Tjerand Bjørnsen

Software Architecture Design for the BioSat CubeSat

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2023

Tjerand Bjørnsen

Software Architecture Design for the BioSat CubeSat

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Abstract

The student organization Orbit NTNU has started work on the organization's third small satellite, the BioSat CubeSat. With this satellite Orbit NTNU aims to push the limits of their engineering expertise, as the satellite's main payload, called the BioBox, shall attempt to grow a plant from seed while in orbit. Another big focus of this mission is to improve on the previous satellite projects by designing a satellite bus that can easily be reused in future missions.

This thesis presents a layered, service-oriented software architecture design for the on-board software of BioSat, focusing on reusability, reliability, and maintainability. The suggested architecture mainly concerns the Command and Data Handling Subsystem of the satellite, but it also addresses the interaction between subsystems, as well as reliability measures. Architectural drivers such as overall design objectives, functional requirements, and non-functional requirements were identified and used as a basis when creating the software architecture design.

A self-contained development environment was created using Docker, and a software project using the Zephyr RTOS was created. A documentation generation system for system and software documentation was also set up using Sphinx. This was done to facilitate the implementation and further development of the suggested architecture.

The software architecture presented in this thesis is believed to facilitate the development of a reusable and maintainable satellite bus. This is achieved through employing a service-oriented architecture pattern, adding abstraction layers that make software independent from the hardware and the operating system, and using standardized services defined in the ECSS Packet Utilization Standard. The reliability of the system is increased by designing services and modules that are highly independent of each other and by employing error handling mechanisms such as watchdog timers, a satellite health monitoring service, and a Failure Detection, Isolation and Recovery module.

Sammendrag

Studentorganisasjonen Orbit NTNU har startet utviklingen av organisasjonens tredje småsatellitt: kubesatellitten BioSat. Med denne satellitten ønsker Orbit NTNU å presse grensene for sin ingeniørkompetanse, da satellittens hovednyttelast, som har fått navnet BioBox, skal forsøke å dyrke en plante fra et frø mens satellitten går i bane rundt jorda. Et annet mål for dette oppdraget er å ta i bruk kunnskapen og erfaringene fra Orbits tidligere satellittprosjekt. Et hovedmål knyttet til dette er å designe en satellittbuss som enkelt kan gjenbrukes i fremtidige oppdrag.

Denne masteroppgaven presenterer en lagdelt, tjenesteorientert programvarearkitektur for programvaren ombord på BioSat, med fokus på gjenbrukbarhet, pålitelighet og vedlikeholdbarhet. Den foreslåtte arkitekturen tar hovedsakelig for seg satellittens subsystem for kommando og datahåndtering (kalt CDHS), men tar også for seg samspillet mellom de forskjellige delsystemene, samt strategier for å øke systemets pålitelighet. Overordnede designmål, funksjonelle krav og ikke-funksjonelle krav ble identifisert og brukt som grunnlag for utformingen av programvarearkitekturen.

Et selvstendig utviklingsmiljø ble opprettet ved hjelp av Docker, og et programvareprosjekt med Zephyr RTOS ble også opprettet. Et dokumentasjonssystem for system- og programvaredokumentasjon ble opprettet ved å bruke programvaren Sphinx. Disse tiltakene ble gjort for å gjøre implementeringen og videreutviklingen av den foreslåtte arkitekturen lettere.

Programvarearkitekturen som presenteres i denne oppgaven antas å gjøre det lettere å utvikle en gjenbrukbar og vedlikeholdbar satellittbuss. Dette oppnås ved å bruke et tjenesteorientert arkitekturmønster, anvende abstraksjonslag som gjør programvaren uavhengig av maskinvaren og operativsystemet, og bruke standardiserte tjenester definert i ECSS sin Packet Utilization Standard. Systemets pålitelighet økes ved å designe tjenester og moduler som er til en stor grad uavhengige av hverandre, og ved å bruke feilhåndteringsmekanismer som watchdogtimere, en dedikert tjeneste for overvåking av satellittens tilstand, og en modul for deteksjon, isolering og gjenoppretting av feil.

Preface

This Master's thesis was conducted during the spring of 2023, as a part of the Master's programme Cybernetics and Robotics at the Norwegian University of Science and Technology in Trondheim. It was written for the student organization Orbit NTNU, and concerns the design of a software architecture for the on-board software of the organization's next satellite project, BioSat.

I went into this project expecting it to be a challenge, and boy was I right. Designing a software architecture more or less from scratch was uncharted waters for me, and it soon became clear that the waves were higher than I first thought. Nevertheless, thanks to the members of Orbit NTNU that would lend me an ear and gladly engage in discussions about whatever I was cooking up, I eventually made it back to shore, and with some treasure to boot. The exact value of that treasure I'm still not sure about, but it looks nice, at least.

I want to thank my supervisor, Sverre Hendseth, for great advice about structuring my thesis, and for motivation at the end of the semester.

Thanks to my fellow team members in the Embedded team at Orbit NTNU for being a great bunch of nerds. Thanks to the members of the software architecture task force that tried to help me understand what the heck I should be doing. Huge thanks to Sigmund Klåpbakken for always having an answer to or being ready to discuss any technical questions I might have had. And a heartfelt thanks to the bouldering gang for keeping me (partially) sane throughout the semester.

I want to thank my family for their support throughout the whole endeavor, even though they didn't have a clue about what I was talking about most of the time. Lastly, I want to give a special thanks to my girlfriend Cecilie Torp Dahl, for her invaluable support throughout the semester, and for her heroic effort trying to make my thesis look pretty.

Contents

Abstract i
Sammendrag ii
Preface iii
List of Figures viii
List of Tables x

1 Introduction **1**
1.1 Orbit NTNU 1
1.2 The BioSat Mission 2
1.2.1 CubeSats 2
1.2.2 BioSat Concept of Operations 3
1.3 Task Description 4

2 Background **5**
2.1 The BioSat Satellite 5
2.1.1 Subsystems 6
2.1.2 On-Board Computer - OBC 9
2.1.3 On-Board Software - OBS 9
2.2 Zephyr RTOS 11
2.2.1 Capabilities 12
2.2.2 Toolchain 13
2.2.3 Configuration 16
2.2.4 Documentation 17
2.3 Docker 18
2.3.1 Docker Architecture 18
2.3.2 Dockerized Development Environment 20
2.4 Packet Utilization Standard 20
2.4.1 Important PUS Terms 21
2.4.2 PUS Context 23
2.4.3 The Space System Service Model 24
2.4.4 The PUS Foundation Model 24
2.4.5 Standard Service Types 25
2.4.6 Mission-Specific Service Types 27
2.5 Cubesat Space Protocol 27
2.5.1 Protocol Stack 28
2.5.2 Notable Features 30

3 Background - Software Architecture **31**
3.1 Faults, Errors and Failures . . . 31
3.2 Software Quality . . . 32
3.2.1 Maintainability . . . 33
3.2.2 Reliability, Robustness, and Resilience . . . 33
3.3 Software Architecture . . . 35
3.3.1 Architectural Drivers . . . 36
3.3.2 Software Architecture Design . . . 38
3.3.3 Software Architecture Patterns . . . 39

4 Identifying the Architectural Drivers **43**
4.1 Design Objectives . . . 43
4.2 Primary Functional Requirements . . . 45
4.3 Software Quality Requirements . . . 47
4.3.1 Maintainability Requirements . . . 47
4.3.2 Reliability Requirements . . . 47
4.3.3 Testability Requirements . . . 48
4.3.4 Reusability Requirements . . . 48
4.3.5 Portability Requirements . . . 48
4.3.6 Efficiency Requirements . . . 49
4.4 Constraints . . . 50
4.4.1 Technical Constraints . . . 50
4.4.2 Non-Technical Constraints . . . 50

5 The BioSat Software Architecture **51**
5.1 Choice of Architecture Pattern . . . 51
5.2 Architecture Design . . . 52
5.2.1 Communication . . . 53
5.2.2 Failure Detection, Isolation and Recovery – FDIR . . . 56
5.2.3 The Low-Level Software Layer . . . 57
5.2.4 The Service Layer . . . 58
5.2.5 The Application Layer . . . 60

6 Detailed Design **62**
6.1 Application Layer . . . 62
6.1.1 Mode Manager . . . 62
6.1.2 Autonomy Manager . . . 63
6.1.3 System-Level FDIR Manager . . . 63
6.2 Service Layer . . . 63
6.2.1 Messaging Service . . . 63
6.2.2 Logging Service . . . 65
6.2.3 Time Synchronization Service . . . 66
6.2.4 CSP Service . . . 67
6.2.5 File Transfer Service . . . 68
6.2.6 Hardware Abstraction Layer - HAL . . . 68
6.2.7 Operating System Abstraction Layer - OSAL . . . 69

6.2.8 Packet Utilization Standard Services - PUS Services 69
6.3 Low-Level Software Layer 77
6.3.1 Bootloader 77
6.3.2 Hardware Drivers 77

7 Development 78
7.1 Creating a Development Environment Using Docker 78
7.1.1 Docker Image 78
7.2 Setting Up Zephyr RTOS for the STM32H745 80
7.3 Software Project Structure 80
7.4 Enabling Debugging Support 80
7.5 Documentation 81

8 Discussion 82
8.1 Does the Software Architecture Design Satisfy the Initial Task? 82
8.2 Hard Decisions and Uncertainties 83
8.3 Future Work 85

9 Conclusion 88

References 90

A BioSat Requirements Specification 94
A.1 Functional Requirements 95
A.1.1 General Functional Requirements 95
A.1.2 CDHS Functional Requirements 96
A.2 Non-Functional Requirements 97
A.2.1 Maintainability 97
A.2.2 Reliability 98
A.2.3 Testability 98
A.2.4 Reusability 98
A.2.5 Portability 99
A.2.6 Efficiency 99

B Development Environment Files 100
B.1 Dockerfile 100
B.2 Docker Compose File 103
B.3 VS Code Dev Container File 104
B.4 VS Code Debugging Support 105
B.4.1 launch.json 105
B.4.2 debug.conf 106
B.4.3 tasks.json 107

Abbreviations

ADCS	Attitude Determination and Control System
API	Application Programming Interface
APID	Application Process IDentifier
CAN	Controller Area Network
CDHS	Command and Data Handling System
ConOps	Concept of Operations
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
CSP	Cubesat Space Protocol
ECC	Error Code Correction
ECSS	European Cooperation for Space Standardization
EPS	Electrical Power System
FDIR	Failure Detection, Isolation and Recovery
FPGA	Field-Programmable Gate Array
GNSS	Global Navigation Satellite System
HAL	Hardware Abstraction Layer
HMAC	Hash-based Message Authentication Codes
IMU	Inertial Measurement Units
MCU	MicroController Unit
MPU	Memory Protection Units
MTTF	Mean Time To Failure
MTTR	Mean Time To Recovery
MTU	Maximum Transfer Unit
OBC	On-Board Computer
OBCP	On-Board Control Procedure
OBS	On-Board Software
OpenOCD	Open On-Chip Debugger
OSAL	Operating System Abstraction Layer
PCB	Printed Circuit Board
PUS	Packet Utilization Standard

RDP	Reliable Datagram Protoco
RTC	Real-Time Clock
RTOS	Real-Time Operating Systems
SDK	Software Development Kit
SDR	Software-Defined Radio
SFP	Small Fragmentation Protocol
SMP	Symmetric MultiProcessing
SOA	Service-Oriented Architecture
SPI	Serial Peripheral Interface
SPP	Space Packet Protocol
ST	Service Type
TCS	Thermal Control System
TLE	Two-Line Element
TT&C	Telemetry, Tracking and Command
UART	Universal Asynchronous Receiver-Transmitter
UDP	Unreliable Datagram Protocol
UHF	Ultra-High Frequency
Ztest	Zephyr Test Framework

List of Figures

1.1 Standard CubeSat sizes 3

2.1 Internal structure and components of BioSat 5
2.2 Structure of a West workspace 14
2.3 Docker architecture 18
2.4 Docker containers 19
2.5 Communication between a subservice provider and a subservice user hosted on-board and on-ground 24
2.6 Example CSP network configuration 28

3.1 Relation between faults, errors and failures 31
3.2 Layered architecture of an embedded system without and with an OS 40
3.3 Layered architecture of an embedded system with a HAL and with both a HAL and an OSAL 41

5.1 Chosen software architecture pattern 52
5.2 Software architecture of BioSat's on-board software 53
5.3 Example CSP network configuration 54
5.4 From request issuing to execution 55

6.1 Internal messaging using the Messaging Service 64
6.2 External messaging using the Messaging Service 65

List of Tables

2.1 PUS standard service types 26
2.2 The CSP services 30

5.1 Selected PUS services 59

A.1 Functional requirements for a generic subsystem 95
A.2 Functional requirements for the CDHS 96
A.3 Maintainability requirements 97
A.4 Reliability requirements 98
A.5 Testability requirements 98
A.6 Reusability requirements 98
A.7 Portability requirements 99
A.8 Efficiency requirements 99

Chapter 1

Introduction

1.1 Orbit NTNU

Orbit NTNU is a volunteer student organization with the goal of educating future engineers of the space industry. The organization works towards this goal by building small satellites of the CubeSat format, providing the members with hands-on experience by designing, developing, testing and operating the satellites. Orbit NTNU is located at the Norwegian University of Science and Technology, and consists of a highly multidisciplinary group of students.

Orbit NTNU has almost two satellites under its belt and a third currently in the design phase, namely SelfieSat, FramSat-1, and BioSat, respectively. SelfieSat was launched in May 2022 and is still operational and performing its mission objective (which is taking selfies from space) on a daily basis. FramSat-1 is nearing the end of its development cycle and is scheduled for launch in late 2023. BioSat, which is the satellite this thesis concerns itself with, is the next big CubeSat project of the organization. BioSat is still in the design phase of development and is scheduled for launch sometime in 2025.

The members work in teams, each team specializing in and being responsible for a specific part of the satellite (i.e. the different subsystems) or of the organization. Students can apply for a position in the organization in any term of their education at NTNU. Since students that have just started on their programme, students that have studied for almost five years, and anyone in between, can join the organization, the initial experience and knowledge of members vary greatly. Because this thesis will likely act as a base for large parts of the design and development efforts around BioSat's on-board software, this thesis tries to acknowledge the different backgrounds the readers might have. This is the reason why the background chapters are as extensive as they are, which hopefully will make the design presented here more accessible to all members of the organization.

1.2 The BioSat Mission

The BioSat mission has the lofty ambition of growing a plant from seed inside a small satellite in space. This mission was chosen because of the challenging nature of the mission objective, the vast learning opportunities it presents, and to contribute to a greater goal of finding solutions to sustaining human life in the increasingly lengthy manned space missions of the near future.

The BioSat satellite will be designed and constructed with this goal in mind. The satellite is a 3U CubeSat with in-house built and designed components, subsystems, and physical structure, and some components that are commercial-off-the-shelf (i.e. purchased and used as-is). The main mission payload is the BioBox, which is a sealed chamber with built-in temperature control, growth light, sensors, and a camera. This is where the plant will be grown from seed. A secondary Software-Defined Radio (SDR) is also on-board the satellite, providing many new possibilities related to communication with the satellite.

The main goal of Orbit NTNU as an organization is to provide its members with unique experience, knowledge and challenges related to space systems engineering. Thus, the main mission objective of the BioSat mission is to create an improved satellite bus (i.e. the main parts of the satellite/not the payload) that builds on the experience and the lessons learned from the previous missions, and to successfully operate it after it has been launched into orbit. The mission objective of growing a plant in space is prioritized after the organization's main education objective. However, it presents a major challenge that is relevant to the efforts being made by organizations such as NASA related to manned missions to the Moon and to Mars. This objective, as well as the secondary payload that is the SDR, represents the "science" objectives of the mission, and are what would be the main objectives in an actual commercial mission. The mission has several other mission objectives that are related to engineering and technology challenges, organizational aspects, and educational goals (Orbit NTNU, 2023*a*).

Below is a brief introduction to the CubeSat concept, which is then followed by the mission's concept of operation.

1.2.1 CubeSats

Cube satellites, or CubeSats, are a group of small satellites that adhere to the CubeSat standard. CubeSats adopt a standardized size and form factor and consist of one or more cubes with sides of 10 cm. A 1U CubeSat is a 10 cm x 10 cm x 10 cm cube with a mass of up to 2 kg. Other sizes are composed of "stacks" of 1U units (California Polytechnic State University, 2020, p. 7). The standardized form factor facilitates the development of CubeSat-compatible off-the-shelf components. An organization developing a CubeSat could for example buy a complete Electrical Power System (EPS, which provides battery power to the rest of the satellite).

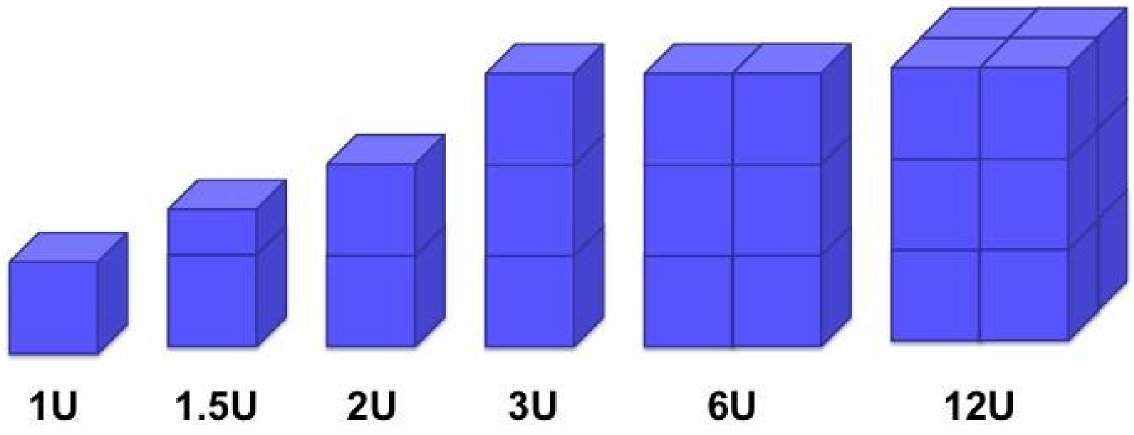

Figure 1.1: Standard CubeSat sizes.

The small size and standardized form factor of CubeSats make them convenient to deploy into their target orbit using a CubeSat deployer. A deployer is a larger spacecraft that contains several smaller satellites, and that is responsible for deploying each satellite to its intended orbit. This method of deployment increases the number of launch opportunities and keeps the cost of launching a small satellite into orbit relatively low.

1.2.2 BioSat Concept of Operations

The concept of operations (ConOps) describes the characteristics of the systems involved in the BioSat mission, which consists of the plant, the life support system, the plant monitoring system, and the communication systems (Orbit NTNU, 2023*b*, p. 4). The ConOps also describe how the satellite is operated after launch in order to fulfill the mission objectives.

The general ConOps can be broken down into the following steps (Orbit NTNU, 2023*b*, p. 5):

1. **Preparation of commands**: Script files or sequences of commands are generated on ground by the operators. These contain actions or commands that should be executed in a specific order in order to produce, process, and downlink payload data.

2. **Uplink commands and configuration data**: The script files/command sequences generated in the previous step are uplinked (i.e. uploaded) to the satellite when it is in range of a ground station it can communicate with.

3. **Plant monitoring and sensors**: The commands are executed according to the uploaded scripts/sequences. This involves plant monitoring operations (creating telemetry packets containing info about the plant's health and the conditions in the BioBox) and taking a picture using the camera inside the BioBox every other hour. The payload data (plant/BioBox status and images) is transferred to the Command and Data Handling Subsystem, where it will be stored until the satellite is in range of a ground station.

4. **Autonomous processing of sensor data**: Some data is processed on-board, such as images being compressed to save downlink bandwidth or payload telemetry being monitored to autonomously detect failures or anomalies.

5. **Downlink telemetry and payload data**: Once the satellite is in range of one of the ground stations, and after the operators have uploaded a new command schedule, the generated telemetry packets and payload data can be downlinked from the satellite to the ground.
6. **Store and process data**: The received telemetry and images are stored and processed on ground. The telemetry is used to monitor the satellite's and the plant's health and operational status. The images can be visually inspected to give indications of the plant's growth and health.

1.3 Task Description

This project was carried out for and in collaboration with the NTNU-based student organization Orbit NTNU, which aims to educate its members in space systems engineering through designing, developing, and operating CubeSats.

The main task of the project is to research and design a suitable software architecture for the on-board software of Orbit NTNU's next satellite project, BioSat. BioSat is Orbit NTNU's third CubeSat project and represents a big step in the organization's goals related to project quality and complexity.

The experience from two previous satellite projects has shown that there is much to gain by focusing on designing a satellite where reusability is a priority. Attempts to reuse software from SelfieSat in FramSat-1 only partially succeeded and resulted in much development time being wasted and large parts of the on-board software being rewritten. The main goal of this task is therefore to design an architecture that lends itself to reusability, where reusable software and mission-specific software are clearly separated.

Most of BioSat's subsystems are still in the concept phase when it comes to their required functionality and attributes. Since the Command and Data Handling System (CDHS) is central to the satellite's operation, the suggested architecture should mainly be concerned with the CDHS. Important design decisions regarding the system-level functionality and design should also be explored and included in the architecture, such as communication between subsystems, handling of telecommands, and reliability measures.

To sum up the key tasks of the project:

- Research what makes up a good software architecture, and how software architectures can facilitate and improve various aspects of a software system.
- Create a list of requirements and design goals that will be used as guidelines when designing the software architecture.
- Suggest a software architecture for the BioSat satellite with a main focus on the CDHS. The software architecture should promote reusability and other qualities that are important in any satellite, such as reliability and maintainability.

Chapter 2

Background

This chapter addresses the technical aspects of the BioSat satellite, as well as the third party software that in some way plays a significant role in the design of the software architecture.

2.1 The BioSat Satellite

The satellite that will be used for the BioSat mission is a 3U CubeSat with deployable solar panels and antennas. It contains two payloads: the main payload, which is the BioBox (see section 2.1.1), and the secondary payload, which is an SDR (see section 2.1.1). An illustration of the satellite can be seen in Fig. 2.1.

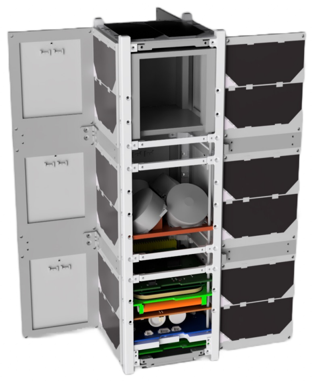

Figure 2.1: Render of BioSat, showing the internal structure and components.

As mentioned in the introduction, BioSat represents a shift in Orbit NTNU's strategy when it comes to software and design reuse. The parts of the satellite that perform general functionality which should or must be present in every satellite, and that are independent of the BioSat mission, are separated from the mission specific parts of the satellite. These mission independent parts make up what will be a framework or platform that future missions can use to implement their respective payloads. The name of this framework or platform is currently undecided, but will be addressed as the satellite bus in the rest of this thesis.

Since this thesis is mainly concerned with the design of the software architecture, where the CDHS is central, only the hardware used by the CDHS is described, namely the On-Board Computer (OBC). Another reason for this is that which hardware will be used by the other subsystems is still largely undecided.

2.1.1 Subsystems

The satellite can be divided into subsystems that each perform a specific set of operations. These subsystems are designed to be as loosely coupled and independent as possible in order to promote reusability and to keep the complexity of the system low. Many of the subsystems contain very little to no functionality that is specifically related to the BioSat mission, and thus could be a good fit for the satellite bus. Different satellite missions have different functional requirements for the various subsystems. For example, reaction wheels in a 1U satellite would be unfeasible due to space limitations. Thus, the subsystems that do become part of the satellite bus framework must be designed with modularity and configurability in mind, such that only the wanted functionality can be enabled.

In some cases, each subsystem has its own dedicated board or microcontroller, but this is not a requirement and will depend on the specific implementation in each satellite. For example, the Attitude Determination and Control System (ADCS) is currently set to be hosted on the OBC, which will also host the CDHS. Ideally, the software for each subsystem should be designed in such a way that it can be reused on any hardware configuration, without having to do major rewrites.

The various teams in Orbit NTNU each have responsibility for up to one subsystem and its design. The teams work somewhat independently on the internal design of the subsystems, and have to cooperate on the design of the interfaces between them. During the writing of this thesis, very little design work has been done for the subsystems (the exception being the CDHS, whose design is one of the main subjects of this thesis). The descriptions of the various subsystems found below therefore only briefly describe the general purpose and behavior of each subsystem as was presented during Orbit NTNU's internal Mission Concept Review that was held on the 24th of April, 2023.

Telemetry, Tracking and Command - TT&C

The telemetry, tracking and command system is responsible for the communication between the ground segment and the satellite, i.e. receiving telecommands from the ground station, determining the location and velocity of the satellite by tracking ranging signals, and transmitting telemetry containing mission data to the ground station. This is done over radio communication.

When talking about sending data to the satellite, the term *uplink* is often used. On the uplink, commands called *telecommands* are issued to the satellite, which are passed to the CDHS for processing. Telecommands include application specific commands, i.e. commands to the payloads in order to perform mission related operations, and commands to the satellite bus, i.e. commands that are not directly related to the mission, but rather to the configuration and operation of the spacecraft.

Transmitting data from the satellite to the ground station is done over the *downlink*. The data transmitted on the downlink is called telemetry, and include sensor data and health reports from the spacecraft, reception, acception and execution statuses related to telecommands, and data generated by any payloads.

Command and Data Handling System - CDHS

The command and data handling system can be considered as the “brain” of the satellite. It is responsible for handling commands from the ground segment related to the operation of the satellite, monitoring the satellite’s health, processing and storing on-board parameters and data, preparing and downlinking telemetry and payload data to the ground segment, synchronizing and distributing the on-board time, autonomously operating the satellite when outside of ground station range, and more. In short, it is responsible for managing the operations of the satellite and handling the data products produced by the various subsystems.

Electrical Power System - EPS

The electrical power system includes the battery and the solar panels. It is responsible for distributing power throughout the satellite, performing relevant measurements such as battery voltage and temperature, generating telemetry packets containing said measurements, and generating events notifying the CDHS when some value exceeds some critical threshold.

The EPS has watchdog timers that are used to ensure the CDHS (and possibly other subsystems) are working correctly. If a command or signal is not sent to the EPS to reset the watchdog timer before the deadline expires, the EPS will turn that subsystem (usually CDHS) on and off again, hopefully resolving the issue.

Attitude Determination and Control System - ADCS

The attitude determination and control system is responsible for estimating where the satellite is pointing (called its attitude/orientation), estimating the satellite’s position in its orbit (called orbit propagation) using a combination of sensor data and orbital parameter datasets (called Two-Line Element (TLE) sets), and control the satellite’s attitude to a specified attitude (e.g. pointing towards the sun for charging or towards the Earth to communicate with the ground segment).

The ADCS utilizes a range of different sensors and actuators in order to perform its functions. Common sensors include sun sensors that measure the angle between the satellite’s orientation and the sun, Inertial Measurement Units (IMU) that measure the acceleration and angular velocity of the satellite, GNSS receivers that are used to obtain the satellite’s position in space, and, less commonly, star trackers that use the location of the stars in order to determine the satellite’s position. Common actuators include thrusters that expel some sort of fuel in order to move or orient the satellite, magnetorquers that induce a momentum on the satellite by generating a magnetic field that interacts with the Earth’s magnetic field, and reaction wheels that generate momentum by accelerating wheels that are mounted in different orientations.

Because of the time sensitive nature of attitude control algorithms, and the large number of hardware devices that has to be interacted with in order to perform them accurately, the ADCS contains (some of) the most time-critical tasks on the satellite. Being able to guarantee that the deadlines of these tasks are met is essential in order to achieve good attitude determination and control accuracy.

Thermal Control System - TCS

The thermal control system ensures the different parts of the satellite don't exceed their respective temperature ratings. A thermal control system may consist of active thermal components and passive thermal components, as well as the sensors needed to monitor the temperature of the different parts of the satellite. Passive thermal components are components that can't be controlled, with examples being heat sinks or reflective paint. Active thermal components are components that *can* be controllers, and include electric heating components (resistors) in order to generate heat and water-cooling systems that use pumps and radiators in order to dissipate heat.

Temperature sensors are placed in key locations within the satellite, such as near CPUs or other power-hungry components which can generate large amounts of heat. When some temperature threshold is exceeded, and the thermal control system fails to regulate it back to a safe value, an event is generated and sent to the CDHS where the problem is dealt with by the Failure Detection, Isolation and Recover (FDIR) functions. A typical response to an over-heating event is temporarily shutting off heat-generating components. A method that is used to distribute the heat from the sun is to ensure that the satellite has a constant (but small) angular velocity. This maneuver is performed by the ADCS, and ensures that the sun hits several of the sides of the satellite, distributing the heat evenly.

Mechanical Structure and Mechanisms

The mechanical structure is responsible for housing the physical components of the satellite. It provides rigidity that protects the other parts of the satellite from the heavy vibrations the satellite experiences during launch, and it provides structure that supports sensors, PCBs, actuators, batteries, wires, solar panels and any other components.

Mechanical mechanisms are devices whose functionality revolves around movement. BioSat will have deployable solar panels along its long sides, which have to be deployed (i.e. opened up/swung out) after the satellite is in orbit. The mechanical mechanisms also include the deployable antennas, which must be deployed in order to increase the signal strength of the radio and allow communication between the satellite and the ground.

BioBox Payload

The BioBox is the primary payload of BioSat. It will be an incubation and growth chamber used to grow a plant from seed in space. This payload includes a life support system and a plant monitoring system that controls the plant's environment in order to ensure the plant grows and survives. To achieve this there will be sensors enabling monitoring of the plant's health and condition, including a camera which will provide visual data of the plant, as well as some form of temperature regulation and light source.

SDR Payload

An SDR operating in the UHF range will be the secondary payload of BioSat. The SDR will be able to communicate with ground stations and perform measurements and signal processing. The SDR can easily be reconfigured by uploading new modulation schemes.

2.1.2 On-Board Computer - OBC

The on-board computer is the main processing unit of the satellite. The term is usually used for the entire microcontroller board that runs the central functionality responsible for the operation of the spacecraft. This is usually where the CDHS (and sometimes also the ADCS) is hosted. The OBC consists of the microcontroller and any peripherals connected to it, such as external memory and storage, communication bus controllers, stack connectors, IMU and other sensors, a Real-Time Clock (RTC) and the more traditional components (resistors, capacitors, etc.) required by the peripherals and the MicroController Unit (MCU).

The design of the OBC is an ongoing process, with the first revision still in development. A trade-off study has been conducted by other members of Orbit NTNU evaluating which MCU is most suited. The 32-bit ARM-based microcontroller STM32H745 was selected as the main MCU for the OBC (Orbit NTNU, 2023*d*). The provided reasons for choosing this MCU were:

- A wide range of communication interfaces.
- High performance; dual core ARM processor (a 480 MHz ARM Cortex-M7 core and a 240 MHz ARM Cortex-M4 core).
- Adequate amounts of internal RAM and Flash: 1 MB RAM and 2 MB Flash.
- RAM and Flash protected by Error Code Correction (ECC), which is a method that can correct random bit flips in memory.
- Adequate temperature ratings of $-40°C$ to $125°C$.
- Is supported by Real-Time Operating Systems (RTOS) such as Zephyr and FreeRTOS.
- Memory Protection Units that can protect against untrusted user programs corrupting data used by the OS, or to protect data processes or to read-protect memory regions.

2.1.3 On-Board Software - OBS

The On-Board Software (OBS) is any software that is running on the satellite. Since the functionality of most of the other subsystems isn't entirely decided yet, there are many decisions regarding those systems' software design and implementation that can't be made yet.

On BioSat the OBS will be distributed between several boards or microcontrollers, the largest (and with most processing power) being the OBC. In the current concept for the other subsystems, the EPS, the TT&C subsystem, the BioBox and the SDR will all run on their separate units. There will therefore be a need for the subsystems to be able to communicate with each other. The proposed solution is to connect the boards through a Controller Area Network (CAN) bus, and have the subsystems communicate with each other using the Cubesat Space Protocol (CSP). CSP is described in section 2.5.

OBC Operating System

One of the few aspects that *is* decided is the operating system that will be used on the OBC, namely the Zephyr RTOS. This was decided based on a trade-off study (Orbit NTNU, 2023*e*) conducted by other members of Orbit NTNU, comparing Zephyr, FreeRTOS, NuttX and ErlendOS (the custom FreeRTOS-based, Unix-like OS that was used on the first satellites developed by Orbit NTNU). The decision to move from ErlendOS to a more mainstream OS was made partly due to some problems related to RAM usage and clock instability, but mostly due to the fact that ErlendOS was developed and maintained almost exclusively by one person. Due to the members' inexperience with kernel development, bugfixes were sometimes very slow and halted development completely in some cases. The decision was therefore made to move to a more mainstream, better supported OS: Zephyr.

The trade-off study found the following pros for moving to Zephyr:

- Great documentation and support.
- Open-source and free.
- Built-in support for a wide array of boards and microprocessors.
- Mature ecosystem with many existing device drivers, libraries, etc.
- Small memory footprint.
- Support for writing tests and unit-tests using the Zephyr Test Framework (Ztest).
- Support for the chosen STM32H745 microcontroller.
- Comprehensive and relatively easy-to-install toolchain.
- Modular kernel that facilitates reuse in future missions.
- Industry relevant, heavily utilized by Nordic Semiconductor, which is one of Orbit NTNU's sponsors. Possible to get help or support from Nordic. Relevant experience for members' future careers.
- Real-time capabilities with support for multithreading, inter-thread synchronization and communication, etc.
- Facilitate development by providing useful features such as device drivers, memory management, shell access, hardware abstraction, etc. See section 2.2.1 for a more comprehensive list.

The following cons were found:

- No dynamic loading: update requires compiling, uploading and booting entire kernel images with the application built into the image, instead of compiling and uploading only the updated module. Will increase compile times and bandwidth usage during uplinking, and requires a bootloader that can handle image upgrades.
- Zephyr can pose an intimidating challenge for less experienced/newer members, as the learning curve is rather steep compared with the Unix-based ErlendOS that was used before.

Zephyr was chosen over the other candidate OSes due to it scoring higher on the selection criteria that were used in the trade-off study, and which corresponds with the pros and cons listed above.

2.2 Zephyr RTOS

Zephyr is a real-time operating system, primarily written in the C programming language. It is based on a small-footprint kernel for use on resource-constrained and embedded systems, with support for a wide range of different architectures.

Most operating systems provide the capability to seemingly run programs in parallel (called multi-tasking), even if there is only one core (or CPU) to run the code on. In systems where there is only one core only one thread of execution can be run at any given point in time, meaning that programs are not actually running in parallel, they just appear to be. The operating system achieves this by using a part of the OS called a scheduler. The scheduler is responsible for rapidly switching between each program, allowing the processor to perform multiple tasks in "parallel". A real-time operating system is designed to provide predictable (deterministic) scheduling of tasks. This is useful for systems (often embedded systems) that have real-time requirements, meaning that some tasks may have strict deadlines that may be met. The deterministic nature of the scheduler allows guaranteeing that the real-time tasks meet their deadlines. This behavior is achieved by, among other methods, providing the user the ability to assign execution priority to the different tasks. (freeRTOS, 2023)

The Zephyr OS is based on a monolithic kernel, which means that all parts of the OS operate in kernel space. Any code written by the user also operates in the same kernel space by default. This exposes the kernel to potential programming errors which could corrupt the kernel or other otherwise independent applications (Zephyr Project, 2023*k*). It does mean, however, that no context switches are required when making system calls. This will save resource use and increase the performance of the system. Zephyr can provide user space and kernel space separation to MCUs that have Memory Protection Units (MPUs), avoiding the risk of application code corrupting the kernel.

When building Zephyr-based applications, the application-specific code is combined with the OS code and compiled into a monolithic binary image (Zephyr Project, 2023*e*). The benefits of this approach are preventing execution of malicious software that is dynamically loaded during runtime, as well as removing some overhead in the kernel needed to otherwise load and unload the dynamically loaded software. A drawback to this approach is related to issuing updates to deployed systems, e.g. when uploading a bug fix. Instead of compiling and uploading only the module that needed fixing, the entire application (including the OS) has to be compiled into the binary image, which then has to be uploaded in its entirety to the device. This may drastically increase the bandwidth usage of uploading updates, which can be problematic in systems with limited bandwidth links (such as a student satellite in orbit). Another drawback is the risk of corrupting the entire system with a faulty update. As will be discussed later, this risk can be mitigated by utilizing a bootloader such as MCUboot.

Zephyr OS has great support and documentation, and is used by several major industry actors such as Nordic Semiconductor, Microchip, and more. In the sections below follow descriptions of some of the prominent capabilities of Zephyr, the toolchain used to build Zephyr applications, as well as the configuration systems used to configure the kernel and the application when building the application.

2.2.1 Capabilities

The list of capabilities below is gathered from the official Zephyr Project (2023*e*) documentation, and contains the capabilities deemed most relevant for a satellite system.

- **Multi-threading services** for cooperative, priority-based, non-preemptive and preemptive threads, with choices for several scheduling algorithms.
- **Compile time registration of interrupt handlers** with support for nested interrupts.
- **Memory allocation services** for dynamically allocating fixed-size or variable-size memory blocks.
- **Inter-thread synchronization** such as binary semaphores, counting semaphores, and mutex semaphores.
- **Inter-thread data passing services** for message queues and byte streams.
- **Power management services** such as tickless idle and an advanced idling infrastructure.
- **Highly configurable**, allowing applications to only include the capabilities of the OS that is needed using highly flexible configuration systems, minimizing the memory footprint of the OS.
- **Cross architecture**: Support for a large number of different boards with different processor architectures, as well as the developer tools needed to develop for them.
- **Optimized device driver model**: A consistent device model for configuring the drivers of the system, initializing the configured drivers and allowing reuse of drivers across platforms that have common devices.
- **Devicetree support**, which can be used to describe the hardware available to the system, and provides handles that can be used when writing software.
- **Native networking stack** with support for several protocols, among others BSD sockets.
- **Partial native POSIX support**, allowing running Zephyr as a Linux application, which aids in development and testing.
- **Virtual file system with LittleFS and FATFS support**.
- **Powerful logging framework** with support for log filtering, object dumping, panic mode, multiple backends (memory, filesystem, console, ...), logging levels and integration with the Zephyr-provided shell.

- **User friendly and full-featured shell interface** with features such as autocompletion, wildcards, coloring, metakeys (arrows, backspace, etc.) and history.
- **Store settings in non-volatile storage**: provides a way to persistently store device configuration settings and runtime state as key-value pairs.
- **Non-volatile storage support**, allowing storage of binary blobs, strings, integers, longs, and any combination of these.

Additionally, Zephyr has support for **Symmetric MultiProcessing** (SMP). SMP is similar to multitasking, except that the different tasks (or threads) *are* executed in parallel. To use SMP, a CPU with two or more cores is necessary. Symmetric means that no core is treated specially by the scheduler, and tasks are distributed between the cores automatically. Zephyr has an optional feature called CPU masking, which enables specifying which physical core a given task can be executed on (Zephyr Project, 2023*i*). This is an important feature, as it will enable running the real-time tasks with strict deadlines (such as ADCS algorithms and telecommand handling) on a dedicated core, increasing the responsiveness of the system.

2.2.2 Toolchain

There are several software tools that are used when managing a Zephyr project, building the application and flashing it to a device. This section contains descriptions of the various tools and their usage in the Zephyr development toolchain.

A brief outline is as follows: West is used to manage the project's repository and any dependencies, as well as facilitate building the application and flashing it. To build the application, West initiates a build using CMake, which will generate and coordinate build files used by the build system Ninja. When the application is built, the resulting binary must be uploaded, or flashed, to the device that will run the application. West can then be used to initiate the flashing process, where the binary is then finally uploaded to the device using a flash software tool, such as J-Link or OpenOCD. Optionally, a bootloader such as MCUboot can be flashed to the device first, which enables some extra capabilities such as safely updating the application on the device.

West

West is Zephyr's command line tool for multi-repository management. It also provides conveniences for building, flashing and debugging Zephyr applications. (Zephyr Project, 2023*l*)

West controls dependencies through files called manifest files, typically named *west.yml*. This file contains definitions for all the other projects the main project depends on, as well as what Git commit each project should be checked out to.

When the command "west update" is run, west will read the *west.yml* file, download/update any dependencies (called projects when using West), and check out the commits specified.

The main project is called the manifest repository. The manifest repository and any other projects will exist in the same folder. For BioSat, this folder may be called *biosatproject*. When "west init" is run, everything inside the **biosatproject** folder becomes part of a west workspace. The structure of the resulting west workspace for BioSat, with Zephyr as a dependency (and other projects needed by Zephyr), may look as shown in Fig. 2.2:

```
biosatproject/            # west workspace
|
| biosat/                 # .git/
|    |-- CMakeLists.txt
|    |-- prj.conf
|    |-- src/
|    |    |-- main.c
|    |-- west.yml         # main manifest with optional import(s) and override(s)
|
|-- modules/
|    |-- lib/
|        |-- tinycbor/    # .git/ project from either the main manifest or some import.
|
|-- zephyr/               # .git/ project
   |-- west.yml          # The dependencies of Zephyr can be automatically imported.
```

Figure 2.2: Structure of a West workspace.

CMake

CMake is used to create the build files needed to compile the application together with the Zephyr kernel. This is done in two steps: the configuration step, where *CMakeLists.txt* build scripts are executed in order to create an internal model of the Zephyr build, and then the build script generation step, where the internal model of the Zephyr build is used to generate build scripts that are native to the host platform (Zephyr Project, 2023*a*). These scripts can be used to recompile the application after most code changes, without having to involve CMake. However, certain changes, such as adding source code files or changing some configuration values, require running CMake to successfully include the changes. Detecting when this is necessary and rerunning CMake is done automatically when initiating the build using West.

CMake supports a range of different build systems, but only Make and Ninja are officially supported by Zephyr. These are the tools that utilize the build scripts generated by CMake.

Ninja

Ninja is the default build system used when building Zephyr applications from build scripts generated by CMake. It is a small build system with a focus on speed, and is designed to operate on build scripts generated by tools such as CMake (Ninja-build, 2023), instead of the handwritten scripts that are often used with Make.

OpenOCD

OpenOCD, or Open On-Chip Debugger, is an open source project that aims to provide debugging, in-system programming and boundary-scan testing for embedded target devices (OpenOCD, 2023). It allows debugging a wide range of microcontrollers using the GDB protocol, as well as writing to several different flash devices, including the internal flash of supported microcontrollers, which is where the program code is usually stored.

MCUboot

MCUboot is a secure bootloader for 32-bit microcontrollers, that defines a common interface for the bootloader and the system flash layout on microcontrollers, as well as enabling easy software upgrades of the application running on the microcontroller (MCUboot, 2023*c*). A bootloader is a program that is responsible for starting up, or booting, another program.

Zephyr has compatibility for MCUboot built in. In order to build an application for use with MCUboot only a few configuration options have to be enabled, and the target microcontroller must have support for and have defined some specific flash partitions. These are

- *boot_partition*: where MCUboot itself will be stored
- *slot0_partition*: primary slot of image 0
- *slot1_partition*: secondary slot of image 0

To flash the application, MCUboot must be built and then flashed to the first partition, **boot_partition**, which is usually located at the start of flash. The Zephyr application must then be flashed to the address of the primary application, without erasing MCUboot from the flash (MCUboot, 2023*b*).

An updated image can be flashed to the secondary slot in order to initiate an application update. Before the swap operation is performed, an integrity check is performed on the updated image. If the integrity check succeeds, the bootloader will perform a swap algorithm that swaps the contents of the primary slot and the secondary slot of the image (MCUboot, 2023*a*).

There are several types of swaps MCUboot can perform. One of them is a test swap. In this boot mode the contents of the secondary slot (the pending update) are booted by swapping the contents of the secondary and the primary slots. The booted image can then mark itself "OK" during runtime by modifying the flash. The swap is then made permanent, and MCUboot will then boot into that updated image until a new swap is issued. If the image fails to mark itself "OK", MCUboot will at the next boot reverse the update by swapping the images back to their "original" places, and attempt to boot the old image (MCUboot, 2023*a*). This mechanism is extremely useful in systems that can only be flashed remotely, such as satellites. If a fault in the updated application for some reason makes it crash, the bootloader will automatically reboot into the previous, stable image. Since the marking of an image as "OK" is performed at runtime, a range of self tests can be executed and evaluated before deciding whether to persist the update or not.

Sysbuild

Sysbuild is a higher-level build system that can be used to combine other buildsystems together. It can be used to build one or more Zephyr applications together, flash them to a device and debug the results. This can be used to build both MCUboot and the Zephyr application at once, and then flash them to the device with the flash offset required for the Zephyr application to be booted by MCUboot (see section 2.2.2). (Zephyr Project, 2023*j*)

Zephyr SDK

The Zephyr Software Development Kit (SDK) contains toolchains for the architectures that are supported by Zephyr. It includes the compilers and cross-compilers used by Ninja or Make to actually compile the C code, as well as other tools such as OpenOCD and QEMU (Zephyr Project, 2023*m*). QEMU is an open source machine emulator and virtualizer that can be used to emulate the hardware of an entire system (QEMU, 2023) in order to e.g. run a Zephyr application on Windows.

2.2.3 Configuration

The Zephyr kernel and subsystems, as well as any application using Zephyr, can be configured at build time to adapt them for specific application and platform needs. Additionally, Zephyr supports describing which hardware is available on the target host, and setting the initial configuration of that hardware. The software and hardware configuration is performed using Kconfig and Devicetree, respectively (Zephyr Project, 2023*d*).

Kconfig

Kconfig is the system used to configure both the Zephyr application and Zephyr itself, and is also the configuration system used to configure the Linux kernel (Zephyr Project, 2023*b*). It is used to enable or disable the functionality of the Zephyr kernel as needed, add networking support, include device drivers, and more. The goal is to enable configuration of the software without having to change any source code.

Configuration options (called symbols) are created in *Kconfig* files. Symbols can be set to be enabled or disabled by default, and can optionally be linked to a configuration value. Symbols can also be automatically enabled or disabled based on the status of other symbols. The symbols can be grouped into menus and sub-menus, which can be accessed using interactive configuration interfaces such as *menuconfig* and *guiconfig*.

During the building of the application, these *Kconfig* files are read, and macros corresponding to the symbols and their values are outputted to a file called *autoconf.h*. This header file can then be used by the source code in order to enable, disable or configure source code based on the configuration options. The *prj.conf* file shown in Fig. 2.2 contains project specific configuration settings that will override the default configuration settings when building the application.

DeviceTree

Devicetree is primarily a hierarchical data structure used to describe hardware, but it is also used to specify the boot-time configuration of hardware (Zephyr Project, 2023g). It is used to provide the source code of the application information about what hardware is available, and the attributes related to that hardware (such as memory addresses).

The input files used by devicetree are devicetree source files (*.dts* files), devicetree include files (*.dtsi* files), devicetree overlay files (*.overlay* files) and devicetree bindings files (*.yaml* files) (Zephyr Project, 2023f). The source files and include files are used to describe the hardware available on a board, as well as the processor running the Zephyr application on that board. The bindings describe the contents of the other files in a way that enables the build system to generate C macros that can be used by applications and device drivers. The overlay files can be used to extend the base devicetree source files, e.g. to enable a peripheral on the board that is disabled by default.

The C macros generated from the devicetree files can be accessed through the *devicetree.h* header file. Any source code for the application, device drivers, tests or even the kernel can access this header in order to get handles that can be used to interact with the hardware (Zephyr Project, 2023g).

2.2.4 Documentation

The documentation of the Zephyr Project is written in the markup language *reStructuredText*, and then compiled into a formatted standalone website using Sphinx. The same approach will be adopted for the documentation of the software for BioSat, which is why this tool is described briefly below.

Sphinx

Sphinx is a documentation generator that translates plain-text files into various output formats, with automatic cross-references/linking, indices, and more (Sphinx, 2023). It can take plain-text source files such as *reStructuredText* files or *Markdown* files, and convert them into e.g. a cohesive, searchable webpage made of HTML files, into a PDF file, or many other formats.

The generated HTML files can be viewed locally on a computer, or hosted on a server. The building and hosting of the documentation can be automated using continuous integration and continuous delivery actions such as Github Actions, providing project-wide documentation that is automatically updated with the latest contributions.

Several community-made extensions exist that extend the capabilities of Sphinx. Some of these include support for incorporating Doxygen-generated documentation (Jones, 2022) and support for including PlantUML diagrams (Sphinx-contrib, 2023).

2.3 Docker

Docker is a platform for developing, distributing and running applications. It provides the capability to package and run an application in a loosely isolated environment called a container, which can be thought of as a sort of lightweight virtual machine. Containers contain everything needed to run the application, meaning that the host system used to run the container is insignificant to the behavior of the containerized application (Docker, Inc., 2023*b*).

2.3.1 Docker Architecture

Docker utilizes a client-server architecture. The Docker client issues commands and gets responses from the Docker daemon, which acts as the server and which does most of the work such as building, running and distributing Docker containers (Docker, Inc., 2023*b*). The client and the daemon can be on either the same system, or on different systems.

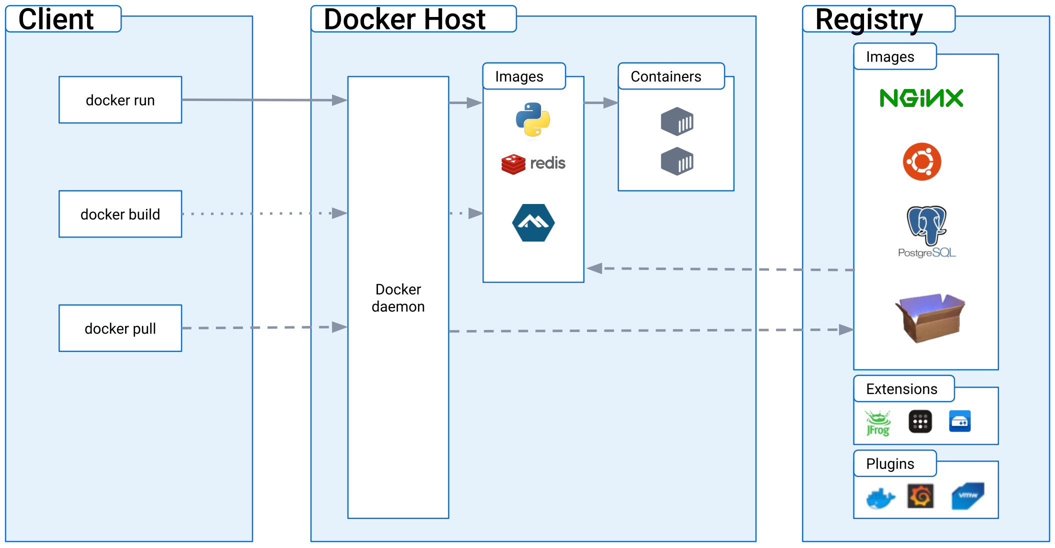

Figure 2.3: Illustration of the Docker architecture from Docker, Inc. (2023*b*).

In addition to the client and the daemon are Docker registries. Docker registries store Docker images, and is an easy method for distributing a Docker image. Distributing images can be done through the public registry Docker Hub (which is open to everyone) or through a private hosted registry (which can have restricted access).

Docker Images

A Docker image is a read-only template for creating Docker containers (Docker, Inc., 2023*b*). They are often based on other images - the Ubuntu image is a popular base for many docker containers that need a Linux-based environment.

Images are built using files called Dockerfiles. These are text files that contain a series of commands that are used to add functionality to and configure the image. A command can involve installing a piece of software the application needs in order to operate, creating or manipulating files, configuring the image's operating system and more. Each command in the Dockerfile adds a layer to the Docker image. The final image is then the result of the base image plus a stack of layers that modifies the base image as specified in the Docker file. When the image is rebuilt after a change has been made to a layer, only the affected layers will be rebuilt.

Docker Containers

A Docker container is an instance of a Docker image that can be run, and that has access to all the software, libraries, settings and code included in the image. It can run on any OS that supports Docker Engine (which contains the docker client and daemon mentioned above), is isolated from other containers and uses its own software, binaries and configurations.

The filesystem used by a Docker container is contained in the image it was created from. When a new container is created from an image, a new, writable layer is added to the existing stack of layers in the image. Any changes made during the runtime of the container are saved to this writable layer. When the container is deleted, so is the writable layer. The original layers of the image remain unchanged, and can be used to create new containers. When several containers are created using the same image, each container adds its own writable layer. The containers use a copy-on-write filesystem, meaning that the containers share the same copy of files, unless they want to make permanent changes to a shared file. Only then is the file copied to the container's filesystem, and the container can then store the changed file in its writable layer (Docker, Inc., 2023*a*). This makes Docker containers very lightweight, as they can share most of their resources. This behavior is shown in Fig. 2.4.

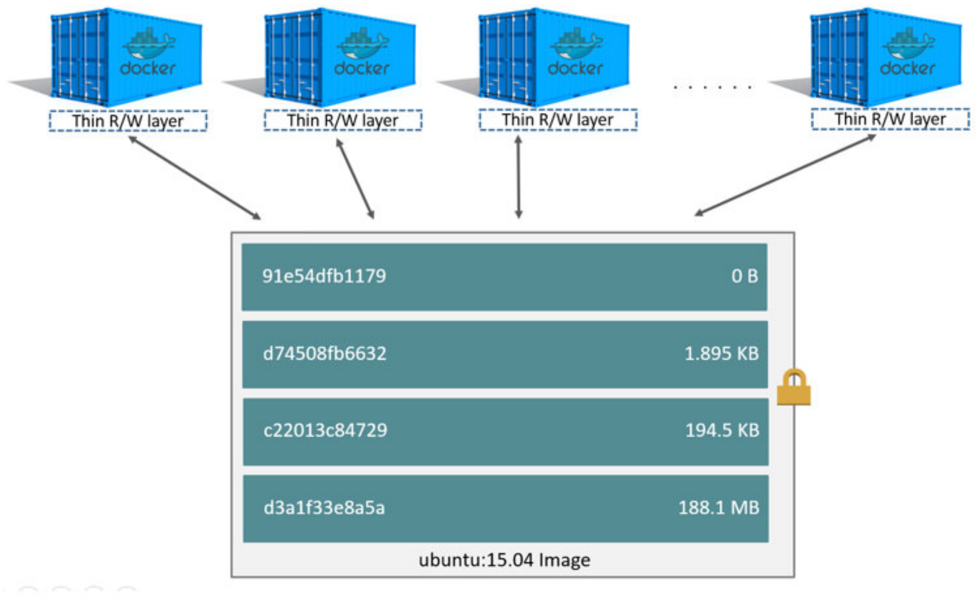

Figure 2.4: Docker containers with writable layers sharing the read-only image layers (Docker, Inc., 2023*a*).

Docker containers can be run through the command line using the Docker client. Several options can be specified when creating or running a container, such as setting environment variables, mounting parts of the local filesystem into the filesystem of the container, running the container in interactive mode which can enable us to get shell access into the running container, and more. *Docker Compose* is a tool meant for defining and running multi-container Docker applications. Another use for Docker Compose is specifying the container options in a file, and automatically applying those options and launching the container using very simple commands.

2.3.2 Dockerized Development Environment

Docker can not only be used to distribute and run finished applications; its ability to emulate a specified environment that is independent of the host system makes it an excellent platform to use for developing software. The main key to this was touched upon in the previous section: the ability to get shell access into a running Docker container. Since a container can contain all the software, libraries, tools, etc., needed to run an application, it can also contain all the tools and libraries needed to develop an application. This enables organizations to create standardized development environments containing all the necessary tools needed, while ensuring that the development environment is the same and behaves the same, independent of whatever machine the developers want to work from.

2.4 Packet Utilization Standard

The Packet Utilization Standard (PUS), officially named *ECSS-E-ST-70-41C – Telemetry and telecommand packet utilization*, is a standard maintained by the European Cooperation for Space Standardization (ECSS). It is part of a series of ECSS Standards intended to be applied for the management, engineering and product assurance in space projects and applications (ECSS, 2016*a*, p. 2).

PUS addresses the utilization of telecommand packets and telemetry packets for the purpose of remote monitoring and control of spacecraft subsystems and payloads (ECSS, 2016*a*, p. 9). The standard defines a set of standardized services that fulfill fundamental operational requirements that apply to most spacecraft. PUS is meant as a "menu of services", where the designers of a mission may select only a subset (or all) of the services in the standard, depending on which functionality the specific mission requires. The standard is adapted to the expectation that different missions require different levels of complexity and capability, and it supports systems with a centralized architecture, systems with a highly distributed architecture (several processes running on multiple boards or microcontrollers), and everything in between.

The standard lists the following criteria as being fulfilled by the standardized PUS services (ECSS, 2016*a*, p. 10):

- **Commonality**: each standard service corresponds to a group of capabilities applicable to many missions.

- **Coherence**: the capabilities provided by each standard service are closely related and their scope is unambiguously specified. Each standard service covers all the activities for managing inter-related state information and all activities that use that state information.
- **Self-containment**: each standard service has minimum and well-defined interactions with other services or on-board functions.
- **Implementation independence**: the standard services neither assume nor exclude a particular spacecraft architecture (hardware or software).

These criteria align with the common design principles for service-oriented software architectures, which are detailed in section 3.3.3. Thus, the standard defines loosely coupled, highly cohesive services, which promotes maintainability, reusability, reliability and several other software quality attributes. See section 3.2 for more background on software quality.

As part of the service definitions, PUS defines a set of service interface requirements for each standard service type. These requirements address the contents of the requests and reports a service can process or generate.

In addition to the standardized services, PUS specifies a generic service and service type model, called the PUS foundation model. This model is applicable to standard services defined by PUS, and can be applied to mission-specific services defined by the organization that is responsible for the space system's design. Using the PUS foundation model, which is applicable to all the services in the system (both standard services and mission-specific services), the architectural consistency of each service type is ensured (ECSS, 2016*a*, p. 25). This means that every service is defined and interacted with in a consistent manner, making the system easier to maintain and understand.

Lastly, PUS specifies the structure and contents of the telecommand packets used to transport the service requests and the telemetry packets used to transfer the service reports (ECSS, 2016*a*, p. 9).

2.4.1 Important PUS Terms

PUS consistently use a selection of terms in order to accurately describe the services and their requirements. This section contains a small glossary of terms that is useful to keep in mind when reading this thesis. This list has been constructed using the definitions in section 3.2 of the standard (ECSS, 2016*a*, p. 12-16).

- **Application Process**: an element of the space system that can host one or more subservice entities.
- **Application Process IDentifier (APID)**: identifier that uniquely identifies an application process.
- **Capability**: functionality of a service or subservice.
- **Data Report**: a report generated by a subservice provider as part of the subservice functionality. It can be generated either directly as a response to a request, or autonomously e.g. as part of a continuous reporting functionality.

- **Event Report**: report related to an occurrence of an event.
- **Instruction**: an elementary constituent of a request that is generated by a subservice user for execution by a subservice provider.
- **Message**: data passed to or from subservice entities. It can be either a request or a report.
- **Notification**: elementary part of a report that is generated by a subservice provider for interpretation by a subservice user.
- **On-Board Parameter**: lowest level of elementary data item on-board. Parameters have a unique interpretation. For example, the battery voltage can be an on-board parameter.
- **Report**: message made of one or more notifications generated by a subservice provider for interpretation by a subservice user.
- **Request**: message consisting of one or more instructions generated by a subservice user for execution by a subservice provider.
- **Service**: a functional element of the space system that provides a number of closely related functions that can be operated remotely. Composed of at least one subservice.
- **Service type**: a container for all the requirements related to the behavior of the service, as well as the requirements related to the interface of the service (i.e. the syntax or contents of the requests and reports of the service).
- **Subservice**: element that makes up a service. Composed of one subservice provider and any number of subservice users that interact through messages.
- **Subservice entity**: operational element of a subservice that is hosted by an application process. It can be either a subservice provider or a subservice user.
- **Subservice provider**: operational element of a subservice that is in charge of execution of the subservice requests and generation of the subservice reports.
- **Subservice user**: operational element of a subservice that is in charge of initiating the subservice requests and processing the subservice reports.
- **Transaction**: set of messages related to the execution of exactly one capability which is exchanged between a subservice user and a subservice provider. The types of transactions are request related transactions, autonomous data related transactions, or event report related transactions.
- **Verification Report**: routing, acceptance or execution verification report.

Most of the relationships between the terms and entities described in the list above will be further explained in sections 2.4.2 and 2.4.4.

2.4.2 PUS Context

A PUS service is composed of PUS subservice entities. Each subservice entity can either be a subservice provider, or a subservice user. Subservice entities are hosted by an application process either on-board the spacecraft, or on-ground (ECSS, 2016*a*, p. 18).

Usually, a subservice provider is hosted by an on-board application process, and a subservice user is hosted by an application process running on the ground. This is not enforced by the standard, however. For example, a subservice user can be hosted on-board the same spacecraft that hosts the corresponding subservice provider, or some equipment on the ground can host a subservice provider that is e.g. used by some subservice user also hosted on-ground.

No restrictions regarding how application processes and subservice entities are implemented or distributed are imposed by the standard (ECSS, 2016*a*, p. 18). There can be any number of on-board application processes, which each can host any number of subservice entities (with the restriction that only one subservice provider of a given subservice type can be hosted by a given application process). An on-board computer or device can host one or more application processes, or one application process can be distributed between multiple devices/OBCs. A given subservice provider can provide its service to any number of subservice users.

Information sent between a subservice user and a subservice provider is called a *message* (ECSS, 2016*a*, p. 19). A message sent from a subservice user to a subservice provider is called a *request*. Each request contains one or more instructions that the subservice provider shall execute. A message sent from a subservice provider to a subservice user is called a *report*. Reports contain one or more *notifications*. Reports can be grouped into three main categories:

- *Verification reports*, which report on the routing, acceptance, start, progress and completion of the request execution.
- *Data reports*, which are generated either on request from a subservice user, or autonomously as a part of a continuous reporting functionality.
- *Event reports*, which contain information related to a specific event detected by a service.

Figure 2.5 shows typical communication between the spacecraft and the ground segment, and illustrates how subservice entities are related to the corresponding subservice and service.

Requests contain a source address that uniquely identifies the subservice user that issued the request. This allows subservice providers to route verification reports and data reports back to the subservice user that requested the service.

The protocol used to transmit messages between on-board subservice entities is not specified in the standard, but the protocol used to communicate between the spacecraft and the ground segment *is*. This protocol is the CCSDS Space Packet Protocol (SPP) (CCSDS, 2020), which is used for communication between the space segment and the ground segment (ECSS, 2016*a*, p. 20). Telecommand packets and telemetry packets are encapsulated into space packets, which are then sent on the uplink or downlink, respectively.

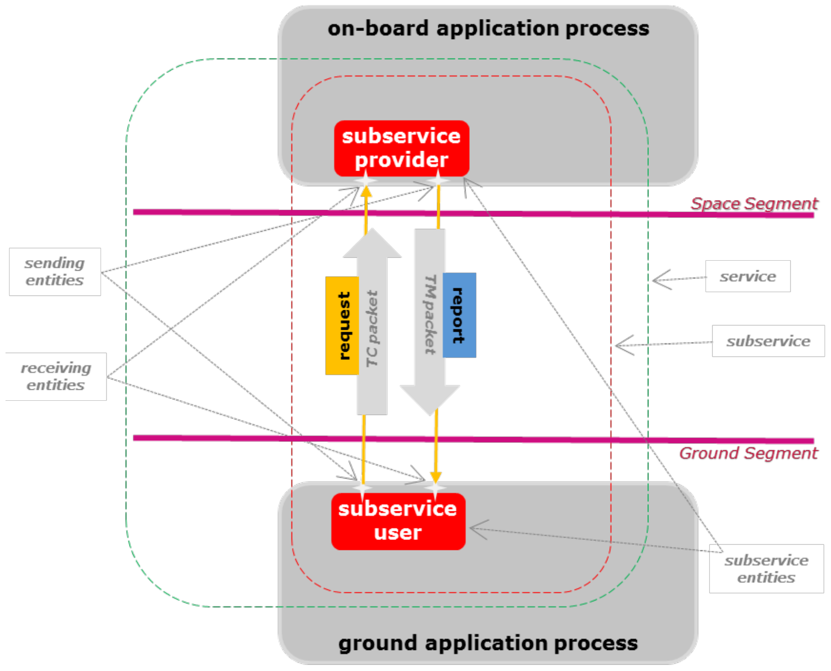

Figure 2.5: Communication between a subservice provider and a subservice user hosted on-board and on-ground, respectively (ECSS, 2016*a*, p. 19).

2.4.3 The Space System Service Model

The space system service model results from the mission-specific deployment of the service types. This can be thought of as the architecture of the space system after the standardized and mission-specific service types, and the corresponding instances (i.e. services), have been chosen (ECSS, 2016*a*, 24).

The space system model specifies each instance (i.e. service) of the chosen service types, the application processes hosting each subservice entity related to the services, and any mission-specific changes made to the standardized services (ECSS, 2016*a*, 24).

2.4.4 The PUS Foundation Model

The PUS foundation model specifies a generic service and service type model, and a set of rules that are applicable to any service type (both mission-specific and standardized) and any of their corresponding services (ECSS, 2016*a*, p. 25).

The foundation model defines generic concepts and associated requirements related to two levels of abstractions (ECSS, 2016*a*, p. 25):

- **The generic service type abstraction level**: Specifies the list of generic object types and business rules used to specify the standardized service types. For each service type, this includes:
  - The service type, the subservice types, and the capability types.
  - The subservice providers and the subservice users.
  - The message types:

        - The request types and the instruction types.
        - The response types and the notification types.
    - The transaction types:
        - For request related transactions:
            - The request type.
            - The associated execution notification type.
            - The data report type (only if service data are generated in response to the request).
        - For autonomous data reporting transactions: the related data report type.
        - For event reporting transactions: the related event report type.
- **The generic service deployment abstraction level**: The set of generic object types and business rules required to implement the service type instances (i.e. services) specified by the space system service model. For each service, this includes:
    - The system context of the service, i.e the system objects required by the service (e.g. an on-board memory object, the on-board file system, some on-board device, etc.).
    - The service, the subservices, and the capabilities provided by the subservices.
    - The messages:
        - The requests and the corresponding instruction slots and instructions.
        - The reports and the corresponding notification slots and notifications.
    - The transactions.

The exact requirements related to the various object types of the two abstraction levels are specified in detail in Chapter 5 of the PUS standard. These requirements can be followed in order to specify mission-specific service types and services that are consistent with the standardized PUS service types and services.

2.4.5 Standard Service Types

The PUS standard service types are a set of service types that provide capabilities that fulfill operational requirements which apply to most spacecrafts. A list of the standard service types can be seen in table 2.1. The service types named "(Reserved)" are deprecated service types that were used in previous versions of the standard. *ST* is short for Service Type.

Table 2.1: The PUS standard service types. *ST* is short for Service Type.

ID	**Name**
ST[1]:	Request Verification
ST[2]:	Device Access
ST[3]:	Housekeeping
ST[4]:	Parameter Statistics Reporting
ST[5]:	Event Reporting
ST[6]:	Memory Management
ST[7]:	(Reserved)
ST[8]:	Function Management
ST[9]:	Time Management
ST[10]:	(Reserved)
ST[11]:	Time-Based Scheduling
ST[12]:	On-Board Monitoring
ST[13]:	Large Packet Transfer
ST[14]:	Real-Time Forwarding Control
ST[15]:	On-Board Storage and Retrieval
ST[16]:	(Reserved)
ST[17]:	Test
ST[18]:	On-Board Control Procedure
ST[19]:	Event-Action
ST[20]:	Parameter Management
ST[21]:	Request Sequencing
ST[22]:	Positino-Based Scheduling
ST[23]:	File Management

The various types of standardized service types include (ECSS, 2016*a*, p. 22-23):

- Service types that provide basic functionality such as providing "ping" functionality, collecting parameter statistics, or generating reports containing on-board parameters.
- Service types that hold requests and release them to another service at a given time or in response to a specific event. Examples are the event-action, the time-based scheduling, the position-based scheduling and the request sequencing service types.
- Service types that provide standardized interfaces, such as to on-board devices, to on-board memory areas or objects, to on-board parameter registries, or to on-board control procedure engines (i.e. a scripting engine that can execute scripts or procedures on-board).

The specification of each of the service types consists of two parts (ECSS, 2016*a*, p. 23):

- A **system requirements specification** that defines the behavior of the service in response to requests and events. These are contained in Chapter 6 (ECSS, 2016*a*, p. 55-424) of the packet utilization standard.

- An **interface requirements specification** that defines the syntax (i.e. the structure and content) of requests to and reports from the service. These are contained in Chapter 8 (ECSS, 2016*a*, p. 445-605) of the packet utilization standard

Every standardized service type is optional to deploy (i.e. optional to include in the space system), and most of the service types are also highly configurable. For some service types, only one or a few of the related subservice types are required, and even some of the subservice types' capabilities might also be optional.

2.4.6 Mission-Specific Service Types

By following the PUS foundation model, mission-specific service types can be added in a way that is consistent with the selected standardized services. A mission might consider adding such a service in order to facilitate development, maintainability and understandability, as the format follows the one known from the standardized services. Examples of potential mission-specific service types are a camera payload service type or a logging service type.

When a mission applies the PUS standard, it should tailor the standardized service types to its needs. When this is done, a mission-specific packet utilization document should be made (ECSS, 2016*a*, p. 24). This document contains the mission-specific service-type model, which includes:

- The PUS standardized service types that will be used in the space system, as well as any mission-specific modifications or additions made to those service types.
- The selection of subservice types that are deployed of a given service type, as well as any optional capabilities of those subservice types that are enabled or disabled.
- Any additional mission-specific service types.

2.5 Cubesat Space Protocol

The Cubesat Space Protocol is a lightweight communication protocol stack designed for communication in distributed embedded systems, such as CubeSats (GomSpace, 2023*b*). The protocol stack is similar in design to the TCP/IP model, and includes several different MAC-layer interfaces (e.g. CAN bus and I2C), a routing protocol and a transport protocol.

Using CSP enables the satellite to deploy a service-oriented architecture, where each subsystem can provide its respective services to any subsystem connected to the CSP network. This has several advantages, some of which are minimizing dependencies between subsystems, promoting service reuse as they can be shared between subsystems, and mitigating the single point of failure that a centralized architecture may suffer.

The topology of a CSP network is similar to that of the TCP/IP model. It consists of several CSP nodes, each with a unique address in the address ranges 0 to 31 (CSP 1.0: 5 address bits) or 0 to 16384 (CSP 2.0: 14 address bits). Each node can expose up to 64 ports that can be individually addressed when sending or receiving messages between nodes. A typical configuration is dividing the CSP network into two segments: one for the space segment and one for the ground segment. Each subsystem can then be designated as a CSP node, in addition to e.g. the operator PC on ground. See Figure 2.6 for a visualization of such a configuration. Assigning CSP addresses 0 to 15 to the nodes of the space segment and 16 to 31 to the nodes of the ground segment allows for easy routing, as any CSP message with a destination address starting with a 1 should be routed to the ground and any CSP message with an address starting with a 0 should be routed to the satellite.

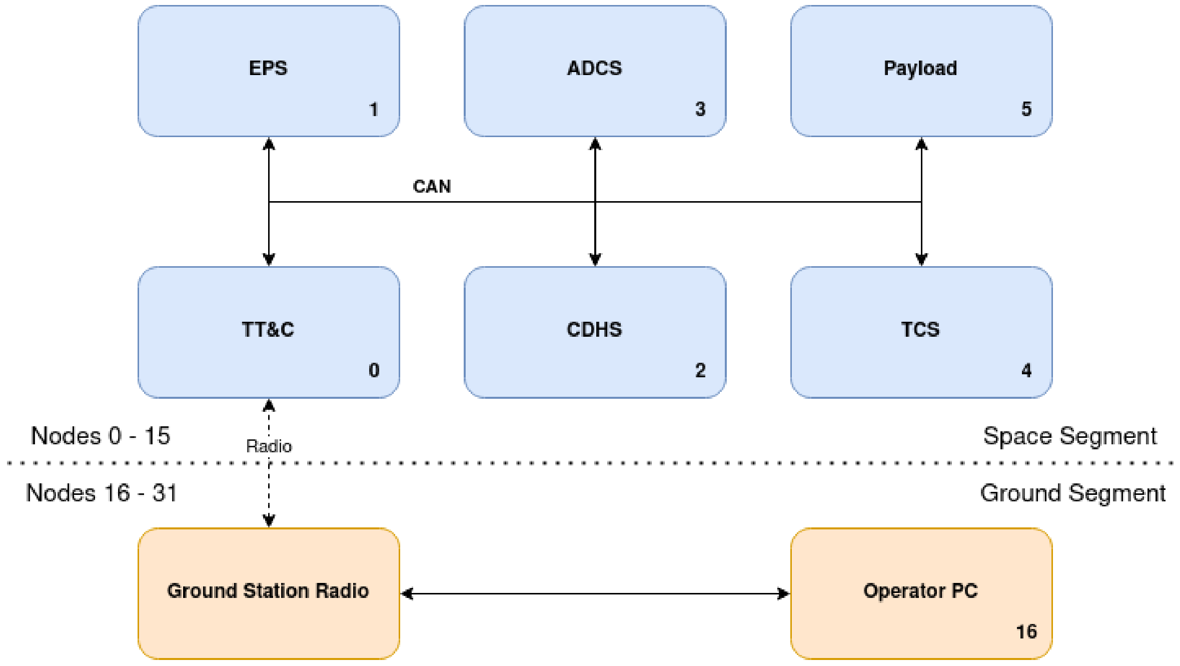

Figure 2.6: Example CSP network configuration.

CSP has been implemented in C, and has been ported to run on FreeRTOS, Linux, MacOS and Windows (GomSpace, 2023*b*). The implementation is also seemingly ported to run on Zephyr, but there is no mention of this in the documentation, and is only made evident by the source code and in the issues and pull requests on the project's GitHub page.

2.5.1 Protocol Stack

Below follows descriptions of the four layers of the protocol stack. The information has been gathered from the documentation pages stored in the project's GitHub repository (GomSpace, 2023*a*).

Layer 1: Physical Layer

CSP does not require any specific physical layer technologies. Which physical layer drivers are needed may depend on which data-link layer protocol (layer 2) is used. For example, if using CAN, a physical CAN bus driver is needed.

Additional drivers can be created if needed, e.g. if there is a need to communicate with CSP nodes connected over 1-Wire.

Layer 2: Data-Link Layer

CSP supports several MAC-layer interfaces, which can be thought of as (a part of) the data-link layer. The supported interfaces are CAN, I2C, RS232 (KISS) and Loopback.

Other MAC-layer interfaces can be added by declaring the corresponding send and receive functions. CSP can thus be extended to support communication over e.g. radio-links or SPI.

Layer 3: Network Layer

Layer 3 consists of a routing protocol. The router is responsible for accepting incoming messages and delivering them to the correct message queues, as well as forwarding messages to other nodes on the network.

Routing is performed by a routing task on each node. When a packet is routed, the destination address contained in the packet's CSP header is looked up in a routing table. Each record in the routing table contains the interface (layer 2) the packet should be sent on (e.g. using CAN or I2C), as well as any via address. For example, given the topology in Figure 2.6, a routing table entry at one of the subsystems could be ”16 CAN 0”, i.e. ”send messages with destination 16 (the operator PC) on the CAN interface, via node 0 (the radio)”. When the radio receives a packet with destination 16, it will look up the destination address in its own routing table, and find that the packet should be forwarded on the radio interface.

There is no route-discovery protocol implemented in CSP, and routing tables are configured at compile time. This means that the topology of the network should be known ahead of deployment. It is possible, however, to configure the routing tables after deployment using the CSP management protocol service (see section 2.5.2).

Layer 4: Transport Layer

CSP implements two different transport layer protocols: the Unreliable Datagram Protocol (UDP) and the Reliable Datagram Protocol (RDP). These are not equal to the UDP and TCP from the TCP/IP model, but share some similarities.

UDP provides a simple transmission service without packet re-ordering, retransmission or hand-shaking, meaning UDP provides an unreliable service. Error checking and packet re-ordering are assumed to be unimportant or handled by the application layer. If a response to a request is never received, it is up to the request issuer to send a new request. The low overhead of UDP makes it a popular choice for real-time tasks that might have the time-budget to wait for delayed packets.

RDP is an implementation of RFC908 and RFC1151, and is intended for use when e.g. transferring larger packets where manual ACKing becomes a bottleneck. A typical use is when transferring files. RDP provides packet re-ordering, retransmission, flow control, and more, which means that a file can be automatically split into suitably-sized packets, sent over the network, and reassembled at the destination. Any missing packets are re-transmitted, and any out-of-order packets are assembled in the correct order at the destination.

2.5.2 Notable Features

As CSP is specifically designed for use on resource-constrained embedded systems such as a CubeSat, it unsurprisingly has a range of features that can be beneficial to a CubeSat project.

Arguably, the most important feature is providing communication between distributed nodes (or subsystems) of the system, allowing direct communication between the ground segment and the subsystems without having to go through a centralized routing application (excluding the radio). This increases the reliability of the system, as it mitigates the single point of failure that a central master node would represent.

CSP also implements 6 basic services that are available on each node. The first 6 port IDs are reserved for these services. Thus, to reach e.g. the ping service of the EPS in Figure 2.6, one would send a message with destination 1 and destination port 1. The services and their respective ports are:

Table 2.2: The CSP services.

Port ID	**Service**
0	CSP Management Protocol: configure memory, routes, etc.
1	Ping: return an ”I’m alive” message
2	Process List: return the current process list
3	Free Memory
4	Free CSP Buffers
5	Uptime: return node uptime

Other notable features are:

- Small memory footprint.
- Needs little processing power, as buffers are Zero-Copy, meaning buffers are never copied from one buffer to another.
- Messages can be assigned a priority, which affects which message is sent first.
- Support for error checking using CRC32.
- OS abstraction with support for FreeRTOS, Linux, Windows and MacOS, and seemingly support for Zephyr RTOS.
- Support for several MAC-layer interfaces such as CAN, I2C and RS232, with possibility of extension to other links.
- Support for encryption.
- Support for authentication using Hash-Based Message Authentication Codes (HMAC).

Chapter 3

Background - Software Architecture

This chapter contains central background theory regarding the design of quality software and software architectures, much of which has been used as a solid foundation for the design decisions presented in later chapters.

3.1 Faults, Errors and Failures

It is important to properly define the language used to describe how a system may fail before the reliability of the system can be discussed. Three terms that are important to distinguish are fault, error, and failure. In (Burns and Wellings, 2009, p. 28) they are given the following definitions:

- A **failure** is a deviation from the specified behavior of the system. This is the external manifestation of some internal problem.
- An **error** is an illegal or unspecified state of the system. Errors are internal problems that result in failures.
- A **fault** is some sort of event that produces an error. Examples of faults can therefore be executing a part of the code that contains a bug, the user giving invalid input, or an event caused by unexpected disturbances from the environment (such as a bit-flip from a cosmic ray).

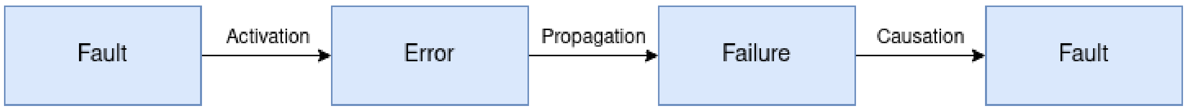

Figure 3.1: Relation between faults, errors and failures as presented in (Burns and Wellings, 2009, p. 29).

A fault is said to be *active* when it produces an error. Before that happens the fault is considered *dormant* (Burns and Wellings, 2009, p. 29). As an example, a dormant fault, such as a bug in a piece of software that will never be executed, might never produce an error and thus will never result in a system failure.

Faults are typically divided into different categories based on their permanence. This is often of high interest in real-time systems. Below follows the categorizations from (Burns and Wellings, 2009, 29):

- **Transient faults**: Faults that appear at a particular time and are only present temporarily. The fault may remain inactive for the entire duration of its presence, in which case it doesn't produce an error, or may become active at any time, producing an error that potentially results in a failure. These faults often occur due to interference from the environment and often in hardware components that are sensitive to interference.
- **Intermittent faults**: Transient faults that appears several times. Intermittent faults may appear due to hardware or software that are sensitive to conditions that keep reappearing. Examples of this are hardware that is sensitive to heat or a communication system that is sensitive to traffic.
- **Permanent faults**: Faults that start at a particular time and that are permanent until fixed. Examples are programming bugs, broken hardware or software design error.

3.2 Software Quality

The quality of software can be described using several characteristics or attributes that may be desirable in a software product. McConnel states that these attributes or characteristics can be separated into two categories: external and internal software qualities (McConnell, 2004, p. 463). External characteristics are characteristics that concern the user of the product, while the developers and designers of the software are also concerned with the internal characteristics. Below follows the characteristics which will be considered in this thesis. The definitions are quoted directly from (McConnell, 2004, p. 463-465).

External characteristics:

- **Correctness**: The degree to which a system is free from faults in its specification, design, and implementation.
- **Usability**: The ease with which a user can learn and use a system.
- **Efficiency**: Minimal use of system resources, including memory and execution time.
- **Reliability**: The ability of a system to perform its required functions under stated conditions whenever required - having a long mean time between failures.
- **Robustness**: The degree to which a system continues to function in the presence of invalid inputs or stressful environmental conditions.

Internal characteristics:

- **Maintainability**: The ease with which you can modify a software system to change or add capabilities, improve performance, or correct defects.
- **Portability**: The ease with which you can modify a system to operate in an environment different from that for which it was specifically designed.

- **Reusability**: The extent to which and the ease with which you can use parts of a system in other systems.
- **Readability**: The ease with which you can read and understand the source code of a system, especially at the detailed-statement level.
- **Testability**: The degree to which you can unit-test and system-test a system; the degree to which you can verify that the system meets its requirements.
- **Understandability**: The ease with which you can comprehend a system at both the system-organizational and detailed-statement levels. Understandability has to do with the coherence of the system at a more general level than readability does.

Below follows elaborations on a few of the characteristics.

3.2.1 Maintainability

The maintainability of a system is inversely related to the complexity of the system; the higher the complexity of a system is, the harder the system is to understand, test and maintain (Ingeno, 2018, p. 82). Thus, reducing the complexity of a system increases said system's maintainability.

Maintainability is often defined as the ease with which a software system can be maintained after the software is in operation (Ingeno, 2018, p. 79). In this thesis, the broader definition given by (McConnell, 2004, p. 463) above is utilized, in order to emphasize that the maintainability of the system is important during development as well, not only after the system has assumed operation. The reasoning behind this is that some parts of BioSat's software system will have their first iterations completed much before others. These can then be considered "deployed" pieces of software. When bugs or design errors with the deployed modules are discovered during development, resolving those issues will be a much smaller task if the software is designed with maintainability in mind. Additionally, a large part of the software will be reused in future satellite missions. This software will undoubtedly require maintenance and bug-fixing during its multi-mission lifetime.

3.2.2 Reliability, Robustness, and Resilience

The terms reliability, robustness, and resilience are often used interchangeably to describe a system's ability to continue operating to specification for a period of time, despite the presence of known or unknown disturbances. Instead, they can be defined with subtle, but important, differences, which might prove useful when discussing a system's ability to operate in varying conditions.

As shown above, McConnel distinguished between reliability and robustness. W. Jones adds a third characteristic that describes dependable performance under even more challenging conditions: resilience. Jones (2021, p. 2) definitions are as follows:

- **Reliability** is the system's ability to perform to specification under nominal or expected conditions. It is often defined as the frequency or mean time between failures.
- **Robustness** is the ability to perform satisfactorily in conditions caused by known problems or variations, but that is outside the expected nominal conditions.

- **Resilience** is the ability to continue to operate despite the appearance of unanticipated events or problems. Sometimes this necessitates operating with degraded functionality.

Methods to increase the reliability of a system can be grouped into two categories: hardware-oriented methods and software-oriented methods. Hardware-oriented methods usually involve using more reliable components or adding hardware redundancy. Reliable components, such as space-graded, radiation-hardened Commercial Off-The-Shelf (COTS) components are much more expensive than their non-space-graded counterparts. Hardware redundancy, such as triple modular redundancy (which is often used for computational logic circuits such as FPGAs (ECSS, 2016*b*, p. 141)), is great at mitigating transient faults and providing backup in case one of the components receives permanent damage. The issue with hardware redundancy is once again cost, and, maybe more important, limited volume inside the satellite. Both cost and space are highly limiting factors in a student-developed 3U CubeSat. Because of this, and to stay within the scope of this thesis, only reliability measures that can be facilitated by the software architecture and design will be considered.

Reliability methods can also be grouped into two other approaches, independent of whether they are hardware-oriented or software oriented. These approaches are *fault prevention* and *fault tolerance*. As the names might suggest, fault prevention focuses on ensuring faults are not introduced into the system during development of the system, while fault tolerance focuses on ensuring the continued operation of the system despite the presence of faults (Burns and Wellings, 2009, p. 33). They are different approaches that should both be applied when developing a reliable system.

Fault Prevention

Burns and Wellings (2009, p. 33) further splits fault prevention into two stages: *fault avoidance* and *fault removal*. Fault avoidance focuses on not introducing faults during the construction of the system. On the hardware side, this is typically selecting the most reliable hardware components. For software, this includes following strict conventions when designing and implementing the software such as following a coding style standard and using proven design methodologies. Another method is to focus on exception prevention, which may involve performing bounds checking on input arguments and taking extra care to make sure edge cases are handled (Ingeno, 2018, p. 98).

The second step of fault prevention, fault removal, focuses on finding and removing faults from the system. This process usually relies on testing the system and the various modules, attempting to check that each part of the system and the system as a whole behave according to specification. Ensuring that the system has good testability is therefore essential to be able to find faults. After faults have been found, the next step is making changes to the software in order to remove them from the system. Maintainability, therefore, plays a large role in fault prevention, as eliminating the discovered faults may be infeasible (in terms of cost and/or time) if the software is too complex or modules are too tightly coupled.

Fault Tolerance

As Burns and Wellings (2009, p. 34) points out, fault prevention has its limitations: testing only reveals the presence of faults, not their absence; testing under realistic conditions might be (practically) impossible (e.g. realistically simulating the space environment); faults may become active only after deployment of the system; hardware will inevitably fail after enough time has passed. Fault tolerance aims to mitigate the effects of those faults that passed undetected through the fault prevention efforts, and the faults that are impossible to prevent before the start of operation.

A system can provide different levels of fault tolerance. Burns and Wellings (2009, p. 34) defines the following levels of fault tolerance:

- **Full fault tolerance**: Operation of the system continues with no loss of functionality or performance, despite the presence of faults. This can only be guaranteed for a limited amount of time.
- **Graceful Degradation**: Operation of the system continues with partial loss of functionality or performance in the presence of faults.
- **Fail Safe**: The system halts operation, and enters a safe state in order to limit the extent and damage of faults.

Which level of fault tolerance is appropriate for a system depends on the functionality and application of the system. Safety-critical systems often require full fault tolerance, but this may be infeasible to achieve. Therefore, many settle for graceful degradation, especially in systems that might suffer physical damage (Burns and Wellings, 2009, p. 34). Not all parts of the system must achieve the same level of fault tolerance; the fault tolerance level of each module or subsystem depends on the criticality of the module/subsystem in mind. Which parts of the system are considered critical varies from application to application, but generally, in a satellite system, single points of failure, in which a fault can render the entire system dead, is considered critical.

3.3 Software Architecture

A system's software architecture is the *structure* of the system's software. It is concerned with higher-level abstraction and decomposing of the system's functionality into modules or components, as well as describing the interfaces and interactions between them (Ingeno, 2018, p. 10). As stated by the author of Clean Architecture, Robert C. Martin, "The goal of software architecture is to minimize the human resources required to build and maintain the required system." In other words, the purpose of the software architecture is to facilitate the development, deployment, operation, and maintenance of the software system (Martin, 2018, p. 5).

In the literature, the distinction between software architecture and software design is rather vague. In fact, C. Martin argues that there is no difference between them at all, as the high-level decisions are always made with some details in mind, and the details support all the high-level decisions (Martin, 2018, p. 4). Despite this, the more traditional definitions of the terms will be used in this thesis. That is, the software architecture concerns the higher-level structure of the software system, and the software design concerns the lower-level details and design of the various components or modules the software architecture is made up of.

3.3.1 Architectural Drivers

Ingeno (2018, p. 124) defines architectural drivers as the considerations or decisions needed to be made that are architecturally significant. They *drive* the design of the software architecture, describing what the architecture design should facilitate and why. The architectural drivers can be thought of as the specifications for a software architecture, and a concrete software architecture design attempts to realize that software architecture specification.

Architectural drivers address different concerns related to software architecture. Some are concerned with the functionality of the system, some with the reasons why a software architecture design is wanted, and others with the limitations imposed on the system. When discussing the drivers, it can be helpful to divide them into different categories, such as:

- Design objectives
- Primary functional requirements
- Quality attribute scenarios
- Constraints

Design Objectives

The design objectives are the motivations for creating a software architecture design, i.e. the reasons for why a design is wanted (Ingeno, 2018, p. 125). A common design objective is to design an architecture that facilitates the implementation of the system, i.e. makes development easier. Another design objective might be to come up with a project proposal, focusing on determining the system's capabilities, estimating the effort and resources required, and evaluating the feasibility of such a design.

As the design objectives are the reasons why a software architecture design is created, they are important drivers of the design process.

Primary Functional Requirements

The primary functional requirements describe the functionality that is critical for the system's behavior and capabilities – they are the requirements describing what the system should be able to do.

It is important to note that although the functional requirements are important to consider when designing an architecture, they almost never determine how the software architecture *must* be designed. That is, the functional requirements don't actually impose any strict restrictions on the architecture design. For example, if functional requirements were the only architectural drivers of a project, almost any architectural design would be suitable. Even "no" architecture (e.g. all the functionality of the system implemented in a single block of code) would be a satisfactory design. While the system would be able to perform whatever task it was supposed to, the performance, maintainability, and most other quality attributes would suffer to such a degree that supporting the system during its lifetime would be practically infeasible.

Therefore, the primary functional requirements should not act as the primary architectural drivers. Instead, one should focus on drivers such as quality attributes and design objectives, while still trying to facilitate the functional requirements as best as possible.

Software Quality Requirements

Software quality attributes (which are described in section 3.2) are some of the main architectural drivers. Designing for the different attributes affect the software architecture significantly, or, rather, the software architecture greatly affects which (and to what degree) software quality attributes can be met.

In the context of software architecture design, the quality attribute drivers should be expressed as measurable, testable, and specific non-functional requirements. A common format is expressing the drivers as *quality attribute scenarios*. These are short descriptions describing how the software should respond to a particular event (Ingeno, 2018, p. 126).

Expressing the scenarios in a measurable and testable way is important in order to assess to what degree a system implementation satisfies the non-functional requirements. For example, *Image compression should be fast* is a bad performance requirement, as it does not imply how the requirement should be measured and tested. Instead, a valid quality attribute scenario could be *When an image of size x is compressed, compression is completed within y seconds.*

Constraints

Ingeno (2018, p. 127) defines constraints as decisions that are imposed on a software project and that must be satisfied by the architecture. These are often set from the start of the project and can be either technical or non-technical in nature.

Technical constraints are often decisions made early on regarding the implementation of the system. Decisions such as choice of programming language, target platform or a specific technology that has to be used, are examples of this. The impact a specific constraint has on the architecture design varies greatly. For example, the constraint on which specific temperature sensor to use may hardly affect the architecture design at all, while the constraint on the operational environment of the system (e.g. having no/extremely limited internet connection in space) may have a huge impact on the design.

Non-technical constraints are, as the name implies, constraints imposed on the non-technical aspects of the project. These can for example be budget constraints or development time constraints, which act as drivers for the architecture design by imposing an upper limit on the design's complexity (the more complex a system is, the more costly/time-consuming it is to develop). Further examples of non-functional constraints are the number and the expertise of the developers that will work on the implementation.

3.3.2 Software Architecture Design

Software architecture design has the goal of satisfying functional requirements, software quality attributes, and constraints (Ingeno, 2018, p. 112), as represented by the architectural drivers. A good software architecture makes it easy to develop, modify, maintain, and extend the software of the system (Martin, 2018, p. 18).

There are two fundamental approaches to software architecture design: a top-down approach, and a bottom-up approach (Ingeno, 2018, p. 117). Which approach is most suitable depends on the size and complexity of the project, how large the team or teams that are working on the project are, and whether the domain is well understood (Ingeno, 2018, p. 121).

Top-down VS. Bottom-up

The top-down approach starts at the highest level of abstraction and then tries to break down the system into more manageable, logical pieces through a process called decomposition. As one moves down the abstraction levels, the design becomes more and more detailed (Ingeno, 2018, p. 117).

Some advantages to top-down architecture design are the ability to handle larger, more complex systems, being able to break down the system into smaller pieces that can be handled by individual teams or developers, giving an estimate of the scope of and the resources needed for the project, as well as potentially facilitate a quicker development phase. Some disadvantages are that the software architecture can become over-engineered, the design process may require a substantial amount of time if the system is complex, and some functional requirements may be missed and a good design may be difficult to achieve if the domain is not well understood (Ingeno, 2018, p. 118-119).

By contrast, the bottom-up approach starts at the lowest level. Necessary components required to satisfy the requirements of the system are identified and developed and are then gradually combined into modules and subsystems. This process continues until all requirements have been met. A pure bottom-up design does not utilize any starting architecture design. Instead, an architecture emerges as the various components and structures are created (Ingeno, 2018, p. 119).

Some advantages to bottom-up architecture design are being a good fit for teams using agile development methodologies (as bottom-up is inherently an iterative process), avoiding big upfront designs which can lead to over-engineering, and allowing development teams to start programming early. A major disadvantage is the amount of refactoring that will be necessary for larger projects. As development progresses, design flaws of the emergent architecture may become apparent, possibly requiring huge refactoring efforts to fix (Ingeno, 2018, p. 120-121).

Designing for Software Quality

Software quality attributes are some of the main concerns when designing a software architecture, as it is where architecture can have the most impact on the system. Software quality attributes will have an impact on each other, so designing with emphasis on one attribute might contribute to or inhibit another attribute (Ingeno, 2018, p. 77).

McConnell (2004, p. 78) states that "Managing complexity is the most important technical topic in software development". The complexity of the system is inversely related to several quality attributes, such as the understandability, testability, and maintainability (Ingeno, 2018, p. 82) of the system. *Decomposition* is the practice of breaking a complex system or module into smaller, more manageable pieces that are easier to understand, develop and maintain. Ingeno (2018, p. 83) lists the following general module attributes that are important when decomposing a system:

- **The size of the module**. Each module should be small in size, and not try to implement too many different capabilities. If a module is too large, it could be divided into two or more smaller modules.
- **The internal cohesion of the module**. Each module should have high internal cohesion. Cohesion is the degree to which elements inside a module belong together, or to which degree the elements work together to fulfill a single purpose (Ingeno, 2018, p. 172).
- **The external coupling**. Each module should have loose external coupling. Coupling is the degree to which a module depends on another module (Ingeno, 2018, p. 168).

Designing for attributes that don't benefit as much from decomposition might require adding functionality to existing or completely new modules or components that specifically address those attributes. Examples are adding software or hardware abstraction layers in order to increase the portability of the software. Another example is adding a monitoring module that checks that other modules are alive using e.g. watchdog timers and initiates some recovery procedure if not, in order to increase the robustness of the system.

3.3.3 Software Architecture Patterns

Software architecture patterns are common design patterns that provide high-level structure and behavior to a software system (Ingeno, 2018, p. 214). Different patterns address different concerns, and their usefulness depends on the specific problem the system tries to solve and the wanted capabilities and attributes of the system.

Layered Architecture

Layered architecture is one of the most common software architecture patterns when it comes to complicated software systems (Ingeno, 2018, p. 216). In layered architectures, the software system is divided into vertically stacked layers, where each layer depends on one or more layers located below them. A given layer is completely independent of any layers located above it.

A very general layered architecture for embedded systems can be achieved by splitting the functionality into 3 layers: a software layer, a firmware layer, and a hardware layer. The hardware layer contains the physical hardware devices that make up the embedded system (Martin, 2018, p. 262). The firmware layer contains code called hardware drivers. The hardware drivers are responsible for making the functionality of the hardware available to the rest of the software, which it does by writing to and reading from registers in the hardware, setting and reading data lines, and more. The software layer is where the behavior of the embedded system is programmed. It uses the firmware layer in order to interact with the environment and perform its intended tasks. If the embedded system utilizes an operating system, this can be placed between the firmware layer and the software layer. The software accesses the systems services through the OS. Fig. 3.2 illustrates the layered architecture of an embedded system with and without an OS.

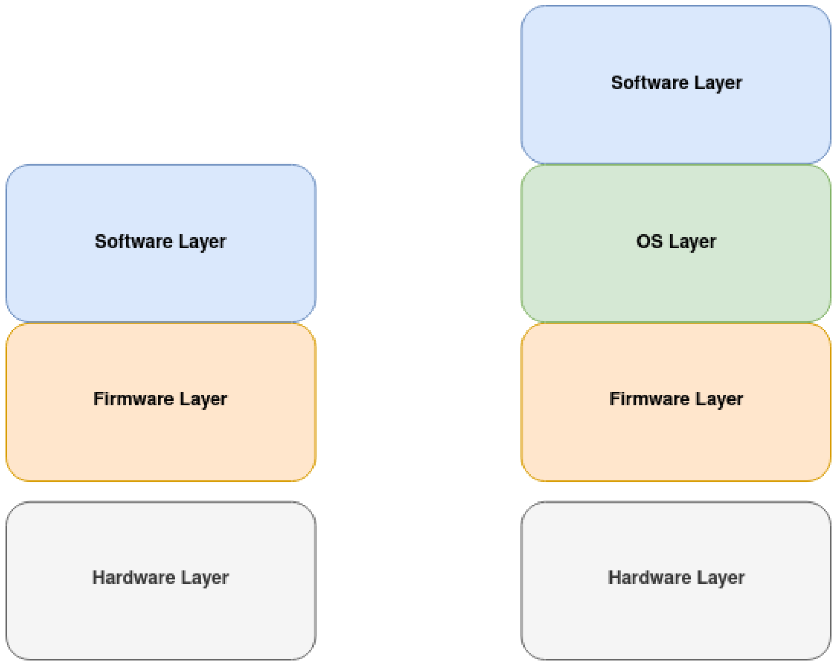

Figure 3.2: Layered architecture of an embedded system without (left) and with (right) an OS.

A common practice when working with embedded systems where portability, maintainability, and testability are of importance, is adding abstraction layers. The *Hardware Abstraction Layer (HAL)* is a layer situated right above the firmware layer. Its purpose is to hide the details of the specific hardware connected to the system (Martin, 2018, p. 264). For example, the software application probably doesn't care about which specific sensor is used to measure the temperature, only that it has the capability to do so. The HAL acts as an interface to any hardware drivers, making the software using the HAL independent of the hardware connected to the system. This may drastically increase the readability and, consequently, the maintainability of the software, as it's not cluttered with details such as register addresses or timing intervals needed to communicate with the hardware. Continuing with the temperature sensor example, using a HAL enables the software to be executed on a range of different systems, as long as that system provides a hardware driver that conforms to the HAL interface used by the software. This increases the portability of the system, as the software can be ported to a different system with a different sensor (or just a newer version of the same sensor) with only having to make a few changes or no changes at all. The system doesn't even have to have a temperature sensor: it can simulate the behavior of a temperature sensor and supply dummy data instead. Being able to run the software on simulated hardware drastically increases the system's testability, as it can be tested on a normal computer. In Fig. 3.3, the architecture of an embedded system with a HAL is shown to the left.

The same principle applies to a system that utilizes an OS. The *Operating System Abstraction Layer (OSAL)* is a layer situated right above the OS layer, and its purpose is to isolate the software layer from the OS layer (Martin, 2018, p. 270). This will enable the project to move between operating systems without having to rewrite major parts of the system's software, as well as facilitate testing of the target hardware (i.e. testing on something other than the embedded system, such as a normal computer running Linux). The architecture of a system with an OSAL is illustrated to the right in Fig. 3.3.

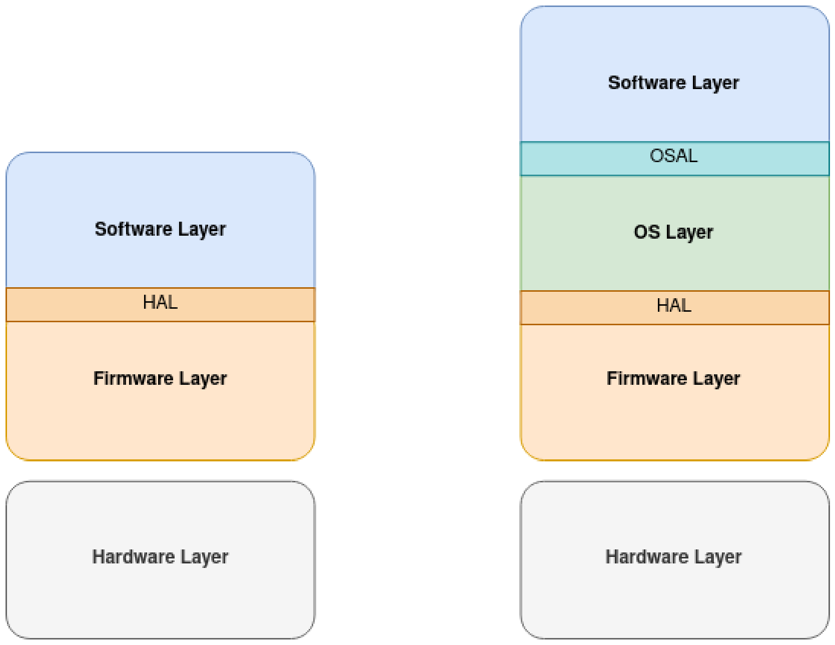

Figure 3.3: Layered architecture of an embedded system with a HAL (left) and with both a HAL and an OSAL (right).

Service-Oriented Architecture

Service-Oriented Architecture (SOA) is a software architecture pattern that revolves around creating loosely coupled, interoperable services (Ingeno, 2018, p. 241). A service is a part of an application that performs a specific task, providing functionality to other parts of the application. Services vary in size, and can also utilize other services in order to perform their task (Martin, 2018, p. 242).

SOAs are a good fit for complex systems where maintainability is of high importance. The services in a SOA should be loosely coupled and independent from each other, which will decrease the complexity of the system and each service. This also makes SOAs a good fit for organizations using agile development methodologies, as each service can be developed more or less independently and presents a more manageable sized problem to solve when developing a solution (Martin, 2018, p. 244). Independent services are easy to distribute across subsystems, making them available to any subsystem or application that might need them. This also presents an opportunity for increased robustness, as service requests to a faulty service in one subsystem can be rerouted to a service in another subsystem, as long as they provide similar functionalities.

Martin (2018, p. 246-248) presents eight design principles from Thomas Erl's book, *Service-Oriented Architecture, Second Edition* (Erl, 2017), that detail how services in a SOA should be designed. Note that not all principles are as relevant in some systems, such as self-contained embedded systems that are disconnected from the internet. These design principles are:

- **Standardized service contract**: Each service should have a standardized service contract, consisting of a technical interface and service description. This ensures that services are interacted with, and can interact with each other, in a standardized way, and that the capabilities of the services are easily understood.
- **Service loose coupling**: Services should be loosely coupled and independent from each other and service consumers. This will make services more maintainable and allow services to be modified while having minimal impact on other services.
- **Service abstraction**: Service contracts should only contain information that is necessary for the service consumer to know. Service implementations should hide details that are not essential in order to use the service.
- **Service reusability**: Services should be designed with reusability in mind, with their functionality independent from a particular technology or the specific application it is used in. This may involve extracting functionality into many separate services, which might increase the complexity of the system. Therefore, caution should be exercised not to make everything into a service. An advantage of reusable services is that an existing service has often already been tested thoroughly, which leads to higher-quality software whenever it can be reused.
- **Service autonomy**: Services should be designed to be autonomous, with increased control over their runtime environments. Dependencies on resources that the service cannot control, such as a shared resource that is used by many other services, should try to be minimized. This will increase the reliability and performance of the service.
- **Service statelessness**: Services should try to minimize the amount of state management happening. In practice, calls to a service should be as independent from previous calls to the service as possible. This will reduce the resources needed by the service to process requests, increasing the number of requests that can be handled reliably.
- **Service discoverability**: A service should be discoverable, both to humans searching manually as well as software applications searching programmatically.
- **Service composability**: Services should be designed to be composable, meaning that a service might utilize other services in order to provide its functionality.

Chapter 4

Identifying the Architectural Drivers

In this chapter, the architectural drivers of the software architecture design for BioSat are identified. The drivers are split into four categories, representing the different aspects that influence the design of the architecture.

4.1 Design Objectives

The design objectives represent the reasons for why a software architecture is designed, i.e. what is the software architecture design trying to accomplish.

Four major design objectives have been identified:

- **Facilitate the development of a software system that satisfies the functional requirements**. This objective requests a software architecture that makes it easier for the developers to implement the required functionality of the system. It also requests a software architecture that makes it easier to reason about the software system, and about whether the current design leads to an implementation that actually satisfies the requirements. This is a broad objective that has a subtle, yet large impact on the software architecture design. For example, this objective incentivizes creating a maintainable, testable, and understandable design.
- **Facilitate a reusable design where large parts of the system can be reused as a mission-independent satellite bus**. A major goal for the BioSat software is the implementation of mission-independent software that can be reused in future satellite missions of Orbit NTNU. Reusable software was not a major priority during the development of SelfieSat and FramSat (Orbit NTNU's two previous satellite projects) – it was assumed that porting the useful parts of the software from SelfieSat to FramSat would be easy. During the development of FramSat, it became apparent that it was in fact not easy, leading to many hours of development time wasted. It culminated in opting to implement a minimal viable product nearing the end of the development period, resulting in large parts of the existing software being scrapped and re-implemented from scratch. Not only does this affect the quality and capabilities of the satellite, but it is

also much less motivating for the developers who have to spend their time fixing other people's code instead of creating something new. Therefore, the software architecture design for BioSat should focus on creating a clear distinction between the mission-dependent and mission-independent parts of the software, enabling future missions to focus on building new software, experimenting with new technology, and refining a stable codebase, instead of spending time fixing and reinventing the wheel.

- **Facilitate the development of a reliable, robust, and resilient system**.
  - The space environment can be rather harsh on electronic systems, and disturbances to the system in the form of bit-flips and latch-ups due to cosmic rays or other radiation are to be expected. These disturbances can produce errors in the system, which may propagate to system failures that can in the worst case lead to the system sustaining permanent damage. Since repairing or re-programming a satellite in-orbit is extremely expensive, it is very important that the system in itself is as fault-tolerant and reliable as possible. If potentially damaging faults and errors can be resolved in time, the expected lifetime of the satellite can be extended significantly.
  - In addition to the expected disturbances or problems, the satellite can experience conditions caused by known problems that are deemed unusual or outside the expected conditions. An example of a known, but unexpected, condition is the on-board battery reaching a very low battery voltage, which, arguably, shouldn't happen if the satellite functions according to specification. Robustness allows the satellite to adapt to the abnormal (but known) situation, for example by initiating failure detection and repair procedures or by temporarily operating with reduced performance.
  - Resilience addresses continued operation in the presence of a fault caused by unknown problems. It is impossible to implement any specific measures to combat a specific unknown fault, as they are, by definition, unknown and unpredictable. Instead, general methods that can detect when faults appear, isolate them, and prevent damage to other parts of the system, are needed. This is a very challenging task to tackle, and achieving a high degree of resilience might be infeasible in a volunteer student-driven project. Nevertheless, efforts should be made to create a design that enables the implementation of some generic, low-complexity resilience measures.
- **Facilitate the development of a maintainable software system in order to support future missions and reduce set-up time for new developers**.
  - Ensuring that the first iteration of the design and implementation is bug-free and without design flaws is practically impossible, and predicting the changes or additions needed for future missions is equally infeasible. As the mission-independent parts of the software designed for BioSat are intended to be reused in an (as of now) undetermined number of future satellite missions, ensuring that the software has high maintainability is incredibly important.

  - The high turnover rate of Orbit NTNU poses a major challenge: most members only stay for one to three years and the most knowledgeable members are constantly leaving the organization as they graduate. This leads to a constant drain of knowledge and expertise, and a constant need for training of new members. To combat this, a software architecture design that focuses on maintainability is essential in order to enable new students to start contributing to the project as soon as possible, and in order to make it easier to preserve the knowledge of the graduating students. This can be extended to include requirements for software with high readability, understandability, testability, etc., as they all influence the maintainability of the system.

Note that all of the design objectives could be alternatively expressed in terms of several non-functional requirements or quality attribute scenarios. However, the design objectives are not meant as a set of strictly defined, specific requirements, but rather as a broader vision that provides context for many of the specific requirements. For example, all four design objectives described above hint towards a design where maintainability is important. The design objectives, however, don't just state that maintainability is important, they also provide a reason for *why* maintainability is important.

4.2 Primary Functional Requirements

Before and during the writing of this thesis no official requirements specification has been produced by the various teams of Orbit NTNU for the different subsystems of the satellite. This was due to efforts being centered around completing the development of the FramSat-1 satellite and due to the design of BioSat's subsystems only being in the concept design stage.

In order to not limit the other teams' design possibilities, functional requirements related to specific subsystems (other than the CDHS) are intentionally excluded. Instead, functional requirements that apply to all subsystems have been created, representing a general functional contract that should be followed by every subsystem.

Since the CDHS acts as the "brain" of the satellite and contains mostly mission-independent functionality, creating functional requirements for that subsystem was deemed necessary in order to be able to design a software architecture with value. These functional requirements were created using different sources of inspiration: the functional requirements for the previous missions (Orbit NTNU, 2023*c*), a conference paper describing the *On-board software architecture in MTG satellite* (Wenker et al., 2017), the ECSS standard *ECSS-E-ST-70-11C – Space segment operability* (ECSS, 2008) which contains requirements related to the operation of a satellite, and the capabilities described by the PUS services (ECSS, 2016*a*). Lists of these functional requirements can be found in the Appendix in section A.1.

When creating the functional requirements the following scenarios were kept in mind:

- **The operators on ground must be able to command any given subsystem independently of any other subsystem** (i.e. there is no central subsystem responsible for verification and routing). This will limit the number of single points of failure, increasing the reliability of the system. It also enables developers to test their subsystem in isolation and ensures the developers don't have to (temporarily) modify another subsystem in order to test their own subsystem. Communicating with a subsystem in orbit is obviously impossible without depending on the TT&C subsystem, as this is the link between the ground and space segments. Even though the two radios (S-Band and SDR) *does* present an opportunity for communication over redundant radios, the SDR should not be considered as a reliable backup, but rather as a payload that can *maybe* be used for that purpose if it is needed.

- **The operators must be able to receive data from the satellite**. They need data related to the satellite's health and operation, as well as data related to the mission (i.e. payload data). This data can either be formatted as telemetry packets, or as larger files. Since the satellite only is in range of a ground station for a limited amount of time, the data must be able to be stored on-board the satellite until it can be transferred to a ground station. To determine in what order data was created, the satellite must keep track of time and timestamp the data sent to the ground.

- **The operators need telecommands to be traceable**, making it clear whether a command reached its destination, whether it was accepted, and whether it started and finished execution. This also applies to any events occurring on-board, necessitating a log of events and actions that can be downloaded and analyzed.

- **The operators must be able to upgrade the software of a subsystem after launch**, as bugs and design flaws will surely be discovered while the satellite is in orbit. The updates must be verifiable and reversible, in case an update contains a breaking bug.

- **The operators must be able to monitor the satellite's health in order to detect and prevent failures**. It is also important that failures are detected and prevented when the satellite is outside ground station range, or when detected faults require swift action not achievable manually by an operator. This necessitates some form of autonomous FDIR system.

- **The operators must be able to schedule commands for execution at a later point in time**, as the satellite will not be reachable from ground during most of its orbit. Additionally, the operators should be able to upload a sequence of commands in order to save time and effort.

- **All subsystems must be able to communicate with all other subsystems**. Subsystems being able to communicate with each other is what enables the autonomous operation of the satellite and the mission. The CDHS also needs to be able to communicate with the other subsystems in order to monitor the satellite's health and perform recovery procedures in case of failures. Direct communication between all the subsystems will also reduce the bandwidth usage of the on-board network and will also require less processing, as e.g. a satellite pointing request to the ADCS from an imaging payload won't have to be passed to a central subsystem, only to then be rerouted to the ADCS.

4.3 Software Quality Requirements

Software quality requirements are non-functional requirements that address how well the system should be able to do something or the quality of the services it delivers. They also address how easy it is for maintainers to develop, maintain and reuse the resulting software. A list of non-functional requirements can be found in section A.2 in the appendix.

Some of the software quality requirements have been derived from the design objectives, some are related to the harsh space environment, and some are related to the mission objectives. Below follows a summary of the different non-functional requirements and the reasons why they are important for the BioSat architecture design. Since most of the constraining values such as available RAM, the power budget, timing requirements, etc. are as of the time of writing still undecided, the requirements do not specify any concrete thresholds. These requirements should therefore be reworked at a later date with the chosen constraints in mind.

4.3.1 Maintainability Requirements

- **The Mean Time To Recovery** (MTTR) is a common requirement related to maintainability. It specifies how long a recovery of the system's functionality after a failure occurs (i.e. fixing the issue and deploying the fix) is expected to take. Choosing a maximum value of MTTR can be very difficult, however, especially for an organization such as Orbit NTNU where the expertise and availability of the members varies greatly.
- **Requirements that apply to the implementation of the software**, such as requirements related to writing documentation, using version control, and following a coding style standard, all affect the maintainability of the software developed. These requirements don't actually affect the architecture design though, as they can be applied to the implementation of any design.
- **Generic requirements that address common design principles used when creating maintainable software** are not as measurable, but may still affect the design in a positive way. For example, requirements regarding the degree of cohesion and coupling of each software module or the interfaces between modules will promote maintainability.

4.3.2 Reliability Requirements

- **The Mean Time To Failure (MTTF)** specifies the expected time a system can operate before a failure occurs. This is a general reliability requirement that doesn't impose or encourage any specific design decisions, but it emphasizes the need for a reliable system.
- **Requests shall be executed in a timely manner**. Failing to execute real-time requests within their deadline may result in degraded performance (e.g. pointing accuracy in the case of ADCS), and may even cause permanent damage to the satellite or mission (e.g. not regulating the temperature of the plant payload in BioSat in time).

- **Execution of higher priority requests should not be significantly delayed by the execution of lower priority requests**. It is important that the resources of the system are not consumed by lower-priority requests, as this can lead to higher-priority and potentially critical requests missing their deadlines. This means that there should exist a mechanism for pausing or preempting lower-priority requests when a higher-priority request has been issued.
- **No single request executed at the wrong time or in the wrong configuration should result in the loss of the mission**. This requirement attempts to reduce the single points of failure in the system, by requiring potential critical requests to be executed in more than one step. This will reduce the chance of critical requests being wrongly issued as the result of a bug, or accidentally issued at the wrong time by the operators.
- **The bootloader shall be protected from accidental writes**. If the area of memory that contains the bootloader is not protected, an accidental write to that area could in the worst case result in the subsystem failing to start up after a reboot. This could, unsurprisingly, be catastrophic for the mission.

4.3.3 Testability Requirements

- **Subsystems and individual modules should be testable independently from other subsystems or modules**. The goal is to allow developers to work on several modules/subsystems independently and concurrently, and to make it easier to isolate functionality for testing. Testers will have a clearer view of where the error may originate from, as the number of dependencies is limited.

4.3.4 Reusability Requirements

- **A subsystem's software should be clearly separated into mission-specific and mission-independent modules**. If a module that can be useful in other missions (such as an image compression module) is implemented independently from a mission-specific module (such as a plant monitoring module), reusing the mission-independent module (in this case the image compression module) could in many situations be as easy as copy-pasting the existing code. It is then important to make sure that the implementation of mission-independent modules is not cluttered with details about specific use cases. For example, an image compression module should not know, nor care that the image being compressed originates from the plant monitoring module.

4.3.5 Portability Requirements

- **The software should be able to be compiled and run on different CPU architectures**. This will enable developers to test the software of the target hardware. Testing off the target hardware can be more convenient (e.g. if they can test on their own PC) and will reduce the number of development boards needed, reducing costs.

- **The software should be able to run on different hardware configurations**. Most software modules do not need to know which specific device or sensor is used to perform some action or to get a measurement, it just needs to know that *some* device or sensor is available and can provide that functionality. For example, the temperature control module in the TCS should not care whether a BME280 is used or a DHT22 is used to measure the temperature, only that it can get *a* temperature measurement (and optionally that the accuracy, resolution, and measurement time is within the given thresholds).
- **The software should be able to be compiled for and run on different operating systems, with degraded performance if required**. Since subsystems might utilize services typically provided by an OS (concurrency, file management, etc.), writing software that is not dependent on a specific OS implementation opens up the possibility of running the software on different OSes. This is useful both for testing, as one can run the software natively on the same computer used for development, and for reusability in future missions, as one might wish to change the OS used on the satellite. Since not all OSes provide the same capabilities, running the software on a different OS than the primary/target OS might require degradation of the performance or functionality of the system. For example, modules designed to be run on a real-time operating system might not be able to meet their real-time deadlines when run on a desktop Linux distribution.

4.3.6 Efficiency Requirements

- **Dynamic memory allocation shall not be used**. This requirement prevents problems such as heap fragmentation and memory leaks, which are common and often critical problems in embedded systems. The requirement might not actually affect the software architecture design much but has a more pronounced impact on the detailed design of individual modules and their implementation. It also makes it possible/easier to reason about the memory usage at compile time.
- **The subsystem should support a low-power mode**. The power budget of a satellite is usually very limited. Thus, each subsystem should strive towards using as little power as possible, as this leaves room for power variations or may enable adding more features to the system.

4.4 Constraints

The constraints are decisions that have already been made, and that must be respected by the design.

4.4.1 Technical Constraints

- **The Zephyr RTOS shall be used as the operating system of the CDHS**. The software architecture design must be able to be implemented in Zephyr without negatively affecting other functional drivers in a significant way. This constraint exists because of a trade-off study that was performed in order to determine which real-time operating system would be best suited for the mission and Orbit NTNU's learning goals, where Zephyr came out as the winner (see section 2.1.3).
- **The CDHS shall be implemented in the C programming language**. Every previous project of Orbit NTNU has used C as the programming language for (most of, if not all) the embedded software. As no major benefit of switching to another language was found, together with the familiarity many of the members have with the language, the choice was made to keep using C as the main language for the embedded software.

4.4.2 Non-Technical Constraints

- **The satellite shall be ready for launch within 2025**. This is the current launch date at the time of writing. It limits the available development time, and, consequently, how complex the system can be.

Chapter 5

The BioSat Software Architecture

In this chapter, a software architecture design for BioSat is suggested. The design is based on and tries to satisfy the architectural drivers identified in Chapter 4. First, a layered, service-oriented architectural pattern is chosen. Then, the design's communication strategy and failure detection and handling strategy are presented. Lastly, a brief introduction to the various components of the software layers and their functionality is given. A more detailed design for many of the components can be found in Chapter 6.

5.1 Choice of Architecture Pattern

A combination of a layered architecture and a service-oriented architecture was deemed the most suitable software architecture pattern for BioSat and possible future satellites developed by Orbit NTNU. This choice was made based on the architectural drivers identified in Chapter 4, most notably the design objectives in section 4.1.

The layered architecture pattern was chosen because it contributed to the objective of creating reusable and portable software and to the objective of creating maintainable software.

The service-oriented architecture pattern was also chosen to promote reusability, both internally in the software system and across satellite projects. It also provides options for increasing the reliability of the system, as redundant services can be distributed between the subsystems. Lastly, a service-oriented architecture makes the utilization of the PUS services easier and more "idiomatic."

The resulting architecture from the combination of a layered architecture and a service-oriented architecture is illustrated in Figure 5.1. It consists of 3 main software layers: the application layer, the service layer, and the low-level software layer. The layers are further refined into services and modules in the next section.

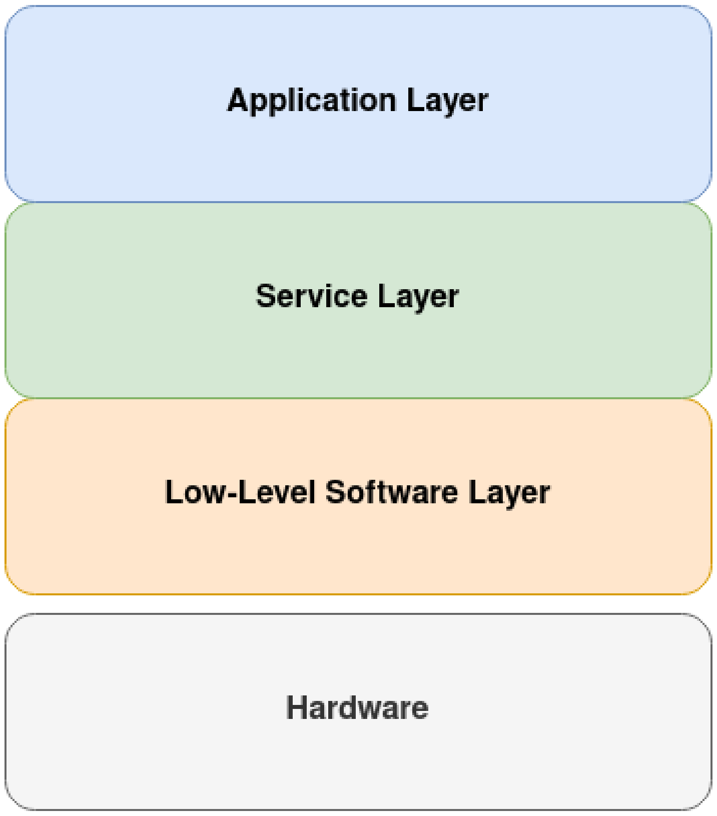

Figure 5.1: The chosen software architecture pattern.

5.2 Architecture Design

This architecture design mainly concerns the CDHS. The other subsystems might follow a similar architectural pattern as the one used for the CDHS but are technically not required to. Which architectural pattern is the best fit for each subsystem will become clearer as the specifications for each subsystem are chosen.

That being said, the software architecture design presented here expects every subsystem to have a clearly defined interface, as specified in the non-functional requirement BIOSAT-SUB-MAI-002 (see A.2.1). This interface consists of one or more service interfaces available to the other subsystems and to the ground segment. How a subsystem chooses to handle service requests is up to the designers of that subsystem – the subsystem might employ a service-oriented architecture such as the CDHS, or it might consist of a single super loop with a switch case for the different requests. The important thing is that the subsystems follow the functional requirements specified in A.1.1.

Figure 5.2 displays a more detailed illustration of the on-board software architecture, where each service and module is fitted into the three layers depicted in Figure 5.1. The functionality of the services and modules are briefly described in the following sections, with a more detailed design suggested in Chapter 6.

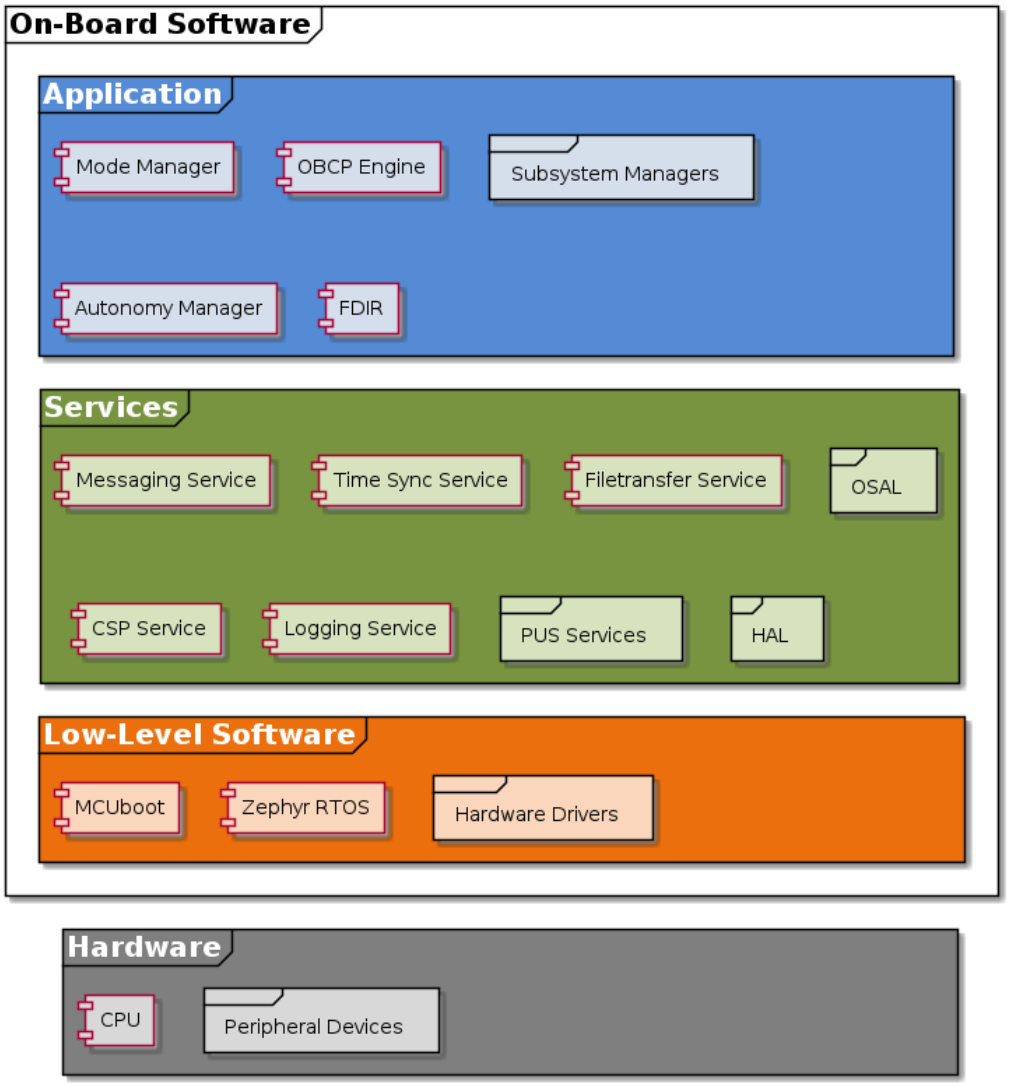

Figure 5.2: Software architecture of BioSat's on-board software.

Note: the software of the subsystems other than CDHS are represented as placeholder application layer modules called "[subsystem] manager." These will be split into their respective services and application layer modules as the subsystems are designed. Every service and module in Figure 5.2, except the subsystem managers, are part of the CDHS.

5.2.1 Communication

The communication in the system (satellite + ground) can be described in two levels: the communication between application processes and the communication between software modules. An *application process* is defined in section 2.4.1 but is in practice synonymous with the software for a subsystem or some application on the ground segment. A *software module* is an element of an application process that can issue or execute a request. Subservice providers and subservice users are software modules, meaning that both the services in the service layer and the application modules in the application layer are software modules. In practice, a software module is usually a task or thread in the application process.

Communication Between Application Processes

In order to fulfill the requirement of every subsystem being directly operable by the operators on ground, a highly distributed topology was chosen for the subsystems. Opting for a distributed topology will increase the robustness of the system in the sense that each subsystem can operate (more or less) independently from each other, allowing continued (but reduced) operation even after a permanent subsystem failure. It will also be beneficial for the maintainability and testability of the system, as the subsystems can, to a larger degree, be developed and tested independently.

CSP will be used as the protocol for communication between the subsystems and for communication between the ground segment and the subsystems. CSP was chosen partly because it was used on the two previous satellite projects of Orbit NTNU, but also due to it being designed for distributed embedded systems and due to it providing useful features such as low resource usage, error checking using CRC32, and HMAC-based authentication. A more extensive list of notable features of CSP can be found in section 2.5.2.

The CSP node topology will be very similar to the topology shown in Figure 2.6. The only difference is the payload node being split into the two payloads of BioSat (the BioBox and the SDR) and the addition of an RS424 link to speed up the transmission of larger mission data products, such as images between the CDHS and the TT&C subsystem. As a consequence of choosing this topology, there is no fundamental difference between communication between the subsystems and communication between a subsystem and the ground segment. The CSP topology for BioSat can be seen in Figure 5.3.

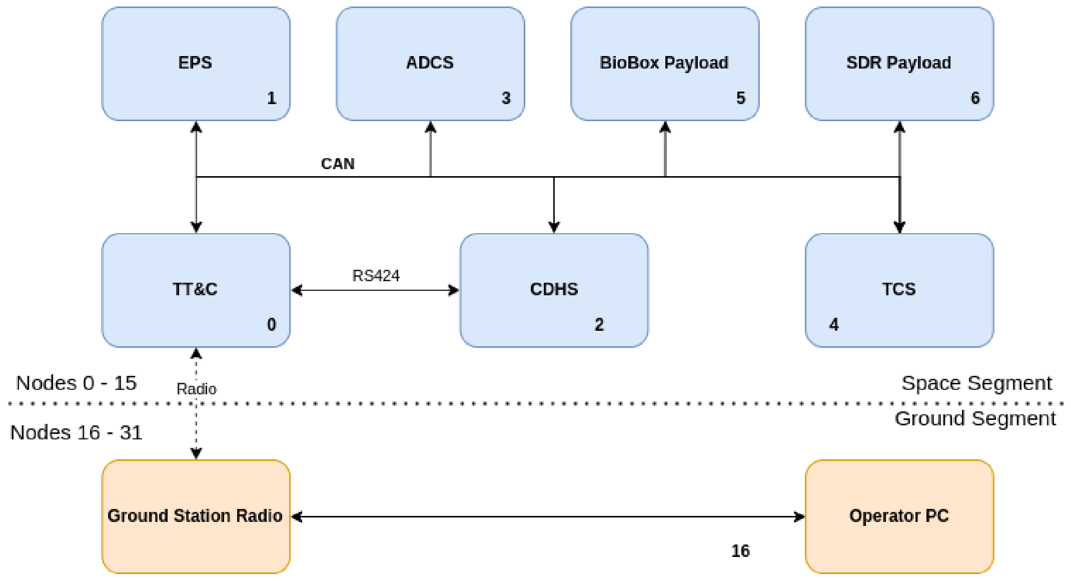

Figure 5.3: Example CSP network configuration.

As mentioned in section 2.5.1, every CSP node is expected to have its own CSP task that contains a routing layer. With the configuration shown in Figure 5.3, this is a hard requirement for the TT&C as it is the node that is responsible for routing packets between the radio link and the on-board CAN link. The CSP task is also the interface between the CSP network and the software modules of an application process.

Communication Between Software Modules

Communication between software modules follows the PUS convention (see sections 2.4.1 and 2.4.2):

- Communication is achieved by sending requests and reports, collectively called messages.
- Messages can be sent between modules of the same application process or between modules of different application processes.
- Messages contain a source address and a destination address that is used for routing.
  - The destination of a request is the module responsible for the request's execution, and the source of a request is the module that issued the request.
  - The destination of a report is the module responsible for processing the report, and the source of a report is the module that generated the report.
- An address is a unique identifier consisting of the APID and the module identifier.
- Requests contain a request identifier that uniquely identifies the request type.

Figure 5.4 illustrates the steps from issuing a request to the start of execution of that request at the destination module. During the execution of a request, execution progress reports may be generated. Once it finishes execution or fails to finish, it may generate successful or failed completion of execution reports. The generation of these reports can be enabled by setting corresponding flags in the request header.

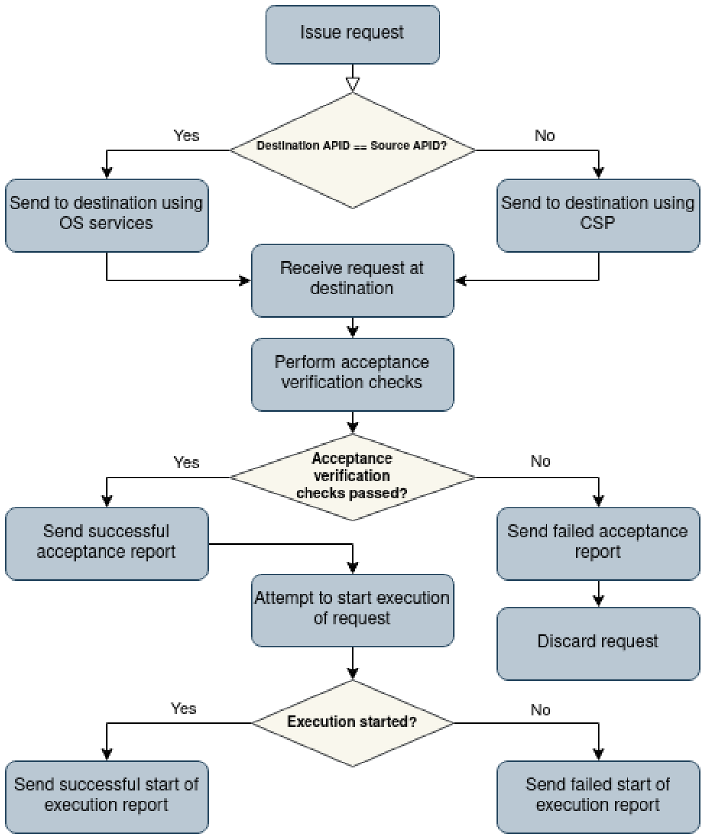
Issue request
Destination APID == Source APID?
Yes
No
Send to destination using OS services
Send to destination using CSP
Receive request at destination
Perform acceptance verification checks
Acceptance verification checks passed?
Yes
No
Send successful acceptance report
Send failed acceptance report
Attempt to start execution of request
Discard request
Execution started?
Yes
No
Send successful start of execution report
Send failed start of execution report

Figure 5.4: From request issuing to execution.

During or after request execution, the module executing the request may generate data reports related to the request that are (usually) sent back to the source of the request.

The PUS (ECSS, 2016*a*, p. 425) defines a **telecommand packet** as ”the data unit that is used to carry a service request from an application process on the ground to an application process on-board” and a **telemetry packet** as ”the data unit that is used to carry a service report from an application process on board to an application process on the ground”. Due to the nature of the ground-to-space communication in this architecture design, there is no difference between telecommand packets and requests and between telemetry packets and reports, except whatever is added by the radio communication protocol.

5.2.2 Failure Detection, Isolation and Recovery – FDIR

Failure detection, isolation and recovery are, as the name implies, a series of mechanisms and strategies used to detect failures and report them to the relevant module, to isolate failures in order to avoid the propagation of the failure and prevent damage to the system, and to recover from failures (i.e., restoring functionality). The FDIR functions are implemented in a hierarchical manner in order to handle failures at the lowest implementation level possible.

Lower-level FDIR functions attempt to detect and resolve faults at the lowest implementation level. Thus, these FDIR functions will be applied in the scope of a single function. Some examples that will be incorporated into the BioSat software are:

- Bounds checking of function arguments.
- Checking and handling of return values.
- Utilizing request timeouts.
- Propagation of error codes.

Higher-level FDIR functions are more intricate and complex, and address the failures that couldn’t be detected or resolved at the lower level. These failures can be related to the functionality of a single module, they might be related to the functionality and communication of two or more modules working together, they might be related to the functionality of a subsystem as a whole, or they might be related to the functionality and communication of two or more subsystems working together (system-level functions).

Some higher-level FDIR functions that will be implemented for BioSat are:

- Retry strategies: repeating requests for a certain amount of times. Often used in communication, e.g. when no acceptance report is received after issuing a request.
- Error-detection and error-correction codes: information is added to some data (the data is usually a message or some data stored in memory) in a way that enables detecting if the data has become corrupted (e.g. during transmission or due to bit-flips). Data can also be encoded to allow correction of errors. An example is the error-detecting code CRC32, which CSP has support for.
- Ping requests: a request is sent to a subsystem or module, with the expectation of receiving a response signaling that the module is alive. If no response is received, this can be an indication that the module (or the communication bus) is bugged.

- Watchdog timers: a timer that generates an error report or that initiates an error recovery procedure if it is not reset (kicked) in time. Since the error recovery procedure often involves restarting the module, a watchdog timer is a great way to recover from transient faults.
- On-board parameter monitoring: important on-board parameters such as battery voltage or CPU temperature is periodically monitored, checking that the values lie within their nominal range.
- Firmware rollback: If an application process fails up properly after a firmware upgrade, or if the updated firmware fails self-tests, the firmware is rolled back to the previous stable version.
- Self-tests: A series of tests a subsystem executes that checks whether the subsystem's modules and hardware behave as expected. Self-tests are usually performed after a subsystem boots up, but can also be requested by some other module (such as an FDIR module).

The exact composition of FDIR functions that will be used in BioSat is, at the time of writing, not yet decided. Most of the functions described above will probably be implemented in some form, but to what extent and in which exact areas/modules they will be applied is not detailed in this design.

Every detected fault is logged for analysis purposes. The entry should contain info about where the fault was detected, which functionality failed or was in an erroneous state, and the corresponding timestamp.

If failures fail to be handled at any of the lower levels, they are propagated to the system-level FDIR manager. The exact functions of this manager are not yet specified and will be partly mission-specific. This manager is added in order to make it easier to add FDIR functions at a later point in time when the design of the satellite has matured.

5.2.3 The Low-Level Software Layer

The low-level software layer contains the software that is, to some degree, dependent on the hardware specifics. This layer includes:

- The Real-Time Operating System.
- The hardware drivers.
- The bootloader.

The hardware drivers are software (often called firmware) that provide access to peripheral hardware devices such as sensors, communication interfaces, actuators, etc. Each specific hardware device will usually require a corresponding hardware driver.

The RTOS is also somewhat dependent on the hardware, as it requires some hardware drivers and needs to be ported to run on the OBC's CPU architecture. Most operating systems created for use in embedded applications already have support for the most common CPU architectures. When the software system is run on a non-real-time OS such as a Linux distribution, for example during testing, the real-time capabilities of the OS are lost.

The bootloader is a small program responsible for starting up (or booting) the application. It is usually located at the start of the flash memory and is the first thing that is run when the OBC receives power. The bootloader is what enables updating the application software after launch. See section 2.2.2 for an introduction to how the bootloader MCUboot works. This is the bootloader that will be used on the OBC on BioSat.

5.2.4 The Service Layer

The service layer contains several loosely coupled services that provide different capabilities to the application modules in the layer above, to other services, or to the operators on ground. This layer contains mostly mission-independent services, meaning most of the services can be reused in future missions as part of the mission-independent satellite bus. It should be noted, however, that many of the services will need a mission-specific configuration.

The services mentioned in this chapter are all hosted by the CDHS. Other subsystems will have to define their own set of services according to the subsystems' needs. The CDHS service layer consists of:

- The Messaging Service.
- The Time Synchronization Service.
- The Logging Service.
- The File Transfer Service.
- The CSP Service.
- A selection of PUS services.
- The Hardware Abstraction Layer.
- Operating System Abstraction Layer.

The Messaging Service is what enables messaging between modules, both internally and externally (including between the CDHS and the ground segment). It provides a priority-based message passing system, where each user (a task/thread) has its own message queues. The Messaging Service acts as a "communication abstraction layer", in the sense that the services or modules using it are not aware of *how* the messages are passed between them. This abstraction layer ensures the modules are independent of the chosen communication method. The modules using the Messaging Service are therefore unaffected by changes in the message passing mechanisms, and even enable adding communication over completely new interfaces such as SPI or I2C without affecting the users of the Messaging Service.

When the Messaging Service receives a message with destination inside the CDHS, it routes the message to the correct message queue using services provided by the OS (e.g. message passing). When it receives a message with destination outside the CDHS, it routes it to the corresponding service or HAL interface, such as the CSP Service or the UART controller. The Messaging Service is also responsible for performing acceptance verification and generating acceptance verification reports before passing a request to a module in the CDHS.

The Time Synchronization Service is responsible for keeping the on-board time reference, and for distributing that time to the other subsystems. The operators can send a synchronization command together with the new time reference in order to synchronize the satellite with the ground segment. The service has the ability to periodically distribute the on-board time reference to other subsystems. It also provides the capability to get the on-board time reference by request, allowing subsystems with inaccurate or drifting clocks to resync when needed.

The Logging Service provides the capability to control and get information on the subsystem's logging settings. The actual logging functionality is provided to the various services and modules through a logging API (see section 6.2.2.

The File Transfer Service is a client/server that provides the capability to send files to the ground segment or other subsystems and to receive files from the ground segment or other subsystems. The file transfer protocol implemented by this service is yet to be decided.

The CSP Service provides the capabilities to send and receive messages to and from other CSP nodes (i.e. the subsystems and the ground segment) on the CSP network. This service will also provide some of the basic CSP services in Table 2.2.

Table 5.1: The selected PUS services and their implementation priority. *ST* is an abbreviation for Service Type.

ID	**Name**	**Priority**
ST[3]:	Housekeeping	Essential
ST[6]:	Memory Management	Essential
ST[11]:	Time-Based Scheduling	Essential
ST[15]:	On-Board Storage and Retrieval	Essential
ST[20]:	Parameter Management	Essential
ST[1]:	Request Verification	Important
ST[5]:	Event Reporting	Important
ST[12]:	On-Board Monitoring	Important
ST[13]:	Large Packet Transfer	Important
ST[18]:	On-Board Control Procedure	Important
ST[19]:	Event-Action	Important
ST[21]:	Request Sequencing	Important
ST[23]:	File Management	Important
ST[2]:	Device Access	Nice to have
ST[4]:	Parameter Statistics Reporting	Nice to have
ST[17]:	Test	Nice to have

The PUS services provide fundamental capabilities related to the operation of the satellite. Some of these capabilities include event generation, file management, peripheral device access, on-board parameter monitoring, on-board control procedures (scripting capabilities), and more. The PUS services that have been selected for the BioSat CDHS architecture and their capabilities are described in greater detail in section 6.2.8. The service definitions can be found in the Packet Utilization Standard (ECSS, 2016*a*). A list of the selected services can be seen in Table 5.1. They have been given a priority rating based on how important their capabilities are for the successful operation of the satellite. The reasoning for the chosen priority ratings can be found in section 6.2.8.

The HAL acts as an interface to the hardware driver software. This enables writing services and application modules that can easily be ported to a different set of hardware without requiring much (or any) refactoring. It also enables the use of hardware drivers that simulate hardware that may not be available on the host machine, e.g. when testing the software on a regular computer during development. The services that make up the HAL are all given the name “controller” (e.g. ”UART Controller”).

The OSAL acts as an interface to the RTOS. Similar to the HAL, the OSAL enables services and application modules to be independent of the specific OS. Since some services or application modules might require the real-time capabilities of an RTOS, the OSAL might not be able to offer a full abstraction of the required features, i.e. when porting the software to a non-real-time OS.

5.2.5 The Application Layer

The application layer contains application modules that are mostly mission-specific. They perform high-level tasks related to the operation of the satellite and the mission. The application layer consists of:

- The Mode Manager.
- The Autonomy Manager.
- The System-level FDIR manager.
- The On-Board Control Procedure (OBCP) engine.
- Subsystem managers.

The Mode Manager is responsible for controlling the satellite’s operational mode. The operational mode is controlled by commands from the ground station, but can also be changed in reaction to on-board events such as reaching a critically low battery voltage level. The operational mode of the satellite determines which of the satellite’s capabilities are enabled and which are disabled.

The Autonomy Manager is responsible for autonomously operating the satellite, enabling the satellite to continue mission operations without relying on requests from the ground segment. The degree of on-board autonomy can be controlled by setting the autonomy level of the satellite.

The System-Level FDIR manager is responsible for detecting, handling, and responding to any failures that are not taken care of in the lower FDIR layers. Its exact functions are still yet to be decided, but it may for example periodically perform tests that try to detect failures related to the operation of the system (e.g. by initiating self-tests in the different subsystems).

The OBCP engine is a scripting engine that is responsible for executing on-board control procedures. OBCPs are script-like programs that are written in a specific language, and that can be generated on ground and uploaded to the satellite for on-board execution. The purpose of OBCPs is the automation of complex operations procedures that can not be implemented by sequencing a series of normal telecommands. The OBCP engine and the execution of OBCPs can be managed by the PUS ST[18] On-Board Control Procedure service.

The subsystem managers are placeholder modules that represent the software of the other subsystems.

Chapter 6

Detailed Design

This chapter contains a more detailed design of the various services and modules described in Chapter 5.

6.1 Application Layer

The modules of the application layer are the most mission-specific of all the CDHS modules. The overall structure of or interface to each module can to some degree be reused in future missions, but the conditions and events these modules react to, and the specific reactions, are likely mission-specific.

6.1.1 Mode Manager

The Mode Manager is responsible for managing the satellite's mode of operation. These modes are yet to be defined, but the general concept is that the satellite's enabled capabilities are dictated by the mode of operation. A common definition of operational modes is related to the battery charge percentage: the lower the battery charge, the more limited the satellite capabilities.

The Mode Manager initiates transitions between operational modes in response to requests from the operators or the FDIR Manager, or in response to event reports. When a mode transition occurs, the Mode Manager is responsible for configuring the various hardware devices and subsystems according to the mode specification. For example, during a transition to a critical battery mode, the Mode Manager could turn any unnecessary devices off and configure MCUs to operate in a low-power mode.

6.1.2 Autonomy Manager

The Autonomy Manager is responsible for the autonomous operation of the satellite and the mission.

The Autonomy Manager's capabilities depend on whatever autonomous behavior the specific mission might need. For example, it could be responsible for initiating image compression whenever an image is transferred to the OBC from a payload. In this design, no specific autonomous capabilities are defined, as exactly what operations are suited for autonomous execution in the BioSat mission is still unclear.

The Autonomy Manager's capabilities are restricted by the autonomy level. The autonomy level can be configured through requests issued from the ground station.

6.1.3 System-Level FDIR Manager

As mentioned in section 5.2.5 the exact functions of the System-Level FDIR Manager are yet to be decided. It should handle failures that could not be handled at lower implementation levels, and generate reports related to any detected failures and any actions performed to isolate and recover from failures.

A function it could perform is initiating self-tests in the different subsystems, and analyzing the test results in order to detect failures. The ST[17] Test Service could for example implement the self-tests initiated by the FDIR Manager.

If all efforts to recover from a failure fails, the FDIR Manager could disable that functionality permanently, or until operators can manually perform diagnosis to attempt to find and resolve the cause of the failure.

6.2 Service Layer

The services of the service layer are mission-independent, and represent the core of general satellite operation. These services can be selected mostly independently from each other, and reused in any future satellite projects.

6.2.1 Messaging Service

The Messaging Service is the other services' and modules' main way of communicating. It acts as an abstraction layer for inter-module communication, with the goal of hiding the details of *how* messages are passed between modules.

Each module has a high-priority message queue and a low-priority message queue assigned to it. The message queues can be read asynchronously, with an associated timeout. When a message is read, the high-priority queue is checked first, and then the low-priority queue. Messages to and from the ground segment will always be of high priority. Optionally, a third priority reserved for ground segment messages could be introduced, in order to ensure telecommands and telemetry is always prioritized.

There are two scenarios to consider for inter-module communication: the communicating modules are located in the same application process (internal messaging), and the two communicating modules are located in different application processes (external messaging). The Messaging Service utilizes the APIDs in the message header to identify whether the message should be routed internally or to one of the other application processes.

Internal Messaging

When the source and destination of a message is in the same application process, the Messaging Service only acts as an abstraction layer to the OS' inter-thread communication services. Figure 6.1 shows this behavior.

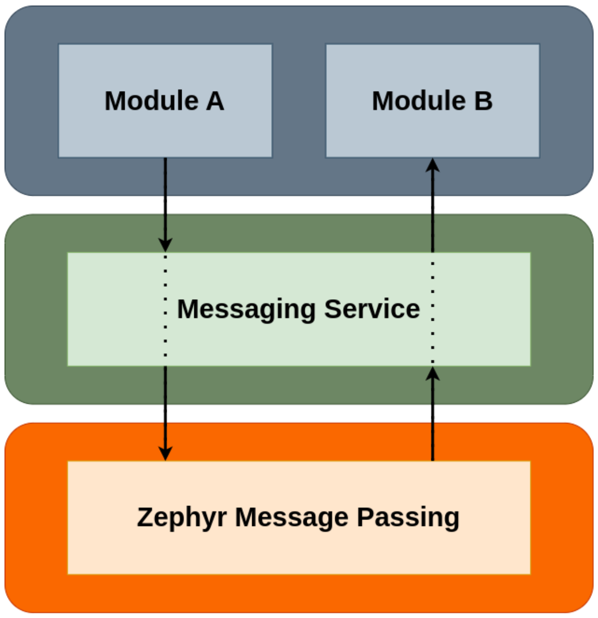

Figure 6.1: Internal messaging using the Messaging Service. A message is sent from module A to module B.

External Messaging

When the source and destination of a message are different application processes, the Messaging Service is responsible for sending the message to the communication interface used for external communication. For the CDHS on BioSat this is the CSP Service. Messages are then sent to the CSP Service for transmission using the CSP UDP packet format.

The path a message takes in this scenario can be seen in Figure 6.2. Here it is assumed that the destination application process/subsystem has implemented a similar service for routing messages.

If a message has the ground segment as destination, and the satellite is currently out of range of a ground station, the Messaging Service is responsible for passing that message to ST[15] for on-board storing.

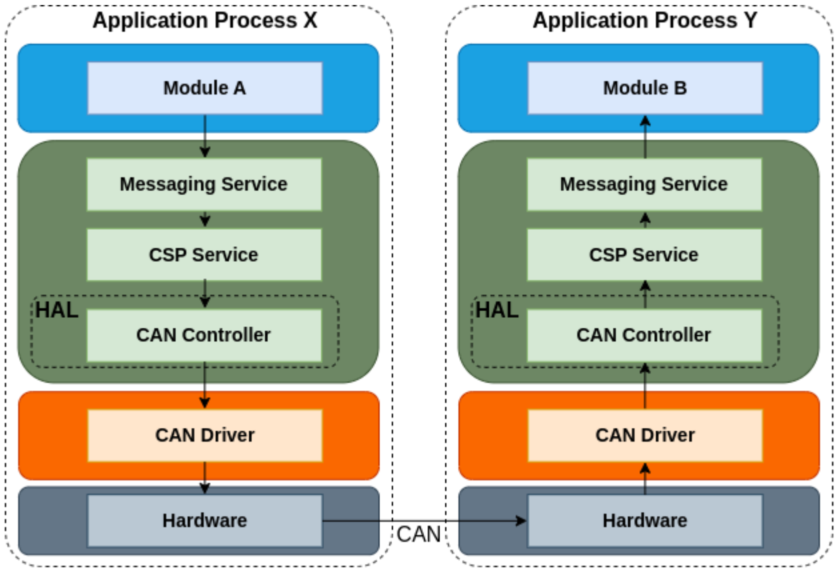

Figure 6.2: External messaging using the Messaging Service. A message is sent from module A to module B over the CSP network.

Request Acceptance Verification

Since the Messaging Service acts as the last routing step before requests reach their destination, the Messaging Service is responsible for initiating acceptance verification. If the request passes the acceptance verification, the Messaging Service will forward the request to the destination. If it fails, the request is discarded.

The actual acceptance verification is performed by the PUS ST[1] Request Verification Service. So when the Messaging Service receives a request, it passes it on to ST[1], which performs the verification checks and generates the corresponding acceptance verification report.

6.2.2 Logging Service

The Logging Service provides capability for controlling and getting information about the logging settings. An exact functional specification must be created once the chosen logging API and its capabilities are better understood. Nevertheless, here is a list of requests the Logging Service should be able to execute:

- Send a list of the names of the modules using the logging API, and their corresponding log module ID (used for controlling settings for that specific module).
- Enable and disable different logging backends (UART, USB, log file, etc.)
- Set log filters per module.
- Get the filepath to a module's log file.

These requests are intended to be issued by the operators on ground.

If a message has the ground segment as destination, and the satellite is currently out of range of a ground station, the Messaging Service is responsible for passing that message to ST[15] for on-board storing.

Logging API

Zephyr provides a highly versatile and feature-rich Logging API. The API is what is used for actually using the logging functionality, such as writing log messages, registering logging modules, etc. Some of the essential features of Zephyr's logging API are:

- Log entry severity levels: ERROR, WARNING, INFO and DEBUG.
- Log filtering based on the severity levels.
- Module-names to clearly distinguish log entry sources.
- Timestamping of log entries.
- Support for different backends, some of which are UART, USB, or files.
- Two processing modes: deferred mode and immediate mode.
  - Deferred mode uses a separate thread to process the log entries when the CPU load is low, and is preferred in situations where performance or execution time of the application is important.
  - Immediate mode processes the log entries immediately, making the response time of the logging shorter but impacting the performance of the application.

Since using already created and tested software is more time efficient and likely more reliable, the Zephyr implementation is recommended to be used for BioSat.

In order to avoid sacrificing portability and reusability, an interface to Zephyr's logging functionality should be created. This will enable the other services to use the logging API without having to directly depend on Zephyr-specific source code. Changing the underlying logging API (e.g. as a result of switching to a different OS) will then have no effect on the modules using the logging API interface, as long as the new underlying API can provide the same functionality.

6.2.3 Time Synchronization Service

The Time Synchronization Service is used for synchronizing the on-board time reference with a chosen on-ground time reference, and to distribute this time reference to the other on-board devices. The on-board time reference is kept by an RTC-module on the OBC.

The service provides two different methods of synchronizing other on-board devices: it can periodically send a synchronize request containing the time reference to the device, and it can provide the time reference in a report in response to a ”get time” request.

Whenever a subsystem reboots, it should request the time reference from the Time Synchronization Service in CDHS.

The service provides the following capabilities to the ground segment and to other subsystems:

- Set time reference. Intended to be used by operators in order to synchronize satellite time with ground time.
- Get time reference.

- Add periodic time distribution destination. Enables operators to specify a destination module (identified by APID + module ID) for periodic time distribution.
- Delete periodic time distribution destination.
- Set time distribution period per destination. Setting the period to 0 disables periodic time distribution.

6.2.4 CSP Service

The CSP Service implements the CSP protocol stack. It provides the capability to send and receive messages to and from the nodes on the CSP network.

The modules in the same application process as the CSP Service usually don't interact with it using the standard requests and reports. Instead, there is an API that the modules can use in order to send and receive messages over CSP. The CSP API provides sockets that a module can read and send through. In this design, most calls to the CSP API are abstracted away. Instead, they are called implicitly through the Messaging Service.

The CSP Service contains a router task that is responsible for outputting the message onto the correct interface. For most subsystems this is the CAN interface, with the exceptions of the CDHS and the TT&C who also can output on the RS424 (UART) interface and the radio interface, respectively.

The router task uses a routing table that can be configured at runtime. The routing table is used to determine which interface to forward a message to, and to specify a via address that the message will be sent to for further routing. The only example of a via address in BioSat's CSP network is the TT&C subsystem. When a message is sent from the CDHS with the ground segment as the destination, the routing table used by the CSP Service provides the address of the TT&C as the via address, and outputs it on the CAN bus or the RS424 link. Which interface is used is specified in the routing table. Thus, CSP messages between ground and CDHS could be routed onto the RS424 link, which provides higher throughput than the CAN bus.

The CSP Service provides the following capabilities through standard request/report messages:

- Set routing table.
- Get routing table.
- Ping.
- Get uptime.

There already exists a CSP implementation written in C called libcsp, which has been used on both SelfieSat and FramSat-1. It is not officially ported to Zephyr RTOS yet, but judging by the commit history, issues and pull requests on the GitHub page, Zephyr has at least partial support. The Zephyr implementation has probably not been as rigorously tested as the officially supported implementations. See this commit for instructions on how to build the CSP library for Zephyr.

6.2.5 File Transfer Service

The File Transfer Service implements a file transfer client and a file transfer server. This service is intended to be used by the PUS ST[23] File Management Service, and can not be used directly through requests and reports as many of the other services are. More specifically, it acts as the underlying file copy handler used by the File Copy Subservice of ST[23] when copying a file from one file system to another (e.g. uploading a file from ground to the CDHS).

The specifics of the design of the service and client are still unclear, as a file transfer protocol has not yet been decided on. Some general requirements are:

- The client must be able to issue requests for uploading files to and downloading files from the remote file system.
- The server must be able to handle upload and download requests.
- The server and the client must be able to split a file into fragments and send them.
- The server and the client must be able to track which fragments of a file have been received, and request missing fragments.
- The server and the client must be able to resend a specific fragment if it is requested.
- The server and the client must be able to reconstruct the file from the received fragments and store it at the correct filepath.
- The server and the client should be able to calculate and send a checksum of the file that is to be transferred.
- The server and the client should be able to verify that the checksum of the received file is equal to the received checksum.

The File Transfer Service will act as a layer on top of the existing communication protocols. Since file transfer involves transferring several file fragments that may arrive out of order or not at all, using CSP's Reliable Datagram Protocol might be a good basis to build the file transfer protocol on top of.

6.2.6 Hardware Abstraction Layer - HAL

The hardware abstraction layer consists of several interfaces to hardware drivers, called hardware controllers. These interfaces enable the creation of hardware-independent modules.

The hardware controllers attempt to present the core functionality of the hardware *type*. This functionality is defined in an interface, which in C code usually takes the form of a header file. For example, the header file for an RTC controller might contain function declarations such as *rtc_set(...)* and *rtc_get()*. The hardware controller can have multiple implementations, usually one per specific device of that device type. For example, the RTC controller might have one implementation for a DS1307 RTC module and another implementation for a DS3231 RTC module. The implementation of a hardware controller implements the functions declared in the header file (*rtc_set(...)* and *rtc_get()*) by using the hardware driver of that device. Changing which implementation to use (e.g. if you have switched to a different RTC module) is then as simple as changing some linker flags when the application is compiled.

6.2.7 Operating System Abstraction Layer - OSAL

The operating abstraction layer serves the same function as the HAL, except it provides an abstract interface to the functionality of the OS instead. This will reduce the efforts needed to port the satellite's software to a different OS. If an OS already has support for the chosen OSAL, porting the software might require no changes at all. A huge benefit of using an OSAL is the possibility to easily run the software on a computer more suited for testing, such as the computers used during development.

POSIX is a set of standards for maintaining software compatibility between operating systems. The POSIX standards provide a programming interface to many common OS services, such as I/O control, file management operations, multi-threading and synchronization mechanisms, and more.

Most UNIX-like systems (e.g. the various Linux distributions) are partly POSIX compliant, meaning they implement a subset of the POSIX capabilities. Zephyr is also partly POSIX compliant, as it implements a subset of embedded profiles PSE51 and PSE52, and BSD Sockets API (Zephyr Project, 2023*h*).

Since writing an OSAL is very time consuming, it is recommended that the existing POSIX support is utilized as much as possible. Custom interfaces can then be created for any required Zephyr functionality that is not covered by POSIX.

6.2.8 Packet Utilization Standard Services - PUS Services

The PUS services implement a range of central capabilities that are useful in most satellite missions. The ECSS standard ECSS-E-ST-70-41C (ECSS, 2016*a*) contains extensive definitions of the capabilities of the various services, and of the interfaces to those services. Using this standard when implementing these services is highly recommended.

The Packet Utilization Standard specifies that the service definitions should be considered as a "menu" from which to choose the applicable services. Since development time is limited, the chosen services have been given an implementation priority. The list of chosen services and their priority can be found in Table 5.1.

In this section, the capabilities of the various services that have been deemed suitable for use in the CDHS is described briefly. The services' roles in the CDHS (or the satellite as a whole) are also described.

Note: Many of the PUS services are very extensive in their functionality. The standard encourages customizing the selected services by choosing the capabilities that are most relevant to each mission. As such, not every capability of each subservice needs to or should be implemented, as this would require too much development time.

ST[1] Request Verification Service

The Request Verification Service (ECSS, 2016*a*, p. 55-63) contains the routing and reporting subservice, the acceptance and reporting subservice, and the execution reporting subservice.

The **routing and reporting subservice** is not used, due to routing to other subsystems primarily taking place in the TT&C subsystem.

The **acceptance and reporting subservice** provides the capabilities to perform acceptance verification for requests, and generate reports notifying of successful or failed acceptance. This subservice is used by the service that is responsible for routing requests to their final destination in the CDHS, which is the Messaging Service (see section 6.2.1. The Messaging Service initiates acceptance verification for a request. The resulting report is sent back to the Messaging Service, which uses it to determine if it should forward the request or discard it. Then the acceptance report is sent to the source of the original request.

The **execution reporting subservice** provides the capabilities to generate reports related to the execution of a request. This subservice is used by all the subservice providers in the CDHS when they are or attempt to execute a request. The subservice provides the capability to generate reports of the following events:

- Successful or failed start of execution.
- Successful or failed progress of execution. Successful reports contain an estimate of the current progress.
- Successful or failed completion of execution.

ST[2] Device Access Service

The Device Access Service (ECSS, 2016*a*, p. 64-77) provides the capability of distributing commands to and acquiring data from the on-board devices.

This service is intended for bypassing the normal interaction with on-board devices, and not as the main method for CDHS-modules to interact with devices connected to the OBC. It can for example be used by the operators for troubleshooting-purposes, or potentially by other subsystems that need access to a device normally controlled by the CDHS (i.e. connected to the OBC).

ST[3] Housekeeping Service

The Housekeeping Service (ECSS, 2016*a*, p. 78-110) contains the housekeeping reporting subservice, the diagnostic reporting subservice, and the parameter functional reporting configuration.

The **housekeeping reporting subservice** provides the capability to generate parameter reports, both periodically or on request. A parameter report contains one or more on-board parameters, their corresponding values, a collection interval representing the time interval at which the parameters were collected, and a parameter report identifier. The housekeeping parameter reports are generated during nominal operations, and contain important data related to the state of the satellite (e.g. battery voltage, temperature measurements, amount of free memory, etc.). The parameter reports are downlinked to the ground segment when the satellite is in range of a ground station. There they can be used for anomaly detection, behavior analysis, or for diagnostic purposes.

The **diagnostic reporting subservice** provides similar capabilities to the housekeeping reporting subservice. The main difference is that this subservice is dedicated to parameter report generation during non-nominal operations, and mainly for diagnostic purposes.

The **parameter functional reporting configuration subservice** is not used. It provides a more convenient way to configure the generation of groups of parameter reports. This subservice can easily be added at a later point in time if deemed necessary.

Both parameter reporting subservices provide the capabilities to enable, disable, and set the period of the generation of individual parameter reports. They also enable creating and deleting new parameter report structures, which can then be used when generating parameter reports.

The parameter values included in the parameter reports can be collected from the ST[20] Parameter Management Service.

ST[4] Parameter Statistics Reporting Service

The Parameter Statistics Reporting Service (ECSS, 2016*a*, p. 111-120) service provides the capability to generate reports containing statistics of on-board parameters. The maximum, minimum, mean and standard deviation values of the parameters measured during a time interval can be reported.

This service is useful for saving downlink bandwidth, as the statistically significant values can be calculated on-board instead of having to downlink every sample and perform the calculation on ground.

ST[5] Event Reporting Service

The Event Reporting Service (ECSS, 2016*a*, p. 121-126) provides the capability to generate event reports when an event is detected.

Events can be on-board failures and anomalies, the initiation of a specific action, or some other specific set of conditions. The System-Level FDIR Manager may for example raise an event if it detects a failure.

When the Event Reporting Service detects an event, a corresponding event report is generated, containing the event ID and any auxiliary data. The mechanisms used to detect events are not clear at the time of writing. All event reports have the same destination, and in CDHS this is the ground segment.

The Event Reporting Service may detect events raised by the ST[12] On-Board Monitoring Service or the System-Level FDIR Manager.

When an event report is generated, the ST[19] Event-Action Service can detect this, which may result in a request related to the event being issued. This can be an important part of the on-board autonomy.

ST[6] Memory Management Service

The Memory Management Service (ECSS, 2016*a*, p. 127-155) contains the raw data memory management subservice, the structured data memory subservice, the common memory management subservice, and the memory configuration subservice.

The **raw data memory management subservice** provides the capability to manage memories that contain raw data (data whose content or structure is not known). That includes loading raw data memory areas, dumping raw data memory areas, and checksumming raw data memory areas. This subservice can possibly be used for loading a specific flash partition with an application image in order to perform a firmware upgrade. This subservice also enables calculating the checksum of that image, which is important for making sure the image has not been corrupted.

The **structured data memory management subservice** provides similar features to the raw data subservice, but for structured data (files, structs, etc.) instead. Whether this functionality is needed is not clear to the author, as the ST[23] File Management Service takes care of the most common file management needs. Because of this, this subservice is only considered a "nice to have".

The **common memory management subservice** provides the capability to abort all ongoing memory dumps.

The **memory configuration subservice** provides the capability to perform memory scrubbing on a memory area (raw or structured) and enable or disable write protection of a memory. Memory scrubbing attempts to correct any bit errors in the memory, and write protecting protects against accidental writes in important areas. Both can extend the lifetime of the satellite and increase its reliability.

ST[11] Time-Based Scheduling Service

The Time-Based Scheduling Service (ECSS, 2016*a*, p. 168-197) provides the capability load requests on-board and schedules the release time of those stored requests. This can be used to issue a request at a time the satellite is outside ground station range and cannot receive direct requests from ground.

Any reports generated from the released request are addressed to the original request issuer (i.e. not the Time-Based Scheduling Service). So any reports generated by requests originally issued by the ground segment have the ground segment as destination address.

ST[12] On-Board Monitoring Service

The On-Board Monitoring Service (ECSS, 2016*a*, p. 198-228) consists of the parameter monitoring subservice, and the functional monitoring subservice.

The **parameter monitoring subservice** provides the capabilities to perform parameter value checks, which include checking that the parameter value lies within specified limits, has the expected value, or that the rate of change of the parameter value lies within specified limits. It also provides capabilities for specifying and configuring parameter checks and intervals. The parameter values are collected from the ST[20] Parameter Management Service.

The subservice provides the capability to raise events when a parameter value exceeds the specified limits. These events are then caught by the ST[5] Event Reporting Service, resulting in event reports that are sent to the ground segment and can optionally be detected by the ST[19] Event-Action Service.

The **functional monitoring subservice** provides the capability to monitor the functional health of on-board elements. These elements can be a hardware device or a software module.

Functional monitoring definitions consist of a set of parameter monitoring definitions. When a specified subset or number of the limits of these parameters are violated, an event is raised.

This subservice enables detecting more complex events or failures, and can as the name implies detect when the functional ability of some element is compromised.

ST[13] Large Packet Transfer Service

The Large Packet Transfer Service (ECSS, 2016*a*, p. 229-236) consists of the large packet downlink subservice, and the large packet uplink subservice. These subservices provide the capabilities to send large messages that would normally exceed the Maximum Transfer Unit (MTU) of the network. It does this by splitting the large packet into numbered, smaller packets that are sent over the network. A corresponding service at the receiving node is then responsible for reconstructing the large packets from the received fragments.

In PUS it is specified that the service provides the capability for sending large packets between the space segment and the ground segment. In the architecture design presented here, however, there is no apparent reason to enforce this limitation. The Large Packet Transfer Service could, therefore, also be used to send large packets from one subsystem to another, if the MTU of the link between those subsystems were a limiting factor.

This service would be automatically used by the Messaging Service when it receives a message that is too large to be sent as is.

Note: Libcsp has implemented a similar feature in the form of an application layer protocol called Small Fragmentation Protocol (SFP). Exactly how this protocol works, and if it is a suitable replacement for the Large Packets Transfer Service should be investigated.

ST[15] On-Board Storage and Retrieval Service

The On-Board Storage and Retrieval Service (ECSS, 2016*a*, p. 256-316) consists of the storage and retrieval subservice and the packet selection subservice.

The **storage and retrieval subservice** provides the capability to store telemetry packets on-board, and retrieve the stored telemetry packets when requested by the ground segment. It also provides the capability to automatically timestamp the stored packets with the time-of-storage, and manage the packet stores by e.g. creating and deleting them, copying the stored packets to a different packet store, and reporting the contents of packet stores.

The **packet selection subservice** provides the capability to define control conditions that enable or disable the storage of certain packets or of all packets that would be stored in a certain packet store.

This service will be used to store any telemetry packets (i.e. messages addressed to the ground segment) on-board when the satellite is out of range from any ground stations. The telemetry packets generated by the ST[3] Housekeeping Service, the ST[5] Event Reporting Service and as a result of time-scheduled telecommands released by the ST[11] Time-Based Scheduling Service, will all be stored in the on-board packet stores. The operators on ground can then request the downlink of the stored telemetry packets.

ST[17] Test Service

The Test Service (ECSS, 2016*a*, p. 318-320) provides the capability to activate test functions implemented on-board and to report the results of those tests. It defines two tests: an are-you-alive connection test, and an on-board connection test. The service's capabilities can be extended with custom tests which can then be used in FDIR procedures.

The are-you-alive connection test is requested by the ground segment. When the subservice receives the request, it generates a corresponding report. If the ground segment receives the report the application process is alive and the communication between it and ground works.

The on-board connection test tests the connection between the application process hosting the subservice and an application process specified in the request. If the ground segment receives a successful connection report it indicates that the two application processes are able to communicate.

ST[18] On-Board Control Procedure Service

The On-Board Control Procedure Service (ECSS, 2016*a*, p. 321-341) consists of the OBCP management subservice and the OBCP engine management service.

The **OBCP management subservice** provides an interface to the OBCP engine that is responsible for executing OBCPs. This interface enables operators to load an OBCP into the engine (the OBCP must be stored somewhere on-board already), unload it (e.g. in order to load a different OBCP), activate it, stop a running OBCP, abort a running OBCP, suspend a running OBCP and resume a suspended OBCP. It also provides the capability of reporting the execution status of a given OBCP.

The **OBCP engine management subservice** provides the capability to control the OBCP engine by issuing start or stop requests. The OBCP engine must be started before the execution of any OBCPs can be initiated by the OBCP management subservice.

This service is entirely dependent on there being implemented an OBCP engine that it can manage. Without an OBCP engine that can execute the OBCPs, this service does nothing.

ST[19] Event-Action Service

The Event-Action Service (ECSS, 2016*a*, p. 342-351) provides the capability to define a list of on-board actions that can be autonomously executed when a corresponding event occurs. This service is associated with one or more ST[5] Event Reporting Services, and should be able to observe any event reports generated by the associated services.

The actions triggered can be any on-board request, such as starting an OBCP or a specific request sequence, configuring hardware or initiating processing of payload data. This service will therefore be important when defining the autonomous behavior of the satellite.

ST[20] Parameter Management Service

The Parameter Management Service (ECSS, 2016*a*, p. 352-357) provides capabilities for managing on-board parameters. The service enables reading the current values of on-board parameters, setting values of parameters, and managing the parameter definitions (e.g. changing where a parameter is stored in memory).

On-board parameters managed by this service could for example contain the most recent battery voltage measurement, the angular rotation rate of the satellite, various sensor readings, status of different hardware devices (e.g. whether the device is on or off), and more.

This service can be used by any module that requires knowledge of the satellite's state or condition. The ST[3] Housekeeping service can use the Parameter Management Service to collect the parameter values that are included in the housekeeping and diagnostic reports. The ST[12] On-Board Monitoring Service could use this service in order to collect values of the parameters that are being monitored. The ST[17] Test Service could define tests that examine a set of parameter values and compare them to a set of expected values or conditions.

Which modules are responsible for updating the values of the various parameters is yet to be decided.

ST[21] Request Sequencing Service

The Request Sequencing Service (ECSS, 2016*a*, p. 358-367) provides the capabilities for releasing requests on-by-one from a sequence of requests that have been uploaded to the satellite. It also provides the capability to load, activate and unload a sequence of requests.

Any reports generated in response to the released requests are sent to the address contained in the source address in the requests. This means that operators on ground could upload a request sequence where each request has the autonomy manager as source address. When the requests are released by the Request Sequencing Service, any reports generated in response to those requests would then be sent to the Autonomy Manager. The reports could also be sent addressed to ground of course, simply by setting the source address in the requests to the address of the ground segment.

The Request Sequencing Service enables operators to automate the operation of the satellite to some degree. This is the intended way to perform mission operations according to the ConOps (see section 1.2.2). Request sequencing will drastically reduce the manual involvement required from the operators, and will also enable the operation of the satellite when it is outside ground station communication range.

ST[23] File Management Service

The File Management Service (ECSS, 2016*a*, p. 403-424) consists of the file handling subservice and the file copy subservice.

The **file handling subservice** provides capabilities for interfacing with the on-board file handling system. It provides operations such as:

- Creating and deleting files.
- Reporting file attributes.
- Restricting access to files by locking and unlocking.
- Finding specific files.
- Creating, deleting and renaming directories.
- Listing the contents of a directory.

The **file copy subservice** provides capabilities for copying or moving files within a file system or between different file systems. This includes:

- Copying a file from a file system on the ground segment to a file system on-board (i.e. uploading files to the satellite).
- Copying a file from a file system on-board to a file system on ground (i.e. downloading files from the satellite).
- Copying a file from a file system in one subsystem to a file system in another subsystem.
- Copying a file from one location in a file system to another location in the same file system.
- Suspending and resuming a file copy operation.
- Aborting a file copy operation.
- Periodically generate reports on the file copy status.
- *Note*: The capabilities above also apply to moving files.

The file handling subservice will be the operators' main interface to the file system of any subsystem that implements this subservice.

The file copy subservice will be the main way of uploading and downloading files to and from the satellite, and of transferring files between subsystems on-board the satellite.

The file copy subservice will use the File Transfer Service as the underlying file transfer handler whenever files are copied/moved between an internal file system and an external file system (i.e. upload/download and transfer between subsystems). This implicitly creates some functional requirements for the File Transfer Service, such as being able to suspend, resume and abort a file transfer operation.

6.3 Low-Level Software Layer

6.3.1 Bootloader

The bootloader is a small program responsible for booting the main Zephyr application that makes up the CDHS' software. It will be flashed onto the MCU on the OBC at the start address of the flash memory.

The bootloader suggested by the author is the MCUboot bootloader, described in section 2.2.2. This bootloader has support and documentation for use with the Zephyr RTOS, which will be an advantage when implementing it.

MCUboot has support for software updates by using the two image slots of each image. When the OBC starts up, MCUboot will boot the Zephyr application loaded in the primary slot of the image. To perform the software update the update application image has to be uploaded to the satellite and stored in the secondary image slot. The updated image can be uploaded to the satellite through the ST[23] File Management Service (and implicitly the File Transfer Service). The image file can then be loaded onto the secondary image slot using the ST[6] Memory Management Service. The software of the satellite can then be updated by swapping the contents of the primary slot and the secondary slot, which MCUboot will do automatically on reboot if the required flags are set in the image headers. Once the updated software is running, a flag confirming successful software update must be set in the image headers in order to make the update permanent.

6.3.2 Hardware Drivers

A large selection of hardware drivers is already implemented for Zephyr. These drivers follow the Zephyr Device Driver Model, which provides a consistent device model for configuring devices (Zephyr Project, 2023*c*).

Any custom device drivers should also follow this model in order to keep the consistency with the existing drivers.

Chapter 7

Development

In this chapter, the efforts that were made to facilitate the development of the on-board software are described. This includes setting up a Docker container with all the necessary development tools (including Zephyr), adding debugging support, and configuring a documentation generation system.

7.1 Creating a Development Environment Using Docker

7.1.1 Docker Image

In order to streamline the development experience for the members of Orbit NTNU, a self-contained development environment was constructed using Docker. This was done by developing a Dockerfile that specifies a Docker Image containing all the development tools necessary for creating, compiling, and flashing Zephyr applications. See B.1 for the created Dockerfile.

Some important concepts used when creating the development image:

- Using arguments and environment variables in the Dockerfile in order to easily make updates to the development environment. For example, the environment variable "ZEPHYR_VERSION" can be set when building the Docker Image to configure which Zephyr version should be installed in the container.
- Installing Zephyr and all its dependencies. Since Zephyr is installed in the container, the computer of the developer can remain completely clean. The only thing needed to be installed locally on the developer's machine is Docker Engine, which is needed to run the Docker container.
- Create a non-root user in order to limit the container's access to the host machine's system files. Also removes any ownership issues related to the source code files being owned by root (which would prevent opening the source files when outside of the container).
- Automatically set up SSH configuration to enable the developer to interact with the project repository and push to GitLab. The private SSH key is added to the container as a secret, which makes it possible to have SSH access to the repository without storing the private key in the Docker Image.

- Installing the Zephyr SDK in the container, which provides a range of debugging and flashing tools, as well as supplier-specific software such as HALs for supported MCUs.
- Adding the container-user to the "plugdev" group in order to enable flashing the board from inside the docker container even as a non-root user.

By starting an interactive shell session in the resulting docker container, the developer would have access to any development tools needed to write Zephyr application source code, compile it into a Zephyr application and flash it to the board. They would also have access to development tools such as git, vim, and picocom (which can be used to get a serial interface to the board when debugging).

Docker Compose

A Docker Compose file was developed in order to make starting the container with the appropriate arguments more convenient (see B.2). Some useful features implemented using the docker compose file:

- When starting the container via the compose file, the directory containing the BioSat software (i.e. what is hosted on the GitLab page) is mounted inside the container. This ensures that changes made to the source files while working from inside the container is persisted after the container is closed.
- The starting directory when entering the docker container is set to the mounted directory.
- The device in "/dev" needed to program the board is passed through to the container, enabling flashing the board without exiting the container.
- The location of the SSH key that will be passed to the container is specified.
- The arguments used when building the Docker Image is conveniently grouped.
- An entry point script is specified, which contains a series of bash commands that will be executed every time the container is started.

Visual Studio Code Development Container

Visual Studio Code has a very useful feature called development containers. This allows automatically opening a VS Code instance from inside the docker container. This means that any development tools installed in the docker container are available to be used in the VS Code terminal, in addition to being able to use VS Code to open and edit files that are inside the container.

Additionally, a "devcontainer.json" file can be created (see B.3). This file specifies which specific docker compose file to use when starting the development container. It also enables specifying a list of VS Code extensions that will be automatically installed inside the development container, but not outside in local instances of VS Code.

The combination of a Docker Container, a Docker Compose file, and a VS Code Development Container essentially creates a completely self-contained development environment complete with the tools, IDE, and extensions needed to efficiently and conveniently develop software for BioSat.

7.2 Setting Up Zephyr RTOS for the STM32H745

Setting up the Zephyr RTOS for creating Zephyr applications for the MCU chosen to be used on the OBC was rather simple. The installation instructions from Zephyr's Getting Start Guide was in essence copied into the Dockerfile, resulting in Zephyr being installed inside the container.

To compile a Zephyr application for the target MCU, the "BOARD" environment variable had to be set to the board identifier of the development kit used during testing, which was "nucleo_h7a3zi_q" (for the STM32 Nucleo-144 development board with STM32H7A3ZI MCU (STMicroelectronics, 2023)).

With these steps, any of Zephyr's sample applications could be built and flashed to the devkit using the West commands "west build" and "west flash", respectively.

7.3 Software Project Structure

The on-board software for BioSat will be structured as a Zephyr application, with a similar structure to the one shown in Figure 2.2. The BioSat directory will contain the main manifest file, "west.yml", which is used to manage the project's dependencies (e.g. MCUboot, libcsp, and Zephyr itself). It will also contain a project configuration file, "prj.conf", which sets Kconfig values that configure the resulting application. The "CMakeLists.txt" specifies the source files that will be used to compile the application. The actual source code is stored in a "src/" folder.

The plan is to have the BioSat repository only contain mission-specific software. The PUS Services will be implemented in a separate library which can then be added to any project as a dependency specified in the manifest file. This will greatly reduce code duplication.

It might be possible to create a satellite bus repository that implements a complete and functioning CDHS that doesn't contain any mission-specific software. The satellite software repositories, such as the BioSat repository, could then use this as a dependency, and build the mission-specific application on top of the satellite bus software.

7.4 Enabling Debugging Support

West has built-in debugging support, and the Zephyr SDK contains several different debuggers (such as OpenOCD) and GDB binaries that can be used to debug a running Zephyr application. To start a debug session in the terminal using GDB simply run the terminal command "west debug".

In-editor debugging can also be enabled in VS Code by creating a standard debug task using the "launch.json" file (see B.4.1) and the Cortex-Debug extension for VS Code. It is possible to build the application using a specific debug Kconfig configuration using overlay config files when building. The debug configuration in the file in B.4.2 gets applied when building the application using the "West: build debug" build-task in B.4.3.

7.5 Documentation

Zephyr uses Sphinx to generate searchable HTML documentation that can be hosted on a web server. This was replicated to be used to generate documentation for the BioSat software. Generating documentation using Sphinx was set up by following the Sphinx Getting Started Guide.

Inside the "doc/" folder in the BioSat repository documentation files can be written in the reStructuredText format. These can then be built into a cohesive and searchable documentation platform using the "make html" command. Additionally, the Breathe extension was installed to be able to generate API documentation from in-code Doxygen documentation. This setup creates a comprehensive documentation that can be used to convey the higher-level concepts of the software modules, while simultaneously documenting the API and usage of the modules.

Chapter 8

Discussion

The software architecture that was designed in Chapter 5 and detailed in Chapter 6 is believed to fulfill the design objectives in section 4.1. It is also believed that it provides a satisfactory answer to the main tasks of the project, which can be found in section 1.3. There have, unsurprisingly, been hard decisions and uncertainty during the design process, as it turns out that designing a satellite's software is hard. Below, to what degree the suggested architecture fulfills the initial task and satisfies the architectural drivers are discussed, the decisions which the author is still uncertain about are discussed, and finally, the possible future work related to the design is discussed.

8.1 Does the Software Architecture Design Satisfy the Initial Task?

The suggested service-oriented architecture pattern is believed to do a satisfactory job of dividing the software into more or less independent modules and services. This will hopefully aid implementation substantially, as the software can more easily be tackled in "bite-sized" problems, and because it is easier to get an overview of the system. The architecture is believed to have a similar impact on maintainability, as the independent nature of the modules leads to needing fewer compatibility changes whenever a module-design changes. Testing and integration might also benefit from the design, due to the requirement of services and modules having clear and standardized interfaces.

The use of the standard services defined in the packet utilization standard (detailed in section 6.2.8) is believed to be of great benefit to the design and implementation of the software architecture. One benefit is being able to rely on a standard that has been used in the industry by leading actors such as the European Space Agency. Such a standard can be relied on to have been thoughtfully created and tested, which any custom service definitions created by Orbit NTNU likely could not (at least not to the same degree). Another benefit is the clear definitions of the services' functionality and their interfaces, which can make it easier for the developers to create the service implementations. The greatest benefit is related to the reusability of the services, as they are specifically designed to be useful in most satellite missions. This enables implementing a PUS library, which again enables future missions to pick and choose from the list of implemented services.

The layered architecture pattern is also a major contribution to the reusability and portability of the software. The two abstraction layers enable writing hardware and OS-independent software, which can then easily be reused in future missions that might have other needs related to hardware and OS capabilities.

When it comes to the reliability of the system efforts have definitely been made towards increasing it. Independent services and modules will make it easier for developers to create correct code, as there are fewer dependencies to keep track of and the intended functionality of each element may be more easily understood. The local error handling strategies, such as bounds checking and return value handling, make for more reliable code. The reliability measures applied on a slightly higher level, such as watchdog timers, error-detection and correction codes, retry strategies, and firmware rollback all contribute to a more reliable (and to some degree robust and resilient) system. Services dedicated to the handling and prevention of failures (i.e. the parameter-monitoring service and the FDIR manager) take this a step further by actively attempting to identify and react to faults, errors, and failures. There is still work to be done in this area, however, as the specific functions of the FDIR manager are not yet defined. To which degree the architecture implements robustness and resilience is hard to say, but some of the measures (such as watchdog timers) can also be used to recover from unknown or unanticipated events.

The largest concern regarding the fulfillment of the initial task is related to the validity of the base that the software architecture is designed from, namely the architectural drivers. Identifying which design objectives are important, which functional requirements are needed, and which quality attributes should be focused on (and in which way), is a challenging task. The drivers identified in Chapter 4 were done so using external sources such as ECSS standards, other design reports about satellite software architecture, old requirements and designs from Orbit NTNU's previous satellite projects, and my own understanding of the BioSat mission and in which areas a new design had the most to gain. As development kicks off, and during most of the development process, these drivers will likely change, and so will the corresponding design. The hope is that when these changes take place, the software architecture designed in this thesis is modular and maintainable enough that any changes are easily accommodated into it without requiring rewriting large parts of the software.

8.2 Hard Decisions and Uncertainties

There are several decisions that were hard to make, and the author is still uncertain as to whether they are the best solutions for the various problems.

One of those decisions was opting for a distributed service-oriented architecture. The service orientation lends itself to concurrent implementation, which is useful in an organization such as Orbit NTNU that have a rather high turnover rate. On the other hand, distributed services might obfuscate the flow of information, as any service could, in theory, communicate with any other service. Having clearly defined interfaces and keeping an overview of all services used by each module/service can potentially mitigate this drawback.

A very hard choice was deciding whether the subsystems should be completely distributed. That is, whether telecommands and telemetry should be handled and generated in a central subsystem (i.e., the CDHS), or if each subsystem should be completely responsible for its own telecommand handling and telemetry generation (the distributed scenario). The distributed scenario was eventually chosen, as the single point of failure the CDHS would create otherwise was too large of a downside. This does, however, mean that every subsystem must be able to perform its own telecommand validation and generate telemetry packets, which would in some cases limit how simple a subsystem implementation can be. A possible solution to this is to create a telecommand verification service and a telemetry generation service on the CDHS, which resource-constrained subsystems could use. Another point of uncertainty is whether bus contention will be an issue, as all subsystems will communicate over the same CAN bus. CAN has built-in bus contention resolution, but it could still have an impact on performance and might delay the arrival of important telecommands. To resolve this issue, it should be possible to set a message priority in order to ensure a message is delivered as quickly as possible to and from the ground segment. Message priorities can possibly be implemented using CSP priorities, but this needs to be further investigated.

There are some drawbacks related to utilizing the PUS standard services. The PUS service definitions are very comprehensive, and they can be (too) challenging to implement compared to the available development time. The PUS implementation could be seen as a long-term investment though, and only the essential services are implemented for each mission. Another drawback with the PUS services is that they all seem useful in some way, which can clearly be seen by the sheer amount of services selected in the design presented in this thesis. In an effort to try to limit the feature creep and keep the complexity low, the different services were assigned an implementation priority rating in Table 5.1. This will hopefully help the developers to prioritize implementing the essential functionality. Useful, but non-essential services can then be implemented if there is available development time before launch.

The last major uncertainty is related to the development period constraint of 2 years. The design presented in this thesis will require large development efforts. Together with members having to learn how to use Zephyr and understand how to create and use the abstraction layers, the suggested might be too complex to realistically implement before launch. The author, therefore, suggests creating a revised minimal configuration of the software architecture that only contains the essential functionality. This revised architecture should be designed in a way that facilitates extending it to include the capabilities provided by the design in this thesis. The (potential) feature creep of this design could have been partially avoided if a combination of the top-down and the bottom-up architecture design approach had been used (instead of a strictly top-down approach), but this proved infeasible as the team members of the Embedded team, which is the team responsible for the CDHS and most of the on-board software, was occupied with developing the FramSat-1 satellite.

8.3 Future Work

Creating a comprehensive software architecture design for the on-board software of a CubeSat proved hard work, and there are seemingly enough problems to be solved to fill several master's theses. Orbit NTNU, therefore, has no shortage of tasks to fulfill and decisions to be made related to the design presented in this thesis. The following list contains future work that could (and in some cases should) be done in order to best utilize the software architecture design suggested in this thesis.

- **Define initialization procedures for the various subsystems**, including procedures for when a subsystem is rebooted. A concept that has been seen in other designs and that could be explored is a "context service." The context service is responsible for periodically saving the operational context of the subsystem to some permanent storage so that the subsystem can fetch its previous context and pick up where it left off before rebooting.
- **Define satellite operational modes**, i.e., which subsystems, services, and autonomous functions are enabled for the various modes and the conditions or events that trigger mode transitions.
- **Define the failure modes of the system**, modeling the potential failures, their expected frequency, and their criticality. It can be useful for creating FDIR functions.
- **Define FDIR functions to be implemented in the system-level FDIR manager**. FDIR could be a thesis by itself, and there are many possible strategies and mechanisms that could be employed, with varying degrees of complexity. Further research and design efforts should be made in this area.
- **Create requirement specification documents detailing functional and non-functional requirements**. Use a requirements standard, and refine the requirements to be testable and measurable. McConnell (2004, p. 468) recommends chapter 9 of *Principles of Software Engineering* (Gilb, 1988) for details on measuring quality attributes. Related to section 3.3.1 and creating good non-functional requirements.
- **Define mechanisms for on-board storage of messages addressed to ground**. How does the ST[15] On-Board Storage and Retrieval Service know whether it should store a telemetry packet or let it pass directly to the radio? It can possibly use the expected interval of ground connectivity, which can perhaps be calculated by the ADCS (i.e. through orbit propagation). Alternatively, it could check with the TT&C subsystem whether a ground station is in range, but checking whether it is in range of a ground station each time it should decide whether to store a package or not is not very efficient. An additional problem that needs to be solved is how the service interferes with normal message sending in order to get a copy of the packets in order to store them.
- **Create a repository for the PUS services that can be used as a library for any project that might need it**. Different missions can pick the services they need.
- **Create a repository for the satellite bus, containing all the mission-independent software**. Future missions could then build mission-specific software on top of a common satellite bus.

- **Add a standardized way to get thread-safe access to on-board devices such as sensors and actuators**. Several threads might attempt to use the same device at the same time, which will lead to race conditions and unpredictable behavior. Solutions would include some form of synchronization mechanism. A possible solution is making the functions in the HAL threadsafe, but it is unclear how difficult this would be.

- **Define a retry-strategy for the Messaging Service**. When a request is sent from the CDHS to another subsystem, and the module that issued that request expects an acceptance report, the Messaging Service should resend that request if a report is not received within a specified timeout. The number of retries and the timeout period should be configurable. An implementation would need to have the messaging service store messages and be able to identify a received acceptance report and correlate it to one of the stored messages.

- **Define the format of requests and reports**. Additionally, service users should be able to identify that a received report corresponds to a specific request. A service user might receive several reports at the same time, and the report received might not be the one corresponding to the request that was most recently issued. The message format used for requests and reports must include an ID field that uniquely identifies individual messages.

- **Investigate if the Small Fragmentation Protocol implemented in libcsp could replace ST[13] Large Packet Transfer Service**. Libcsp apparently has a fragmentation protocol that can be used for transferring messages with a size larger than the network's MTU. Whether this can replace ST[13] must be investigated further, as the documentation of the protocol is severely lacking. See the last paragraph of the page about CSP MTUs on libcsp's GitHub page.

- **Figure out how on-board parameters are updated**. The ST[20] Parameter Management Service provides capabilities for reading and setting on-board parameters. This service will be used by many other modules in order to gain access to the state of the satellite and its various elements. Many of these parameters are related to sensor measurements or other states that change over time. Efforts should be made to figure out the best way to update the values of these parameters. Problems that need to be solved are:

  - How often should a parameter be updated? Updating a sensor measurement parameter every time the sensor is read would probably have a negative impact on performance, especially in real-time tasks such as the ADCS control algorithms.

  - Which module should be responsible for updating the various parameters? Should the parameter be updated by the module that performs the measurement or that causes the change? Or should there be a dedicated service that is responsible for periodically reading sensors and polling modules for their status, and then updating the parameter values?

- **Define software update procedure**. In the current design, the capability to perform in-orbit updates exist, but the approach is very rudimentary and required a lot of manual input. Currently, an application image must be uploaded to the satellite as a file, which must then be loaded into the correct image slot by using the ST[6] Memory Management Service, and then setting the flags in the image header for initiating an image update on reboot. A better solution would be to design a service that implements software updates in a more reliable way. The service could for example automate the loading of the software image into the correct image slot and setting the corresponding header flags. It could also perform tests after booting into an updated image for the first time, and, if it passed the tests, set the flag in the image header that would persist the update.

- **Define autonomy capabilities and levels**. The capabilities of the Autonomy Manager should be decided on. This will depend on the mission data that is generated by BioSat. Any autonomous functions that are identified during development that are useful for the general operation of the satellite could also be implemented here. The autonomy levels of the satellite should be defined. The ECSS standard ECSS-E-ST-70-11C defines four autonomy levels and defines requirements related to on-board autonomy (ECSS, 2008, p. 32-34), and could be a good source of inspiration.

Chapter 9

Conclusion

A list of architectural drivers (i.e. the reasons for why a software architecture should be produced and what it should facilitate) have been identified for the BioSat software architecture, and a requirements specification with higher-level functional and non-functional software requirements based on those drivers has been suggested. These requirements should be revised and elaborated on as the development of the satellite progresses.

A service-oriented software architecture with a focus on reliability, maintainability, and reusability has been proposed. The suggested architecture mainly concerns the Command and Data Handling Subsystem, but it also addresses the interaction between the various subsystems. The three main benefits of the architecture design are:

- *Ease of development*, as it clearly separates the functionality into cohesive and loosely coupled services and modules. Furthermore, the independent nature of the services and modules designed in this thesis makes it possible for developers to concurrently implement multiple services and modules, and it also makes testing and integration easier. The separation of concerns and the use of standardized interfaces in the design are also believed to make it easier for developers to maintain the software.
- *High degree of reusability in future missions.* The architecture attempts to clearly separate functionality that is useful to most satellite missions from the functionality that is specific to and varies with each mission. This can to a large degree be credited to the use of the Packet Utilization Standard services. Additionally, the two abstraction layers in the design ensure that most of the software can be run on a range of different operating systems and hardware configurations.
- *A fault tolerant and reliable system.* By opting for a service-oriented architecture design, local error handling is made easier to implement correctly as there are fewer dependencies to manage per service or module. By using CSP as the communication protocol between the subsystems and between the ground segment and the satellite, each subsystem can be commanded directly from the ground, possibly allowing satellite operation even in the event of a complete subsystem failure (though with reduced functionality). The inclusion of a satellite health monitoring and reporting module, a Failure Detection, Isolation and Recovery module, and the use of watchdog timers in the design further contribute to the robustness and resilience of the system.

The software architecture design was not created alongside active development or implementation of the system and therefore might suffer from being overengineered and too complex. This is especially a concern with regard to the (currently) very short deadline of 2 years until launch. Therefore, it is recommended that the architecture is revised during development and that a priority list of the services, modules, and features should be created. The parts of the architecture that are not implemented in time can then be implemented for future missions if needed.

Lastly, **a self-contained development environment has been created** using Docker, greatly reducing the effort required to install the necessary toolchain and configure debugging support. This development environment will make it much easier for both new and old members to get involved and to contribute to the project, and will also reduce the time wasted troubleshooting development-related issues, as all members will use the same environment. A documentation generation system that will be used to generate searchable system-level and software-level documentation has also been set up for the project.

References

Burns, A. and Wellings, A. (2009), *Real-time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*, International computer science series, Addison-Wesley.

California Polytechnic State University (2020), 'CubeSat Design Specification'.
URL: *https://static1.squarespace.com/static/5418c831e4b0fa4ecac1bacd/t/62193b7fc9e72e0053f00910/1645820809779/CDS+REV14_1+2022-02-09.pdf*

CCSDS (2020), CCSDS 133.0-B-2 - Space Packet Protocol, Recommended Standard, The Consultative Committee for Space Data Systems.
URL: *https://public.ccsds.org/Pubs/133x0b2e1.pdf*

Docker, Inc. (2023*a*), 'About storage drivers'.
URL: *https://docs.docker.com/storage/storagedriver/*

Docker, Inc. (2023*b*), 'Docker overview'.
URL: *https://docs.docker.com/get-started/overview/*

ECSS (2008), ECSS-E-ST-70-11C – Space segment operability, Standard, European Cooperation for Space Standardization.
URL: *https://ecss.nl/standard/ecss-e-st-70-11c-space-segment-operability/*

ECSS (2016*a*), ECSS-E-ST-70-41C - Telemetry and telecommand packet utilization, Standard, European Cooperation for Space Standardization.
URL: *https://ecss.nl/standard/ecss-e-st-70-41c-space-engineering-telemetry-and-telecomm*

ECSS (2016*b*), *ECSS-Q-HB-60-02A – Techniques for radiation effects mitigation in ASICs and FPGAs handbook*.
URL: *https://ecss.nl/hbstms/ecss-q-hb-60-02a-techniques-for-radiation-effects-mitigatio*

Erl, T. (2017), *Service-Oriented Architecture: Analysis and Design for Services and Microservices*, Prentice Hall Service Technology Series, 2 edn, Prentice Hall, Boston.

freeRTOS (2023), 'What is an RTOS?'.
URL: *https://www.freertos.org/about-RTOS.html*

GomSpace (2023*a*), 'CSP Documentation: The Protocol Stack'.
URL: *https://github.com/libcsp/libcsp/blob/develop/doc/protocolstack.md*

GomSpace (2023*b*), 'The Cubesat Space Protocol README'.
URL: *https://github.com/libcsp/libcsp*

Ingeno, J. (2018), *Software Architect's Handbook: Become a Successful Software Architect by Implementing Effective Architecture Concepts*, Packt Publishing, Limited, Birmingham.

Jones, H. W. (2021), Going beyond reliability to robustness and resilience in space life support systems, 50th International Conference on Environmental Systems.
URL: *https://ttu-ir.tdl.org/handle/2346/87122*

Jones, M. (2022), 'Breathe's documentation'.
URL: *https://breathe.readthedocs.io/en/latest/*

Martin, R. C. (2018), *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Robert C. Martin series, Prentice Hall, Boston.

McConnell, S. (2004), *Code complete*, 2nd ed. edn, Microsoft Press, Redmond, Wash.

MCUboot (2023*a*), 'Bootloader design'.
URL: *https://docs.mcuboot.com/design.html*

MCUboot (2023*b*), 'Building and using MCUboot with Zephyr'.
URL: *https://docs.mcuboot.com/readme-zephyr.html*

MCUboot (2023*c*), 'Mcuboot readme'.
URL: *https://github.com/mcu-tools/mcuboot*

Ninja-build (2023), 'Ninja'.
URL: *https://ninja-build.org/*

OpenOCD (2023), 'What is OpenOCD?'.
URL: *https://openocd.org/doc/html/About.html*

Orbit NTNU (2023*a*), *BioSat Mission Background and Objectives - BIOSAT-MAR-001*. Internal document, access can be granted on request.

Orbit NTNU (2023*b*), *Concept of Operations - Project BioSat – BIOSAT-CONOPS-002*. Internal document, access can be granted on request.

Orbit NTNU (2023*c*), *FramSat OBC requirements*. Internal document, access can be granted on request.

Orbit NTNU (2023*d*), *OBC microcontroller tradeoff*. Internal document, access can be granted on request.

Orbit NTNU (2023*e*), *OBC Operating system trade-off*. Internal document, access can be granted on request.

QEMU (2023), 'About QEMU'.
URL: *https://www.qemu.org/docs/master/about/index.html*

Sphinx (2023), 'Getting Started'.
URL: *https://www.sphinx-doc.org/en/master/usage/quickstart.html*

Sphinx-contrib (2023), ‘PlantUML for Sphinx README’.
URL: *https://github.com/sphinx-contrib/plantuml*

STMicroelectronics (2023), ‘STM32 Nucleo-144 development board with STM32H7A3ZI MCU, SMPS, supports Arduino, ST Zio and morpho connectivity’.
URL: *https://www.st.com/en/evaluation-tools/nucleo-h7a3zi-q.html*

Wenker, R., Legendre, C., Ferraguto, M., Tipaldi, M., Wortmann, A., Moellmann, C. and Rosskamp, D. (2017), On-board software architecture in mtg satellite, pp. 318–323.
URL: *https://www.researchgate.net/publication/318893422_On-board_software_architecture_in_MTG_satellite*

Zephyr Project (2023*a*), ‘Build System (CMake)’.
URL: *https://docs.zephyrproject.org/latest/build/cmake/index.html*

Zephyr Project (2023*b*), ‘Configuration System (Kconfig)’.
URL: *https://docs.zephyrproject.org/latest/build/kconfig/index.html*

Zephyr Project (2023*c*), ‘Device Driver Model’.
URL: *https://docs.zephyrproject.org/latest/kernel/drivers/index.html*

Zephyr Project (2023*d*), ‘Devicetree versus Kconfig’.
URL: *https://docs.zephyrproject.org/latest/build/dts/dt-vs-kconfig.html*

Zephyr Project (2023*e*), ‘Introduction’.
URL: *https://docs.zephyrproject.org/latest/introduction/index.html*

Zephyr Project (2023*f*), ‘Introduction to devicetree: Input and output files’.
URL: *https://docs.zephyrproject.org/latest/build/dts/intro-input-output.html*

Zephyr Project (2023*g*), ‘Introduction to devicetree: Scope and purpose’.
URL: *https://docs.zephyrproject.org/latest/build/dts/intro-scope-purpose.html*

Zephyr Project (2023*h*), ‘POSIX Support’.
URL: *https://docs.zephyrproject.org/latest/services/portability/posix.html*

Zephyr Project (2023*i*), ‘Symmetric Multiprocessing’.
URL: *https://docs.zephyrproject.org/latest/kernel/services/smp/smp.html*

Zephyr Project (2023*j*), ‘Sysbuild (System build)’.
URL: *https://docs.zephyrproject.org/latest/build/sysbuild/index.html*

Zephyr Project (2023*k*), ‘User Mode Overview’.
URL: *https://docs.zephyrproject.org/latest/kernel/usermode/overview.html*

Zephyr Project (2023*l*), ‘West (Zephyr’s meta-tool)’.
URL: *https://docs.zephyrproject.org/latest/develop/west/index.html*

Zephyr Project (2023*m*), 'Zephyr SDK'.
URL: *https://docs.zephyrproject.org/latest/develop/toolchains/zephyr_sdk.html*

Appendix A

BioSat Requirements Specification

This appendix contains lists of requirements, none of which are officially decided upon but that has been used as assumptions when designing the software architecture. Since no requirement naming standard has been decided on as of the writing of the thesis, the requirement names are only placeholders.

A.1 Functional Requirements

These lists are by no means extensive and should not be interpreted as a comprehensive list of functional requirements, but rather as representation of the primary functional requirements that should be included for any implementation.

A.1.1 General Functional Requirements

These requirements apply to any subsystem.

Table A.1: Functional requirements for a generic subsystem.

BIOSAT-SUB-CMD-001	The subsystem shall be directly commandable from the ground segment.
BIOSAT-SUB-CMD-002	The subsystem shall provide the ability to respond with an acceptance report and an execution status report in response to a request to one of the subsystem's subservices.
BIOSAT-SUB-COMM-001	The subsystem shall be able to communicate with any other subsystem.
BIOSAT-SUB-COMM-002	The subsystem should contain a file transfer client and service.
BIOSAT-SUB-FDIR-001	The subsystem shall be able to provide internal parameters and configurations on request, for diagnostic purposes.
BIOSAT-SUB-FDIR-003	The subsystem shall periodically kick a watchdog timer in order to signal that it is alive. Failing to kick the watchdogtimer will trigger some form of recovery action, such as a reboot.
BIOSAT-SUB-INFO-001	The subsystem shall keep a log of all significant events and actions occurring in the subsystem if it provides a file transfer service. If not, it should utilize the logging service of another subsystem (typically the CDHS).

A.1.2 CDHS Functional Requirements

These requirements apply only to the CDHS.

Table A.2: Functional requirements for the CDHS.

BIOSAT-CDHS-TELEM-001	The CDHS shall provide a service for packaging and storing timestamped telemetry packets.
BIOSAT-CDHS-TELEM-002	The CDHS shall provide storage for mission-payloads and other files.
BIOSAT-CDHS-TELEM-003	The CDHS shall periodically create telemetry packets with housekeeping data (engineering data related to the internal state of the satellite, such as battery voltage, temperature, amount of available memory, etc.).
BIOSAT-CDHS-CMD-001	The CDHS shall be able to store received commands for execution at a later point in time.
BIOSAT-CDHS-CMD-002	The CDHS shall provide the capability to execute a list of commands or actions specified in a script.
BIOSAT-CDHS-TIME-001	The CDHS shall keep an on-board reference time that can be synchronized with an on-ground reference time.
BIOSAT-CDHS-TIME-002	The CDHS shall distribute the on-board reference time to the other subsystems.
BIOSAT-CDHS-FDIR-001	The CDHS shall monitor important on-board parameters and generate events if a monitored parameter is outside of the configurable thresholds.
BIOSAT-CDHS-FDIR-002	The CDHS shall have capabilities to autonomously detect, isolate and recover from failures in the satellite system.
BIOSAT-CDHS-UPD-001	The CDHS shall be able to receive and store at least one extra boot image from the ground segment.
BIOSAT-CDHS-UPD-002	The CDHS shall be able to update its software using an uploaded boot image.
BIOSAT-CDHS-AUTO-001	The CDHS shall control the operational mode of the satellite.
BIOSAT-CDHS-AUTO-002	The CDHS shall provide the capability to autonomously execute mission operations (such as mission data generation and mission data storage or processing). The autonomous actions depend on the autnomoy levels and the operational mode of the satellite.

A.2 Non-Functional Requirements

These requirements apply to any subsystem, but may vary in strictness depending on the subsystem in question.

These lists are by no means extensive and should not be interpreted as a comprehensive list of non-functional requirements. The requirements that include a specific value threshold must be redefined for each subsystem.

A.2.1 Maintainability

Table A.3: Maintainability requirements.

BIOSAT-SUB-MAI-001	The subsystem shall have a Mean Time To Recovery (MTTR) of maximum X days. X depends on the subsystem, a more essential subsystem such as CDHS may have a stricter requirement than e.g. a payload.
BIOSAT-SUB-MAI-002	The subsystem shall have a clearly defined interface that only exposes the information needed to utilize its services.
BIOSAT-SUB-MAI-003	Each module (service, subservice, library, driver, etc.) in the subsystem shall have a clearly defined interface that only exposes the information needed to utilize its functionality.
BIOSAT-SUB-MAI-004	Each module in the subsystem shall be designed to have high internal cohesion and loose external coupling.
BIOSAT-SUB-MAI-005	Each module in the subsystem shall be documented. This applies to both the interface and to internal/private functionality.
BIOSAT-SUB-MAI-006	The software implementation of the subsystem shall follow the Orbit Code Standard.

A.2.2 Reliability

Table A.4: Reliability requirements.

BIOSAT-SUB-REL-001	The execution of a lower-priority request should not affect the timeliness of higher-priority requests.
BIOSAT-SUB-REL-002	The subsystem's Mean Time To Failure (MTTF) should be lower than X, where X depends on the subsystem in question.
BIOSAT-SUB-REL-003	Requests shall be executed in a timely manner.
BIOSAT-SUB-REL-004	No single command executed at the wrong time or in the wrong configuration should result in the loss of the mission. This can be ensured by implementing all critical commands as a two-part command, e.g. enable/execute.
BIOSAT-SUB-REL-005	The bootloader shall be protected from accidental writes.

A.2.3 Testability

Table A.5: Testability requirements.

BIOSAT-SUB-TEST-001	The subsystem shall be testable independently from the other subsystems.
BIOSAT-SUB-TEST-002	Each module of the subsystem shall be testable using unit-tests that can be run independently from other modules, except for critical dependencies.

A.2.4 Reusability

Table A.6: Reusability requirements.

BIOSAT-SUB-REUS-001	The subsystem's software should be clearly separated into mission-specific and mission-independent modules.

A.2.5 Portability

Table A.7: Portability requirements.

BIOSAT-SUB-PORT-001	The software should be able to be compiled for several CPU architectures.
BIOSAT-SUB-PORT-002	The software should be able to run on different hardware configurations.
BIOSAT-SUB-PORT-003	The software should be able to be compiled for and run on different operating systems, with degraded performance if required.

A.2.6 Efficiency

Table A.8: Efficiency requirements.

BIOSAT-SUB-EFF-001	Dynamic memory allocation shall not be used.
BIOSAT-SUB-EFF-002	The subsystem should support a low-power mode.

Appendix B

Development Environment Files

The following files were developed to create a self-contained development environment that can be used when developing on-board software for the BioSat satellite.

B.1 Dockerfile

```
# Development image meant for developing the on-board software of CubeSats
# running the Zephyr RTOS

FROM ubuntu:22.04

ARG USER_NAME="dev"
ARG HOME_PATH="/home/${USER_NAME}"
ENV ZEPHYR_VERSION="3.2.0"
ENV ZEPHYR_SDK_VERSION="0.15.2"

# User and group ID, used to run container as non-root user
ARG UID=1000
ARG GID=${UID}

# Specify shell used during Docker image build
SHELL ["/bin/bash", "-c"]

# Set non-interactive frontend for apt-get to skip any user confirmations
ENV DEBIAN_FRONTEND=noninteractive

# Set time zone
ENV TZ="Europe/Oslo"
RUN ln -fns /usr/share/zoneinfo/$TZ /etc/localtime && \
    echo $TZ > /etc/timezone

# Install dependencies
RUN apt-get update -y && \
    apt-get upgrade -y && \
    apt-get install --no-install-recommends -y \
        git \
        cmake \
        ninja-build \
        gperf \
        ccache \
        dfu-util \
        device-tree-compiler \
        wget \
        python3-dev \
        python3-pip \
        python3-setuptools \
        python3-tk \
        python3-wheel \
        xz-utils \
        file \
        make \
        gcc \
        gcc-multilib \
        g++-multilib \
        libsdl2-dev \
        libmagic1 \
        openssh-client \
        vim \
        locales \
        picocom \
        bash-completion \
        sudo \
        udev \
        default-jre \
        graphviz \
        plantuml \
        latexmk \
        texlive \
        texlive-formats-extra

# Clean up stale packages
RUN apt-get clean -y && \
apt-get autoremove --purge -y && \
    rm -rf /var/lib/apt/lists/*

# Fix locale issues
RUN locale-gen en_US.UTF-8
ENV LANG en_US.UTF-8
ENV LANGUAGE en_US:en
ENV LC_ALL en_US.UTF-8

# Install West and python dependencies
RUN pip3 install -U west && \
    pip3 install -r https://raw.githubusercontent.com/zephyrproject-rtos/zephyr/zephyr-v${ZEPHYR_VERSION}/script
    pip3 install sphinxcontrib-plantuml && \
    pip3 check

# Create '${USER_NAME}' user and group
RUN groupadd -g ${GID} -o ${USER_NAME} && \
    # User is added to the 'plugdev' group to enable flashing as a non-root user
    # using udev rules
    useradd -u ${UID} -d ${HOME_PATH} -m -s /bin/bash -g ${USER_NAME} -G plugdev ${USER_NAME} && \
    # Copy .dotfiles such as .bashrc to get a nice prompt
    cp -a /etc/skel/. ${HOME_PATH} && \
    # Fix ownership issues with VS Code
    chown -R ${USER_NAME}:${USER_NAME} ${HOME_PATH} && \
    # Enable user to use 'sudo' without password
    printf "%s" "${USER_NAME} ALL = NOPASSWD: ALL" > /etc/sudoers.d/${USER_NAME} && \
    chmod 0440 /etc/sudoers.d/${USER_NAME}

# Download Zephyr SDK
# TODO: Find out what parts of the Zephyr SDK is needed for biosat and download
# only that. This will hopefully reduce image size significantly.
WORKDIR ${HOME_PATH}
RUN wget https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v${ZEPHYR_SDK_VERSION}/zephyr-sdk-${ZEPH
    wget -O - https://github.com/zephyrproject-rtos/sdk-ng/releases/download/v${ZEPHYR_SDK_VERSION}/sha256.sum |
    tar xvf zephyr-sdk-${ZEPHYR_SDK_VERSION}_linux-x86_64.tar.gz && \
    rm zephyr-sdk-${ZEPHYR_SDK_VERSION}_linux-x86_64.tar.gz

# Run the Zephyr SDK bundle setup script and copy the udev rules
WORKDIR ${HOME_PATH}/zephyr-sdk-${ZEPHYR_SDK_VERSION}
RUN ./setup.sh -c -h && \
    cp ${HOME_PATH}/zephyr-sdk-${ZEPHYR_SDK_VERSION}/sysroots/x86_64-pokysdk-linux/usr/share/openocd/contrib/60-

# Get bash completion script for West and source completion scripts
RUN wget https://raw.githubusercontent.com/zephyrproject-rtos/zephyr/zephyr-v${ZEPHYR_VERSION}/scripts/west_comm
        -O /etc/bash_completion.d/west-completion.bash

# Set up SSH
RUN mkdir -p ${HOME_PATH}/.ssh && \
    ln -s /run/secrets/orbit-gitlab-key ${HOME_PATH}/.ssh/orbit-gitlab-key && \
    printf "%s\n\t%s\n\t%s\n\t%s\n" \
        "Host gitlab.orbitntnu.com" \
        "HostName gitlab.orbitntnu.com" \
        "User git" \
        "IdentityFile ${HOME_PATH}/.ssh/orbit-gitlab-key" \
        > ${HOME_PATH}/.ssh/config && \
    ssh-keyscan -H -T 60 gitlab.orbitntnu.com >> ${HOME_PATH}/.ssh/known_hosts

# Source the Zephyr Environment Script
RUN echo "source /biosatproject/zephyr/zephyr-env.sh" \
        >> ${HOME_PATH}/.bashrc

USER ${USER_NAME}
```

B.2 Docker Compose File

```
version: '3.8'
services:
  dev:
    # Build image using the Dockerfile in this directory
    build: .

    environment:
      - ZEPHYR_VERSION=3.2.0
      - ZEPHYR_SDK_VERSION=0.15.2
      - BOARD=nucleo_h7a3zi_q
      - TERM=xterm-256color

    # Set hostname for cleaner prompt
    hostname: biosat

    # Set name of image built by Dockerfile
    image: biosat-dev-image

    container_name: biosat-dev-container

    # Sets initial directory if using container without VS Code
    working_dir: /biosatproject/biosat

    # Run in privileged mode to give access to devices. Enables flashing
    # from the container.
    # TODO: Find a method that doesn't give access to all devices on host,
    # as this may be a security issue. For more info, see
    # https://stackoverflow.com/questions/24225647/docker-a-way-to-give-access-to-a-host-usb-or-serial-device
    privileged: true

    # Add private GitLab SSH key as a secret
    secrets:
      - orbit-gitlab-key

    volumes:
      # Mount the current folder to /biosat in the container, keeping changes
      # made while working inside the container
      - type: bind
        source: ..
        target: /biosatproject
      # Create a volume for USB devices. Needed to register device in container
      # after unplugging/replugging device.
      - /dev/bus/usb/:/dev/bus/usb/

    # Script that will be run on container start
    entrypoint: ["/biosatproject/biosat/scripts/entrypoint.sh"]

secrets:
  orbit-gitlab-key:
    # Must point to an existing key on the host machine
    file: ~/.ssh/orbit-gitlab-key
```

B.3 VS Code Dev Container File

The devcontainer.json file used in the development environment.

```
{
// Name of the Dev Container.
"name": "BioSat Development",

// Docker Compose file that will be used
"dockerComposeFile": [
"../docker-compose.yml"
],

// The service in the docker-compose.yml file that will be run.
"service": "dev",

// The path in the container VS Code should open by default when connected.
"workspaceFolder": "/biosatproject/biosat",

"customizations": {
"vscode": {
// VS Code extensions that will be automatically installed in the container
"extensions": [
"ms-vscode.cpptools-extension-pack",
"xaver.clang-format",
"cschlosser.doxdocgen",
"trond-snekvik.simple-rst",
"trond-snekvik.devicetree",
"nordic-semiconductor.nrf-kconfig",
"streetsidesoftware.code-spell-checker",
"jebbs.plantuml",

// Arm debug support
"mcu-debug.debug-tracker-vscode",
"marus25.cortex-debug",
"mcu-debug.memory-view",
"mcu-debug.rtos-views"
]
}
},
}
```

B.4 VS Code Debugging Support

B.4.1 launch.json

File used to define the debug task in VS Code.

```
{
    // Use IntelliSense to learn about possible attributes.
    // Hover to view descriptions of existing attributes.
    // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
    "version": "0.2.0",
    "configurations": [
        {
            // Configuration options
            "name": "Nucleo STM32H7A3ZI",
            "type": "cortex-debug",
            "request": "launch",
            "device": "STM32H7A3ZIQ",
            "preLaunchTask": "West: build debug",

            // Debug options
            "cwd": "${workspaceFolder}",
            "executable": "./build_debug/zephyr/zephyr.elf",
            "runToEntryPoint": "main",

            // Toolchain options
            "servertype": "openocd",
            "toolchainPrefix": "arm-zephyr-eabi",
            "armToolchainPath": "${userHome}/zephyr-sdk-${env:ZEPHYR_SDK_VERSION}/arm-zephyr-eabi/bin",
            "serverpath": "${userHome}/zephyr-sdk-${env:ZEPHYR_SDK_VERSION}/sysroots/x86_64-pokysdk-linux/usr/bi
            "configFiles": ["/biosatproject/zephyr/boards/arm/nucleo_h7a3zi_q/support/openocd.cfg"],
            "svdFile": "/biosatproject/biosat/svd/STM32H7A3x.svd",
        }
    ]
}
```

B.4.2 debug.conf

Debug configuration that can be used to enable useful features when debugging a Zephur applications.

```
# This is a Kconfig fragment which can be used to enable debug-related options
# in the application.

# Compiler
CONFIG_DEBUG_OPTIMIZATIONS=y

# console
CONFIG_CONSOLE=y

# UART console
CONFIG_SERIAL=y
CONFIG_UART_CONSOLE=y

# logging
CONFIG_LOG=y
CONFIG_APP_LOG_LEVEL_DBG=y
```

B.4.3 tasks.json

File used to define build tasks in VS Code.

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "West: build",
            "type": "shell",
            "command": "west",
            "args": [
                "build"
            ],
            "group": {
                "kind": "build",
                "isDefault": true
            },
            "presentation": {
                "echo": true,
                "reveal": "always",
                "focus": true,
                "panel": "shared",
                "showReuseMessage": true,
                "clear": true
            },
            "problemMatcher": [
                "$gcc"
            ],
            "detail": "Build the project using West"
        },
        {
            "label": "West: build pristine",
            "type": "shell",
            "command": "west",
            "args": [
                "build",
                "--pristine"
            ],
            "group": {
                "kind": "build"
            },
            "presentation": {
                "echo": true,
                "reveal": "always",
                "focus": true,
                "panel": "shared",
                "showReuseMessage": true,
                "clear": true
            },
            "problemMatcher": [
                "$gcc"
            ],
            "detail": "Do a pristine build of the project using West"
        },
        {
            "label": "West: build debug",
            "type": "shell",
            "command": "west",
            "args": [
                "build",
                "-d",
                "build_debug",
                "--",
                "-DOVERLAY_CONFIG=debug.conf"
            ],
            "group": {
                "kind": "build",
            },
            "presentation": {
                "echo": true,
                "reveal": "always",
                "focus": true,
                "panel": "shared",
                "showReuseMessage": true,
                "clear": true
            },
            "problemMatcher": [
                "$gcc"
            ],
            "detail": "Build the project in debug configuration using West"
        },
        {
            "label": "West: build debug pristine",
            "type": "shell",
            "command": "west",
            "args": [
                "build",
                "-d",
                "build_debug",
                "--pristine",
                "--",
                "-DOVERLAY_CONFIG=debug.conf"
            ],
            "group": {
                "kind": "build"
            },
            "presentation": {
                "echo": true,
                "reveal": "always",
                "focus": true,
                "panel": "shared",
                "showReuseMessage": true,
                "clear": true
            },
            "problemMatcher": [
                "$gcc"
            ],
            "detail": "Do a pristine build of the project in debug configuration using West"
        },
        {
            "label": "West: build with mcuboot",
            "type": "shell",
            "command": "west",
            "args": [
                "build",
                "--",
                "-DOVERLAY_CONFIG=mcuboot.conf"
            ],
            "group": {
                "kind": "build",
            },
            "presentation": {
                "echo": true,
                "reveal": "always",
                "focus": true,
                "panel": "shared",
                "showReuseMessage": true,
                "clear": true
            },
            "problemMatcher": [
                "$gcc"
            ],
            "detail": "Build MCUboot and the project using West"
        },
        {
            "label": "West: build debug with mcuboot",
            "type": "shell",
            "command": "west",
            "args": [
                "build",
                "-d",
                "build_debug",
                "--",
                "-DOVERLAY_CONFIG=debug.conf mcuboot.conf"
            ],
            "group": {
                "kind": "build",
            },
            "presentation": {
                "echo": true,
                "reveal": "always",
                "focus": true,
                "panel": "shared",
                "showReuseMessage": true,
                "clear": true
            },
            "problemMatcher": [
                "$gcc"
            ],
            "detail": "Build MCUboot and the project in debug configuration using West"
        }
    ]
}
```