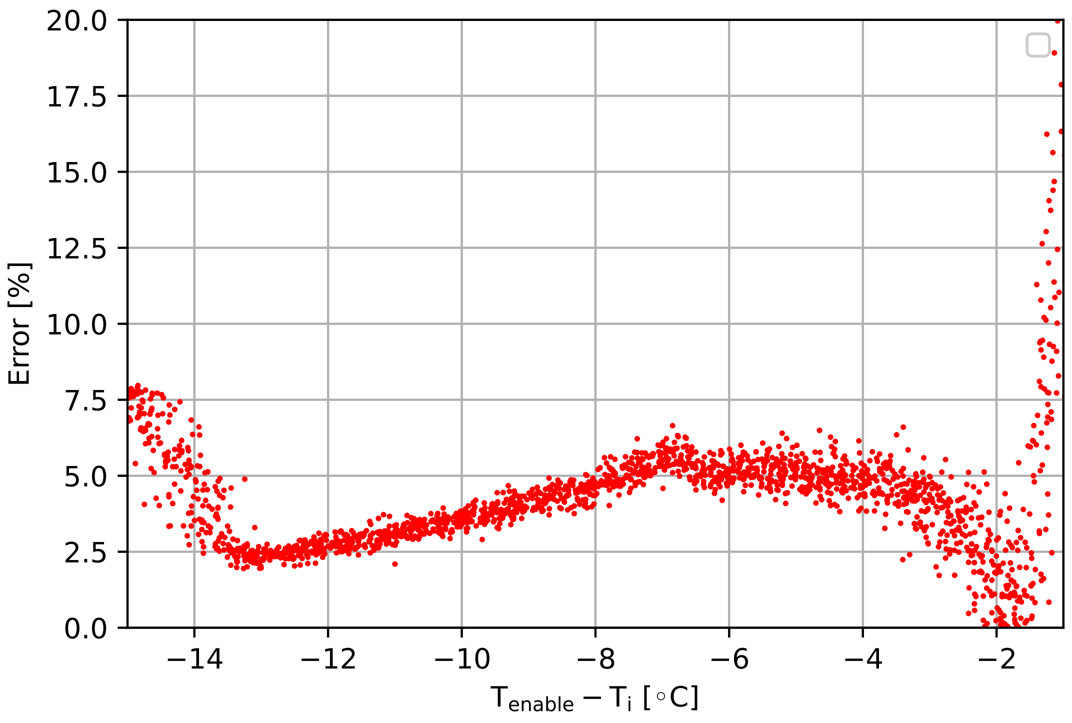


FIGURE 5.16: Percentage error discrepancy between inferences performed on desktop and the spacecraft hardware with the compressed model. The error is significantly higher than Figure 5.15.

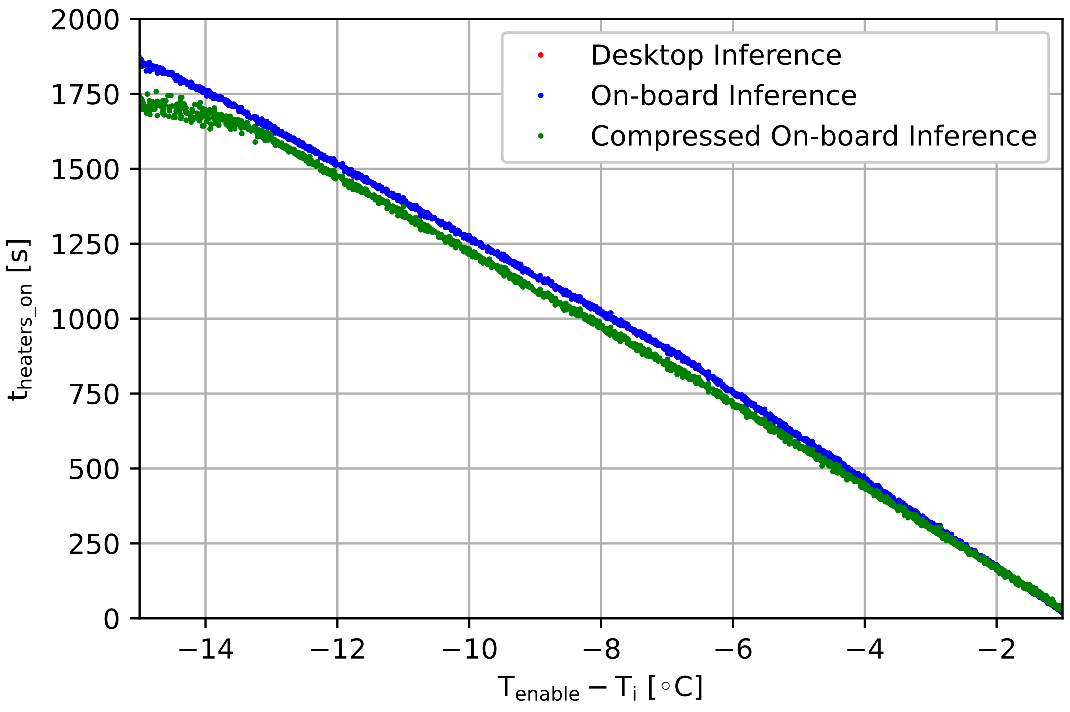


FIGURE 5.17: A comparison between inference accuracy made on the desktop, onboard using the uncompressed model, and onboard using the compressed model. The inferences from the uncompressed model have negligible error in comparison to the desktop model. Noticeable errors are present in the compressed model.

used for developing the conventional linear model from Chapter 4 offers a direct comparison in accuracy, resource usage and inference speed. It was observed through the correlation matrix data analysis that the two internal battery heaters had differing correlations with the temperature on the external faces of the spacecraft. The correlation between all solar panel surface temperatures and the temperature of the heaters for batteries three and four showed an average 86.5% correlation, as opposed to a 90% correlation for batteries one and two. The reduced correlation could be caused by inhomogeneity in the thermal conduction of the spacecraft structure. This highlights an advantage of using an ML approach. Training an ML model with knowledge of this correlation can allow it to adjust the weights applied to data on that sensor axis, compensating for the nonlinearity in the system. Comparing the inferences performed by the TensorFlow model and the linear model on the host machine showed a higher accuracy in the prediction of the usage of the battery heaters using the trained TensorFlow model, achieving an average error of roughly 41 seconds. From the plot of training and validation loss, it is expected that a higher accuracy could be achieved by allowing the training to run over more epochs. However, the scope of this section was to demonstrate the feasibility of performing inference of a trained ML model on the spacecraft hardware; hence, the achieved error is deemed satisfactory. In preparing the trained model for inference on a microcontroller using TFLM, only 1.7% of the Flash space reserved for the onboard digital twin was required. Whilst requiring more system resources than the conventional modelling approach, the still-modest requirements show that the flight computer can support multiple trained models for onboard inference using the digital twin platform. Performing an inference was found to take over twice the time required for an inference using the linear model. However, it is important to note that the trained model offers significant benefits over the linear model in terms of accuracy, making the increase in inference time a reasonable trade-off. When the digital twin platform is extended to add further onboard inferences, including from ML, model compression is available to reduce Flash usage at the expense of accuracy. It is acknowledged that the model may be overfitting the trained data due to the small scope of the dataset used. The model is expected to predict the time remaining until the battery heaters enable with high accuracy when used in the Wombat thermal vacuum chamber but may have less accuracy in a general environment. The achieved accuracy is deemed acceptable for this demonstration however, as this section aimed to discover if the computationally constrained computer used on the BCM can perform onboard ML inference. Moreover, from an analysis of an open source dataset comprising four 1U CubeSats in an International Space Station deployed orbit, the thermal fluctuation is seen to be predictably consistent [167]. Thus, an ML model trained from on-orbit data is expected to accurately perform useful inferences when exposed to similar orbital environments in future missions.

5.5. CONCLUSIONS AND FUTURE WORK

With the continually increasing complexity of spacecraft systems, the data collected by onboard sensors is growing exponentially. The burden on communications, and as a result, the EPS, to downlink this data can have negative consequences on spacecraft operations. Moreover, expecting ground operators to identify faults in the large data volumes collected can lead to undetected faults. Compensating for this, FDIR routines are adapting to advanced approaches that extend beyond a reactionary dump into safe mode toward attempting to verify faults through model-based inferences. In recent years, work in this field has been brought closer to the edge, accelerated by the requirements of the growing IoT industry. However, more work needs to be done in the literature on developing FDIR routines capable of predicting and preventing faults in advance. Where this topic has been researched, it has only been trialled on offline simulations. This chapter aimed to address the gap in the literature for developing intelligent predictive algorithms to be used by FDIR routines for fault prediction capable of running onboard spacecraft hardware. It demonstrated through experimentation that the resource-constrained flight computer enabling the BCM could perform onboard inference with the digital twin platform. This capability onboard a spacecraft is essential for achieving higher levels of autonomy and is necessary for spacecraft operating in a communications-limited environment.

Development of the digital twin platform saw a strong focus on breaking dependency on hardware. This was achieved by implementing an RPC approach, adopting a separate server and client for performing remote inferences. An application-specific

ChannelIO object implemented the communication between the server and the client. As this is the only area that requires knowledge of the platform the digital twin is running on, the developed library supports portability across various system architectures. Moreover, this implementation made it possible to develop the digital twin under TDD principles by making fake ChannelIO implementations that allow for direct manipulation of transferred buffers. The granular testability proved the platform's robustness, which was necessary as the intention is for the flight computer to use it for autonomous fault prediction. After the digital twin platform was developed, it was ported to the spacecraft hardware by providing a communication implementation between the two cores on the flight computer.

The design decision to adopt a client/server model for the digital twin platform provides flexibility in upgrading the inference backend. As the digital twin client and server run in separate firmware onboard the flight computer, the digital twin server firmware can be updated without necessitating the same for the digital twin client. This allows the inference models running on the digital twin server to be updated using on orbit data collected for a mission without the risk of breaking flight application software. The robustness of the novel design will assist in iteratively developing the digital twin platform on the Binar-2, 3 and 4 satellites on orbit with low risk.

A data replay backend for the objects abstracting the hardware from the application code was created to test the reliability of the onboard digital twin platform. This backend offered a repeatable way to perform HITL testing of the digital twin implementation with varying starting conditions. The data selected for the HITL testing was taken from the sixth cold swing of the thermal vacuum experiment done at ANU. The conditions for the flight logic to preventatively cancel a task were required to be having the battery heaters used during task execution, as well as the future battery capacity prediction to drop below 50%. From the three trials performed, the onboard digital twin platform was shown to reliably detect a potential future fault and recover from it when required. The developed platform contributes to the research field concerned with implementing and using intelligent predictive algorithms able to run onboard spacecraft hardware to prevent faults on orbit.

The modularity of the developed digital twin platform was shown to support upgrades to the inference engine without impacting the flight logic core. This was demonstrated by updating the conventional linear model used for the battery heaters usage prediction to an NN trained on the same dataset collected at ANU. Using the same dataset as the conventional linear model enabled a direct comparison of the accuracy of the inferences. Transferring the trained NN onto the flight computer showed it only required a modest amount of resources. Performing inferencing using the reserved testing set (the sixth cold swing recorded at ANU) showed that the trained model had a higher prediction accuracy than the conventional modelling. This shows the improvements that could be achieved for spacecraft autonomy by combining the digital twin platform with onboard inferencing of training machine learning models.

From converting pre-trained machine learning models to software compiled into program memory, it was evident that a significant increase in system resources was required over a conventional modelling approach. Whilst the model trained for demonstration in this chapter only used 1.7% of the Flash memory allocated for the onboard digital twin implementation, a more complex model is expected to require more system resources. With the digital twin platform anticipated to grow to support multiple onboard trained model inferencing, Flash space may become constrained. The use of model compression was explored at the expense of inference accuracy. The possibility of storing trained models in the onboard storage (OBS) will be explored to circumvent this issue. Whilst the microcontroller core hosting the digital twin client is limited to 1024 KiB, the OBS implemented on the BCM has 1 Gbit of Flash storage, with footprint compatible support for a 2 Gbit replacement. The larger storage potential will significantly increase the number of trained model inferences possible onboard the spacecraft without sacrificing accuracy.

With the extensibility of the developed digital twin platform demonstrated, the scope of onboard inferences can grow with minimal effort to support mission requirements. One such application to be explored in future work is the development of a digital twin module that uses Deep Reinforcement Learning (DRL). The machine learning case study explored in this chapter is inherently a regression problem. Specifically, the input data from the onboard temperature sensors can be consistently mapped to the output, representing the time remaining until the battery heaters are enabled, and this relationship is unlikely to change. However, in the harsh environment of space, often spacecraft systems are dynamically changing, along with the faults that plague them. The ability to update policies for handling dynamically changing faults requires a model to explore the problem space and learn from successes and failures. Deep Reinforcement Learning (DRL) merges deep neural networks and reinforcement learning to enable automatic feature learning and end-to-end problem-solving. DRL empowers models to explore and learn from successes and failures, reducing the need for extensive domain knowledge. This results in more flexible and adaptable models [168].

Recent literature has shown increasing interest in the potential application of DRL to the space domain, often used for mission planning and resource utilisation. Furfaro et al. explore the application of DRL in the context of spacecraft guidance [169]. Their study investigated the development of an adaptive guidance algorithm for spacecraft performing soft landings autonomously. The contribution of their work demonstrates the potential of DRL techniques in enhancing spacecraft autonomy. More specifically, the adaptive guidance algorithm developed in their study demonstrates the capacity of DRL to enhance spacecraft operations by accommodating dynamic constraints and optimising fuel consumption during the descent phase of missions. Through DRL, their approach finds that spacecraft can better adapt to changing conditions and environmental variables, making them more self-sufficient in achieving successful landings. Harris et al. present an innovative approach to addressing complex spacecraft decision-making problems using DRL techniques by formulating them as Partially Observable Markov Decision Processes (POMDPs) [170]. The study investigates autonomous orbital insertion and station keeping in offline simulations, demonstrating the versatility of the proposed DRL framework in addressing different aspects of spacecraft control. The research shows that the DRL-based agent outperformed expert-designed policies in both scenarios, demonstrating promise for the potential of DRL for spacecraft autonomy.

Similarly, DRL algorithms have been used to optimise scheduling and task assignment for multiple simulated satellites in LEO [171]. Notably, DRL approaches have led to the development of spacecraft attitude controllers that surpass industry-standard pointing accuracy in simulation environments [172]. Furthermore, the applicability of DRL-based methods to the complex task of mapping and navigation around small, unknown celestial bodies has been discussed, offering potential solutions for future space exploration challenges [173]. Whilst shown to be feasibly deployed on flight-suitable hardware [174], it is extremely common to see the application of DRL in the literature to be limited to offline simulations, in part due to the computational constraints of spacecraft hardware [173]. Whereas the NN discussed in this chapter has fixed weights, DRL models require ongoing learning and complex policies that may be beyond the limits of modern embedded hardware to be used effectively.

Moreover, whilst mission planning and resource utilisation are common topics in the literature for DRL applications, their applicability to FDIR routines is seldom mentioned. It is the opinion of the candidate that this area holds the greatest potential for enhancing a space mission. An entry into an FDIR routine due to a communications system fault, such as a transceiver malfunction or signal interference, can be challenging to recover from due to the dynamic nature of the failure. In such a failure, recovery options are often context-dependent. They will vary greatly depending on the specific fault scenario, be it adjusting transmission power, changing data compression, or attempting data error correction. The dynamic nature of these faults would require exploration strategies to identify the best solution. A NN, such as the one demonstrated in this chapter, would be a poor choice for this problem as it does not inherently incorporate exploration methods and rather is typically trained offline on a static dataset with the learned weights and biases being fixed. In contrast, the exploration-exploitation approach taken by a DRL agent would allow the agent to attempt different recovery actions and learn from their outcomes. The agent will learn from past communication fault incidents and will adapt the chosen recovery procedures to address similar issues if they occur again in the future.

The concept of a digital twin, as discussed in this thesis, entails a virtual replica of an engineering system unparalleled in exactitude. At the beginning of the mission, the developed digital twin modules will meet that concept. However, throughout a mission in a dynamic space environment, the digital twin and the physical spacecraft can likely diverge in similitude. The result will be more akin to an 'ideal' digital twin platform that is unable to provide useful feedback to the physical spacecraft, especially during FDIR routines, as the models statically assume perfect spacecraft health. A digital twin module running a DRL agent can adapt its policy based on real-time data and unpredictable situations as the spacecraft ages on orbit, allowing for a twin with exacting similitude for the full mission lifecycle. As the technology for utilising machine learning at the edge continues to develop rapidly, it is expected that the modest hardware powering small spacecraft will become capable of using embedded DRL models. Future work will deploy DRL-based FDIR routines on small spacecraft using the candidate's digital twin framework.

Alongside inferences that prevent faults on orbit, the digital twin platform can also be extended to provide lost value estimation. Part of the inference chain used in this chapter provided predictions on the usage of spacecraft battery heaters using thermal sensor data. However, in the hostile space environment, spacecraft sensors can fail. Whilst redundancy is an option, it may only be possible on some sensors due to the small form factor of the CubeSat. Guo et al. proposed a solution to this problem using the Space Shuttle Main Engine. As a novel approach, system data was used to train an NN to understand the relationships between onboard sensors. These relationships formed an influence sensor map to show how other measurements can directly influence a measurement. The produced network was then used to check the validity of sensor readings and to provide an estimated value for an identified faulty sensor [175]. The positive results showing the approach's feasibility were collected from execution on the Digital Transient Model of the Space Shuttle Main Engine [175], rather than deploying onto target hardware.

Similarly, the feasibility of lost value estimation was again demonstrated by the recovery of a joint angle from a simulation of a robotic manipulator on-orbit [176]. Lost value estimation, however, has yet to be demonstrated on orbit for mission recovery. The real-time capabilities of the flight computer selected for the BCM provide a platform to extend this research. An interrupt-driven failure detection approach can be used to identify faulty sensor readings, simplifying the process implemented by Guo et al. by removing the need to perform onboard inference at this stage. Using sensor readings from Binar 2, 3 and 4 on orbit, nominal operation ranges for sensor readings can be understood from statistical analysis to invoke a recovery interrupt when the readings stray from this range. The interrupts triggered from a faulty reading can polymorphically substitute a replacement object for the backend of a sensor at runtime to instead invoke the digital twin client to estimate a correct value for the sensor on orbit for recovery.

Extending on this concept, virtual sensors can be included onboard the spacecraft, 'sampled' using the onboard digital twin platform. This appeals to small form factors that may not have enough peripheral resources to include extra sensors or where sensor placement is deemed too difficult. An application on the Binar CubeSat platform is temperature estimation at the magnetorquer coils. As the torque produced by the magnetorquer coils is proportional to the current flowing through the windings, the torque produced is inversely proportional to the resistance of the winding. This means that the torque produced will not be constant with fluctuating temperatures on-orbit. Including temperature sensors to compensate for this is difficult due to the placement of the magnetorquer coils and insufficient remaining ADC channels on board. Using an external temperature probe, an NN can be trained to understand the relationship between the temperature at the coils and other sensors onboard the satellite. The digital twin platform can then be extended to encompass this inference to estimate the temperature of the coils on orbit, allowing attitude control algorithms to compensate for it.

An onboard digital twin also shows an application for mission planning without intervention from the ground. Two features currently in development by the candidate for the digital twin platform are an embedded implementation of the simplified general perturbations (SGP) model for orbital propagation and the international geomagnetic reference field (IGRF) magnetic field prediction. These inferences allow the spacecraft to forward propagate on-orbit and estimate the magnetic field intensity at the new position. Chaining these inferences could aid in autonomously preparing attitude control commands without requiring planning from the ground.

The high-performance Cortex-M7 core's capabilities are sufficient for the current implementation of the onboard digital twin. However, as the scale of the digital twin platform grows, the microcontroller core's processing throughput may start to lag with high-frequency complex digital twin requests. In these situations, if the frequency of queries to the digital twin server is higher than the speed of returning a result to the client, it could potentially result in a backlog of digital twin queries, causing operational problems. Such a backlog could threaten a mission if the digital twin query is part of an FDIR routine. In anticipation of these potential problems, the digital twin RPC server was developed to be decoupled from the hardware it runs on. Decoupling the communications channel from the RPC server allows the server to move to multiple separate onboard microprocessors instead of the secondary flight computer core without impacting the RPC logic. Moving toward a distributed computing system can prevent latency issues from high-frequency complex queries. Moreover, the RPC framework's cross-language compatibility allows the server to be written in a different language than the client to leverage other programming languages' speed and efficiency for future onboard inferences, further alleviating potential latency issues.

Accelerated by the growing global interest in IoT devices, machine learning inferencing at the edge has become possible. In the coming years, the field of edge computing is expected to develop capabilities for on-device learning [163]. This could provide unprecedented benefits to spacecraft autonomy, giving a satellite the tools required to learn how it interacts with the environment it operates in whilst on orbit. When this technology is developed, the digital twin platform can be adapted to support self-upgrades of existing inference procedures when more data about the environment becomes available.

A sensor data replay method was developed to provide a repeatable method for testing the reliability of the digital twin to suggest task cancellations. Using on-orbit sensor data collected by Binar 2, 3 and 4, the HITL simulation replays can be extended. As flight logic continues to undergo development, on-orbit data can be replayed into the application to verify the response. Moreover, where a failure may have occurred on orbit, the data leading up to the failure point can be replayed on the ground engineering model to understand the chain of events that led to system failure. This can aid in developing robust firmware and a better understanding of how to recover from failure when it occurs.

REFERENCES

[72] A. Wander and R. Förstner, "Innovative Fault Detection, Isolation and Recovery Strategies On-Board Spacecraft: State of the Art and Research Challenges," en, *Conference on Control and Fault-Tolerant Systems, SysTol*, p. 9, 2012. DOI: 10.1109/SysTol.2013.6693950.

[136] The Pigweed Authors, *Pigweed*, Mar. 2023. [Online]. Available: https://github.com/google/pigweed.

[156] W. G. Schneeweiss, "Advanced Fault Tree Modeling," *Journal of Universal Computer Science*, vol. 5, no. 10, pp. 633–646, Oct. 1999. DOI: 10.3217/JUCS-005-10-0633. [Online]. Available: https://lib.jucs.org/article/27601/.

[157] X. Olive, "FDI(R) for satellites: How to deal with high availability and robustness in the space domain?" eng, *International Journal of Applied Mathematics and Computer Science*, vol. 22, no. 1, pp. 99–107, 2012, ISSN: 1641-876X. [Online]. Available: https://eudml.org/doc/208103.

[158] L. Portinale, D. Codetta-Raiteri, A. Guiotto, and S. DiNolfo, "ARPHA: A software prototype for fault detection, identification and recovery in autonomous spacecrafts," *Acta Futura*, pp. 99–110, Jan. 2012. DOI: 10.2420/ACT-BOK-AF05.

[159] Y. Kato, T. Yairi, and K. Hori, "Integrating Data Mining Techniques and Design Information Management for Failure Prevention," en, in *New Frontiers in Artificial Intelligence*, T. Terano, Y. Ohsawa, T. Nishida, A. Namatame, S. Tsumoto, and T. Washio, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2001, pp. 475–480, ISBN: 978-3-540-45548-6. DOI: 10.1007/3-540-45548-5_65.

[160] *ECSS-E-ST-70-11C: Space engineering Space Segment Operability*, en, Jul. 2008.

[161] *Open-amp*, Apr. 2023. [Online]. Available: https://github.com/OpenAMP/open-amp.

[162] S. Jamal, S. Goyal, A. Grover, and A. Shanker, "Machine Learning: What, Why, and How?" In *Bioinformatics: Sequences, Structures, Phylogeny*, A. Shanker, Ed., Singapore: Springer Singapore, 2018, pp. 359–374. DOI: `10.1007/978-981-13-1562-6_16`. [Online]. Available: `https://doi.org/10.1007/978-981-13-1562-6_16`.

[163] B. Murmann and B. Hoefflinger, *NANO-CHIPS 2030: On-Chip AI for an Efficient Data-Driven World*, en, ser. The Frontiers Collection. Cham: Springer International Publishing, 2020, ISBN: 978-3-030-18338-7. DOI: `10.1007/978-3-030-18338-7`. [Online]. Available: `http://link.springer.com/10.1007/978-3-030-18338-7`.

[164] M. G. S. Murshed, C. Murphy, D. Hou, N. Khan, G. Ananthanarayanan, and F. Hussain, "Machine Learning at the Network Edge: A Survey," *ACM Computing Surveys*, vol. 54, no. 8, 170:1–170:37, Oct. 2021, ISSN: 0360-0300. DOI: `10.1145/3469029`. [Online]. Available: `https://doi.org/10.1145/3469029`.

[165] M. Merenda, C. Porcaro, and D. Iero, "Edge Machine Learning for AI-Enabled IoT Devices: A Review," en, *Sensors*, vol. 20, no. 9, p. 2533, Apr. 2020, ISSN: 1424-8220. DOI: `10.3390/s20092533`. [Online]. Available: `https://www.mdpi.com/1424-8220/20/9/2533`.

[166] The TensorFlow Authors, *TensorFlow Lite for Microcontrollers*, Apr. 2023. [Online]. Available: `https://github.com/tensorflow/tflite-micro`.

[167] A. Jara, P. Lepcha, S. Kim, H. Masui, T. Yamauchi, G. Maeda, and M. Cho, "On-orbit electrical power system dataset of 1U CubeSat constellation," en, *Data in Brief*, vol. 45, p. 108 697, Dec. 2022, ISSN: 2352-3409. DOI: `10.1016/j.dib.2022.108697`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S2352340922009027`.

[168] Y. Li, *Deep Reinforcement Learning: An Overview*, Nov. 2018. DOI: `10.48550/arXiv.1701.07274`. [Online]. Available: `http://arxiv.org/abs/1701.07274` (visited on 10/23/2023).

[169] R. Furfaro, A. Scorsoglio, R. Linares, and M. Massari, "Adaptive generalized ZEM-ZEV feedback guidance for planetary landing via a deep reinforcement learning approach," *Acta Astronautica*, vol. 171, pp. 156–171, Jun. 2020, ISSN:

0094-5765. DOI: 10.1016/j.actaastro.2020.02.051. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0094576520301247 (visited on 10/30/2023).

[170] A. Harris, T. Teil, and H. Schaub, "Spacecraft Decision-Making Autonomy Using Deep Reinforcement Learning," en, 2019.

[171] X. Wang, J. Wu, Z. Shi, F. Zhao, and Z. Jin, "Deep reinforcement learning-based autonomous mission planning method for high and low orbit multiple agile Earth observing satellites," en, *Advances in Space Research*, vol. 70, no. 11, pp. 3478–3493, Dec. 2022, ISSN: 02731177. DOI: 10.1016/j.asr.2022.08.016. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0273117722007335 (visited on 10/23/2023).

[172] J. Elkins, R. Sood, and C. Rumpf, "Autonomous Spacecraft Attitude Control Using Deep Reinforcement Learning," Oct. 2020.

[173] D. M. Chan and A.-a. Agha-mohammadi, "Autonomous Imaging and Mapping of Small Bodies Using Deep Reinforcement Learning," in *2019 IEEE Aerospace Conference*, Mar. 2019, pp. 1–12. DOI: 10.1109/AERO.2019.8742147. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8742147?casa_token=q_aA8MI5diUAAAAA:ygsORzfavSj33YJJ1JY3seAw4GQbs5ZqBNI4inTzJuSbXtqArPeQD1OIuhM2LXGgDmt3Ct4tOyk (visited on 10/23/2023).

[174] C. Wilson and A. Riccardi, "Enabling intelligent onboard guidance, navigation, and control using reinforcement learning on near-term flight hardware," *Acta Astronautica*, vol. 199, pp. 374–385, Oct. 2022, ISSN: 0094-5765. DOI: 10.1016/j.actaastro.2022.07.013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0094576522003502 (visited on 10/25/2023).

[175] T.-H. Guo and J. Nurre, "Sensor failure detection and recovery by neural networks," en, in *IJCNN-91-Seattle International Joint Conference on Neural Networks*, vol. i, Seattle, WA, USA: IEEE, 1991, pp. 221–226, ISBN: 978-0-7803-0164-1. DOI: 10.1109/IJCNN.1991.155180. [Online]. Available: http://ieeexplore.ieee.org/document/155180/.

[176] S. Jaekel and B. Scholz, “Utilizing Artificial Intelligence to achieve a robust architecture for future robotic spacecraft,” in *2015 IEEE Aerospace Conference*, Mar. 2015, pp. 1–14. DOI: `10.1109/AERO.2015.7119180`.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

The primary purpose of the work described in this thesis is to improve the reliability of satellite missions by reducing the contribution of software to their high failure rate. A major challenge impeding mission success, software faults were identified to exist across the full mission lifecycle, from development to operations.

The Binar Software Framework was designed by the candidate to improve mission success by addressing the shortcomings of previous failed space missions. An analysis of the missions identified a common failure overlap due to inadequate testing, documentation, code review and code reuse. The reliability of the BSF was ensured by maintaining a strict focus on these areas during development. Chapter 2 details this development cycle, incorporating examples from the BSF. Safe code reuse was emphasised, which involved designing software with portability through adherence to the SOLID design principles. These principles allow for the rapid mission-concept-to-orbit goal of the BSP to be achieved by enabling the reuse of the platform-independent hardware abstraction layer between missions. As demonstrated in other industries, adhering to time-tested principles leads to higher-quality software. The most frequent cause of mission failure, however, was found to be insufficient software testing. The BSF addresses this by promoting decoupled interfaces, enabling mocked modules to be substituted for hardware implementations. Breaking the dependency on hardware enabled the use of TDD principles. The importance of tests written during the TDD process was stressed to prevent late changes to software from causing mission failures. Using a CI/CD platform automates the execution of unit tests before the software is permitted to merge into the master branch, providing accountability for software quality. Code review also ensures strict documentation standards, helping future developers understand and maintain the code base. Only approved software from the upstream master repository was deployed onto Binar-1 to ensure error-free releases and clear version history. The concepts explored in this work can be applied to any small spacecraft to increase mission success and promote code reusability.

The capability of autonomy in satellites is currently being underestimated when it comes to fault prevention. The currently adopted approach is to identify an occurring fault, transition into a safe mode and await instructions from a ground operator. However, as missions push past LEO, the challenges associated with deep space can make this an unacceptable risk. Moreover, the growing complexity of spacecraft systems has led to vast amounts of data being collected by onboard sensors, making it challenging for ground operators to identify faults. Onboard autonomy needs to adapt to advanced approaches beyond simply reacting and dumping into safe mode, instead aiming to predict faults through model-based inferences. The developed digital twin platform aimed to address this gap in the literature by developing a platform in which intelligent predictive algorithms could be used onboard the satellite for fault prediction. The onboard digital twin platform was demonstrated using a computationally inexpensive thermal model of the Binar-1 Cubesat and a pre-trained ML model to predict a future fault. The results showed that the resource-constrained flight computer on the BCM could perform onboard inferencing with the digital twin platform, successfully preventing a low-power state from occurring on the satellite. This technology will become crucial for spacecraft operating in communication-limited environments.

6.1. CONTRIBUTIONS

This work has made a significant original contribution to the field of spacecraft software development by formulating software development recommendations based on past mission failures, designing and implementing a CubeSat software framework (the BSF) that incorporates these recommendations, which has been deployed on Binar-1 and will be deployed on the Binar-2, Binar-3 and Binar-4 spacecraft, developing and testing new types of onboard decision-making capabilities, and leveraging machine learning techniques on devices with limited resources.

Adopting a learning-from-incident approach to applying reliable software development practices to satellites represents a pioneering improvement to spacecraft software reliability. This approach generated recommendations to prevent the reoccurrence of common mission failures due to software:

- Reuse code safely through development with attention to portability.
- Ensure software reliability through test-driven development.
- Increase code base robustness and accountability by enforcing optimal code base maintenance.

The recommendations can be utilised to reduce software faults and increase the chances of mission success.

According to the developed software recommendations, a custom in-house software framework was developed for the BSP, allowing for the successful adoption of a rapid mission-concept-to-orbit model. By developing the software libraries comprising the framework in adherence to the SOLID principles of object-oriented programming, the safe reusability of the codebase is guaranteed through portability and extensibility. This approach has facilitated the BSP to quickly progress with the development of the Binar-2, 3 and 4 satellites. The modularity achieved from decoupling software modules from hardware has enabled the BSF to be developed using test-driven development ideology, increasing the reliability of the codebase. Moreover, the custom CI/CD pipeline automates building, testing, documentation and review on every commit, ensuring that the code which programs the Binar spacecraft has accountability.

The thesis advances spacecraft autonomy by exploring its benefits beyond navigation and payload optimisation, representing a significant advancement in small satellite software development. The developed novel onboard digital twin platform demonstrates the feasibility of using onboard autonomy for fault prevention. Through a unique modelling approach, the battery heating system usage on Binar-1 was simulated on the memory-constrained flight computer to predict critical low-power failures in advance. Other CubeSat groups can follow this approach to produce models capable of running on the memory-constrained flight computers enabling their missions to prevent faults during operations through onboard prediction. Furthermore, the digital twin platform was expanded to pioneer trained machine learning inferencing onboard the spacecraft for fault prevention. The results showing that the flight logic could autonomously use the digital twin platform to chain available inferences to predict an upcoming fault provides the groundwork for future research in this area, significantly improving the reliability and success of satellite missions.

6.2. FUTURE WORK

The contributions of this thesis lay the foundation for expanding the boundaries of satellite autonomy and resilience, paving the way for a new era of reliable and fault-tolerant space missions. Further work can be done to improve the effectiveness of the onboard digital twin and refine the software framework.

The findings from this research are partially limited due to the on-orbit performance of Binar-1. A hardware fault on the CubeSat prevented some of the on-orbit benefits of adopting the proposed software recommendations from being validated. The software framework will be deployed on Binar-2, 3 and 4, and the findings from adopting reliable software development practices will be disseminated to the community.

Future work on the BSF will see the codebase maintained to support future missions from the Binar Space Program. The strong modularity achieved from adherence to SOLID design principles enables the flight software to be decoupled from the hardware it runs on. Together with the data replay system developed for feeding recorded data into peripheral objects, a functional simulator of the Binar CubeSat platform will be developed. Platform simulation on a host machine will give ground operations a deeper insight into how the Binar CubeSat will respond to failures encountered on orbit. Host simulation will also provide a medium to verify the developed digital twin platform for fault prediction without requiring hardware.

Some apprehension still exists in the community about relinquishing control over the mission operations plan to autonomous software. CubeSats have been identified as a risk-accessible platform to trial advanced autonomy concepts before deployment on larger missions. The onboard digital twin platform, verified in a lab-based environment, will be tested on-orbit onboard Binar-2, 3 and 4. The efficacy of having a digital twin platform running onboard a memory-constrained CubeSat flight computer capable of predicting faults and suggesting changes to the mission operations plan will be disseminated to the community. The propositions for the application of autonomy presented in this thesis aim to increase community acceptance of relinquishing control over spacecraft operations to onboard autonomy.

Unmodelled factors in the computationally inexpensive models created were found to have caused an inconsistent error during experimental inferencing. Whilst these errors were addressed through implementing onboard machine learning inferencing, the increased accuracy comes with using more system resources. Further on-orbit testing, therefore, will contribute to the refinement of using computationally inexpensive modelling, potentially compensating for unmodelled factors without requiring increased system resources. Alongside this, on-orbit data will be collected for further testing of the capabilities of the CubeSat platform for onboard machine learning inferencing. With growing demand from the IoT industry, memory-constrained machine learning has gained traction and is being brought closer to the edge. However, further research is still needed to use this technology to develop FDIR routines that can predict and prevent faults in advance.

The digital twin platform was developed with a focus on extensibility. The scope of onboard inferences can grow with minimal effort to support future mission requirements. Short-term additions to the digital twin platform will see orbit propagation and magnetic field estimation available onboard. These tools will assist with the autonomous preparation of attitude control commands to meet mission goals without requiring planning from the ground. Further future work will see available inferences extend beyond this to satisfy the requirements of interplanetary missions.

This thesis applied the digital twin platform as a tool for fault prevention. Beyond this, however, other capabilities are apparent for having a virtual model of the spacecraft available onboard. One such application could be for lost value estimation. In the harsh conditions of space, onboard sensors can fail. The conventional approach to compensate for this is hardware redundancy. However, the small CubeSat form factor may make this unfeasible for every sensor onboard. The digital twin platform can be extended to create an 'influence sensor map' detailing how sensor measurements onboard the spacecraft are correlated. Using an interrupt-driven failure detection approach for identifying faulty sensor readings, the digital twin 'influence sensor map' can substitute approximate sensor values. This technology can aid in extending a mission's lifetime when components start to degrade. As an extension, virtual sensors can be included onboard the spacecraft, 'sampled' using the digital twin platform. Through experimentation on the ground, sensors onboard the spacecraft can be correlated with external sensors placed at desired locations around the spacecraft. Models created from the correlations can then be integrated into the digital twin platform. This application immensely benefits small satellite form factors that may not be able to include a physical sensor due to sampling constraints or difficulty in placement.

Alongside flexibility in upgrading the inference backend, the design decision to implement the digital twin platform using a client/server approach decouples the client and server from needing to run on the same computer. As the digital twin server's requirements grow in complexity, the server can be ported to other processors to leverage their speed and efficiency for future Binar missions. A successful port of this would be a precursor to adopting the digital twin platform on larger-scale spacecraft.

In a demonstration of the application of the digital twin platform, this thesis experimented with onboard ML inferencing. The next barrier for this technology will be performing ML training onboard the spacecraft without intervention from the ground. The separation between the digital twin client and the digital twin server aids in trialling this technology without the risk of impacting flight logic. Having this capability onboard a spacecraft is paramount for fault prevention. It would allow the spacecraft to adapt to changing environmental conditions. In Earth's orbit, solar weather increases atmospheric drag, resulting in an accelerated orbital decay [177]. A spacecraft flying an orbital propagator might start to stray in similitude to propagation simulations if it does not account for the influence of solar weather. Onboard ML training capabilities can help compensate for changes in the orbital environment. As a secondary payload on a mission, the injected orbit of a CubeSat may often have some deviation from what was expected. In the coming years, CubeSats will more commonly be placed into lunar orbit. Lunar mascons are known to degrade satellite orbits. Suppose a secondary payload CubeSat is injected into an unstable orbit. In that case, onboard ML training can be leveraged to learn more about the orbit, contributing to autonomous mission planning to meet mission objectives effectively. The digital twin platform can be extended to support self-upgrades of existing inference procedures when more data about the spacecraft's environment becomes available.

6.3. FINAL REMARKS

In conclusion, the thesis offers a solution to the high failure rate of spacecraft software by introducing reliable software development practices to flight software demonstrated on a CubeSat and proposing the development of an onboard digital twin capable of predicting and preventing faults during mission operations. These contributions aim to ensure the success and longevity of space missions and contribute to the field of space systems engineering. The results of this study demonstrate the viability of a cutting-edge approach to satellite autonomy and resilience, with practical applications that can revolutionise the space industry. As this field continues to evolve, the insights and innovations presented in this thesis will undoubtedly play a pivotal role in shaping its future trajectory.

REFERENCES

[177] V. U. J. Nwankwo, S. K. Chakrabarti, and R. S. Weigel, "Effects of plasma drag on low Earth orbiting satellites due to solar forcing induced perturbations and heating," en, *Advances in Space Research*, vol. 56, no. 1, pp. 47–56, Jul. 2015,

ISSN: 0273-1177. DOI: `10.1016/j.asr.2015.03.044`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0273117715002562`.

APPENDIX A

TECHNICAL SOFTWARE DETAILS

This appendix contains the technical details of a selection of software libraries developed for the Binar Software Framework. The modules outlined in the subsequent sections are entirely original works and constitute a significant portion of the overall efforts undertaken as part of this PhD.

A.1. TECHNICAL DETAILS OF THE GPIO LIBRARY

The GPIO library in the BSF abstracts the interaction between the flight computer and hardware digital signals. It provides two main objects for controlling inputs and outputs: GPI and GPO.

The GPI object provides the ″IsHigh″ method to poll the pin the templated derivation observes, checking for a voltage above the internal reference threshold. The GPO object provides three methods for controlling the hardware output pin:

- ″SetLow″: sinks the pin to ground
- ″Toggle″: inverts the existing state of the object
- ″SetHigh″: pulls the pin to 3.3V

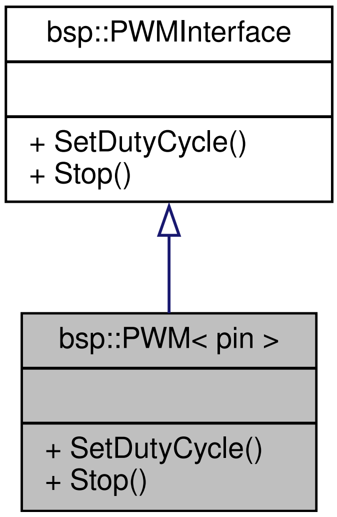


FIGURE A.1.1: A UML representation of the PWM object from the GPIO library. The PWM object is templated on the magnetorquer control pins.

GPOs can also provide controlling signals when configured in pulse width modulation (PWM) mode. The PWM object in the GPIO library (Figure A.1.1) provides this functionality. Through the ”SetDutyCycle” method, the application code can configure the duration for which the output PWM signal is high. This signal will constantly output until the ”Stop” method is called, or the PWM object goes out of scope. On Binar-1, PWM outputs were used for controlling the magnetorquers.

A.2. TECHNICAL DETAILS OF THE ANALOG LIBRARY

The Analog library abstracts how the flight computer samples voltages using its ADC. It provides the high-level Analog module for achieving this. The Analog module uses three internal objects to collate the relevant information: AnalogVoltageReader, AnalogCurrentReader, and AnalogTemperatureReader.

The public-facing AnalogVoltageReader object inherits and implements the same interface as the implementation object that it privately aggregates. The implementation object privately aggregates the four system voltages to be read; battery voltage (power_vbat_), X-axis solar panel voltage (power_solar_panel_xx_), Y-axis solar panel voltage (power_solar_panel_yy_), and 3V3 regulated voltage (power_3v3_). Each object implements the templated Analog object, providing the abstraction over their analog

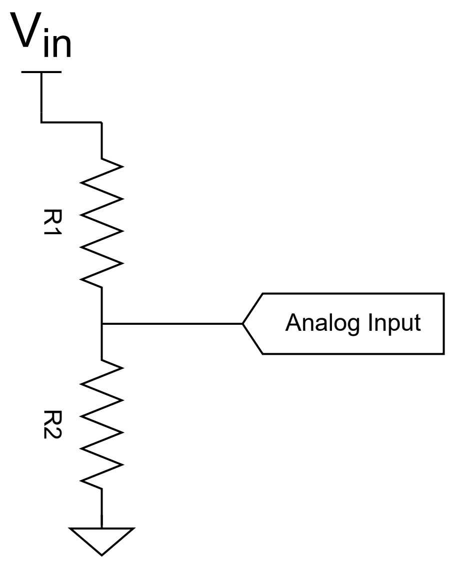


FIGURE A.2.1: A resistor divider circuit, ubiquitous on the BCM. It scales the voltage to be sampled into an accessible range by the flight computer.

inputs. Calling GetVoltage on these objects calls the internal "GetRaw" method to perform the hardware sampling, returning a 16-bit integer.

As voltages on the flight computer can only be sampled under the reference voltage of 3.3 V, most analog inputs must first be scaled using a resistor divider network shown in Figure A.2.1.

The voltage read at the analog input pin is scaled from the input voltage, V_{in}, by Equation A.2.1:

$$V_{analog\_input} = V_{in} * \frac{R_2}{R_1 + R_2} \tag{A.2.1}$$

The analog voltage at the input pin is then compared against the reference voltage of 3.3 V. With a resolution of 16 bits, the resultant raw integer value from the ADC will then be within the range of 0 to 65,534, mapping to 0 V to 3.3 V. As the various templated Analog objects possess different resistor values for their respective divider networks, the objects encapsulate the values of the resistors required to convert the raw reading into a floating point voltage to be used in application code. Encapsulating the resistor values within the implementation objects enables conversion methods to be easily updated with developing hardware. Calling "GetVoltages" on the Analog-

VoltageReader implementation object will call the "GetVoltage" method on all private templated objects, constructing the returns into a data structure for application code.

The AnalogCurrentReader and AnalogTemperatureReader objects follow the same structure as the AnalogVoltageReader object, adding a further conversion step to convert the read voltage value into the expected units. This extra step is done by the respective templated implementations using equations from the sensor data sheets. Abstracting this conversion within the templated implementations allows the sensors that sample the inputs to be updated without impacting the application code.

A.3. TECHNICAL DETAILS OF THE INTERRUPT LIBRARY

The Interrupt library is used in the BSF to provide custom ISRs abstracted from the processor-specific interrupt system. Interrupt inputs are enabled during the hardware configuration stage of the software development. When enabled for an input channel, the generated hardware abstraction layer for the STMicroelectronics microcontroller creates grouped callback functions tagged with a weak attribute symbol. This symbol informs the compiler that the function implementation can be discarded during compilation if a strong symbol is found with the same signature, effectively allowing free functions to be overridden. In the context of the Binar-1 flight software, this allows strong overriding symbols to exist within the C++ environment to allow the implementation to take advantage of the language features.

The implementation of GPI interrupts in the BSF is shown in Figure A.3.1. When these types of interrupts are detected, the STMicroelectronics HAL software will call the GPI callback handler. With a strong symbol provided in the BSF for this function, the interrupt results in the execution of the custom handler code. The ISR checks to see what GPI triggered the interrupt. In the BSF, this can be either from the power system, GPS, or transceiver. Discerning the input, the ISR checks to see if the application code has registered a handler for the request in the IVT. If an entry is found, the interface pointer is polymorphically dereferenced, and the handler is executed. The UART communications-related interrupts on the BSF are implemented similarly. Interrupts

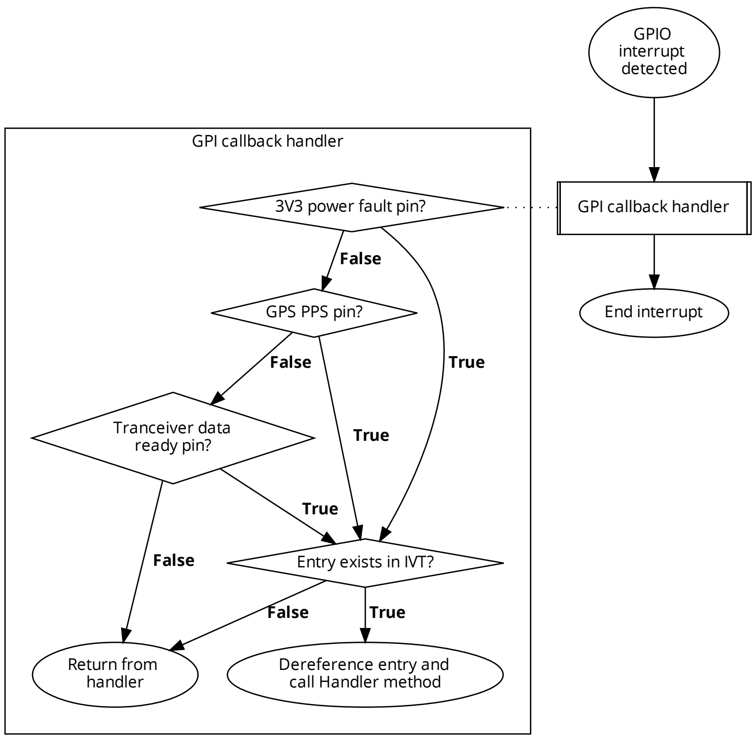


FIGURE A.3.1: The logic flow during application runtime when a GPI interrupt is detected.

are used during an asynchronous receive operation to notify the flight computer when a complete message can be actioned. The strong override in the BSF for these types of interrupts filters the interrupt request device based on the UART channel, similar to the GPI channel in Figure A.3.1.

Firing interrupts when using asynchronous communications introduces challenges when the data length being transferred is unknown. Unique delimiters are an option if present in the message being transmitted. In this case, a character match interrupt is enabled before initiating an asynchronous receive. As characters are received by the DMA peripheral, they are checked against the character to match. When found, the peripheral notifies the main processor to trigger a UART interrupt which will eventually flow through to the user-registered handler for the channel. However, a different approach is required when the messages do not contain a delimiting character or have a predefined length. The flight computer's DMA peripheral can trigger an interrupt when the data line being read falls idle. Depending on the baud rate used, this is defined as the length of the transmission time for one character. When detected, the STMicroelectronics HAL software will call the UART interrupt handler to discern which callback type to execute. Before being filtered through to the DMA idle line callback, this is captured and redirected to the receive complete interrupt handler to be grouped with the rest of the transmission interrupts.

A.4. TECHNICAL DETAILS OF THE OWNERSHIP LIBRARY

The Ownership library in the BSF provides controlled access to system peripherals to avoid common issues with concurrent programming. It does this by requiring application code to interact with a peripheral through two methods; ″Claim″ and ″Take″.

The ownership manager supports asynchronous operations by requiring peripheral access to take out a mutex. This approach ensures that only one execution thread can access a peripheral at a given time. This approach means that an application will continue execution with a failed access error that can be handled immediately if a peripheral is unavailable. The ″Claim″ and ″Take″ methods allow application code to access a peripheral. Claim allows for immediate usage of a peripheral, whereas ″Take″ allows for sustained use of a peripheral, keeping it enabled during the scope of the ″Take″. Both methods extensively use the lambda expression feature of C++. A lambda describes anonymous functors, which can be used as a parameter in a method call. Using lambdas as parameters to the ″Claim″ and ″Take″ methods allows the conventional approach of using an object to be inverted; the application code provides a function to the singleton ownership manager to use an already existing peripheral object in. Through this implementation, access to peripherals is controlled, the usage of peripheral objects can be application specific, and the memory requirements for using the peripheral are known statically at compile time.

The PeripheralHandler object is a thin wrapper encapsulating the aggregated peripheral pointers within its private scope to prevent application code from having direct access to the memory. Application code wanting to use the resource taken must do so through the ″Use″ method, taking a lambda specifying how the peripheral intends to

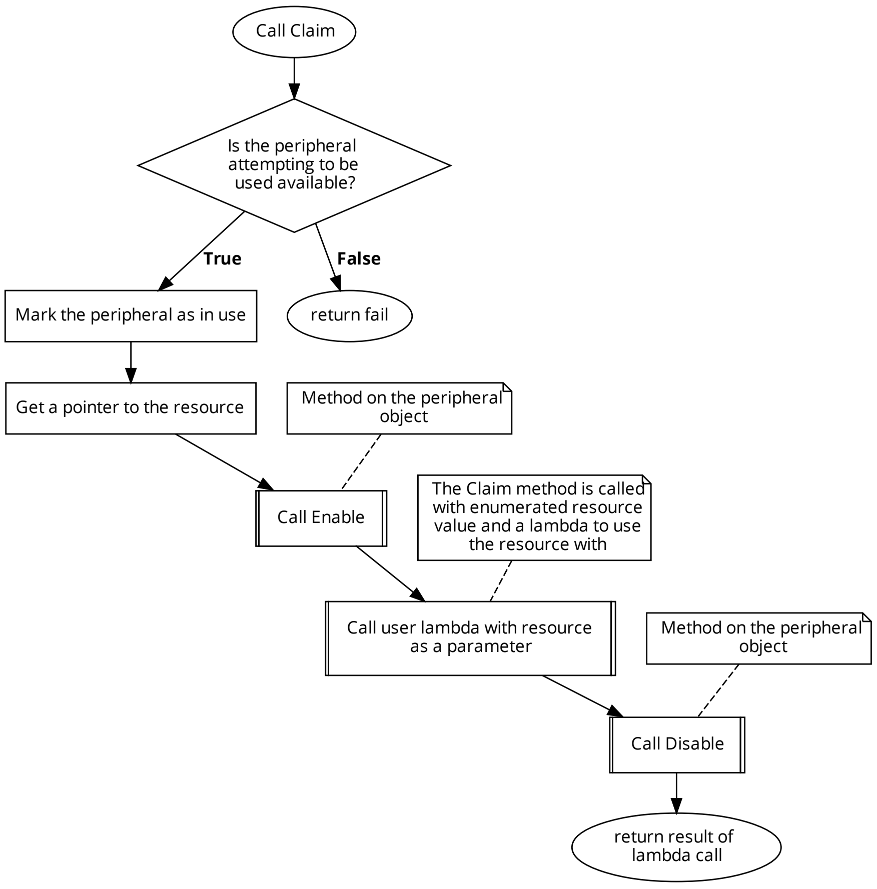

Call Claim
Is the peripheral attempting to be used available?
True
False
Mark the peripheral as in use
return fail
Get a pointer to the resource
Method on the peripheral object
Call Enable
The Claim method is called with enumerated resource value and a lambda to use the resource with
Call user lambda with resource as a parameter
Method on the peripheral object
Call Disable
return result of lambda call

FIGURE A.4.1: A high-level depiction of the logic of the "Claim" method in the Ownership library. A call to "Claim" will temporarily get ownership of a peripheral.

be used and injecting the encapsulated resource into it. This approach prevents the memory address of the peripheral from leaking into application code, a preventative measure for potential failure through a race fault.

In the case of a "Claim", shown in Figure A.4.1, the ownership manager, upon finding the peripheral unused, will call the "Enable" method for the object and then temporarily move the pointer for the peripheral into the lambda provided as a parameter to the "Claim" method. The peripheral can then be freely used within the lambda scope. After lambda execution, the peripheral manager will call the "Disable" method on the object and propagate the return to the highest-level application code. If the object is seen to be already in use, the call to "Claim" will ignore the lambda parameter and return a fail instead.

When the application code attempts to take a resource from the manager, the "Take" method follows a similar execution flow as ″Claim″, differing in what the peripheral is used for when it is found to be available. As shown in Figure A.4.2, a PeripheralHandler object is constructed with the pointer to the resource acquired by the manager. This constructed object is then moved from the ″Take″ method to the application code to prevent it from dropping out of scope when the ″Take″ method completes.

A.5. TECHNICAL DETAILS OF THE IMU LIBRARY

The BSF uses the IMU library to abstract how the flight computer interacts with the IMU peripheral. During initialisation, the constructor attempts to communicate with the peripheral to access the identification register. Suppose the device fails to return the correct register identification. In that case, the spacecraft reboots using the System Context library (Section 3.2.9) to construct the ownership manager (Section 3.2.2) with a faked implementation. If the correct identification is returned, the constructor initialises the peripheral before putting it in a low-power state. The ″Enable″ and ″Disable″ methods, used by the ownership manager when accessing the peripheral, bring the device out of and return it to the lower power state.

Performing a read on either the magnetometer or IMU die is done using the ″Get-Magnetometer″, ″GetGyroscope″, and ″GetAccelerometer″ methods. These methods encapsulate how the data returned to the processor from the IMU peripheral is converted into meaningful information. It is then returned to the application code through a simple data structure. This further supports code reuse by abstracting how the raw values are scaled. Moreover, it allows for easy module testing by providing faked implementations for these methods.

The Self Check library uses the ″GetStatus″ method to verify the integrity of the device during a non-critical system check.

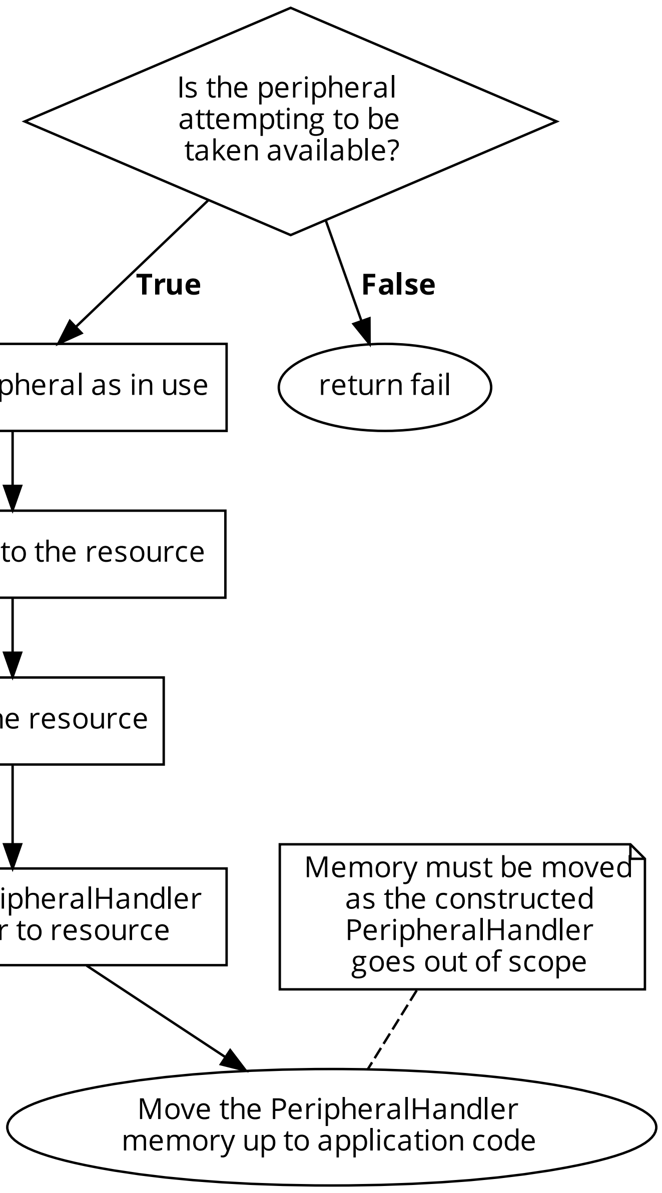

Is the peripheral attempting to be taken available?
True
False
Mark the peripheral as in use
return fail
Get a pointer to the resource
Enable the resource
Construct PeripheralHandler with pointer to resource
Memory must be moved as the constructed PeripheralHandler goes out of scope
Move the PeripheralHandler memory up to application code

FIGURE A.4.2: A high-level depiction of the logic of the "Take" method in the Ownership library. A call to "Take" will assume ownership of the peripheral until the returned PeripheralHandler is yielded or goes out of scope.

A.6. TECHNICAL DETAILS OF THE MAGNETORQUER LIBRARY

The BSF uses the Magnetorquer library to interact with the attitude control peripheral. The magnetorquer attitude control device consists of three independent axes, xx_, yy_, and zz_, derived from the MagnetorquerDriverInterface object. The hardware implementation of the MagnetorquerDriver object in the Binar-1 flight software templates on the MagnetorquerAxis_t enumerated class.

The library's MagnetorquerDriver object provides methods to enable/disable the axis drivers and pass requested dipole moments from the application code to the lower-level drivers. The "SetDipole" method sets the magnetic torque produced by the magnetorquers by adjusting the average voltage applied to each axis. This is achieved by using a PWM signal (Section A.1) set using the "SetPWM" method (Figure A.6.1) on the MagnetorquerDriver object. As the voltage applied to the magnetorquers is scaled from the battery voltage, this method starts by reading the battery voltage to compensate for changes due to battery capacity in the duty cycle calculation. The polarity of the signal is set depending on the sign attributed to the requested dipole moment; if the polarity is high, the H-bridge phase pin is set high, and the produced waveform will be applied forwards over the coil. Inversely, with a low phase, the produced waveform will be applied backwards over the coil. Finally, the status is read from the device using the "GetStatus" method to return to the application code. The self check library also uses this method to determine if either the X or Y-axis magnetorquer driver has failed and needs to be replaced with a redundant driver.

A.7. TECHNICAL DETAILS OF THE STORAGE LIBRARY

The Storage library in the BSF is responsible for abstracting how the application code interacts with the storage peripheral. The public-facing Storage object aggregates a private StorageImpl object derived from the same interface as the public-facing object. Calls to the public methods delegate through to the private implementation. The StorageImpl object aggregates the private member of type w25m02gw_t, memory_ctx_,

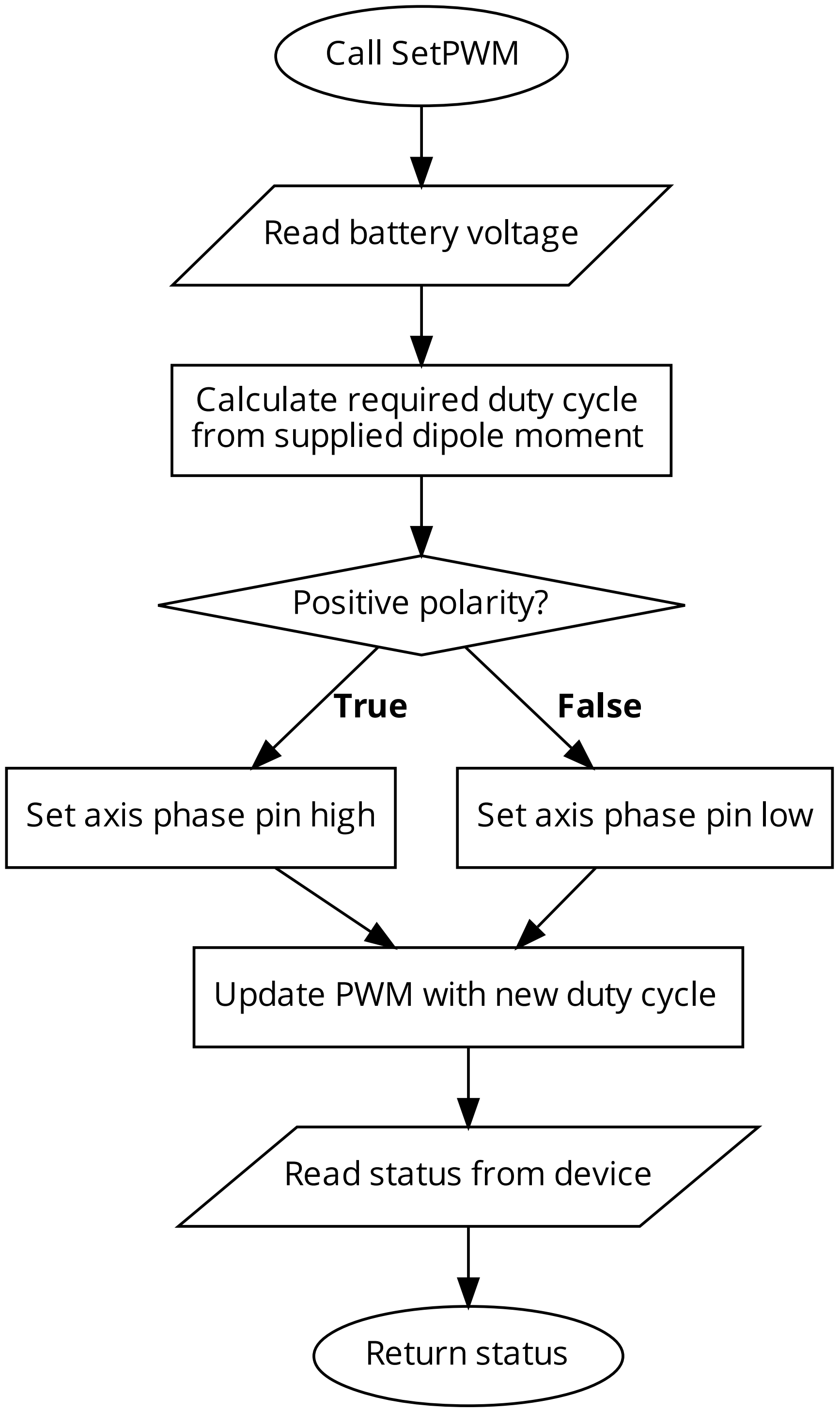


FIGURE A.6.1: A high-level overview of the "SetPWM" method in the magnetorquer library. The torque produced by the attitude control device is controlled by changing the average voltage felt on the axis.

83	0	1	0	1	0	0	1	1
82	0	1	0	1	0	0	1	0
71	0	1	0	0	0	1	1	1
66	0	1	0	0	0	0	1	0
4	0	0	0	0	0	1	0	0

FIGURE A.8.1: Demonstration of the exclusive OR operation. The binary representation of each integer on the left is compared for an isolated high bit. In this example, only one-bit location in the bytes has an exclusive high bit, and hence the resulting calculated checksum would have the decimal value 4.

a custom low-level driver created for interacting with the hardware on the satellite. The low-level driver provides function pointers to be implemented in application code to communicate with the peripheral. In the context of the Binar hardware, communications are implemented with the device using SPI. When constructed during the peripheral initialisation stage of the flight software, the StorageImpl mounts a file system on the peripheral, storing the filesystem information in its private member variables. The directories used in the file system are for voltage, current, and temperature logs. The "Write", "Read", and "Delete" methods invoke the file system to manipulate the data as requested.

A.8. TECHNICAL DETAILS OF THE GPS LIBRARY

The application code uses the GPS library to retrieve location and timing information from the GPS peripheral. The GPS transmits a collection of NMEA strings containing the information. These ASCII strings contain a start delimiter, $, a stop delimiter, *, data and a checksum. The checksum for NMEA strings is calculated by performing an exclusive OR (XOR) operation on each character. For a bit in the checksum byte to be high, only one other bit in the same place across all characters comprising the string can also be high (Figure A.8.1).

The implementation in the GPSBase object for calculating the checksum is shown in Figure A.8.4. The method first iterates over the starting delimiter character, as this is not included in checksum calculations. Each character in the string is iterated over and examined, performing an XOR operation with the running checksum variable and the character. The checksum at the end of the data string is then compared with the calculated checksum. The result is propagated to the calling code.

The GPS object inherits the GPSBase to add interaction with the hardware. The constructor for the GPS object allows the hardware to be configured through the aggregated custom venus838_t driver. This driver implements the required functionality from the device datasheet, giving the constructor scope access to changing the device baud rate, transmission frequency, and enabled NMEA strings. Of the six NMEA strings transmitted from the GPS on Binar-1, only two are required; Global Positioning System Fix Data (GPGGA) and Recommended Minimum Specific GNSS Data (GPRMC). GPGGA provides spacecraft altitude, while GPRMC provides time, latitude, longitude, velocity and coordinated universal time (UTC) date. When strings are received from the GPS peripheral, they are given to the "Parse" method inherited from the GPSBase to extract and convert the required information. The "Parse" method is summarised in Figure A.8.2.

If called with a char pointer to a valid memory location, the "Parse" method will attempt to break the received block of strings into multiple individual strings by splitting the string block on the starting $ delimiter. The resulting substrings are further split on the comma delimiter to isolate each string element. Whilst the pointer to the isolated string element, or token, has a valid memory address, the "Parse" method will check to see if data can be extracted. It first checks if the token under examination is the name of the GPGGA string. If found, the checksum of the isolated GPGGA string is calculated to verify that the data contained within is correct. If the checksum matches, the method attempts to parse the altitude from the string. While incrementing through the delimited substrings, the "Parse" method will also check to see if the token under examination contains the name of the GPRMC string. The information extraction process from the GPRMC string is similar to the GPGGA string. As the GPRMC string comes after the GPGGA string in the NMEA outputs of the GPS, a check for the success of both extractions follows parsing the GPRMC string. If string parsing is successful, the parsed values are assembled into a GPSReadings struct and are returned to the application code. All other execution pathways in the method will eventually return a fail object.

When the hardware GPS peripheral receives a fix on a GPS satellite, the device will

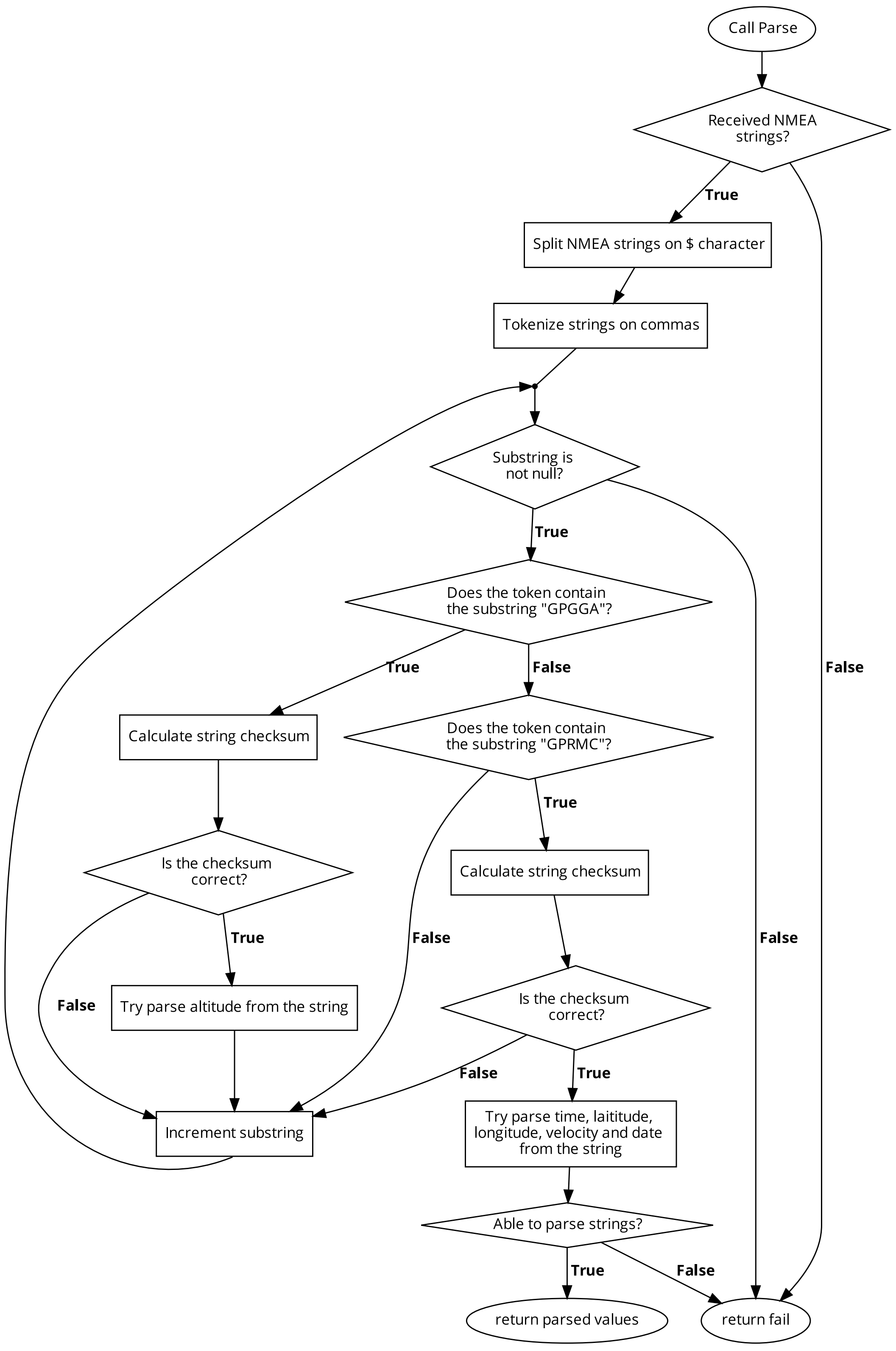

Call Parse
Received NMEA strings?
True
False
Split NMEA strings on $ character
Tokenize strings on commas
Substring is not null?
True
False
Does the token contain the substring "GPGGA"?
True
False
Calculate string checksum
Does the token contain the substring "GPRMC"?
True
False
Is the checksum correct?
True
False
Try parse altitude from the string
Calculate string checksum
Is the checksum correct?
True
False
Increment substring
Try parse time, laititude, longitude, velocity and date from the string
Able to parse strings?
True
False
return parsed values
return fail

FIGURE A.8.2: The logic for parsing latitude, longitude, altitude, velocity, time and date from the GPS NMEA strings.

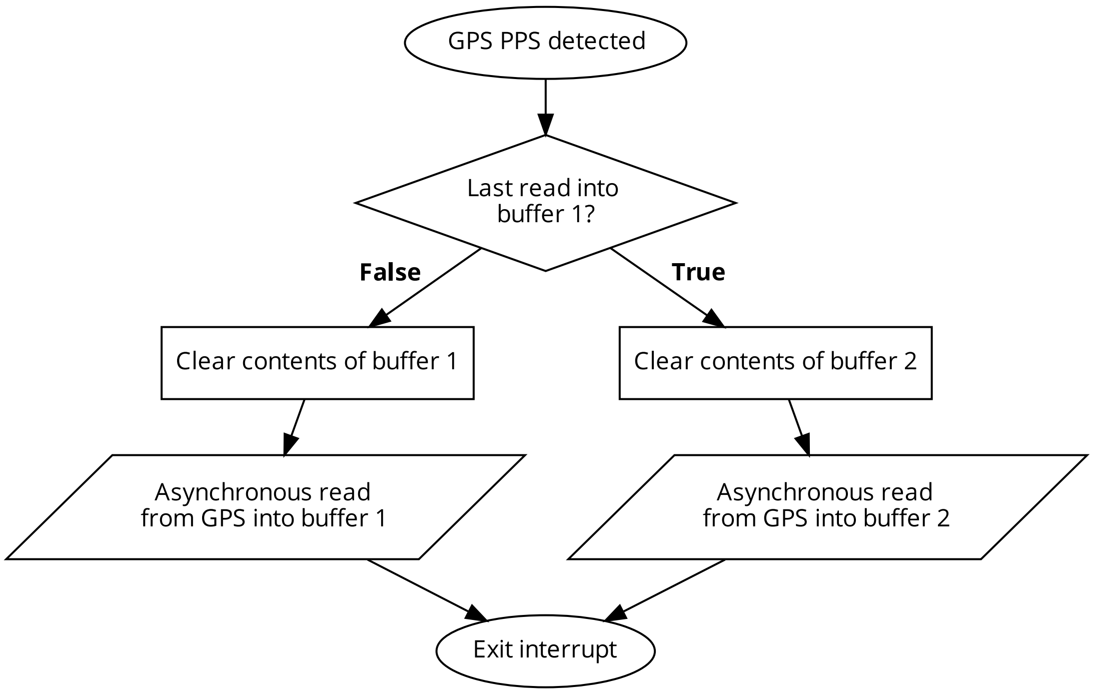

GPS PPS detected
Last read into buffer 1?
False
True
Clear contents of buffer 1
Clear contents of buffer 2
Asynchronous read from GPS into buffer 1
Asynchronous read from GPS into buffer 2
Exit interrupt

FIGURE A.8.3: Asynchronously reading into a double buffer implementation triggered by an external GPI interrupt from the GPS.

emit a 1 Hz pulse per second (PPS) synchronised with the top of the current second. By inheriting the ExternalInterruptInterface object from the Interrupt library (Section 3.2.1), the flight computer is instructed to execute the handler of the PPS monitoring object when the GPI interrupt is received. This interrupt signifies that data is about to be emitted from the device. When the ISR is triggered, the logic shown in Figure A.8.3 is executed.

As the GPS outputs up to 800 bytes of data every second, attempting to parse received messages without missing new messages introduces a challenge. If information is not parsed quickly enough, it may be overwritten by an incoming message. A solution to this was implemented by using a double buffer approach; the strings from the previous second can be stored and parsed in one buffer whilst receiving the new strings in the other. The ISR scope has access to this through the aggregated DmaReceiving object. In the ISR, the interrupt checks which of the two buffers was the last read into to not overwrite the data being parsed. The buffer not currently being operated on is cleared, and an asynchronous transaction is started from the peripheral into the buffer memory using the aggregated GPS channel instantiation of the templated Serial library.

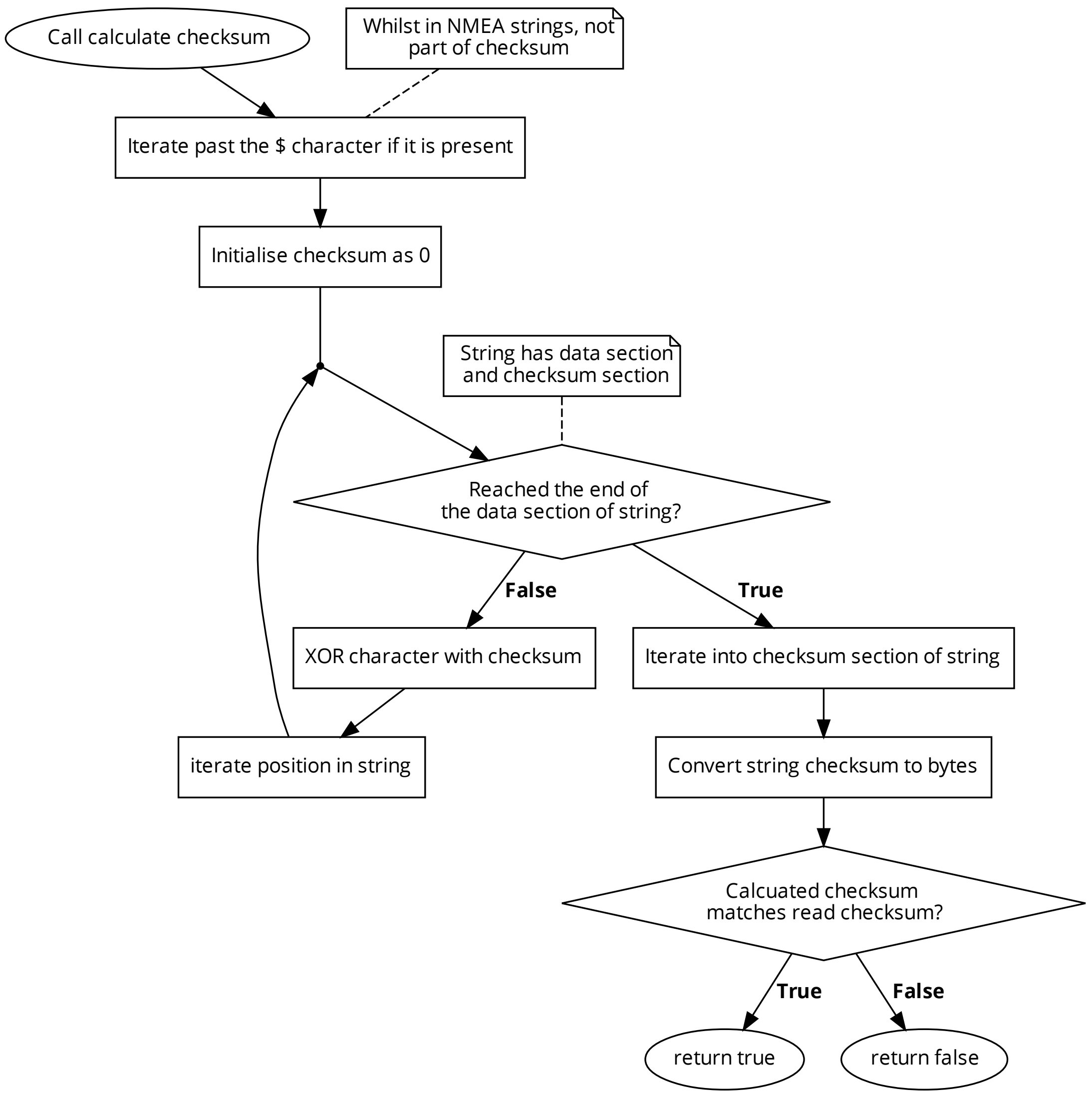

Call calculate checksum
Whilst in NMEA strings, not part of checksum
Iterate past the $ character if it is present
Initialise checksum as 0
String has data section and checksum section
Reached the end of the data section of string?
False
True
XOR character with checksum
Iterate into checksum section of string
iterate position in string
Convert string checksum to bytes
Calcuated checksum matches read checksum?
True
False
return true
return false

FIGURE A.8.4: The logic for calculating the checksum of a received NMEA string.

The asynchronous transactions are set to cancel when the receiving line falls idle. After initiating the asynchronous transaction, the internal boolean, buffer_1_2_, is inverted to keep track of which buffer to use for the next transaction.

Another interrupt, GPSReceiveComplete, is used to signify the first time an asynchronous transaction has been completed. This interrupt allows the implemented "ReadGPS" method (Figure A.8.5) in the GPS object to return fail unless the hardware has started transmitting. The handler for the ISR flips the internal private boolean variable gps_transmission_status_ to high, accessible through the public "GetGPSTransmissionStatus" method. When the application code attempts to read the GPS, it first checks to see if the GPS has started transmission by querying this method. Once strings have started to be emitted, the GPS module will check what the active buffer is to know which data will not be mutated during parsing. This data is then copied to the appropriate private internal buffer within the GPS object (buffer_one_copy_ or buffer_two_copy_) before attempting to parse the strings using the "Parse" method inherited from the GPSBase to return to the application code.

A.9. TECHNICAL DETAILS OF THE I2C TO UART LIBRARY

The I2C to UART Library translates the serial communications between the flight computer and the transceiver. After assigning means to communicate with the IC, a reset command is issued using the "Reset" method. The device is then initialised in first-in-first-out mode by calling the "FIFOEnable" method. The communications speed is configured by calling the "SetBaudRate" method with the baud rate provided to the constructor. A write instruction informs the device to output a GPO interrupt when converted data is ready. Finally, the line control mode is set using the "SetLineCtrl" method, configuring communications to use 8-bit mode, with no parity bits and a single stop bit.

Using the Interrupt library (Section 3.2.1), an object was created to handle when the data-ready GPO interrupt was received from the device. When triggered, this handler would claim the I2CToUART object from the ownership manager and call the

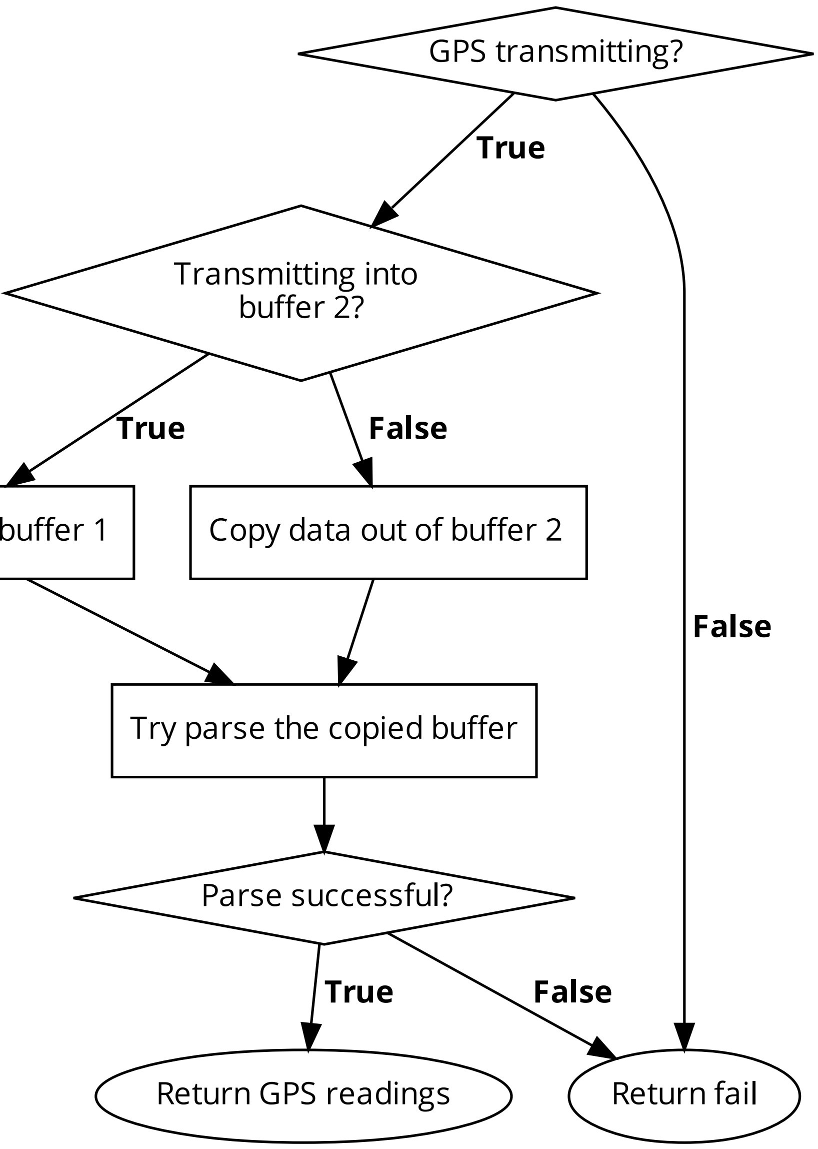

GPS transmitting?
True
False
Transmitting into buffer 2?
True
False
Copy data out of buffer 1
Copy data out of buffer 2
Try parse the copied buffer
Parse successful?
True
False
Return GPS readings
Return fail

FIGURE A.8.5: Copying out non-mutating buffers for parsing by the public-facing GPS object.

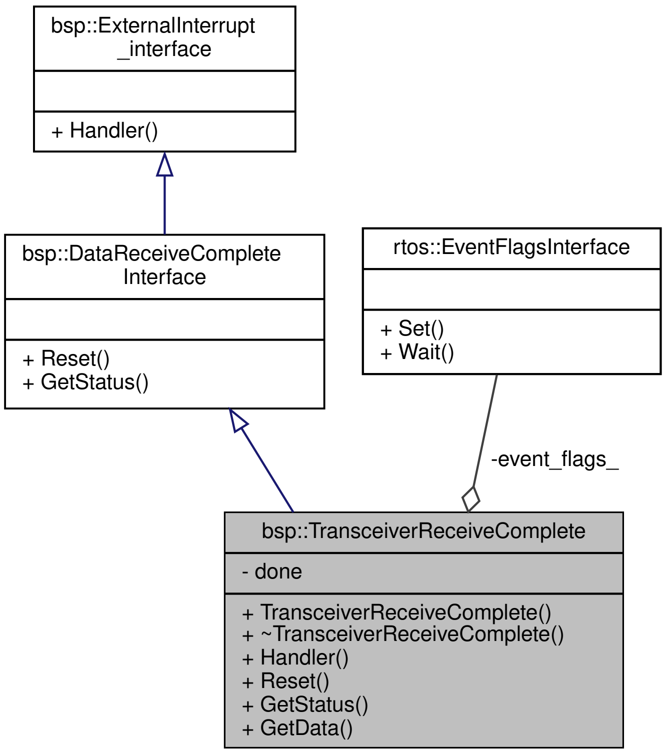

bsp::ExternalInterrupt_interface
+ Handler()
bsp::DataReceiveCompleteInterface
+ Reset()
+ GetStatus()
rtos::EventFlagsInterface
+ Set()
+ Wait()
-event_flags_
bsp::TransceiverReceiveComplete
- done
+ TransceiverReceiveComplete()
+ ~TransceiverReceiveComplete()
+ Handler()
+ Reset()
+ GetStatus()
+ GetData()

FIGURE A.9.1: A UML representation of the transceiver receive complete interrupt object on Binar-1.

"Read" method to append the available bytes into a static file-scoped buffer within the interrupt library. Another interrupt, TransceiverReceiveComplete (Figure A.9.1), monitors the transaction and is triggered upon completion.

The TransceiverReceiveComplete interrupt is constructed with an event flag. Therefore, it is used in the ground station communications RTOS thread (Section 3.2.8) to yield the thread while waiting for the event flag to be set. As messages transmitted from the ground station are COBS encoded (Section 3.6.2) to be delimited on a zero byte, the handler loops over the newly received sequence of bytes, searching for the zero. When the zero byte is found, the private done_ boolean (Accessible through the "GetStatus" method) and the aggregated event_flags_ member are set.

Finally, the communications thread copies the transferred data from the file-scoped buffer into the thread via the "GetData" method to be processed by the remote procedure call (RPC) server. It resets the interrupt using the "Reset" method, clearing the file-scoped buffer and inverting the done_ boolean.

A.10. Technical Details of the RTOS Library

The BSF uses the RTOS library to emulate multi-threading capabilities on the single execution core available to the flight logic. Constructing a thread requires two template parameters to be passed through to the constructor for the ThreadBase object; the thread stack size in bytes and the thread priority. The thread stack size is used during compilation to allocate the ThreadBase private member, thread_buffer_, which serves as the memory stack for the thread. This member allows the memory resource required for the thread to be known statically at compile time, as opposed to the standard practice of the FreeRTOS kernel requesting heap memory for threads. The thread priority determines which thread is to be executed when multiple threads are in the ready state. The initialised thread buffer and priority are passed to the constructor of the privately aggregated ThreadImpl object for use by the underlying FreeRTOS kernel. These are then stored in the thread_attributes_ private member variable. The thread_control_block_ private member is used to store the execution context of the thread when runtime switches to another thread. This ensures the thread can be resumed where it was yielded. After initialising these variables, the new thread is registered with the kernel, providing the thread's "Run" method as the entry point.

When some threads have been created, the constructor for the public-facing RTOS object calls the "Initialise" method, which reads the kernel state using the protected "GetKernelState" method. If the thread returns an inactive state, the protected method KernelInitialize is called. These method calls delegate to the methods of the same name within the aggregated implementation object. Finally, the RTOS kernel is started using the "StartKernel" method, which begins thread execution.

In the FreeRTOS implementation in the BSF, a thread can be in one of four states: running, blocked, suspended and ready. Figure A.10.1 shows the relationship between the states. When a thread is first created in application code by inheriting the Thread-Base and instantiating the derived thread class, it is put into the ready state by the RTOS. When the RTOS kernel is started by executing the "StartKernel" method on the RTOS object, the thread constructed with the highest priority will be moved into the running state. Using cooperative mode scheduling, the kernel will not immediately

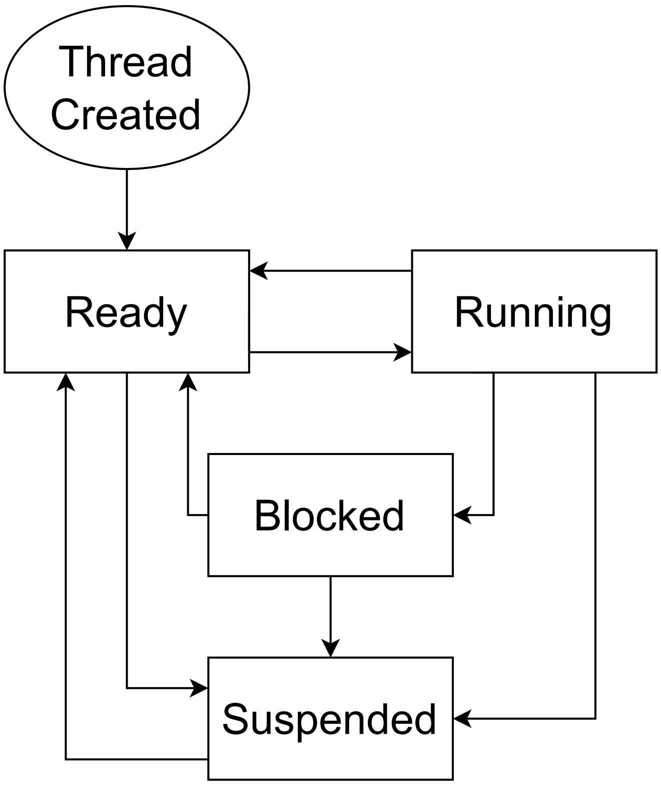


FIGURE A.10.1: Adapted from [178].

switch tasks if a higher priority task than the one currently executing becomes available. Instead, it will wait until the current task yields its kernel time and moves into the blocked state. In the BSF, threads are moved into the blocked state during calls to delay execution and while waiting for event flags to be triggered.

Threads are synchronised in the RTOS library through the EventFlags object. The private member variable next_flag_ is a static member of the EventFlagsImpl class. This attribute means it is accessible by all instances of the EventFlags object. It is used during the object's constructor to assign the next available unique identifier into the event flag, stored as a 32-bit unsigned integer in flag_. The private member flag_id_ is used internally by the FreeRTOS library to store a pointer to the created event flag. In the flight software, an event flag is used in the ground communications thread to signal when the spacecraft has received a message to process. The communications thread is given the highest priority among the other threads. Hence, when the RTOS kernel is started, the communications thread runs first. It sets up an asynchronous serial DMA transaction, providing a callback interrupt extended from the Interrupt library

(Section 3.2.1) to set the event flag using the ”Set” method on the EventFlags object. The communications thread can then yield processing time using the ”Wait” method, moving into the blocked state until the event flag is set. This way, the spacecraft can always listen for incoming transmissions while switching between the task scheduler and beacon threads.

A.11. TECHNICAL DETAILS OF THE SYSTEM CONTEXT LIBRARY

The system context library abstracts how the application code performs processor-specific functionality. The ”Reboot” method performs a soft restart of the satellite without cutting power to the flight computer. ”ConfigureNonVolatileMemory” abstracts read and write access to internal non-volatile memory. On Binar-1, registers inside the real-time clock (RTC) were used for storing information regarding:

- IMU, GPS, transceiver, and independent magnetorquer axis usage
- State of system voltages
- State of system currents
- State of system temperatures
- Redundant power line usage
- Failed firmware update
- Corrupted application code
- Target flash bank
- Boot count

The peripheral device usage registers are set and cleared when the ownership manager enables and disables the corresponding peripheral. The system voltage, current, and temperature states are updated accordingly whenever the channel’s analog hardware abstraction object is read. Each channel is compared against hardcoded ranges set from systems testing, notifying if the reading has crossed a lower warning, upper warning, lower fail or upper fail.

The satellite has a redundant regulator from battery voltage to 3.3 V to power the flight computer. The redundant power usage register contains information on which regulator is being used. Both regulators are enabled at system startup, consuming unnecessary power. If the register is reset, the flight computer will set the register and attempt to disable the redundant regulator. If the primary regulator had failed and the flight computer was being powered from the redundant regulator, this would cause the flight computer to power cycle. Upon re-entering this section of the boot process, the flight computer will identify the set flag in the register, discerning that the redundant regulator is required and will progress with the remainder of the boot process.

Performing a firmware update on Binar-1 wrote directly from telecommand into flash memory. Whilst being the simplest approach, if the spacecraft failed to receive all firmware packets, the application space being written to would not be safe to boot. The failed firmware update flag would signify this having occurred, used during the boot process to determine if a Safe Mode entry was required.

Before leaving the bootloader, the flight software checks if the application space it is about to jump to has been corrupted by checking the corrupted application flag. If reset, the bootloader sets this register. After jumping to the application space, the first instruction is to reset this register. If, however, the flash memory the application code resides in became corrupt, this instruction would not be executed. Stuck at an address with no progression, the watchdog timer will eventually lapse, and the flight computer will reboot into its bootloader. The corrupted application code register would still be set, and the spacecraft would boot into Safe Mode instead.

The 2048 KiB flash memory space on the flight computer used on Binar-1 included two programmable application spaces. Dual booting was supported on the satellite by setting the target flash bank register before issuing a reboot command. The bootloader checks this register to identify which application space to boot into.

Lastly, the boot count register increments each time a reboot is performed, keeping track of the total number of system resets.

A.12. TECHNICAL DETAILS OF THE SYSTEM LOGGER LIBRARY

The System Logger library abstracts how the flight logic writes and retrieves systems logs. The SystemLogger object aggregates various subsystem logs to achieve this task. The ″Write″ method implementation for each protected member is similar, differing only in how the information to log is collected and the name of the file to log. A high-level flow chart of the ″Write″ method logic for these members is given in Figure A.12.1.

The ″Write″ method on the system log implementations first constructs the appropriate analog object to collect data from using the hardware-abstracted analog object discussed in Section 3.2.1. If the read on the ADC object is successful, the sampled data is assembled into a data structure with the current timestamp to be serialised for storage. The serialised byte stream is then COBS encoded to provide a method of delimiting stored messages with minimal overhead. This encoded, serialised byte stream is then passed to the helper function WriteLogToDisk, depicted in Figure A.12.2.

WriteLogToDisk first attempts to access the storage peripheral through the ownership manager, detailed in Section 3.2.2. With the storage device claimed temporarily, a file name is assembled using the analog channel type with the current hour through the day. For example, if the voltage logger object called the WriteLogToDisk function at 05:00 UTC, the filename would take the form */voltage/5.log*.

As only 24 hours' logs were stored on Binar-1, the function checked if the previous day's log needed to be deleted. It does this by checking the last edited file in the OBS metadata. If the last edited file is not the same as the current time, then the existing file is deleted to be replaced. Finally, the file metadata is updated to note the time the file was last edited, and the byte stream is appended to the end of the file.

An accumulated log file over 24 hours is too large to downlink in a single telecommand from the spacecraft. Hence, the ″PrepareLog″ method (Figure A.12.3) on the protected members is used to break up the serialised logs for transmission.

When the ″PrepareLog″ method is called on each SystemLoggerInterface implemen-

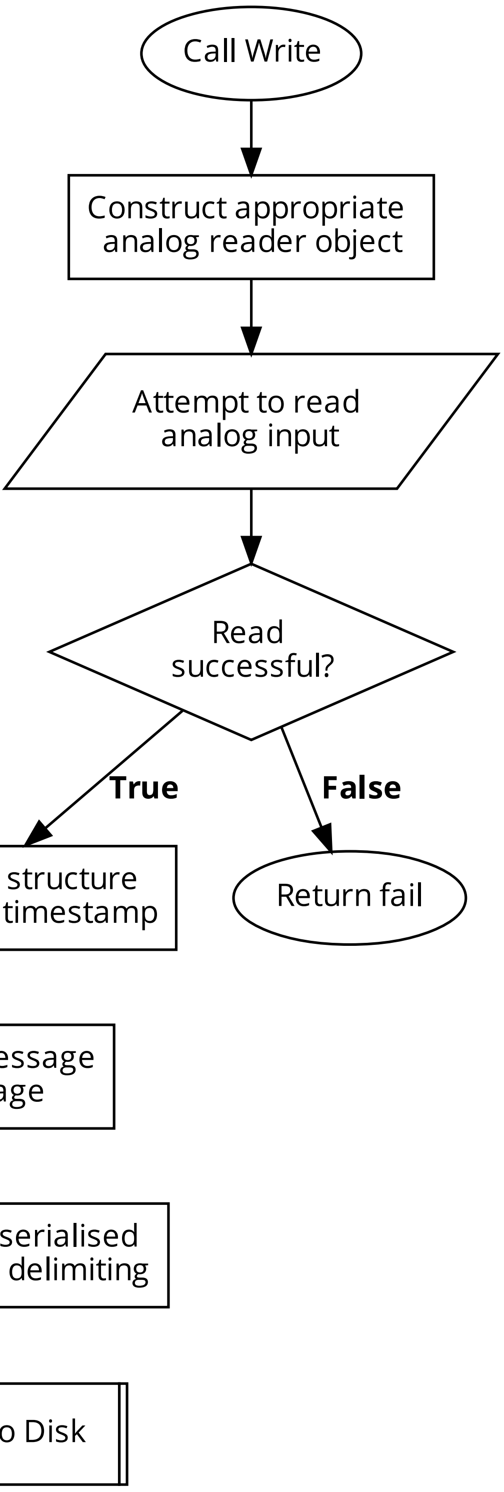

Call Write
Construct appropriate analog reader object
Attempt to read analog input
Read successful?
True
False
Return fail
Assemble data structure to serialise with timestamp
Serialise message for storage
COBS encode serialised byte stream for delimiting
Write Log to Disk

FIGURE A.12.1: A high-level depiction of the "Write" method on an object derived from the SystemLoggerInterface object. Data to log is given a timestamp and serialised using COBS encoding to delimit previously logged messages with a 0 character.

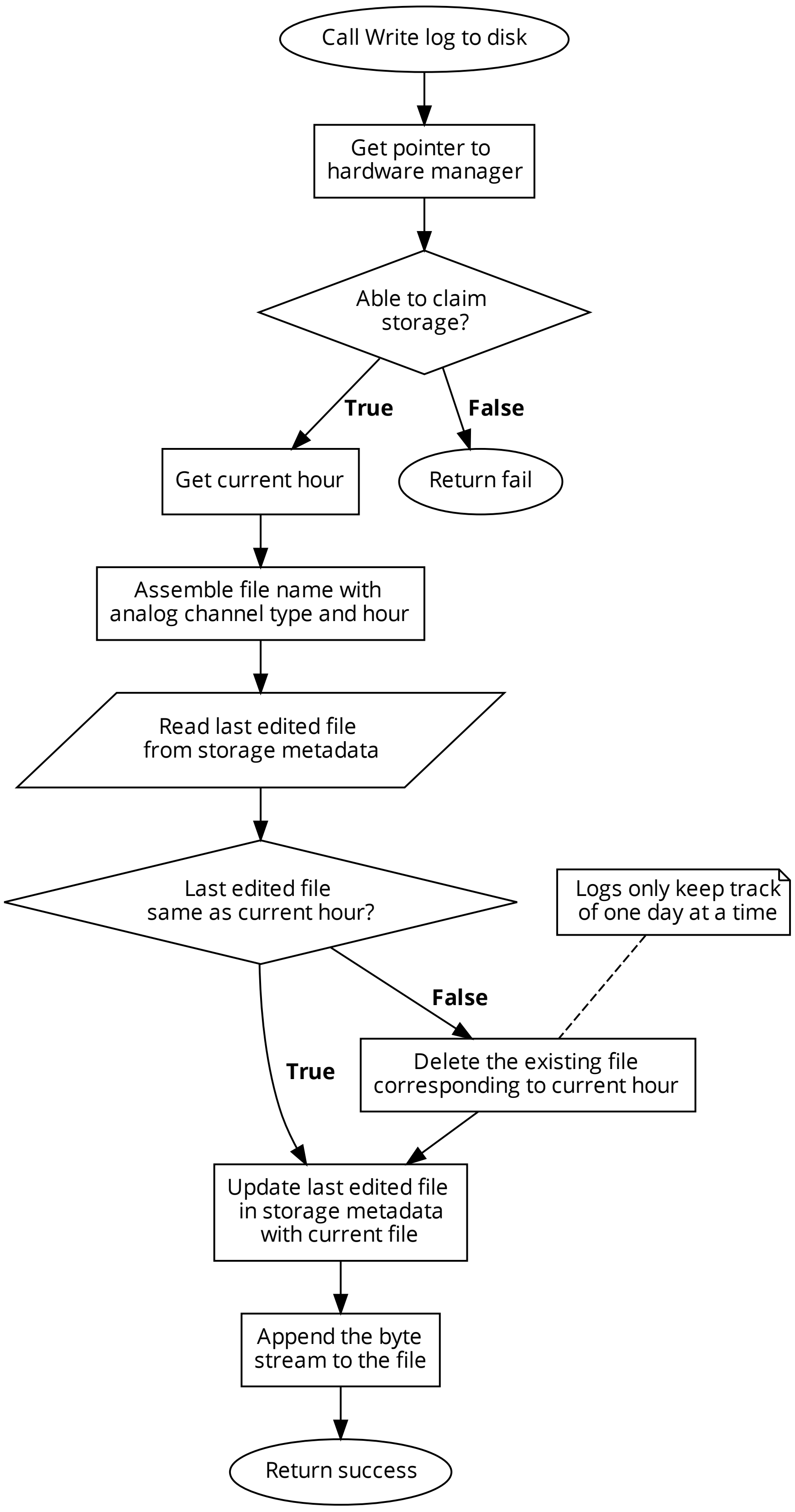

Call Write log to disk
Get pointer to hardware manager
Able to claim storage?
True
False
Get current hour
Return fail
Assemble file name with analog channel type and hour
Read last edited file from storage metadata
Last edited file same as current hour?
Logs only keep track of one day at a time
False
True
Delete the existing file corresponding to current hour
Update last edited file in storage metadata with current file
Append the byte stream to the file
Return success

FIGURE A.12.2: A high-level overview of the WriteLogToDisk function in the system logger library. This function uses the Ownership library (Section 3.2.2) to access the Storage peripheral (Section 3.2.5) to write the serialised logs to storage.

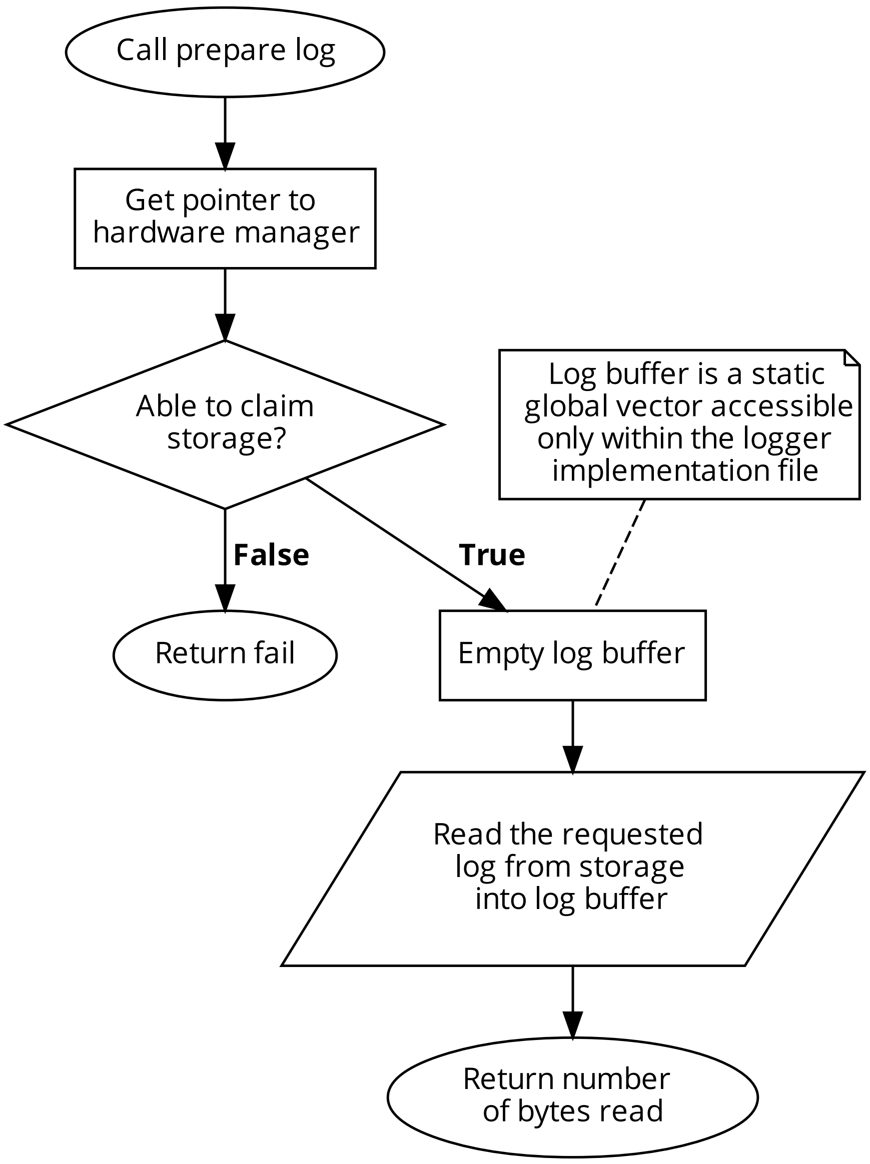

Call prepare log
Get pointer to hardware manager
Able to claim storage?
False
True
Log buffer is a static global vector accessible only within the logger implementation file
Return fail
Empty log buffer
Read the requested log from storage into log buffer
Return number of bytes read

FIGURE A.12.3: A high-level overview of the "PrepareLog" method on an object derived from the SystemLoggerInterface object. This method moves the requested stored log from the OBS to RAM in preparation for downlinking.

tation, the storage device is temporarily claimed from the ownership manager. The file-scoped static vector in the implementation file is cleared to fill with new logs to download. This vector is large enough to contain the entire log file in RAM to only need to read from the storage device once per log request. Using the analog channel's type and the requested hour, the log is read from the storage device into the file-scoped buffer, and the number of bytes in the log is returned to be transmitted to the ground.

The final stage to retrieve logs is to call the ″GetLogChunk″ method on the Logger object. Using the index parameter with a fixed log segment size of 1024 bytes, the function returns a sub-span of the static file-scoped vector to be returned to the ground.

A.13. TECHNICAL DETAILS OF THE MANIPULATOR LIBRARY

The Manipulator library abstracts how the application code interacts with the manipulator payload. The emulator implementation logs the angle sent to the spacecraft for later retrieval by a ground operator. Retrieving the log is done first by calling the ″PrepareLog″ method, which copies the log from the OBS to a static file-scoped logger buffer in the implementation file. Subsequent calls to GetLogChunk will return a span over the requested segment of up to 1024 bytes to be downlinked.

A.14. TECHNICAL DETAILS OF THE STATIC LIST LIBRARY

The Static List library is used in the BSF to provide a sortable data structure to hold tasks to be executed at certain times without requiring the use of dynamic memory.

The static list is a linked list that chains objects of type MemoryPoolNode together. The MemoryPoolNode base class aggregates a private member, used_, that keeps track of the usage state of the node. It is used in the private MemoryPoolNode array object in StaticList to discern if the node can store data through the publicly accessible ″GetUsed″ and ″SetUsed″ methods. ″ClearUsed″ marks a node as free to use. The MemoryPoolNode object derived from the base adds the datatype and maximum

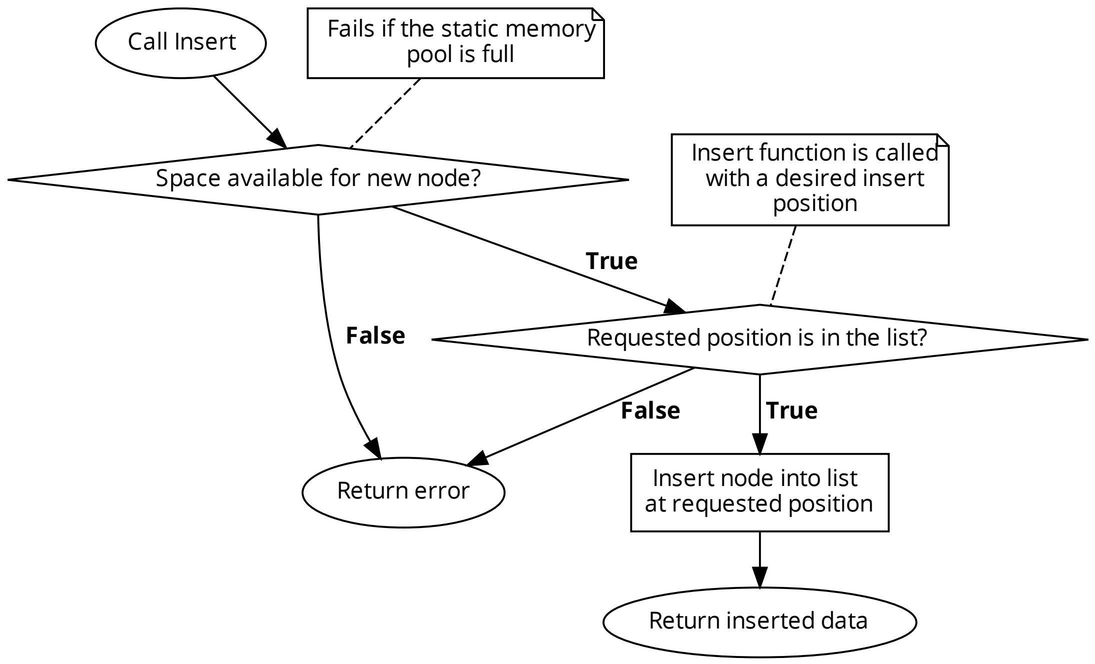

Call Insert
Fails if the static memory pool is full
Space available for new node?
Insert function is called with a desired insert position
True
False
Requested position is in the list?
False
True
Return error
Insert node into list at requested position
Return inserted data

FIGURE A.14.1: A flowchart depicting the logic of the "Insert" method in the StaticList class.

length template parameters. It also adds private "Next" and "Previous" methods, which are used internally by the object to return pointers to the next and previous elements in a linked list comprising multiple nodes.

Application code attempting to insert data into the StaticList object does so through the "Insert" method, detailed in Figure A.14.1. Application code calls the "Insert" method with a pointer to where the new data is to be inserted and the data to insert.

The "Insert" method starts by attempting to allocate space for the node in the memory pool array, iterating through the pool and using the "GetUsed" method in the MemoryPoolNode object. If the memory pool is full, an error is returned to the application code, signifying that a node must first be cleared. With space available for insertion, the location to insert the new node is searched for by calling the "Next" method to update the private aggregated MemoryPoolNode pointer member and comparing the memory address with the provided node pointer parameter. Failing to find the location to insert at, the method will propagate an error to the application code. The new node is inserted if the location is found, and the data is returned to the application code. The "PushFront" and "PushBack" methods are used to insert a node at the head and tail of the linked list if middle insertion is not required. The ″Erase″ method is used to erase an existing node. The method will check whether the requested node exists in the list using the ″FindInPool″ method, which iterates over the internal private memory pool array. If the node to erase is found, the surrounding nodes connecting it to the list will be updated to skip over the erasing node, pointing to each other instead. The node will then be marked as free to use again in the memory pool by calling the ″ClearUsed″ method on the MemoryPoolNode object. The logic of the ″Erase″ method is shown in Figure A.14.2. In the implementation of task scheduling in flight software, this method is used when the ground station wishes to cancel a previously scheduled task. When executing a task, the application code calls the ″Pop″ method, depicted in Figure A.14.3.

The ″Pop″ method removes the node from the head of the list and returns it to the calling code. It does this by getting a pointer to the first node in the list through the ″Begin″ method, using the aggregated private LinkedList object in the StaticList. If the list is not empty, the node to be popped is queried to see if anything connects to it, signifying that it is part of a larger list. This check is required because the private aggregated LinkedList object needs to track the head of the list. If the node to be popped is removed from the list incorrectly, it could leave a dangling chain of nodes with no entry. Hence, the head of the list is updated to point to the second node if one exists, else it is updated to point to nothing. Finally, the node being popped is marked as free to use in the memory pool, and the data is returned to the calling code. In the Binar flight software case, the nodes contain a task object executed when returned through ″Pop″.

The private ″Allocate″ method (Figure A.14.4), used internally by the StaticList object, is used to query the statically allocated memory pool for empty nodes for data insertion. It achieves this by iterating over the memory pool array, examining each node using the private aggregated MemoryPoolNode pointer. It calls the ″GetUsed″ method to check if the node is in use. If an empty node is found, the method will copy the user-provided data into it and mark it as in use, returning a pointer to its location in the memory pool to be inserted into the list.

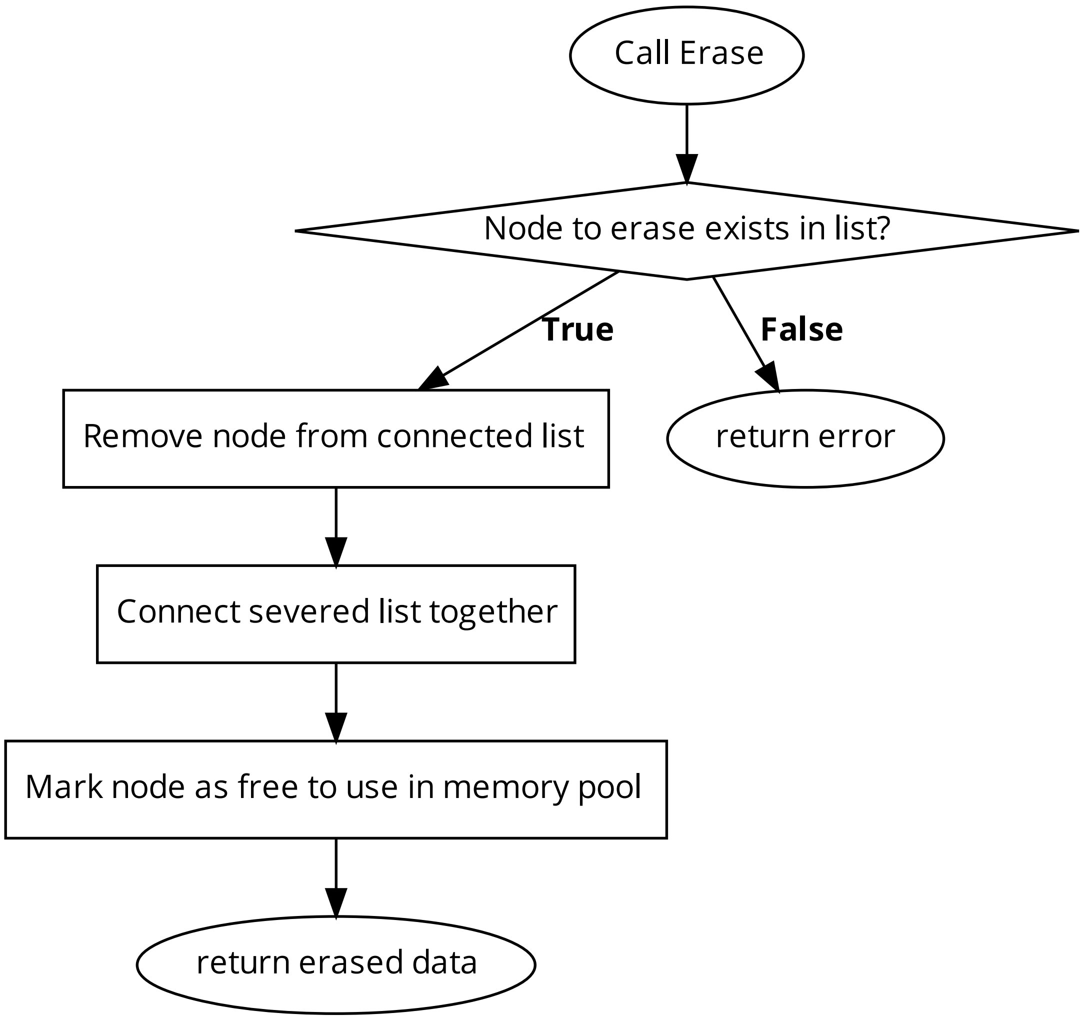

Call Erase
Node to erase exists in list?
True
False
Remove node from connected list
return error
Connect severed list together
Mark node as free to use in memory pool
return erased data

FIGURE A.14.2: A flowchart depicting the logic of the "Erase" method in the StaticList class.

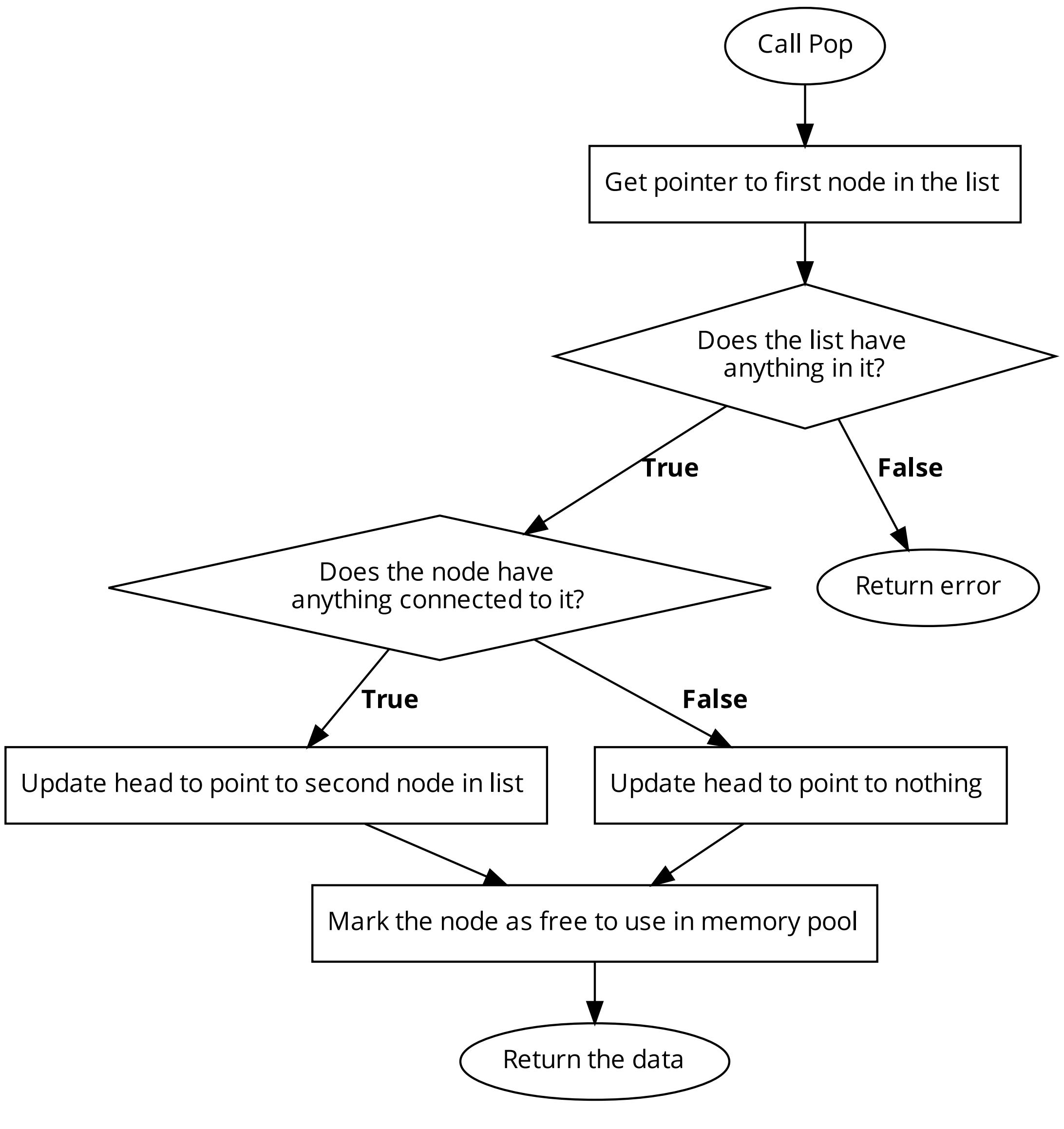

Call Pop
Get pointer to first node in the list
Does the list have anything in it?
True
False
Does the node have anything connected to it?
Return error
True
False
Update head to point to second node in list
Update head to point to nothing
Mark the node as free to use in memory pool
Return the data

FIGURE A.14.3: A flowchart depicting the logic of the "Pop" method in the StaticList class.

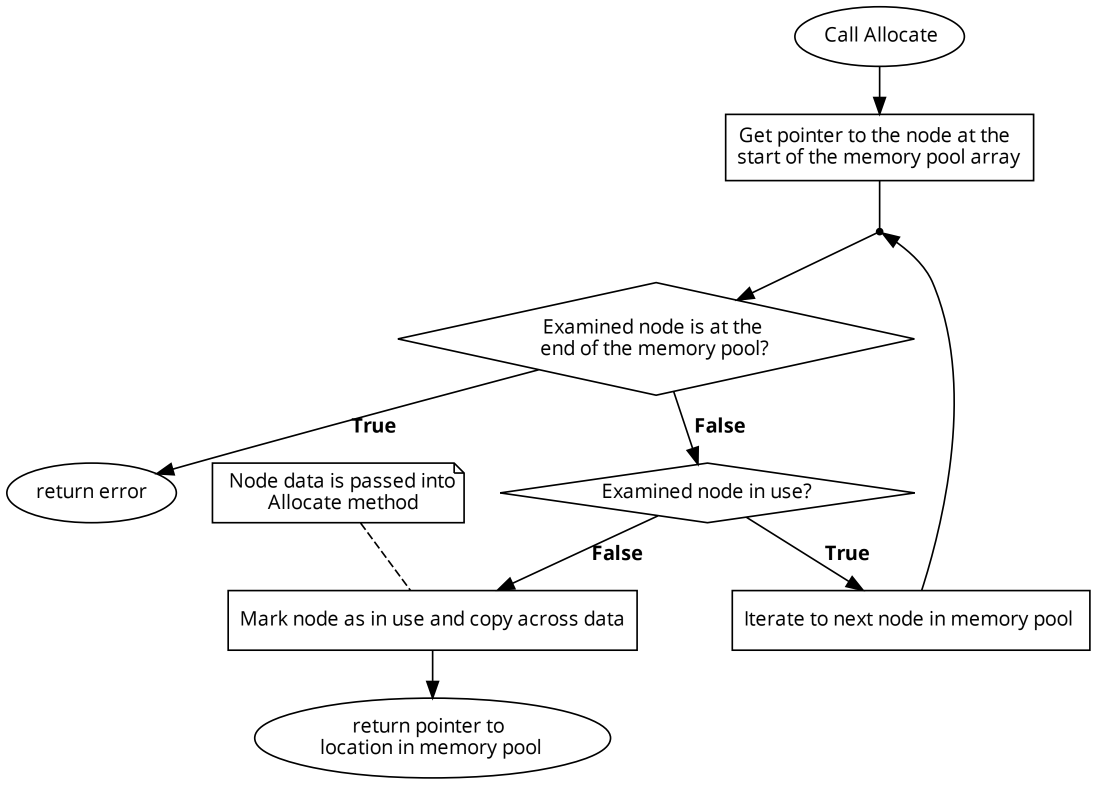

Call Allocate
Get pointer to the node at the start of the memory pool array
Examined node is at the end of the memory pool?
True
False
return error
Node data is passed into Allocate method
Examined node in use?
False
True
Mark node as in use and copy across data
Iterate to next node in memory pool
return pointer to location in memory pool

FIGURE A.14.4: A flowchart depicting the logic of the private "Allocate" method in the StaticList class.

A.15. TECHNICAL DETAILS OF THE BOOTLOADER LIBRARY

The flight software uses the bootloader library to configure the microcontroller's flash memory and decide which application bank to boot into. Generic portable functionality is provided in the BootloaderBase object, and hardware-specific features are provided in the BootloaderImpl object. The BootloaderBase object provides implementations for four methods; "PrepareForProgramming", "ProgramFirmware", "Program", and "FinaliseProgramming".

The "ProgramFirmware" method (Figure A.15.1) acts as an entry point to the firmware programming process. It provides the behaviour of the firmware writing process without requiring the implementation of the methods that perform it.

The "PrepareForProgramming" method calls the "ConfigureFlashProtection" method to disable the flash protection in preparation for writing to the memory. A successful return will permit the function to progress into calling the "InitialiseMicrocontrollerFlash" method.

The "Program" method provides the function calls that perform the write instruction to the hardware. The method first erases the existing flash memory targetted for a write using the "EraseApplicationFlash" method. A successful return progresses to calling the "WriteFirmwareToFlash" method. After writing to the flash memory, the "FinaliseProgramming" method is called to re-enable the flash protection, making it immutable until the next firmware update.

For the Binar-1 hardware implementation, the implementation object, BootloaderImpl, uses the flash_app_start_address_ and flash_app_end_address private variables for determining which bank to target for programming. As there are two application banks to program to, flash_app_start_address_ is set during the constructor for the BootloaderImpl object to be the start address of one of these banks. The flash_app_end_address_ is then updated as the targeted bank's end. Once set, these values are immutable until the next construction at boot. The flash memory is organised into two banks with eight sectors each. Table A.15.1 details how the two megabytes of available internal flash memory were segmented into two boot and two application banks. The boot banks are

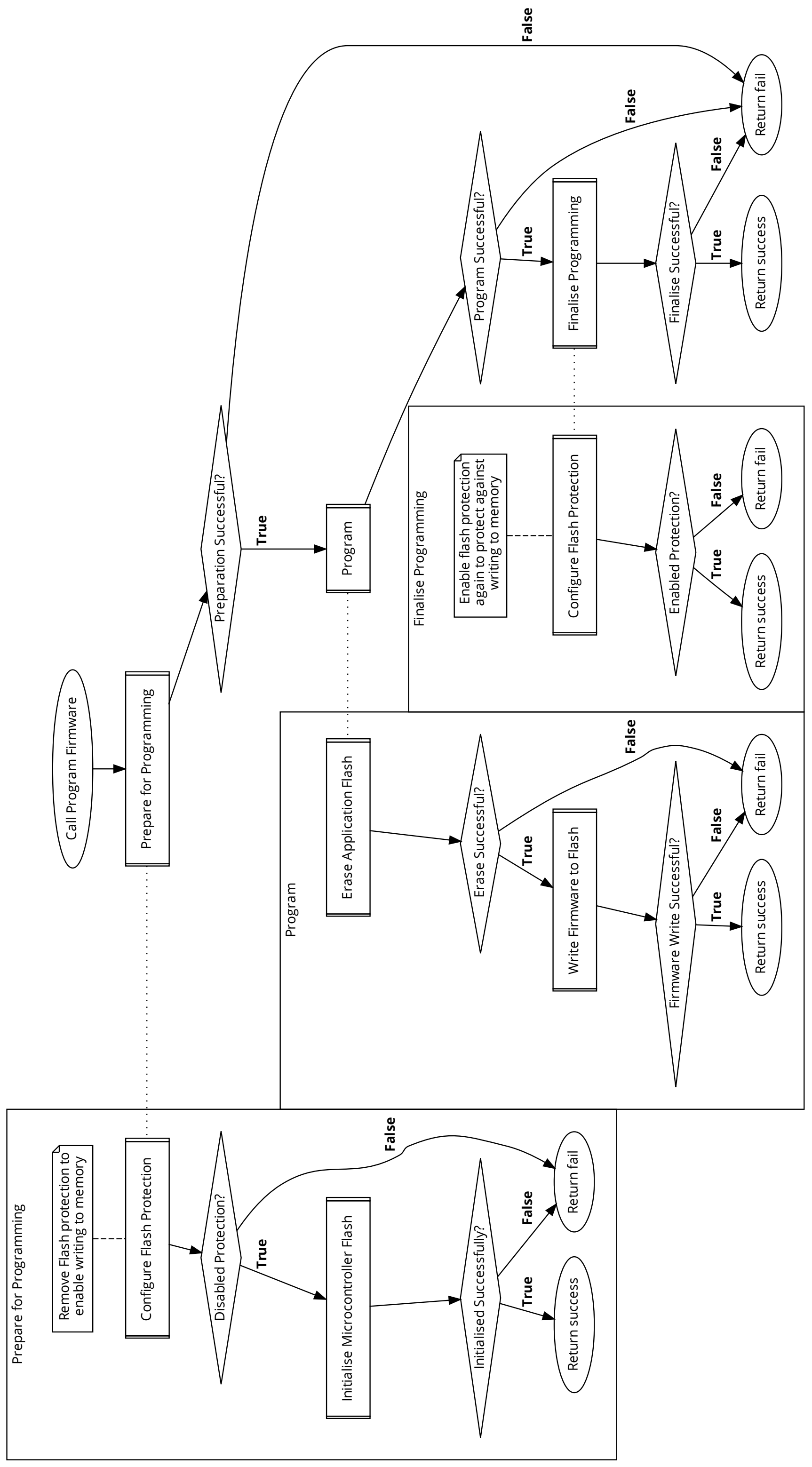


FIGURE A.15.1: A flowchart showing the behaviour provided for programming firmware using the BootloaderBase.

TABLE A.15.1: The flash memory allocation in the primary flight computer on Binar-1. The first megabyte of flash was used to hold the Start-Up/Bootloader firmware in the boot bank, and the second megabyte was for the two application binaries.

Address	**Usage**	**Address**	**Usage**
0x08000000	Boot Bank One	0x08100000	App Bank One
0x08020000		0x08120000	
0x08040000		0x08140000	
0x08060000		0x08160000	
0x08080000	Boot Bank Two	0x08180000	App Bank Two
0x080A0000		0x081A0000	
0x080C0000		0x081C0000	
0x080E0000		0x081E0000	

immutable, but both application spaces can be programmed.

Each 128 KiB sector has independent flash protection. The hardware implementation of the ”ConfigureFlashProtection” method updates the protection at the provided application bank address (Either 0x08100000 or 0x08180000) with the requested modifier.

The ”InitialiseMicrocontrollerFlash” method provides an optional method before writing to flash, where device-specific instructions can be executed. In the Binar-1 hardware, this clears any flash flags that may be present that hinder write access to flash memory - End of program, flash operation error, write protection error, and program sequence error.

As the available flash memory is segmented into two boot banks and two application banks, the ”EraseApplicationFlash” method can only erase the flash memory of the target memory location to avoid corrupting other memory. The method first gets the starting sector to erase, 0x08100000 or 0x08180000, depending on the target application bank being prepared for programming. The boot banks cannot be erased. The four contiguous sectors following the provided base address are erased.

The ”WriteFirmwareToFlash” method (Figure A.15.2) uses the aggregated object firmware_reader_ to receive firmware segments and write them to flash memory. The method loops on the ”ReceivedAllBytes” method on the firmware_reader_ member ob-

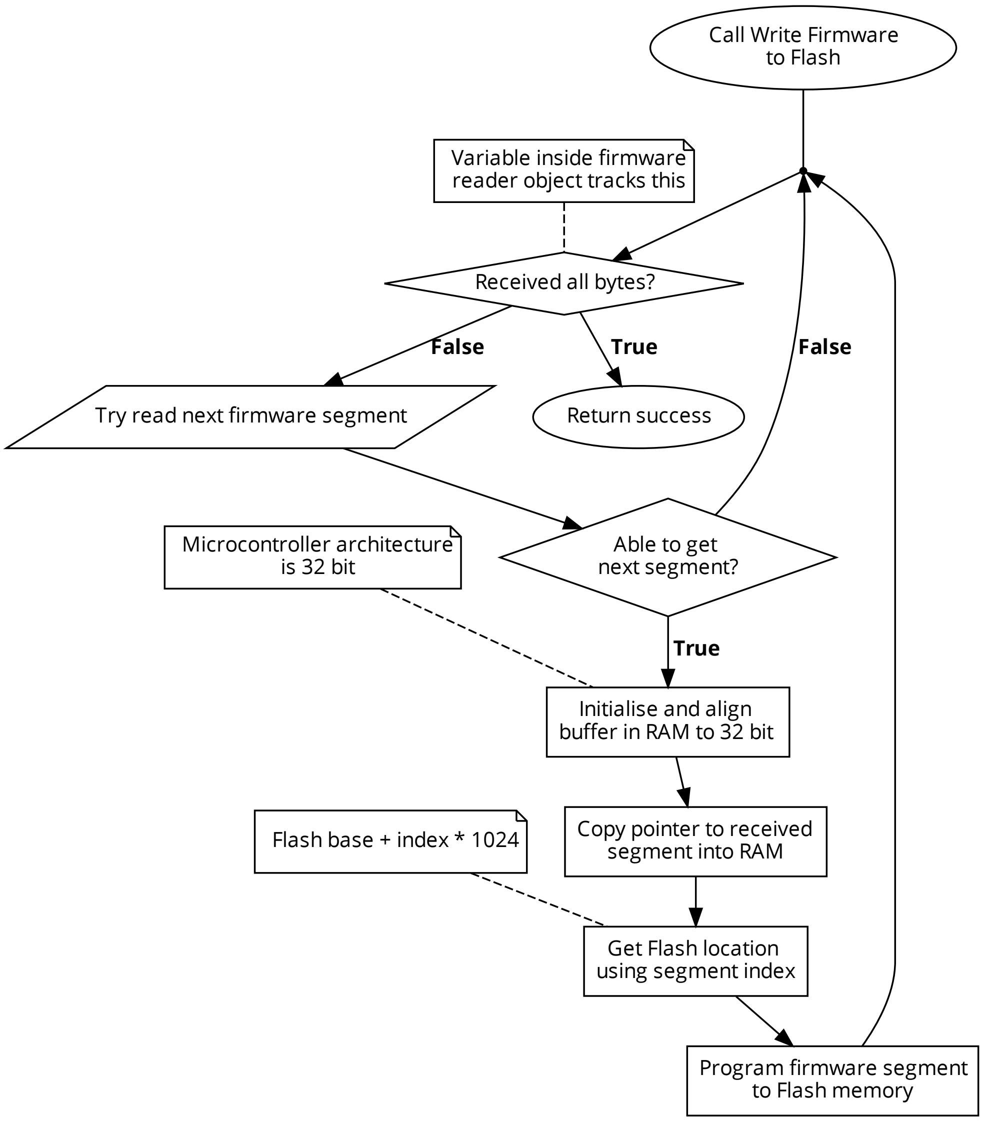

Call Write Firmware to Flash
Variable inside firmware reader object tracks this
Received all bytes?
False
True
False
Try read next firmware segment
Return success
Microcontroller architecture is 32 bit
Able to get next segment?
True
Initialise and align buffer in RAM to 32 bit
Flash base + index * 1024
Copy pointer to received segment into RAM
Get Flash location using segment index
Program firmware segment to Flash memory

FIGURE A.15.2: A high-level overview of the logic for the "WriteFirmwareToFlash" method on the BootloaderImpl object. This method is concerned with programming a received segment of firmware into program memory.

ject. It attempts to receive a firmware segment by calling the ”GetNextSegment” method on the provided FirmwareReaderInterface implementation, returning a firmware segment up to 1024 bytes in length with the corresponding index through the binary. The received raw byte stream is 8-bit aligned, and hence in preparation to be written to flash memory, is copied into a 32-bit aligned temporary buffer in RAM. Using the segment-accompanied index, the location to program to is calculated by multiplying the index by the fixed firmware segment length and summing it with the application bank base address. The firmware segment is then written to the location.

A.16. TECHNICAL DETAILS OF THE FIRMWARE READER LIBRARY

The firmware reader library in the BSF is used to abstract how firmware is received by the spacecraft. It forms a central part of the firmware update process. The Firmware reader object contains several private member variables that aid the firmware receiving process. The segment_tracker_ member is a boolean vector with a maximum length equating to the quotient of the application flash space size and the maximum segment size of 1024 bytes. It keeps track of which segments have been received. The done_ variable is a boolean used to denote when the spacecraft receives the final packet of the firmware binary. When the satellite receives a request for a firmware update, the request packet contains the total number of segments calculated on the ground that will be transmitted. This number is stored in non-volatile memory (Section 3.2.9) before the satellite reboots into the Start-Up/Bootloader firmware. When the FirmwareReader object is constructed in the Start-Up/Bootloader firmware, this value is copied from the non-volatile memory into the number_of_segments_ private member variable to be used during the firmware upload process. It is also used to construct the FirmwareReaderImpl object to resize the segment_tracker_ vector to be the same length as the number of segments to receive.

Alongside the private member variables, the FirmwareReaderImpl object aggregates four private member objects; firmware_segment_, serial_, server_. sender_, and receiver_. The firmware_segment_ member is a data structure that stores a firmware segment along with its corresponding index. It is updated by the ”SetNextSegment” method. Once the firmware segment has been copied into the object's memory, the index is used as an accessor for the segment_tracker_ vector, marking the segment as successfully received. Application code that needs to access the segment does so through the "GetSegment" method. The serial_ member, a pointer to a SerialInterface (Section 3.2.1) implementation, abstracts how the firmware segments are received. The pointer is dereferenced within the "GetNextSegment" method to call the "AsynchronousRead" method on the serial object to receive the encoded byte stream containing the next firmware segment. As the StartupMode server object, server_, is decoupled from how data is transmitted to the spacecraft, the same pointer to the ReceiveInterface implementation, receiver_, is shared between the FirmwareReaderImpl and the StartupModeServer objects. When an encoded byte stream is received through the SerialInterface implementation, the receiver_ private member pointer is dereferenced, and the "SetEncodedBuffer" method is called with the encoded packet. The server_ member is then used to invoke the underlying RPC decoder to extract the next firmware segment stored within the ReceiveInterface implementation. The sender_ pointer to a SendInterface object is used by the "GetNextSegment" method to transmit a response to the ground requesting the next firmware segment to be transmitted.

The "GetNextSegment" method (Figure A.16.1) is used to read a firmware segment from the firmware source. When called, an asynchronous receive begins on the serial input object provided to the FirmwareReader constructor. The method checks if it has been previously called to determine what response message to send to the ground. If it is the first time the method is called, the spacecraft received a "BeginFirmwareUpdate" request and performed a reboot. In this situation, the method will transmit a "BeginFirmwareUpdate" response to signify to the ground station that it is ready to receive firmware packets. Every other call to the method transmits a regular "FirmwareUpdate" response.

After transmitting the response to the ground, an interrupt handler is registered to be triggered upon receiving a message. The execution then enters a loop, waiting for a received byte stream or a timeout. On Binar-1, this timeout was set to 4 seconds. When the spacecraft receives a byte stream, the ReceiveInterface implementation shared with the Start-Up Mode RPC server is dereferenced to insert the received bytes into its

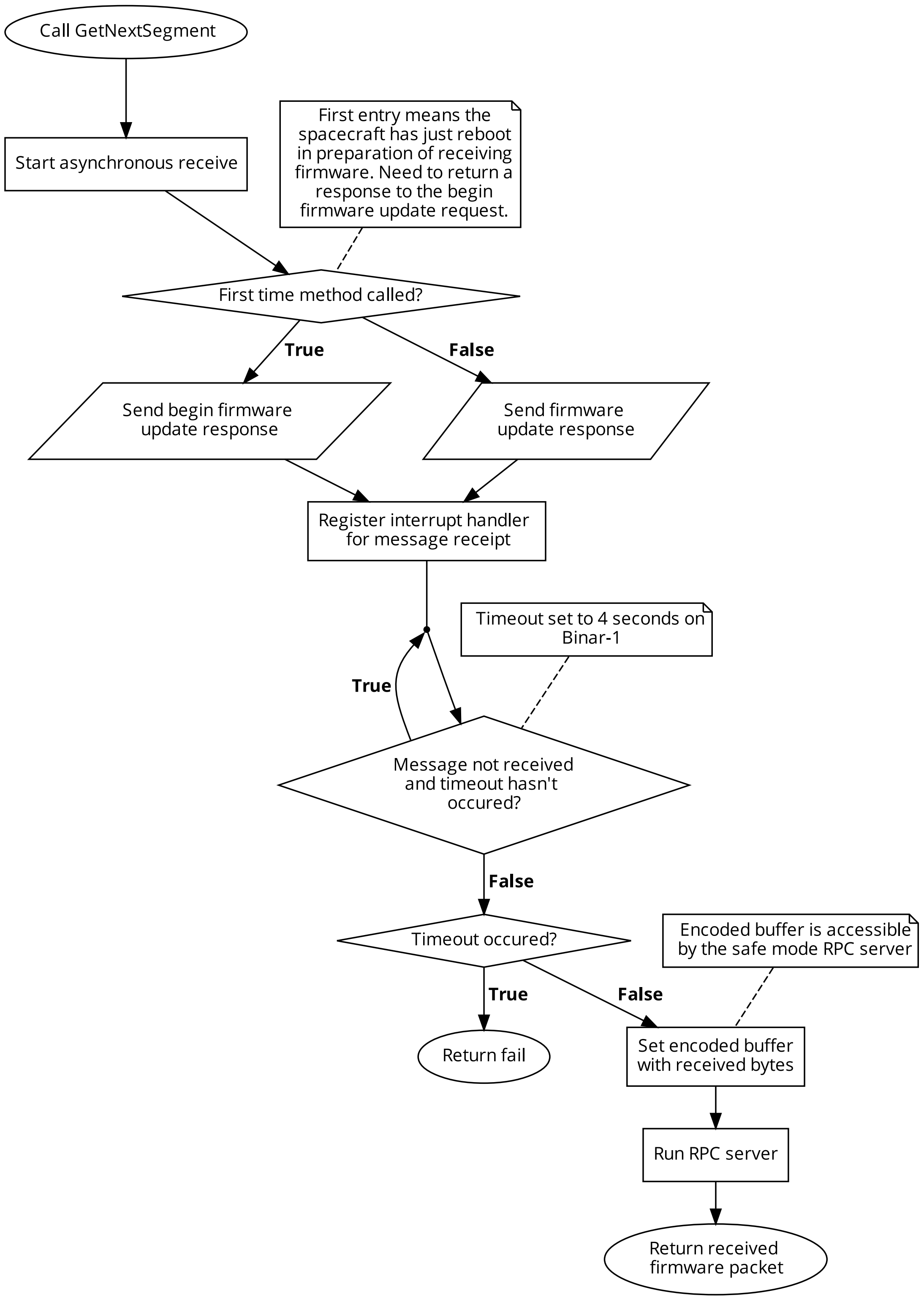

Call GetNextSegment
Start asynchronous receive
First entry means the spacecraft has just reboot in preparation of receiving firmware. Need to return a response to the begin firmware update request.
First time method called?
True
False
Send begin firmware update response
Send firmware update response
Register interrupt handler for message receipt
Timeout set to 4 seconds on Binar-1
True
Message not received and timeout hasn't occured?
False
Timeout occured?
Encoded buffer is accessible by the safe mode RPC server
True
False
Return fail
Set encoded buffer with received bytes
Run RPC server
Return received firmware packet

FIGURE A.16.1: A high-level overview of the "GetNextSegment" method for the Binar-1 FirmwareReaderImpl object. On the satellite, firmware segments are received through the Start-up Mode RPC server.

internal encoded buffer. The Safe Mode RPC server is then run to extract the firmware segment from the byte stream, which is returned to the calling code.

When a ”FinishFirmwareUpdate” request is received, the Start-Up Mode RPC server calls the ”CheckForMissingSegment” method on the FirmwareReader object. This method will iterate through the private member vector segment_tracker_ vector, checking for any false elements, signifying the FirmwareReader failed to receive a firmware segment. If found, the segment index is returned to the ”FinishFirmwareUpdate” request handler, in which case a ”MissedFirmwareSegment” response is transmitted to the ground. The ground station will resend the failed segment before issuing a ”Finish-FirmwareUpdate” request again. Once all segments have been received, the RPC server calls the ”Finish” method on the FirwmareReader object. This method updates the private member variable done_ to a true state. The Start-Up/Bootloader firmware using the FirmwareReader object loops on the ”ReceivedAllBytes” method, which returns the status of the done_ variable. Hence, using this method, the Start-Up/Bootloader firmware knows when it is safe to progress with the execution.

A.17. TECHNICAL DETAILS OF THE STARTUP LIBRARY

The start-up library in the BSF abstracts the start-up functionality required by the satellite. For Binar-1, this functionality included tracking the boot count, implementing the ISS separation timer, and running system self-checks.

When the StartUp module (Figure A.17.1) is executed, the aggregated BootCount object stores a counter in non-volatile memory that keeps track of the number of spacecraft boots. If the boot count is zero, indicating the satellite has just been released from the ISS, the ”StartTimer” method on the BootTimer object is executed. In the Binar-1 implementation for this method, the execution was trapped in a thirty-minute loop. Inside this loop, the only logic executed by the flight computer is a watchdog reset. When elapsed, the antenna module is instructed to deploy by enabling the burn wire GPO (Section 3.2.1) to release the antenna doors. The ”IncrementBootCount” method of the BootCount object is then used to indicate that the spacecraft has satisfied

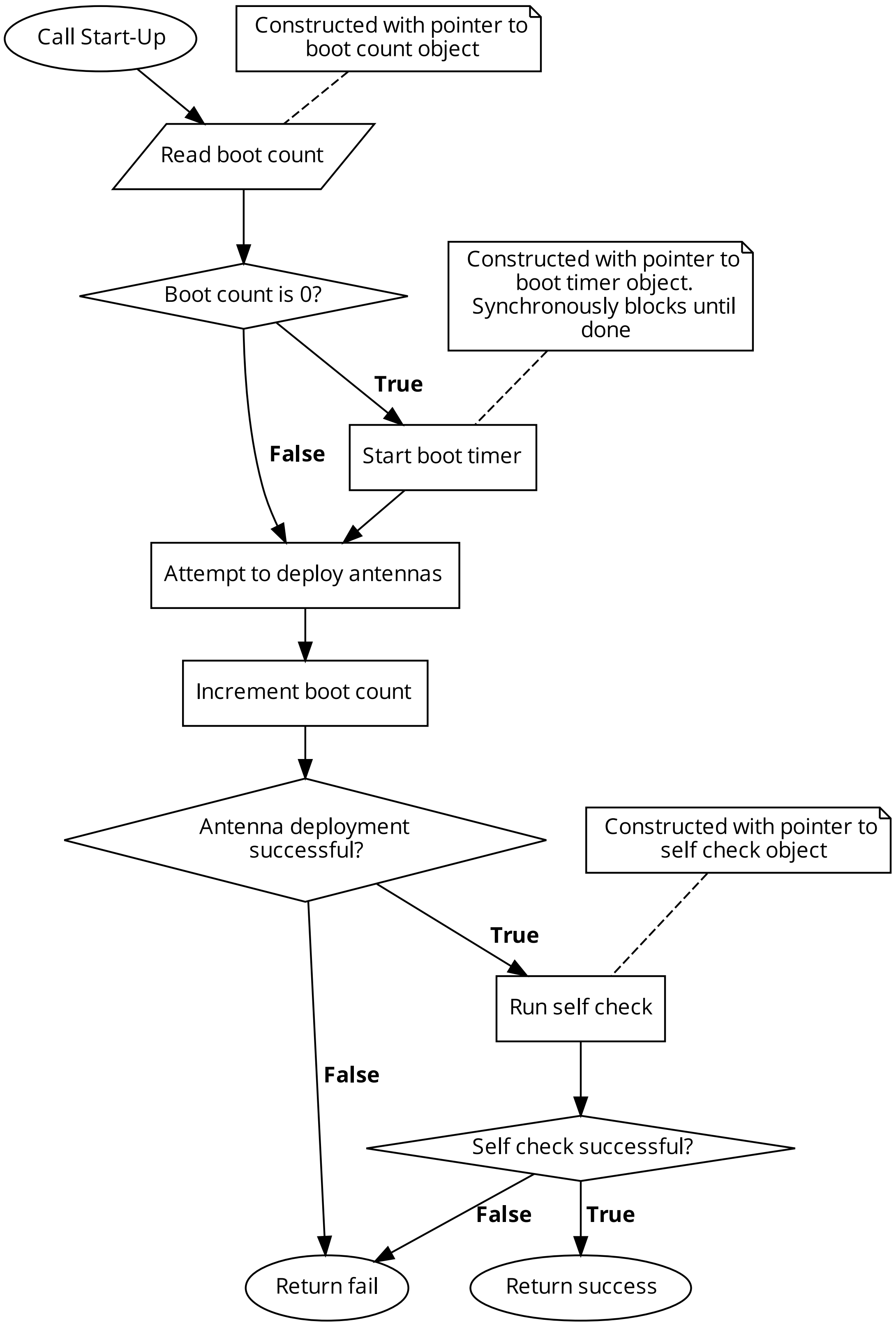

Call Start-Up
Constructed with pointer to boot count object
Read boot count
Boot count is 0?
Constructed with pointer to boot timer object. Synchronously blocks until done
True
False
Start boot timer
Attempt to deploy antennas
Increment boot count
Antenna deployment successful?
Constructed with pointer to self check object
True
Run self check
False
Self check successful?
False
True
Return fail
Return success

FIGURE A.17.1: A high-level overview of the logic for the StartUp functor in the Start-Up library. This object is used during the boot of the spacecraft. It is designed to be configurable to run any tasks required at start-up for the mission.

its deployment requirements, so subsequent reboots do not invoke the boot timer. Execution progresses into examining if the antenna doors are deployed correctly. If not, a fail is returned to the calling code. Likewise, a failed self-check will return fail to the calling code. The Start-Up/Bootloader firmware uses the propagated failures to determine if a Safe Mode entry (Section 3.7.2) is required.

A.18. TECHNICAL DETAILS OF THE DETUMBLE LIBRARY

The Detumble library is used to slow the angular precession of the spacecraft when it is ejected into orbit. Detumbling is performed by executing the public method "Run". This method is intended to be used within a loop to attempt to stop the angular procession of the spacecraft repeatedly. Figure A.18.1 provides a high-level depiction of the method logic. The method starts by attempting to read the onboard IMU (Section 3.2.3). If the reading is successful, the method will calculate the normalised scalar rotational velocity. Comparing this value against the stop_condition_dps_ value initialised in the constructor for the module, a success condition will be returned if spacecraft tumbling has been slowed sufficiently. If further detumbling actuation is required, however, the method will use the magnetometry data read from the IMU to calculate the rate of change of the magnetic field measurements. The maximum magnetorquer dipole is then applied to each magnetorquer axis with the polarity set depending on the axial rate of change. The method returns fail as the stop condition has not yet been met, instructing the calling code that the "Run" method needs to be called again.

A.19. TECHNICAL DETAILS OF THE SCHEDULING SERVICE LIBRARY

Flight application software uses the scheduling service library in the BSF to manage tasks to be executed on orbit. The publicly available methods are "Create", "Read", "Update", and "Delete". Adding a new task to the task scheduler is done through the

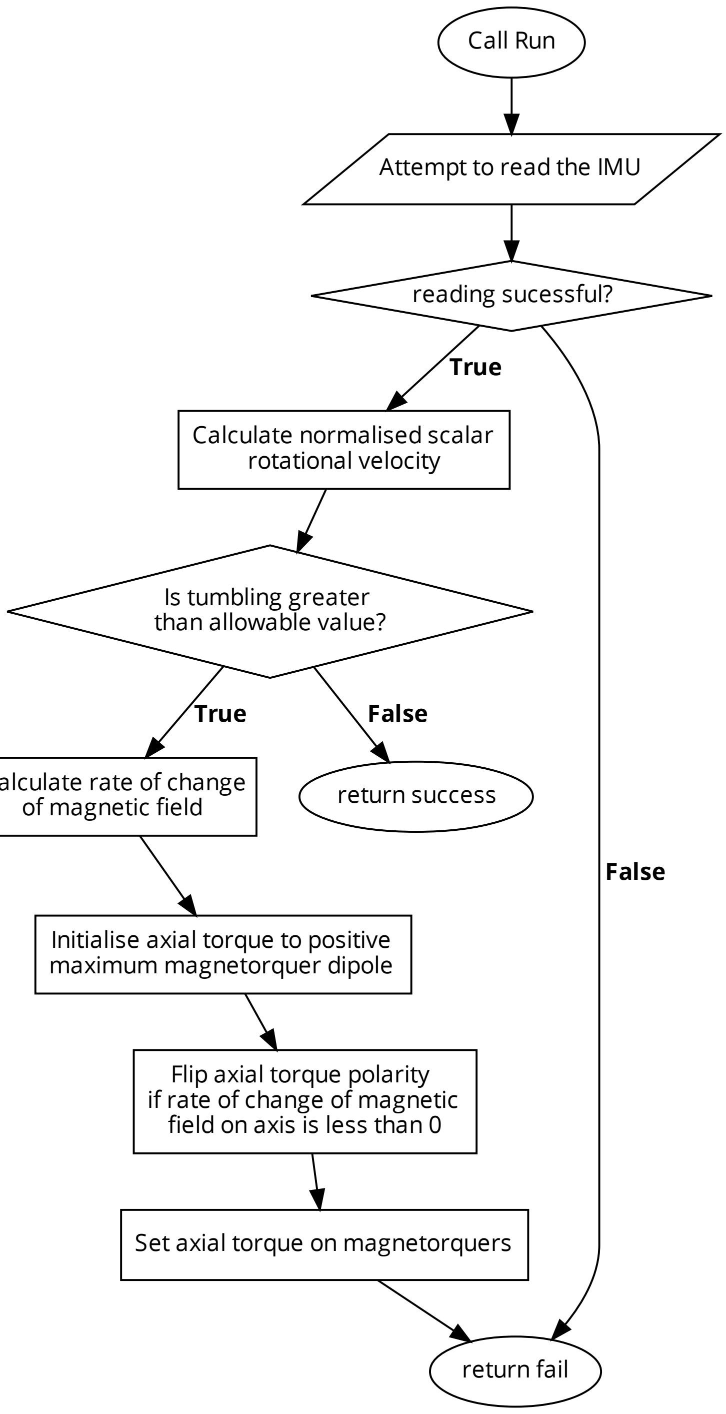

Call Run
Attempt to read the IMU
reading sucessful?
True
False
Calculate normalised scalar rotational velocity
Is tumbling greater than allowable value?
True
False
Calculate rate of change of magnetic field
return success
Initialise axial torque to positive maximum magnetorquer dipole
Flip axial torque polarity if rate of change of magnetic field on axis is less than 0
Set axial torque on magnetorquers
return fail

FIGURE A.18.1: A flowchart depicting the logic of the "Run" method of the Detumble class. The method is intended to be run in a loop until the desired stop condition is met, signified by returning success.

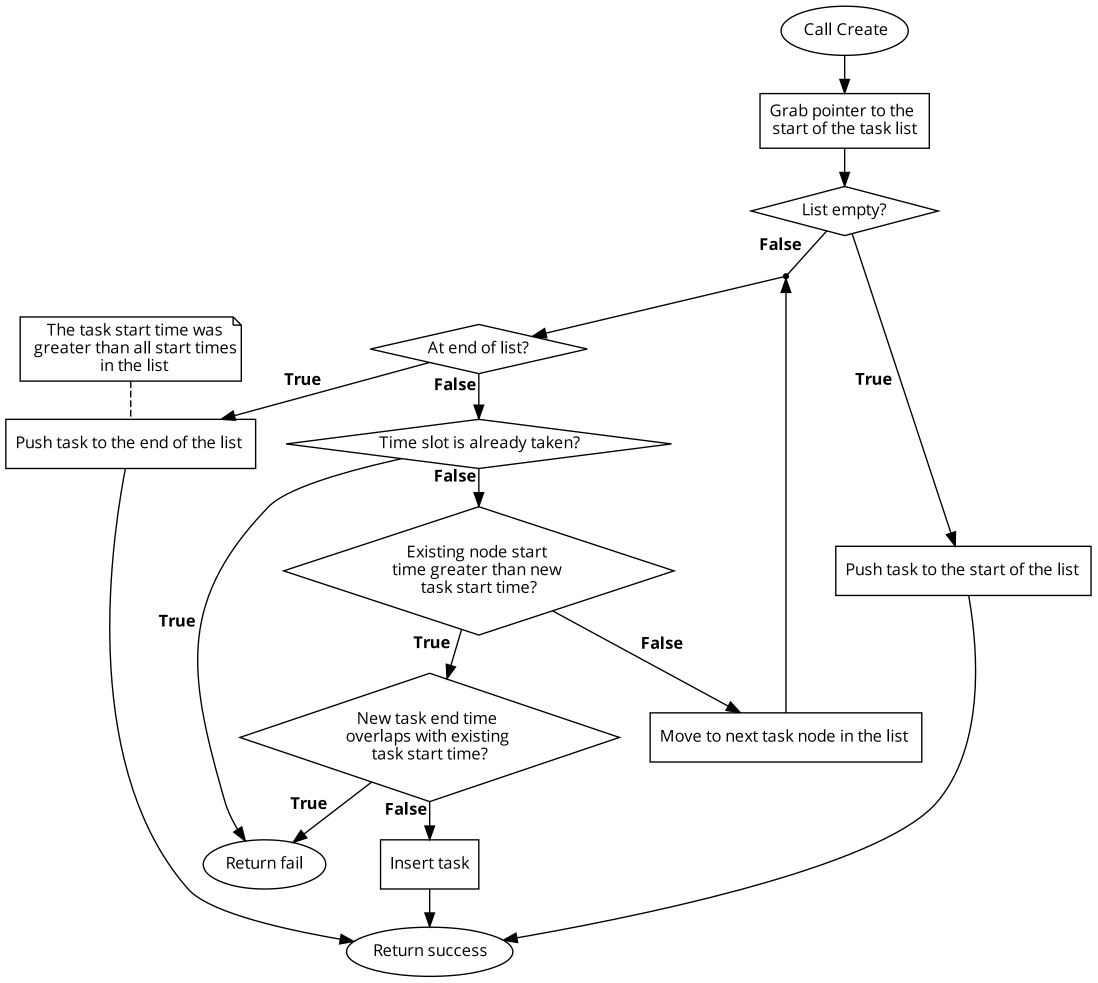

Call Create
Grab pointer to the start of the task list
List empty?
False
True
Push task to the start of the list
At end of list?
True
False
The task start time was greater than all start times in the list
Push task to the end of the list
Time slot is already taken?
True
False
Existing node start time greater than new task start time?
True
False
Move to next task node in the list
New task end time overlaps with existing task start time?
True
False
Return fail
Insert task
Return success

FIGURE A.19.1: A high-level overview of the "Create" method on the TimeSeries-TaskScheduler object. This method takes a task to schedule and inserts it into the list at the correct location if the task queue is not full.

"Create" method (Figure A.19.1). This method, called with a task to execute and a start and end time, iterates through the private member StaticList object, identifying where an insertion is possible. The "Create" method will fail if another task exists in the list with the same start time as the new task attempting to be inserted. Furthermore, the method will fail if the inserting task's end time overlaps with the existing task's start time already in the list. In the situation where these conditions do not exist, however, the method will insert the requested task in the list, sorted according to task start times.

As the tasks are sorted by task start time, reading from the list (Figure A.19.2) needs to pop the task at the head of the list using the "Pop" method on the StaticList object. Once removed from the list, the pointer to the AtSpecifiedTimeInterface implementation

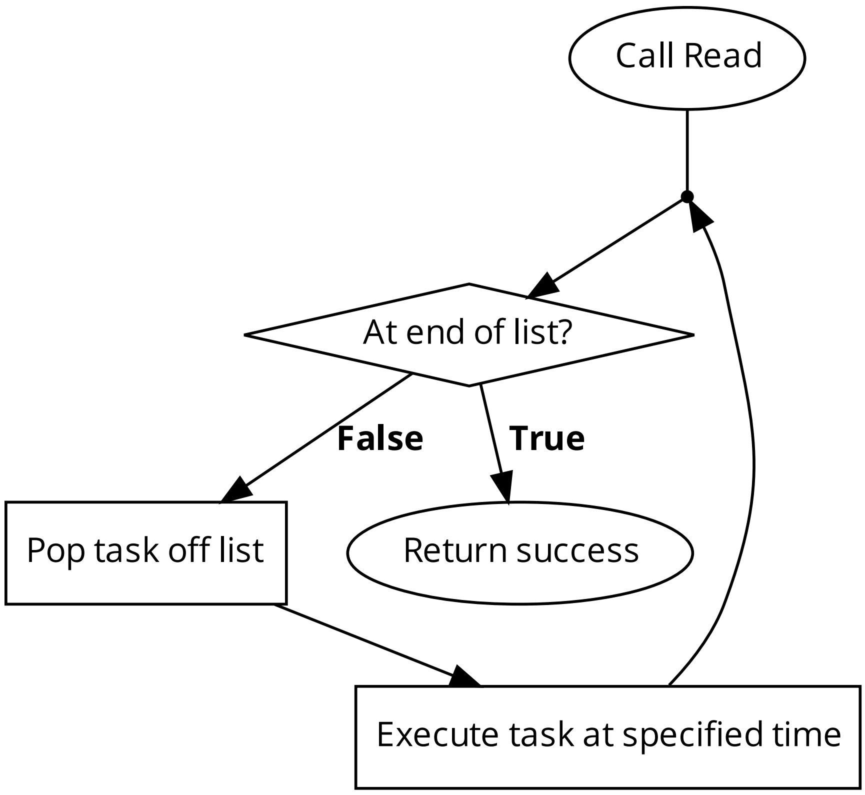


FIGURE A.19.2: A high-level overview of the ″Read″ method on the TimeSeries-TaskScheduler. This method is used by application code to read tasks from the queue for execution.

is dereferenced and provided the task to execute. This approach abstracts how the task is run in the software.

Support for changing the task scheduled in the StaticList object is achieved through the ″Update″ method (Figure A.19.3). This method, called with the task to update, iterates through the StaticList object to find the requested task. Once found, it temporarily creates a backup of the task and then proceeds to erase the old task using the ″Delete″ method on the StaticList object. The ″Create″ method is then called with the new replacement task. If creating a new task fails, the temporary backup is restored.

If a previously scheduled task is no longer required, it can be removed from the list using the ″Delete″ method (Figure A.19.4). This method iterates through the StaticList object, checking for the task to remove. When found, the task is removed using the ″Erase″ method on the StaticList object.

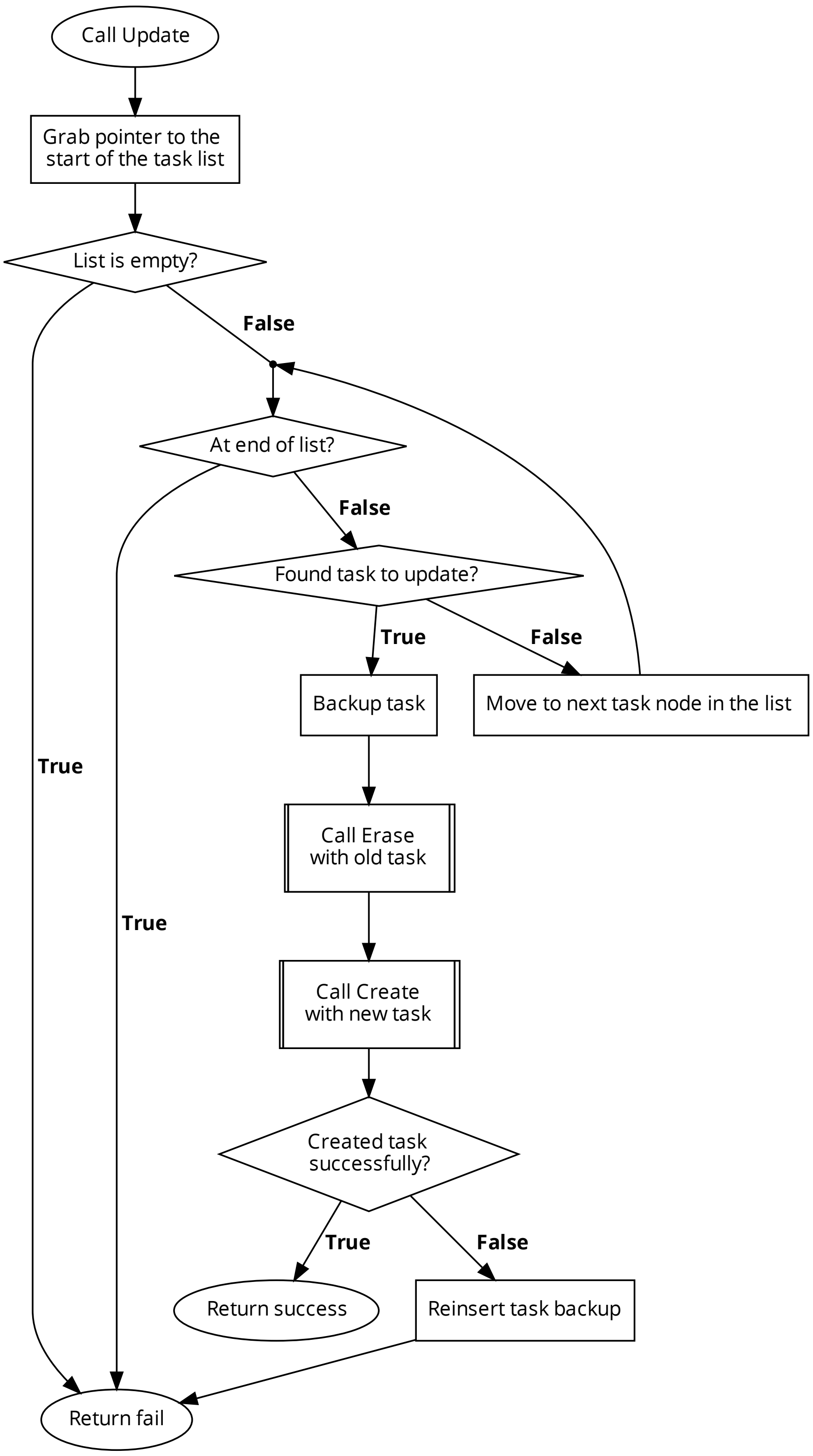


FIGURE A.19.3: A high-level overview of the "Update" method on the TimeSeries-TaskScheduler object. This method is used to update an existing task in the queue.

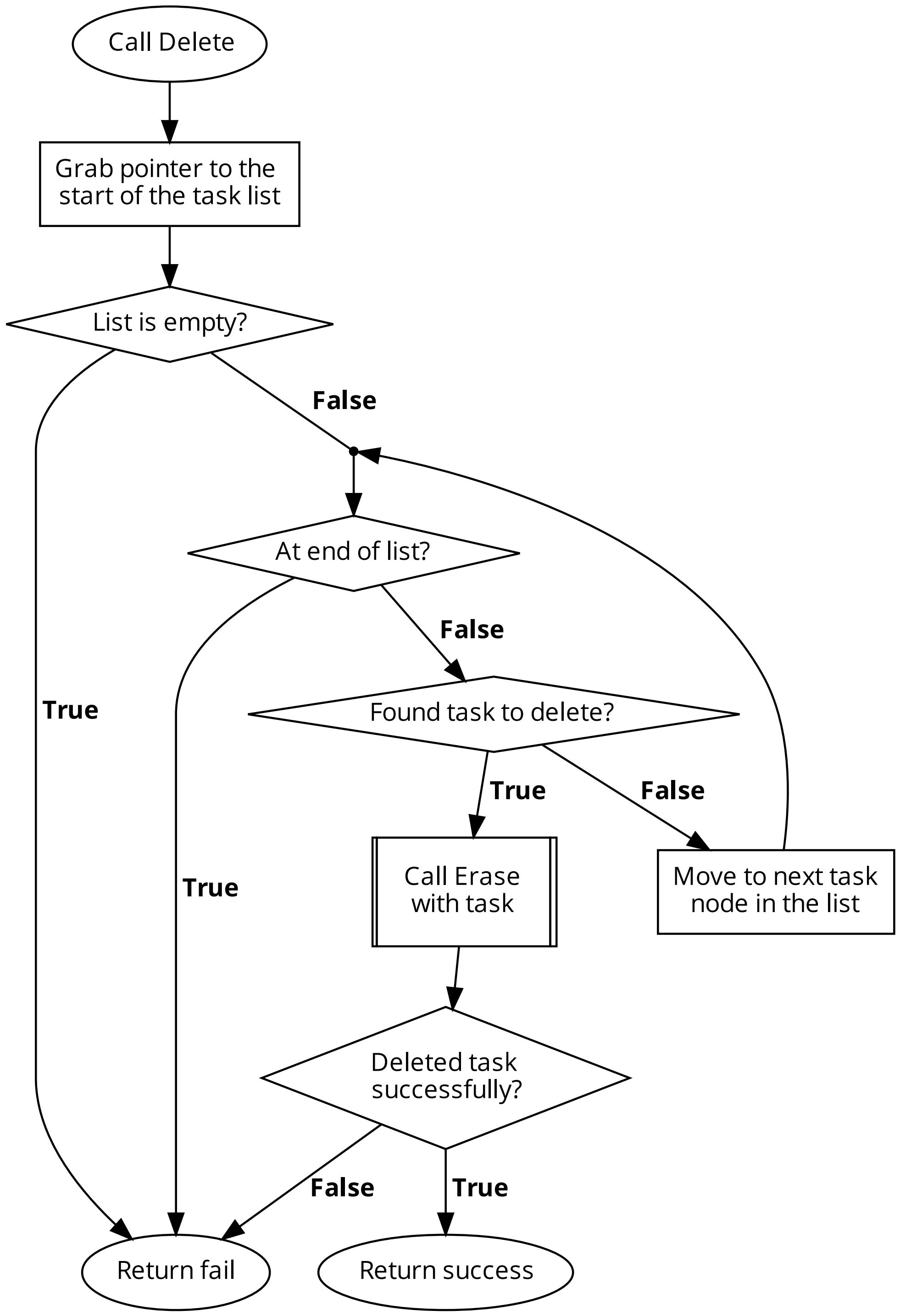


FIGURE A.19.4: A high-level overview of the "Delete" method on the TimeSeries-TaskScheduler object. This method is used to remove an existing task from the queue.

The tasks available on Binar-1 comprised payload usage and system checks. The ″Take Picture″ task enables the camera payload on Binar-1 to take a photo. The configurable parameters for the task allowed for the white balance, exposure, red gain and blue gain to be adjusted. The captured image is stored on the payload computer and requires further command from the ground station to retrieve. After taking images, the ″Get Image Names″ task can be scheduled to return a comma-delimited string of the names of all the images on the storage. These are saved with a date and time stamp. The returned image names from the task execution are stored on the onboard storage (Section 3.2.5) to retrieve during a pass over the Curtin University ground station. As the images taken by the payload require multiple passes to downlink in full resolution, the ″Generate Thumbnail″ task can be scheduled to create thumbnail versions of all images on the payload. The task has the configurable parameter of the thumbnail width in pixels to use for generation. One option supported for downlinking images was compressing the image to reduce the file size. This task had two configurable parameters, the image name to compress retrieved from the ″Get Image Names″ task and a percentage by which to compress the image. Another option for downlinking images was to segment them into several tiles using the ″Split Image″ task. This task takes two configurable parameters; the file name retrieved from the ″Get Image Names″ task and the number of sub-images to segment into. Each image segment could then be downlinked separately and reassembled on the ground. To downlink an image, the image data first needs to be transferred from the payload computer storage to the onboard storage to reduce the time required for the payload to be enabled for power conservation.

Transferring the image is done using the ″Get Image″ task. The task would be scheduled when the spacecraft was out of range of the Curtin University ground station to prepare the image for downlink when it became visible. The configurable parameter was the name of the image to downlink. Alongside the Earth observation camera, Binar-1 also had a star tracker camera. This payload was used through the ″Use Star Tracker″ task. This task powers on the payload camera, waits for the payload to take an image, and then because the image size is significantly smaller than the Earth observation camera, immediately reads the image data into RAM to be transferred into the onboard storage. The image can then be downlinked over the ground station.

The ″Self-Check″ task uses the Self Check library (Section 3.5.3) to identify faults on the spacecraft. Scheduling the task requires the check level; critical systems, non-critical systems, or both.

The ″Move Manipulator″ task is used to actuate the emulated robotic manipulator using the Manipulator library (Section 3.2.11). The configurable parameter for the task is the angle by which to actuate the manipulator.

The ″Detumble″ task uses the Detumble library (Section 3.5.1) to slow the angular procession of the spacecraft. On boot into the application code, the detumble task is automatically scheduled. The ground station can also schedule the task if further detumbling is required on orbit.

A.20. TECHNICAL DETAILS OF THE SELF CHECK LIBRARY

The flight software uses the self-check library to examine critical and non-critical systems onboard the spacecraft for faults.

The critical systems on Binar-1 comprised the power, temperature, and error logging systems. The power system check iterates over every voltage and current analog channel using the Analog library (Section 3.2.1) and compares the sampled reading against hardcoded nominal bounds set when characterising the satellite. The same process also applies to the temperature system. The error logging system checks the error logs in the non-volatile registers detailed in Section 3.2.9 for entries made during system runtime.

The non-critical systems on Binar-1 comprised the magnetorquer, GPS, and IMU systems. The magnetorquer system check and IMU system check objects invoke the ″GetStatus″ method of their hardware abstracted objects, discussed in Sections 3.2.3 and 3.2.4. The GPS system check object attempts to take the device from the ownership manager and perform a GPS reading, as seen in Figure A.20.1. As the GPS is held in a low power state, it should be able to hot-start per the device datasheet in one

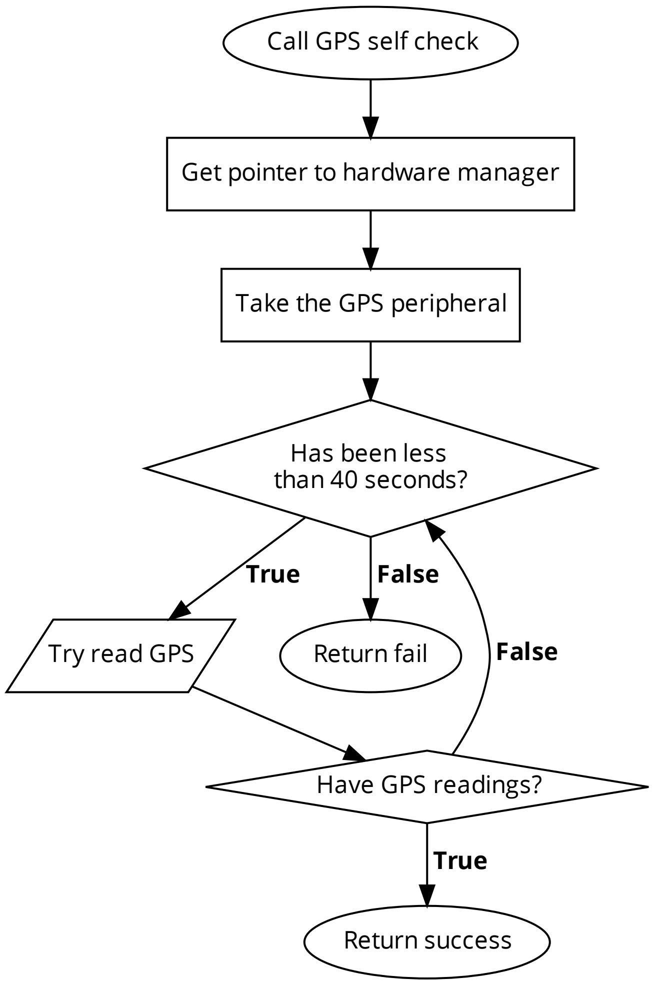


FIGURE A.20.1: A high-level overview of the "CheckSystem" method of the aggregated non-critical GPS system derived from the SystemCheckInterface.

second. However, it is given 40 seconds to start up, one second more than is required to procure a satellite fix from a cold start. For a 40-second duration, the GPS is queried for readings. If readings are received, the device is successfully verified. The system check fails if the 40 seconds lapse with no valid readings.

A.21. TECHNICAL DETAILS OF THE RPC LIBRARY

The BSF uses the RPC library to facilitate communications between the ground station and the spacecraft. Table 3.4 exhaustively lists the available procedure calls in the various operation modes. This section describes each of the modes.

The ”GetVoltages”, ”GetCurrents”, and ”GetTemperatures” methods use their corresponding private member analog reader to collect raw analog readings to return to the ground.

Ping is used to receive a quick acknowledgement response from the spacecraft to verify that the flight computer is online.

The ”FirmwareUpdate” method takes three request types as parameters; begin firmware update, firmware update, and finish firmware update. As firmware cannot be updated in Application Mode, the firmware update and finish firmware update requests are ignored. If a firmware update request is received in Application Mode, the system_ctx_ pointer is used to write the target flash bank and the number of segments to non-volatile memory. Following this, it reboots into Start-Up Mode to complete the firmware update process. When rebooted, the flight software sends a begin firmware update response to the ground, signalling it is ready to start receiving firmware segments. The Start-Up Mode RPC server handles receiving firmware using the privately aggregated firmware_reader_ object, an implementation of the FirmwareReaderInterface from the firmware reader library (Section 3.4.2). When a firmware update request is received with a firmware segment and corresponding index, it is passed to the firmware reader object to be handled through the ”SetNextSegment” method. When a finish firmware update request is received, the RPC server will check for missing segments using the ”CheckForMissingSegment” method before finally transmitting a finish firmware update response.

The ”ScheduleTask” method determines what task the ground operator has requested and calls the ”Create” method on the private scheduler_ member to insert it into the task list. The application thread has access to the same scheduler, using it to execute the tasks requested from the ground.

”GetErrorRegisters” uses the private SystemContextInterface pointer to read and return the error registers from non-volatile memory to the ground operator.

The ”DeleteImages” request uses the private payload_ member to enable the payload camera, send a request to the payload computer to delete all stored images, and then power it down.

"GetImuData" requests access to the IMU (Section 3.2.3) from the ownership manager (Section 3.2.2) to perform a reading of the gyroscope, accelerometer and magnetometer by using the "ReadIMU" method.

The "DownloadLog" RPC method accepts two request types; begin system log download and system log download. Receiving the begin system log download request, the "PrepareLog" method of the system logger library (Section 3.2.10) is called with the log type and requested hour to retrieve from memory. The log is stored temporarily in RAM to release the storage peripheral, and the response contains the number of segments in the log to transmit. The response message can contain a maximum segment of 1024 bytes. After the ground receives the response, it transmits a system log download request to the CubeSat to start receiving segments. The "DownloadLog" method handles these by taking the next available sequence of 1024 bytes using the GetLogChunk on the system logger object, serialising them to be returned to the ground.

The "DownloadManipulatorLog" RPC method operates similarly. It also has two request types; begin manipulator log download request and manipulator log download request. The retrieval of bytes, however, differs in accessing; the manipulator logs are not part of the system logger object. Hence, the ManipulatorLogger type (Section 3.2.11) is used to prepare the logs for download from RAM and segment them into transmittable segments.

The "DownloadImage" RPC method accepts three request types; begin image download, image download request, and image names download request. The image download process is the same as the "DownloadLog" and "DownloadManipulatorLog" request types; the image to download is moved from onboard storage into RAM using the "PrepareImageForDownload" method and is segmented for download using the "GetImageChunk" method. The image names download request retrieves the string of available images from the onboard storage through the "GetImageNames" method to aid in selecting an image to download.

The beacon thread in the application code checks against an externally-linked file-scoped reference disable time to determine if beacons are to be muted from the CubeSat.

Ninety minutes must have elapsed for beacons to be transmitted again. When the ”DisableBeacon” request is received, this reference time is updated to the current time on the spacecraft, starting another ninety minutes of silence from the satellite.

As the flash memory for the flight computer on Binar-1 was configured to support multiple application binaries, the ChangeFlashBank RPC method was added to swap between them. This method writes the requested target flash bank register into non-volatile memory and then performs a system reboot. Before booting, the Start-Up/Bootloader firmware will query this bank to determine which application will be booted.

The error registers in non-volatile memory store the state of errors that occurred on the satellite and instruct the CubeSat to boot into Safe Mode. To escape Safe Mode, ground operators can erase the status of specific error registers using the ”Clear Error Registers” request.

The Binar-1 satellite had a redundant BCM onboard to assume spacecraft control in case of a major fault. The redundant BCM supported all of the RPC methods discussed in this section. However, this BCM was not connected to the transceiver, so transmissions intended for it first had to pass through the primary BCM. Only the supervisory computers on the BCM could communicate with each other. The Router object was used to route the transmission to the intended target. When the ”Forward” method is called to pass a message down the computer chain, the unique hardware identifier for the computer is consulted against a lookup table to determine if the computer belongs to the primary or redundant BCM. Discerning the location would determine which private member to use for forwarding the message. After forwarding the message, the Router object waits for a response with a ten-second timeout before the RPC transaction is cancelled. If a response is received, it is sent back up the computer chain using the ”SendBuffer” method on the sender_ object. When the correct computer receives the message, the request object is run through a switch statement to find the appropriate method to handle the request. For all of these methods, after performing the request and collecting the required information, the response object is serialised using the protobuf library and transmitted to the ground by dereferencing the serial_ pointer and calling the ”Send” method with the resulting byte stream.

REFERENCES

[178] STMicroelectronics, *FreeRTOS on STM32 - 3 FreeRTOS introduction*, Oct. 2019. [Online]. Available: `https://www.youtube.com/watch?v=Svlv099-jRA`.

APPENDIX B

First Author Journal Publication Reprints

B.1. **Paper 1** – Reliable Software Development for Small Satellite Missions: A Binar-1 Case Study

Journal of Small Satellites, under review.

Stuart R. G. Buchan, *Nathaniel D. Brough, Fergus W. Downey, Daniel C. Busan, Benjamin A. D. Hartig, Robert M. Howie, Phil A. Bland, Jonathan Paxman, and Martin Cupak*

Reliable Software Development For Small Satellite Missions: A Binar-1 Case Study

Topic area - other relevant topics (small satellite software development)

Stuart R. G. Buchan, Nathaniel D. Brough, Fergus W. Downey, Daniel C. Busan, Benjamin A. D. Hartig, Robert M. Howie, Philip A. Bland, Jonathan Paxman, Martin Cupak
Space Science and Technology Centre, Curtin University
Perth, WA, Australia

Corresponding author: Stuart Buchan - stuart.buchan@postgrad.curtin.edu.au

ABSTRACT

As the computers enabling small spacecraft increase in capability and decrease in cost, the role that flight software plays in mission potential is becoming increasingly important. Consequently, common pitfalls that are inherent in software design can endanger small spacecraft mission success. A review of space mission mishaps of the past two decades reveals that code reuse, software testing and poor code base maintenance were frequently responsible. This paper provides recommendations for software design for small spacecraft to increase the likelihood of success for small satellite missions, using Binar-1, Western Australia's first locally developed spacecraft, as a case study. In the first application of the SOLID principles of object-oriented programming (a set of principles intended to increase the maintainability and flexibility of a code base) to spacecraft software development, we demonstrate safe code portability between missions, and an increase in test coverage of the code base through test-driven development. Good code base maintenance is discussed with a focus on creating a development pipeline involving continuous integration and continuous deployment of spacecraft software. Implementing the recommendations for small spacecraft software design on the Binar platform has resulted in the development of safe, reliable and reusable software.

1. Introduction

In a 2005 study of 156 on-orbit spacecraft failures, Tafazoli found that six percent were attributed to incorrect software commands and flaws, demonstrating the importance of a robust software design methodology to a successful mission (Tafazoli, 2009). Seventeen years later, as more functionality is implemented in, or enabled by software running on smaller, cheaper, and more powerful computing equipment, this is becoming more true than ever (Leveson, 2012). However, the CubeSat form factor and onset of electronics miniaturisation has dramatically reduced the cost required to conduct research in space, enabling universities to explore this field with teams of students. This is not without risk, however, as the skills available in the educational branch of the space industry may struggle to safely implement the complex software powering future spacecraft. This is reflected in the commonality of mission-critical risks presenting in the spacecraft being produced from these groups (Latachi et al., 2020).

There is little published work documenting the role software plays in CubeSat failures (Latachi et al., 2020). Due to this, new developers may potentially suffer the same pitfalls as developers before them. As reliable software design is critical in risk mitigation (Latachi et al., 2020), it is common to see university CubeSat groups seek out mature software frameworks for implementation into their projects. Open source software frameworks such as NASAs Core Flight Software aim to aid in cost and schedule savings for mission development, whilst improving the reusability and quality of a code base. The framework has proven flight heritage on missions such as the Lunar Reconnaissance Orbiter and the Lunar Atmosphere and Dust Environment Explorer, demonstrating its suitability in a space mission (McComas et al., 2016). JPLs open source F Prime framework aims to achieve similar benefits for small satellites. Having been used on missions such as ASTERIA, Lunar Flashlight and NEA Scout, it has proven flight heritage in CubeSat applications (Bocchino et al., 2018). If CubeSat developers are willing to select their OBC for compatibility with a framework, KubOS provides a software stack that can aid in rapid development by supplying the API for interfacing with hardware (Plauche, 2017). It is also common for university CubeSat groups to purchase and use a commercial Software Development Kit (SDK) for their launches, attracted to the appeal of flight heritage with in-orbit validation and guaranteed long term support (Doyle et al., 2020). Whilst using a mature framework or an SDK can be useful for university groups to achieve the short term goal of a launch, it can become a detriment to a space program with the intention of flying multiple spacecraft. To fully benefit from the use of existing software solutions, the software written for the CubeSat needs to conform to the existing structure. This can couple the code base to the specific software solution being used, making it difficult to break from it in the future if it no longer meets mission requirements. In the instance of solutions with proprietary software, use on subsequent missions may carry with it extra licensing costs. Using an SDK for the development of EIRSAT-1, Doyle et al. mention that the knowledge base of the software development team regarding mission software design is encompassed within the confines of using the licensed SDK, further stressing the potential difficulty in breaking from the use of a kit. It was recommended that teams with the intention of flying multiple missions seek out full in-house development (Doyle et al., 2020). This recommendation is often heeded in the case where a University group wishes to produce reliable software that can be reused on future missions (Askari et al., 2017)(Hishmeh et al., 2009).

From an independent analysis however, Latachi et al. identified custom CubeSat flight software as a highly critical system in the success of a mission due to the lack of flight heritage. To decrease the risk of mission failure, four areas were identified to promote the development of a safe, generic software architecture for spacecraft: modularity, reliability, re-usability and extensibility (Latachi et al., 2020). These sentiments are seen to be reflected in other University groups that have written their own custom CubeSat software (Alanazi and Straub, 2019). Modularity was mentioned to be a key driver for the development of ITASAT, where Carrara et al. mention that a major objective of their CubeSat project is to develop a multi-mission platform (Carrara et al., 2017). Coupled with modularity, extensibility is demonstrated in the software architecture for the Masat-1 and Kysat CubeSat missions, where a distinctive layered approach has been adopted. Each loosely-coupled layer consists of various modules, each encapsulating their own set of functionalities that do not interfere with one another (Latachi et al., 2020)(Hishmeh et al., 2009). Developing the code base in such a manner allows for hardware to be changed with only minor modifications to the software, as only the module encapsulating the driver for the hardware needs to be changed. Moreover, extra modules can be added to layers without breaking the software structure, permitting extensibility of the code base. Whitney et al. propose that a key element in the extensible software design of OpenOrbiter is the componentization of modules, allowing for scalability (Whitney et al., 2015). Supplementary to extensibility, Hishmeh et al. discuss the importance of ensuring the modules developed have a simple interface. This increases readability of the code base, and therefore the likelihood of maintainability and reusability (Hishmeh et al., 2009). Both the Masat-1 and the Galassia CubeSat teams stress the importance of separating the functionality of the payload software from the flight logic. This was mentioned to enable reusability by allowing the majority of the code base to be reused between missions (Latachi et al., 2020). For Askari et al., implementing this level of modularity was key in achieving a program requirement of only needing to develop software for the new payloads being flown between missions (Askari et al., 2017). De Souza et al. were able to achieve reusability through a layered software approach, where only the lower layers close to the hardware need to be changed for deployment on future hardware (de Souza et al., 2022).

Whilst commonly recommending the development of in-house modular, reusable CubeSat software, it is often seen in published work that CubeSat software developers seldom offer advice to new developers on how to achieve this goal. The concepts of modularity, abstraction and layered software design may seem overwhelming to University teams embarking on the development of their first CubeSat, which in turn may increase the likelihood of the team electing to use a mature software framework or an SDK. As previously mentioned, whilst using an existing software solution decreases the time-to-orbit for an initial mission, the ongoing costs and tight coupling of the code base to the solution being used can make further development difficult. Furthermore, the University team may miss out on the potential learning experience of developing their own software framework.

Fortunately, independent bodies exist in an attempt to standardise the development of spacecraft systems, including software development. The European Cooperation for Space Standardisation (ECSS) is one such body. The design practices previously mentioned by the CubeSat software developers are in line with the topics discussed in publications for software engineering and software product assurance, ECSS-E-ST-40C (Space Engineering Software, 2009) and ECSS-Q-ST-80C (Space Product Assurance, 2017). The software engineering and software product assurance documents are grounded in the creation of unambiguous, verifiable and traceable system requirements. The clauses therein encourage the developer to satisfy the produced requirements throughout the mission life cycle. Adherence to the standards promotes the creation of safe, reliable, maintainable code, increasing the likelihood of mission success. Whilst offering a high level overview of the necessary elements for reliable spacecraft software design, the documents however still do not offer in depth advice on how to implement modularity, reliability, re-usability and extensibility in space systems.

From the works discussed, software developed for CubeSats is often based around requirements, or in the case of Masat-1, influenced by fault detection, isolation and recovery (FDIR) routines (Latachi et al., 2020). Learning from incidents (LFI), in particular accident analysis, is a common technique used by the aviation industry to learn and prevent recurrence of mishaps (Clare and Kourousis, 2021). However, it does not seem to be a common driver for CubeSat software development, perhaps as previously mentioned due to little published work on CubeSat failures existing. Yet, often described as a test-bed for larger spacecraft platforms, the development of CubeSat software can be influenced by the accidents that occur in larger spacecraft missions. Therefore, with the proven ability to aid in sustainable developments in system safety, the LFI concept was applied in the development of software for the Binar platform.

Binar-1 (named after the Noongar word for fireball) is Western Australia's first satellite, designed and built by the Space Science and Technology Centre at Curtin University in Perth. At its core, the 1U CubeSat is controlled by a motherboard that is designed entirely in-house, with a focus on subsystem design and integration. The completed core occupies just 0.25U, consisting of primary and redundant flight computers, a GPS, attitude determination and control system (ADCS), electrical power system (EPS) and memory storage, leaving significantly more room for payload than most similar platforms. Having this capability in-house enables prototype testing to destruction to stress the limits of the spacecraft, and facilitates hardware design reuse between missions. In addition to the hardware development, the Binar Space Program has developed the software stack powering Binar-1.

The custom software consists of flight application code, a hardware abstraction layer, operating system abstraction layer, utilities, system resource manager, and board support layer for low-level drivers for on-board peripherals. Writing custom software for the motherboard meant that on-board peripherals could be leveraged in the most optimal and efficient manner to function as expected. Being developed in parallel to the iterative hardware design process, software development saw adhesion to abstracted software design that enabled hardware to be changed without requiring much modification to the software. This ubiquitous abstraction has become a core feature of the Binar spacecraft that will allow software to be reused on future missions with minimal changes.

This paper provides an analysis of historical space mission failures due to inadequate software, using the findings to create recommendations for CubeSat software developers on writing reliable software for small spacecraft. As a case study, these recommendations are applied to the development of the Binar Software Framework, with the perceived benefits discussed. Unfortunately, a hardware fault on the adaptor board connecting the Binar CubeSat core to the transceiver on Binar-1 prevented communications between the motherboard and the transceiver. This meant that ground control was only able to communicate directly with the transceiver. Due to this, the on-orbit benefits of adopting the recommendations could not be validated. However, we have seen some benefits of adherence to the methods used in the paper as we are currently using the code base for development of our next three spacecraft. As the program progresses and the spacecraft produced become more complex, it is expected further benefits will become apparent.

Section 2 analyses the missions, finding that the areas of code reuse, software testing and poor code base maintenance are seen to be frequently responsible in mission failures. Following this, Section 3 then discusses how software can be written for small spacecraft to prevent these failure reasons from reoccurring, using Binar-1 as a case study. Where applicable, it mentions how the software aligns with the clauses specified in the ECSS documents. Finally, the benefits of adopting the proposed recommendations are discussed in Section 4 with a focus on safety, reliability and reusability, increasing the likelihood of small spacecraft mission success.

2. Spacecraft Software Failures

Whilst the importance of software on spacecraft is well understood (Nilchiani, 2009), failures are still common in the space industry specifically resulting from mishaps related to software, often due to insufficient testing prior to launch and poor coding practices (Ji et al., 2019) (Leveson, 2012). This trend is also observed in the small spacecraft community (Langer and Bouwmeester, 2016). In a 2017 study, Venturini et al. (Venturini et al., 2018) conducted 23 interviews of organisations with CubeSat industry experience. Respondents to the interviews identified flight software as a high-risk area, repeatedly stressing the importance of robust software design in the reduction of risk. Ideally, this paper would analyse small spacecraft mission failures due to software, however with a lack of definitive publications on this topic it will use examples from larger spacecraft missions instead. The scope of the missions selected for this paper was limited to only those that publicly disclose software to be a critical point of failure from the result of an accident investigation review. The spacecraft failures that met this criterion were the on-orbit collision involving DART in 2005, the potential atmospheric destruction of the Mars Climate Orbiter in 1998, and the self-destruction of Ariane-5 and Zenit-3SL shortly after lift-off in 1996 and 2000, respectively. Code reuse, software testing and code base maintenance were frequently mentioned in the investigation reports for these missions. The overlap of software failures in the missions selected can be seen in Table 1. It is hoped that this paper will serve to raise awareness of common spacecraft software failure issues, and showcase methods to prevent similar failures in the future. This section of this paper will examine the selected missions and discuss the reasoning behind their software failures. Following this, Section 3 will discuss how the Binar Software Framework addresses these common failure areas to produce safe, reusable spacecraft software to promote small satellite mission success.

Table 1: A summary of the cause of spacecraft software failures.

	Failure Reason			
Mission	Testing	Documentation	Code Review	Code Reuse
DART	*		*	
MCO	*	*		*
Ariane-5	*	*		*
Zenit-3SL	*		*	

2.1. DART

The Demonstration for Autonomous Rendezvous Technology (DART) project was a 2005 NASA mission to demonstrate the hardware and software required to perform an automated docking with another spacecraft in orbit (Rumford, 2003). The target spacecraft for docking, MUBLCOM (Multiple Paths, Beyond Line of Sight Communications), remained operational in orbit after having satisfied its primary goal. Eleven hours into the mission, DART collided with MUBLCOM and proceeded to its end of life sequence, failing to achieve any of the fourteen on-orbit mission objectives (Lilley, 2012).

The Mishap Investigation Board (MIB) identified significant software faults that led to mission failure (Croomes, 2006). The first point of failure occurred as a result of an attempt at program error correcting logic. The spacecraft used a standard Kalman filter approach to estimate position and velocity vectors using GPS and inertial data. However, the spacecraft also introduced a software reset when the error between the estimated and measured data became too great. After the reset, the flight computer sampled the GPS and used those readings as its new estimate. This initial reading however was consistently 0.6 m/s inaccurate which would again cause the readings to drift and trigger a software reset, repeating every three minutes. The MIB discovered that the design requirements dictated an algorithm that should have been able to correct a discrepancy up to 2 m/s, well above the error range that forced the spacecraft into a software reset loop. The MIB proposed that the failure to detect the inadequate control algorithm stemmed from lack of testing across the full range as specified by the design requirements. Furthermore, in the calculation of its position and velocity vectors, DART weighted the estimated value far higher than it should have. This was caused by a late change of a gain constant in software that did not undergo sufficient testing. The new gain factor, attributed to changing mission requirements, meant that the algorithm could never actually converge. Ultimately, this caused excessive course correction manoeuvres during the short mission length, leading to the early mission retirement and contributing to the mishap with MUBLCOM. The failed docking and subsequent collision can also be attributed to an overly strict control sequence to initiate the docking manoeuvre. The waypoint that DART was to enter to begin the first on-orbit mission objective was too small, and as a result the spacecraft continued to drift toward MUBLCOM. The algorithm for the docking sequence did not account for the possibility of navigational errors, and hence there was no ability to recover after this waypoint was missed. An algorithm for collision avoidance was built into the spacecraft, however it depended on the same navigational data source that led to the early retirement fault and hence was ineffective at preventing the failure.

2.2. Mars Climate Orbiter

The Mars Climate Orbiter (MCO) was a 1998 NASA mission intended to study the Martian climate from a low circular orbit. The intention was to use an initial elliptical orbit through the upper Martian atmosphere to lower the orbital apses to achieve a low circular orbit (Leveson, 2012). However, after beginning its orbital insertion burn, communication with the spacecraft was lost. The MIB determined that MCOs initial elliptical trajectory passed too deep into Mars's atmosphere, either destroying the spacecraft or causing it to skip out of the atmosphere and into a heliocentric orbit (Stephenson et al., 1999). During the investigation, the failure was traced back to code reused from Mars Global Surveyor (MGS) (launched in 1996). The code in question assisted with the calculation of the small forces required to prevent saturation of the spacecraft's reaction wheels due to the build up of angular momentum. The MGS code required imperial units in its calculation of the impulse bit, which then was to be converted to metric units to be used by the propulsion control routines. However, the reaction control system thruster size on MCO was increased from MGS, requiring changes to the control algorithms. Whilst the equations used for MGS did take into consideration the conversion between pounds per second to Newtons per second, this value was buried in the code with a lack of sufficient documentation. Coupled with a failure to meet interface specification requirements through testing (Edward A. Euler et al., 2001), when the new thruster equation was substituted into the reused software this conversion factor was missed and hence was mistakenly removed. As a result, the software interface specification for MCO received thruster parameters in imperial units rather than the required SI units, which ultimately led to the spacecraft dropping its periapsis to an estimated 57 km (Edward A. Euler et al., 2001).

2.3. Ariane 5

As a successor to Ariane 4, much of the already flight-proven software was reused to accelerate the development of Ariane 5. Unfortunately the first flight in 1996 resulted in a catastrophic failure 37 seconds after lift off when the self-destruct system triggered the destruction of the launch vehicle. The post incident investigation into the failure tracked the cause back to the inertial reference system failing after having encountered a software exception (J. L. Lions, 1996). Incorrect data coming from the inertial reference system instructed the on-board computer to fully deflect the solid booster engines, causing the vehicle to veer off course. The flight computer could not swap to the redundant inertial reference system as it too had suffered the same failure. The software exception on both inertial reference systems was found to have been caused by an integer overflow in the attempt to convert a 64-bit float to a 16-bit signed integer. The conversion software did not handle the case where the float would be too large to fit into the integer. This error was located in the software that aligned the inertial reference system on the launch pad. After lift off, the function, which was designed for Ariane 4, continued for 40 seconds into flight. However, the designers had overlooked the need to update the flight logic to account for the higher acceleration from Ariane 5's more powerful Vulcain engines. The velocity that the vehicle reached early into flight resulted in a larger value than could be converted to the smaller integer. The Ariane 501 Inquiry Board mentioned that had the inertial reference system and complete flight control system been tested extensively, the software errors leading to the failure could have been detected (J. L. Lions, 1996).

2.4. Zenit-3SL

Zenit-3SL was a Ukrainian expendable launch vehicle primarily used to deliver communications satellites into geosynchronous orbits. It was a multinational collaboration through the Sea Launch project that used a mobile equatorial sea-based launch pad for easier access to equatorial orbits (Harvey, 2007). The third mission of the program in 2000 failed shortly after launch when the rocket self-destructed after a significant course deviation (Gorbenko et al., 2012). The failure analysis panel determined that the launch failure could be attributed to a software error prematurely disabling the second stage engine. The root cause was traced to software failing to close a pressurisation system valve, leading to an excess loss in produced thrust and early engine termination (Harvey, 2007). The error was traced to a late change in mission requirements resulting in a software engineer modifying control software. During this modification, the code concerned with valve operation was accidentally deleted (Gorbenko et al., 2012). Pre-launch checks of the software following this modification unfortunately failed to identify the missing command to close the valve (Harvey, 2007).

2.5. Summary of Failure Reasons

The key areas leading to software failure shown from the missions discussed are insufficient software testing, inadequate documentation, lack of code review and code reuse. A summary of how these areas contributed to the failure of the discussed missions can be seen in Table 1, highlighting the overlap that is present. It is evident that the largest contribution can be attributed to lack of software testing, with inadequate documentation, insufficient code review, and unsafe code also causing failures. The remainder of this paper will discuss possible preventative measures for these failures in the development of reliable small spacecraft software using the development for Binar-1 as a case study.

3. Recommendations For Small Spacecraft

3.1. Safe Code Reuse

As the goal of the Binar platform is to facilitate rapid mission concept-to-orbit, hardware and software reusability is of high priority to reduce the development time of future spacecraft. Yet, as can be seen from the failures of Ariane 5 and MCO, code reuse is not without risk. The Binar Software Framework was designed to maximise code reusability whilst minimising risk by focusing more on platform portability rather than direct reusability. Software reusability refers to the development and maintenance of software modules that can be reused between compiled binaries. With a focus on meeting the needs of a broad range of software requirements, code developed with reusability in mind can be large, complex and difficult to maintain (Mooney, 1995). In the cross-platform context, source portability refers to adapting existing source code to a wide range of environments. With a reduction in generalisation, software developed this way can be easier to produce and maintain in future implementations whilst also benefiting from increased reliability (Mooney, 1995).

Developing software in this manner can be achieved by following time-tested software principles, designed to streamline the process of writing good code. Two common collections of principles that are frequently used are GRASP and SOLID. GRASP (General Responsibility Assignment Software Patterns) is a collection of nine software principles, namely controller, creator, indirection, information expert, low coupling, high cohesion, polymorphism, protected variations, and pure fabrication (Larman and Kruchten, 2005). The collection of principles aim to guide the developer into writing robust object oriented software through module responsibility assignment. Similarly, the SOLID principles of object-oriented programming, namely Single Responsibility, Open-Closed, Liskov Substitution, Interface Segregation and Dependency Inversion, aim to increase the maintainability and flexibility of a code base by resolving improper dependencies between software modules (Martin, 2000). Whilst these collections of principles may differ in some aspects, the general ideology is that software written adhering to established principles should be of a higher standard than software written without. In this section, Unified Modelling Language (UML) is used to show applications of each of the SOLID principles to the development of the Binar Software Framework in the first use of SOLID for spacecraft software development. Whilst the GRASP principles do provide a beneficial reference on robust software design, the scope of this section is constrained to only the methodologies used during the development of the Binar Software Framework. For reference, the UML class diagram arrows used in this paper are defined in Table 2. The observed benefits are discussed in Section 4.

Table 2: The UML class diagram arrows used in this paper with their meanings.

Arrow	Meaning	Description
——→	Association	Child object exists within and is used by the parent
- - - -→	Dependency	Changes to the parent object will affect those that depend on it
——▷	Inheritance	Child object inherits (and can extend) the functionality of a parent
- - - -▷	Implementation	Child object implements the methods defined in the abstract parent
——◇	Aggregation	Objects are associated, but the child can exist without the parent object

3.1.1. Single Responsibility Principle

The single responsibility principle is concerned with the division of program logic amongst multiple objects encapsulating a single small responsibility each (Martin, 2014). The concept promotes using multiple objects with well-defined interfaces over using a singular object to control multiple aspects of an application. As a general rule, an object conforming to this specification is said to have a single responsibility if the object only has a singular reason to change. An example of this in the Binar Software Framework can be seen in the analogue to digital conversion software module (Figure 1) where the current and temperature measuring objects are implemented as independent classes instead of being amalgamated into one complex monolithic object responsible with measuring both.

In our approach, a base class accounting all features required by the analogue objects was implemented from an interface, and then separate specific analog classes inherited and extended this functionality. This prevented the

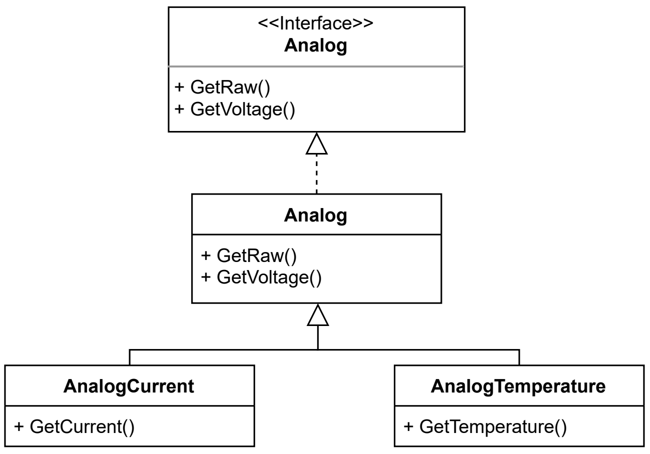


Figure 1: The application of the Single Responsibility Principle to the ADC module in the Binar Software Framework. The AnalogCurrent and AnalogTemperature modules are decoupled from each other by implementing their required functionality within the scope of their respective objects. A change to either of the AnalogCurrent or AnalogTemperature objects will not result in a change in the other.

non-cohesive current and temperature objects from being coupled together causing unnecessary rigidity in the module design. This is important in the context of code reuse as the prevention of coupling means that changes to subtypes of the Analog module will not invoke a change in other modules.

3.1.2. Open-Closed Principle

The Open-Closed principle (Martin, 2014) refers to the development of software modules that are open for extension, but closed for modification. The functionality of modules under this principle can be extended upon when the requirements of an application change, but the functionality that has already been implemented cannot be modified. The principle recommends that changing project requirements should be addressed by adding new code rather than by rewriting code that already works. This is commonly implemented through the use of abstract objects in object-oriented programming design (Martin, 2014). An example of the Open-Closed principle in the Binar Software Framework can be seen in the GPS library (Figure 2), where the design of the GPS module includes a base class with a set of concrete public methods which describe functionality in terms of the abstract method *ReadGps*. The abstract object *GpsBase* gets implemented in the inheriting module, GPS, thus extending the functionality of the module without modifying the source code.
The Open-Closed principle is helpful in this context because it explicitly highlights that not all code can be safely reused, favouring the development of modules that separate generic functionality from the detailed implementation. As GPS devices output data through a series of strings with a checksum, this functionality can be developed independent of the GPS, decoupling the module from the specific hardware chosen.

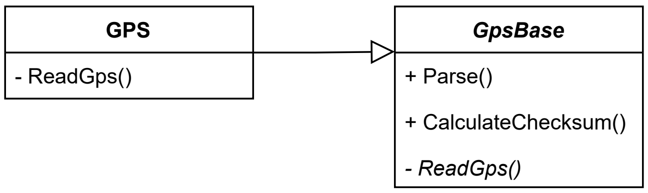


Figure 2: The application of the Open-Closed Principle to the GPS module in the Binar Software Framework. The functionality of the GpsBase class is extended through the addition of new code in the inheriting GPS object, without the need for modification of the original base class.

3.1.3. Liskov Substitution Principle

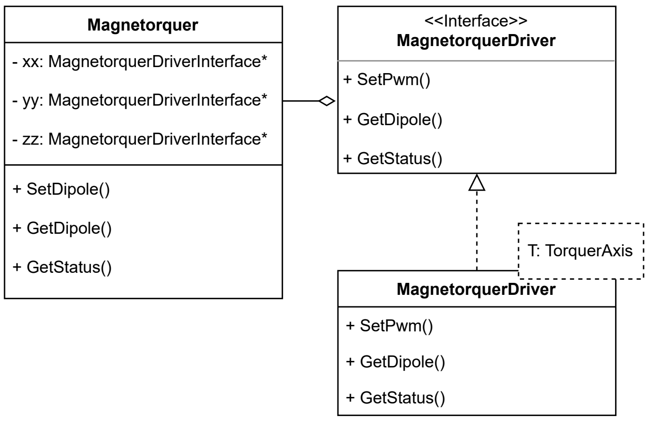


Figure 3: The application of the Liskov Substitution Principle to the magnetorquer module in the Binar Software Framework. The Magnetorquer object has three private pointers to axes that implement the MagnetorquerDriver interface object. Any of the template concretions of the MagnetorquerDriver class are eligible for substitution into these private pointers as they inherit from the MagnetorquerDriver interface object

The Liskov Substitution principle proposes that subtypes of modules must be substitutable for their base types in application code (Liskov, 1987). Analogous with run-time polymorphism, it is one of the enabling factors that makes the Open-Closed principle discussed in Section 3.1.2 possible. The principle relates to code reusability through increasing portability by decoupling generic module software from the detailed implementation. Furthermore, this aids in future maintainability by providing a clear contract with the application code on how to use a specific module, rather than being concerned with how it is actually implemented. An application of the Liskov Substitution principle in the Binar Software Framework can be seen in the magnetorquers software module (Figure 3), where a templated MagnetorquerDriver object can be used by the Magnetorquer class as it inherits from the MagnetorquerDriver interface object.

In this module, the publicly visible magnetorquer object controls the MagnetorquerDriver objects through references to their abstract base class. The concrete MagnetorquerDriver subtype is templated on all available magnetorquer axes, making the addition of redundant axes compatible with the public module without modification. This is important as it reduces the coupling between objects in a module. Implementing functionality through pointers to abstract interfaces allows the higher level software to use the lower level objects without having to be concerned how they operate, thus permitting changes to interactions with spacecraft hardware without forcing a recompile of the full module.

3.1.4. Interface Segregation Principle

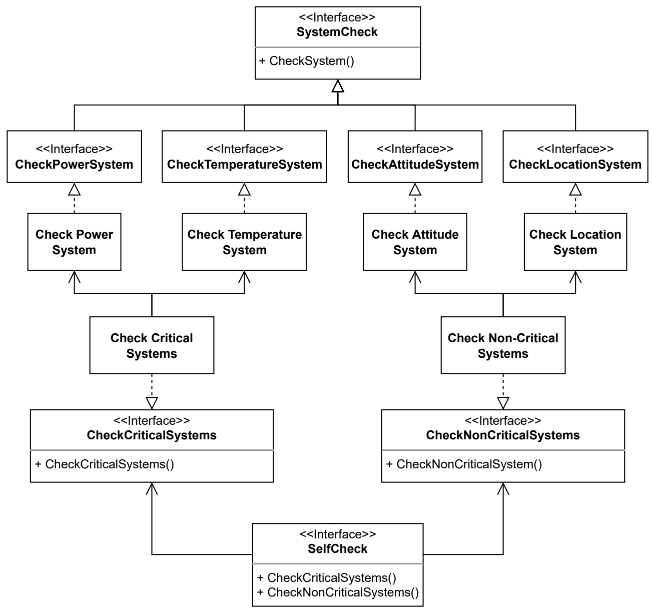


Figure 4: An application of the Interface Segregation Principle to the self check module in the Binar Software Framework. In this module, each subsystem that can be checked on-board the spacecraft implements a unique, decoupled interface. The critical and non-critical system check implementation objects then associate the implementations of the unique interfaces. This prevents a check done with the critical system object from becoming coupled to a non-critical system, allowing the systems being checked to be extended upon without forcing a recompile of unrelated modules.

The interface segregation principle states that software module subtypes should not be forced to depend on modules they do not use (Martin, 2014). To achieve this, it recommends avoiding monolithic interfaces that contain many unrelated methods. Rather, the principle favours multiple smaller interfaces that get implemented where needed. This principle is closely related with the maintainability aspect of code reuse by removing code duplication as much as possible, whilst avoiding the pitfalls of non-cohesive software interfaces. An example of this principle implemented in the Binar Software Framework can be seen in Figure 4, where subsystems that are checked in the self test module are isolated from one another, and are implemented where required.

The alternative, more rigid approach to implementing the self check module would have been to have a singular self check interface that the subtypes of SystemCheck interface all depended on. Whilst simple to implement, this pattern would result in methods being added in the standard self check interface that would have only been used exclusively by specific subtypes. This would then couple all the clients of this interface together, and open up the possibility for a module to be affected by changes that other clients force upon the class. The resulting pollution of the interface could potentially lead to errors when code is reused. Due to the likelihood of on-board systems expanding as mission requirements become more complex, the interface segregation principle allows for these potential changes by decoupling objects from non-cohesive interfaces and therefore allows subtypes to not be forced to depend on methods they don't need to. Moreover, implementing this principle aids in high testability of software modules due to the granularity achieved in the responsibilities of the interfaces. In the example seen in Figure 4, rather than having the system classes that inherit from the abstract interface depend directly on one monolithic SelfCheck interface object, each subclass of the abstract class is given its own interface to depend on. Self check objects can then implement the individual interfaces that they will actually make use of for checking, rather than all systems on board. For testing the self check module on the host, the interfaces responsible for interacting with the hardware are able to be mocked and substituted into the module under test through polymorphic dependency injection.

3.1.5. Dependency Inversion Principle

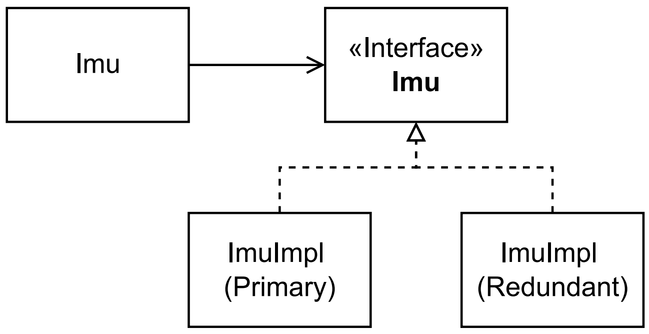


Figure 5: An example of dependency inversion in the IMU software module. Application code wanting to access the IMU hardware will use the high-level object that depends on the same abstraction as the lower level implementations. This decouples the high-level IMU object from the hardware it is interacting with, allowing the IMU object to be reused even if hardware requirements change.

Rather than the traditional application architecture approach of a direct dependency chain between high-level software and low-level implementations, the dependency inversion principle states that high-level software and low-level software are to both depend on abstractions (Martin, 2014). An example of this can be seen in Figure 5, where the high-level public facing IMU object and the lower level private ImuImpl objects both depend on the abstracted IMU interface.
If the application code is interested in interacting with the IMU, it does so through a high-level object that implements the same IMU interface used by the implementation objects, rather than depending on the implementations directly. The

software using the peripheral has then become abstracted from specifically how it receives the IMU data and instead focuses on relevant details that are independent of the platform. This principle is important for code reusability as it creates a modular interface where low-level implementations can be changed without affecting high-level code that interacts with them. As Binar-1 has a primary and a redundant flight computer with different architectures on board, they will need different software binaries to operate. Moreover, considering the redundant flight computer is to take over from the primary flight computer in the event of an emergency, it will need to be able to operate with the same peripherals. It would be time-consuming and error-prone to rewrite the software concerned with peripheral interaction simply because it is to be run on a different microcontroller. The application of the dependency inversion principle means that the high-level flight software can be reused between the computers, with the implementation simply swapped out.

The Binar Software Framework further leverages the Dependency Inversion Principle by making use of compile-time configurable implementation linking. Using custom command-line arguments, the compiler will link generalised high-level objects with their computer specific implementation to build binaries for the two flight computers on-board the Binar bus. As the hardware for Binar develops in the future, the application of this principle means that new drivers can be implemented without any risk of breaking high-level flight logic.

3.1.6. Failure Prevention

The usage of SOLID principles can help reduce the likelihood of errors appearing through the reuse of code. In the case of the Mars Climate Orbiter discussed in Section 2.2, when the small forces software from the Mars Global Surveyor mission was reused, it was tightly coupled to the thruster control equation being used. This was demonstrated by the fact that the software had to be modified, and also that the modification of the software broke its functionality. If it were possible to separate and encapsulate the section of the software concerned with calculating the thruster parameters, leaving the genuinely transferable code agnostic of the thruster being used to interact with a well-defined interface, then the design team could have possibly written a plugin for the new thruster to conform to the interface specification. Inserted into the small forces software in a modular approach, this could ensure successful implementation portability by removing the ambiguity between units by providing a documented interface specifying the inputs and outputs of the object.

Similarly, with the Ariane 5 rocket discussed in Section 2.3 care was not taken when reusing code from Ariane 4 to ensure that a narrowing conversion from a large data type would fit within the bounds of a smaller one used in application code. If the module concerned with using the inertial reference data was decoupled from reading it, instead interacting with an interface, then the remaining portable high-level code could have been transferred without the application-specific data conversion. Engineers could then write a new module for the low-level software to interact with the different hardware whilst conforming to the interface for it and have the high-level software interact with it seamlessly.

Being able to reuse code safely involves designing software to be understandable, flexible and maintainable. Adhering to the SOLID principles of object-oriented program design, the modules written for the Binar Software Framework ensure that a developer can safely reuse code on future missions. Modules having only a single responsibility mean they are decoupled from each other and won't be effected by changes to other modules. Changes are expected, mostly due to changing hardware on the spacecraft, and so high-level libraries controlling low-level drivers polymorphically allow for new drivers to be written that use the same interface. If the hardware requires a more feature-rich driver, the interface can be extended whilst still satisfying the polymorphic requirements of the high-level controlling object. When these drivers are added to support new hardware, new targets can be added to build files in the code base to selectively link the software framework against revisions at compile time. Moreover, drivers that do not rely on hardware can also be linked at compile time for testing the software independent of a spacecraft engineering model.

The usage of the SOLID programming principles contributes greatly in adherence to the clauses outlined in ECSS-Q-ST-80C. The standard mentions that the development of software should apply design methods that have performed successfully in a similar application to ensure dependability and safety (Space Product Assurance, 2017). The large volume of published work regarding SOLID proves the successfulness of the time-tested guidelines, and has even been observed in the scientific community to increase maintainability of a code base, whilst decreasing dependencies and complexities (Källén et al., 2014). The ECSS-Q-ST-80C document also specifies that the development of software with the intention of future reuse is to be implemented with a minimum dependency on hardware. The modularity achieved from adherence to the SOLID principles aids in meeting this standard, promoting safe reuse on future missions.

3.2. Software Testing

3.2.1. Decoupling as a Tool for Testing

A commonly reported reason for spacecraft failures in the missions discussed in Section 2 was the failure to adequately test modules of the software, to the extent that the Ariane-5 investigation board mentioned explicitly that testing could have identified fatal errors during development (J. L. Lions, 1996). From a CubeSat perspective, the survey conducted by Venturini et al. (Venturini et al., 2018) saw an emphasis on the importance of software testing in the success of a mission. The failure to test is a common challenge faced in embedded software development due to hardware dependency. In these situations, the lack of ability to test is due to the tight coupling between high-level software and the low-level software that is concerned with interactions between embedded subsystems. Furthermore, barriers to testing that arise are related to commonly only having a single spacecraft engineering model to test on and the commonality for hardware revision. This scarcity of testing hardware can introduce issues during the testing process where software may cease to work on future hardware revisions (Garousi et al., 2018).

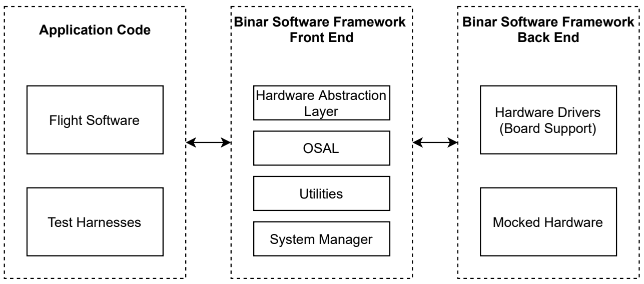

Application Code
Flight Software
Test Harnesses
Binar Software Framework Front End
Hardware Abstraction Layer
OSAL
Utilities
System Manager
Binar Software Framework Back End
Hardware Drivers (Board Support)
Mocked Hardware

Figure 6: Testability of the Binar Software Framework is increased through abstracting away low-level hardware dependencies.

Decoupling flight application software from the embedded hardware on the spacecraft allows testing on various satellite models, both physical and virtual. Tipaldi et al. (Tipaldi et al., 2018) discuss the benefits of using black box testing verification in a high-level test procedure language to verify the flight software over the lifetime of Meteosat Third Generation. Taking advantage of the module decoupling that is achieved through the implementation of the SOLID principles discussed in Section 3.1 however, the white box testability of the Binar code base can be increased. In the Binar Software Framework, testing of libraries that interact with low-level hardware through an interface layer is done by polymorphically substituting mocked objects in place of the modules that require a hardware dependency, leveraging the Liskov Substitution Principle. Favouring many small interfaces over one monolithic interface, as recommended by the Interface Segregation Principle, adds granularity to the code base and hence only those objects concerned with hardware interaction need to be mocked when testing on the host. Tests can then be written based on how the libraries are to be used in application code, enabling testing of the software framework without a hardware dependency as can be seen in Figure 6. Having fine granularity in system interfaces also allows for fault injection into the module under test through faked implementations of interfaces, adhering to the software testing standards outlined in ECSS-E-ST-40C (Space Engineering Software, 2009). This allowed the software to be written in parallel with the hardware development of Binar-1, prior to performing integration testing. Furthermore, this module decoupling opens the possibility for extending upon the functionality of the software for future missions without breaking old tests. Moreover, polymorphically substituting mocked objects in place of hardware dependencies enabled libraries developed for the Binar Software Framework to adhere to test-driven development to meet the desired test coverage.

3.2.2. Test-Driven Development

Test-driven development refers to a method of developing software in which requirements of software are converted to test cases as a precursor to the actual development of the software itself. The process involves firstly writing a test that won't succeed, followed by writing just enough code to make the test pass, and then lastly refactoring to remove duplication in the software written to pass the tests (Beck, 2003). Adhesion to this method of development aids in meeting the software testing standards outlined in ECSS-E-ST-40C, which highlights that traceability between requirements and all software units is essential in order to produce meaningful test results (Space Engineering Software, 2009). Aside from having 100% test coverage, software developed through this method will be as lean and simple as possible. According to ECSS-Q-ST-80C, software can be assured to be dependable and safe through 100% code coverage at the unit testing level. Moreover, it is mentioned that this metric is critical during the feasibility review of reusing the software in the future (Space Product Assurance, 2017). The creation of the Binar Software Framework saw strict adherence to this design method during development. The benefits of using the TDD approach in the development of the software framework include the ease of performing positive and negative testing. These types of testing, enabled through the conversion of software requirements directly into test cases, ensure that the libraries developed will safely handle both correct and erroneous data to guarantee conformity of on-board software to system requirements. Positive testing ensures that the software is able to meet the requirements specified, and negative testing ensures stability through proving the software will behave as expected under the worst case scenarios, as recommended in ECSS-E-ST-40C (Space Engineering Software, 2009). High testability of libraries was assisted through propagation of errors from the low-level software to higher level code, providing transparency of the inner workings of a software module. This is important as the calling code generally has more information on how the error should be handled than what may be available in the scope of the library. This increases overall code portability and facilitates library reuse for future spacecraft.

An example of this in the Binar Software Framework can be seen from the development of the communications library. The operational requirements were determined prior to writing the library, outlining how the spacecraft should handle telecommands. Converting these operational requirements into test cases, negative testing was heavily employed to ensure safe execution pathways when receiving telecommands that either can't be decoded or were received in the wrong operational mode. Likewise, positive testing asserted that responses to correct telecommands were as expected. The test cases that enabled the development of libraries in the Binar Software Framework are executed frequently, ensuring that system requirements are always met.

3.2.3. Failure Prevention

One of the failure points for the DART mission discussed in Section 2.1 was an inadequate control algorithm that failed to correct for a velocity error of up to 2 m/s, even though this was specified as a design requirement. Considering the TDD methodology starts with the translation of software requirements into test cases, mission-critical requirements are factored into the software early, and any further development that could break this requirement would be highlighted when the test for the requirement begins to fail. Learning from this mishap, all requirements for the libraries written for the Binar Software Framework were defined and translated to test cases before the libraries themselves were written to prevent software requirements being missed. Another area of failure for the DART mission occurred when an engineer changed a gain constant that resulted in an iterated calculation never converging. This error demonstrates the importance of continuous testing not just during library development, but consistently during the implementation of the library to ensure changes made to the software do not go out of sync with the tests written for it. From the adoption of TDD for the software framework enabling Binar-1, a change such as this would cause test cases to begin to fail, highlighting that the library is not safe for release.

It was evident through writing test cases for the Binar Software Framework that it became easy to test how the libraries would handle erroneous inputs, rather than relying on the assumption that libraries would always perform as expected. This was critical as the MIB mentioned during the DART mishap investigation that during pre-flight testing a software model assumed the GPS measured velocity perfectly (Croomes, 2006). Having the ability to stress-test all libraries used on Binar-1 provides further reassurance in the success of the mission.

Software testing also aids in the adherence to ECSS-E-ST-40C regarding the reuse of code. It is mentioned that for code to be reused, it needs to adhere to the functional requirement baseline of the project (Space Engineering Software, 2009). As TDD involves the conversion of software requirements to test cases, the reused software will need to verifiably adhere to the functional requirements to be included in the code base.

During the mishap investigation for Ariane 5 discussed in Section 2.3, the report mentioned that had the software for the inertial reference system and flight control system been tested extensively, the software errors leading to the failure could have been detected. By developing the libraries through the TDD approach, 100% test coverage is possible.

3.3. Optimal Code Base Maintenance

3.3.1. Documentation

In a 2012 review of the involvement of software in spacecraft accidents, Leveson (Leveson, 2012) acknowledges that inadequate documentation is a factor in spacecraft mission failures. In the missions discussed, this is particularly relevant to the MCO and Ariane 5 mission failures which recommend improvements to code base documentation in their failure analysis reports. However, from a survey of 48 candidates with experience in the software engineering industry, Forward et al. (Forward and Lethbridge, 2002) found that whilst participants thought positively of the usefulness of software documentation, it was seen to be seldom updated. To prevent issues arising as a result of poor software documentation, whilst acknowledging the difficulties of documentation maintenance, documentation generation is employed on the Binar platform to maintain up-to-date documentation. Having access to generated documentation pages can be very useful for future developers and maintainers to understand how to maintain and use the numerous libraries that comprise the Binar Software Framework. To fully utilise this tool, Binar developers are encouraged to use increased verbosity in variable and function names. Non-cryptic names are desired over their shorthand counterparts to reduce ambiguity and promote self-documenting code. To ensure that code generation is invoked often, it has been integrated into the Binar software development pipeline to automatically build when a new commit is pushed.

3.3.2. From Local Machine To Spacecraft

The code base for the Binar Software Framework is hosted on a remote version control repository that is always guaranteed to have the latest stable release of the software. This is done through the implementation of Continuous Integration and Continuous Deployment (CI/CD) online. Each developer has their own private fork of the upstream repository to continue development in an isolated environment from other developers doing the same. When a developer pushes software changes to their fork and creates a merge request into the upstream repository, it executes the CI/CD platform remotely. It then notifies an assigned team member to start the code review process. The CI/CD platform automates the building and testing of all software libraries written in the repository. If any library fails to build or complete its tests, the merge request will be automatically blocked from merging into the upstream branch until these issues are resolved. This prevents potentially buggy or broken software from merging into the upstream repository, which is used for code deployment onto the spacecraft. The results of the test execution include the code coverage report, which is generated into an access-controlled web page providing a line-by-line breakdown of the coverage within the source code. This provides a quantifiable metric of test coverage, and assists the developer on adding tests to increase the coverage.

During pipeline execution, a team member is assigned to the merge request to conduct a code review by visually inspecting and assessing all changes in accordance with the design rules imposed by the team, and to identify any accidental incorrect program logic. Using an online version control platform for this, discussion threads can be added to specific sections of the software changes to clarify decisions made during development, to recommend further changes be made, and also provide a history of important software changes. Complementary to this, the CI/CD platform also runs a static analysis on all libraries in the code base, generating an access-controlled web page containing the analysis report to be referenced during code review.

Upon completion of the review process, the team member assigned to review the merge request is given the option to merge the changes into the upstream repository only if the CI/CD pipeline succeeds. This prevents any software that is written by a singular engineer from getting deployed to the spacecraft without explicit approval from at least one other team member. Once successfully merged, the last stage of the pipeline is executed to generate a third access-controlled web page containing the documentation scraped from the source code. A summary of the process from code development to deployment on the spacecraft can be seen in Figure 7.

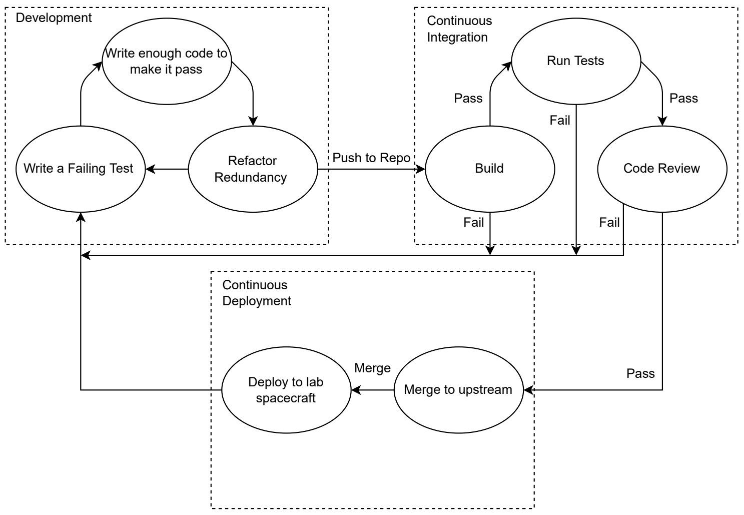


Figure 7: The Binar Software Framework development pipeline.

3.3.3. Failure Prevention

The importance of documentation being stressed early in the development of the Binar Software Framework was partly influenced by the mishap discussed in Section 2.2, where the Mars Climate Orbiter was lost during its orbital insertion manoeuvre. The mishap investigation report mentions that the equation used to calculate the small forces lacked sufficient documentation of the built-in conversion factor, hence being accidentally deleted by the developers reusing the software from MGS. Thorough documentation practices could potentially have identified the equation used in the software to have included the conversion factor, notifying the design team to adjust it accordingly. Also contributing to the mishap leading to the loss of Ariane 5, software documentation could be a solution to the investigation board's recommendation to verify the range of values taken by a variable in software. This has been achieved in the Binar Software Framework through extensive documentation of variables which are scrutinised during the code review process. The necessity of software documentation is stressed in ECSS-E-ST-40C, which insists that software documentation is to be constantly updated throughout the mission lifecycle (Space Engineering Software, 2009). Having software documentation automatically generated with every commit aids in meeting this standard, and ensures that errors related to insufficient documentation will not propagate into the spacecraft applications.

The push for implementing a pipeline getting software written on a local machine to the spacecraft stemmed from the mishap seen on the DART and Zenit-3SL missions discussed in Section 2, where a late change to control software led to the failure of the spacecraft. The ECSS-Q-ST-80C document mentions that software that is modified is to undergo regression testing (Space Product Assurance, 2017). If this standard was adhered to, it is expected that a change to the gain constant on these missions would trigger old tests to fail. Incorporating a CI/CD platform that runs tests online prior to allowing an update to be merged could potentially have identified these changes as faulty software, rejecting the commit and hence preventing the change propagating to release code. In the situation where tests seem to be passing on a surface level but in actuality lack sufficient depth, the generation of a code coverage report would notify the reviewer that the module is insufficient for inclusion into the upstream repository, and to hence reject the merge.

One of the recommendations by the investigation board following the failure of Ariane-5 mentions the importance of reviewing all software (J. L. Lions, 1996). This was a strong influence for the code review and static analysis stage of the development pipeline imposed by the Binar software team, in which merges are rejected by a reviewer if the documentation is deemed insufficient, or software bugs are identified. The ECSS-Q-ST-80C document recommends that the evaluation of code is to be performed in parallel with the development process, aiding in the conformity to good coding standards and the reduction of software errors (Space Product Assurance, 2017). The code review stage of the pipeline aids in meeting this standard, preventing ambiguous and potentially buggy software from reaching release. Moreover, it is recommended that software is to be placed under configuration control after passing all unit tests successfully. Requiring the software being merged into the main repository to pass all unit tests aligns the developed software with this standard, providing assurance that the software propagating onto the spacecraft is dependable and safe.

Adopting a strict approach to code base maintenance can ensure code clarity, maintainability, portability and safety. A strong documentation standard can ensure a lack of ambiguity on how to implement software modules in application code, especially when internal software is reused, increasing the possible longevity of the framework for future Binar spacecraft. These attitudes toward software design can prevent mishaps associated with individual engineers altering software from occurring.

4. Benefits on Binar-1

Implementing the recommendations for small spacecraft software design on the Binar Software Framework has aided in the development of safe, reliable and reusable software.

Leveraging the SOLID principles of object-oriented program design, the software developed for the Binar bus is strongly positioned to be portable between hardware revisions. The example given for the application of the Single Responsibility principle on the Binar Software Framework was in regard to the ADC module. For future development on Binar spacecraft in the context of this module, this principle supports the intention of using a higher precision ADC reading on certain channels, requiring external ADC hardware. Interfacing with the external hardware will require a dependency on a library that will only be used for the higher precision subset of ADC measurements. Having a separate module concerned with the subset of ADC measurements prevents linking against unnecessary libraries that won’t be needed for other ADC measurements.

The effectiveness of the Open-Closed principle for code reuse has already been demonstrated in the development of Binar-2. Due to the manufacturer deprecating the GPS used on Binar-1, the hardware needed to be swapped out in favour of their new module with a different pinout. If the GPS software module had been developed with a tight coupling to the hardware, any dependencies of this module would suffer from a cascading need to change the GPS object to a new implementation. Aside from slowing down development time and making code reuse difficult, this could also introduce unforeseen errors into the code base. The implementation of the Open-Closed principle supports code reuse by simply extending the already working GPS base object by implementing a new subtype for it.

The current version of the Binar hardware on-orbit has drivers for redundant magnetorquers on the x and y axes. However, the corresponding coils were not implemented in Binar-1. For the development of Binar-2, these redundant drivers are intended to be used. From the adoption of the Liskov Substitution principle, support for these drivers can be added to the software module by adding extra axes to the templated magnetorquer driver object discussed in Section 3.1.3. The Binar Software Framework runs system checks on all peripherals during the spacecraft boot sequence. If the check determines a magnetorquer driver is faulty, the constructor for the publicly visible magnetorquer object can automatically switch to the redundant templated driver on-orbit, passing in a pointer to the driver polymorphically. This means the high-level magnetorquer module in application code can actuate agnostic of the actual magnetorquers being used as this is not necessary information to couple to the driver. This enables code reusability as it allows for any number of redundant coils to be controlled without modification to the high-level magnetorquer module, which can therefore be fearlessly depended on in application code.

As the primary purpose of Binar-1 was to be a technical demonstration of the hardware and software produced by the Binar Space Program, the software developed for system checking was constrained to systems embedded on the Binar bus. For future Binar missions, as the payloads flown become more complex, extending the self check module to support payload integrity checking is of high priority. From the adoption of the Interface Segregation principle, a corresponding base class for checking payloads simply needs to be added similarly to the systems shown in Figure 4. This payload system can then be implemented in a new systems interface class that will also implement checking the power system. This demonstrates the creation of cohesive groups of system checking objects within the module, showing how the library can be extended with future systems that don't depend on modules they won't use.

Through the application of the Dependency Inversion principle (Section 3.1.5), it was seen that much of the software developed for the Binar Software Framework has gone towards high-level libraries to be depended on in application code that are decoupled from the low-level hardware implementations. This currently allows application code to be cross-compiled between the differing architectures of the flight computers on board. As the Binar Space Program develops, a need has been highlighted for flight application code to be run natively on a desktop computer. This could allow for development of payload software without needing a Binar bus, and enable highly accurate simulations of on-orbit operations.

From strict adherence to the test-driven development methodology, the software running on board Binar-1 can be guaranteed to meet mission requirements. Moreover, due to the simplicity of setting up test cases, the libraries can be stress tested for the worst case scenario to increase reliability. As an extension to the portability achieved through the application of the SOLID principles, a large part of the software developed is decoupled from hardware and is hence easily testable. This aligns with the section regarding code reuse in ECSS-Q-ST-80C, which specifies that code to be reused should be assessed against requirements traceability, testing, and coverage (Space Product Assurance, 2017). The hardware decoupling achieved also allowed for software to be developed in parallel to the hardware revisions. Whilst this aided in preventing some bugs from occurring, it should be noted that testing on the hardware is also encouraged when possible. Testing on the host enables the behaviour of the modules to be verified, however issues may arise when running a hardware-in-the-loop test due to the disparities that exist between the platforms the software is running on.

The biggest issues encountered during hardware integration for Binar-1 were related to data structure alignment, power domain access from microcontroller internal peripherals, and timing issues. These issues were not identified during unit testing as they were specifically hardware related. Coupling unit testing on the host with continuous deployment onto hardware can help with bug prevention internal to software modules, and identify any runtime issues that may occur.

Adopting good code base maintenance skills from the beginning of the Binar Space Program has led to a code base which is well documented, readable and maintainable. The benefits of this have been seen from the Binar Space Program team growing since its conception. As new team members begin to contribute to the code base, the plentiful documentation provides clarity on how to implement the many custom libraries developed for the framework. This decreases the time required to train new team members, and gets them contributing to the code base quicker. Clarity of software already in the code base has allowed for easy maintenance as hardware revisions are released and capabilities grow.

Enforcing a local machine to spacecraft cycle consisting of continuous integration and continuous deployment using cloud-based test running and code review means that no unexpected software will make it to space. Enforcing the use of testing over the full life cycle of the mission means that late changes made to the spacecraft libraries can be guaranteed not to break functionality.

5. Conclusion

With the complexity of software increasing in spacecraft missions, reliable software development is critical in the prevention of mission failure. The main contribution of this paper is demonstrating techniques for writing safe, reliable and maintainable software for small satellites through showcasing methods to prevent recurrence of common spacecraft software faults. As such, it is intended for small spacecraft developers planning on writing their own software who may not have prior experience. Where applicable, adherence to ECSS standards has been demonstrated. The missions showcasing common software failures were selected due to their transparency in disclosing software faults as reasons for mission failure. Being directed at small spacecraft software developers, this paper would have benefited from a discussion of CubeSat software faults. However, due to the lack of publications in this area, this could not be included. As software is shown in this paper to be a contributing factor in spacecraft mission mishaps, it is the hope that publications are more prevalent in the future relating to this topic. Although the benefits from recommendations requiring on-orbit validation were not able to be demonstrated in this paper due to a hardware fault on Binar-1, The Binar Space Program has already noticed benefits specifically regarding code base reuse as we progress into building our next satellites. As the program matures and we continue to develop the framework using the recommendations discussed in this paper, we expect further benefits to become more apparent.

Learning from the overlapping mishaps of previous mission failures, the Binar Software Framework has been developed to maximise mission success. The areas of code reuse, software testing and code base maintenance were addressed with examples of each from the core functionality of the Binar Software Framework. Safe code reuse was highlighted to be more complex than simple reimplementation, but was rather discussed in the context of developing software from the ground up with portability in mind. This was achieved in the Binar Software Framework through adherence to the SOLID design principles. Through these principles, the program goal of rapid mission concept-to-orbit can be achieved by reusing the platform independent hardware abstraction layer of the Binar Software Framework between missions. Although the SOLID principles were adhered to during development of the Binar Software Framework, other similar principle collections exist, such as GRASP. Regardless of the principles chosen for development, software written in accordance with time-tested principles will be of a higher standard than software written without.

Lack of software testing was seen to be the most common reason for failure in the missions discussed. It was demonstrated that developing software with decoupled interfaces aids with mocking modules to be substituted polymorphically for hardware implementations to break hardware dependencies and open up the possibility for test-driven development. To alleviate failures discussed in late changes to control software causing mission failures, the importance of tests written during the TDD process are stressed not only for use during development, but also during the full life cycle of the mission to highlight when software begins to break. Through the use of a CI/CD platform, when a developer made changes to software, all unit tests would be run remotely before software would be given the option for a merge request. Leveraging the pipeline to generate code coverage and static analysis reports assists the developer assigned to review the merge request, ensuring that software authorised to merge into the master branch has accountability. This was partly for the prevention of poorly designed code being added to the code base.

Code review also enforced strict adherence to writing documentation. This documentation, both through meaningful, clear and consistent variable names along with commenting, was done to ensure that future developers would be able to easily read, understand and maintain the code base. After giving approval to merge software upstream, the only software that was could be deployed onto Binar-1 was pulled from the upstream master repository. This was to ensure that the software to be deployed onto the spacecraft was free of any errors, and gave a clear version history of software that was released. Binar-1 was selected as a case study for this paper as the software framework was being written for the Binar Space Program. However, the same concepts can be applied to any small spacecraft to increase the likelihood of mission success, and promote the reusability of the code base for future missions.

6. Future Work

The mishaps discussed in this paper are a small subset of a large body of reference. It was observed from the majority of published failures that a common theme was lack of understanding how the mission would respond under the worst case scenarios. From the missions discussed, the Binar Software Framework has been built in a robust manner to prevent critical system errors from jeopardising the mission. However, it still does not allow for in-depth testing of the relationships between the spacecraft subsystems during large-scale failures. Taking advantage of the code base modularity achieved through the application of the SOLID principles, driver backends can be written in the Binar Software Framework to break dependency on hardware. This means that both offline spacecraft simulations and hardware-in-the-loop simulations are possible with minimal development effort to increase the quality of the software design and in-depth testability. This enables adherence to elements of software testing specified in ECSS-E-ST-40C, which describes the importance of demonstrating that the developed software can perform in a representative operational environment (Space Engineering Software, 2009). Therefore, moving forward the Binar team will build a low-level bounded simulation implementation of spacecraft subsystems for offline simulations. The backends for peripherals that give the spacecraft knowledge about its environment, such as the GPS, can be substituted with software that communicates with a simulation running on a host machine. This will allow the high level flight logic to remain unchanged, yet gain a deeper insight on how it would respond to a simulated space environment. A. da Silva et al. discuss the benefits of using fault injection into simulations of software developed for the Solar Orbiter mission, showing a strong improvement in software fault tolerance (da Silva et al., 2014). Applying a similar method to future Binar spacecraft will allow the satellite software to be tested on a much deeper level, providing earlier flight software verification. Developing the capability to replay researched failure scenarios into the spacecraft flight software will provide a deeper understanding of the causes for failure, and more importantly how to recover from them.

7. Acknowledgements

This research was supported by an Australian Government Research Training Program (RTP) Scholarship and received institutional support from Curtin University.

References

Alanazi, A. and Straub, J. (2019): Engineering Methodology for STUDENT-DRIVEN CubeSats, *Aerospace*, vol. 6(5), p. 54, number: 5 Publisher: Multidisciplinary Digital Publishing Institute.

Askari, H.A., Eugene, E.W.H., Nikicio, A.N. et al. (2017): Software Development for Galassia CubeSat - Design, Implementation and In-Orbit Validation, p. 9.

Beck, K. (2003): *Test-Driven DEVELOPMENT: By Example*, Addison-Wesley signature series, Addison-Wesley.

Bocchino, R., Canham, T., Watney, G. et al. (2018): F PRIME: An OPEN-SOURCE Framework for SMALL-SCALE Flight Software Systems, *Small Satellite Conference*.

Carrara, V., Januzi, R.B., Makita, D.H. et al. (2017): The ITASAT CubeSat Development and Design, *Journal of Aerospace Technology and Management*, vol. 9(2), pp. 147–156.

Clare, J. and Kourousis, K.I. (2021): Learning From INCIDENTS: A Qualitative Study in the Continuing Airworthiness Sector, *Aerospace*, vol. 8(2), p. 27, number: 2 Publisher: Multidisciplinary Digital Publishing Institute.

Croomes, S. (2006): *Overview of the DART Mishap Investigation Results*, *Tech. rep.*, National Aeronautics and Space Administration.

da Silva, A., Sánchez, S., Polo, O.R. et al. (2014): Injecting Faults to Succeed. Verification of the Boot Software On-Board Solar Orbiter's Energetic Particle Detector, *Acta Astronautica*, vol. 95, pp. 198–209.

de Souza, K.V.C.K., Bouslimani, Y., and Ghribi, M. (2022): Flight Software Development for a Cubesat Application, *IEEE Journal on Miniaturization for Air and Space Systems*, pp. 1–1.

Doyle, M., Gloster, A., O'Toole, C. et al. (2020): Flight Software Development for the EIRSAT-1 Mission, in *Proceedings of the 3rd Symposium on Space Educational Activities*, pp. 157–161, arXiv:2008.09074 [astro-ph].

Edward A. Euler, Steven D. Jolly, and H. H. Curtis (2001): The Failures of the Mars Climate Orbiter and Mars Polar LANDER: A Perspective From the People Involved, p. 22, American Astronautical Society.

Forward, A. and Lethbridge, T.C. (2002): The Relevance of Software Documentation, Tools and Technologies: a Survey, in *Proceedings of the 2002 ACM symposium on Document engineering*, DocEng '02, pp. 26–33, New York, NY, USA: Association for Computing Machinery.

Garousi, V., Felderer, M., Karapıçak, C.M. et al. (2018): Testing Embedded Software: A Survey of the Literature, *Information and Software Technology*, vol. 104, pp. 14–45.

Gorbenko, A., Kharchenko, V., Tarasyuk, O. et al. (2012): A Study of Orbital Carrier Rocket and Spacecraft FAILURES: 2000-2009, *Information & Security: An International Journal*, vol. 28, pp. 179–198.

Harvey, B. (2007): *The Rebirth of the Russian Space Program: 50 Years After SPUTNIK, New Frontiers*, 1st ed., New York: Springer.

Hishmeh, S.F., Doering, T.J., and Lumpp, J.E. (2009): Design of Flight Software for the KySat CubeSat Bus, in *2009 IEEE Aerospace conference*, pp. 1–15, iSSN: 1095-323X.

J. L. Lions (1996): *Ariane 5 Flight 501 FAILURE: Report By the Inquiry Board*, *Tech. rep.*, European Space Agency.

Ji, X.Y., Li, Y.Z., Liu, G.Q. et al. (2019): A Brief Review of Ground and Flight Failures of Chinese Spacecraft, *Progress in Aerospace Sciences*, vol. 107, pp. 19–29.

Källén, M., Holmgren, S., and Hvannberg, E.P. (2014): Impact of Code Refactoring Using OBJECT-ORIENTED Methodology on a Scientific Computing Application, in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pp. 125–134.

Langer, M. and Bouwmeester, J. (2016): Reliability of CubeSats - Statistical DATA, DEVELOPERS' Beliefs and the Way Forward, in *Proceedings of the AIAA/USU Conference on Small Satellites*, p. 12.

Larman, C. and Kruchten, P. (2005): *Applying UML and PATTERNS: An Introduction to OBJECT-Oriented Analysis and Design and Iterative Development*, Prentice Hall PTR.

Latachi, I., Rachidi, T., Karim, M. et al. (2020): Reusable and Reliable FLIGHT-CONTROL Software for a FAIL-SAFE and COST-EFFICIENT Cubesat MISSION: Design and Implementation, *Aerospace*, vol. 7(10), p. 146.

Leveson, N.G. (2012): Role of Software in Spacecraft Accidents, *Journal of Spacecraft and Rockets*.

Lilley, S. (2012): Critical SOFTWARE: Good Design Built Right.

Liskov, B. (1987): Data Abstraction and Hierarchy, *ACM SIGPLAN Notices*, vol. 23(5), pp. 17–34.

Martin, R.C. (2000): Design Principles and Design Patterns, *Object Mentor*, p. 34.

Martin, R.C. (2014): *Agile Software Development, Principles, Patterns, and Practices*, 1st ed., Harlow: Pearson, oCLC: 856863624.

McComas, D., Wilmot, J., and Cudmore, A. (2016): The Core Flight System (CFS) COMMUNITY: Providing Low Cost Solutions for Small Spacecraft, Salt Lake City, UT, nTRS Author Affiliations: NASA Goddard Space Flight Center NTRS Report/Patent Number: GSFC-E-DAA-TN33786 NTRS Document ID: 20160010300 NTRS Research Center: Goddard Space Flight Center (GSFC).

Mooney, J.D. (1995): Portability and Reusability: Common Issues and Differences, in *Proceedings of the 1995 ACM 23rd annual conference on Computer science - CSC '95*, pp. 150–156, Nashville, Tennessee, United States: ACM Press.

Nilchiani, R. (2009): Valuing Software-Based Options for Space Systems Flexibility, *Acta Astronautica*, vol. 65(3-4), pp. 429–441.

Plauche, R. (2017): Building Modern CROSS-PLATFORM Flight Software for Small Satellites, *Small Satellite Conference*.

Rumford, T.E. (2003): Demonstration of Autonomous Rendezvous Technology (DART) Project Summary, pp. 10–19, Orlando, FL.

Space Engineering Software (2009): ECSS-E-ST-40C: Space Engineering Software.

Space Product Assurance (2017): ECSS-Q-ST-80C: Space Software Product Assurance.

Stephenson, A.G., LaPiana, L.S., Mulville, D.R. et al. (1999): *Mars Climate Orbiter Mishap Investigation Board Phase 1 Report*, *Tech. rep.*

Tafazoli, M. (2009): A Study of On-Orbit Spacecraft Failures, *Acta Astronautica*, vol. 64(2-3), pp. 195–205.

Tipaldi, M., Legendre, C., Koopmann, O. et al. (2018): Development Strategies for the Satellite Flight Software On-Board Meteosat Third Generation, *Acta Astronautica*, vol. 145, pp. 482–491.

Venturini, C., Braun, B., Hinkley, D. et al. (2018): Improving Mission Success of CubeSats, in *Proceedings of the AIAA/USU Conference on Small Satellites*.

Whitney, T., Straub, J., and Marsh, R. (2015): Design and Implementation of Satellite Software to Facilitate Future CubeSat Development, in *AIAA SPACE 2015 Conference and Exposition*, Pasadena, California: American Institute of Aeronautics and Astronautics.

B.2. **PAPER 2** – EXPERIMENTAL RESULTS AND MODELLING OF THE BINAR-1 CUBESAT ON-BOARD POWER MANAGEMENT IN THERMAL VACUUM CONDITIONS

In preparation for journal submission.

Stuart R. G. Buchan, *Fergus W. Downey, Kyle McMullan, Benjamin A. D. Hartig, Eduardo Trifoni, Joice Mathew, Phil A. Bland, Jonathan Paxman, and Robert M. Howie*

Experimental Results and Modelling of the Binar-1 CubeSat On-Board Power Management in Thermal Vacuum Conditions

Stuart R.G. Buchan[a,*], Fergus W. Downey[a], Kyle McMullan[a], Benjamin A.D. Hartig[a], Eduardo Trifoni[b], Joice Mathew[b], Phil A. Bland[a], Jonathan Paxman[a], Robert M. Howie[a]

[a]*Space Science and Technology Centre, School of Earth and Planetary Science, Curtin University, Kent Street, 6102 WA, Perth, Australia*
[b]*National Space Test Facility, Australian National University, Cotter Road, 2611 ACT, Canberra, Australia*

Abstract

The electrical power system on-board a spacecraft has a crucial role in the success of a mission. Central to this however, the health of the lithium-ion cells frequently used in CubeSat applications degrades when exposed to sub-zero temperatures. With the external faces of a spacecraft in the Low Earth Orbit environment expected to reach -40°C, the necessity of battery heating in the success of a mission is clear when using this battery technology. It therefore became a key element in the design choices for Binar-1, Western Australia's first locally made satellite. The decision was made for Binar-1 that this functionality be delegated from the flight computer to analog circuitry. This is not without risk as the battery heating circuitry composes one of the largest loads on the electrical power system, increasing the possibility of a low-power state occurring on the spacecraft. Thermal simulation can offer insight into battery heater usage, but is a computationally expensive process reserved for high-end desktop computers. This highlights a gap for computationally inexpensive models able to run on-board a spacecraft, predicting in advance when these low power states are to occur. Coupled with software capable of suggesting alterations to a mission operations plan, CubeSats can demonstrate a proof-of-concept approach for safe operations in a communications limited environment when encountered with an emergency state that cannot wait for input from a ground operator. Using data collected from a thermal vacuum experiment in the National Space Test Facility at the Australian National University, the response of the electrical power system and battery heaters in Binar-1 was characterised from being subject to the conditions of Low Earth Orbit. This information is used in the creation of two predictive models of when the battery heaters are to be enabled, influenced by the thermal state inside and external to the spacecraft. From a simulated orbital analysis and a further vacuum chamber test, a model is selected to be used on the next Binar spacecraft to allow the satellite to optimally postpone tasks when the usage of the battery heaters are to be prioritised.

Keywords: Binar Space Program, National Space Test Facility, Thermal Vacuum, Small Satellite, CubeSat, Agile mission planning, Mission Autonomy

1. Introduction

Operating from the Space Science and Technology Centre at Curtin University, the Binar Space Program is enabling a new Western Australian space capability. The primary output from the program, the Binar CubeSat Motherboard (BCM), is a highly integrated custom CubeSat motherboard that comprises a primary and redundant flight computer, a Global Positioning System (GPS), Attitude Determination and Control System (ADCS), on-board storage and Electric Power System (EPS). The BCC was flight-proven for the first time onboard Binar-1 in 2021, and is now being integrated into the next missions from the program.

As the maiden flight of the custom BCM was on Binar-1, the mission was intended to be a technology demonstrator to show the suitability of the Binar Space Program's hardware in a Low Earth Orbit (LEO) environment. Downey et al. [1] provides an overview of the Binar-1 mission and its objectives. The EPS powering the BCM comprises four lithium-ion batteries, seen in Figure 1. Each of the batteries has a dedicated flexible printed circuit board (PCB) heater wrapped over the external casing for thermal maintenance. This is a requirement as the number of discharge cycles and capacity of lithium-ion cells, frequently used in CubeSat applications, have been shown to be dramatically reduced from exposure to conditions below 0°C [2][3]. Rather than activating the thermal maintenance circuitry at 0°C, a safety factor was added

*Corresponding author
Email addresses: stuart.buchan@postgrad.curtin.edu.au (Stuart R.G. Buchan), fergus.downey@postgrad.curtin.edu.au (Fergus W. Downey), kyle.mcmullan@postgrad.curtin.edu.au (Kyle McMullan), ben.hartig@postgrad.curtin.edu.au (Benjamin A.D. Hartig), eduardo.trifoni@anu.edu.au (Eduardo Trifoni), joice.mathew@anu.edu.au (Joice Mathew), p.bland@curtin.edu.au (Phil A. Bland), j.paxman@curtin.edu.au (Jonathan Paxman), robert.howie@curtin.edu.au (Robert M. Howie)

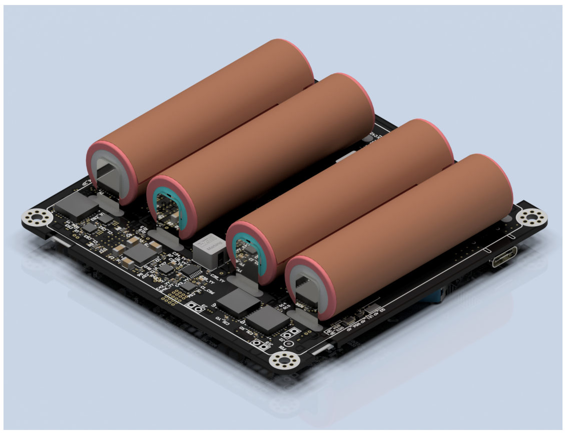


Figure 1: The top side of the Binar CubeSat Motherboard, detailing the EPS. The four batteries that power the EPS are wrapped by flexible printed circuit board (PCB) battery heaters, triggered by analog circuitry to enable when the battery temperature drops below 5°C.

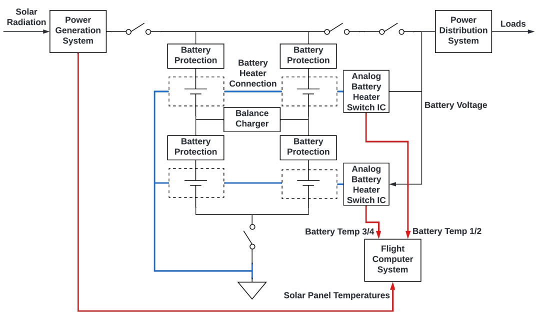


Figure 2: A high level block diagram of the EPS on Binar-1. Highlighted in blue is the electrical connections between the parallel heating circuits, represented by the dashed boxes around the power sources. Each circuit has an independent analog battery heater integrated circuit for operation. The temperature feedback into the flight computer from the solar cells and the battery heating circuitry is shown in red.

on the BCM for the two groups of two heaters to be enabled when the temperature in the battery compartment drops below 5°C. The resistor-programmable temperature switches are set to disable when the temperature set point reaches 10°C to conserve energy. Figure 2 provides a high level block diagram of the EPS on the BCM installed in Binar-1. The battery heating system is separated from the flight logic, being instead implemented through analog circuitry. This was done to ensure the heaters can be enabled as soon as required. As the circuitry warms the batteries through power dissipation, the battery heating circuitry composes one of the largest loads on the EPS. This poses a risk as without control over this load the spacecraft could potentially encounter an electrical low-power state if the heaters are enabled during a period of high current draw on-orbit. This highlights the necessity for on-board autonomy for the spacecraft to predict in advance when these low power states are to occur, without relying on input from an operator.

2. On-board Autonomy

On-board autonomy for spacecraft has aided in the success of numerous historical space missions [4]. It has been successfully demonstrated to be an invaluable resource in a communications denied or limited environment. Straub et al. summarise that in these situations, missions may need to hand control over to autonomous software to handle an emergency response that cannot wait for input from ground operators [5].

The Curiosity Mars rover relied extensively on autonomy during the EDL due to a communications link delay that extended beyond the predicted entry time [6]. Even with a consistent bent pipe link with Earth during EDL, the subsequent Perseverance rover likewise had to rely on autonomy due to the extensive link delay [7]. Both spacecraft autonomously responding to inputs from on-board sensors allowed the rovers to land safely in a situation where ground operators could not provide the real-time input that was needed.

Even without the extensive time delay induced by deep space missions, the importance of autonomy is paramount in a mission in the case where communications are lost. During the landing phase of the Beresheet Lunar mission, a hardware fault requiring a component reset occurred. However, a loss of communications with the ground station prevented reset requests from reaching the spacecraft. By the time communications were restored, the altitude of the lander was too low to slow the descent, and the spacecraft was lost [8][9].

On-board autonomy has also been shown historically to successfully suggest changes to an operations plan. The first Mars rover, Sojourner, relied on this capability as communication could only be made with ground operators twice per Sol. Using on-board autonomy, the spacecraft could infer which commands from the previously sent operations plan could safely be executed when encountering terrain difficult to traverse [10]. Extending on the responsibilities for autonomy on spacecraft, the Deep Space One mission became the first spacecraft to fly an on-board mission planner. The successful mission was able to demonstrate that if the pre-set operations plan is not able to be executed, the spacecraft could respond by autonomously suggesting an alternate solution in an agile manner [11].

Although historical missions have demonstrated that autonomous capabilities on spacecraft can provide a higher level of return than those requiring a ground-based link [4], its extensive use has been met with some scepticism. A 2013 survey of the planetary science community found that there is apprehension toward the acceptance of autonomous control of spacecraft [5]. This is understandable, yet necessary in the context of expensive flagship planetary missions. This highlights a possible application for CubeSats. CubeSats have demonstrated operation in environments extending beyond LEO, whilst maintaining a fast development cycle and low cost [12]. Prior to more widespread adoption on future interplanetary missions however, Feruglio et al. stress that CubeSat mission autonomy needs to be improved [13]. Often described as a 'test bed' for larger missions, the CubeSat form factor can also be leveraged to trial mission autonomy techniques with a significant reduction in risk over larger missions. As the computers powering small spacecraft become smaller, cheaper and more powerful, applications requiring significant processing power are becoming possible in a small spacecraft form factor [5].

To this aim, the Binar Space Program used data obtained from a space environment experiment to develop a model to be used for on-orbit power management predictions. Featuring as a software payload on the upcoming Binar 2, 3 and 4 missions, the inference from the model will allow the spacecraft to autonomously suggest changes to the operations plan if the possibility of a low power state is deemed to occur. This paper details the creation of the model, and the results of preliminary testing.

3. Testing Procedure

The Binar-1 engineering model was placed in the centre of the Wombat XL thermal vacuum chamber at the Advanced Instrumentation and Technology Centre in Canberra, Australia. Situated at the National Space Test Facility (NSTF) at the Mount Stromlo Observatory of the Australia National University, the Wombat XL is a 5.76 cubic meter thermal vacuum chamber used for the testing of space-based instruments and small satellites. It features a liquid nitrogen fed platen and shroud system for thermal cycling from -170°C to +150°C at a rate of 3°C per minute. The support stand for the satellite was made of 3D printed Ultem to limit thermal conduction as much as possible from the platen shown in Figure 3. Seven chamber supplied thermal sensors were placed on the exposed faces; one on each of the solar panels in the bottom right corner (next to the surface mounted thermal sensors

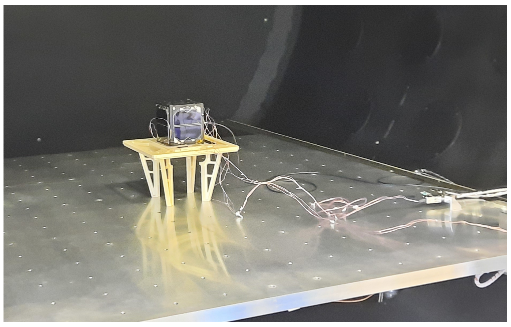


Figure 3: The Binar-1 engineering model placed in the Wombat XL vacuum chamber with thermal probes attached.

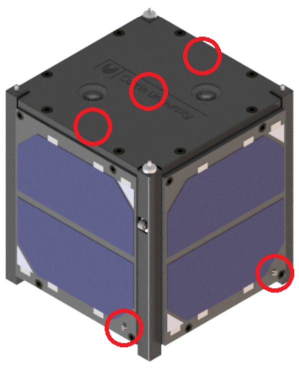


Figure 4: A render of Binar-1 showing the location of the thermal sensors during the experiment. Three sensors were placed on the bottom face of the satellite facing the platen (shown on the top of the satellite in this figure), and one was placed in the corners of all four sides of the solar panels next to the surface mounted panel thermal sensors (only two shown in this figure).

on the panels), and three on the top face of the satellite (shown circled in Figure 4). These sensors were logging with a one second period, coinciding with the internal logging frequency of the sensors inside Binar-1. Internal to the satellite, battery and regulated voltage and current, X and Y axes solar panel voltage and current, individual solar panel temperature, and battery compartment temperature were recorded.

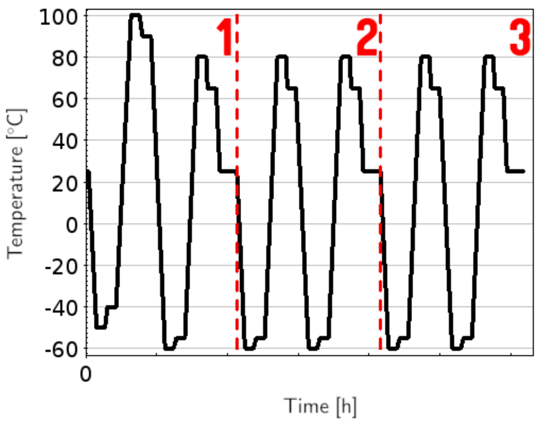
One of the primary objectives for testing Binar-1 at the NSTF was to understand and characterise the functionality of the analog battery heating circuitry in LEO conditions. From calculations done prior to the experiment, Binar-1 was expected to undergo a worst-case temperature swing between -40°C and +65°C on the external faces during an orbit. Initially it was decided to adopt a schedule for the Wombat XL to dwell at -40°C and +90°C in the attempt to have the surface of the solar panels reach -30°C and +80°C respectively to stress test the satellite. The chamber temperature set points were set to be -50°C and +100°C to achieve a thermal gradient of 3°C/min, before stepping to the desired dwell temperature as the platen and shroud approached the set points, as seen in Figure 5(a). Figure 5(b) shows the experimentally recorded average platen and shroud temperature during the experiment.

The chamber pressure was brought to 6E-7 Torr after an initial overnight pump-down. The platen and shroud temperatures were varied, using the feedback from the thermal sensors on the spacecraft to achieve the desired temperature of -30°C on the surface of the panels. Following the initial cold swing, reaching the desired +80°C set point on the surface of the cells took longer than anticipated. This caused the batteries in the spacecraft to reach the temperature limit. To maintain optimal battery health, the maximum internal temperature was intended not to exceed +65°C. However, the maximum recorded temperature of the batteries reached +75°C during the first hot swing, far exceeding this range. After completing the first thermal cycle, it was decided to amend the temperature profile to change over -55°C to +65°C, with the intention of having the surface of the cells reach -45°C to +55°C respectively. The battery heating circuitry could then be stress tested further by allowing a colder external temperature to be reached, whilst protecting the integrity of the battery health on the hot swings. The remainder of the testing schedule continued with these updated values, as seen from the profile in Figure 5.

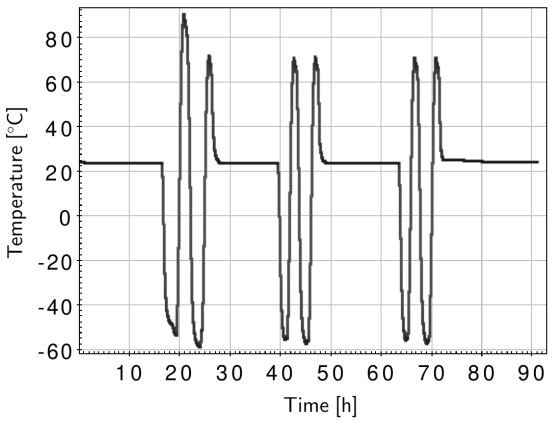
4. Results

4.1. Battery Heating System

The recorded temperature of the batteries internal to the spacecraft is plotted against the solar panel surface temperature in Figure 6(a), showing that the battery temperature did not drop below 6°C during the six thermal cycles performed in the chamber. This demonstrates that the analog battery heating circuitry functioned successfully. Figure 6(b) shows a magnified view of the first two cold swings, showing the response of the spacecraft internal battery temperature to the temperature at the surface of the solar cells over a roughly ten hour period. The spacecraft internal battery temperature lags the solar panel surface temperature, dropping until it reaches 6°C. At this point the battery heaters activate, warming the batteries until they disable at 12°C. The enable and disable temperature differing slightly from the desired set points can be attributed to inaccuracies in the resistors used in the analog logic, the temperature sensors, and the analog to digital converter on-board the flight computer. As the inaccuracy has caused the usage of the heaters to shift in a more conservative direction however, this is deemed acceptable. As the chamber is dwelling at -50°C, the spacecraft eventually cools down enough to re-enable the battery heaters. The heating process again disables at the cut-off temperature, coinciding with the chamber temperature increasing to the hot dwell. After the temperature lag ceases, it is observed that the temperature begins to increase again



(a) The proposed temperature profile for the experiment, comprising six thermal cycles segmented into three days.



(b) The experimentally recorded average platen and shroud temperature during the experiment.

Figure 5: The proposed temperature profile for the experiment and the experimentally recorded average platen and shroud temperatures. Initially the set points for the vacuum chamber were to change over -40°C to +90°C, attempting to get the surface of the panels to -30°C and +80°C respectively. On the hot swing however, the batteries reached +75°C, far exceeding the safe battery temperature of +65°C. Due to this, the thermal profile was amended to change over -55°C to +65°C, allowing for stress testing the battery heating circuitry whilst maintaining safe battery temperatures. Between thermal cycles, the chamber was brought to an ambient temperature.

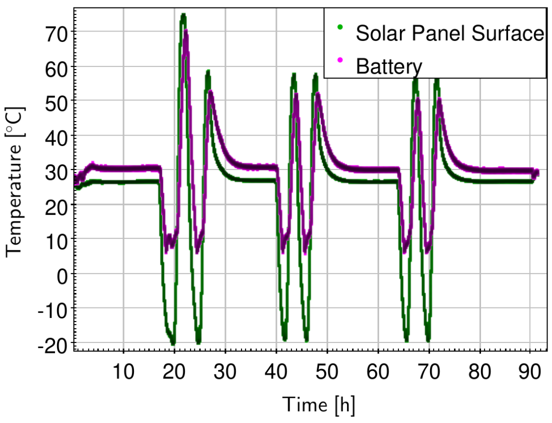
just after the 20 hour mark. The operation of the battery heaters is shown (Figure 6(a)) to consistently repeat during the remaining thermal cycles in the vacuum chamber. This invokes high confidence in the heaters to operate as expected in a LEO environment, keeping the batteries in the optimal temperature range and extending the on-orbit lifetime. As the heaters maintain the battery temperature through power dissipation, their frequent usage, though required, poses a significant load on the EPS.

4.2. Electrical Power System

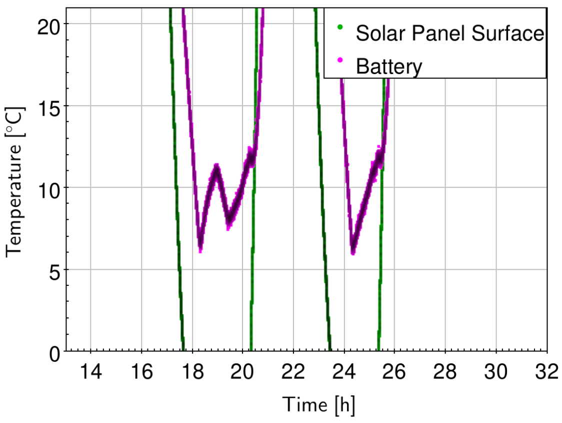
The electrical power system on Binar-1 consisted of four 18650 lithium-ion batteries in a two series and two parallel (2S2P) configuration providing approximately 50 Wh of power with a nominal voltage of 7.4 V. Fully charged, the batteries supply 8.3 V to the motherboard. During the initial overnight pump-down in the Wombat XL, the battery voltage was brought to full charge via a power harness to start the experiment. Binar-1 remained connected to the power harness until the third day of testing.

Figure 7 depicts the battery voltage sharply decreasing each time the battery heaters are enabled. The connected power harness brought the batteries back to full charge for the first two days.

The batteries are shown to have a consistent current draw at approximately 60 mA in Figure 8(a) prior to the battery heating circuitry being enabled. During usage, the current spikes to over 600 mA, explaining the drop in battery voltage seen in Figure 7. The total current draw from the batteries is seen to slope downward during heater usage, attributed to the generated heat increasing the resistivity of the circuit. It should be noted that the current drop is minimal as the battery voltage is also dropping during heater usage. Figure 8(a) shows that the first thermal cycle involves three current peaks, as opposed to the two peaks seen during the remaining thermal cycles. Figure 8(b) shows a magnified view of the current draw during this thermal cycle. The extra peak can be attributed to the first cold dwell lasting for a longer duration than the remaining dwells, resulting in the battery heating circuitry being used twice. As the thermal management of the EPS is done through two groups of two heaters, it is possible for the heaters to be enabled independently. This is clear in Figure 8(b) where a plateau at approximately 300 mA is present on multiple occasions. The variance in the time the current draw remains at these plateaus shows that the heat generated from one of the independent battery heating circuits has an influence on the operation of the other. From the data collected in the chamber however, it is apparent that a single battery heating circuit is not sufficient to keep the entire battery compartment at an optimal temperature. It has been observed that the operation of the heaters imposes a significant power draw on the EPS, and as such limits the power available for other systems. On average, the budget for the power storage system estimates a power usage of 0.370 Whr/orbit. With the satellite estimated to use a total of 1.493 Whr/orbit, this equates to



(a) The temperature of the batteries on Binar-1 plotted against the temperature of the solar panel surface over the experiment duration.



(b) Magnified view of the temperature profile during the first two cold swings, detailing the temperature maintenance of the batteries from the operation of the heaters.

Figure 6: The temperatures of the batteries on-board Binar-1 during the experiment plotted against the temperature of the space simulation chamber. The battery heating circuitry prevents the temperature of the batteries from dropping below 6°C. The heaters are enabled twice in the first cold swing as the chamber was held at the cold set point for a longer duration than the second swing.

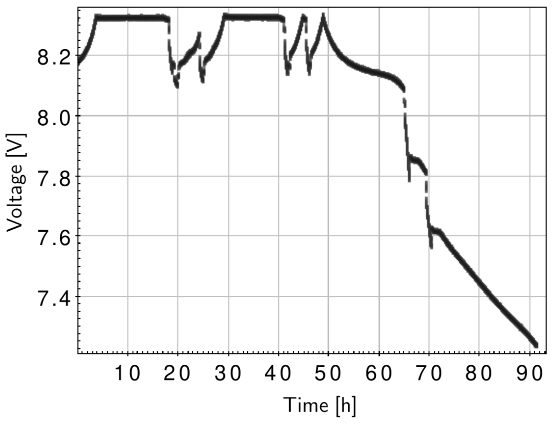
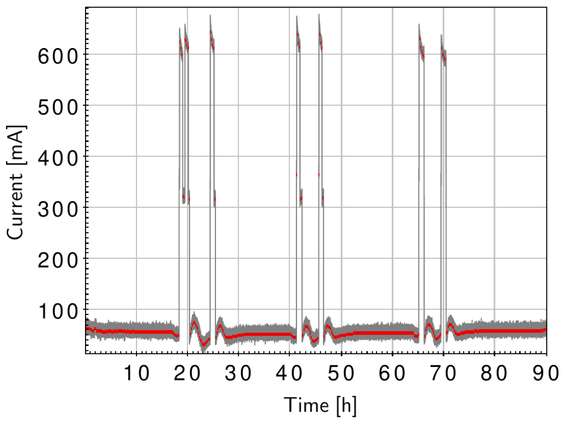
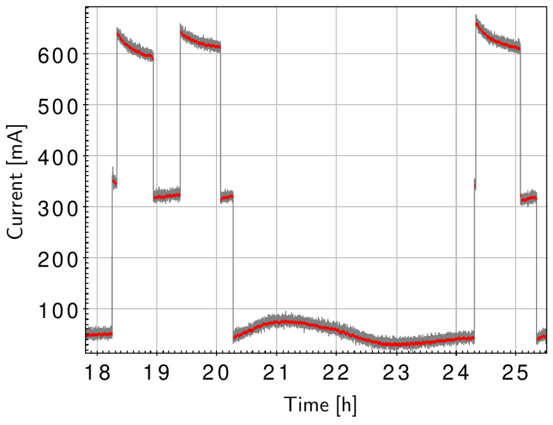


Figure 7: The change in spacecraft battery voltage during the experimentation in the Wombat XL. Initially plugged into the power harness, the batteries charged completely. During sections of high power usage, the voltage profile is seen to drop before charging again. Roughly 50 hours into the experiment, the power harness was removed, allowing the batteries to discharge.



(a) The change in battery current over the experiment duration. The current draw is shown to be consistent for all instances of heater usage.



(b) Magnified view of the battery current profile during the first two cold swings, detailing the magnitude of the power increase during battery heater operation. When enabled, the current spikes roughly ten times higher than the regular operation. Two plateaus are present in the current response, one at roughly 600 mA, and one at roughly 300 mA. This can be attributed to the two groups of battery heaters being enabled and disabled independently. The battery heaters previously mentioned being enabled twice during the first cold swing is further shown here with two corresponding current peaks.

Figure 8: The change in the battery current over the experiment duration. A rolling average has been displayed in red over the data to reduce the noise.

roughly 25% of the power used by the satellite per orbit. Whilst in the Wombat XL, Binar-1 was running low power tasks. In conjunction with the increased power draw, it is expected that a power intensive task such as payload operation could potentially cause a low power state on-orbit. Therefore, it is of crucial importance that usage of the battery heaters can be predicted so that power utilisation can be managed.

5. Thermal Modelling

Alongside physical testing, characterising the stresses induced on a spacecraft from the thermal environment of LEO is possible through the use of thermal simulation software. This is often an integral part of the design phase of a satellite as information obtained regarding the thermal properties of the spacecraft can help influence decisions made when finalising the design. A thermal model of Binar-1 was implemented in Ansys 2021 Thermal Analysis Software to assess the applicability of the Binar CubeSat bus in a LEO-like environment. The thermal model was generated after the launch of Binar-1 to enable improvement of future launches reusing the majority of the bus design. The thermal profile used during the testing at the NSTF is useful for the verification of the Binar-1 thermal model. A summary of the values of emissivity used for the thermal radiative properties of the Binar-1 thermal model is outlined in Table 1. Furthermore, Table 2 shows a summary of the assumptions made in the creation of the Binar-1 thermal model. The third and fourth cold swings (second day) of the experimentation done at the NSTF were simulated to verify the accuracy of the Binar-1 thermal model. These swings were selected as the input

Table 1: A summary of the values of emissivity used in the Binar-1 thermal modelling, adapted from [14].

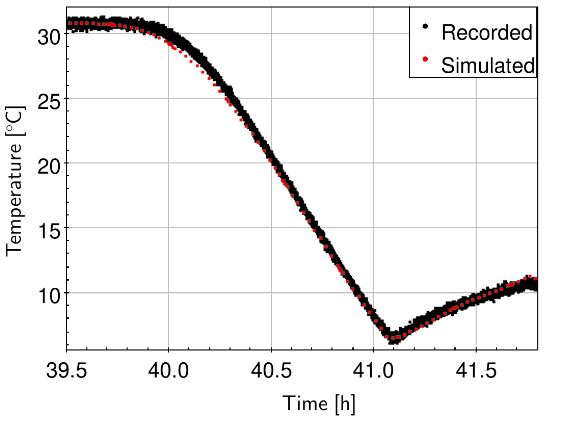
	Emissivity		
Material	**Infrared**	**Visible**	**Total Hemispheric**
Wombat XL Shroud			0.85
Wombat XL Platen			0.15
Aluminium	0.86	0.86	
Solar Panels	0.8	0.86	
PCBs	0.6	Not Exposed	
PV Cell	0.85	0.8	
Polyimide	0.84	Not Exposed	

conditions were identical to the fifth and sixth swings.

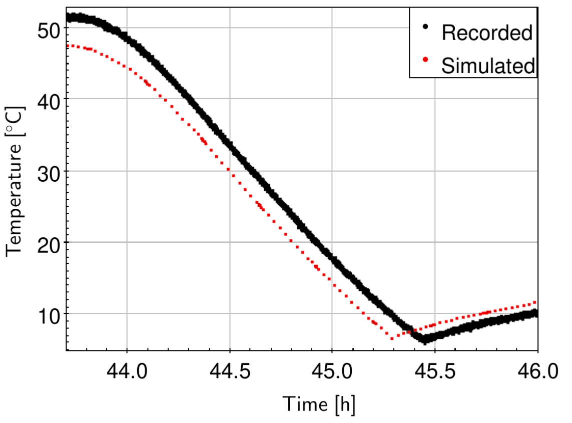
The thermal response of the battery temperature of the Binar-1 thermal model to the simulated Wombat XL chamber temperature is plotted in Figure 9 against the recorded values from the experiment in the chamber. Figure 9(a) shows that the temperature of the batteries in the Binar-1 thermal model closely match the experimental data when entering a cold swing from ambient conditions. The response seen in Figure 9(b) shows less accuracy in the thermal modelling when entering a cold swing from a hot environment, however shows a similar gradient to the recorded data. This inaccuracy can partly be attributed to the assumptions outlined in Table 2, introducing errors in the simulated satellite. Furthermore, the temperature condition preceding the response seen in Figure 9(a) was at a steady state, as this corresponded to the first cold swing after an overnight lull between testing days. The initial temperature condition for the response shown in Figure 9(b) however was the coldest point of the preceding cold dwell. This was done to achieve a realistic initial thermal gradient for the hot-to-cold swing, but resulted in a discrepancy between the simulated and experimental cold swing starting temperatures. Showing a simulated thermal response with high similitude to the experimental data, the Binar-1 thermal model can be subject to varying thermal inputs with confidence that the response seen will be similar to that of the physical satellite. This can aid in characterising the spacecraft in potential mission orbits, and will influence the design decisions for subsequent Binar spacecraft.

6. Power Management Modelling

As the battery heating system is hardware controlled, the BCM is not able to control its operation through software. This can be problematic as the total battery power draw during heating is approximately ten times higher than regular operation (Figure 8), capable of rapidly depleting the batteries. With limited power available onboard, this stresses the need for a prediction of when the heaters are to be operational. Thermal simulation, however, is a computationally expensive process, even when performed on a desktop computer. The drastically reduced capabilities of the primary flight computer on the BCM



(a) The recorded temperature of the third cold swing plotted against the output of the Binar-1 thermal model. The initial thermal condition for the simulation was an ambient steady state. The thermal simulation closely matches the data recorded in the Wombat XL.



(b) The recorded temperature of the fourth cold swing plotted against the output of the Binar-1 thermal model. The thermal simulation of Binar-1 leads the trend recorded in the Wombat XL, attributed to the initial thermal condition for the simulation starting at the coldest point of the preceding cold dwell.

Figure 9: A comparison between the experimental recordings of the third and fourth cold swings in the Wombat XL and the thermal response from simulating the Binar-1 thermal model. The discrepancy in accuracy can be attributed to the initial thermal conditions. The simulation of the third cold swing used an ambient steady state starting condition, whereas the simulation of the fourth cold swing used the coldest point of the preceding cold swing. This was done to achieve a realistic initial thermal gradient for the hot-to-cold swing.

Table 2: The simplifications made in the creation of the Binar-1 thermal model.

Simplification	Impact	Justification
No energy is lost to free space; Radiation transfer for exposed faces in the simulation is limited to surface-to-surface.	Minor	Reduces simulation time.
Specific heat capacity of aluminium material was reduced by 10% to account for the holes in the frame of the spacecraft not being modelled.	Moderate	Reduces nodes in the mesh.
As the thermal properties of all structural materials are similar to aluminium alloy, they are all given the properties of the Ansys defined material 'Aluminium Alloy'.	Minor	Reduces setup time between thermal simulations.
As the radiative properties of the external non-structural materials are similar to black anodised aluminium, they are all given the properties of black anodised aluminium as measured by [14].	Minor	Reduces setup time between thermal simulations.
Battery cells assumed to be thermal properties of lithium-ion as measured by [15].	Moderate	Compensating for inhomogeneity in cells significantly increases simulation complexity, and was deemed beyond the scope of this model.
Battery heaters are given the thermal properties of the most abundant material, Polyamide.	Minor	Compensating for copper traces in the heaters significantly increases simulation complexity, and was deemed beyond the scope of this model.
PCBs given the thermal properties of the most abundant material, FR4 as measured by [16].	Moderate	Compensating for copper traces and integrated circuits on the PCBs significantly increases simulation complexity, and was deemed beyond the scope of this model.
Solar cells are given the thermal properties of the most abundant material, gallium arsenide as measured by [17].	Minor	Compensating for cover glass and metal tabs on solar cells significantly increases simulation complexity, and was deemed beyond the scope of this model.
General approximate value for contact thermal conductance found through trial and error and set for all surface-to-surface contacts.	Significant localised errors	Getting true values for all surface-to-surface contacts involves disassembly of the satellite, followed by significant testing of each material. Deemed beyond the scope of this model.

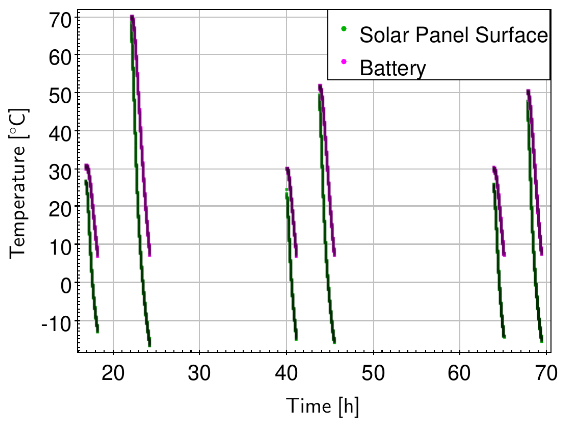


Figure 10: Isolated view of the temperature response of the spacecraft to entering a cold swing in the Wombat XL. In each of the six groups of trends, the right (pink) trend shows the thermal response inside the battery compartment at the core of the satellite. The left (green) trend shows the thermal response of the surface of the solar cells. The segments have been taken from when the surface of the cells begin to respond to a cold cycle and stop when the battery heaters are enabled.

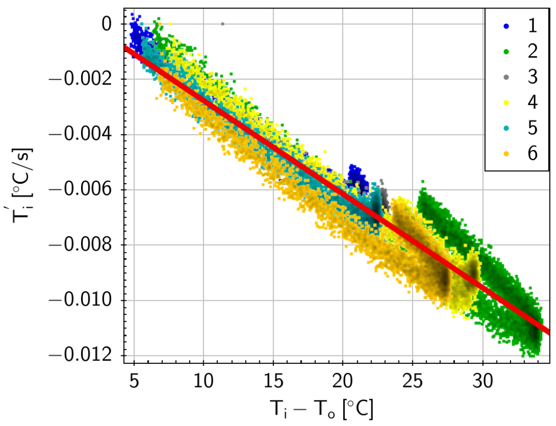


Figure 11: The relationship between the difference in the internal and external temperatures of the spacecraft and the rate of change of the temperature internal to the spacecraft at the batteries for each of the cold swings. A linear fit has been applied to the data. All six series shown in Figure 10 are superimposed here, showing a consistent trend.

would prove infeasible to effectively utilise the simulation on orbit. Hence, simplified linear modelling is employed in the prediction of battery heater usage instead. This section details the creation of this a model, and discusses its applicability for autonomously suggesting changes to the mission plan on-orbit without needing to communicate with the ground.

From the data collected on the internal and external spacecraft temperature sensors during testing at the NSTF facility, the sections leading into a cold swing can be isolated to assess the relationship between the readings prior to the battery heaters being enabled. Figure 10 shows the temperature on the surface of the solar panels and at the spacecraft batteries for each of the experiment cold swings. As the temperature inside of the spacecraft is influenced by the temperature measured at the surface of the solar cells, a model can be generated that relates the difference in the internal and external temperatures to the rate of change of temperature internal to the spacecraft at the batteries. The internal temperature is taken to be the average between the two thermal sensors in the spacecraft battery compartment. Likewise, the external temperature from the average of the four solar panel thermal sensors. A ten minute rolling average of these readings was used for smoothing and a linear approximation was trialled due to its low complexity to assess the fit (Figure 11):

$$T_i^{'} = -3 * 10^{-4}(T_i - T_o) - 6 * 10^{-4} \tag{1}$$

Where T_i and $T_i^{'}$ refer to the spacecraft battery temperature and the corresponding rate of change, and T_o refers to the temperature at the surface of the solar cells.

The model given in Equation 1 has a coefficient of determination of 0.8833. This shows a relatively strong linear fit in the region between entering a cold swing and enabling the battery heaters. The present error can be partially attributed to noise in the sensors recording the temperature, and from the data processing to obtain the relationship. Moreover, the lower right of Figure 11 shows a particularly noisy region of the data present in all of the superimposed series. This can be attributed to the difference between the internal and external temperatures starting to re-converge after they separate, showing that the relationship is not bijective. However, the corresponding rate of change to the points that re-converge seem to fit with the linear approximation with minor error. Knowing that the analog battery heaters are programmed to turn on at 6°C, Equation 1 can be used to give a reasonable prediction of the time until the battery heaters are enabled.

$$T_{enable} - T_i = T_i^{'} * t_{heaters\_on} \tag{2}$$

$$\therefore t_{heaters\_on} = \frac{T_{enable} - T_i}{-3 * 10^{-4}(T_i - T_o) + 6 * 10^{-4}} \tag{3}$$

Where T_{enable} refers to the battery heaters enable condition, 6°C. The model given in Equation 3 is propagating the instantaneous rate of change to predict when the battery heaters will be enabled. Due to this, at the start of the cold swing when the delta between the internal and external temperatures is the smallest, the model will be unstable when the corresponding rate of change of the internal temperature is minimal. This is evident in Figure 12, where the time taken between the surface of the cells responding to a cold cycle and the battery heaters enabling for the fifth cold swing is plotted against the predictive

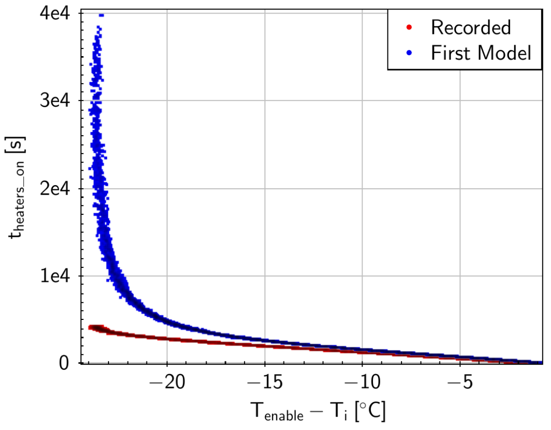


Figure 12: The predicted amount of time in seconds until the spacecraft battery heaters are enabled plotted against the difference between the battery temperature and the battery heater enable temperature. The actual recorded time remaining is also plotted for reference. The model is unstable above a 20°C separation between the internal temperature and the battery heater enable temperature as it relies on propagating the instantaneous rate of change of the battery temperature. When the difference between the internal and external temperatures is minimal, the model tends towards an infinite amount of time required. As the temperature gradient stabilises, the prediction increases in accuracy.

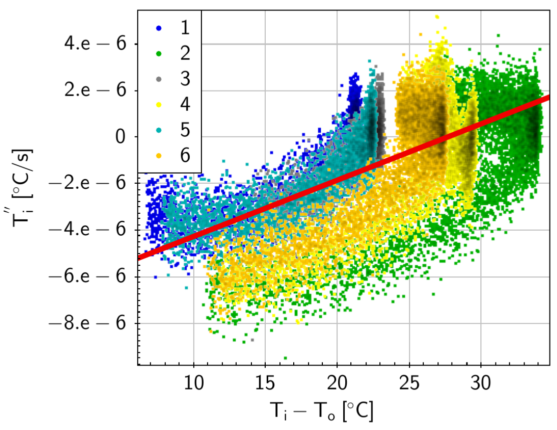


Figure 13: The relationship between the difference in the internal and external temperatures of the spacecraft and the second derivative of the temperature internal to the spacecraft at the batteries for each of the cold swings. All six series shown in Figure 10 are superimposed here. A linear fit has been applied to the data.

model. As the delta between the internal and external temperatures increases and the rate of change becomes more constant, the accuracy begins to increase. Below a 20°C temperature difference between the battery temperature and the heater enable temperature, the model can reliably predict the time remaining before the heaters were enabled in the Wombat XL chamber.

In an attempt to compensate for the inaccuracies introduced from propagation of small rates of change, a second linear approximation was produced relating the difference between the temperature at the surface of the solar cells and the internal batteries, and the second derivative of the internal temperature response with respect to time (Figure 13):

$$T_i^{"} = 2 * 10^{-7}(T_i - T_o) - 7 * 10^{-6} \tag{4}$$

Where T_i" refers to the second derivative of the temperature of the batteries. The model given in Equation 4 has a coefficient of determination of 0.3988 showing a poor linear fit in comparison to the first model. Approximating the second derivative of the internal temperature to be constant with respect to time (due to only a very minor change), an estimation for the time required to reach the battery heater enable temperature can be given by:

$$T_{enable} - T_i = T_i^{'} t_{heaters\_on} + \frac{1}{2} T_i^{"} * t_{heaters\_on}^2 \tag{5}$$

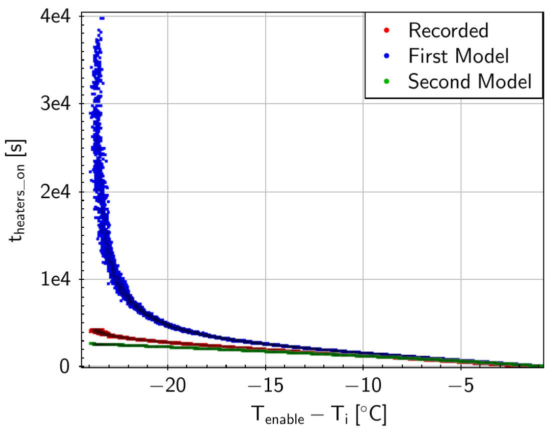
$$\therefore t_{heaters\_on} = \frac{-T_i^{'} \pm \sqrt{T_i^{'2} + 2T_i^{"}(T_{enable} - T_i)}}{T_i^{"}} \tag{6}$$

Plotting the predicted time for the battery heaters to be enabled using the second model against the recorded data and the first model (Figure 14(a)) shows that the inaccuracies due to propagation of the instantaneous rate of change have been addressed. Figure 14(b) provides a magnified view of the stable range of the first model, showing the predictions based on the second model seem to track the recorded data with higher accuracy. The first model, whilst maintaining a similar gradient to the recorded data, consistently predicts a longer time till enable than the recorded data. Applying both predictive models to the sixth cold swing (Figure 15) however shows a higher accuracy with the first model, with both predictive models under-predicting the time remaining until the battery heaters are enabled. The prediction inaccuracy between different cold swings can be attributed to the errors present in fitting the experimental data to a linear approximation. As these models were produced from limited data collected during one experiment with a single spacecraft, the authors acknowledge the possibility of overfitting the models to the data. The next section therefore applies the generated models on-board the Binar-1 engineering model in the Curtin University vacuum chamber, and to the Binar-1 thermal model in a simulated LEO environment to assess their efficacy in heater usage prediction.

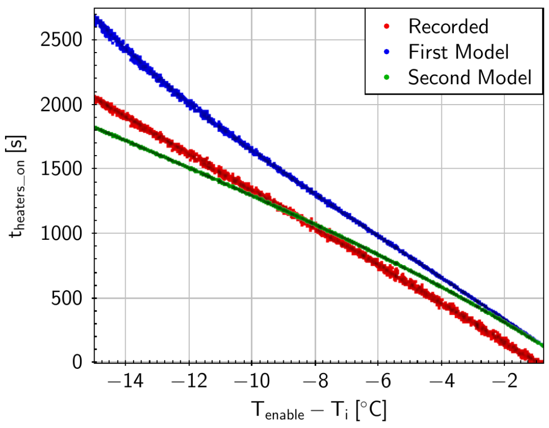
7. Model Verification

7.1. Thermal Vacuum

The engineering model of Binar-1 was placed in the thermal vacuum chamber at Curtin University to test the derived models on-board the satellite during thermal cooling. A summary of the differences to the Wombat XL



(a) The predicted amount of time in seconds until the spacecraft battery heaters are enabled plotted against the difference in the battery temperature and the enable temperature (6°C). Both predictive models and the recorded data from the fifth cold swing are shown. The second model compensates for the inaccuracies introduced due to the non linearity seen in the first model, giving a higher prediction accuracy in the higher temperature difference ranges.



(b) Magnified view of the predictive models. The first model produced consistently predicts a higher time estimation than the recorded data until the heaters are enabled. The second model is seen to more accurately track the recorded data.

Figure 14: Predictions from both derived models of the time remaining until the spacecraft battery heaters are enabled, plotted against the recorded data from the fifth cold swing in the Wombat XL.

Table 3: Chamber differences

	Curtin University	Wombat XL
Volume (m^3)	0.034	5.76
Temperature Range (°C)	-100 to +100	-170 to +180

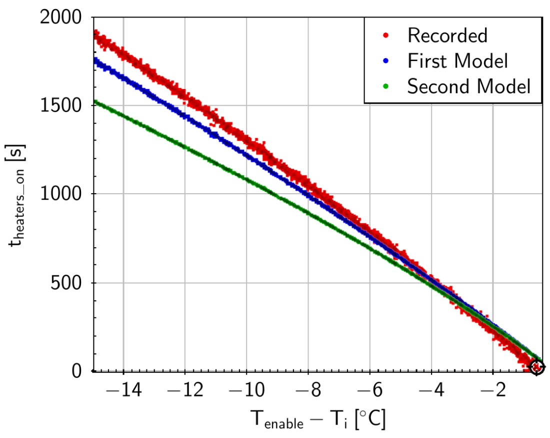


Figure 15: Magnified view of both predictive models of the time remaining until the spacecraft battery heaters are enabled, plotted against the recorded data from the sixth cold swing in the Wombat XL.

can be seen in Table 3. After pumping down to an average chamber pressure of 7.5E-4 Torr, ten hours of thermal vacuum testing was performed (Figure 16) before the chamber was permitted to return to ambient conditions. During this period, the battery heating circuitry was enabled twice. The subsets of the solar panel surface and battery temperatures pertaining to the regions preceding the battery heaters being enabled are highlighted in black and orange.

Applying the predictive models to the first and second instances preceding heater usage (Figures 17 and 18 respectively), the first predictive model is seen to more closely track the time remaining until the heating circuitry is enabled than the second. Both predictive models in this instance consistently under-predict the remaining time until the heaters are enabled. The predictions are seen to converge with the actual time remaining as the enable temperature set point is neared.

7.2. *LEO Simulation*

With the Binar-1 thermal model showing high similitude to the engineering model tested at the NSTF, the thermal input to the simulation was updated to emulate the temperatures expected over the course of an orbit.

The on-orbit thermal simulation of the Binar-1 thermal model used an incident radiation on the satellite from direct sunlight of 1400 W/m^2, an Earth albedo of 150 W/m^2, and IR radiation from Earth of 200 W/m^2 [18]. The spacecraft was assumed to be nadir pointing, with the exposure time of thermal radiation from the sun being distributed across the exposed faces evenly. An excerpt of the thermal response of the internal spacecraft temperature is plotted against the two prediction models in Figure 19, focusing on the section of the orbit where Binar-1 is

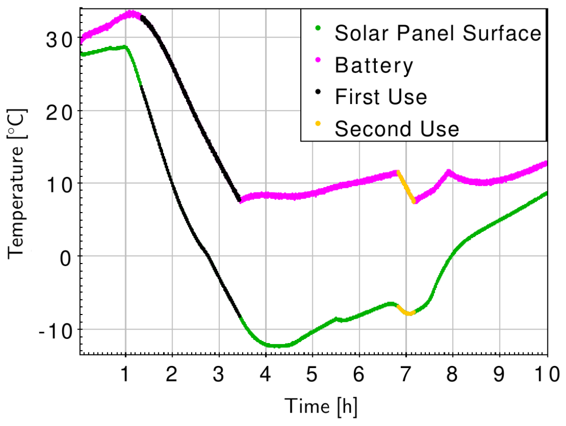


Figure 16: The change in the battery temperature and solar panel surface temperature during the thermal vacuum test at Curtin University. The heating circuitry was enabled twice during the test. The regions applicable to model usage are highlighted in black and orange. Completion of the first usage of the models saw only one heater activate. As such, the battery warming portion following the first usage of the heating circuitry takes longer than the second as the chamber shroud was continued to be cooled in an attempt to trigger the second heater. The completion of the second use of the predictive models features both heaters activated.

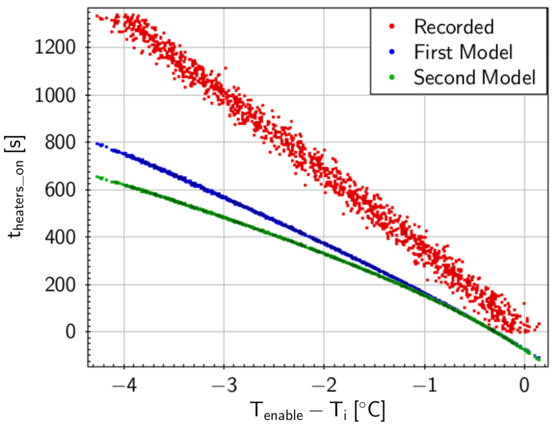


Figure 18: The predicted amount of time in seconds until the spacecraft battery heaters are enabled plotted against the difference in the battery temperature and the enable temperature. Also present is the actual recorded time until the heaters were enabled. This section is taken from the temperature profile preceding the second usage of the battery heating circuitry, highlighted in orange in Figure 16. The first predictive model tracks the actual remaining time slightly more accurately than the second model

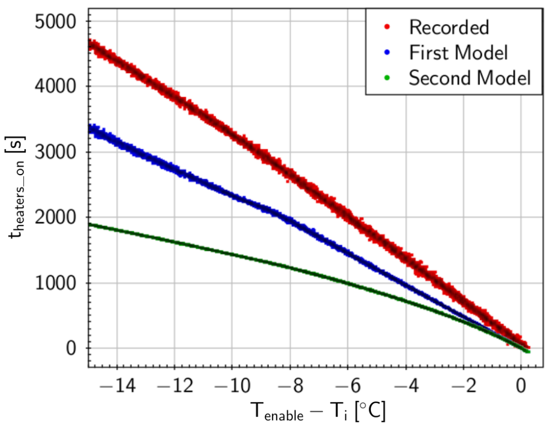


Figure 17: The predicted amount of time in seconds until the spacecraft battery heaters are enabled plotted against the difference in the battery temperature and the enable temperature during the experiment in the Curtin University vacuum chamber. Also present is the actual recorded time until the heaters were enabled. This section is taken from the temperature profile preceding the first usage of the battery heating circuitry, highlighted in black in Figure 16. The first predictive model tracks the actual remaining time more accurately than the second model.

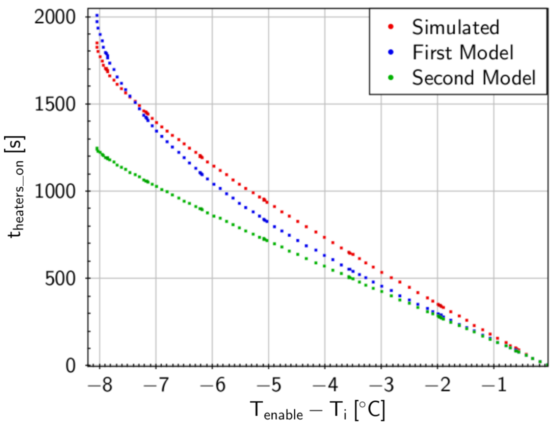


Figure 19: The predicted amount of time in seconds until the simulated spacecraft battery heaters are enabled plotted against the two produced models. The simulated spacecraft temperature was started at the thermal equilibrium (6°C). The data range is focused on the section of the orbit where Binar-1 is eclipsed by Earth, and the internal battery temperatures are starting to drop. The first model has a higher accuracy than the second in the prediction of the recorded data.

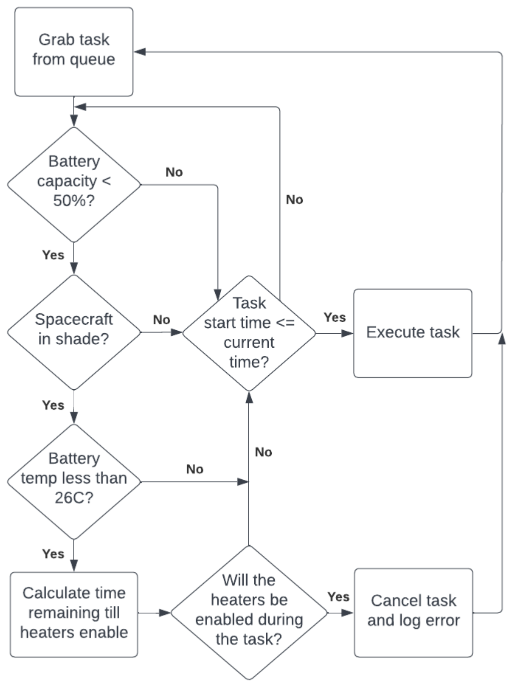

Grab task from queue
Battery capacity < 50%?
No
No
Yes
Spacecraft in shade?
No
Task start time <= current time?
Yes
Execute task
Yes
Battery temp less than 26C?
No
No
Yes
Calculate time remaining till heaters enable
Will the heaters be enabled during the task?
Yes
Cancel task and log error

Figure 20: The proposed implementation of the predictive modelling in the Binar flight logic. If there is insufficient battery capacity to execute a task and run the battery heaters, a series of checks are employed before calculating the expected time remaining before the heating circuitry is operational. This can be used to cancel the execution of a task that could otherwise cause the spacecraft to enter a low-power state.

eclipsed by Earth and the internal battery temperatures are starting to drop.

It is evident that the predictive models closely match the simulated response of the battery temperatures of the Binar-1 thermal model, giving high confidence of the efficacy of the model in LEO-like environments.

8. Discussion

The relative low complexity of the models allows them to be included in the spacecraft flight logic to predict the usage of the heaters. The application mode of the Binar software framework contains a thread that executes scheduled tasks from a queue. Each task in the queue contains the software to execute and the start and end times of the task. In the software developed for Binar-1, when a task start time arrives, the task would be executed regardless of the status of the system vitals. From the experimentation done with the battery heaters, this is shown to be problematic as a high power task coupled with battery heater usage could potentially see the spacecraft encounter a low-power state. Therefore, the modelling derived in Section 6 can be used prior to task execution to assess if the spacecraft can safely progress with doing so. Figure 20 provides a proposed integration into the flight logic. After a task is popped from the queue, the battery capacity is to be checked to determine if the predictive modelling is required. If a sufficient charge is present, the flight logic can progress with regular task execution. However, if the spacecraft has insufficient charge, the predictive modelling is to be used to determine if the spacecraft is safe to progress with task execution. The average solar panel voltage is to be checked to determine if the Earth eclipses the sun from the spacecraft. Following this, a further check is employed to verify that the internal battery temperature is less than the heater enable temperature summed with the maximum usable range of the model as discussed in Section 6, 26°C. In the case which this check succeeds, a prediction of the time remaining until the heaters are enabled will be calculated. Summed with the current time, this predictive value can be referenced against the task end time to estimate if the heaters are to be operational during the task. This condition will see the task cancelled, and an error logged on board to be communicated to the ground station.

The responses seen from applying the predictive models to further thermal vacuum testing at Curtin University and the Binar-1 thermal model show a higher predictive accuracy in the time remaining until heater usage using the first model over the second. This relationship is also present in the application of the predictive models to the sixth cold swing recorded at the Wombat XL. Moving forward therefore, the first model is to be implemented in the spacecraft flight logic for the prediction of the usage of the battery heating circuity. The error in the prediction of the battery heater usage varies between the application to the data collected at the Wombat XL, the data collected at Curtin University, and a simulation of the Binar-1 thermal model in a LEO environment. This is likely due to unmodelled factors being present in the model, as it simplifies a relationship that is inherently non-linear. However, this is expected as the intention of the model is to provide a simplified computationally inexpensive means of prediction which by nature will deviate from a high accuracy model. Due to this, it is deemed sufficient to progress to an on-orbit test of the predictive model as a precursor to further development.

9. Conclusions and Future Work

Being the first spacecraft built by the Binar Space Program, Binar-1 comprised many custom components that lacked flight heritage. Due to this, testing in a simulated space environment prior to launch was crucial in understanding the performance of the spacecraft in LEO conditions. With this objective in mind, verification of the battery heating system electrical characteristics was of high priority.

The testing procedure in the Wombat XL thermal vacuum chamber demonstrated that the battery heating system functioned as expected, keeping the internal temperature of the spacecraft from dropping below 6°C, demonstrating its reliability in LEO like environments. It was seen however that when the battery heaters were enabled, the power requirement from the electrical power system increased dramatically. As the battery heating system cannot be controlled by the flight logic, it stressed the necessity for on-board autonomy to predict when the heaters are to be used. With this, the spacecraft can make changes to the operations plan to prevent power intensive tasks running on the spacecraft during periods of high current draw from the heaters, possibly resulting in a low-power state.

From the data recorded during the testing in the thermal vacuum chamber, two models were generated relating the difference between the internal and external temperatures of the spacecraft to the first and second derivatives of the internal battery temperature with respect to time. These models were shown to predict the enabling of the battery heaters when the spacecraft entered a cold swing in the thermal vacuum chamber with sufficient accuracy. The engineering model of Binar-1 underwent further thermal vacuum testing at Curtin University with the two predictive models derived in Section 6 running on-board. Both models were found to track the estimated time until the battery heaters were enabled. To simulate the efficacy of the models on-orbit, a thermal model of Binar-1 was used. Verified against the recorded data in the Wombat XL, the Binar-1 thermal model was subject to a thermal simulation of a LEO environment. Applying the predictive models on the thermal data produced by the simulation of the Binar-1 thermal model showed that the first predictive model estimated the time remaining for the usage of the battery heaters with a higher accuracy than the second predictive model. This was likewise demonstrated from the results of the thermal vacuum testing at Curtin University, and the sixth cold swing in the Wombat XL. Due to this, moving forward the first model will be tested in the flight logic on future Binar spacecraft.

It was evident from the application of the models that the presence of unmodelled factors caused an inconsistent error during experimental predictions. Further testing on-orbit will contribute to refinement of the predictive model, potentially compensating for unmodelled factors and increasing the accuracy of the prediction of battery heater usage. Feruglio et al. demonstrate the benefits of using artificial neural networks on a CubeSat to improve mission autonomy when applied to a payload optimisation problem. Having been proven feasible, they propose that their research may be extended to encompass autonomous failure detection and isolation [13]. Acknowledging the predictive modelling proposed in this paper simplifies an inherently non-linear relationship, the predictive modelling may also be improved by using orbital data in training of a machine learning model of battery heater usage, with on-board inference.

Some apprehension exists in the community for relinquishing control over the mission operations plan to autonomous software. On-board autonomy on spacecraft however is crucial for safe operations in a communications limited environment to handle emergencies that cannot wait for input from ground operators. CubeSats offer a platform to trial on-board autonomy with a significant reduction in risk over larger, more expensive missions. The derived model will thus be used on Binar-2, 3 and 4 to infer when a task is not safe to execute due to power usage concerns without requiring input from a ground operator.

10. Acknowledgements

This research was supported by an Australian Government Research Training Program (RTP) Scholarship. It received Strategic Research Investment support from Curtin University and institutional support Australian National University.

References

[1] F. Downey, S. Buchan, B. Hartig, D. Busan, J. Cook, R. Howie, P. Bland, J. Paxman, Binar Space Program: Binar-1 Results and Lessons Learned, in: Proceedings of the Small Satellite Conference, 2022.
URL `https://digitalcommons.usu.edu/smallsat/2022/all2022/56`

[2] S. Ma, M. Jiang, P. Tao, C. Song, J. Wu, J. Wang, T. Deng, W. Shang, Temperature effect and thermal impact in lithium-ion batteries: A review, Progress in Natural Science: Materials International 28 (6) (2018) 653–666. `doi:10.1016/j.pnsc.2018.11.002`.
URL `https://www.sciencedirect.com/science/article/pii/S1002007118307536`

[3] O. Erdinc, B. Vural, M. Uzunoglu, A dynamic lithium-ion battery model considering the effects of temperature and capacity fading, in: 2009 International Conference on Clean Electrical Power, 2009, pp. 383–386. `doi:10.1109/ICCEP.2009.5212025`.

[4] J. Straub, A Review of Spacecraft AI Control Systems, WMSCI 2011 - The 15th World Multi-Conference on Systemics, Cybernetics and Informatics, Proceedings 2 (2011) 20–25.

[5] J. Straub, Attitudes towards Autonomous Data Collection and Analysis in the Planetary Science Community, Galaxies 1 (1) (2013) 44–64, number: 1 Publisher: Multidisciplinary Digital Publishing Institute. `doi:10.3390/galaxies1010044`.
URL `https://www.mdpi.com/2075-4434/1/1/44`

[6] D. Way, J. Davis, J. Shidner, Assessment of the Mars Science Laboratory entry, descent, and landing simulation 148 (2013) 563–581.

[7] F. Abilleira, G. Kruizinga, S. Aaron, K. Angkasa, T. Barber, P. Brugarolas, D. Burkhart, A. Chen, S. Demcak, J. Essmiller, J. Gilbert, E. Gustafson, J. Kangas, M. Jesick, S. E. McCandless, S. Mohan, N. Mottinger, C. O'Farrell, R. Otero, C. Pong, M. Ryne, J. Seubert, P. Thompson, S. Wagner, M. Wong, Mars 2020 Perseverance trajectory reconstruction and performance from launch through landingAccepted: 2021-10-15T15:58:51Z Publisher: Pasadena, CA: Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2021 (Aug. 2021).
URL `https://trs.jpl.nasa.gov/handle/2014/52267`

[8] H. Shyldkrot, E. Shmidt, D. Geron, J. Kronenfeld, M. Loucks, J. Carrico, L. Policastri, J. Taylor, The First Commercial Lunar Lander Mission: Beresheet, 2019.

[9] Y.-B. Kim, H.-J. Jeong, S.-M. Park, J. H. Lim, H.-H. Lee, Prediction and Validation of Landing Stability of a Lunar Lander by

a Classification Map Based on Touchdown Landing Dynamics Simulation Considering Soft Ground, Aerospace 8 (12) (2021) 380, number: 12 Publisher: Multidisciplinary Digital Publishing Institute. doi:10.3390/aerospace8120380.
URL https://www.mdpi.com/2226-4310/8/12/380

[10] H. W. Stone, Mars Pathfinder Microrover: A Small, Low-Cost, Low-Power SpacecraftAccepted: 2004-09-30T06:15:24Z (1996).
URL https://trs.jpl.nasa.gov/handle/2014/25424

[11] D. Bernard, E. Gamble, G. Man, G. Dorais, B. Kanefsky, W. Millar, K. Rajan, J. Kurien, N. Muscettola, P. Nayak, Spacecraft autonomy flight experience - The DS1 Remote Agent Experiment, in: Space Technology Conference and Exposition, American Institute of Aeronautics and Astronautics, Albuquerque,NM,U.S.A., 1999. doi:10.2514/6.1999-4512.
URL https://arc.aiaa.org/doi/10.2514/6.1999-4512

[12] T. J. Martin-Mur, B. Young, Navigating MarCO, the first interplanetary CubeSatsAccepted: 2020-07-16T16:15:57Z Publisher: Pasadena, CA: Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2019 (Feb. 2019).
URL https://trs.jpl.nasa.gov/handle/2014/49242

[13] L. Feruglio, S. Corpino, Neural networks to increase the autonomy of interplanetary nanosatellite missions, Robotics and Autonomous Systems 93 (2017) 52–60. doi:10.1016/j.robot.2017.04.005.
URL https://www.sciencedirect.com/science/article/pii/S0921889016304419

[14] J. H. Henninger, Solar absorptance and thermal emittance of some common spacecraft thermal-control coatings, nTRS Author Affiliations: NASA Goddard Space Flight Center NTRS Report/Patent Number: REPT-84F0248 NTRS Document ID: 19840015630 NTRS Research Center: Legacy CDMS (CDMS) (Apr. 1984).
URL https://ntrs.nasa.gov/citations/19840015630

[15] H. Maleki, S. Al Hallaj, J. R. Selman, R. B. Dinwiddie, H. Wang, Thermal properties of lithium-ion battery and components, Journal of the Electrochemical Society 146 (3) (1999) 947, publisher: IOP Publishing.

[16] Z. Peterson, FR4 Thermal Properties to Consider During Design (Nov. 2020).
URL https://www.nwengineeringllc.com/article/fr4-thermal-properties-to-consider-during-design.php

[17] Z. Tian, S. Lee, G. Chen, A Comprehensive Review of Heat Transfer in Thermoelectric Materials and Devices, Annual Review of Heat Transfer 17 (Jan. 2014). doi:10.1615/AnnualRevHeatTransfer.2014006932.

[18] A. Raslan, G. Michna, M. Ciarci, Thermal Simulation of a CubeSat, in: 2019 IEEE International Conference on Electro Information Technology (EIT), 2019, pp. 453–459, iSSN: 2154-0373. doi:10.1109/EIT.2019.8833736.

APPENDIX C

FIRST AUTHOR CONFERENCE ABSTRACTS

C.1. THE BINAR CUBESAT PROGRAM: DESIGN AND DEVELOPMENT OF A CUBESAT DIGITAL TWIN

Stuart R. G. Buchan [1], Phil A. Bland [1], Jonathan Paxman [2], Benjamin A. D. Hartig [1], Nathaniel D. Brough [1], Fergus W. Downey [1], Daniel C. Busan [1], Martin Towner [1], Martin Cupak [1], Robert M. Howie [1]

[1]Space Science and Technology Centre, Curtin University, Perth, Australia

[2]Department of Mechanical Engineering, Curtin University, Perth, Australia

This conference abstract was presented at the 19th Australian Space Research Conference (ASRC 2019) in Adelaide, Australia.

The Binar CubeSat Program: Design and Development of a CubeSat Digital Twin

Stuart R. G. Buchan[*1], **Phil A. Bland**[1], **Jonathan Paxman**[2], **Benjamin A. D. Hartig**[1], **Nathaniel D. Brough**[1], **Fergus W. Downey**[1], **Daniel C. Busan**[1], **Martin Towner**[1], **Martin Cupak**[1], **Robert M. Howie**[1]

[1]Space Science and Technology Centre, Curtin University, Perth, Australia
[2]Department of Mechanical Engineering, Curtin University, Perth, Australia

*contact: stuart.buchan@postgrad.curtin.edu.au

Unlike Earth constrained engineering development, the design of spacecraft poses a significant problem - due to the cost of launch, testing in the operative environment becomes difficult. This therefore shows the advantage of simulation software to emulate the environment the satellite is to operate in. Writing separate code for on-board and simulation purposes introduces the possibility of deviation from the modelled environment, lowering mission confidence. Through the use of abstraction layers the Binar CubeSat Program leverages the simulation environment further in the production of a “Digital Twin” - tailor made software that runs simultaneously in a simulated environment, as well as the hardware environment. Breaking the barrier between simulation code and implementation code allows us to accurately portray how the spacecraft will respond to its environment, increasing confidence in our mission.

C.2. BINAR CUBESAT PROGRAM: MISSION TWO PAYLOADS AND OPERATION PLAN

Stuart R. G. Buchan[1], Fergus W. Downey[1], Benjamin A. D. Hartig[1], Daniel C. Busan[1], Jacob Cook[1], Chelsea Tay[1], Kyle McMullan[1], Tristan Ward[1], Robert M. Howie[1], Phil A. Bland[1], and Jonathan Paxman[2]

[1]Space Science and Technology Centre, Curtin University, Perth, Australia

[2]Department of Mechanical Engineering, Curtin University, Perth, Australia

This conference abstract was presented at the 36th Small Satellite Conference (SmallSat 2022) in Utah, USA.

SSC22-WKVIII-07

Binar Space Program: Mission Two Payloads and Operations Plan

Stuart Buchan, Fergus Downey, Benjamin Hartig, Daniel Busan, Jacob Cook, Chelsea Tay, Kyle McMullan, Tristan Ward, Robert Howie, Phil Bland, Jonathan Paxman
Space Science and Technology Centre, School of Earth and Planetary Sciences, Curtin University
GPO Box U1987, 6845 WA, Perth, Australia;
stuart.buchan@curtin.edu.au

ABSTRACT

The second mission of Western Australia's Binar Space Program consists of three 1U CubeSats targeting a 2023 launch. Aiming to improve the platform for future missions, the primary purpose of Binar 2, 3 and 4 is on-orbit testing of radiation shielding alloys developed by CSIRO. In this first-of-its-kind experiment, all three simultaneously deployed Binar spacecraft will contain radiation sensing payloads to assess the efficacy of various compositions of Australian made radiation shielding alloys. Alongside this, hardware changes to the Binar platform are discussed, including deployable solar arrays, additional communications solutions, and a removable payload bay. The Iridium network will be leveraged to test its suitability for CubeSat targeted re-entry. Several software-based payloads are implemented, including on-board hardware emulation, enabling an industry partner to control the spacecraft in a demonstration of remote operations capability. An undergraduate student lead project will continue on from Binar-1 to see a star tracker flown for testing alternative methods of attitude determination. From a community perspective, strengthening the engagement between amateur radio operators and the Binar Space Program will be explored by expanding on what amateurs can do with on-orbit satellites. Lastly, autonomous agile mission planning will be tested through an on-board multipurpose simulation running on the dual-core flight computer.

Introduction

The Binar Space Program was conceived as an engineering undergraduate honours project at Curtin University in 2018. The initial scope of the project was to purchase COTS subsystems from satellite manufacturers to assemble Western Australia's first spacecraft. It became evident however that this could hinder the Program in three areas: cost, size and subsystem limitations. Due to this, the scope of the Binar Space Program shifted towards in-house development of spacecraft subsystems. Three years later in 2021, Binar-1 was launched into Low Earth Orbit in what was the first flagship mission of the Space Science and Technology Centre at Curtin University. Leveraging the high reusability of the Binar platform, the Binar Space Program is already developing the next iteration of CubeSats: Binar-2, 3 and 4. Deployed simultaneously from the International Space Station, the three 1U satellites will be carrying various payloads for experimentation in Low Earth Orbit.

The design of the mission two satellites and the payloads flown are heavily biased towards platform development of the Binar CubeSat, supporting future missions from the Space Science and Technology Centre. This paper will provide an overview of the mission two satellites, detailing the areas that differ from the Binar-1 CubeSat design, namely the development of deployable solar panels; a custom communications solution comprising UHF, Iridium and S-band; and a removable payload bay. It will then discuss the hardware and software payloads being flown; the former comprising an undergraduate student built star tracker and an in-house designed radiation sensor, and the latter a robotic arm emulator and an autonomous agile mission planning software tool. Alongside its usage as a form of communication with the CubeSat, the Iridium module is also discussed as a tool for tracking the satellite as it re-enters the atmosphere at the end of life. Finally, a brief concept of operations is given for the three satellites, detailing the mission timeline.

CubeSat Overview

Binar-2, 3 and 4 are three 1U CubeSats planned for a 2023 launch through the JAXA launch broker SpaceBD. Deploying simultaneously from the International Space Station, the three spacecraft will be inserted into a 400 km altitude, 51 degree inclined mission orbit. The spring assisted deployment method will see the three spacecraft slowly separate over the mission lifetime. Continuing with a strong focus on platform development for future reuse, Binar-2, 3 and 4 will extend upon Binar-1 to feature deployable solar panels, a custom communications system, and a removable payload bay, seen in Figure 1.

At the centre of the satellite (3) is the Binar CubeSat Core, a highly integrated custom circuit board that comprises a primary and redundant flight computer, a Global Positioning System (GPS), Attitude Determination and Control System (ADCS) (Magnetorquer based, 1), on-board storage and Electric Power System (EPS) in a 0.25U stack. The 1U frame is surrounded by solar cells, both face mounted (4), and deployable (2). Finally, at the base of the satellite is the removable payload bay (6), housing the radiation sensing primary payload (5).

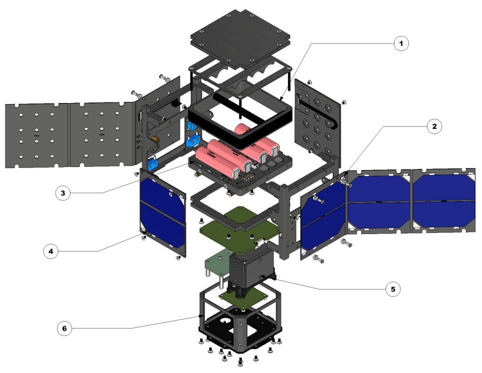


Figure 1: Exploded render of the mission two satellite design with key sections detailed.

Deployable Solar Panels

To satisfy the power budget for Binar 2, 3 and 4, additional solar cells need to be added to the existing 1U form factor. A deployable solar array system is implemented for the mission to meet current power requirements and to develop the in-house capability for supporting future missions with deployable arrays. The deployable solar array will have a chassis mounted panel that is attached to the mission two spacecraft in the same manner as the surface mounted cells on Binar-1, with the addition of two panels attached in series via rigid-flex PCBs on either side of the spacecraft, shown in Figure 2.

The deployable arrays will be held in a stowed configuration by burn wires, which when severed will allow for unfolding to occur. The primary actuation method for unfolding the solar panels will be nickel-titanium shape memory alloy. Compliant actuation was chosen due to its size, mass, and power efficiency, and also due to a reduced part count over classical actuation methods.

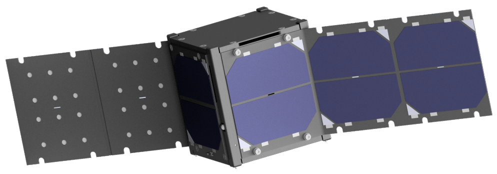


Figure 2: Render of the mission two spacecraft design with solar panels deployed.

Custom Communications Solution

Due to time restrictions and a lack of communications experience in the Binar Space Program, a COTS communications system was used on Binar-1. However, the solution purchased was very large, lacked redundancy, a repeatable deployment system, and had a black box software package and hardware design that was not easily modified to meet launch requirements. The integration issues that followed hindered some functionality of the spacecraft. Based on experiences from Binar-1, a custom communications solution has been developed for the mission two satellites that integrates three communication methods. The integrated system will contain a UHF, Iridium, and S-Band based communications system. The primary communications method will be the planar monopole deployable UHF. Based off of the OpenLST design, components used in the UHF solution have flight heritage. The UHF communications solution will operate in the amateur frequency band, satisfying our requirement for community engagement. The Iridium solution offers a much higher communications availability in comparison, not requiring the spacecraft to be over a ground station for communications. S-Band will be trialled for the mission as a test platform for future missions with higher data throughput requirements. A custom communications solution guarantees meeting requirements, and facilitates testing during the full mission life cycle. Moreover, a custom communications solution can be developed to efficiently use the space inside the spacecraft frame, increasing the space reserved for payloads.

Passionate about generating excitement for the space industry, the Binar Space Program plans to leverage the amateur radio platform on the mission two satellites to expand on what amateurs can do with spacecraft on-orbit. Conforming to the basic requirements of using the amateur radio band, the spacecraft will frequently generate data on a frequency accessible to radio amateurs, allowing them to design and build their own ground stations to listen to Binar-2, 3, and 4 as they pass overhead. For amateurs that have a license to transmit, the CubeSats will also allow amateurs to send packets of data for temporary storage to be recalled later. Due to the importance of the SatNOGS network in locating Binar-1, these amateurs are encouraged to link their ground stations into the SatNOGS network to listen to other satellites, and to recall their stored data packets from other ground stations around the world.

Removable Payload Bay

The payload bay is designed to provide a separation between the core spacecraft systems and the experiments being flown, adding abstraction and removing hardware-enforced links between the core bus and payloads. This removes the need for the Binar CubeSat Core to be modified for each launch. This will ensure the Binar bus can be tested, verified and locked into a working design that can be reused for future launches. The addition of the payload bay enables payloads to be assembled and tested separately to the main CubeSat to ensure functionality before being integrated into the core system. This design choice was influenced by the BinarX program, which will see Western Australian school students developing payloads to be flown on Binar spacecraft.

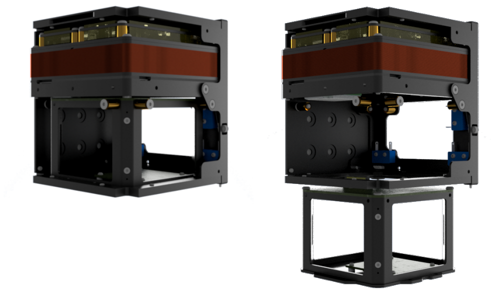


Figure 3: Render of the payload bay in the stowed and removed configuration.

The payload bay is designed to be a rigid unit that is inserted into the CubeSat from below. It interfaces with the spacecraft motherboard through a custom designed payload adaptor PCB that mates with the motherboard when inserted into the CubeSat frame. Each individual payload will connect to the payload adaptor board using a locking cable connector. The following section provides details about the payloads that will be flown in the payload bay on the Binar-2, 3 and 4 missions.

Payload Design

Radiation Detection

The primary objective of the Binar-2, 3 and 4 satellites is on-orbit testing of radiation shielding alloys developed by the Commonwealth Scientific and Industrial Research Organisation (CSIRO). It is well understood that spacecraft in orbit around Earth are subject to radiation originating from the Sun. The Earth's magnetic field traps electrons, protons and heavy ions in common spacecraft orbits, and radiative particles are also known to traverse through these orbits.[1] As spacecraft move further away from the protection Earth provides from ionisation radiation, the dosage increases.

The electronics that enable spacecraft are susceptible to the detrimental effects of the radiation present in space. Total radiant exposure and time-dependent radiant flux are two major areas spacecraft electronics are susceptible to.[1] In high risk components such as commercially available off the shelf integrated circuits (ICs), these areas manifest in single event effects (SEE) and total ionising dose (TID) radiation events. TID refers to the damage components face from long-term exposure to a radioactive environment. This impacts areas such as device voltage threshold levels and component leakage currents, leading to irreparable faults. Conversely, SEEs refer to the short term damage ICs can suffer from transient events. In these situations, energy deposition from a radiative particle can cause bit-flip errors that effect software runtime.[2] These faults often present through degraded integrity of non-volatile flash memory, impacting data storage recall, and also through program memory corruption, potentially resulting in a processor executing erroneous instructions that can lead to system failure.[3]

To combat the detrimental effects radiation has on sensitive electronic components, shielding is often employed externally to a spacecraft, or sometimes internally directly around specific components.[1] Traditionally, aluminium is used in spacecraft shielding due to its density properties. However, recent advancements into the usage of composite materials in radiation shielding have shown an increase of shielding efficacy and a reduction in mass, whilst still providing high mechanical strength[4].[5] Due to this, composite materials are being viewed as a viable alternative for spacecraft radiation shielding. Device sensitivity to radiation events vary largely depending on the component used. In the same orbit, the SEE rate can increase by several orders of magnitude in technologies more susceptible to radiation.[6] With compositions of alloys and epoxy showing promise in the future of spacecraft radiation shielding, the primary science experiment on mission two will determine the efficacy of newly developed compositions both for radiation protection close to home, and as the Binar Space Program explores deeper into the solar system.

The purpose of the radiation detection payload is to test the efficacy of the shielding material developed by CSIRO. All three of the CubeSats will contain a radiation sensor developed in-house, but will have varying compositions of the structure surrounding the payload. The payload is expected to provide input into the decisions regarding the structural design of Binar CubeSats as the program targets missions beyond Low Earth Orbit. The payload consists of three independent radiation detection methods, and the varying shielding compositions. Figure 4 highlights the detection methods of the payload, as well as the supporting circuitry.

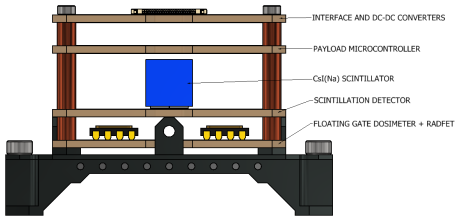


Figure 4: Cross sectional view detailing the radiation detector payload.

Shielding

Whilst the radiation sensors will be identical between the launch two spacecraft, the shielding surrounding the payload will vary. Binar-2 will operate as a control, with Binar-3 and Binar-4 offering varying compositions to assess the efficacy of the radiation shielding methods in LEO.

The control shielding will comprise of 3mm aluminium plates, being the same grade aluminium used for the spacecraft frame. As the payload is to influence the decisions made for the composition of future spacecraft frames, this is to represent a spacecraft structure with no radiation shielding. The impinging radiation data collected from this will be compared against a shielding composition of doped aluminium, and against dual layer aluminium and epoxy plates. Whilst the doped aluminium may provide better total radiation shielding protection, the dual layer shielding is being investigated to ascertain shielding to mass ratio performance.

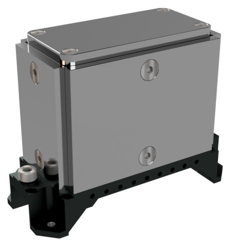


Figure 5: A render of the shielding surrounding the radiation detection payload.

Scintillation Detector

The scintillation detector is the primary method of measuring impinging radiation on the payload. It comprises a sodium-doped caesium iodide crystal fixed to the top of four silicon photomultipliers (SiPM). When ionising radiation passes through the crystal, it re-emits the absorbed energy in the form of light to be detected by the SiPMs. When struck with the emitted photons, the SiPMs will generate a voltage to be measured by the payload microcontroller. This voltage is directly proportional to the energy of the ionising radiation. Due to this, the voltage recorded by the payload microcontroller can be matched to the energy range of the expected radiating particles, and hence discern what particle produced the scintillation. This can provide insight into if the radiation received is due to pass-throughs of the radiation shielding, or if the shielding itself is producing secondary ions.

Floating Gate Dosimeter

The floating gate dosimeter is a digital radiation sensor located on the secondary sensor board of the payload seen in Figure 4. Unlike the scintillation detector, this sensor cannot determine the particular impinging particle. The detection method sees the implementation of a floating gate capacitor. The floating gate normally exists in a charged state, of which it can hold indefinitely until a dose of ionising radiation is received. The impinging radiation causes the floating gate to discharge at a proportional rate to the dose received. After reaching a discharge threshold, the floating gate is recharged to its nominal state. Using the discharge rate coupled with the charge cycle count, the instantaneous dosage intensity can be inferred.

RadFET

The radiation-sensitive metal-oxide-silicon field-effect transistor (radFET) payload is used to determine the total amount of radiation received over the lifetime of the mission. A reverse bias current is forced into the radFET and the voltage is read at the source. The amount of gate leakage current is permanently changed as radiation impinges the sensor,[7] allowing the computer sampling the voltage to infer the total amount of radiation that has been received.

Targeted Re-entry

Alongside offering redundancy in communications, flying an Iridium module on the mission two spacecraft enables the exploration of small satellite re-entry systems. Leveraged through a dedicated operational mode, Binar-2, 3 and 4 will de-activate unnecessary systems and focus power towards transmitting Iridium beacons at high frequencies. These re-entry beacons comprise power, temperature and location information which the ground station will use to monitor the satellite entering the atmosphere. Testing of the Iridium based tracking system on the mission two spacecraft will be a precursor to creating a future targetable re-entry system, with the eventual goal of re-entering a Binar spacecraft above the Western Australian Desert Fireball Network.[8] Early re-entries over the network will see optical tracking of the satellite as it ablates in the atmosphere. Future stretch goals involve the implementation of heat shielding and a landing system to attempt to land a CubeSat in the Australian desert.

Robotic Arm Emulation

Aiding in the bridging of industry and research, the Binar Space Program has an ongoing collaboration with the Fugro Space Automation AI and Robotics Control Complex (SpAARC). The SpAARC is developing unique remote operation capabilities to support space robotics projects both within Australia and internationally. On the mission two satellites, the collaboration with the Binar Space Program will see Fugro SpAARC remotely operating an emulation of a robotic arm from their ground station in Perth, Western Australia. As a case study, a solar panel is to have failed to deploy correctly, requiring the use of an on-board robotic manipulator to aid the deployer in completing its full actuation.

Requesting the solar panel deployment angle from Fugro SpAARC, the spacecraft will generate, log, and return a random angle to the ground station. A task can then be scheduled to actuate the virtual payload, involving interpolation of the three degrees of freedom simulation. The updated virtual manipulator state and the solar panel angle will be logged on the spacecraft before being returned to Fugro SpAARC when queried and displayed in real-time on their remote operations front-end software.

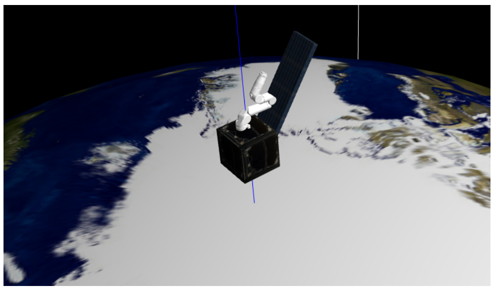


Figure 6: Screenshot from the Fugro SpAARC remote operations tool, showing a manipulator and a deployable panel on-orbit.

The simple experiment aims to demonstrate the capabilities being developed in Western Australia, supporting the growth of local space industries.

Star Tracker

Passionate about lowering the barrier to entry of working on space missions in Australia, the Binar Space Program will be flying a star tracker payload built by undergraduate students at Curtin University. The aim of the payload is to test the hardware the could be used on future Binar missions that extend beyond LEO. The hardware will consist of a lens, image sensor, microcontroller, supporting electronics and a mechanical structure. The microcontroller will be running star tracking algorithms written by the students to perform attitude determination. The payload is expected to produce lessons learned, and some information about the suitability of COTS lenses, CMOS sensors and microcontrollers for star tracking applications.

Digital Twin

The digital twin payload aims to create a virtual replica of the Binar spacecraft that can be used for on-board simulations. The goal is to develop a multipurpose simulation in which subsystems possess knowledge about their own state, the environment they operate in, and how they integrate with other submodules in the system. On the mission two satellites, the payload will recommend changes to the mission plan based on on-board power consumption. Figure 7 shows a high level flowchart of the digital twin implementation.

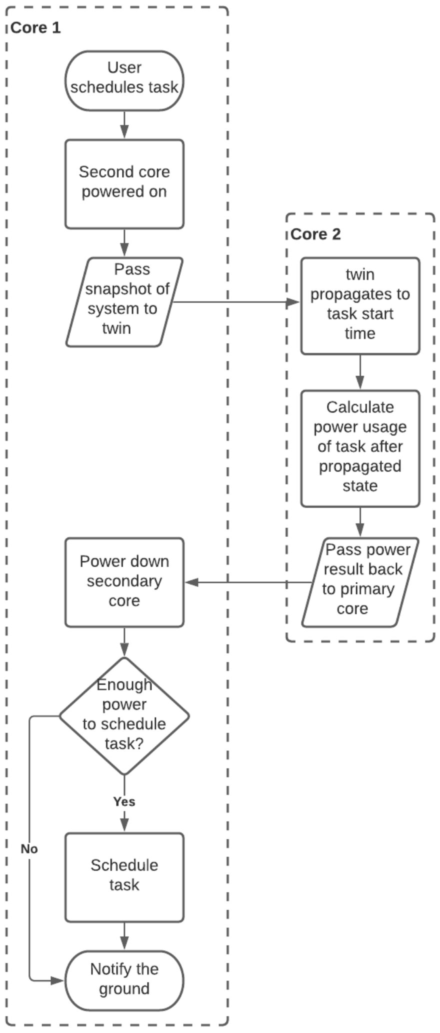

Core 1
User schedules task
Second core powered on
Pass snapshot of system to twin
Core 2
twin propagates to task start time
Calculate power usage of task after propagated state
Pass power result back to primary core
Power down secondary core
Enough power to schedule task?
Yes
No
Schedule task
Notify the ground

Figure 7: Digital twin virtual payload proposed functionality diagram.

One core of the flight computer will be dedicated to running the simulation to ensure that possible intense loads from interpolations will not impact the flight logic of the spacecraft. When the ground station schedules a task for the spacecraft to complete, the flight logic will power the second core of the flight computer on, passing a snapshot of the satellite into the simulation. This snapshot will contain current readings of all spacecraft systems, inclusive of on-board power usage and thermal information. Whilst the main core continues executing the flight logic, the second core will propagate forward on-orbit until the start time of the scheduled task. The spacecraft will then estimate the total power usage, including the battery heaters, and the expected continued usage of on-board peripherals. The flight logic is then notified of the outcome of the simulation, and is able to determine if it is safe to progress with task execution, or recommend another time slot in an agile manner.

The models used by the digital twin payload during simulation were derived from testing in lab-based environments. A future goal of the digital twin payload is to facilitate on-orbit inference of pre-trained machine learning models using data obtained from the Binar satellites on-orbit.

Concept of Operations

A high level overview of the operation modes and corresponding state transitions for the mission two spacecraft can be seen in Figure 8. The red arrows represent automated transitions, whilst the green show manual transitions.

Post-deployment from the ISS, the three spacecraft will enter a separation sequence where the EPS will power the spacecraft and begin a 30 minute timer. This is to satisfy the ISS deployment requirement of refraining from deployment actuation until a safe distance has been achieved. After this timer has lapsed, the planar monopole UHF antenna will deploy followed by the solar arrays. Following a successful deployment, the spacecraft will transition into the bootloader to run a system health check. Verifying nominal system operations, the spacecraft will transition into a low power mode with high frequency, long duration communications to aid in locating the spacecraft from the ground station. A telecommand received from the ground station will permit the spacecraft to transition into the nadir pointed application mode where the spacecraft systems will be brought online. Two weeks post deployment from the international space station, the spacecraft will begin to achieve it's on-orbit objectives, as shown in Figure 9.

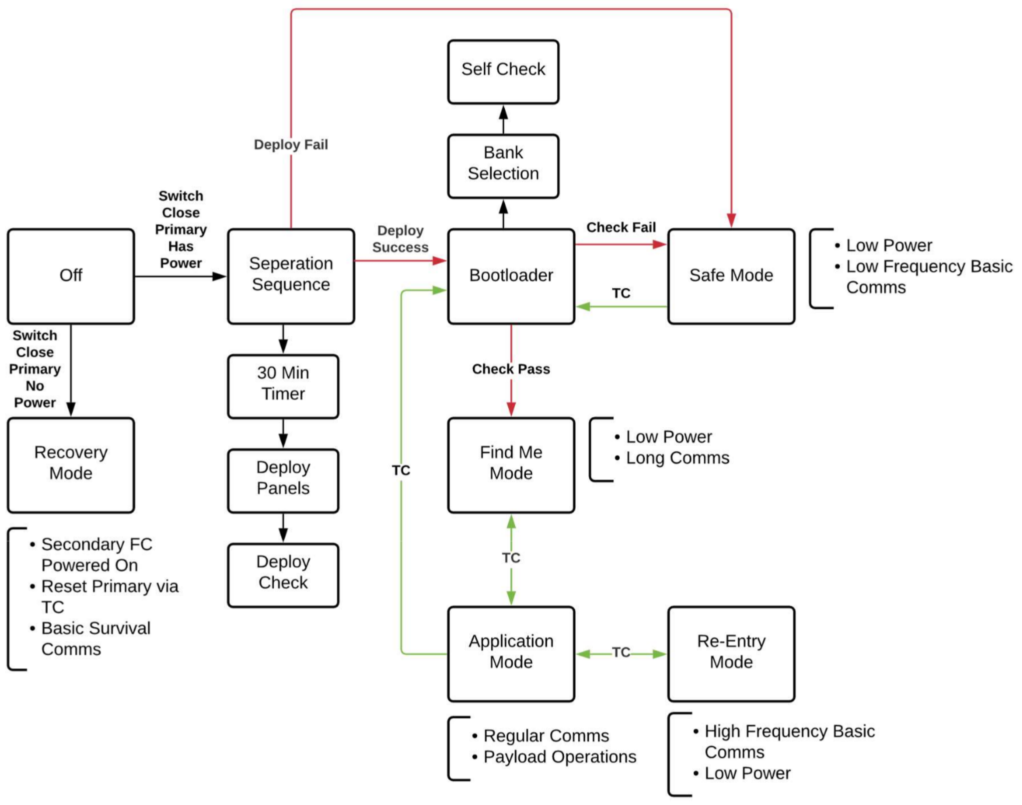


Figure 8: Binar-2,3 and 4 Operation Modes.

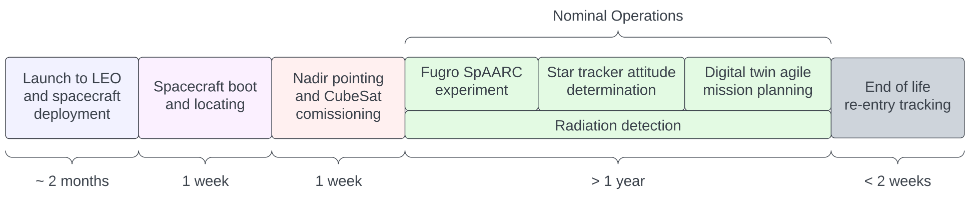

Nominal Operations
Launch to LEO and spacecraft deployment
Spacecraft boot and locating
Nadir pointing and CubeSat comissioning
Fugro SpAARC experiment
Star tracker attitude determination
Digital twin agile mission planning
Radiation detection
End of life re-entry tracking
~ 2 months
1 week
1 week
> 1 year
< 2 weeks

Figure 9: Proposed timeline of the Binar-2, 3 and 4 CubeSats.

To maximise the chance of satisfying the industry collaboration requirements, the Fugro SpAARC remote operations experiment will be the first on-orbit objective. During an overhead pass, the Binar Space Program will hand over control of the satellites to the SpAARC ground station facility in Perth, Western Australia. This experiment will be completed multiple times with the three satellites to demonstrate the robustness of their remote operations capability. Following completion of the demonstration, the three Binar spacecraft will be tasked with using the star tracker payloads to capture images of the visible stars. These images will be used on-board the spacecraft in attitude determination to be referenced against location information from the GPS. As the three CubeSats are expected to slowly drift over the course of the mission, the images captured will provide a comprehensive view of stars visible from LEO and hence will be also downlinked for use in future testing. After having sampled a sufficient catalogue of stars, the satellites will begin the agile planning stage of the mission using the on-board digital twin. The digital twin payload is to suggest changes to scheduled task times if the on-board simulations predict insufficient available power to safely complete a task. To force this situation to occur, high power tasks will be scheduled for periods of the orbit where the CubeSats are in Earth's shade, not charging, and using battery heating. To verify the reasonableness of the simulations recommendations, an override will be granted to allow a task to run regardless of the agile recommendations. Observing the spacecraft booting into safe mode will demonstrate the recommendation was sound, showing the plausibility of using on-orbit agile mission planning in more complex cases.

The nominal operations stage of the mission timeline is expected to last for at least one year. Whilst sequentially operating the previously mentioned payloads, the primary radiation sensing payload will be constantly operational, downlinking measured radiation data that leaks through the shielding flown. This information will be periodically transmitted along with general housekeeping data. When green time exists within the nominal operations schedule, the Binar Space Program will work with the amateur radio community to listen and transmit to Binar-2, 3 and 4.

Two weeks prior to the end of life, the CubeSats will be transitioned into the re-entry mode via a telecommand from the ground station. This mode will orient the spacecraft such that the deployable cells create sufficient drag to aid in orbit degradation. This mode will see the spacecraft enter a low power mode, conserving energy to support high frequency communications over the Iridium network to track re-entry in the atmosphere.

Conclusion

Binar-2, 3 and 4 comprise the second launch of the Binar Space Program, with a strong focus on platform development for the future. The three 1U CubeSats will be primarily testing radiation shielding compositions created by the CSIRO. The varying compositions used will be assessed based on the three radiation detection methods flown, showing types of radiation received, dosage intensity, and total ionising dose received. The data collected from this experiment will aid in determining the efficacy of shielding on CubeSats, enabling future Binar missions that extend beyond LEO.

Platform development will also be explored on mission two through the testing of deployable cells, redundant communications, and the unique removable payload bay. The deployable cells on the spacecraft will provide an extra power margin over the surface-mounted only cells on Binar-1, supporting growing payload needs as the platform is reused in the future. Flying multiple communications solutions allow for a backup communication method through the Iridium network, with S-band also supporting testing of higher bandwidths as data throughput requirements increase in the future. The Iridium communications solution will also be flown, with a focus on transmitting telemessages from the spacecraft during spacecraft re-entry to aid in tracking. Second to this are software and hardware payloads that will help the Binar Space Program develop necessary systems for future missions planned from the Space Science and Technology Centre at Curtin University.

As the main goal of the platform is to mature towards a static design where only the payload changes between launches, the removable payload bay will be tested on the mission two satellites carrying an in-house developed radiation sensor, and a star tracker camera. The radiation sensor will provide information on the radiation that passes through the payload-surrounded shielding, whilst the undergraduate star tracker payload will provide preliminary data for the development of future deep-space attitude determination as Binar spacecraft strive further from Earth.

Software payloads are used on the mission two satellites for facilitating industry engagement, and for agile mission planning. The ongoing collaboration with Fugro SpAARC will see the inclusion of a robotic arm emulator to be controlled by Fugro ground station operators, demonstrating the use of their remote operation facility. When commanded to schedule a task for completion, the spacecraft will use the digital twin software to propagate on-orbit and assess the power state of the spacecraft for safe task execution.

Finally, the mission two spacecraft will expand on what amateur radio operators can do with a spacecraft on-orbit. Whilst supporting unencrypted access to beacons transmitted from the spacecraft, the satellites will also allow radio amateurs with a transmission license to store small data packets on the spacecraft that can be downlinked using the SatNOGS network.

The Binar Space Program intends to launch satellites frequently to support the program goal of space systems education at Curtin University. Binar-2, 3 and 4 provide a test platform in the constant development of the Binar CubeSat bus. The data collected from the operations of the satellites and the payloads they are flying will influence the decisions made for the next Binar CubeSats, and aid in the eventual development of satellites that venture beyond LEO.

Acknowledgements

This research was supported by an Australian Government Research Training Program (RTP) Scholarship and received institutional support from Curtin University, Fugro SpAARC and AROSE.

References

[1] E.G. Stassinopoulos and J.P. Raymond. The space radiation environment for electronics. *Proceedings of the IEEE*, 76(11):1423–1442, November 1988. Conference Name: Proceedings of the IEEE.

[2] Keith E. Holbert and Lawrence T. Clark. Radiation Hardened Electronics Destined For Severe Nuclear Reactor Environments. Technical Report DOE-ASU–NE00679, 1238384, February 2016.

[3] Paul Madle. STM32H7 Radiation Test Report. Technical report, Open Source Satellite, May 2021.

[4] Masayuki Naito, Satoshi Kodaira, Ryo Ogawara, Kenji Tobita, Yoji Someya, Tamon Kusumoto, Hiroki Kusano, Hisashi Kitamura, Masamune Koike, Yukio Uchihori, Masahiro Yamanaka, Ryo Mikoshiba, Toshiaki Endo, Naoki Kiyono, Yusuke Hagiwara, Hiroaki Kodama, Shinobu Matsuo, Yasuhiro Takami, Toyoto Sato, and Shin-ichi Orimo. Investigation of shielding material properties for effective space radiation protection. *Life Sciences in Space Research*, 26:69–76, August 2020.

[5] Masayuki Naito, Hisashi Kitamura, Masamune Koike, Hiroki Kusano, Tamon Kusumoto, Yukio Uchihori, Toshiaki Endo, Yusuke Hagiwara, Naoki Kiyono, Hiroaki Kodama, Shinobu Matsuo, Ryo Mikoshiba, Yasuhiro Takami, Masahiro Yamanaka, Hiromichi Akiyama, Wataru Nishimura, and Satoshi Kodaira. Applicability of composite materials for space radiation shielding of spacecraft. *Life Sciences in Space Research*, 31:71–79, November 2021.

[6] W. L. Bendel and E. L. Petersen. Proton Upsets in Orbit. *IEEE Transactions on Nuclear Science*, 30(6):4481–4485, 1983.

[7] Andrew Holmes-Siedle and Leonard Adams. RADFET: A review of the use of metal-oxide-silicon devices as integrating dosimeters. *International Journal of Radiation Applications and Instrumentation. Part C. Radiation Physics and Chemistry*, 28(2):235–244, January 1986.

[8] P. A. Bland, P. Spurný, A.W. R. Bevan, K. T. Howard, M. C. Towner, G. K. Benedix, R. C. Greenwood, L. Shrbený, I. A. Franchi, G. Deacon, J. Borovička, Z. Ceplecha, D. Vaughan, and R. M. Hough. The Australian Desert Fireball Network: a new era for planetary science. *Australian Journal of Earth Sciences*, 59(2):177–187, March 2012. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/08120099.2011.595428.

APPENDIX D

Coauthored Conference Abstracts

Binar Space Program: Binar-1 Results and Lessons Learned

Fergus W. Downey [1], **Stuart R. G. Buchan** [1], Benjamin A. D. Hartig [1], Daniel C. Busan [1], Jacob Cook [1], Phil A. Bland [1], Jonathan Paxman [2]

[1]Space Science and Technology Centre, Curtin University, Perth, Australia

[2]Department of Mechanical Engineering, Curtin University, Perth, Australia

This conference abstract was presented at the 19th Australian Space Research Conference (ASRC 2019) in Adelaide, Australia. Reason for inclusion — I was the software lead for the Binar-1 spacecraft, and also assisted with mechanical design and satellite construction. I was a core member of the failure analysis panel, contributing in discerning what went wrong with the CubeSat on orbit. I also contributed to the lessons learned from the results of the mission.

SSC22-WKII-06

Binar Space Program: Binar-1 Results and Lessons Learned

Fergus Downey, Stuart Buchan, Benjamin Hartig, Daniel Busan, Jacob Cook, Robert Howie, Phil Bland, Jonathan Paxman
Curtin University
Space Science and Technology Centre, School of Earth and Planetary Sciences, Curtin University, GPO Box U1987, 6845 WA, Perth, Australia; (+61) 0488 442 133
Fergus.downey@curtin.edu.au

ABSTRACT

The Binar Space Program is a recently formed space research and education group part of the Space Science and Technology Center at Curtin University in Western Australia. Recently launching the first CubeSat from the state, Binar-1, the team is making steps towards creating a sustainable mission schedule for research and education. The Binar-1 mission primary objective was to demonstrate the custom designed systems made by PhD students and engineers at the university. The main technology being demonstrated was the integrated Binar CubeSat Core, which compacted the Electrical Power System, Attitude Determination and Control System, and flight computer system into 0.25U. Alongside this, the team also aimed to learn about end-to-end spacecraft mission design and engage with the public to build an understanding of the importance of space industry and research in the country. Binar-1 was deployed from the International Space Station on the 6th of October 2021, and initially was silent for 15 days until the Binar team was able to make contact by enabling a secondary beacon. This paper will present the Binar-1 mission including the custom design, operations, failure analysis, and results before finally summarizing the lessons learned by the team while flying Western Australia's first space capability.

INTRODUCTION

With the foundation of the Australian Space Agency (ASA), a new wave of space industry and research has begun in the country. One of the research groups is the Space Science and Technology Center (SSTC) located at Curtin University in Western Australia. With a history in global fireball entry tracking and space situational awareness technologies[1], the research center formed a new branch in the Binar Space Program. This program aims to help develop the skills necessary for working in the space industry by performing valuable space research at the university with frequent CubeSat missions. The first mission performed by the Program, Binar-1, was a technology demonstrator mission that tested the custom designed systems put together by a team of PhD students and engineers.

The design of Binar-1 took inspiration from the first CubeSats developed and launched by universities, focusing on using custom design systems rather than purchasing Commercial Off the Shelf (COTS) solutions. This design decision was made for many reasons; however, the main purpose was to reduce the cost of future missions and build capabilities which can be upscaled to more complex space missions. This technology skill growth has been vital for the team as it now works towards its future missions in Binar-2, Binar-3, and Binar-4.

Having first been conceptualized in the middle of 2018, this paper will present the complete lifecycle of Binar-1. First, the Binar-1 mission goals and design will be detailed. Next, it will discuss the operations, recovery process, and results of the mission. Finally, it will provide a summary of the lessons learned and how these lessons will be implemented into future Binar missions.

BINAR-1 MISSION

Binar-1 was launched from cape Canaveral on the 29th of August 2021 onboard a SpaceX Falcon 9 rocket as a ride share on the International Space Station (ISS) commercial resupply mission CRS-23. The CubeSat was then deployed along with 2 others (Maya-3 and Maya-4) on the 5th of October from the Kibo module and JEM Small Satellite Orbital Deployer (J-SSOD). The launch was coordinated with the Japanese Aerospace Exploration Agency (JAXA) through SpaceBD, a commercial space company in Japan. Figure 1 is a photo taken from on-board the ISS of the CubeSat deployment. Binar-1 can be seen in the top right of the image with the Earth and ISS solar panels seen in the background.

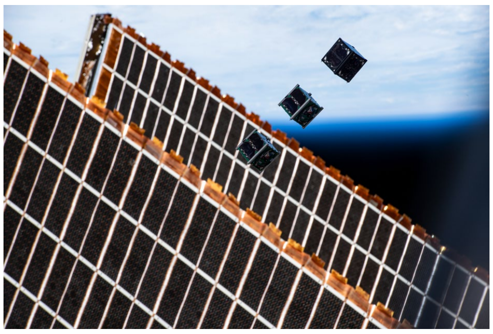


Figure 1: Binar-1, Maya-3, and Maya-4 just after deployment from the J-SSOD with the ISS solar arrays and Earth pictured in the background.

The main objectives of the Binar-1 mission were to:

- Demonstrate the custom designed systems created by the Binar Space Program,
- Educate staff and students about end-to-end spacecraft mission design, and
- Spread awareness about the importance of a space sector in Western Australia.

Alongside these objectives, the CubeSat was also flown with two secondary payloads: an undergraduate student led star tracker, and a high-resolution Earth imagery camera.

Of the custom designed systems being tested, the primary novel system is the integrated Binar CubeSat Core (BCC). Contained inside the 0.25U package is an Electrical Power System (EPS), Attitude Determination and Control System (ADCS), and flight computer system. Also, custom designed by the team was the Binar structure, the Binar Software Framework, and integration method for the communications system and payload cameras.

BINAR-1 DESIGN

The design, testing and integration of Binar-1 took place over the course of 3 years at Curtin University. The initial concept for the design was to make up the satellite from CubeSat COTS systems to meet the mission objectives. This process educated the team on what was typical for CubeSat missions and how to start its own design process. The decision to move to a custom design was made from observations of the COTS solutions architectures and the benefits that could be achieved from a custom designed system. While the systems were modular and made to work together across suppliers, the team noticed that the systems available were not able to achieve the team goals. The system solutions are larger than needed and limited to the design, making it hard for modification to be made without major intervention. Also, the cost of purchasing the systems is greater than if the hardware is custom designed. Moreover, this benefit of custom designing the systems will help the Binar team to reduce future cost and build skills for designing more complex systems in future missions. As such, the team decided to go forward with a custom design due to the many benefits it had alongside the ability to modify and compact the design.

As a result, the custom design of Binar-1 included the BCC, Binar structure, Binar Software Framework, and the payloads. Alongside these systems was a COTS communication system. The system block diagrams for Binar-1 are presented in Figure 2, separated into its power and signal connections.

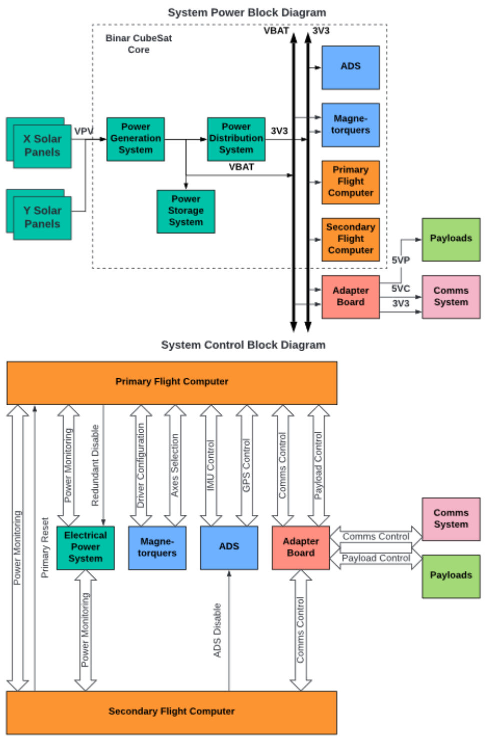


Figure 2: Simplified Binar-1 system block diagrams.

Binar CubeSat Core

The integrated BCC is the primary technology demonstrator objective of Binar-1. Containing the integrated EPS, ADCS, and flight computer system, the goal of the design was to compact these systems to make more space for payloads. Learning from the initial Binar-1 design which used COTS systems, the primary requirements for the BCC were based around compactness, safety, reusability, reliability, and testability. The complete design was achieved using computer aided design software, allowing for optimal placement of electronic parts and mechanical structures.

The EPS found on the BCC contains the typical subsystems of an EPS including a power generation subsystem, power storage subsystem, and power distribution subsystem. The power generation subsystem consisted of two Maximum Peak Power Controllers (MPPC) which are supplied from solar panels on the X and Y faces of the CubeSat. The solar panels were assembled by the Binar Space Program, making modifications to existing assembly methods to optimize and simplify the process for 1U panels[2]. Connected to the power generation subsystem, the power storage subsystem consisted of four lithium-ion 18650 battery cells in a 2S2P configuration. To meet the mission and launch safety requirements, the power storage subsystem also included battery heating and an ISS launch qualified battery protection system. Finally, the distribution subsystem consisted of a dual redundant 3.3V converter to power the remaining systems on the BCC. The distribution subsystems for the payloads and communications system were found on the adapter board which connected the BCC to the PC104 connectors used by the COTS communication system.

The ADCS contained on the BCC consisted of an Inertial Measurement Unit (IMU), Magnetometer, Global Positioning System (GPS), and a 3-axes magnetorquer. Combining the complete system into the BCC, the EPS and flight computer system were able to integrate with the ADCS from the beginning of development. This process helped to remove integration issues and increase confidence in the design. The IMU and magnetometer were used for feedback in the attitude control system and operation of the magnetorquers. Integrated with the batteries inside the 0.25U BCC, the magnetorquers consisted of two X and Y axes iron core magnetorquers and one large Z axis vacuum core magnetorquer. These coils are all operated by driver circuits located on the BCC.

The flight computer system consists of two flight computers, one primary and one secondary, as well as an external memory device. The system uses the primary flight computer for all flight operations and control, relying on the secondary flight computer only if the primary flight computer fails. The external memory device provides 4GB of on-board payload and system log storage for the primary flight computer which can be requested from the ground via telecommand. Housed around the BCC is part of the Binar structure including the RBF bracket, magnetorquer mount and top cap. These all fit inside the rest of the Binar structure detailed in the following section. A BCC that was used for lab testing is presented in Figure 3. Due to the reduced hardware cost of the BCC, the team was able to assemble multiple versions of the core for integration testing and verification.

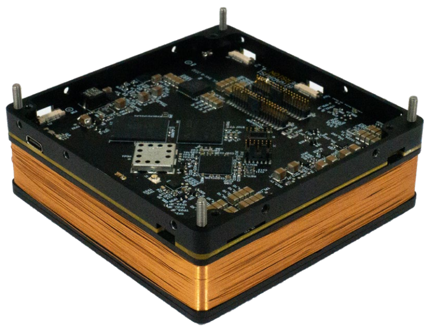


Figure 3: A flight model equivalent of the Binar CubeSat Core (BCC) that was used for lab testing.

By combining three of the main systems of a CubeSat into a single core, the Binar team was able to meet its design requirements. The compact design compared to the original COTS Binar design has allowed the team to increase its payload space, which in turn will benefit the team in future missions when reused. Safety has been implemented, protecting the batteries from accidental shorts or over charge and over discharge conditions. Reliability was implemented with extensive integration testing during design iterations, and simple additions of redundancy were possible. The testability of the design is also made easier through the combination of the Binar Software Framework which was designed to work with the BCC. This direct access to BCC is what will enable the Binar Space Program to fly more complex payloads on future missions.

Binar Structure

The Binar structure was designed by the Binar team to meet the launch requirements of the JEM Payload Accommodation Handbook Vol 8. Rev D[3]. The structural design consisted of two rail halves which were connected to the BCC in the center. The antenna and payload were then used to constrain the satellite at the top and bottom. This design was able to meet the launch requirements due to the tight tolerancing of the BCC holding the satellite together. Other parts of the structure included those found in the BCC and the payload mounting plate. The mounting plate was designed to also act as a counter mass for the BCC to move the center of gravity as close to the geometric center of the satellite as possible, assisting with the attitude control system. The exploded view (Figure 4) presents the structural design of Binar-1. The exploded BCC seen in the center connects to the adapter board and transceiver forming the stack. This was then fastened to the two main rail halves which form the structure. The antenna, payload and solar panels were then fastened to the six sides of the CubeSat.

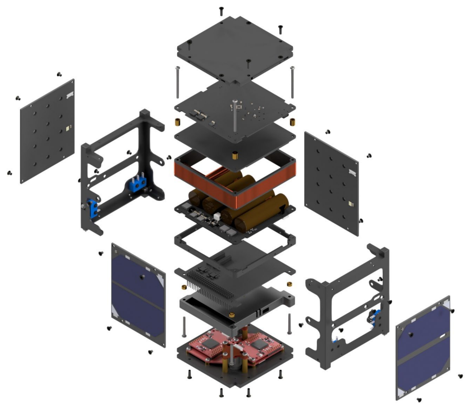


Figure 4: Exploded view of Binar-1

Binar Software Framework

Matching the hardware design goals of creating a safe, reusable, reliable, and testable platform, the Binar Software Framework was written to enable rapid mission concept-to-orbit. Specifically structured around areas that commonly contribute to software related mission failures, namely insufficient software testing, lack of documentation, unsafe code reuse and cursory code review, the Binar Software Framework has provided the Program with a reliable code base that can be reused on future Binar missions.

Comprising of flight application code, a hardware abstraction layer, utilities, resource manager, and board support layer, the codebase adheres to an abstracted software design that enables hardware to be changed with only minor modifications to the software. Moreover, the loose module coupling within the abstracted design allows hardware dependencies to be broken during unit testing to increase the testability of the code base.

Communications System

The communications system on Binar-1 was the only system that was supplied by a COTS provider. The decision to use COTS for this part of the satellite was based on the team size and amount of experience. At the time of decision, the small team was only made up of PhD students and part-time engineers. With significant focus on developing the BCC, Binar structure, and Binar Software Framework, the team was unable to commit the time to learn about communications design. As such it was decided that the best course of action for the success of Binar-1 was to purchase a COTS communications system.

The data requirements for transmitting the flight logs and payload images were achievable with a UHF communications system. The team decided to use the COTS system recommended in the initial COTS design of Binar-1 to meet this requirement. The deployed antenna can be observed in Figure 5. As the BCC was not based on the PC104 standard to optimize its space efficiency the team used an adapter board to connect the two together. This board was also used to adapt the BCC to the payloads.

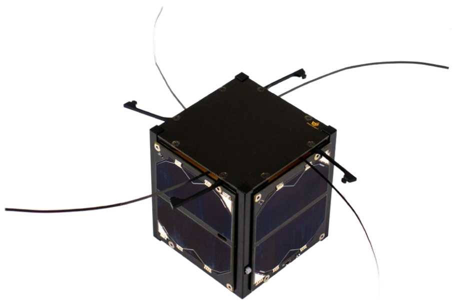


Figure 5: The Binar-1 engineering model with the antennas deployed.

Payloads

Two payloads were flown on Binar-1 including a high-resolution camera for Earth imagery of Western Australia and a student led star tracker camera for developing more precise attitude determination capabilities at the Binar Space Program. Both payloads were originally planned as primary mission objectives, however, after the change to a custom design, the purpose of the payloads shifted to demonstrating the functionality of the BCC and its ability to operate payloads. The payloads were integrated together with a custom PCB that connected to the same adapter board used by the communications system. The two cameras were both COTS cameras, the first being selected by the Binar team to give the best resolution for Earth imagery in the available payload space (80m/pixel), and the other being a low-resolution camera that was selected from a range of cameras tested by the team of undergraduate students. The final payload system was mounted to the bottom of the satellite.

Ground Segment

As part of the mission plan for Binar-1, the team also needed to develop its own ground station (Figure 6) and operation software for the Binar-1 mission. Made from a combination of custom and COTS components the Binar ground station was built and placed on top of the engineering building on the Curtin University Bentley campus. The operating software was designed as part of a collaboration with Fugro Space Automation, AI, and Robotics Control Complex (SpAARC). The complete design was tested on existing satellites in LEO in preparation for the deployment of Binar-1.

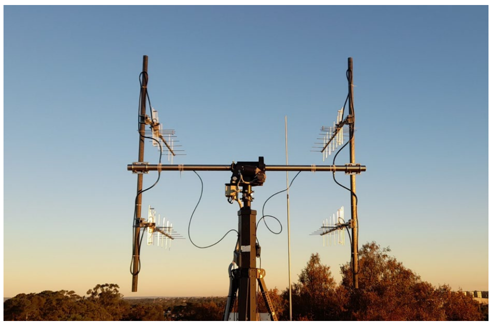


Figure 6: The Binar ground station located at the Curtin University Bentley campus in Western Australia.

INTEGRATION AND TESTING

Integration and testing of the Binar-1 flight model was conducted using the facilities available at Curtin University. Testing was separated into two parts, being the integration and testing performed on the custom designed systems and the testing performed to meet the launch requirements. The regulatory requirements necessary for launch are documented in the JEM Payload Accommodation Handbook Vol 8. Rev D. which included battery safety testing, vibration testing, and interface verification testing.

Binar Testing Procedures

To verify the functionality of the custom Binar-1 platform the team developed testing processes throughout the course of the design. One of the main testing processes that was developed was the integration testing process of the BCC. A benefit of the integrated design was the straightforward process of verifying the connections and operation software for each system on the integrated core. This helped to build confidence in the hardware and software design as faults were identified and removed early in the custom design process.

Typical to CubeSat testing programs, the team performed thermal vacuum testing using a modified vacuum chamber at Curtin University (Figure 7). The modification included a liquid nitrogen shroud and electric heater which can reach surface temperatures of -100°C to +150°C. Testing in this chamber was also verified with a test using the Wombat XL located at the National Space Test Facility (NSTF), Australian National University (ANU), in Canberra. The verification test was done before the assembly of the flight model using the Binar-1 engineering model. Results from this testing was important to verify the modified vacuum chamber due to the COVID-19 pandemic restricting the ability of West Australians to travel without quarantine. This meant the team could not return to Canberra to perform the vacuum testing again with the flight model.

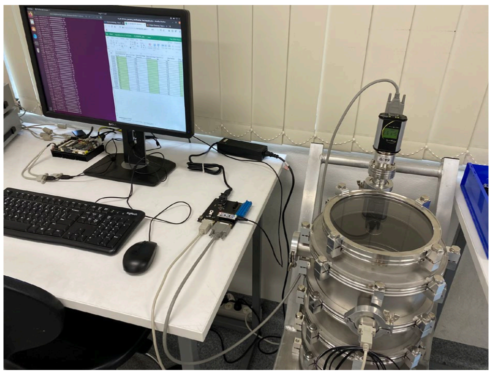


Figure 7: Vacuum testing performed at Curtin University.

Although the team would have liked to have performed more testing on the BCC and Binar-1, challenges with the communication system and a misunderstanding of changes in the regulatory requirements led to delays in the design and integration process, reducing the available testing time of Binar-1. Changes in the launch regulations meant that the antenna needed to be modified and re-tested to include two burn wires. This change was necessary due to the requirement for inadvertent antenna deployment inside the J-SSOD having its maximum allowable force reduced to below the force exerted by the COTS antenna. The schedule was also affected by a difference in the engineering model COTS UHF transceiver and the flight model COTS UHF transceiver. To reduce cost the team purchased a 1W variant of the transceiver for the engineering model and a 2W variant for the flight model. Due to a misunderstanding of documentation, it was unknown to the team until the beginning of testing the flight model transceiver that I2C was not usable on the 2W variant. This added to the delays as a new adapter board had to be made to change the connection to the transceiver.

As a result, only the very basic system level verification was performed on Binar-1 between the BCC and the transceiver alongside the necessary regulatory testing requirements. This means that only the beacon, detumble, and basic telecommands of Binar-1 was tested before launch, and no full Day-In-The-Life (DIL) testing could be completed as planned. One observation that was noted in the basic system testing was another challenge involving the UHF transceiver. A difference in receiving power usage from the datasheet was noticed which effected the power budget. The team decided that it was still in the best interest of the program to continue with the launch and perform a system update early in the mission to reduce the power usage of some of the other systems.

Regulatory Testing Requirements

The main regulatory testing requirements necessary for Binar-1 to meet were included in the safety review process. This included testing all the systems of Binar-1 that could cause damage to the launch vehicle, ISS, or astronauts on-board. The most significant of these tests was the battery verification testing, safety inhibit testing, and the vibration testing.

The battery verification and safety inhibit testing was the critical path of the Binar-1 assembly and testing process. Requiring a batch qualification of the lithium-ion cells on Binar-1, the process was time consuming due to the lack of resources able to perform the tests. After qualification, with flight model cells approved, the battery safety inhibits required testing as well. This included a short-circuit test, over-charge test, over-discharge test, switch inhibit test, and insulation test. These tests were performed at various stages of the assembly and integration procedure. After the complete assembly, a final battery cycle test was required before and after the vibration testing to finally verify the structural integrity of the battery cells and qualify for launch.

Vibration testing was performed at Curtin University using the available facilities. Similar to other CubeSat launches to the ISS, Binar-1 was qualified to all possible ISS resupply mission launch vehicles. The final testing before delivery was important to ensure that the satellite had been assembled correctly and that the antenna modification would not inadvertently deploy inside the J-SSOD.

MISSION OPERATIONS AND RESULTS

Deployed from the ISS at approximately 5:20pm (AWST) on the 6th of October, Binar-1 was required to wait 30 minutes before deploying its antennas and starting to beacon. The first possible attempt at receiving from the ground station in Western Australia was expected at approximately 11:00pm (AWST) however, the team also planned to use the SatNOGS[4] service to look for signals earlier. The beacon string contained the satellite name, GPS data, critical power and temperature information, and a unique message from the Binar team. Unfortunately, no communications were received on the first pass, or on any of the SatNOGS passes. This prompted the team to start attempting to communicate with the satellite and search the sky for its location. However, these attempts soon ended as the other two CubeSats launched along with Binar-1, Maya-3 and Maya-4, successfully established contact within the first day of operations, successfully confirming the expected location of Binar-1. As such, this prompted the team to start a failure mode analysis to determine if a recovery could be made.

Failure Mode Analysis

A benefit of the Binar-1 custom design was the knowledge of the system available to the team. By stepping through how the satellite would behave after deployment, the team was able to closely analyze the possible operation paths and determine if any possible software bugs or hardware failures could have caused the communication silence.

The first step of the failure analysis and the starting point for operation was the EPS. Being one of the most common reasons to failure[5], and necessary for powering the rest of the Binar systems, the EPS had been heavily tested throughout the design process. One possible failure point was found involving an interaction between the power distribution subsystem and the flight computer system. When the flight computer booted, one of the first actions it performs was to disable the secondary distribution subsystem. Before performing the task, the flight computer sets a system flag into memory and then resets it after the task is complete. If the system power cycles after disabling the redundant distribution subsystem, then at re-boot the system flag should still be set, and the flight computer will know that the redundant distribution subsystem is being used. However, a flaw was found in that the flag was being set in volatile memory causing it to reset if power was lost to the flight computer. This would have resulted in a flight computer power cycling event where it would continuously disable the redundant distribution subsystem. This flaw was found to not be the reason for failure due to the next attempts made by the team, however it was still an error that needed to be corrected in future implementations of the BCC.

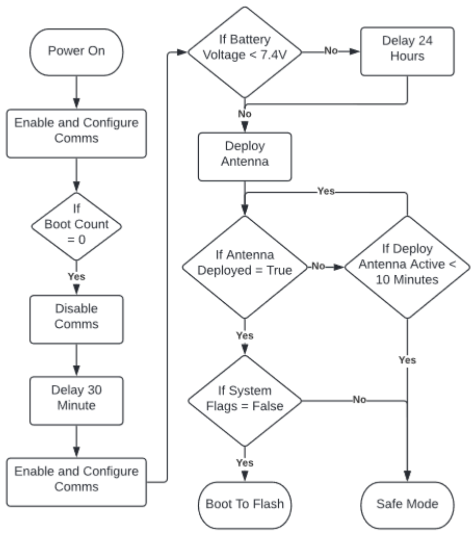


Figure 8: Binar bootloader software flow chart.

To continue the failure analysis process, the team assumed the flight computer system still had power. From here the team analyzed the flight software to determine if any logic errors or bugs had gone unnoticed since delivery. First the 30-minute wait must occur after deployment where the communications system is powered off. After the 30-minutes, the communications system is powered on, and the antennas deployment circuit will activate if the satellite has over 50% battery. The deployment burn wires will be switched on until the successful deployment condition is met, or until the 10-minute timeout is reached. If the battery is less than 50% then the satellite will wait for 24 hours before attempting the deployment. After this, the bootloader makes two system checks before deciding to boot into application code or safe mode. The first check is the system critical checks. This check will scan the system flags for any reported faults. These flags are only set in the system critical check application in application mode or in safe mode. This means that during the first boot, this check will always pass. The other check that is performed is a check on the antenna deployment condition. If the antennas did not deploy correctly then the CubeSat would be put into safe mode, were a beacon would be broadcast at reduced frequency to conserve power. The flow chart for this process is summarized in Figure 8.

During testing, safe mode was tested by disabling the 30-minute wait time, disabling the 10-minute time out of the antenna deployment system, and holding down the deployment sensor switch. This was done to conserve time in the Binar testing program due to the schedule losses mentioned previously. As a result, when testing the satellite would boot directly into safe mode so that the functionality could be tested. In testing, these beacons were received correctly however, this was only observed when the delay timers were not enabled. In this state the software would enable and configure the communications system before jumping straight into safe mode. However, if the communications system doesn't receive any signals for 255 seconds, the configuration is reset to its default mode. This is where a software logic error was found as when the timers are enabled, and if the antennas didn't deploy within 255 seconds, then the communications system would not be configured properly. To test that the error existed, the team performed a test with the engineering model and verified that this was a possibility for failure on Binar-1.

Another theorized possibility is that the antenna did deploy correctly and boot into the flash application software operation mode. In the application mode, the communications system would have been configured again however, the theory was put forward that the poorly tested adapter board, that had a last-minute modification, had failed. Due to the nature of the last-minute modification, the board was not thermal vacuum tested or vibration tested correctly which could have caused a solder joint to break meaning that the flight computer was potentially not able to communicate with the communications system or power it.

Fortunately for the Binar team, if the COTS communications system was powered and in its default mode, it could be configured from the ground. As a result, the team concluded that the best action would be to attempt to send the configuration commands to the satellite and see if the beacon could be received. After first confirming with the engineering model that this was possible, the team attempted to communicate with the communications system hoping that the adapter board was operating correctly.

Partial Recovery

To attempt the recovery, first the team attempted to configure the communications system in the required mode for the safe mode beacon to be received. These attempts were made over multiple passes to no success. It is still unclear as to the team on whether commands were received by the satellite or not as it depended on how many of the antennas were deployed, the attitude of the satellite during the passes, and if the adapter board was operating properly.

With these attempts not being successful, the team decided to attempt to put the COTS communication system into its own beacon mode. This beacon mode was built into the system and could be configured in a similar way to the desired configuration. The attempts to enable this mode were successful on the first attempt, partially recovering the location and status of Binar-1. The first beacon was received at approximately 5:21pm (AWST) on the 21st of October (Figure 9), almost exactly 15 days after the deployment from the ISS. The beacons enabled were operating with a shorter period and lower bit rate to try and help the team to locate the satellite on more ground station waterfall plots using the SatNOGS network.

One of the risks of enabling the shorter period beacon was that the power balance of Binar-1 would not be stable from the increased frequency of the beacon. As a result, the team needed to turn off the beacon as soon as possible. Unfortunately, the team was unable to turn off the beacon on the first attempt and was only able to switch the beacon into a shorter period mode 23 hours after the first signal had been heard. During this time, the beacon was seen around the world on the SatNOGS website before the beacons started to appear with a longer period. The longer period beacons were seen for another 11 and a half days until Binar-1 made its final recorded transmission at approximately 7:03am (AWST) on the 2nd of November. Although the team made many attempts to recover the satellite again after this date, it is suspected that the satellite ran out of charge at this point due to a combination of the compromised power budget and constant power cycling causing start-up applications to run regularly. The power cycling could be observed as the message contained in the beacon would revert to the default message. Another observation that was made was the bit rate of the beacons not being re-configured when the power cycle occurred. This led the team to believe that the cause of failure was likely due to the adapter board.

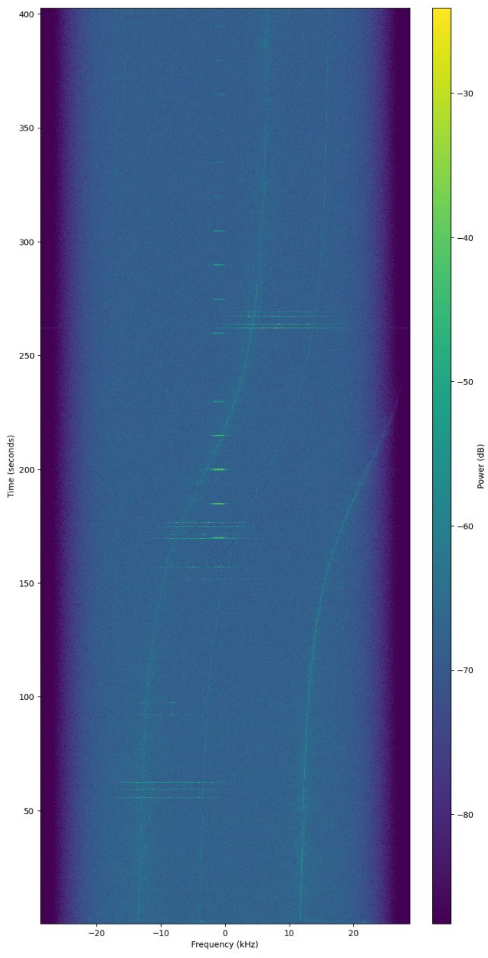


Figure 9: SatNOGS plot from the first observed communications with Binar-1. The wider signals are the transmissions from the ground station.

Results

From the failure analysis, the team believes that the last-minute adapter board modification was the cause of the lost communications with the flight computer. This belief comes from the beacon bit rate not being reset when the communications system was being power cycled, suggesting that the connection between the flight computer and the transceiver were not made correctly and likely broke during vibration testing, launch, or when exposed to the environment of space.

Although only a partial recovery was made, the Binar team was still able to infer some of the operations of the BCC from the beacon mode activated on the transceiver. It was clear to the team that the EPS was able to power the BCC until the final communication was made. Knowing the power budget problem before launch it is clear to the team that the solar panels, MPPTs, batteries, battery heaters, and the distribution subsystems were operational to some extent. The flight computer was also operating as expected as it was turning the communications system on. Alongside these two systems there was also valuable knowledge gained about the deployment switches and the performance of the structure during launch and in space. All of this could only be learnt by the team by delivering the mission.

LESSONS LEARNED

Although the mission was only considered a partial success in terms of the technology demonstration objective, the goal of educating staff and students was considered a success in terms of the many lessons that have been learned in the design process. Being a team starting with no knowledge about spacecraft design, the mission was always going to be challenging. The value of the lessons learned will help the team to overcome these challenges on the next launches from the Binar Space Program and be passed down to new students and staff beginning to work on the project. Although some of the lessons learned may not be new to more developed CubeSat design teams, the team believes that sharing the lessons learned will continue to build the literature around CubeSat design and hopes to help those who are yet to start the design process.

The first lesson learned by the team was the importance of locking down high-level mission objectives at the beginning of the design process. Although this is challenging when first starting the design, if possible, settle early on the budget and mission objectives. With these refined, defining requirements to meet the objectives is made easier. If the mission objectives are changed, start the process again and perform design reviews again if the objective changes are significant. One advantage of the decision to perform a custom design was the ability to easily adapt the design to some of these changing requirements, however this still meant that work needed to be repeated every time a change was made, significantly impacting the launch schedule.

Although power budgeting was performed in detail, the budget was only tested with the engineering model, and not to a suitable level of detail due to the missed DIL testing with the flight model. To improve its practices in the future the Binar team has learned to perform tests as you fly and not alter the engineering model to save costs. This costly operation may have been detrimental to the Binar-1 mission as the unbalanced power budget caused by the communication system was likely one of the reasons for communications loss.

Due to unexpected delays in the assembly process some testing was cut short. The team learned that it could be far better prepared for unexpected delays and prioritize its test program better if delays occur. Implementing this into the program will help to assess launch risk, and better manage the decision to either delay launch or remove some testing processes and assess the risks. Being able to present this plan to the mission leaders prior to the assembly can also help to better prepare the leadership team for delays and risk acceptance.

Although parts of the assembly and testing were shortened to make the launch, the team learnt important lessons about operation planning at the deployment of Binar-1. It was overlooked by the team the importance of putting in place an operations plan and setting up times for observations. This is something the team hopes to integrate into its DIL testing in the future to improve the performance of the operations plan and ready the team to operate the next set of Binar CubeSats.

The final lesson learnt relates to the goals of the Binar Space Program and the achievements observed by designing and assembling the satellite as a Program. Through the custom design, the Binar Space Program has learned and benefitted during the design and will continue to benefit in its future designs in different ways to how COTS comprised CubeSats benefit. This lesson will continue to be implemented by the Binar Space Program as it progresses into the future, aiming to work on its own payloads and platforms to continue building design experience at the university so that it will be able to deliver more complex space missions in the future.

CONCLUSION

Binar-1 was the first CubeSat launched by the Australian state of Western Australia. The custom designed CubeSat primary objective was to demonstrate the functionality of the integrated Binar CubeSat Core (BCC) which consisted of three of the satellites main systems including the Electrical Power System (EPS), Attitude Determination and Control System (ADCS), and flight computer system. The other objectives of the mission were to provide education to staff and students about end-to-end spacecraft design, and to spread awareness about the importance of space research and industry in the state.

After being deployed from the International Space Station (ISS) on the 5th of October 2021, Binar-1 was radio silent for almost exactly 15 days until a secondary beacon was enabled by the team. The secondary beacon was observed around the world by the SatNOGS network, until it stopped 11 and a half days later on the 2nd of November. This result has partially achieved the primary mission objective of the Binar-1 satellite demonstrating that the EPS and flight computer system on the BCC were operating in space, however no flight data could be collected to verify the systems completely.

Binar-1 has been successful at educating staff and students about end-to-end spacecraft design and provided a range of lessons learned which will be used in future Binar launches. These lessons include locking down mission requirements early, performing power budget testing with flight model systems, preparing for testing delays, planning for satellite operation, and the importance of using custom designed systems when aiming to perform consistent CubeSat missions.

Having learned these lessons and partially demonstrating the BCC, the team is now moving forward with implementing the lessons learned on its future missions. This will continue to grow the awareness of space in Western Australia as the team aims to deliver three 1U CubeSats, Binar-2, Binar-3, and Binar-4, in its next launch planned for 2023.

ACKNOWLEDGEMENTS

This research was supported by an Australian Government Research Training Program (RTP) Scholarship and received Strategic Research Investment support from Curtin University.

The authors would also like to acknowledge the continued support of AROSE and Fugro SpAARC, and thank them for their advice with ground station setup and operations planning.

REFERENCES

1. Robert M. Howie, Jonathan Paxman, Philip A. Bland, Martin C. Towner, Martin Cupak, Eleanor K. Sansom, and Hadrien A. R. Devillepoix. How to build a continental scale fireball camera network. Experimental Astronomy, 43(3):237–266, June 2017.
2. Andrew Dahir, Ariel Sandburg, and James Paul Mason. Advancement, Testing and Validation of an Innovative SmallSat Solar Panel Fabrication Process. SmallSat 2018, SSC17-WK-50.
3. JEM Payload Accommodation Handbook, Vol 8. Rev D. Small Satellite Deployment Interface Control Document. https://humans-in-space.jaxa.jp/kibouser/library/item/jx-espc_8d-d1_en.pdf
4. SatNOGS, Satellite Networked Open Ground Station. https://satnogs.org/
5. Raja Pandi Perumal, Holger Voos, Florio Dalla Vedova, Hubert Moser, and LuxSpace Sarl. Small Satellite Reliability: A decade in review. SmallSat 2021, SSC21-WKIII-02.

BIBLIOGRAPHY

[1] E. Kulu, *Nanosats Database*, en. [Online]. Available: https://www.nanosats.eu/index.html.

[2] H. Heidt, J. Puig-Suari, A. Moore, S. Nakasuka, and R. Twiggs, "CubeSat: A New Generation of Picosatellite for Education and Industry Low-Cost Space Experimentation," in *Proceedings of the 14th Small Satellite Conference*, Aug. 2000. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2000/All2000/32.

[3] R. Twiggs and M. Swartwout, "SAPPHIRE – Stanford's First Amateur Satellite," Vicksburg, Mississippi, USA, Oct. 1998.

[4] B. Engberg, J. Ota, and J. Suchman, "The OPAL Satellite Project: Continuing the Next Generation of Small Satellite Development," 1995.

[5] S. Lee, A. Hutputanasin, A. Tooriean, W. Lan, R. Munakata, J. Carnahan, D. Pignatelli, and A. Mehrparvar, *CubeSat Design Specification*, en, Feb. 2014.

[6] A. Chin, R. Coelho, R. Nugent, R. Munakata, and J. Puig-Suari, "Cubesat: The pico-satellite standard for research and education," in *AIAA SPACE 2008 Conference*. Sep. 2008. DOI: 10.2514/6.2008-7734. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2008-7734.

[7] D. Pignatelli, R. Nugent, R. Munakata, and W. Lan, *Poly Picosatellite Orbital Deployer Mk. III Rev. E User Guide*, Mar. 2014. [Online]. Available: https://static1.squarespace.com/static/5418c831e4b0fa4ecac1bacd/t/5806854d6b8f5b8eb57b83bd/1476822350599/P-POD_MkIIIRevE_UserGuide_CP-PPODUG-1.0-1_Rev1.pdf.

[8] G. Skrobot and R. Coelho, “ELaNa – Educational Launch of Nanosatellite: Providing Routine RideShare Opportunities,” *Small Satellite Conference*, Aug. 2012. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2012/all2012/42.

[9] *JEM Payload Accommodation Handbook*, Nov. 2018. [Online]. Available: https://iss.jaxa.jp/kiboexp/equipment/ef/jssod/images/jpah_vol.8_b_en.pdf.

[10] M. Shan, J. Guo, and E. Gill, “Review and comparison of active space debris capturing and removal methods,” en, *Progress in Aerospace Sciences*, vol. 80, pp. 18–32, Jan. 2016, ISSN: 03760421. DOI: 10.1016/j.paerosci.2015.11.001. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0376042115300221.

[11] E. S. Nightingale, L. M. Pratt, and A. Balakrishnan, “The CubeSat Ecosystem: Examining the Launch Niche,” Jerusalem, Israel, Dec. 2015. [Online]. Available: https://www.ida.org/-/media/feature/publications/t/th/the-cubesat-ecosystem-examining-the-launch-niche-paper-and-presentation/d-5678.ashx?la=en&hash=320B4F2FAC75A8B30350A1471EEE7AB8.

[12] M. Swartwout, “The First One Hundred CubeSats: A Statistical Look,” en, *Journal of Small Satellites*, p. 21, 2013.

[13] S. W. Asmar and S. Matousek, “Mars Cube One (MarCO) Shifting the Paradigm in Relay Deep Space Operation,” en, in *SpaceOps 2016 Conference*, American Institute of Aeronautics and Astronautics, May 2016, ISBN: 978-1-62410-426-8. DOI: 10.2514/6.2016-2483. [Online]. Available: http://arc.aiaa.org/doi/10.2514/6.2016-2483.

[14] Andrew Good and JoAnna Wendel, *The MarCO Mission Comes to an End*, Feb. 2020. [Online]. Available: https://mars.nasa.gov/news/8408/the-marco-mission-comes-to-an-end.

[15] L. McNutt, L. Johnson, P. Kahn, J. Castillo-Rogez, and A. Frick, "Near-Earth Asteroid (NEA) Scout," en, in *AIAA SPACE 2014 Conference and Exposition*, San Diego, CA: American Institute of Aeronautics and Astronautics, Aug. 2014, ISBN: 978-1-62410-257-8. DOI: 10.2514/6.2014-4435. [Online]. Available: http://arc.aiaa.org/doi/10.2514/6.2014-4435.

[16] M. I. Desai, F. Allegrini, R. W. Ebert, K. Ogasawara, M. E. Epperly, D. E. George, E. R. Christian, S. G. Kanekal, N. Murphy, and B. Randol, "The CubeSat Mission to Study Solar Particles," *IEEE Aerospace and Electronic Systems Magazine*, vol. 34, no. 4, pp. 16–28, Apr. 2019, ISSN: 1557-959X. DOI: 10.1109/MAES.2019.2917802.

[17] L. Mohon, *NEA Scout Status Update*, Text, Dec. 2022. [Online]. Available: http://www.nasa.gov/centers/marshall/news/2022/nea-scout-status-update.html.

[18] Denise Hill, *Artemis I Payload CuSP CubeSat Mission Update*, Text, Dec. 2022. [Online]. Available: https://blogs.nasa.gov/sunspot/2022/12/08/artemis-i-payload-cusp-cubesat-mission-update/.

[19] Pumpkin, Inc., *Motherboard Module (MBM)*, en, Store, Feb. 2023. [Online]. Available: http://www.pumpkinspace.com/store/p49/Motherboard_Module_%28MBM%29.html.

[20] Pumpkin, Inc., *Intelligent Protected Lithium Battery Module with SoC Reporting (BM2)*, en, Store, Feb. 2023. [Online]. Available: http://www.pumpkinspace.com/store/p198/Intelligent_Protected_Lithium_Battery_Module_with_SoC_Reporting_%28BM2%29.html.

[21] Endurosat, *1U CubeSat Solar Panel X/Y*, en-US, Store, Feb. 2023. [Online]. Available: https://www.endurosat.com/cubesat-store/cubesat-solar-panels/1u-solar-panel-x-y/.

[22] Pumpkin, Inc., *GNSS Receiver Module (GPSRM 1) Kit*, en, Store, Feb. 2023. [Online]. Available: http://www.pumpkinspace.com/store/p58/GNSS_Receiver_Module_%28GPSRM_1%29_Kit.html.

[23] Endurosat, *UHF Transceiver II CubeSat Communication*, en-US, Store, Feb. 2023. [Online]. Available: https://www.endurosat.com/cubesat-store/cubesat-communication-modules/uhf-transceiver-ii/.

[24] Endurosat, *UHF CubeSat Antenna Module*, en-US, Feb. 2023. [Online]. Available: https://www.endurosat.com/cubesat-store/cubesat-antennas/uhf-antenna/.

[25] Endurosat, *1U CubeSat Structure*, en-US, Store, Feb. 2023. [Online]. Available: https://www.endurosat.com/cubesat-store/cubesat-structures/1u-cubesat-structure/.

[26] Tensor Tech, *Tensor Tech ADCS - MTQ Integrated Attitude Determination and Control System with Magnetorquers*, en-US, shop, Oct. 2023. [Online]. Available: https://www.cubesatshop.com/product/tensor-tech-adcs-mtq-integrated-attitude-determination-and-control-system-with-magnetorquers/ (visited on 10/03/2023).

[27] A. E. Kalman, *CubeSat Kit Motherboard (MB)*, en, Mar. 2012. [Online]. Available: http://www.pumpkininc.com/space/datasheet/710-00484-E_DS_MBM.pdf.

[28] A. E. Kalman, *CubeSat Kit™ Battery Module 2 (BM 2)*, Sep. 2022. [Online]. Available: http://www.pumpkininc.com/space/datasheet/710-01640-F_DS_BM_2.pdf.

[29] A. E. Kalman, *CubeSat Kit™ GPSRM 1 GNSS Receiver Module*, Aug. 2022. [Online]. Available: http://www.pumpkininc.com/space/datasheet/710-00908-D_DS_GPSRM_1.pdf.

[30] *User Manual UHF Transceiver Type II*, en, Nov. 2018.

[31] *Tensor Tech Brochure*, 2022. [Online]. Available: https://www.cubesatshop.com/wp-content/uploads/2023/01/Tensor-Tech-Brochure-2022-1-compressed.pdf (visited on 10/03/2023).

[32] *SpaceX Smallsat Rideshare Program*, en. [Online]. Available: http://www.spacex.com (visited on 10/13/2023).

[33] J. Bouwmeester, M. Langer, and E. Gill, "Survey on the implementation and reliability of CubeSat electrical bus interfaces," en, *CEAS Space Journal*, vol. 9, no. 2, pp. 163–173, Jun. 2017, ISSN: 1868-2502, 1868-2510. DOI: 10.1007/s12567-016-0138-0. [Online]. Available: http://link.springer.com/10.1007/s12567-016-0138-0.

[34] J. Bouwmeester and N. Santos, "Analysis of the Distribution of Electrical Power in Cubesats," en, in *Proceedings of The 4S Symposium*, Spain: ESA, CNES, 2014, p. 12.

[35] G. F. Dubos, J.-F. Castet, and J. H. Saleh, "Statistical reliability analysis of satellites by mass category: Does spacecraft size matter?" en, *Acta Astronautica*, vol. 67, no. 5, pp. 584–595, Sep. 2010, ISSN: 0094-5765. DOI: 10.1016/j.actaastro.2010.04.017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0094576510001347.

[36] E. National Academies of Sciences and Medicine, *Achieving Science with CubeSats: Thinking Inside the Box*. Washington, DC: The National Academies Press, 2016, ISBN: 978-0-309-44263-3. DOI: 10.17226/23503. [Online]. Available: https://nap.nationalacademies.org/catalog/23503/achieving-science-with-cubesats-thinking-inside-the-box.

[37] M. Swartwout, *Mission Success for Universities and Professional Programs*, en, Jan. 2023. [Online]. Available: https://sites.google.com/a/slu.edu/swartwout/cubesat-database/repeat-success.

[38] G. Johnson-Roth, "Mission Assurance Guidelines for AD Mission Risk Classes," *Aerospace Corporation, TOR-2011 (8591)-21*, 2011.

[39] Committee on Achieving Science Goals with CubeSats, Space Studies Board, Division on Engineering and Physical Sciences, and National Academies of Sciences, Engineering, and Medicine, *Achieving Science with CubeSats: Thinking Inside the Box*, en. Washington, D.C.: National Academies Press, Oct. 2016, ISBN: 978-0-309-44263-3. DOI: `10.17226/23503`. [Online]. Available: `https://www.nap.edu/catalog/23503` (visited on 10/03/2023).

[40] D. Dvorak, "NASA Study on Flight Software Complexity," en, in *AIAA Infotech@Aerospace Conference*, Seattle, Washington: American Institute of Aeronautics and Astronautics, Apr. 2009, ISBN: 978-1-60086-979-2. DOI: `10.2514/6.2009-1882`. [Online]. Available: `https://arc.aiaa.org/doi/10.2514/6.2009-1882`.

[41] M. Langer and J. Bouwmeester, "Reliability of CubeSats - Statistical Data, Developers' Beliefs and the Way Forward," en, in *Proceedings of the AIAA/USU Conference on Small Satellites*, Aug. 2016, p. 12. [Online]. Available: `https://digitalcommons.usu.edu/smallsat/2016/TS10AdvTech2/4`.

[42] J. Bouwmeester and M. Langer, *Results of CubeSat Survey on Electrical Interfaces & Reliability*, en, 2016. DOI: `10.4121/UUID:591FF8F8-B495-4D1C-9C5C-69F6F85ACE78`. [Online]. Available: `https://data.4tu.nl/articles/_/12694949/1`.

[43] D. José Franzim Miranda, M. Ferreira, F. Kucinskis, and D. McComas, "A Comparative Survey on Flight Software Frameworks for 'New Space' Nanosatellite Missions," en, *Journal of Aerospace Technology and Management*, e4619, Oct. 2019, ISSN: 2175-9146. DOI: `10.5028/jatm.v11.1081`. [Online]. Available: `http://www.scielo.br/scielo.php?script=sci_arttext&pid=S2175-91462019000100341&lng=en&nrm=iso&tlng=en`.

[44] Jamie Chin, Roland Coelho, Justin Foley, Alicia Johnstone, Ryan Nugent, Dave Pignatelli, Savannah Pignatelli, Nikolaus Powell, Jordi Puig-Suari, William Atkinson, Jennifer Dorsey, Scott Higginbotham, Maile Krienke, Kristina Nelson, Bradley Poffenberger, Creg Raffington, Garrett Skrobot, Justin Treptow,

Anne Sweet, Jason Crusan, Carol Galica, William Horne, Charles Norton, and Alan Robinson, *CubeSat 101: Basic Concepts and Processes for First-Time CubeSat Developers*, en, Oct. 2017.

[45] M. D. Grubb, "Increasing the Reliability of Software Systems on Small Satellites Using Software-Based Simulation of the Embedded System," English, Master's thesis, West Virginia University, United States – West Virginia, 2021. [Online]. Available: `https://www.proquest.com/docview/2563685922/abstract/BD0ACAD670454C57PQ/1`.

[46] J. Eickhoff, *Onboard Computers, Onboard Software and Satellite Operations*, en, ser. Springer Aerospace Technology. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ISBN: 978-3-642-25170-2. DOI: `10.1007/978-3-642-25170-2`. [Online]. Available: `http://link.springer.com/10.1007/978-3-642-25170-2`.

[47] F. Stesina and S. Corpino, "Investigation of a CubeSat in Orbit Anomaly through Verification on Ground," en, *Aerospace*, vol. 7, no. 4, p. 38, Apr. 2020, ISSN: 2226-4310. DOI: `10.3390/aerospace7040038`. [Online]. Available: `https://www.mdpi.com/2226-4310/7/4/38`.

[48] *JPL Institutional Coding Standard for the C Programming Language*, en, Mar. 2009.

[49] A. Alanazi and J. Straub, "Engineering Methodology for Student-Driven CubeSats," en, *Aerospace*, vol. 6, no. 5, p. 54, May 2019, ISSN: 2226-4310. DOI: `10.3390/aerospace6050054`. [Online]. Available: `https://www.mdpi.com/2226-4310/6/5/54`.

[50] M. Swartwout and C. Jayne, "University-Class Spacecraft by the Numbers: Success, Failure, Debris. (But Mostly Success.)," *Small Satellite Conference*, Aug. 2016. [Online]. Available: `https://digitalcommons.usu.edu/smallsat/2016/TS13Education/1`.

[51] G. C. Messenger and M. S. Ash, *Single Event Phenomena*, en. Boston, MA: Springer US, 1997, ISBN: 978-1-4615-6043-2. DOI: `10.1007/978-1-4615-6043-2`.

[Online]. Available: `http://link.springer.com/10.1007/978-1-4615-6043-2` (visited on 10/12/2023).

[52] E. Stassinopoulos and J. Raymond, "The space radiation environment for electronics," *Proceedings of the IEEE*, vol. 76, no. 11, pp. 1423–1442, Nov. 1988, ISSN: 1558-2256. DOI: `10.1109/5.90113`.

[53] P. Madle, "STM32H7 Radiation Test Report," Open Source Satellite, Tech. Rep., May 2021. [Online]. Available: `https://www.opensourcesatellite.org/downloads/KS-DOC-01251_STM32H7_Radiation_Test_Report.pdf` (visited on 06/01/2022).

[54] C. Seidleck, K. LaBel, A. Moran, M. Gates, J. Barth, E. Stassinopoulos, and T. Gruner, "Single event effect flight data analysis of multiple NASA spacecraft and experiments; implications to spacecraft electrical designs," en, in *Proceedings of the Third European Conference on Radiation and its Effects on Components and Systems*, Arcachon, France: IEEE, 1996, pp. 581–588, ISBN: 978-0-7803-3093-1. DOI: `10.1109/RADECS.1995.509840`. [Online]. Available: `http://ieeexplore.ieee.org/document/509840/` (visited on 10/11/2023).

[55] R. C. Moore, "Satellite RF Communications and Onboard Processing," en, in *Encyclopedia of Physical Science and Technology*, Elsevier, 2003, pp. 439–455, ISBN: 978-0-12-227410-7. DOI: `10.1016/B0-12-227410-5/00884-X`. [Online]. Available: `https://linkinghub.elsevier.com/retrieve/pii/B012227410500884X` (visited on 10/12/2023).

[56] A. D. P. Hands, K. A. Ryden, N. P. Meredith, S. A. Glauert, and R. B. Horne, "Radiation Effects on Satellites During Extreme Space Weather Events," en, *Space Weather*, vol. 16, no. 9, pp. 1216–1226, 2018, ISSN: 1542-7390. DOI: `10.1029/2018SW001913`. [Online]. Available: `https://onlinelibrary.wiley.com/doi/abs/10.1029/2018SW001913` (visited on 10/11/2023).

[57] K. E. Holbert and L. T. Clark, "Radiation Hardened Electronics Destined For Severe Nuclear Reactor Environments," en, Tech. Rep. DOE-ASU–NE00679,

1238384, Feb. 2016, DOE–ASU–NE00679, 1 238 384. DOI: 10.2172/1238384. [Online]. Available: http://www.osti.gov/servlets/purl/1238384/ (visited on 06/01/2022).

[58] C. Underwood, R. Ecoffet, S. Duzeffier, and D. Faguere, "Observations Of Single-event Upset And Multiple-bit Upset In Non-hardened High-density SRAMs In The TOPEX/ Poseidon Orbit," in *1993 IEEE Radiation Effects Data Workshop*, Snowbird, UT, USA: IEEE, 1993, pp. 85–92, ISBN: 978-0-7803-1906-6. DOI: 10.1109/REDW.1993.700572. [Online]. Available: http://ieeexplore.ieee.org/document/700572/ (visited on 10/12/2023).

[59] J. A. Starek, B. Açıkmeşe, I. A. Nesnas, and M. Pavone, "Spacecraft Autonomy Challenges for Next-Generation Space Missions," en, in *Advances in Control System Technology for Aerospace Applications*, ser. Lecture Notes in Control and Information Sciences, E. Feron, Ed., Berlin, Heidelberg: Springer, 2016, pp. 1–48, ISBN: 978-3-662-47694-9. DOI: 10.1007/978-3-662-47694-9_1. [Online]. Available: https://doi.org/10.1007/978-3-662-47694-9_1.

[60] D. Bernard, E. Gamble, G. Man, G. Dorais, B. Kanefsky, W. Millar, K. Rajan, J. Kurien, N. Muscettola, and P. Nayak, "Spacecraft autonomy flight experience - The DS1 Remote Agent Experiment," en, in *Space Technology Conference and Exposition*, Albuquerque,NM,U.S.A.: American Institute of Aeronautics and Astronautics, Sep. 1999. DOI: 10.2514/6.1999-4512. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.1999-4512.

[61] D. Q. Tran, S. Chien, G. Rabideau, and B. Cichy, *Safe agents in space : Preventing and responding to Anomalies in the Autonomous Sciencecraft Experiment*, en, Jul. 2005. [Online]. Available: https://trs.jpl.nasa.gov/handle/2014/41168.

[62] B. A. Bauer and W. M. Reid, "Automating the Pluto Experience: An Examination of the New Horizons Autonomous Operations Subsystem," en, in *2007 IEEE Aerospace Conference*, Big Sky, MT, USA: IEEE, 2007, pp. 1–10, ISBN: 978-1-4244-0524-4. DOI: 10.1109/AERO.2007.352645. [Online]. Available: http://ieeexplore.ieee.org/document/4161523/.

[63] S. Speretta, F. Topputo, J. Biggs, P. Di Lizia, M. Massari, K. Mani, D. Dei Tos, S. Ceccherini, V. Franzese, A. Cervone, P. Sundaramoorthy, R. Noomen, S. Mestry, A. d. C. Cipriano, A. Ivanov, D. Labate, L. Tommasi, A. Jochemsen, J. Gailis, R. Furfaro, V. Reddy, J. Vennekens, and R. Walker, "LUMIO: Achieving autonomous operations for Lunar exploration with a CubeSat," en, in *2018 SpaceOps Conference*, Marseille, France: American Institute of Aeronautics and Astronautics, May 2018, ISBN: 978-1-62410-562-3. DOI: 10.2514/6.2018-2599. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2018-2599.

[64] J. Doubleday, S. Chien, C. Norton, K. Wagstaff, D. R. Thompson, J. Bellardo, C. Francis, and E. Baumgarten, "Autonomy for remote sensing — Experiences from the IPEX CubeSat," in *2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, Jul. 2015, pp. 5308–5311. DOI: 10.1109/IGARSS.2015.7327033.

[65] M. Uo, K. Shirakawa, T. Hasimoto, T. Kubota, and J. Kawaguchi, "Hayabusa Touching-Down to Itokawa - Autonomous Guidance and Navigation," *The Journal of Space Technology and Science*, vol. 22, no. 1, 1_32–1_41, 2006. DOI: 10.11230/jsts.22.1_32.

[66] T. Kohout, A. Näsilä, T. Tikka, M. Granvik, A. Kestilä, A. Penttilä, J. Kuhno, K. Muinonen, K. Viherkanto, and E. Kallio, "Feasibility of asteroid exploration using CubeSats—ASPECT case study," en, *Advances in Space Research*, Past, Present and Future of Small Body Science and Exploration, vol. 62, no. 8, pp. 2239–2244, Oct. 2018, ISSN: 0273-1177. DOI: 10.1016/j.asr.2017.07.036. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S027311771730546X.

[67] T. M. Davis and D. Melanson, "XSS-10 microsatellite flight demonstration program results," in *Spacecraft Platforms and Infrastructure*, P. T. Jr. and M. Wright, Eds., International Society for Optics and Photonics, vol. 5419, SPIE, 2004, pp. 16–

25. DOI: 10.1117/12.544316. [Online]. Available: https://doi.org/10.1117/12.544316.

[68] H. Leppinen, P. Niemelä, N. Silva, H. Sanmark, H. Forstén, A. Yanes, R. Modrzewski, A. Kestilä, and J. Praks, "Developing a Linux-based nanosatellite on-board computer: Flight results from the Aalto-1 mission," *IEEE Aerospace and Electronic Systems Magazine*, vol. 34, no. 1, pp. 4–14, Jan. 2019, ISSN: 1557-959X. DOI: 10.1109/MAES.2019.170217.

[69] I. Latachi, T. Rachidi, M. Karim, and A. Hanafi, "Reusable and Reliable Flight-Control Software for a Fail-Safe and Cost-Efficient Cubesat Mission: Design and Implementation," en, *Aerospace*, vol. 7, no. 10, p. 146, Oct. 2020, ISSN: 2226-4310. DOI: 10.3390/aerospace7100146. [Online]. Available: https://www.mdpi.com/2226-4310/7/10/146.

[70] A. Almazrouei, A. Khan, A. Almesmari, A. Albuainain, A. Bushlaibi, A. Al Mahmood, A. Alqaraan, A. Alhammadi, A. AlBalooshi, A. Khater, A. Alharam, B. AlTawil, B. Alkhzaimi, E. Almansoori, F. Jarrar, H. Issa, H. Alblooshi, M. T. Ansari, N. Alzaabi, N. Braik, P. Dimitropoulos, P. Marpu, R. Alialali, R. Alhammadi, R. Al-haddad, S. Almazrouei, S. Bahumaish, V. Thu, and Y. Alqassab, "A Complete Mission Concept Design and Analysis of the Student-Led CubeSat Project: Light-1," en, *Aerospace*, vol. 8, no. 9, p. 247, Sep. 2021, ISSN: 2226-4310. DOI: 10.3390/aerospace8090247. [Online]. Available: https://www.mdpi.com/2226-4310/8/9/247.

[71] M. Smith, A. Donner, M. Knapp, C. Pong, C. Smith, J. Luu, P. Pasquale, and B. Campuzano, "On-Orbit Results and Lessons Learned from the ASTERIA Space Telescope Mission," *Small Satellite Conference*, Aug. 2018. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2018/all2018/255.

[72] A. Wander and R. Förstner, "Innovative Fault Detection, Isolation and Recovery Strategies On-Board Spacecraft: State of the Art and Research Challenges," en, *Conference on Control and Fault-Tolerant Systems, SysTol*, p. 9, 2012. DOI: 10.1109/SysTol.2013.6693950.

[73] M. Shaw, M. Denis, O. Camino, A. Accomazzo, and P. Jayaraman, "Lessons Learned from Safe Modes on ESA's Interplanetary Missions: Mars Express, Venus Express and Rosetta," in *SpaceOps 2008 Conference*, May 2008. DOI: 10.2514/6.2008-3257. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2008-3257.

[74] D. Lakey, "FAST: A new MEX Operations concept, quickly!" en, in *SpaceOps 2012 Conference*, Stockholm, Sweden: American Institute of Aeronautics and Astronautics, Jun. 2012. DOI: 10.2514/6.2012-1257123. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2012-1257123.

[75] M. Nayak, "CloudSat Anomaly Recovery and Operational Lessons Learned," en, in *SpaceOps 2012 Conference*, Stockholm, Sweden: American Institute of Aeronautics and Astronautics, Jun. 2012. DOI: 10.2514/6.2012-1295798. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2012-1295798.

[76] I. Clerigo, S. Sangiorgi, and J. Volpp, "Avoiding Cluster Safe Modes," en, in *SpaceOps 2012 Conference*, Stockholm, Sweden: American Institute of Aeronautics and Astronautics, Jun. 2012. DOI: 10.2514/6.2012-1262380. [Online]. Available: https://arc.aiaa.org/doi/10.2514/6.2012-1262380.

[77] A. G. Hoekstra, B. Chopard, D. Coster, S. Portegies Zwart, and P. V. Coveney, "Multiscale computing for science and engineering in the era of exascale performance," en, *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 377, no. 2142, p. 20 180 144, Apr. 2019, ISSN: 1364-503X, 1471-2962. DOI: 10.1098/rsta.2018.0144. [Online]. Available: https://royalsocietypublishing.org/doi/10.1098/rsta.2018.0144.

[78] P. E. Wellstead, *Introduction to physical system modelling*, en. London ; New York: Academic Press, 1979, ISBN: 978-0-12-744380-5.

[79] C. Taber and R. Timpone, *Computational Modeling*. Thousand Oaks, California, 1996. DOI: 10.4135/9781412983716. [Online]. Available: https://methods.sagepub.com/book/computational-modeling.

[80] S. J. Moss and P. Davidsson, Eds., *Multi-Agent-Based Simulation: second international workshop, MABS 2000, Boston, MA, USA, July: revised and additional papers*, en, ser. Lecture notes in computer science ; 1979. Lecture notes in artificial intelligence. Berlin ; New York: Springer, 2001, ISBN: 978-3-540-41522-0.

[81] J. Mylopoulos, *Conceptual Modelling and Telos*, en, 1992.

[82] P. Hehenberger and D. Bradley, *Mechatronic Futures: Challenges and Solutions for Mechatronic Systems and their Designers*. Springer International Publishing, 2016, ISBN: 978-3-319-32156-1. [Online]. Available: https://books.google.com.au/books?id=LAlkDAAAQBAJ.

[83] B. Kouvaritakis and M. Cannon, *Model Predictive Control*, en, ser. Advanced Textbooks in Control and Signal Processing. Cham: Springer International Publishing, 2016, ISBN: 978-3-319-24853-0. DOI: 10.1007/978-3-319-24853-0. [Online]. Available: http://link.springer.com/10.1007/978-3-319-24853-0 (visited on 10/31/2023).

[84] H. Park, S. Di Cairano, and I. Kolmanovsky, "Model Predictive Control for spacecraft rendezvous and docking with a rotating/tumbling platform and for debris avoidance," Jun. 2011, pp. 1922–1927. DOI: 10.1109/ACC.2011.5991151.

[85] B. Açıkmeşe, J. M. Carson III, and D. S. Bayard, "A robust model predictive control algorithm for incrementally conic uncertain/nonlinear systems," en, *International Journal of Robust and Nonlinear Control*, vol. 21, no. 5, pp. 563–590, 2011, ISSN: 1099-1239. DOI: 10.1002/rnc.1613. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rnc.1613 (visited on 10/31/2023).

[86] R. Rosen, G. von Wichert, G. Lo, and K. D. Bettenhausen, "About The Importance of Autonomy and Digital Twins for the Future of Manufacturing," en, *IFAC-PapersOnLine*, vol. 48, no. 3, pp. 567–572, 2015, ISSN: 24058963. DOI: 10.1016/j.ifacol.2015.06.141. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2405896315003808.

[87] E. Glaessgen and D. Stargel, "The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles," in *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference*, American Institute of Aeronautics and Astronautics, Jun. 2012. DOI: 10.2514/6.2012-1818. [Online]. Available: https://arc.aiaa.org/doi/abs/10.2514/6.2012-1818 (visited on 11/02/2023).

[88] F. Faniyi, P. R. Lewis, R. Bahsoon, and X. Yao, "Architecting Self-Aware Software Systems," en, in *2014 IEEE/IFIP Conference on Software Architecture*, Sydney, Australia: IEEE, Apr. 2014, pp. 91–94, ISBN: 978-1-4799-3412-6. DOI: 10.1109/WICSA.2014.18. [Online]. Available: http://ieeexplore.ieee.org/document/6827105/.

[89] Y. Ye, Q. Yang, F. Yang, Y. Huo, and S. Meng, "Digital twin for the structural health management of reusable spacecraft: A case study," en, *Engineering Fracture Mechanics*, vol. 234, p. 107 076, Jul. 2020, ISSN: 0013-7944. DOI: 10.1016/j.engfracmech.2020.107076. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0013794419315759.

[90] E. J. Tuegel, A. R. Ingraffea, T. G. Eason, and S. M. Spottswood, "Reengineering Aircraft Structural Life Prediction Using a Digital Twin," en, *International Journal of Aerospace Engineering*, vol. 2011, pp. 1–14, 2011, ISSN: 1687-5966, 1687-5974. DOI: 10.1155/2011/154798. [Online]. Available: http://www.hindawi.com/journals/ijae/2011/154798/.

[91] C. Li, S. Mahadevan, Y. Ling, S. Choze, and L. Wang, "Dynamic Bayesian Network for Aircraft Wing Health Monitoring Digital Twin," en, *AIAA Journal*, vol. 55, no. 3, pp. 930–941, Mar. 2017, ISSN: 0001-1452, 1533-385X. DOI: 10.2514

/1.J055201. [Online]. Available: https://arc.aiaa.org/doi/10.2514/1.J055201.

[92] D. Shangguan, L. Chen, and J. Ding, "A Digital Twin-Based Approach for the Fault Diagnosis and Health Monitoring of a Complex Satellite System," en, *Symmetry*, vol. 12, no. 8, Aug. 2020, ISSN: 2073-8994. DOI: 10.3390/sym12081307. [Online]. Available: https://www.mdpi.com/2073-8994/12/8/1307 (visited on 11/02/2023).

[93] Y. Peng, X. Zhang, Y. Song, and D. Liu, "A Low Cost Flexible Digital Twin Platform for Spacecraft Lithium-ion Battery Pack Degradation Assessment," in *2019 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*, May 2019, pp. 1–6. DOI: 10.1109/I2MTC.2019.8827160.

[94] A. Qahmash, I. Al-Darraji, A. O. Khadidos, G. Tsaramirsis, A. O. Khadidos, and M. Alghamdi, "On-Board Digital Twin Based on Impedance and Model Predictive Control for Aerial Robot Grasping," en, *Journal of Sensors*, vol. 2023, May 2023, ISSN: 1687-725X. DOI: 10.1155/2023/9662100. [Online]. Available: https://www.hindawi.com/journals/js/2023/9662100/ (visited on 10/29/2023).

[95] H. Huang, L. Yang, Y. Wang, X. Xu, and Y. Lu, "Digital Twin-driven online anomaly detection for an automation system based on edge intelligence," *Journal of Manufacturing Systems*, vol. 59, pp. 138–150, Apr. 2021, ISSN: 0278-6125. DOI: 10.1016/j.jmsy.2021.02.010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0278612521000467 (visited on 10/30/2023).

[96] R. Magargle, L. Johnson, P. Mandloi, P. Davoudabadi, O. Kesarkar, S. Krishnaswamy, J. Batteh, and A. Pitchaikani, "A Simulation-Based Digital Twin for Model-Driven Health Monitoring and Predictive Maintenance of an Automotive Braking System," en, in *Proceedings of the 12th International Modelica Conference*, Jul. 2017, pp. 35–46. DOI: 10.3384/ecp1713235. [Online]. Available: http://www.ep.liu.se/ecp/article.asp?issue=132%26article=3.

[97] E. M. Kraft, "The Air Force Digital Thread/Digital Twin - Life Cycle Integration and Use of Computational and Experimental Knowledge," en, in *54th AIAA Aerospace Sciences Meeting*, San Diego, California, USA: American Institute of Aeronautics and Astronautics, Jan. 2016, ISBN: 978-1-62410-393-3. DOI: 10.2514/6.2016-0897. [Online]. Available: http://arc.aiaa.org/doi/10.2514/6.2016-0897.

[98] F. Tao and M. Zhang, "Digital Twin Shop-Floor: A New Shop-Floor Paradigm Towards Smart Manufacturing," en, *IEEE Access*, vol. 5, 2017, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2756069. [Online]. Available: http://ieeexplore.ieee.org/document/8049520/.

[99] M. Tafazoli, "A study of on-orbit spacecraft failures," en, *Acta Astronautica*, vol. 64, no. 2-3, pp. 195–205, Jan. 2009, ISSN: 00945765. DOI: 10.1016/j.actaastro.2008.07.019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0094576508003019.

[100] N. G. Leveson, "Role of Software in Spacecraft Accidents," en, *Journal of Spacecraft and Rockets*, May 2012. DOI: 10.2514/1.11950.

[101] D. McComas, J. Wilmot, and A. Cudmore, "The Core Flight System (cFS) Community: Providing Low Cost Solutions for Small Spacecraft," in *30th Annual AIAA/USU Conference on Small Satellites*, Salt Lake City, UT, Aug. 2016. [Online]. Available: https://ntrs.nasa.gov/citations/20160010300.

[102] R. Bocchino, T. Canham, G. Watney, L. Reder, and J. Levison, "F Prime: An Open-Source Framework for Small-Scale Flight Software Systems," *Small Satellite Conference*, Aug. 2018. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2018/all2018/328.

[103] R. Plauche, "Building Modern Cross-Platform Flight Software for Small Satellites," *Small Satellite Conference*, Aug. 2017. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2017/all2017/204.

[104] M. Doyle, A. Gloster, C. O'Toole, J. Mangan, D. Murphy, R. Dunwoody, M. Emam, J. Erkal, J. Flanaghan, G. Fontanesi, F. Okosun, R. R. Nair, J. Reilly, L. Salmon, D. Sherwin, J. Thompson, S. Walsh, D. de Faoite, U. Javaid, S. McBreen, D. McKeown, D. O'Callaghan, W. O'Connor, K. Stanton, A. Ulyanov, R. Wall, and L. Hanlon, "Flight Software Development for the EIRSAT-1 Mission," in *Proceedings of the 3rd Symposium on Space Educational Activities*, 2020, pp. 157–161. DOI: `10.29311/2020.39`. [Online]. Available: `http://arxiv.org/abs/2008.09074`.

[105] H. A. Askari, E. W. H. Eugene, A. N. Nikicio, G. C. Hiang, L. Sha, and L. H. Choo, "Software Development for Galassia CubeSat – Design, Implementation and In-Orbit Validation," en, in *31st International Symposium on Space Technology and Science (ISTS), 26th International Symposium on Space Flight Dynamics (ISSFD), and 8th Nano-Satellite Symposium (NSAT)*, Jun. 2017, p. 9.

[106] S. F. Hishmeh, T. J. Doering, and J. E. Lumpp, "Design of flight software for the KySat CubeSat bus," in *2009 IEEE Aerospace conference*, Mar. 2009, pp. 1–15. DOI: `10.1109/AERO.2009.4839646`.

[107] V. Carrara, R. B. Januzi, D. H. Makita, L. F. d. P. Santos, and L. S. Sato, "The ITASAT CubeSat Development and Design," en, *Journal of Aerospace Technology and Management*, vol. 9, no. 2, pp. 147–156, Apr. 2017, ISSN: 2175-9146. DOI: `10.5028/jatm.v9i2.614`. [Online]. Available: `http://www.jatm.com.br/ojs/index.php/jatm/article/view/614`.

[108] T. Whitney, J. Straub, and R. Marsh, "Design and Implementation of Satellite Software to Facilitate Future CubeSat Development," en, in *AIAA SPACE 2015 Conference and Exposition*, Pasadena, California: American Institute of Aeronautics and Astronautics, Aug. 2015, ISBN: 978-1-62410-334-6. DOI: `10.2514/6.2015-4500`. [Online]. Available: `https://arc.aiaa.org/doi/10.2514/6.2015-4500`.

[109] K. V. C. K. de Souza, Y. Bouslimani, and M. Ghribi, "Flight Software Development for a Cubesat Application," en, *IEEE Journal on Miniaturization for Air and*

Space Systems, pp. 1–1, 2022, ISSN: 2576-3164. DOI: 10.1109/JMASS.2022.3206713. [Online]. Available: https://ieeexplore.ieee.org/document/9891727/.

[110] Space Engineering Software, *ECSS-E-ST-40C: Space Engineering Software*, Mar. 2009.

[111] Space Product Assurance, *ECSS-Q-ST-80C: Space Software Product Assurance*, Feb. 2017.

[112] J. Clare and K. I. Kourousis, "Learning from Incidents: A Qualitative Study in the Continuing Airworthiness Sector," en, *Aerospace*, vol. 8, no. 2, p. 27, Feb. 2021, ISSN: 2226-4310. DOI: 10.3390/aerospace8020027.

[113] R. Nilchiani, "Valuing software-based options for space systems flexibility," en, *Acta Astronautica*, vol. 65, no. 3-4, pp. 429–441, Aug. 2009, ISSN: 00945765. DOI: 10.1016/j.actaastro.2009.02.012.

[114] X.-Y. Ji, Y.-Z. Li, G.-Q. Liu, J. Wang, S.-H. Xiang, X.-N. Yang, and Y.-Q. Bi, "A brief review of ground and flight failures of Chinese spacecraft," en, *Progress in Aerospace Sciences*, vol. 107, pp. 19–29, May 2019, ISSN: 0376-0421. DOI: 10.1016/j.paerosci.2019.04.002.

[115] C. Venturini, B. Braun, D. Hinkley, and G. Berg, "Improving Mission Success of CubeSats," in *Proceedings of the AIAA/USU Conference on Small Satellites*, Aug. 2018. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2018/all2018/273.

[116] T. E. Rumford, "Demonstration of autonomous rendezvous technology (DART) project summary," in *Space Systems Technology and Operations*, International Society for Optics and Photonics, vol. 5088, SPIE, 2003, pp. 10–19. DOI: 10.1117/12.498811. [Online]. Available: https://doi.org/10.1117/12.498811.

[117] S. Lilley, *Critical Software: Good Design Built Right*, Jan. 2012. [Online]. Available: https://nsc.nasa.gov/docs/default-source/system-failure-case-studies/good-design-built-right.pdf?sfvrsn=c6fecf8_4.

[118] S. Croomes, "Overview of the DART Mishap Investigation Results," en, National Aeronautics and Space Administration, Tech. Rep., 2006. [Online]. Available: https://www.nasa.gov/pdf/148072main_DART_mishap_overview.pdf.

[119] A. G. Stephenson, L. S. LaPiana, D. R. Mulville, P. J. Rutledge, F. H. Bauer, D. Folta, G. A. Dukeman, R. Sackheim, and P. Norvig, "Mars Climate Orbiter Mishap Investigation Board Phase 1 Report," NASA, Tech. Rep., Nov. 1999. [Online]. Available: https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf.

[120] Edward A. Euler, Steven D. Jolly, and H. H. Curtis, "The failures of the Mars Climate Orbiter and Mars Polar Lander: A perspective from the people involved," in *Proceedings of the 24th Annual AAS Guidance and Control Conference*, American Astronautical Society, Jan. 2001, p. 22. [Online]. Available: http://web.mit.edu/16.070/www/readings/Failures_MCO_MPL.pdf.

[121] J. L. Lions, "Ariane 5 Flight 501 Failure: Report by the Inquiry Board," European Space Agency, Tech. Rep., Jul. 1996. [Online]. Available: https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf.

[122] B. Harvey, *The rebirth of the Russian space program: 50 years after Sputnik, new frontiers*, en, 1st ed. New York: Springer, 2007, ISBN: 978-0-387-71354-0.

[123] A. Gorbenko, V. Kharchenko, O. Tarasyuk, and S. Zasukha, "A Study of Orbital Carrier Rocket and Spacecraft Failures: 2000-2009," *Information & Security: An International Journal*, vol. 28, pp. 179–198, Jan. 2012. DOI: 10.11610/isij.2815.

[124] J. D. Mooney, "Portability and reusability: Common issues and differences," en, in *Proceedings of the 1995 ACM 23rd annual conference on Computer science - CSC*

'95, Nashville, Tennessee, United States: ACM Press, 1995, pp. 150–156, ISBN: 978-0-89791-737-7. DOI: 10.1145/259526.259550.

[125] C. Larman and P. Kruchten, *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and Iterative Development*. Prentice Hall PTR, 2005, ISBN: 978-0-13-148906-6. [Online]. Available: https://books.google.com.au/books?id=tuxQAAAAMAAJ.

[126] R. C. Martin, "Design Principles and Design Patterns," en, *Object Mentor*, p. 34, 2000. [Online]. Available: http://staff.cs.utu.fi/staff/jouni.smed/doos_06/material/DesignPrinciplesAndPatterns.pdf.

[127] R. C. Martin, *Agile software development, principles, patterns, and practices*, en, 1. ed., Pearson new international ed. Harlow: Pearson, 2014, ISBN: 978-1-292-02594-0.

[128] B. Liskov, "Data Abstraction and Hierarchy," *ACM SIGPLAN Notices*, vol. 23, no. 5, pp. 17–34, Jan. 1987, ISSN: 0362-1340. DOI: https://doi.org/10.1145/62139.62141.

[129] M. Källén, S. Holmgren, and E. P. Hvannberg, "Impact of Code Refactoring Using Object-Oriented Methodology on a Scientific Computing Application," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, Sep. 2014, pp. 125–134. DOI: 10.1109/SCAM.2014.21.

[130] V. Garousi, M. Felderer, C. M. Karapıçak, and U. Yılmaz, "Testing embedded software: A survey of the literature," en, *Information and Software Technology*, vol. 104, pp. 14–45, Dec. 2018, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2018.06.016.

[131] M. Tipaldi, C. Legendre, O. Koopmann, M. Ferraguto, R. Wenker, and G. D'Angelo, "Development strategies for the satellite flight software on-board Meteosat Third Generation," en, *Acta Astronautica*, vol. 145, pp. 482–491, Apr. 2018, ISSN: 00945765. DOI: 10.1016/j.actaastro.2018.02.020.

[132] K. Beck, *Test-driven Development: By Example*, ser. Addison-Wesley signature series. Addison-Wesley, 2003, ISBN: 978-0-321-14653-3.

[133] A. Forward and T. C. Lethbridge, “The relevance of software documentation, tools and technologies: A survey,” in *Proceedings of the 2002 ACM symposium on Document engineering*, ser. DocEng ’02, New York, NY, USA: Association for Computing Machinery, Nov. 2002, pp. 26–33, ISBN: 978-1-58113-594-7. DOI: 10.1145/585058.585065.

[134] A. da Silva, S. Sánchez, O. R. Polo, and P. Parra, “Injecting faults to succeed. Verification of the boot software on-board solar orbiter’s energetic particle detector,” en, *Acta Astronautica*, vol. 95, pp. 198–209, Feb. 2014, ISSN: 0094-5765. DOI: 10.1016/j.actaastro.2013.11.004.

[135] J. Martin, “Data-Base Action Diagrams,” in *Managing the Data-base Environment*, Prentice-Hall, 1983, pp. 375–411, ISBN: 978-0-13-550582-3.

[136] The Pigweed Authors, *Pigweed*, Mar. 2023. [Online]. Available: https://github.com/google/pigweed.

[137] A. Zeedan and T. Khattab, “CubeSat Communication Subsystems: A Review of On-Board Transceiver Architectures, Protocols, and Performance,” *IEEE Access*, vol. 11, pp. 88 161–88 183, 2023, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3304419. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10224067 (visited on 10/27/2023).

[138] S. Cheshire and M. Baker, “Consistent overhead byte stuffing,” *IEEE/ACM Transactions on Networking*, vol. 7, no. 2, pp. 159–172, Apr. 1999, ISSN: 1558-2566. DOI: 10.1109/90.769765.

[139] F. Downey, S. Buchan, B. Hartig, D. Busan, J. Cook, R. Howie, P. Bland, and J. Paxman, “Binar Space Program: Binar-1 Results and Lessons Learned,” in *Proceedings of the Small Satellite Conference*, Aug. 2022. [Online]. Available: https://digitalcommons.usu.edu/smallsat/2022/all2022/56.

[140] S. Ma, M. Jiang, P. Tao, C. Song, J. Wu, J. Wang, T. Deng, and W. Shang, “Temperature effect and thermal impact in lithium-ion batteries: A review,” en, *Progress in Natural Science: Materials International*, vol. 28, no. 6, pp. 653–666, Dec. 2018,

ISSN: 1002-0071. DOI: 10.1016/j.pnsc.2018.11.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1002007118307536.

[141] O. Erdinc, B. Vural, and M. Uzunoglu, "A dynamic lithium-ion battery model considering the effects of temperature and capacity fading," in *2009 International Conference on Clean Electrical Power*, Jun. 2009, pp. 383–386. DOI: 10.1109/ICCEP.2009.5212025.

[142] J. Straub, "A Review of Spacecraft AI Control Systems," *WMSCI 2011 - The 15th World Multi-Conference on Systemics, Cybernetics and Informatics, Proceedings*, vol. 2, pp. 20–25, Jan. 2011.

[143] J. Straub, "Attitudes towards Autonomous Data Collection and Analysis in the Planetary Science Community," en, *Galaxies*, vol. 1, no. 1, pp. 44–64, Jun. 2013, ISSN: 2075-4434. DOI: 10.3390/galaxies1010044. [Online]. Available: https://www.mdpi.com/2075-4434/1/1/44.

[144] D. W. Way, "Preliminary assessment of the Mars Science Laboratory entry, descent, and landing simulation," in *2013 IEEE Aerospace Conference*, Mar. 2013, pp. 1–16. DOI: 10.1109/AERO.2013.6497404.

[145] F. Abilleira, G. Kruizinga, S. Aaron, K. Angkasa, T. Barber, P. Brugarolas, D. Burkhart, A. Chen, S. Demcak, J. Essmiller, J. Gilbert, E. Gustafson, J. Kangas, M. Jesick, S. E. McCandless, S. Mohan, N. Mottinger, C. O'Farrell, R. Otero, C. Pong, M. Ryne, J. Seubert, P. Thompson, S. Wagner, and M. Wong, *Mars 2020 Perseverance trajectory reconstruction and performance from launch through landing*, en, Aug. 2021. [Online]. Available: https://trs.jpl.nasa.gov/handle/2014/52267.

[146] H. Shyldkrot, E. Shmidt, D. Geron, J. Kronenfeld, M. Loucks, J. Carrico, L. Policastri, and J. Taylor, "The First Commercial Lunar Lander Mission: Beresheet," in *Proceedings of the 2019 AAS/AIAA Astrodynamics Specialist Conference*, Aug. 2019.

[147] Y.-B. Kim, H.-J. Jeong, S.-M. Park, J. H. Lim, and H.-H. Lee, "Prediction and Validation of Landing Stability of a Lunar Lander by a Classification Map Based on Touchdown Landing Dynamics' Simulation Considering Soft Ground," en, *Aerospace*, vol. 8, no. 12, p. 380, Dec. 2021, ISSN: 2226-4310. DOI: 10.3390/aerospace8120380. [Online]. Available: https://www.mdpi.com/2226-4310/8/12/380.

[148] H. W. Stone, *Mars Pathfinder Microrover: A Small, Low-Cost, Low-Power Spacecraft*, en, 1996. [Online]. Available: https://trs.jpl.nasa.gov/handle/2014/25424.

[149] T. J. Martin-Mur and B. Young, *Navigating MarCO, the first interplanetary CubeSats*, en, Feb. 2019. [Online]. Available: https://trs.jpl.nasa.gov/handle/2014/49242.

[150] L. Feruglio and S. Corpino, "Neural networks to increase the autonomy of interplanetary nanosatellite missions," en, *Robotics and Autonomous Systems*, vol. 93, pp. 52–60, Jul. 2017, ISSN: 0921-8890. DOI: 10.1016/j.robot.2017.04.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889016304419.

[151] J. H. Henninger, *Solar absorptance and thermal emittance of some common spacecraft thermal-control coatings*, Apr. 1984. [Online]. Available: https://ntrs.nasa.gov/citations/19840015630.

[152] H. Maleki, S. Al Hallaj, J. R. Selman, R. B. Dinwiddie, and H. Wang, "Thermal properties of lithium-ion battery and components," *Journal of the Electrochemical Society*, vol. 146, no. 3, p. 947, 1999.

[153] Z. Peterson, *FR4 Thermal Properties to Consider During Design*, en, Nov. 2020. [Online]. Available: https://www.nwengineeringllc.com/article/fr4-thermal-properties-to-consider-during-design.php.

[154] Z. Tian, S. Lee, and G. Chen, "A Comprehensive Review of Heat Transfer in Thermoelectric Materials and Devices," *Annual Review of Heat Transfer*, vol. 17, Jan. 2014. DOI: `10.1615/AnnualRevHeatTransfer.2014006932`.

[155] A. Raslan, G. Michna, and M. Ciarcià, "Thermal Simulation of a CubeSat," in *2019 IEEE International Conference on Electro Information Technology (EIT)*, May 2019, pp. 453–459. DOI: `10.1109/EIT.2019.8833736`.

[156] W. G. Schneeweiss, "Advanced Fault Tree Modeling," *Journal of Universal Computer Science*, vol. 5, no. 10, pp. 633–646, Oct. 1999. DOI: `10.3217/JUCS-005-10-0633`. [Online]. Available: `https://lib.jucs.org/article/27601/`.

[157] X. Olive, "FDI(R) for satellites: How to deal with high availability and robustness in the space domain?" eng, *International Journal of Applied Mathematics and Computer Science*, vol. 22, no. 1, pp. 99–107, 2012, ISSN: 1641-876X. [Online]. Available: `https://eudml.org/doc/208103`.

[158] L. Portinale, D. Codetta-Raiteri, A. Guiotto, and S. DiNolfo, "ARPHA: A software prototype for fault detection, identification and recovery in autonomous spacecrafts," *Acta Futura*, pp. 99–110, Jan. 2012. DOI: `10.2420/ACT-BOK-AF05`.

[159] Y. Kato, T. Yairi, and K. Hori, "Integrating Data Mining Techniques and Design Information Management for Failure Prevention," en, in *New Frontiers in Artificial Intelligence*, T. Terano, Y. Ohsawa, T. Nishida, A. Namatame, S. Tsumoto, and T. Washio, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2001, pp. 475–480, ISBN: 978-3-540-45548-6. DOI: `10.1007/3-540-45548-5_65`.

[160] *ECSS-E-ST-70-11C: Space engineering Space Segment Operability*, en, Jul. 2008.

[161] *Open-amp*, Apr. 2023. [Online]. Available: `https://github.com/OpenAMP/open-amp`.

[162] S. Jamal, S. Goyal, A. Grover, and A. Shanker, "Machine Learning: What, Why, and How?" In *Bioinformatics: Sequences, Structures, Phylogeny*, A. Shanker, Ed., Singapore: Springer Singapore, 2018, pp. 359–374. DOI: 10.1007/978-981-13-1562-6_16. [Online]. Available: https://doi.org/10.1007/978-981-13-1562-6_16.

[163] B. Murmann and B. Hoefflinger, *NANO-CHIPS 2030: On-Chip AI for an Efficient Data-Driven World*, en, ser. The Frontiers Collection. Cham: Springer International Publishing, 2020, ISBN: 978-3-030-18338-7. DOI: 10.1007/978-3-030-18338-7. [Online]. Available: http://link.springer.com/10.1007/978-3-030-18338-7.

[164] M. G. S. Murshed, C. Murphy, D. Hou, N. Khan, G. Ananthanarayanan, and F. Hussain, "Machine Learning at the Network Edge: A Survey," *ACM Computing Surveys*, vol. 54, no. 8, 170:1–170:37, Oct. 2021, ISSN: 0360-0300. DOI: 10.1145/3469029. [Online]. Available: https://doi.org/10.1145/3469029.

[165] M. Merenda, C. Porcaro, and D. Iero, "Edge Machine Learning for AI-Enabled IoT Devices: A Review," en, *Sensors*, vol. 20, no. 9, p. 2533, Apr. 2020, ISSN: 1424-8220. DOI: 10.3390/s20092533. [Online]. Available: https://www.mdpi.com/1424-8220/20/9/2533.

[166] The TensorFlow Authors, *TensorFlow Lite for Microcontrollers*, Apr. 2023. [Online]. Available: https://github.com/tensorflow/tflite-micro.

[167] A. Jara, P. Lepcha, S. Kim, H. Masui, T. Yamauchi, G. Maeda, and M. Cho, "On-orbit electrical power system dataset of 1U CubeSat constellation," en, *Data in Brief*, vol. 45, p. 108 697, Dec. 2022, ISSN: 2352-3409. DOI: 10.1016/j.dib.2022.108697. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352340922009027.

[168] Y. Li, *Deep Reinforcement Learning: An Overview*, Nov. 2018. DOI: 10.48550/arXiv.1701.07274. [Online]. Available: http://arxiv.org/abs/1701.07274 (visited on 10/23/2023).

[169] R. Furfaro, A. Scorsoglio, R. Linares, and M. Massari, "Adaptive generalized ZEM-ZEV feedback guidance for planetary landing via a deep reinforcement learning approach," *Acta Astronautica*, vol. 171, pp. 156–171, Jun. 2020, ISSN: 0094-5765. DOI: `10.1016/j.actaastro.2020.02.051`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0094576520301247` (visited on 10/30/2023).

[170] A. Harris, T. Teil, and H. Schaub, "Spacecraft Decision-Making Autonomy Using Deep Reinforcement Learning," en, 2019.

[171] X. Wang, J. Wu, Z. Shi, F. Zhao, and Z. Jin, "Deep reinforcement learning-based autonomous mission planning method for high and low orbit multiple agile Earth observing satellites," en, *Advances in Space Research*, vol. 70, no. 11, pp. 3478–3493, Dec. 2022, ISSN: 02731177. DOI: `10.1016/j.asr.2022.08.016`. [Online]. Available: `https://linkinghub.elsevier.com/retrieve/pii/S0273117722007335` (visited on 10/23/2023).

[172] J. Elkins, R. Sood, and C. Rumpf, "Autonomous Spacecraft Attitude Control Using Deep Reinforcement Learning," Oct. 2020.

[173] D. M. Chan and A.-a. Agha-mohammadi, "Autonomous Imaging and Mapping of Small Bodies Using Deep Reinforcement Learning," in *2019 IEEE Aerospace Conference*, Mar. 2019, pp. 1–12. DOI: `10.1109/AERO.2019.8742147`. [Online]. Available: `https://ieeexplore.ieee.org/abstract/document/8742147?casa_token=q_aA8MI5diUAAAAA:ygsORzfavSj33YJJ1JY3seAw4GQbs5ZqBNI4inTzJuSbXtqArPeQD1OIuhM2LXGgDmt3Ct4tOyk` (visited on 10/23/2023).

[174] C. Wilson and A. Riccardi, "Enabling intelligent onboard guidance, navigation, and control using reinforcement learning on near-term flight hardware," *Acta Astronautica*, vol. 199, pp. 374–385, Oct. 2022, ISSN: 0094-5765. DOI: `10.1016/j.actaastro.2022.07.013`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/S0094576522003502` (visited on 10/25/2023).

[175] T.-H. Guo and J. Nurre, "Sensor failure detection and recovery by neural networks," en, in *IJCNN-91-Seattle International Joint Conference on Neural Networks*, vol. i, Seattle, WA, USA: IEEE, 1991, pp. 221–226, ISBN: 978-0-7803-0164-1. DOI: 10.1109/IJCNN.1991.155180. [Online]. Available: http://ieeexplore.ieee.org/document/155180/.

[176] S. Jaekel and B. Scholz, "Utilizing Artificial Intelligence to achieve a robust architecture for future robotic spacecraft," in *2015 IEEE Aerospace Conference*, Mar. 2015, pp. 1–14. DOI: 10.1109/AERO.2015.7119180.

[177] V. U. J. Nwankwo, S. K. Chakrabarti, and R. S. Weigel, "Effects of plasma drag on low Earth orbiting satellites due to solar forcing induced perturbations and heating," en, *Advances in Space Research*, vol. 56, no. 1, pp. 47–56, Jul. 2015, ISSN: 0273-1177. DOI: 10.1016/j.asr.2015.03.044. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0273117715002562.

[178] STMicroelectronics, *FreeRTOS on STM32 - 3 FreeRTOS introduction*, Oct. 2019. [Online]. Available: https://www.youtube.com/watch?v=Svlv099-jRA.