Embedded software development methods for mission critical satellite operations

Henry Sanmark

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo, May 26, 2017

Thesis supervisor:

Prof. Ville Kyrki

Thesis advisor:

M.Sc. (Tech.) Ignacio Chechile

Aalto University
School of Electrical
Engineering

AALTO UNIVERSITY
SCHOOL OF ELECTRICAL ENGINEERING

ABSTRACT OF THE
MASTER'S THESIS

Author: Henry Sanmark

Title: Embedded software development methods for mission critical satellite operations

Date: May 26, 2017 Language: English Number of pages: 8+87

Department of Electrical Engineering and Automation

Professorship: Automation technology

Supervisor: Prof. Ville Kyrki

Advisor: M.Sc. (Tech.) Ignacio Chechile

Mission critical flight software for spacecraft is essential for spacecraft functionality. It controls all subsystems such as payloads, and multiple standards and conventions have been developed to provide reliability and robustness for the software. The most significant feature of developing spacecraft software is the codesigning of the hardware and software, where robustness must be taken into consideration.
There are multiple traditional methods for developing mission critical embedded software. These methods struggle from slow development velocity and do not favor possible changing requirements. New methods for developing software have emerged in latest decades, but they are designed for different fields of industry and not for mission critical embedded software.
This thesis studies how to improve embedded mission critical software development in a small company with limited resources and a tight development schedule. The thesis was developed as an assignment for Finland based NewSpace company ICEYE.
In this thesis, the traditional spacecraft software development conventions are provided as a background information. Then, different modern software development techniques are presented. After that, this information is applied to ICEYE software development, where final analysis and developed systems are presented. The focus of development phases is divided into four subsections, which are development tools, hardware and software codesign, continuous integration and continuous delivery, and lastly, testing. Finally, the conclusions and integration of these four analyses are provided, with suggestions for future work. The results propose agile acceptance test-driven approaches with the trackable requirements from system level with justified use of development tools. The final system should be tested in real simulated functional scenarios, which occur during the actual mission. This thesis also presents developed configurations, testing environment and analysis of selected tools.

Keywords: spacecraft, satellite, flight software, NewSpace, mission-critical, embedded system, acceptance test-driven development, continuous integration, Robot Framework

AALTO-YLIOPISTO
SÄHKÖTEKNIIKAN KORKEAKOULU

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä: Henry Sanmark

Työn nimi: Sulautettujen tehtäväkriittisten satelliittijärjestelmien ohjelmistokehitysmenetelmät

Päivämäärä: May 26, 2017 Kieli: Englanti Sivumäärä: 8+87

Sähkötekniikan ja automaation laitos

Professuuri: Automaatiotekniikka

Työn valvoja: Prof. Ville Kyrki

Työn ohjaaja: DI Ignacio Chechile

Tehtäväkriittinen lento-ohjelmisto on oleellinen osa avaruusaluksen toimintaa. Se hallitsee kaikkia alijärjestelmiä kuten hyötykuormia, ja useat erilaiset standardit sekä käytännöt pyrkivät varmistamaan ohjelmiston eheyden sekä luotettavuuden. Eräs huomionarvoisin ominaisuus tämänkaltaisen ohjelmiston kehittämisessä on ohjelmiston ja laitteiston yhtäaikainen kehittäminen, missä eheys pitää ottaa huomioon.
Tällaisen ohjelmiston kehittämisestä on olemassa useita perinteisiä tapoja. Nämä kuitenkin kärsivät hitaasta kehitysnopeudesta eivätkä taivu mahdollisiin muuttuviin vaatimuksiin. Viimeisinä vuosikymmeninä on kehitetty uudempia kehittyneitä tapoja kirjoittaa ohjelmistoja, mutta nämä on kehitetty pääasiassa muiden teollisuudenalojen tarpeisiin eivätkä täten toimi itsessään tehtäväkriittisille sulautetuille järjestelmille.
Tämä diplomityö tutkii tehtäväkriittisen sulautetun järjestelmän ohjelmiston kehittämisen tehostamista, jonka lisäksi etsittiin ratkaisuja, millä tavalla kyseisiä toimia voidaan käyttää hyväksi rajoitetuilla resursseilla sekä nopealla kehitysaikataululla. Diplomityö tehtiin toimeksiantona ICEYE-nimiselle suomalaiselle NewSpace-yritykselle.
Tässä diplomityössä esitellään aluksi perinteiset ohjelmistokehitysmenetelmät avaruusohjelmistolle. Myös erilaiset modernit ohjelmistokehitystekniikat esitellään. Näiden pohjalta analysoidaan ICEYEn nykyistä ohjelmistokehitystä ja diplomityön aikana kehitetyt järjestelmät esitellään. Analysoinnin pääpaino on jaettu neljälle eri osa-alueelle, jotka ovat kehitystyökalut, laitteiston ja ohjelmiston yhdenaikainen kehitys, jatkuva integraatio sekä jatkuva julkaisu ja lopuksi testausmenetelmät. Lopuksi nämä kaikki kiedotaan yhteen ja tehdään lopulliset päätelmät, sekä esitellään tulevaisuuden kehitysehdotukset. Tulokset esittävät, että kehityksessä pitäisi suosia hyväksymistestauksen kautta perittyjä ketteriä kehitysperiaatteita, jossa järjestelmätason vaatimukset voidaan jäljittää koko kehityksen aikana. Käytössä olevien työkalujen täytyy olla oikeutettuja. Lopullinen järjestelmä testataan simuloiduilla tehtävänaikaisilla tapahtumilla. Tämä diplomityö myös esittää kehitetyt asetustiedostot, testausympäristön sekä työkalujen analysoinnin.

Avainsanat: satelliitti, lento-ohjelmisto, NewSpace, tehtäväkriittisyys, sulautettu järjestelmä, hyväksymistestaus, jatkuva integraatio, Robot Framework

Preface

My first touch to spacecraft development was in 2013 when I started working with first Finnish nanosatellite Aalto-1. Later, its software development was my topic for Bachelor's Thesis. It did not stop there, and later I was involved in ICEYE project in 2014 - way before it was a real company. Two years later, I am finishing my Master's Thesis in the same project as I used to work with before. The quick impulsive thought about involving myself in the spacecraft development became my career and passion, while my expertise on embedded software development has increased significantly. Thing which I could never have imagined when I started studying in Aalto University. This is more than I could have ever hoped for.

I thank Ville Kyrki for supervising my Master's Thesis and giving valuable feedback during the development. The greatest thank you goes to Ignacio Chechile, who was my advisor and provided expert level feedback for my thesis and for the whole ICEYE software development. Other thanks go to the rest of ICEYE team who have been supportive during these months and also for Aalto Space Crew for giving me the opportunity to create something where the sky is not the limit.

Even if spacecraft projects have provided me amazing experiences, the humble thanks go to the amazing other people around me during these years. These years spent as an active teekkari in Aalto Student Culture have given me an opportunity to attend and create awesome small and big events, meet new people, create phenomenal projects around the whole of Finland, such as sending over 1,500 fellow students to visit schools, and of course break a world record by building the world's largest sauna. Especially people behind these groups: IE13, IE14, TJ14, ASH13, AS, RBH, VT, NC, !111111, Joutomiehet, Tempaus2016, Tempausesikunta, Tempaus Tapahtumatoimikunta, AYY, Polyteekkarimuseo and all others which I forgot, deserve their own thanks. Because of you, I have learned more than I could have ever asked for and found my own passion and ambition to do great things. Great thanks also go to my family and closest friends for their support during these years. It has been an indispensable resource at all times.

My years in Otaniemi have been a huge adventure. I consider these adventures as a massive book of stories, and this Master's Thesis is the last Chapter of that book. It is time to put this piece of work in the Library next to the Shield in the wall and start exploring the next episode of this adventure.

Libertas. Technologitas. Naturalitas.

In the cradle of technology, Otaniemi, May 26, 2017

Henry J. O. Sanmark

Contents

Abbrevations and acronyms

AFDIR	Advanced fault detection, isolation and recovery
AR	Acceptance Review
ARM	Advanced RISC Machine
ATDD	Acceptance test-driven development
AWS	Amazon Web Services
BIT	Built-in-testing
CD	Continuous Delivery
CDR	Critical Design Review
CI	Continuous Integration
CMOS	Complementary metal–oxide–semiconductor
COTS	Commercial off-the-shelf
DevOps	Development and Operations
ECSS	European Cooperation for Space Standardization
EMC	Electromagnetic compatability
ESA	European Space Agency
ETB	Electrical Test Bench
FDIR	Fault detection, isolation and recovery
FIR	Fuzzy Inductive Reasoning
FTA	Fault Tree Analysis
GOCE	Gravity Field and Steady-State Ocean Circulation Explorer
GS	Ground Segment
IC	Integrated circuit
IDE	Integrated Development Environment
I/O	Input and output
LEO	Low Earth Orbit (altitude 300-900 km)
MPI	Message Passing Interface
MVP	Minimum valiable product
NASA	National Aeronatics and Space Administation
NewSpace	Startup company in space industry
OBC	On-board computer
OBCSW	On-board computer software
ORR	Operational Readiness Review
OS	Operating system
OSRA-P	On-board Software Reference Architecture for Payload
PA	Project Assurance
PC	Personal computer
PCB	Printed Circuit Board
PCM	Power Conditioning Module
PDR	Preliminary Design Review
POC	Proof-of-concept
PUS	Packet utilization standard
QA	Quality Assurance
QR	Qualification Review

RISC	Reduced Instruction Set Computing
RPU	Radar Processing Unit
RTL	Register Transfer Level
RTOS	Real-time operating system
SaaS	Software as a Service
SAR	Synthetic Aperture Radar
SDL	Specification and Description Language
SDRAM	Synchronous dynamic random-access memory
SoC	Single System on Chip
SPL	Secondary Program Loader
SRAM	Static random-access memory
SRR	System Requirements Review
SSC	Swedish Space Corporation
SSH	Secure Shell
TC	Telecommand
TLM	Transaction Level Modeling
TM	Telemetry
UML	Unified Modeling Language
VCS	Version Control System

1 Introduction

On-board computer software is in a central role in satellite mission operations. It is responsible for all satellite functionality, such as instruments which are uniquely designed for the mission. In addition, OBCSW (On-board Computer Software) is designed for robust and reliable embedded hardware which must tolerate space environment such as heat and radiation. Alongside OBCSW, other spacecraft subsystems contain software which follows the same principles. To ensure reliable design, different organizations such as ESA (European Space Agency) have developed multiple standards called ECSS (European Cooperation for Space Standardization) which define project management, project assurance, engineering and sustainability. [1]

In these methods, large software components are developed based on hardware iterations and their detailed requirements lists. Even though the main factor of software development in embedded satellite applications is mission-criticalness, traditional methods using strictly defined standards might lack in development velocity and do not allow modern software development practices known in common industry. For example, after passing a milestone, it might be impossible to introduce changes to the project. [2] In other industries such as in information technology, flexible practises for software development have been developed. However, these practices are not directly applicable for embedded software development because of hardware codesign and especially when mission criticalness must be considered [3] even though ECSS standards do not directly restrict agile methods [4].

After small and agile companies started to emerge in space industry during 2000s, it has been crucial to determine more flexible methods to develop space systems with tight schedule and limited resources. This space industry relies on nanosatellite and microsatellite technologies. In addition, COTS (commercial off-the-shelf) hardware and open source software are widely used. These space industry startups are called "NewSpace" startups, and they usually consist of 20-30 employees, meaning that every single step cannot be verified because of limited resources. Additional challenges to NewSpace companies are small teams, rapidly evolving hardware iterations, tight schedules and low budgets. Above all, the focus of these companies is completely business driven when time to market defines if a company continues existing or not. This differentiates NewSpace projects from traditional spacecraft projects and university CubeSat projects. Therefore, the focus is based on largest issues and a MVP (minimum viable product) must be designed before running out of resources. These companies need to find a compromise to develop software with modern agile methods but also consider strict requirements based on space environment and satellite industry.

This Master's Thesis was written as an assignment for a Finnish NewSpace company ICEYE Ltd. The purpose of the thesis is to improve ICEYE software development processes for embedded satellite flight software and also research how these methods can be applied in general in other projects. In this thesis, different aspects of software development practices are analysed and verified to work in space industry - and especially in startups. [5] [6]

1.1 Problem statement and research questions

The aim of the thesis is to find solutions to use modern software development methods in embedded spacecraft flight software. This includes both on-board computer software and subsystem software in satellites. Spacecraft software development differs from traditional embedded software development because of mission criticalness and robust requirements [5] [7, pp. 269-275] [8]. Because software development has no unambiguous guideline, four main problems are disclosed and overall analysis is based on their results. Therefore, developed solutions and analysis are possible utilize in a larger scope. In addition, all these methods are applied in a startup company which has small teams and a fast development cycle. Therefore, the problem statement can be phrased as:

> *"How to improve embedded software development in mission critical satellite systems, when robustness and fast development cycle in a small "NewSpace" company must be taken into consideration?"*

The research problem will be answered from the perspective of the following research questions:

1 What development tools can be used and are they blocking or improving the development?

2 How can hardware design be efficiently implemented into software?

3 How can continuous integration be used in mission critical embedded software development?

4 How can system level tests be implemented for flight software?

This thesis presents all developed configurations and environments which improve the software development process for flight software. Even if software exists also, for example, in ground stations, these are out of the scope of the thesis. In addition, research analysis is also presented in theoretical form and giving possible future implementation suggestions. All these points are integrated and analysed from the perspective of how they improve the overall software development process.

1.2 Structure of the thesis

The work has been divided into six chapters, where the first one is this introduction Chapter. The second Chapter presents the overall background and common guidelines about spacecraft software and hardware. Chapter 2.3 also covers information about software development, and introduces development trends in other fields of industry. The third Chapter presents the ICEYE project, where target platform is briefly described in order to understand thesis results and tool arrangements. Design, implementation and analysis of the proposed software development methods are described in Chapter 4. In Chapter 5, overall results are discussed and proposals introduced. Finally, the conclusions are presented in Chapter 6.

2 Background

This chapter provides background understanding and motivation of this thesis. It introduces the fundamental requirements for traditional spacecraft projects as a basis for understanding the challenges which NewSpace companies face in their design. First, traditional spacecraft flight software and its functionality in general are presented to give the reader an overview of the technical challenges and possible future approaches. This covers how requirements are defined in software, what are the main mission-specific functionalities and how software architecture is designed. Secondly, the basic principles of mission-criticalness is introduced. Last, software development models are presented and compared to each other. These software development models are first presented in general and then applied in mission critical embedded systems for satellites.

2.1 Spacecraft flight software and hardware

An OBC (on-board computer) is a vital part of spacecraft functionality. According to ESA (European Space Agency) [9], OBC is referred to as the computer of the satellite's avionic sub-system; the unit where the OBCSW run. OBCSW itself controls everything on the spacecraft from subinstruments to data handling and command transmitting [10]. Therefore, when referring to OBC it must be clarified that *"OBC"* refers to hardware of the computer, and *"OBCSW"* to the software side of the OBC. Besides OBCSW, software is also found in payloads and therefore the overall spacecraft flight software consists of multiple different software components. The whole system is not only limited to flight software. Alongside flight software, whole system consists of both spacecraft software and ground segment (GS) software, which provide the overall operational spacecraft. The abstract illustration of the whole system is presented in Figure 1.

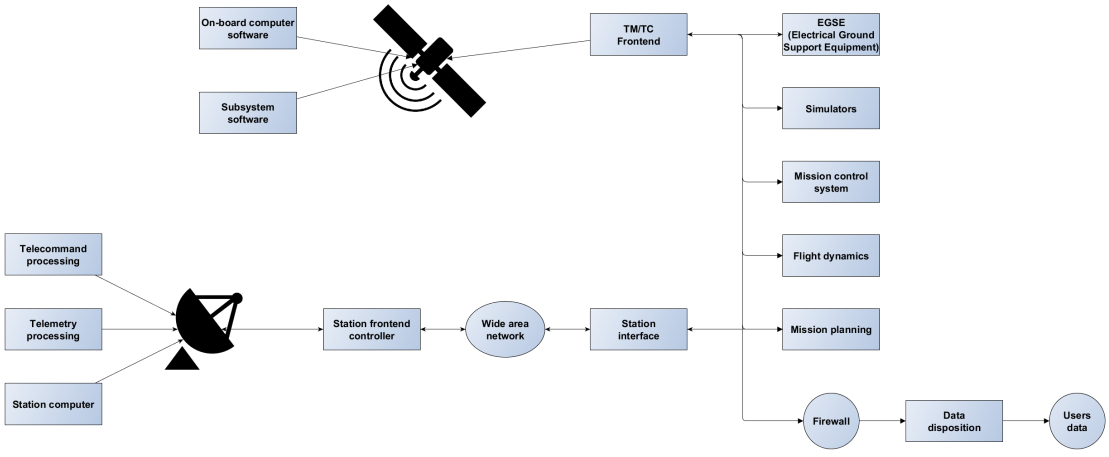

Figure 1: *The typical space system, with the emphasis on software elements. The system consist of the spacecraft and ground segment, which itself contains multiple subsystems. Based on the illustration developed by Jones et al. [11]*

Eickhoff emphasizes [12, pp. 5] several aspects which distinguish OBC from standard industry embedded controllers or automotive controllers. These are:

- Significant failure robustness only achievable by internal redundancy.
- EMC (Electromagnetic Compatibility) to the space environment conditions.
- Radiation robustness against high energetic particles.
- Supports a large number of different types of interfaces which are at least partly redudant.

These requirements cannot be achieved by standard approaches in IC (Integrated Circuit) designs which are used in regular PC microprocessors. Therefore space application processors require a lower circuit integration density and further manufacturing specifics. This results in lower processor clock frequencies.

Comparable principles are also defined for the software. Therefore, OBCSW needs to be:

- Real-time control software
- Allows both interactive spacecraft remote control and autonomous control
- Architecture is usually service based, covering several I/O (Input/Output) levels:
  - Data I/O handlers and data bus protocols
  - Control routines for payloads, ADCS (Attitude Determination and Control System), thermal and power subsystems
  - FDIR (Failure Detection, Isolation and Recovery) routines

When analyzing OBCSW functionality, it can be determined that it has multiple operational purposes. The major functions of OBCSW are [12, pp. 89-90]:

- Telecommand processing from commands from GS
- Telemetry generation for status monitoring by GS
- Attiture and orbit control
- Power control
- Thermal control
- Payload instruments control
- System status monitoring
- Fault detection, isolation and recovery

From the spacecraft mission point of view, OBC and OBCSW have to be detailed concerning the following additional requirements [12, pp. 5]:

- Command and control of the payload
- Operations concept must be based on international spacecraft uplink/downlink data transmission standards

- Telecommand (TC) and telemetry (TM) packet management must comply to the customer's baseline such as PUS (Packet utilization standard) by Europan Space Agency [13]
- Spacecraft mission operations concept has to take into account ground station visibility, ground station network, link budgets and operational timeline from ground commands
- Control all nominal platform and payload functions from the ground
- Control all FDIR procedures from the ground
- Support OBCSW updates, patches and mission extension functions

Spacecraft flight software is not limited only to OBCSW. Every subsystem, such as payloads contains its own hardware and software, which are developed alongside OBCSW. While OBC controls different subsystems with different interfaces, the subsystem embedded software must also cover some of the requirements presented above while they are focusing on a single dedicated operation. Therefore, the hardware and software designs of subsystems follow their operational requirements. [14] The simplified picture of the whole spacecraft hardware and software structure is presented in Figure 2.

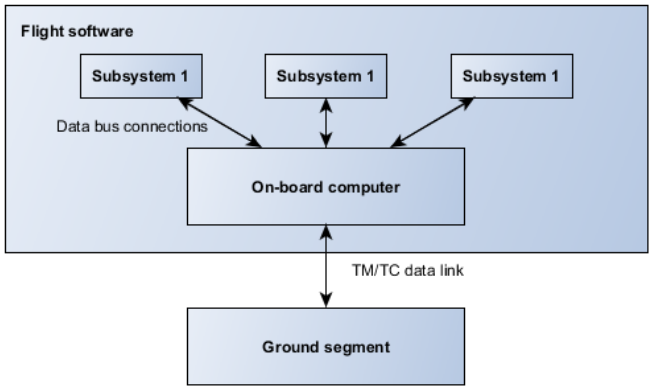

Figure 2: *A simplified structure of spacecraft flight software. Every payload has its own software and hardware module, which are controlled by OBC and OBCSW. Data link is established to communicate with the ground station.*

As for OBC hardware, RISC (Reduced Instruction Set Computing) chips are dominant in today's designs [12, pp. 42]. ARM (Advanced RISC Machine) based solutions have become more common because of energy and cost efficiency as well as high performance [15], and multiple modern nano- and microsatellites especially uses ARM architecture [16] [17] [18] [19].

Besides the process architecture, OBC hardware includes memory - SRAM/SDRAM and PROM/EEPROM - mass storage unit, I/O ports, data buses, debug interfaces, transponder interfaces, power supplies, interrupt controllers, thermal control units, timers, fault tolerance devices, such as for CMOS latchup, and reconfiguration units [12, pp. 126] [20, pp. 636-639]. All of these modules must fulfill the preliminary requirements of OBC

hardware which are presented in the list above. Figure 3 shows an example how OBC is connected to overall avionics system, where interfaces are connected to I/O board and finally to different subsystems. The Figure presents what different bus types are possible to use and what subsystems, such as GPS and PCM, are connected to OBC.

2.1.1 Software architecture

As stated earlier, OBCSW has a central role in satellite functionality. There are two alternative architectural approaches, *static architecture* and *dynamic architecture*. [10] In static architecture, software is built upon on different functional modules and layers where all information is shared with well-designed interfaces, such as an application layer and operating system as well as a drivers layer with cyclic or time driven executions. In dynamic architecture, the software is assumed to be real-time. Thus, all operations are based on tasks, threads and interrupts - meaning that the dynamic architecture is completely event driven. [12, pp. 99, 120]

One of the major modules of the software architecture is OBCSW kernel, which can be defined as the bottom-most layer of OBCSW architecture. Kernel is the core component of controlling all fundamental processes of the OBCSW functionality, such as controlling the startup sequence after booting, initializing all flight software components, interlinking software modules and controlling the scheduler. [12, pp. 117] It is also a core component of all OS (Operating System) functionalities which exist in modern OSs, such as system level logging and health checks. For example, many RTOS and GNU/Linux based systems are used to provide core kernel functionalities, but also other approaches are possible, such as simple "software in the loop" which is not, however, encouraged [21, pp. 580]. There are commercial, open source and in-house developed solutions available. However, when referring to OBCSW kernel, it may refer to implementation design of the software instead of the OS kernel, such as Linux kernel [12, pp. 117].

For payload software, there is no specific model for defining architecture because of different purposes and operational requirements. However, SSF (Space Systems Finland) and ESC (Evolving Systems Consulting) are currently aiming to define a new model for payload software architecture named OSRA-P (On-board Software Reference Architecture for Payload). [22] OSRA-P can be defined as static or dynamic architecture, but focuses on defining component models for developing payload software, identifying different layers and their interfaces. The example of static three-layer software architecture for OSRA-P is presented in Figure 4.

2.1.2 Fault detection, isolation and recovery

A key feature of OBCSW functionality is FDIR (Fault Detection, Isolation and Recovery). This means that in case of anomalies, with automatic or manual procedures, the spacecraft can detect system failures, locate their origin based on gathered information, and last, do recovery maneuvers to possibly fix the problem or restore functionality with an alternative plan [23].

NASA (National Aeronatics and Space Administration) has declared basic procedures for implementing FDIR techniques [23] which are:

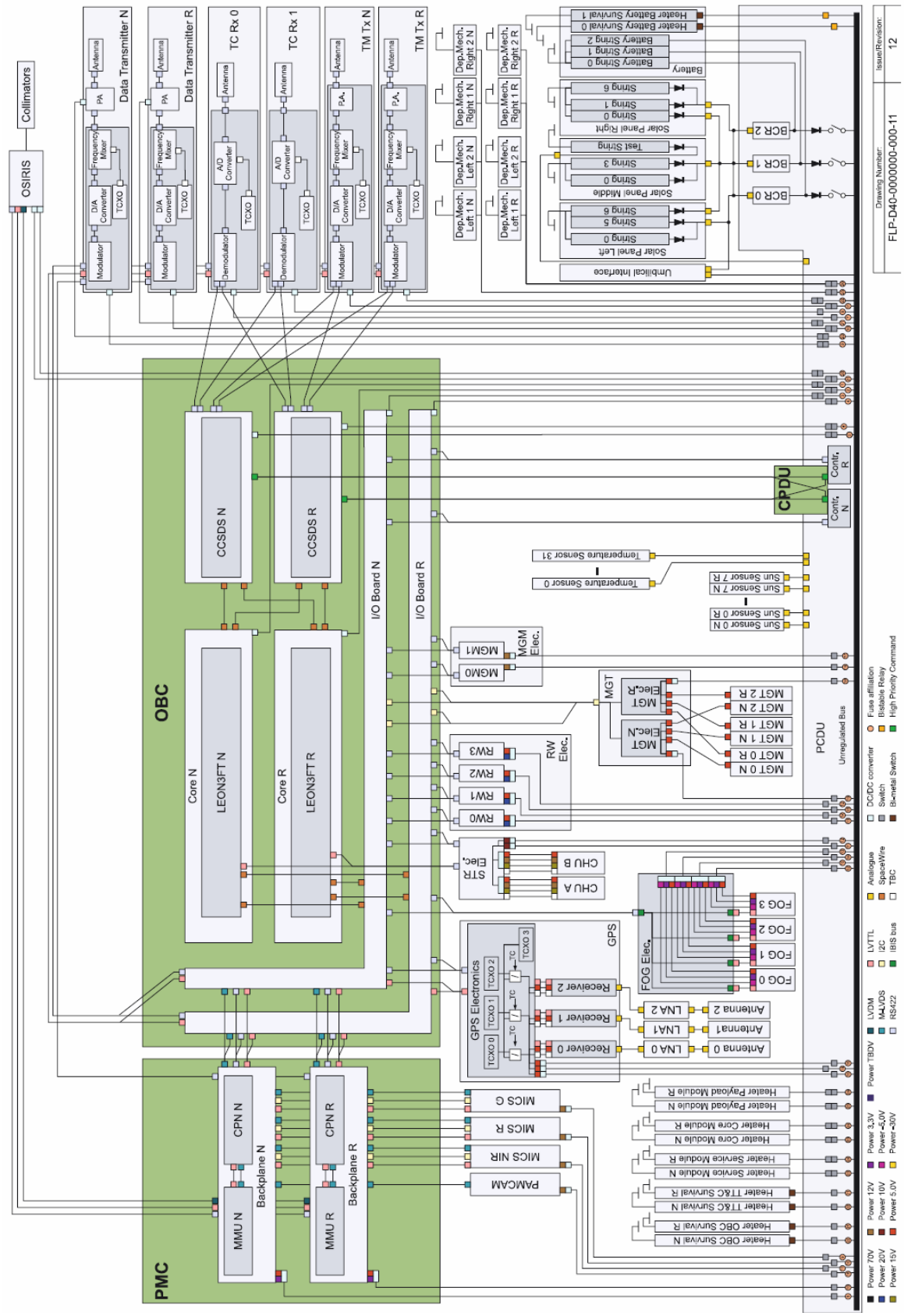

Figure 3: *An example of OBC as an electrical block diagram. The Figure shows how OBC is connected to different payloads with its interfaces. Extracted from Eickhoff's illustration. [12, pp. 53]*

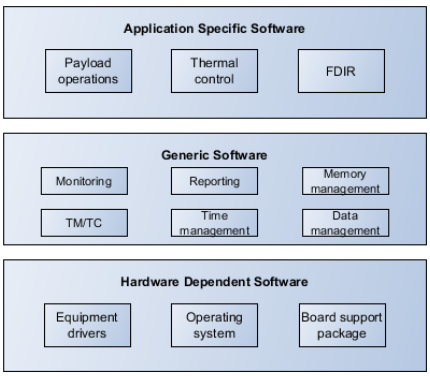

Figure 4: *The static three-level software architecture for OSRA-P model. The similar approach can also be detected in OBCSW static architecture, where functional blocks are defined in different layers. [22]*

- Fault detection techniques
  - Automatic or manual detection of failure using testing equipment
  - BIT (built-in-testing), where faults are detected automatically with self-diagnostics and health checks from sensor data
  - Using BIT with test points and self-test meters
  - Keeping BIT as simple as possible but effective in needs of operational needs
  - Using a watchdog to detect system out-of-tolerance conditions
  - Using an active BIT, where a test pattern is written to unit and compared to expected result
  - Using centralized BIT or decentralized BIT, where either a central unit supervises all modules or every single module has its own fault monitoring
- Fault isolation techniques
  - Manually visualizing the burned-out components
  - Using BIT, automatically detect the origin from detected error signals
  - Using an IC to use a boundary scan, where IC is divided into regions which are accessible with scan operations
  - Boundary scan can be defined as unit testing, which can be also applied as a part of system level testing
- Recovery techniques
  - Identifying a recovery technique, either complete recovery with redundant system components, functional recovery with alternative paths or degraded functional recovery.
  - With complete recovery, a backup unit is brought up

- With functional recovery, an alternative unit taken into control in system providing limited features with limited resources
- With degraded recovery, it must be determined which components must be turned off without losing the ability to control the spacecraft until possible repairs are made, such as safe mode

For FDIR implementation, there are two possible main concepts. First concept is "hard wired" FDIR, where a lower level module triggers a fixed function of the next FDIR level in case of anomalies. The final level of the FDIR procedure may go up to the highest level, and therefore, result in using a safe mode. In this case, fixed functions have to be declared and OBCSW patched for every level. In addition, appropriate status telemetry must be generated to report the GS what happened. The second possible concept is more flexible, where ESA PUS (Packet Utilization Standard) is used with service concepts. In this case, anomalies detected by PUS triggers an event which handles appropriate actions and reports proper telemetry messages. This method is more complex to implement and test, but offers reconfiguration during flight and more coherent structure. [12, pp. 117]

FDIR has also been studied recently and more advanced solutions have been developed that aim to improve FDIR procedures. One proposed approach is SMART-FDIR, which uses AI (Artificial Intelligence) to improve real-time performance, robust architecture, auto-learning and decision making capabilities for every FDIR iteration. As an example, SMART-FDIR is used in GOCE (Gravity Field and Steady-State Ocean Circulation Explorer) Satellite System. In this system, fault detection uses FIR (Fuzzy Inductive Reasoning) and dynamic I/O-model. Fault isolation uses possibilistic logic theory for system behavioral model, and recovery uses logical and structural model reconfiguration and newly activated behaviours. [24] Another proposed approach is AFDIR (Advanced Fault Detection, Isolation and Recovery). This adds more features to traditional FDIR with Kalman filtering, weighted sum-squared residual test, generalized likelihood test, random sample consensus method, and various spacecraft simulations for computing "expected" values. It uses two integrative diagnosis methods: probabilistic reasoning, using Bayesian networks, and model-based diagnosis, using causal networks. [25] Studies based on both of these methods have provided significant advances to conventional FDIR procedures, such as dynamic system modeling, false alarm management, better AI-based failure analysis and deduce underlying failures from multiple, superficial indications or symptoms [24] [25] [26].

In NewSpace companies, the similar advances may not be possible due to limited resources. Therefore there are even simpler ways to implement FDIR, which can be a simple loop which checks different conditions or their combinations, and then acts with corresponding actions. ICEYE software developers stated that the main approach if FDIR design is to keep the FDIR as simple as possible, because huge combinations FDIR conditions may lead to too intensive testing which is considered as a risk in terms of schedule.

2.1.3 Software requirements and standards

To create reliable OBCSW, requirements for defining its functionality must be defined. Requirements might come from physical requirements considering the space environment,

functional requirements and project management perspective. Eickhoff describes [12, pp. 130] that OBCSW requirements should define its architectural structure, functional requirements with function hierarchies, algorithm performance requirements, data handling and operational requirements, scheduling and timing precision, FDIR, development processes and verification/validation requirements. In these requirements, it must be defined precisely how software must operate during its execution.

For example, it is possible to construct a "Function Tree" which describes all functions and its relations to spacecraft operations and data handling. Based on Function Tree, it is easy to define a table of contents for functional requirements of OBCSW alongside non-functional requirements. An example of Function Tree and of a requirements tree based on it is shown in Figure 5.

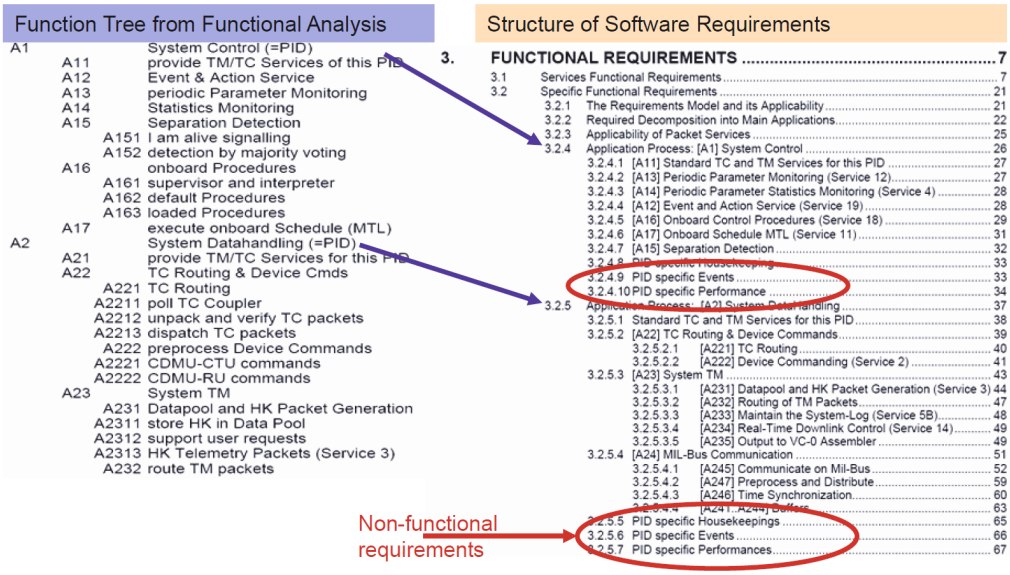

Figure 5: *An extracted example of Function Tree and a requirements Table based on it. Function Tree is generated from functional analysis which is then converted into software requirements. [12, pp. 133]*

There are numerous ways to model requirements and their relationships with different systems. One of the commonly used ways is UML (Unified Modeling Language). UML is used to visualize the design of the system, which can be used for defining how different subsystems and methods interfere with each other and what are their relations. [12, pp. 144] UML is not the only possible notation, and depending on the developer's choice, it is also possible to use, for example, simplified flowcharts or modeling languages other than UML. Evaluating the benefits of different methods is not in the scope of this thesis. The choice of the requirements modeling practice should be left to developers which themselves know the best practice in their working environment.

OBCSW requirements do not define how features are implemented, but more how it must operate. Requirements are later verified by other methods, which are [12, pp. 135]

- Requirements analysis

Table 1: *ECSS-Q-ST80C criticality levels and their definitions*

	SW Criticality Level Definition
Level A	Software that if not executed, or if not correctly executed, or whose anomalous behavior can cause or contribute to a system failure resulting in: *Catastrophic consequences (Loss of mission)*
Level B	Software that if not executed, or if not correctly executed, or whose anomalous behavior can cause or contribute to a system failure resulting in: *Critical consequences (Endangering mission)*
Level C	Software that if not executed, or if not correctly executed, or whose anomalous behavior can cause or contribute to a system failure resulting in: *Major consequences*
Level D	Software that if not executed, or if not correctly executed, or whose anomalous behavior can cause or contribute to a system failure resulting in: *Minor or Negligible consequences*

- Review of design
- Code inspection
- On-board software tests

The most distinguishing attribute which has an influence in requirements definition is the space environment. This is especially significant in spacecraft mechanics and hardware design, but from the software point of view, it must be also considered. The dominant factor of space environment restriction is the Sun, which provides all the heat in solar system and consists of 99.9% of total mass. Solar activity, such as solar waves and sunspots, causes significant radiation for spacecraft which might result to reduced lifetime. Even in LEO (low Earth orbit) spacecraft, the effects due to radiation of solar activity are significant. For example, particles hitting the sensitive part of an IC, it may cause SEUs (single-event upsets), causing changes in logical state of the device or even SEL (singe-event latch-up) which with CMOS material can result in a burn-out. To avoid resulting in completely catastrophic functionality of spacecraft, software must be designed to handle these anomalies. [21, pp. 17-30] Alongside error-correcting codes and other self-maintain procedures, the main principles of FDIR procedures were presented in Chapter 2.1.2.

In terms of software, many requirements can be classified by their criticality. The idea is that different software components are ranked based on their criticality and therefore giving information on how the specific component must be tested. ECSS-Q-ST-80C [27] describes the criticality as shown in Table 1.

Alongide physical requirements, also governmental and commercial standards exist for spacecraft development. These standards are used to provide a common backbone for developing spacecraft and are used for improved development cycles, convergent structures, better combatibility and sharing information about reliable minimal requirements. In addition, they provide guidelines for good design and coding practices, requirements towards extensive testing practices and information for external partners in terms of

validation and verification [12, pp. 166]. In United States, NASA holds most of the governmental standards which cover, for example, project engineering, systems engineering and technical definitions [7, pp. 269]. In Europe, ESA develops ECSS standards for the same use, and they are used to conflate development practises [11]. ECSS provides multiple standards for different purposes and they are divided into different fields of engineering. The main approach for ECSS standards is that they are divided into three different main branches, which are "Management", "Product Assurance" and "Engineering". Inside these three main branches, there are also four levels of standards [4]:

- *Level 0:* Defines the discuss policy, architecture and objectives of ECSS.
- *Level 1:* Describes the strategy within management, product assurance and engineering by highlighting the requirements and interfaces of level 2 elements.
- *Level 2:* Explains objectives and functions of each domain. It is considered as branch-specific level.
- *Level 3:* Lists methods, procedures and tools to achieve the requirement of the level 2 documents. It is also known as technical domain specific level.

From software point of view, a family of ECSS-E-40 standards exist, and they define everything in these three main branches regarding software engineering. Also the family of ECSS-Q-80 considers software [4]. Their main purpose is to define "requirements on those processes broken down into component activities" and "their expected inputs and outputs". ECSS-E-70 processes are defined in Table 2 [4] [11].

Because software development is driven by multiple requirements and standards, the overall activity should be emphasized. Therefore, at project end one must provide a compatibility matrix, stating achieved compliance and which documents, review minutes, product assurance reports prove the compliance [12, pp. 172]. Example of the process is presented in Figure 6.

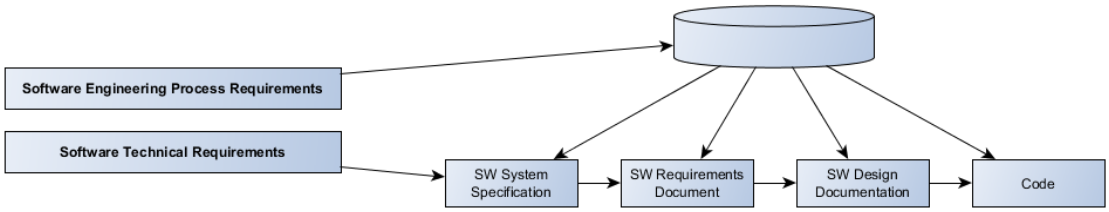

Figure 6: *Traditional software and process requirements, driving development process. The Figure presents how multiple documents define different phases of the software development until the last working piece of code is developed. [12, pp. 172]*

Table 2: *ECSS-E-70 engineering processes and their related reviews*

System engineering	• System requirements engineering • System integration • System validation • Carried out by the customer • Involves SRR (System Requirements Review)
Software Requirements Engineering	• Software-requirements analysis • Software top-level architectural design • Carried out by the supplier • Related to PDR (Preliminary Design Review)
Software Design Engineering	• Designing of software items • Coding and unit testing • Integration • Validation with respect to the technical specification • Carried out by the supplier • Involves CDR (Critical Design Review)
Software Validation and Acceptance	• Validation with respect to requirements baseline – Milestone is QR (Qualification Review) – Carried out in supplier's environment • Software delivery and installation • Software acceptance test – Milestone is AR (Acceptance Review) – Carried out by customer
Software Operations Engineering	• Preparation of software operations procedures • Preparation of plans for operational testing • Software operations proper • User support • Involves ORR (Operational Readiness Review)
Software Maintenance	• Software problems analysis • Software problems correction • Re-acceptance • Software migration • Software retirement

When software development is analysed in terms of QA (Quality Assurance) and PA (Project Assurance), there should be experts available to ensure that software requirements are met throughout the development cycle. These requirements are defined in several points defined below. The role of these experts is to accept or reject design solutions based on risks and lessons learned from other projects. Therefore, experts work as "supervisors" who analyse every single design detail from the early concept to the end of application life. [21, pp. 263, 608] Therefore, the following software process assurance points must be defined:

- Phases
- Phase input(s) and output(s)
- Completion status
- Milestones

- Dependencies
- Responsibilities
- Customer actions in milestones

It can be summed up that software development cycle is significantly based on standardized methods and peer-reviewing by experts, who give significant feedback about current direction of the spacecraft development alongside software development. However, this also emphasizes the strictness of methods which may prevent introducing new possible solutions for spacecraft development and therefore sticking in old conventions.

2.2 Principles of failsafe

All mission-critical or safety-critical systems can be defined as failsafe systems. The basic idea of a failsafe system is that it can handle its anomalies and understand what can possibly fail. In this context, failures mean events which could lead to degenerative or catastrophich consequences. Therefore, these systems are designed so that they respond to possible failures and inherently respond in a way that will cause no or minimal harm to other equipment, environment or people. Alongside spacecraft systems, also medical systems and military systems can be defined as failsafe systems. The difference between mission-critical systems and safety-critical systems is that failures in mission-critical systems can lead to a loss of mission while failures in safety-critical systems can lead, in addition to the loss of mission, to human injuries or environmental damage. Satellites are usually defined as mission-critical systems. [7, pp. 1-7]

2.2.1 Design and coding

To understand the basic concepts of designing a failsafe system, it must be defined what external elements have an influence on system and what internal elements can cause failures. External systems can be classified as space environment and user interaction from ground station. [7, pp. 89-91] However, external elements are strictly defined by standards as defined in Chapter 2.1.3, and human interaction is restricted by GS policies. By following well-defined guidelines sufficient information about design policies is provided. Internal policies refer to system design itself, where complex code structure is less provable and critical systems might interact with less reliable subsystems [7, pp. 89-99]. The same policies apply also here: following standardized or common guidelines helps integrating systems with each other. There are three strategic principles to be followed when designing failsafe systems, which are:

- Readability
- Redundancy
- Provability

Readability refers to code design, which is easy to read and understood by humans. This involves transparent naming of variables, functions and extensive commenting. It must

be clear for developers how to maintain code, and because human readable is also human recoverable, which also promotes failsafe principles. This also indicates a factor that software design is not only limited to one developer, and other team members can verify the software or continue the work. Even if there is no unambiguous way to write code, it is preferable that company-wide coding conventions are defined to promote coherent software structure between modules. ESA has defined some coding conventions, but there are also other possible procedures which can be used [7, pp. 91-93] [28].

The second principle is *redundancy*, where it must be ensured that different software components do not overlap with each other. A practical example is that a new software module developed by another software engineer breaks some functionality in another module which causes anomalies in software execution, such as disabling some self-checks. Interfaces between modules must be defined accurately as defined in Chapter 2.1.3. To improve redundancy, automated system level regression testing can also be used in CI (Continuous Integration). CI and testing procedures are discussed in detail in Chapters 4.5 and 4.6 [7, pp. 94-97] [29].

Redundancy is close to the third principle, *provability*. In small systems, it is easy to verify all units throughout, but when more complex systems are introduced this turns out a significant issue. Provability can be achieved with proper testing practices which include, for example, unit testing, acceptance testing and functional system testing. Besides validation and verification with testing practices, modular software structure and encapsulation also provide provability while improving readability and redundancy. With proper software design and self-check functionality, it can also be ensured that software functions are executed correctly while giving proper logging information, such as warning and error messages. The basic principle in provability is to verify reality, meaning that software must be ensured to work as planned during operation. [7, pp. 97-99] [20, pp. 665-668] [30]

2.3 Software development operations

While earlier chapters discussed spacecraft development in general with respect to software development, this Chapter presents details about actual software development with "classical space" projects and current trends in modern industry.

As stated earlier, milestones are set during the software development. In "classical" flight software development V-model is used, and therefore the milestones of reviews can be presented as in Figure 7 [12, pp. 170].

In this model, the left side presents the project and requirements definition and its phases, the bottom presents the implementation and right side testing and integration. V-model is easy to implement because it provides clear milestones and straightforward steps to implementing software, and counterparts for the validation and verification. Therefore, while developing software on left side of the model, that specific phase is verified and tested at the same level of the right side.

While the V-model is widely used, another traditional development model is called "waterfall model". It can be referred to as a predecessor of V-model. [31] In waterfall model, all development phases follow each other staged from design to final product. The flaw of this model is that sequences must follow each other, and therefore, after some

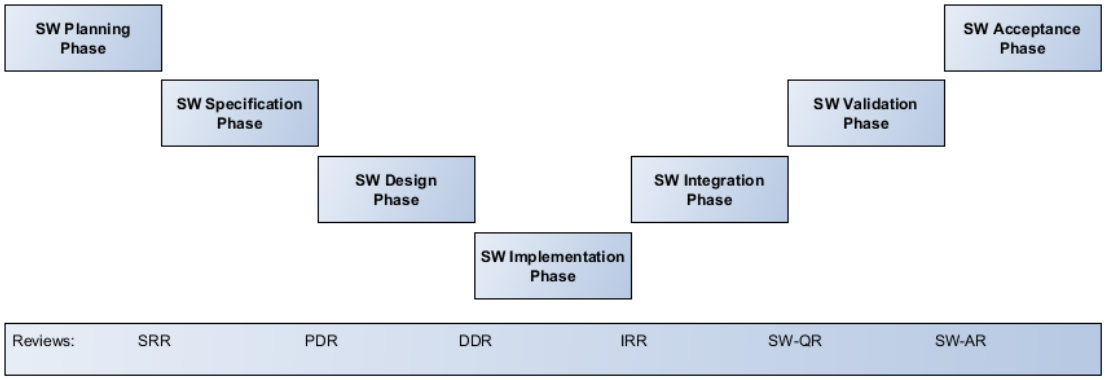

Figure 7: *Software development process and review milestones. The process follows the classical V-model, where left side presents project definition and requirements of the spacecraft and right side the testing and integration. The corresponding review milestones are presented according to development phase. [12, pp. 170]*

sequences it is hard to fix previous phases if problems occur. [20, pp. 662-663] [31]

Even if V-model and waterfall model have been widely used in traditional software development in spacecrafts, they lack aspects related to project management. V-model and waterfall model offer great ways to directly follow ECSS-defined standards and design milestones but they do not support changed requirements during the software development. Even if mission critical software must be well defined with strict requirements considering the space environment, a project may change during the development and even strict requirements may change over time. In addition, especially in longer spacecraft development projects, these models might not respond to changed market or technology advances which are faster than project development itself. Again, V-model and waterfall model also work, and therefore, adopting new methods is a slow process considering the risks. Larman [32] has reviewed several problems related to the waterfall, which can also be seen in the V-model:

- Waterfall works best for projects with little change, little novelty, and low complexity
- Waterfall pushes high-risk and difficult elements to end of the project
- Waterfall aggravates complexity overload
- Waterfall is poorly suited to deal with changing requirements
- Waterfall encourages late integration
- Waterfall produces unreliable up-front schedules and estimates

When analysing these problems and comparing them to previously presented software requirements considering the spacecraft, we can clearly see that especially complexity, changed requirements after reviews, for example, significant changes after Critical Design Review and late integration are problems which do not fullfil the fundamental principles of flight software development. Even though the well-defined strict requirements play a

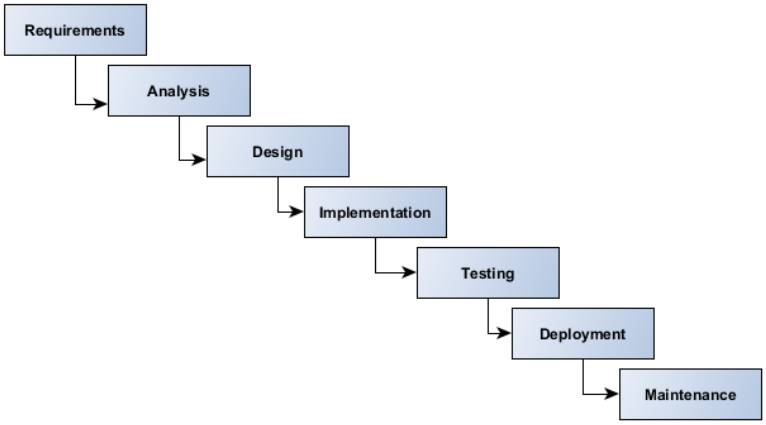

Figure 8: *Waterfall software development model and its phases. In waterfall principle, every phase starts only when the previous phase has ended and this leads to final product.*

huge role, it cannot be assumed that software development follows direct path without obstacles or changes. Therefore, more advanced ways to develop software are required which promotes the role of the developers and constantly living project. However, strict requirements of the spacecraft development must also be held constantly in mind.

2.3.1 From waterfall to agile

Agile is a commonly used umbrella-term for different iterative software development practices, methods and techniques. Two most common practices are Extreme Programming (XP) and Scrum. Agile is often accepted as a better development practice than waterfall [31]. ITEA (The Information Technology for European Advancement program) reported [33] that in industry level, 68 projects provided the following results about the benefits of agile compared to traditional methods:

- Development costs reduced by 70%
- Quality over 3 times better than average
- Customer satisfaction was on average 4.9 in a scale from 0 to 5
- 70% increase of developer satisfaction with process

The basic idea of the agile methods is that software development is divided into multiple self-organizing cross-functional teams, where software is developed within fast iterations where parts are developed as sub-assemblies rather than as a one final product. [34]. This method emphasizes adaptive planning, constant changes, continuous improvement and early delivery. Agile methods are based on four main values of which the "agile manifesto" [34] is written from. These four main values of agile methods are:

Table 3: *Results of SSC's research of combining ECSS standards and agile practices. [4]*

Factor	Requirement	Agile Strategies	SSC Practices
Software Management Process	Identify a suitable software development process and a technical review process	Management models like Scrum, Agile Modeling, DSDM	Develop a Software Development Plan for following Scrum for each project
Software Engineering Process	Define and document engineering process for requirements, architecture, design, coding and testing	Techniques like Test Driven Development (TDD), Continuous Integration, Pair Programming	Sprint planning is done at the start of each sprint and the use TDD for development
Verification and Validation (V&V) Process	Determine the effort required to implement V&V process, Define the process and document it	Agile paradigm forces to have visible goals and frequent reviews	Frequent review meetings in Scrum and TDD ensures process verification and minimizes product defects
Assurance Program Implementation	Develop a comprehensive software product assurance plan	Sprint planning meetings in Scrum	Practices like sprint planning meetings, daily standup meetings and sprint post-mortem analysis helps to assure this implementation
Software Process Assurance	Confirm the suitability of software development process selected according to ECSS-E-40	Suggests daily stand-up meetings, sprint retrospectives in Scrum	Process assurance is achieved through daily standup meetings and frequent sprint retrospectives
Software Product Quality Assurance	Identify the set of requirements to assure the quality of the final software product	Sprint retrospectives in Scrum	Frequent sprint retrospectives are used to identify success factors and failures

- **Individuals and interactions**
  - Very short feedback loop and adaptation cycle
  - Self-organization and motivation are important, as are interactions like co-location and pair programming
- **Working software**
  - Quality focus
  - Working software is more useful and welcome than just presenting documents

to clients in meetings.

- **Customer collaboration**
  - Efficient and face-to-face communication
  - Requirements cannot be fully collected at the beginning of the software development cycle, therefore continuous customer or stakeholder involvement is very important.
- **Responding to change**
  - Iterative, incremental and evolutionary
  - Agile methods are focused on quick responses to change and continuous development.

The main difference between agile methods and waterfall methods are the approach of quality and testing. As seen in waterfall model, testing phase follows the implementation and building whereas in agile, these are considered to be run in the same iteration. Therefore testing is done alongside the whole development phase, not just for the final product. This provides that software pieces can be validated during the whole development cycle and new features can be added with shorter feedback time [1] [34] [35] [36] [37]. Figure 9 presents the main differences between waterfall and agile methods.

The challenges of implementing agile processes in spacecraft design have been discussed. In an industrial case study for SSC (Swedish Space Corporation), it was analysed how agile methods with ECSS standards can be used in practice. The report [4] and its findings are shown in Table 3.

In a report by Nicoll [38], it was shown that when developing safety-critical software which followed EN 50128 standard with safety level 4 (highest), using agile methods provided improved development cycles and verified process. It was also noted, that especially certain elements of agile methods improved safety, which were [31]

1 Test-first development

2 Early incremental production of working code

3 Pair-programming

When it comes to high level formal safety regulations, such as ECSS standards, it was detected that agile methods are completely compatible with them. Therefore, it is encouraged that spacecraft software developers implement agile methods instead of traditional iterative approaches [1] [31] [37].

2.3.2 DevOps for embedded systems

While agile methods are widely used in industry and are proposed for the current approach of software development, even more advanced techniques for software development have emerged which can be described as the next step of agile. One of those techniques is called DevOps (Development and Operations). It has emerged to combine *agile developers* and *agile operations*. [3]

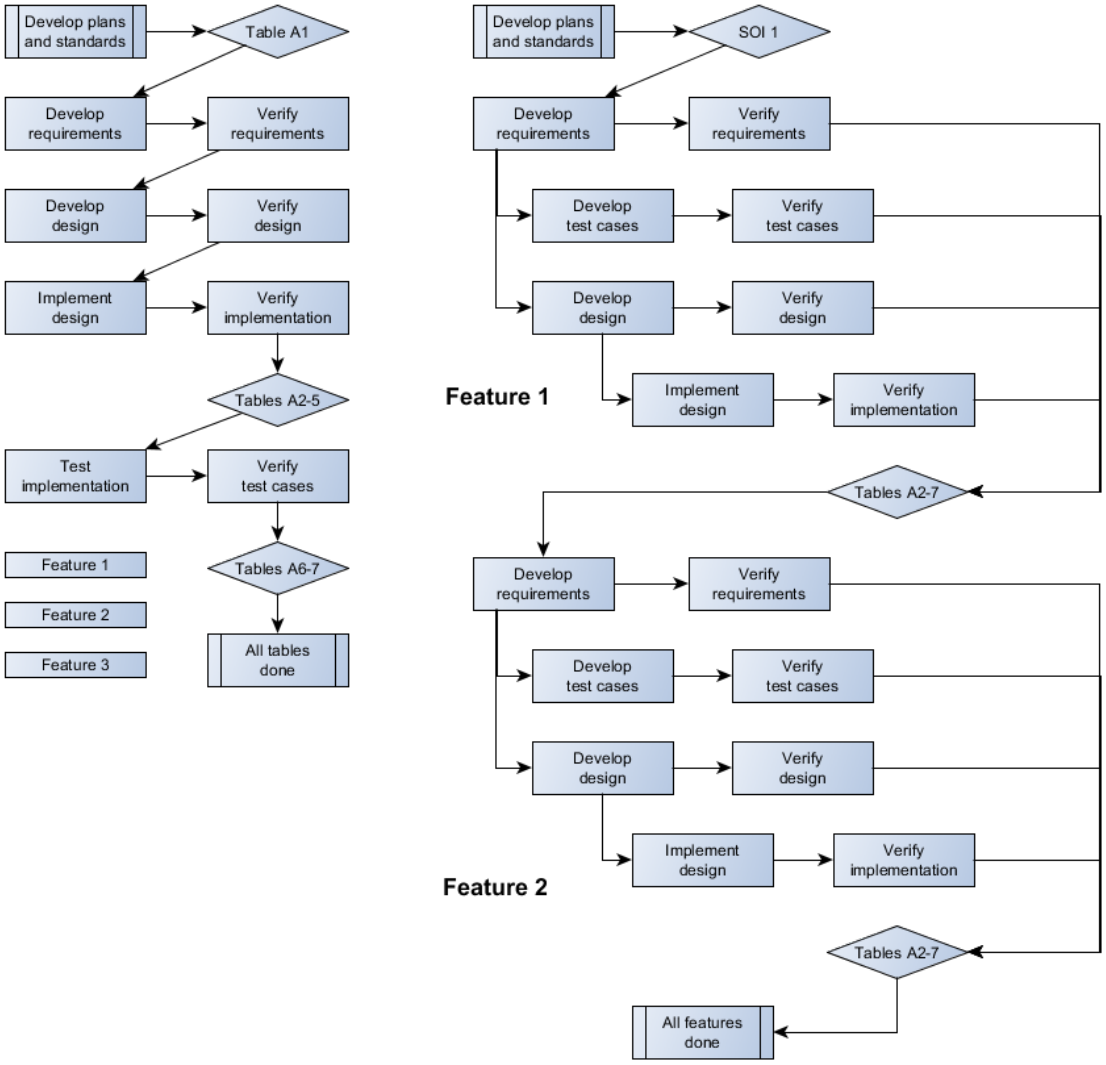

Figure 9: *Differences between waterfall (left) and agile (right) development methods for the similar piece of software, which covers safety level DO-178B. Multiple features have their requirements defined in tables which are implemented into a working code. [31]*

DevOps as a term consists of multiple practices, which emphasize collaboration of developers and operations in each stage of DevOps toolchain [3]. This means that alongside improving the actual "development" process which consists of programming, version control, building and others, the "operations" part focuses on software delivery, configuration, monitoring and other operational methods. This is achieved with multiple practices, such as [39]:

- *Culture:* People over processes and tools. Software is made by and for people.
- *Automation:* Automation is essential for DevOps to gain quick feedback.
- *Measurement:* DevOps finds a specific path to measurement. Quality and shared, or at least aligned, incentives are critical.

Table 4: *DevOps toolchain elements and their descriptions.*

Plan	This stage involves requirements definition of the software, both functional and non-functional requirements alongside project defined requirements, such as release plan. It also defines all metrics related to software production.
Create	Create consists of software programming, building and configuration. It is well tied with version control tools and build tools.
Verify	In this stage, software quality is measured with different testing procedures such as acceptance testing, regression testing and performance testing.
Staging	Involves all activities which are essential before deploying the software. This includes build approvals, package configuration, triggered releases and staging or holding. This phase is also referred to as "preproduction" or "packaging".
Release	The actual software release in production. In this phase, release schedule is defined, fallbacks and recovery are described, and software deployment is executed.
Configure	All configuration activities after the deployment. This requires actions such as infrastructure storage, database and network provisioning and application configuration.
Monitor	The last stage, where feedback from end-user is collected and analysed for further development.

- *Sharing:* Creates a culture where people share ideas, processes, and tools.

The key of implementing DevOps can be expressed with *DevOps toolchain*, which presents the stages of software development lifecycle and it is coordinated with DevOps practices. In this toolchain, DevOps stages are defined and practical actions related to it are defined. The elements of DevOps toolchain are shown in Table 4 and relations between stages in Figure 10.

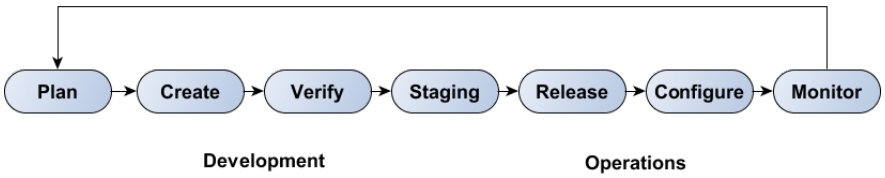

Figure 10: *Phases of DevOps toolchain, which presents the difference between "development" and "operations" and how phases follow each other.*

However, even though DevOps is considered as a new era of software development, it faces several problems considering the embedded systems and especially mission critical software. Especially, when talking about *operations*, it should be kept in mind that DevOps model was built for information technology services, such as web and SaaS (Software as a Service) applications. For example, continuous deployment on embedded mission

critical software differs significantly from deploying web based applications, and therefore, does not facilitate operations in spacecraft development the same way [3]. The main difference is that with embedded systems, software is just one part of the development alongside hardware. Therefore, the presence of hardware introduces challenges for DevOps in embedded systems, especially when development is hardware-driven [40].

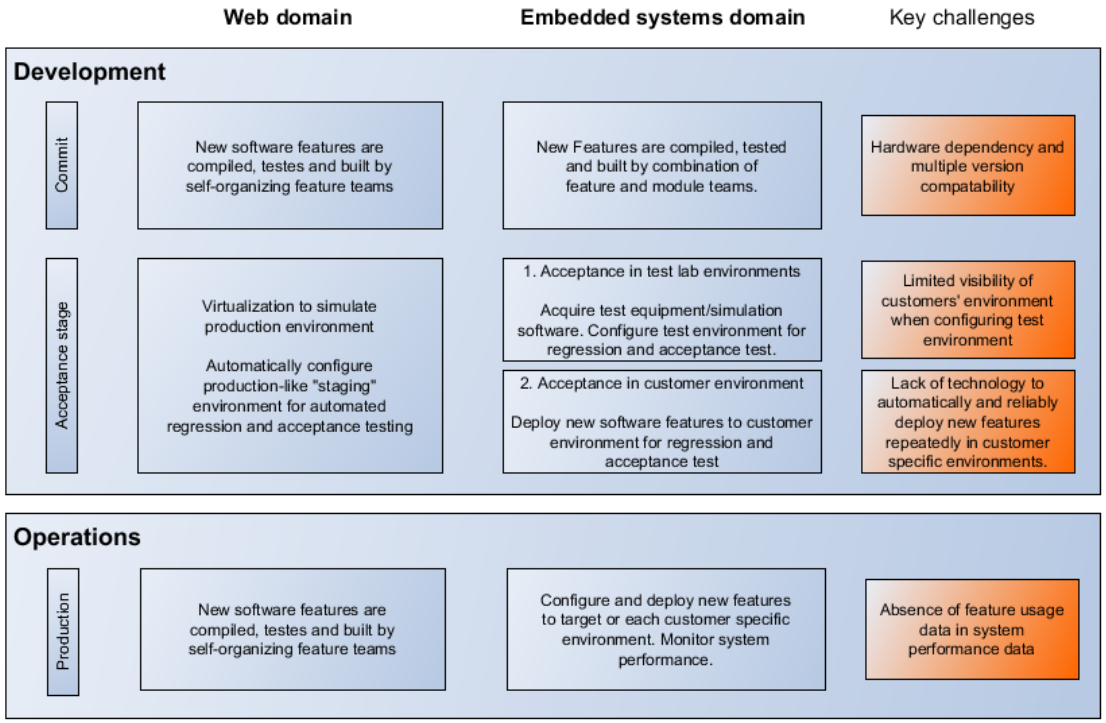

Figure 11: *The main differences between web domain and embedded domain in DevOps procedures. Figure shows the significant features while applying DevOps for different processes and which key challenges emerged. Based on the illustration by Lwakatare et al. [40]*

In a case study performed by Lwakatare et al. [40] it was shown that DevOps in embedded systems faced four major challenges. The first one (1) is hardware dependency and combatibility with multiple versions. The result of this is that these companies have silos in development teams inside different modules. Even though development cycles are short on software side, there are much longer cycles on hardware side, causing challenges. The second (2) issue is limited visibility to customer's production environment. Providing testing environments with required reliably, it cannot be guaranteed that testing environments matches the development environment. Third issue (3) is about scarcity of tools. While there are numerous tools available for SaaS application development, embedded software can be unique and there are no specific tools to automatically deploy required software to production. This is also connected to a previously presented issue that for mission critical software, possibly no reliable software is not available and continuous deployment is not a preferred approach. The last issue (4) is about monitoring system performance data. It has been hard for companies to collect data in post-deployment which

could be used as feedback for product improvement. The main differences between DevOps in web applications and embedded systems regarding these four issues are presented in Figure 11.

However, even if DevOps in embedded domain faces challenges, most of its features can be applied for development life cycle. DevOps practices must be fitted in company's needs and environment, and after realizing challenges, some of the practices can be applied after modifications. When we are considering mission critical software, such as spacecraft software, DevOps must be analysed differently.

For example, NASA Jet Propulsion Laboratory has adopted DevOps practices for their real-time telemetry gathering. They use state-of-the-art DevOps techniques and in contrary to previous researches, they emphasize fast feedback and opinion for failure. Therefore, the slogan for NASA *"Failure is not an option"* has been questioned because fast feedback from mistakes in DevOps is considered a key element [41]. Naturally failure is not an option after flight model has been launched to space, but if stages after *staging* are performed in-house and *customer* is considered as a current hardware design, DevOps can be applied. Therefore, in contrast to web applications, systems are not deployed to production with fast iterations and monitored by customer, but instead, with peer-reviews and simulations with currently codesigned hardware. This naturally requires that sufficient tools, such as CI tools, are available, as presented when describing challenges for DevOps in embedded systems.

Besides technical challenges, DevOps provides improvements in software development and operations when its stages are defined slightly differently. DevOps still contains several challenges which should be taken into account when analysing the whole software development life cycle.

3 ICEYE project

In this Chapter, ICEYE as a company and its satellite project is presented in order to give the reader an understanding of the case study. First, an overview of ICEYE mission is presented as a background information. In the second section, ICEYE software structure is presented, and lastly, the greatest constraints regarding design conventions are briefly discussed.

3.1 Project definition

ICEYE is a Finnish startup satellite company, which aims to provide near-realtime radar images using a constellation of microsatellites. In this context, near-realtime refers to a delay of a few hours in contrast to over 24 hour delay of common image providers. In contrast to traditional optical sensors, ICEYE satellites use SAR (Synthetic Aperture Radar), which allows taking images through cloud cover and at night. [42]

The original idea of the radar was to monitor arctic seas, such as movement of icebergs and icebreakers. However, the scope of the radar has been expanded and currently ICEYE satellites aim to take images for multiple other purposes. These are, for example, monitoring agriculture, logistics, disaster relief and climate changes all over the Earth. [42]

ICEYE started in 2012 when its founders participated in Aalto Ventures program and was later funded to develop radar's technical prototype. The company itself was established in late 2014, and in 2015, a radar prototype was built succesfully. After new fundings, ICEYE has grown and currently consists of around 30 employees and aims to launch its first POC (Proof-of-Concept) satellite in 2017.

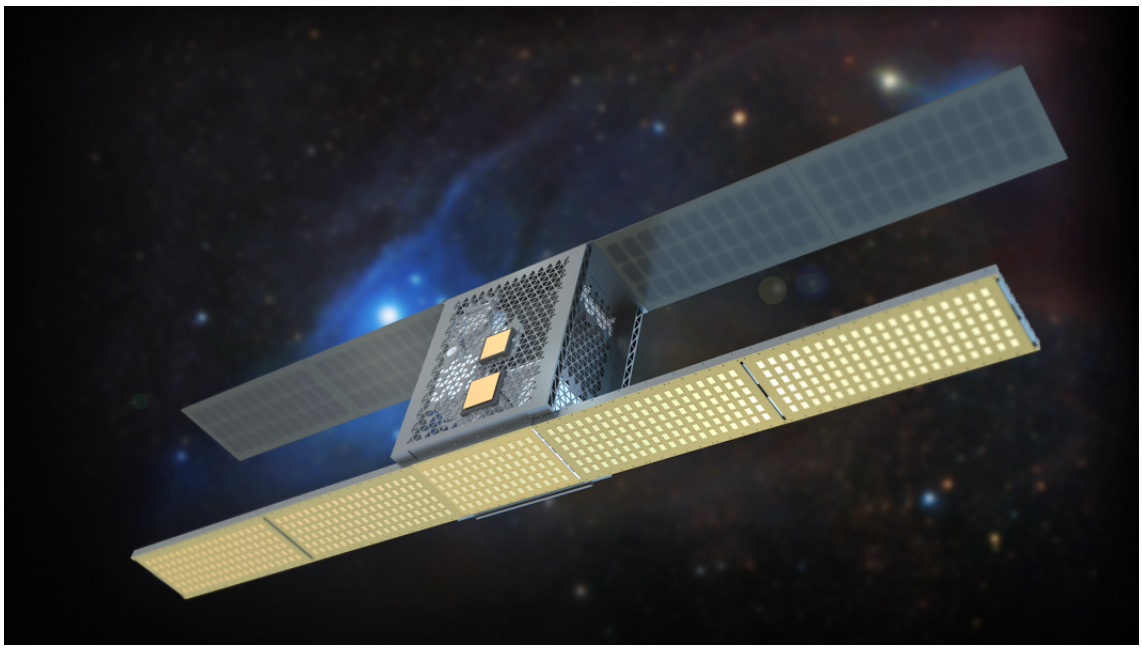

Figure 12: *Concept image of ICEYE satellite.*

Satellites themselves have mass less than 100kg and size around 3.2m x 0.5m x

0.33m. Some of the subsystems are ordered as COTS (commercial-of-the-shelf), but some subsystems are developed in-house. In-house developed subsystems are RPU-PCM (Radar Processing Unit - Power Conditioning Module) box which contains processing board, tranceiver, combiner & divider and CDR lite, OBC box containing OBC with S-band and X-band adapters and X-band antenna. Systems such as radios, ADCS and other are purchased from suppliers. [43]

3.2 Software structure

Flight software is structured the same way as defined in Chapter 2.1. However, the whole flight software is referred here as OBSW (on-board software) and it is divided into OBC software, and all software running on self-designed subsystems called MCU software, referring to "microcontroller software". The basic structure is presented in Figure 13. [44]

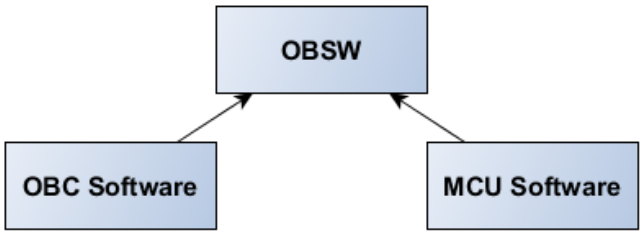

Figure 13: *ICEYE on-board software structure. Flight software consists of OBC software running on OBC and MCU software running on subsystems.* [44]

OBC software is running on a Beaglebone-board, which itself runs a custom build of GNU/Linux called "ICEYE OS" with in-house developed software. Therefore, ICEYE OS is running the minimal GNU/Linux distribution and OBCSW, which controls for example FDIR, packet utilization and others presented in earlier Chapters. It has full control of all the software which runs the satellite. MCU software itself refers to multiple pieces of software, which control all subsystems. More detailed information about these sublevels is discussed in Chapters 3.2.1 and 3.2.2. The functional allocation diagram of OBSW is presented in Figure 14.

Besides ICEYE OS and MCU software, so called OBC API has been developed with Python. With OBC API, it is possible to write scripts which can communicate with subsystems through the CAN bus. Its purpose is to build test scripts or others to allow users to control satellite subsystems, but the actual OBCSW has the highest authority to actually execute commands on MCU software.

3.2.1 ICEYE OS

ICEYE OS has been developed with Buildroot [45], which is a set of tools, such as Make-files, and patches to build and maintain a complete lightweight embedded GNU/Linux distribution and its cross-compilation toolchains. Alongside the embedded Linux distribution, it contains custom flight software in order to control everything on satellite. The basic approaches to use buildroot-based distribution are [46]

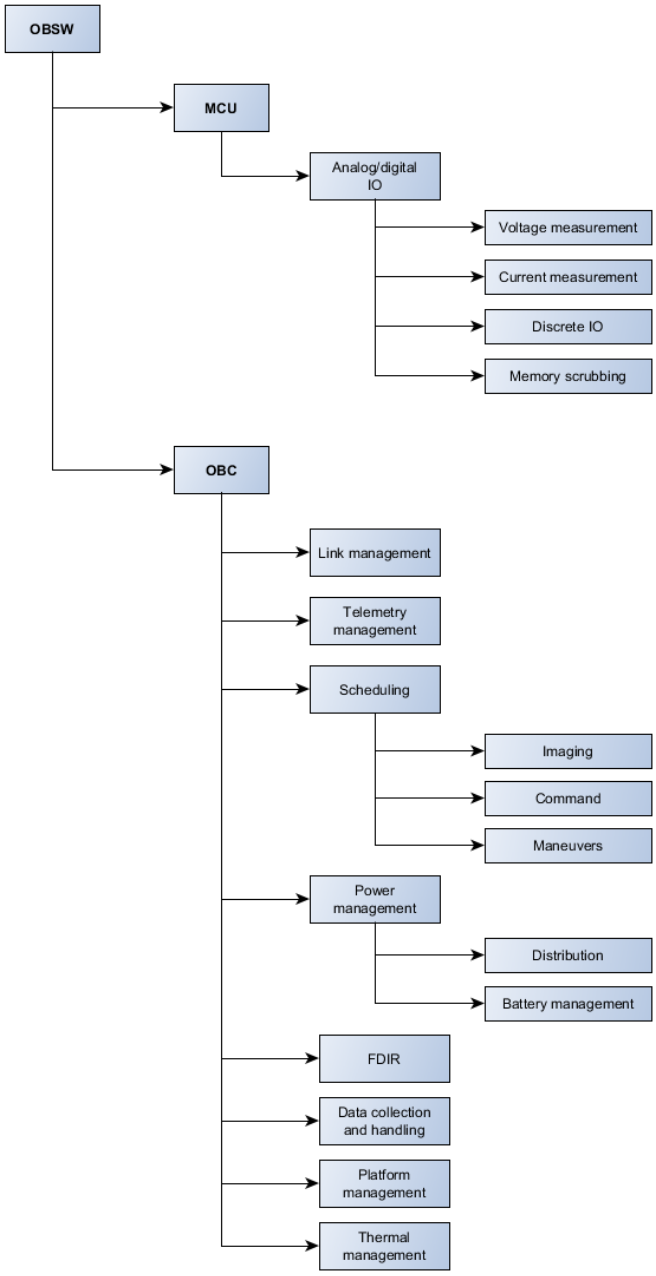

Figure 14: *The abstract functional allocation diagram of ICEYE OBSW. Figure shows how OBC software and MCU software have different functionalities and how software itself is constructed.* [44]

- Full control of all software that runs on OBC
- Smaller software size, and therefore, easier to update software
- Easy to manage kernel modification

In Chapter 2.1. it was described that OBCSW should be an RTOS, but for ICEYE OS and GNU/Linux, this is not the case. However, the MCU software described in

Chapter 3.2.2. runs on RTOS and it is responsible for crucial data processing for the flight mission. ICEYE OS, however, works as an upper level supervisor for all subsystems and instruments. Therefore, with properly integrated FDIR procedures, it fulfills the ICEYE mission requirements. In addition, this approach has been used in several projects earlier, such as Aalto-1, and it is considered as a working solution. [18]

3.2.2 MCU

MCU software is divided into several pieces of software, which run all subsystems of the spacecraft. They are based on ARM Cortex-R boards and run under FreeRTOS. The following subsystems are considered as MCU software: [47]

- Processing board for data processing
- PCM (Power Conditioning Module)
- CDR, module that accepts low-power input from PCM and high-current input from battery and provides all power regulation and switching to boards.
- Propulsion Controller

This software consists of code blocks which communicate with the spacecraft, handle the telemetry and telecommands regarding the subsystem, present the initialization and interfaces between modules and introduce used variables for the subsystem. Therefore, all subsystems share some common functionalities. The generic layout of MCU software is presented in Figure 15.

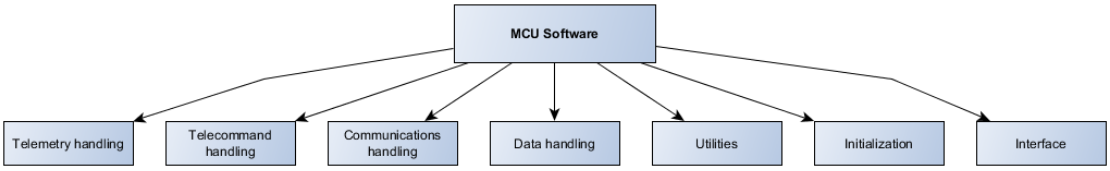

Figure 15: *General MCU software architecture. Figure shows which common modules and features every MCU application contains.* [47]

The development principle of MCU software is to avoid code duplication. Therefore, some parts of the software libraries are shared with the same repository and are used to compile the payload software binary. The shared library is called a *common library*. Alongside the common library, FreeRTOS codebase is also in a shared repository and therefore used for building software. Therefore, when compiling MCU software for a subsystem, it compiles the common library, FreeRTOS library and the actual subsystem code. The build process is described in Figure 16.

3.3 Design constraints

ICEYE can be defined as a "NewSpace" company, which means it is a startup company in space market. A startup company has limited funding, resources and time to provide

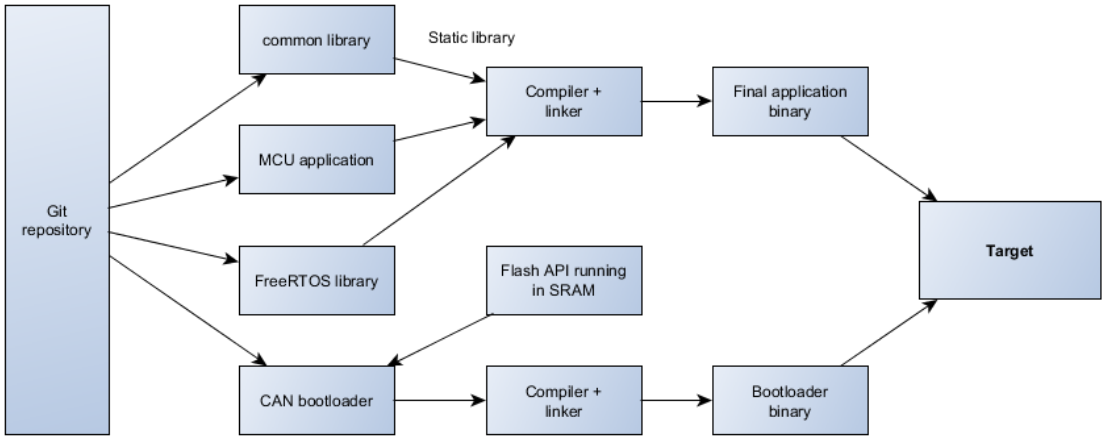

Figure 16: *MCU software compilation process. Software is maintained inside git repository, and where it is finally constructed with three software libraries and CAN bootloader software resulting in the final binary.* [47]

the MVP. This leads to a situation where every single step cannot be verified. Leaps in development must be taken and verified at the system level [43]. This conflicts with previously defined approaches, where every single step should be verified. It promotes risks considering the mission, but with simplified design and focusing on MVP in terms of high-level requirements, it is possible to build a sufficient and reliable product with sufficient testing.

Due to limited resources, iterative development must be executed. This means that instead of strictly following book examples and verifying every single step, development process follows the iteration of: design, build, test, fail, learn, repeat. Therefore, the cornerstones of development of the satellite are based on the same principles as agile and DevOps principles where failures are welcome and feedback is used to improve development.

In terms of software, even though unit testing is not used, testing is a crucial part of the system. Therefore, software requires system level testing over unit level testing which means that a proper test plan must be defined. The basic design approach is that ICEYE tries to achieve redundancy at the satellite level, not at unit level. In this thesis, testing plan for ICEYE is presented.

In addition, software development should use practices, which promote fast development velocity in order to improve overall spacecraft development. Therefore, in this thesis, all development practices which are supported by selected tools and are easily documented are presented, leading to fully functional flight software.

4 Embedded software development methods

This chapter presents the empirical design and analysis of the current software development methods at ICEYE. First, the general design conventions based on a strict requirements listing are discussed. Conventions are analysed and based on company wide interviews, studies and lessons learned from other projects, a new design convention is proposed for embedded software development. Second, development tools and their integration to each other are presented and analysed, possible constraints taken into account and solutions provided. This includes suggestions for improved tool integration with configurations which were designed during the thesis. After that, hardware and software codesign is analysed and a new approach is proposed based on company-wide research and literature review. Fourth, continuous integration principles and the current design are presented and its functionality analysed, while also promoting technical challenges in terms of embedded development. A new improved approach is presented, which integrates with other tools and earlier presented conventions. Last, a software system testing with the newly selected test framework and designed test structure is introduced and test plan for future testing is proposed.

It must be noted that plans may change dramatically during the satellite development. This Chapter presents the current design and suggestions for future implementations, which might change over time.

4.1 Overview

As presented in Chapter 3.2, ICEYE OBSW consists of OBC software and MCU software. Embedded software is responsible for all functionality of the spacecraft. The overall design follows the common conventions of software design presented in general design approaches. Instead of following all strictly defined phases and verifying all steps, some shortcuts have be taken and focus laid on system level functionality. This is because of limited resources and short development velocity cycle of a NewSpace company.

Even if ICEYE software is analysed only on the system level, it does not erase the fact that software must ensure mission criticalness and verify functionality before launch. Therefore, based on researches presented in Chapter 2, requirements analysis must be defined so that it meets the challenges of presented standards, such as ECSS standards, and every step which is distinguished from standardized approach is validated.

In following sections, general information on how software is constructed is provided and current approaches are validated. In case of better approaches, new suggested development methods are presented and results provided. Also, all developed systems which were designed during the thesis are presented, which support new suggested conventions and principles.

These Chapters and selected outcomes are based on the general background information about spacecraft software development, which was discussed in Chapter 2. In addition, modern software development methods are applied which were also discussed in Chapter 2. Everything should be considered from the a perspective that the target platform is a NewSpace company with limited resources and tight development schedule.

All information has been gathered from the following sources:

- Interviews with the employees
- Analysis of results from the a company-wide Critical Design Review
- ICEYE internal documentation
- Measurements of development phases
- Quality of software based on research and developer experience

After every subsection has been analysed and outcomes proposed, the overall conclusions are discussed and potential obstacles identified. These results and designed solutions were implemented directly to ICEYE software development. Therefore, the results and user feedback were received from developers after implemented solutions were taken in use.

4.2 Requirements definition and development practices

The main principle of defining software for ICEYE is that strong software requirements come from other subsystems. Because software controls the other systems, it cannot be defined what are the general software requirements, but instead, what are the requirements for each subsystem or individual software component.

On payload software, the requirements of the software functionality come from hardware capabilities. Based on interviews with the electronics team, it was pointed out that when a new piece of hardware is developed and provided to the software team, software team starts to implement code based on the requirements which hardware sets. This requires the understanding of the hardware for the software developers, which is not always a favorable approach, especially when software developers are purely software-oriented. The codesign of hardware and software is discussed in more detail in Chapter 4.4.

When considering the FDIR, FTA (Fault Tree Analysis) [48] is used to determine which actions must be covered with FDIR. FTA is used to

- Resolve the causes of system failure
- Quantify system failure probability
- Evaluate potential upgrades to a system
- Optimize resources in assuring system safety
- Resolve causes of an incident
- Model system failures in risk assessments

FTA itself is a deductive failure analysis, which is used in resolving the undesired events in a way that logic diagram with specific symbols are used. [48] In ICEYE, FTA is used to determine (1) which failure occurs, (2) isolation action, (3) conditions to detect a failure and (4) action after the detected failure.

Based on Critical Design Review, which occured during the thesis in January 2017 [49], it was stated that the previous design of the FDIR was not sufficient. FDIR was mainly dependent on the scheduler and proper FDIR procedures were not defined for different

subsystems. Therefore the FTA-based FDIR approach was selected and is currently under development. In addition, Critical Design Review stated that software *must* define all those requirements which are not defined in a specific payload, taking a leading role in requirements definition as a software perspective even if the payload design and mission operation is the base of the requirements. This, however, indicates a fundamental problem within limited resources and time: even if there are limited resources to get MVP to operate, it cannot be achieved by taking shortcuts in defining mandatory requirements in terms of provability and redundancy.

When considering different requirements levels, NASA has noticed that when defining system level requirements, which is essential for the ICEYE, some systems requirements were not rigorously tracked and controlled [50]. This results in confusion during the test cycle about success criteria. Therefore, systems requirements need to be clearly traceable back to science requirements and down to instrument and satellite requirements. Thus, systems requirements must be clearly derived from science requirements, then rigorously documented and configuration controlled. This also occurs on software side, where software requirements are derived from payload functionality. However, it is noticeable that handling software requirements differently than other project requirements creates unnecessary work. Therefore, NASA recommends to manage flight software requirements within existing project-level requirements management and verification processes [51].

In a fast developing company, new challenges and requirements are met constantly and everything cannot be defined beforehand. In NASA's "lessons learned" [52] article about traditional requirements development, common strict goals for the team were emphasized from the beginning. The lessons learned concluded the following:

- Identify the interested parties and their goals, objectives, and requirements up front.
- Use one requirements development process from the start, and stick to it.
- Ensure parties of interest are bringing forth mature requirements and that these requirements are funded.
- Write specifications and verifications that can be tested and measured.

However, when considering a NewSpace company, this approach might not be suitable, and it has also been considered that NASA's approaches are not flexible enough in terms of changing requirements and small teams. This, however, is directly relevant to traditional software development methods, such as waterfall, which was discussed in Chapter 2.3. Therefore, this thesis suggests that in requirements definition, agile practices are used as presented in Chapter 2.3.1.

Even if mission criticalness demands that requirements are strict and not constantly changing, it limits the ability to meet new challenges. Therefore, when analysing software requirements, or requirements in general, mission operations requirements are most likely to stick to an early definition but requirements considering the payloads, especially those developed in-house, might change constantly. In these cases, this thesis proposes to consider *acceptance test-driven development* (ATDD) where initial requirements come from mission operation and subsystem software requirements are test-driven. Therefore, based on ATDD principles [53], the designed and therefore proposed structure is following:

1 Define high-level requirements considering the mission, which are most likely not to change
2 Identify the software architecture, which is used to construct software.
3 Start implementing test cases per software, how it should perform in flight
4 Define software requirements considering the test cases. This also helps identifying the test plan for system level testing.
5 If software requirements change after iterations, re-define the test plan and consider this as a new software requirements list
6 Review the test plan, and requirements, after iterations

ATDD can be easily considered in terms of system level testing which was introduced as an essential point in NewSpace product development. After developers have identified the use cases of the software functionality as test cases in system level, test cases can be turned into proper requirements which fulfill the mission criticalness. If it is seen that the level of test cases are not defined well enough, it is easy to switch back to testing and therefore define more requirements for the system with more details. The requirement listing is iterative, and at the same time, it provides a proper test plan for the system level testing. Because we are considering the mission critical software, reviewing the test plan is mandatory as it is the only possible way to ensure that all required functionality and mission criticalness is covered.

ATDD also helps us in defining software where hardware sets the restrictions. Hardware developers can define system level test cases from the hardware perspective, which can be converted to software requirements. Therefore, development requires more interaction between HW developers and SW developers, where HW developers provide the "use case" and test plan for the single piece of hardware, which can be included in the whole system level acceptance test. The details about HW/SW codesign is presented with more details in Chapter 4.4.

ATDD, among other agile methods, can provide straightforward development style and significant improvements for the small teams [37] [54] [55]. On the other hand, it must be well defined how these methods are used in practice because small teams probably do not suffer from same issues as larger companies, considering the development methods, and therefore every single practice, such as ATDD, must be verified in development model separately. The main aspects of different development methods are the tradeoffs between *cost*, *time*, *quality* and *scope*. [55] In this specific case in ICEYE and considering the development environment as a NewSpace company, those tradeoffs for ATDD can be presented as following:

- Cost
  - \+ Prevents re-designing the whole board or software after requirements might change using automation
  - \- Updating system level requirements and test cases demands additional work, and thus, costs
- Time

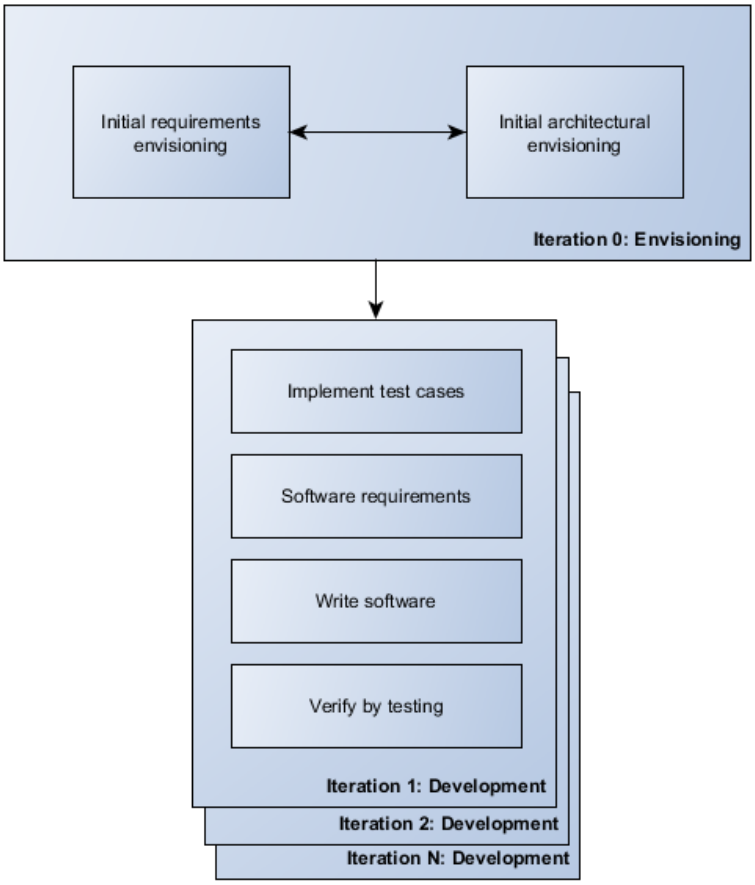

Figure 17: *Acceptance test-driven development approach. At the first iteration, the fundamental requirements which are likely not to change are defined. When these requirements are defined, software tests are written and software modules are then developed with agile principles. This is processed until enough iterations have passed and software is considered ready.*

  + More straightforward development phases with specific requirements and test cases improve the development velocity with improved feedback
  - Defining the abstract system level requirements could be time-consuming in the beginning in contrast to traditional methods. In addition, promoting new methods for developers requires additional time.

- Quality
  + Better communication between hardware and software teams, ensuring better quality of the product, and test cases promote robustness of the code
  - Misleading communication and poorly defined requirements/test cases could lead to potentially decreased performance

- Scope
  + No additional changes to scope, while fundamental requirements themselves do not change.

- No additional changes to scope, while fundamental requirements themselves do not change.

Based on the analysis above and using ATDD with the other development methods presented in other Chapters, especially interface-driven HW/SW codesign in Chapter 4.4, it is recommended to use ATDD as a potential approach in software development.

4.3 Development tools

Development tools are used to improve the development process. Tools must be used in a way that they support the development cycle and ease the produce the final software. In this case, "development" means the whole software development lifecycle from requirements definition to the final product, not only the codebase development.

Every single tool in use must be qualified and its purpose justified during the development. It can be a risk that companies try to apply several tools into their software production just because it is modern software development tool, but it does not actually provide any benefits. The tool may even decrease the development velocity by adding an unnecessary production step. For example, if we analyse the DevOps toolchain presented in Chapter 2.3.2, we must justify how we apply each step on company's software development process and which tool provides the best result to achieve it. Therefore, when choosing tools, the following questions have been set in this thesis, which are used to analyse the use of tools:

1 In what development phase the tool is used?

2 How the tool is used, who uses it?

3 What benefits the tool provides in the development cycle?

4 What does it improve in development? (code quality, development velocity, verifying)

5 What costs it causes to developers and company?

6 How long does it take to apply the tool in development cycle?

Alongside checking how tools are used, it must also be defined how different tools are integrated with each other. In this context, *integration* means how tools work with each other and that overall workflow is seamless. For example, if a company uses same development tools created by a single provider, interfaces between several tools are usually well defined. However, if different tools from different providers are used, the interfaces and compatability should be ensured with, for example, third party plugins or self-developed interface tools.

In Chapter 2.3.2, it was underlined that one of the faced obstacles in embedded systems development is the availability of tools. This issue is highlighted in cases where a custom piece of hardware can be programmed only with a proprietary software and toolchains have no open source alternatives. In this case, it might be an issue that developers are strictly tied to, for example, one single IDE (Integrated Development Environment) which could restrict using other possible tools while interfaces do not work as expected and IDE itself does not support all required features.

For NewSpace companies, the case is however that not all hardware is custom made and most of the subsystems are bought as COTS. In addition, in cases where custom hardware is developed, for example OBC, microprocessors and its features are developed by other providers. Only a minimal work is actually developed in-house, and therefore other subsystem developers should provide enough documentation and tools for development.

The use of open source tools is encouraged, if possible. These tools do not have possible license fees, and community support and documentation is probably sufficient. In addition, these tools usually support third party plugins and integration with other tools is usually easier than in proprietary tools [56]. If there are alternatives between open source and proprietary ones, it should be analysed if a proprietary tool provides more benefits in usage.

The choice of tools is always dependant on the develoment lifecycle and which other tools are present. Therefore, it is not encouraged to only analyse single tools which can be used in the whole development, but instead how they integrate with each other.

Alongside tools for version control, reviews, building and others, it should also be considered if automation can be applied in software development. One of the DevOps principles is to automate some stages as much as possible such as CI. [39] Therefore, when considering the tools available for software development in embedded systems, it should also be considered *can it be automated*? Considering the available tools, the design on testing and other issues, automation can also be applied in embedded systems in the same way as in SaaS applications. These possible solutions are discussed in this thesis and results for analysed.

In case of potential obstacles considering the mission criticalness or embedded systems are identified, they are discussed and potential alternative approaches presented even if they are not implemented in the current solution.

4.3.1 Version control and code review

In projects where multiple developers make changes to source code base, it is necessary to have tools which improve source code or other information management between participants. Besides information sharing, backups and logging ability for source code is required in case of erroneous piece of code. While multiple developers have access to codebase and especially in mission-critical software, peer-reviewing of the code is required in order to keep software development coherent and not to add unnecessary or harmful code. In this Chapter, the use of version control software Git and review tool Gerrit is analysed as well as their functionalities at ICEYE. [57] [58]

Git is an open source VCS (Version Control System) which is largerly used in industry-level software development and also in smaller teams. It is a distributed revision control, meaning that it can be accessed in different networks and therefore software developers do not need to share the same network. Instead of working with several software versions, git focuses on working with *revisions*, where changes developers make to fix a specific bug, infrastructure changes that were needed to support that fix, another set of changes that didn not work out, and some work in progress that was interrupted to work on that urgent bug fix. It was originally developed by Linus Torvalds, but currently maintained by Junio Hamano and other developers. [57]

VCSs, such as git, have become de facto tools for modern software development where the software development can be distributed and allowing numerous different developers to co-operate [57]. Even if different VCSs provide different features and slightly different approaches, their main idea remains the same. VCSs do not only promote the developers work by faster release cycles, but also promotes marketing capabilities and product management [59] [60]. An example of common git workflow with different branches is presented in Figure 18.

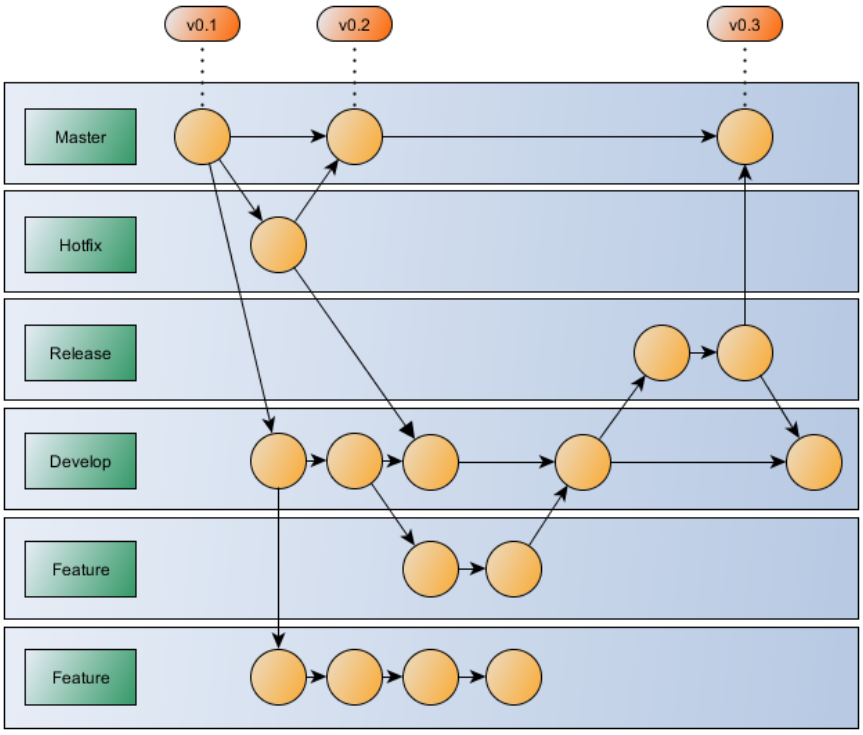

Figure 18: *An example of git development flow. Different software modules are developed in multiple branches, which are later merged into master branch for different software releases. The provided branch names and commits to them are considered only as an example and do not present the actual flow at ICEYE.*

Gerrit is a web based code collaboration tool, which consists of repository management and review tools. It is intented to work with Git, and therefore works only with Git based repositories. Gerrit allows managing repositories between team members alongside reviewing other developers' latest commits, which promotes code quality and ensures coherent structure of development. When modern review tools have emerged, such as Gerrit, they have provided valuable information about defect detection in post-release. Case studies have shown [61] [62] that using proper review methods creates higher quality and less defect-prone software. This however requires that reviewing is done efficiently, and components with low review participation are estimated to contain up to five additional post-release defects. [61]

Alongside Git and Gerrit, a tool called **Repo** is also used. Repo is simply a repository management tool which allows managing multiple different repositories, or projects, by

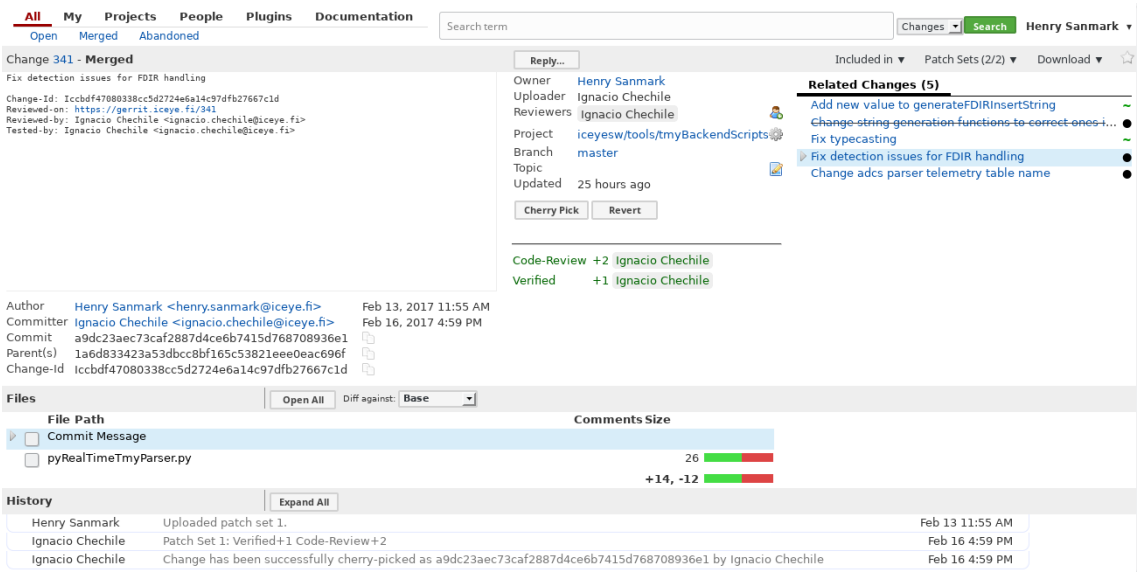

Figure 19: *An example of Gerrit window after a reviewed and merged code. Gerrit presents all information about the commit, such as commit message, reviewers and, changed files and related changes. In this example, ground segment telemetry parser for FDIR has been fixed.*

unifying them into one single command. At ICEYE different projects are developed under different repositories, such as different subsystem software, and therefore managing every single one of them and keeping them locally up to date is achieved using a Repo. [63]

The basic pattern of working with repositories using Git, Repo and Gerrit is the following:

1 Use `repo start` to start a new topic branch.

2 Edit the files.

3 Use `git add` to stage changes.

4 Use `git commit` to commit changes.

5 Use `repo upload` to upload changes to the review server.

Reviewing is also a part of the ICEYE software development, using Gerrit. However, in contrast to traditional software projects, ICEYE software development is done in a small team and in addition, reviews are not always performed correctly. Usually commit verification is just clicked through without the actual review process in order to speed up the development. We can, however, analyse the current reviewing process and therefore consider how these processes affect to whole software development practices which leads us to two questions:

1 Is there a relationship between code review coverage and post-release defects?

2 Is there a relationship between code review participation and post-release defects?

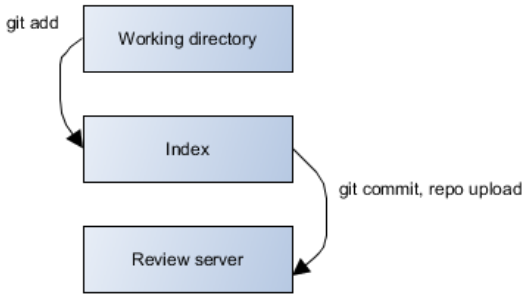

Figure 20: *The basic workflow of using Git, Repo and Gerrit. From local folder, changes are added to index, where git and repo keeps files updated and then transfers them to the review server which is Gerrit in this case.*

Table 5: *A taxonomy of the considered control (top) and reviewing metrics (bottom). [61]*

	Metric	Description	Rationale
Prod	Size	Number of lines of code.	Large components are more likely to be defect-prone.
	Complexity	Cyclomatic complexity	More complex components are likely more defect-prone.
Process	Prior defects	Number of defects fixed prior to release.	Defects may lingerin components that were recently defective.
	Churn	Sum of added and removed lines of code.	Components that have undergone a lot of change are likely detect-prone.
	Change entropy	A measure of the volatily of the change process.	Components with a volatile change process, where changes are spread amongst several files are likely defect-prone.
Human factors	Total authors	Number of unique authors.	Components with many unique authors likely lack strong ownership, which in turn may lead to more defects
	Minor authors	Number of unique authors who have contributed less than 5% of changes	Developers who make few changes to a component may lack the expertise required to perform the change in a defect-free manner. Hence, components with many minor contributors are likely defect-prone.
	Major authors	Number of unique authors who have contributed at least 5% of changes	Similarly, components with a large number of major contributors, i.e., those with component-specific expertise are less likely to be defect-prone.
	Author ownership	The proportion of changes contributed by the author who made the most changes.	Components with a highly active component owner are less likely to be defect-prone.
Coverage	Proportion of reviewed changes	The proportion of changes that have been reviewed in the past.	Since code review will likely catch defects, components where changes are most often reviewed are less likely to contain defects.
	Proportion of reviewed churn	The proportion of churn that has been reviewed in the past.	Despite the defect-inducing nature of code churn, code review should have a preventative impact on defect-proneness. Hence, we expect that the larger the proportion of code churn that has been reviewed, the less defect prone a module will be.
Participation	Proportion of self-approved changes	The proportion of changes to a component that are only approved for integration by the original author.	By submitting a review request, the original author already believes that the code is ready for integration. Hence, changes that are only approved by the original author have essentially not been reviewed.
	Proportion of hastily reviewed changes	The proportion of changes that are approved for integration at a rate that is faster than 200 lines per hour.	Prior work has shown that when developers review more than 200 lines of code per hour, they are more likely to produce lower quality software. Hence, components with many changes that are approved at a rate faster than 200 lines per hour are more likely to be defect-prone.
	Proportion of changes without discussion	The proportion of changes to a component that are not discussed.	Components with many changes that are approved for integration without critical discussion are likely to be defect-prone.

In order to analyse the code and reviewing methods, the taxonomy of metrics is defined in Table 5 which can be used as a base for analysing the value of reviews. McIntosh et. al. [61] have used these metrics and analysed how the review process affects in code development based on the extracted commits, defects and code lines using Multiple Linear Regression (MLR) models. In their research, it was found that:

> *If a large proportion of the code changes that are integrated during development are either: (1) omitted from the code review process (low review coverage), or (2) have lax code review involvement (low review participation), then defect-prone code will permeate through to the released software product.*

Basically the case-study conclusions can be expressed as:

- Code review coverage metrics only contribute to a significant amount of explanatory power to two of the four defect models when we control for several metrics that are known to be good explainers of software quality.
- Two of the three code review participation metrics contribute significant amounts of explanatory power to the defect models of each of the studied software releases.
- Components with low review participation are estimated to contain up to five additional post-release defects.

While the research was targeted to software projects which were neither mission-critical nor embedded projects Qt-project, VTK (The Visualization Toolkit) and ITK (Insight Segmentation and Registration Toolkit) the same principles of writing and reviewing commits still exists. While the results were not dependant on the lines of code, even if it was taken into consideration, results can also be applied to the software project which is the size of ICEYE project. However, because the model requires that majority of commits are linked to reviews, and while ICEYE uses reviewing tools but they do not use them efficiently in practice due to strict schedules and limited resources, the same model cannot be applied for ICEYE software development directly. Therefore, calculated values for benefits of reviews in the current development methods cannot be presented.

This however indicates the importance of the review process, and is also an example of the situation where tools have been taken into development cycle but not used properly. The review is used in small teams by interacting with each other, and only single developers are responsible for unique parts of the software. Thus, even if review process is practiced, it is not used in Gerrit properly. The worst case is that the tool actually slows down the development velocity depending on the faced problems and resources allocated to tools maintenance. Based on the interviews, it was noted that Gerrit is in fact considered as an obstacle while using version controlling. Even if only one reviewer is required to accept patches, they are not read through or actually reviewed and the latency between the actual push and review process could lead to possible merge conflicts which has happened.

Based on this, it is presented that Gerrit is dropped in the review process. In the current scope, it does not provide any benefits when it is not used and reviewing should only be performed only between different team members. Thus, reviewing should be emphasized in terms of software criticalness, but in current situation this should be done between team members which are responsible for their piece of software. Even if this leads to potential dangers in "software ownership", this does not have any direct answer in terms of resources. However, when the number of developers and reviewers increases, the currently used Gerrit should be re-evaluated and taken back into the process.

Besides Gerrit, in general it is possible to use other repository management tools such as GitLab [64]. While GitLab also supports reviewing, it also supports CI, wikis, test automation and other features. However, comparing these different solutions is not in the scope of this thesis.

4.3.2 Build tools and build process

In order to create embedded software, cross-compilation toolchain and is required to generate proper binary files which run on spacecraft - either on BeagleBone (ICEYE OS)

or ARM Cortex-R (MCU software). ICEYE OS is built with Buildroot tools, which were presented in Chapter 3.2. MCU software use custom toolchain provided by MCU provider. In this Chapter, the internal structure of both ICEYE OS building and MCU software building is analysed and proposed changes presented.

Buildroot contains mainly just a set of Makefiles that download, configure, and compile software with the correct options. Alongside Makefiles, there are several patches available for different software. These are mainly the ones related to toolchain, such as `gcc`, `binutils` and `uClibc`. Makefiles are divided into multiple subdirectories which define different configuration options and building targets for GNU/Linux environment, such as kernel, toolchain, processor architecture, user space software, which in this case is ICEYE flight software, bootloader, initial system and filesystem. When the main Makefile, which calls all the sub-Makefiles, is executed, it creates the corresponding output folders for binaries, generates toolchain targets and last, builds all targets defined in Makefile's `TARGETS` variable. [45]

When Buildroot has been succesfully compiled, it will produce different software assets which are [43]

- MLO file - X-loader / SPL (Secondary Program Loader)
- U-boot images
- Root filesystem

When ICEYE OS has been built, it can be flashed to OBC using U-boot bootloader, which is the universal embedded system bootloader.

While Buildroot is open-source and highly customizeable toolkit, it is possible to run with multiple different toolchains and environments. Besides, it can be easily integrated to the CI workflow as a Jenkins job, so its usage is highly encouraged. While current solution with Buildroot is not the only acceptable one, and embedded GNU/Linux distributions and software can be built also with tools such as Yocto Project [65], Buildroot provides all required tools to develop ICEYE OS and include it in an efficient pipeline. However, analysing different toolkits to build embedded GNU/Linux software is not in the scope of this thesis.

Because MCU software is built for completely different environment, ARM Cortex-R running with FreeRTOS, it cannot use the same tools than ICEYE OS, and therefore, requires a completely different set of toolchain and building methods.

MCU software is mainly developed with the Eclipse-based IDE. The tool differs from Eclipse in a way, that it contains all toolchains and building options, such as Makefile generation, directly within the IDE. Therefore, in order to build software for MCU, developer must only execute "Build" command inside the IDE to generate executable binary.

However, this method results in issues when integrating tools to each other. For example, implementing the IDE inside CI causes problems when build servers must have additional dependencies to run the IDE, even if it would be called through command line options. In addition, because IDE uses its custom configuration parameters to, for example, generate Makefiles, it makes it more difficult for developers to track down issues relating to build process, which also conflicts with the earlier defined problems of trackable requirements.

Building the software relying only on the IDE and its configurations is considered as a bad practice [66] which also restricts the development capabilities. Therefore, the build process must be platform-independent. However, IDEs can be useful to easily develop software on single developer's PC, but limitations arise when some of the processes, which are taken care of by the IDE itself, should be integrated with multiple other tools which are essential for the pipeline.

In order to get rid of the IDE building tools, own build process must be defined. Even if toolchain is provided by MCU provider, it can be separated from the actual IDE and used with self-defined compiler options. Writing own Makefiles from the scratch is considered a hard process especially in large software projects. A better approach was designed during the thesis research. Therefore, **CMake** is used to write platform and compiler independent configuration files, which define the whole build process.

CMake is an open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and to generate native makefiles and workspaces that can be used in the compiler environment of developer's choice. [67] This allows getting rid of IDE dependant configurations and providing flexible way to maintain dependencies and optimizing the whole build. In addition, when using platform independent tools, integrating build process to CI is much more fluent.

As presented in Chapter 3.2.2, every piece of MCU software consists of three parts: (1) FreeRTOS library, (2) commonLib and (3) application itself. Therefore, when building MCU software, two levels of CMake files has been defined.

- Main CMake- file, located under application root
- Subfolder/library files, located under every subfolder containing source files

The main idea is that "main file" contains all necessary settings for building the application and calls other files under the project tree. In the main file, all necessary folders are included and the actual application is built. All subfolder/library files generates only objects which are used to build the program in main file. Therefore, developer must refer to subfolder contents as objects to build an application succesfully. However, FreeRTOS and commonLib are built as library files (.a) and are linked to main executable during the last phases of build process. See appendices A and B for an example of the CMakeList.txt files used to build MCU software.

After the binary has been built, it must be converted to the right binary format with object and hex convertion. This is called a "post-build step". In this case, MCU provider tools are used. The final resulted binary file is ready to be flashed to the OBC.

To understand how the actual code is processed, we can refer to Texas Instruments "ARM Optimizing C/C++ Compiler" site which covers the whole process and which is used as a base for writing CMake files. The same process is also presented in Figure 21. [68]

1 The compiler accepts C/C++ source code and produces ARM assembly language source code.

2 The assembler translates assembly language source files into machine language relocatable object files.

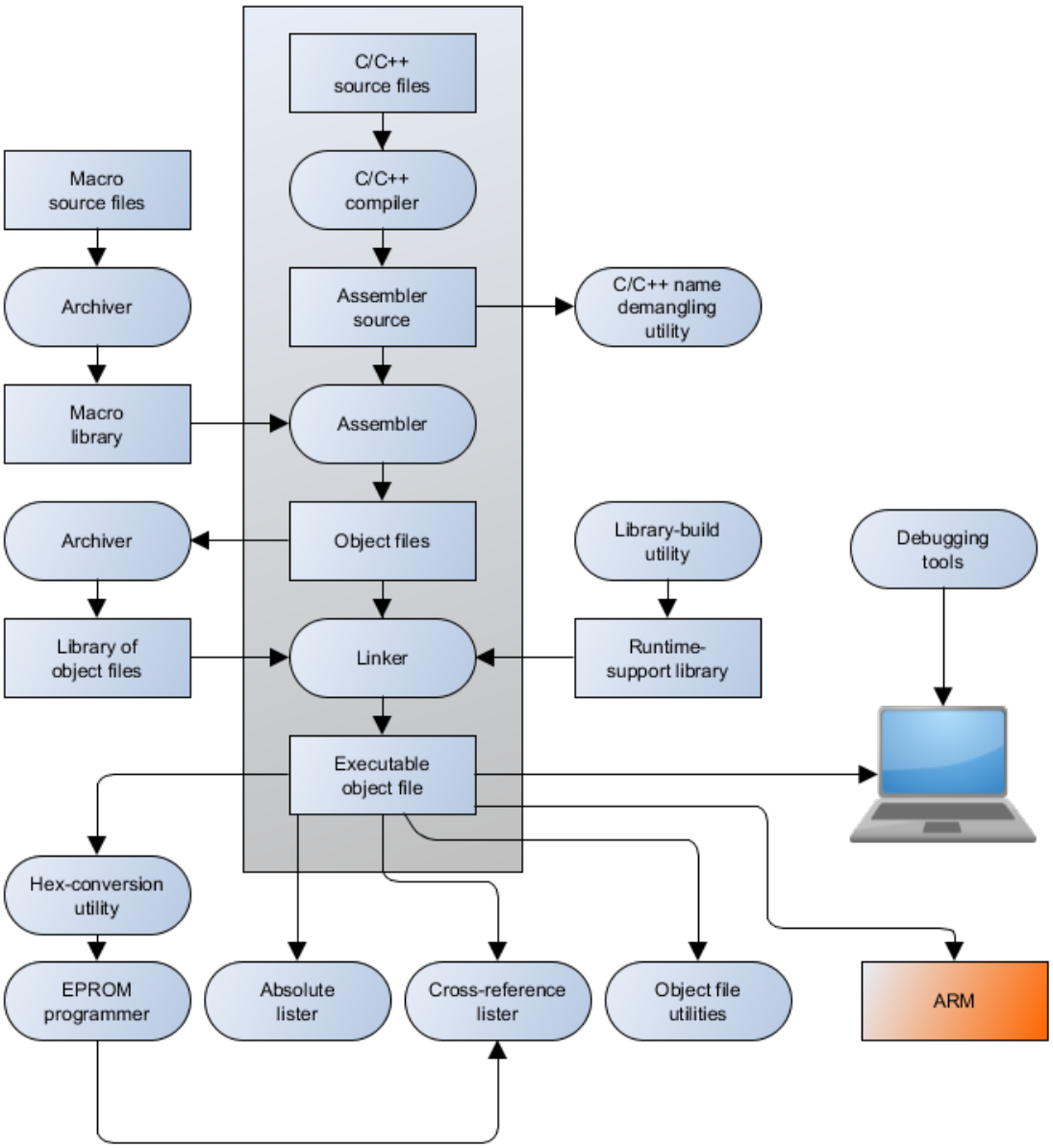

Figure 21: *ARM software development flow and tools relating to it. Every phase is handled with corresponding build tools and combined together within maintained CMake files, which result in final executable ARM binary. [68]*

3 The linker combines relocatable object files into a single absolute executable object file. As it creates the executable file, it performs relocation and resolves external references. The linker accepts relocatable object files and object libraries as input.

4 The archiver allows you to collect a group of files into a single archive file, called a library. The archiver allows you to modify such libraries by deleting, replacing, extracting, or adding members. One of the most useful applications of the archiver is building a library of object files.

5 The run-time-support libraries contain the standard ISO C and C++ library functions, compiler-utility functions, floating-point arithmetic functions, and C I/O functions that are supported by the compiler.

6 The library-build utility automatically builds the run-time-support library if compiler and linker options require a custom version of the library.

7 The hex conversion utility converts an object file into other object formats. You can download the converted file to an EPROM programmer.

8 The absolute lister accepts linked object files as input and creates .abs files as output. You can assemble these .abs files to produce a listing that contains absolute, rather than relative, addresses. Without the absolute lister, producing such a listing would be tedious and would require many manual operations.

9 The cross-reference lister uses object files to produce a cross-reference listing showing symbols, their definitions, and their references in the linked source files.

10 The C++ name demangler is a debugging aid that converts names mangled by the compiler back to their original names as declared in the C++ source code. As shown in Figure 21, you can use the C++ name demangler on the assembly file that is output by the compiler; you can also use this utility on the assembler listing file and the linker map file.

11 The disassembler decodes object files to show the assembly instructions that they represent.

12 The main product of this development process is an executable object file that can be executed in a ARM device.

It is remarkable that the process of how the final executable object binary is generated, is the same no matter what set of IDE or project handlers are used. However, the availability of toolchains could restrict the choice of available open source tools if they do not work with each other.

The achieved outcome of this build process indicates that it is recommended to promote platform and compiler independent configuration files and building options. Alongside unifying the building environment for different developers and CI build servers, it also provides easier way to share same configurations via Git repositories rather than configuring every single build computer separately using the manual in documentation.

4.4 Hardware and software codesign

A fundamental question of developing embedded systems is the codesign of hardware and software. A common issue is that developing a piece of hardware and developing software is considered as different jobs and done by different teams. For example, hardware is designed by the electronics team and software is developed by the software team. The greatest issue considering the HW/SW codesign is that hardware design defines the requirements considering the software, and therefore software development can be considered as hardware driven. [40] Thus, the common issue is that software can be developed only after the hardware has been developed while it could be achieved concurrently. Especially when

systems are complex, concurrent development is more difficult and requires engineers who master both hardware and software. [69]

In this Chapter, the current hardware and software development principles, considering their relation and dependency on each other, are analysed and an improved proposal of the future development practices is provided.

4.4.1 Product design workflow

As stated earlier, software development is hardware driven meaning that software requirements are defined based on the design of the hardware. To understand how the current approach is implemented, it is required to analyse the basic simplified principles of the hardware development practices and how its requirements are translated into software requirements. The detailed explanation of hardware development is not presented in this thesis.

When considering "hardware" in terms of this thesis, it is commonly referred to PCB (Printed Circuit Board) and its components where custom software is developed. PCBs are developed by the electronics team while software team develops the software based on the PCB schematics. During the interviews and analysing the current approach of the HW/SW development, simplified flow can be described as follows:

1 Mission operation requirements are analysed

2 System specification and the architecture of the spacecraft hardware is defined

3 PCB is designed based on system level architecture and module-level requirements. PCB schematics follow the company guidelines, for example, in considering pins and annotations

4 When schematics are considered ready, it will be passed on to the software team. At the same time, the PCB is being manufactured.

5 The software team writes the corresponding software based on schematics and system level requirements which declare threshold values and operational variables

6 Interfaces between different pieces of software are defined during software development

7 Hardware and software are tested when software is considered ready and manufactured PCB arrives

8 Repeat to the next PCB iteration

Even though this process results in the working embedded system, it has multiple significant disadvantages considering the development velocity and other project management. In addition, when newly written software catches problems of the hardware, it is more difficult to fix problems, which itself leads to slower development velocity. [69] The most notable issue is time delay between PCB iteration and software development. During the interviews, it was stated that updating the OBC API and actual software takes three weeks before software is implemented to hardware, even if software team receives the schematics of the PCB before it gets manufactured. This also causes another issues to rise, which

is currently not a problem at ICEYE, but is not sustainable and can cause problems in other companies or facilities: software developers need to know how hardware works and to be able to read schematics. Therefore, software developers should also be hardware oriented. It is usually an advantage for companies that hardware developers also know software and vice versa. Even if as embedded software developers, they know how HW works but as software oriented they do not have as high experience with hardware as is required. This can lead to potential development issues where interfaces between hardware and software are not clear enough. Thus, interfaces between hardware and software should be clearly defined and not relying on the fact, that hardware developers know how to write software, and that software developers know how to build hardware. Even if knowing both of these sides is mandatory for embedded systems developers, they are significantly different expertises.

In traditional spacecraft projects, newly emerged solutions exist, such as Virtual Platforms. In this concept, the whole PCB is emulated the same way as virtual machines, so all electronics are available as a software way before PCB is manufactured. [70] [71] [72] This is not used at ICEYE in terms of resources, but can be considered in larger projects.

While current experience between hardware and software is not considered as a problem by developers, during the interviews it was admitted that after potential growth of the company, from NewSpace company to a company with bigger revenues, it could be a potential risk. Besides, the current slow development velocity was considered a problem but still a sufficient solution in current development.

In the next section, a proposal of how to improve the interfaces between software and hardware is provided alongside the improved development velocity from hardware iterations to a working piece of software, where earlier described agile practices are applied.

4.4.2 Proposed future approach

Because developing hardware differs from developing software, it is not recommended to use hardware-oriented or software-oriented approach [69]. For example, usually software developers argue that their agile method could be more efficient because software is easy to modify and develop in small iterations, while hardware designers claim that their approach is more efficient resulting in a working system. However, the working solution requires the combination of the hardware-oriented development and software-oriented development.

Jerraya et al. [69] suggest the first step of improving the hardware-software codesign is to design proper interfaces. Therefore, instead of developing PCB and software as "separate projects", the co-development should rely on commonly designed interface where both hardware and software are implemented by separate teams concurrently, leading into better documented process and whole structure. For example, at OBC API is used as an interface between hardware and software for testing purposes but it is developed after the actual software is written. Therefore, it is suggested to enlarge the role of the OBC API and consider it as the main interface of the development, and with its interfaces develop both software and hardware concurrently, by redesigning its current functionality. In addition, ESA has proposed [14] that when developing interfaces between hardware and software, especially in payloads, which is the case at ICEYE, the interfaces should be standardized to ease the HW/SW codesign. This is mainly because of the following reasons [14]:

Table 6: *Five abstract interface levels for codesigning hardware and software. [69]*

Explicit interfaces	The currently used model for SoC (Single System on Chip) design describes hardware as RTL (Register Transfer Level) modules. The CPU acts as the HW/SW interface, and designers use explicit memory and I/O architectures to detail the software down to assembly code or low-level C programs.
Data transfer	At this level, the CPU is abstract. Hardware and software modules interact by exchanging transactions through an explicit interconnect structure, a model generally referred to as TLM (Transaction-level Modeling). In addition to designing interfaces for different hardware modules, refining a TLM model requires designing a CPU subsystem for each software subsystem.
Synchronization	At this level, the interconnect and synchronization are abstractions. The hardware and software modules interact by exchanging data following well-defined communication protocols. The MPI (Message Passing Interface) is an example of this approach. Refining an abstract HW/SW interface model requires first designing the interconnect and then correcting the synchronization schemes. Data transfer must also be refined down to the RTL.
Communication	At this level, the communication protocol is abstract. The hardware and software modules interact by exchanging abstract data without regard to the protocol used or the synchronization and interconnect the design will implement. The design typically uses the SDL (Specification and Description Language) to abstract communication. Refining an SDL model requires first selecting a communication protocol for example, message passing or shared memory and then following the refinement steps used in lower abstraction levels.
Partitioning	The ultimate abstraction level is the functional model in which hardware and software are not partitioned. Designers can use a variety of models to abstract HW/SW partitioning, including sequential programming languages such as C/C++, concurrent languages, and higher-level models such as algebraic notation. Refining such a model requires first separating the software and hardware functions and then performing the refinements used in higher abstraction levels.

1 Payloads have to fulfil a common set of requirements to interface the platform:
   - Initialisation sequence: boot software to RTOS or similar
   - Interface Management such as SpaceWire and/or MIL-STD-1553
   - Standard services such as SOIS, PUS-C, FDIR support services, file management for direct data access
   - Payload mode management
   - OBCP Interpreter

2 Architecture of the payloads tend to adopt a common configuration:
   - One processor to manage the interface to platform
   - The same or a dedicated processor to manage the payload
   - A dedicated set of analysers, imagers
   - Possibly some local processing (CPU/FPGA) and storage units

According to ESA analysis [14] of the topic:

"We have reached a maturity status where standardization and rationalization of payload interface, functionalities and provided standard services is achievable. While this is a lesser issue for missions that have a single 'big' payload it can be important for systems that have to host multiple payloads each with very similar interfaces."

Therefore, when developing PCBs for ICEYE mission, or in general, standardized interfaces of different payloads can be applied. This also provides advances in the long run.

However, considering the process and their complexity, 5 different abstraction levels have been designed, which are required for sufficient HW/SW codesign to be implemented. The ultimate goal is to develop both HW and SW with abstraction levels defined in Table 6, where development is interface-driven rather than HW/SW-driven. [69]

In order how to understand how these abstraction levels work in practice, we can refer to Figure 23 which defines the full HW/SW interface codesign leading to a working example of the embedded system.

Standardized interface-driven abstraction development model provides multiple benefits to the earlier approach, which is also validated by previously presented researches. Besides, every abstraction level can be considered as an agile iteration which can apply the agile principles. Thus, when considering the overall improvements of the embedded systems development, this approach is recommended. In practice, the following procedure should be considered for ICEYE development:

1 *Partitioning:* Implement abstract communication with OBC API functions and relevant methods as hardware interfaces. In practice, this means all the functions which are relevant to the communication, but do not define the actual implementation. This is also a base for OBC API architecture.

2 *Communication:* Define the communication in a way that shows how data is passed between modules. This requires that the actual protocol is also selected. In this, the

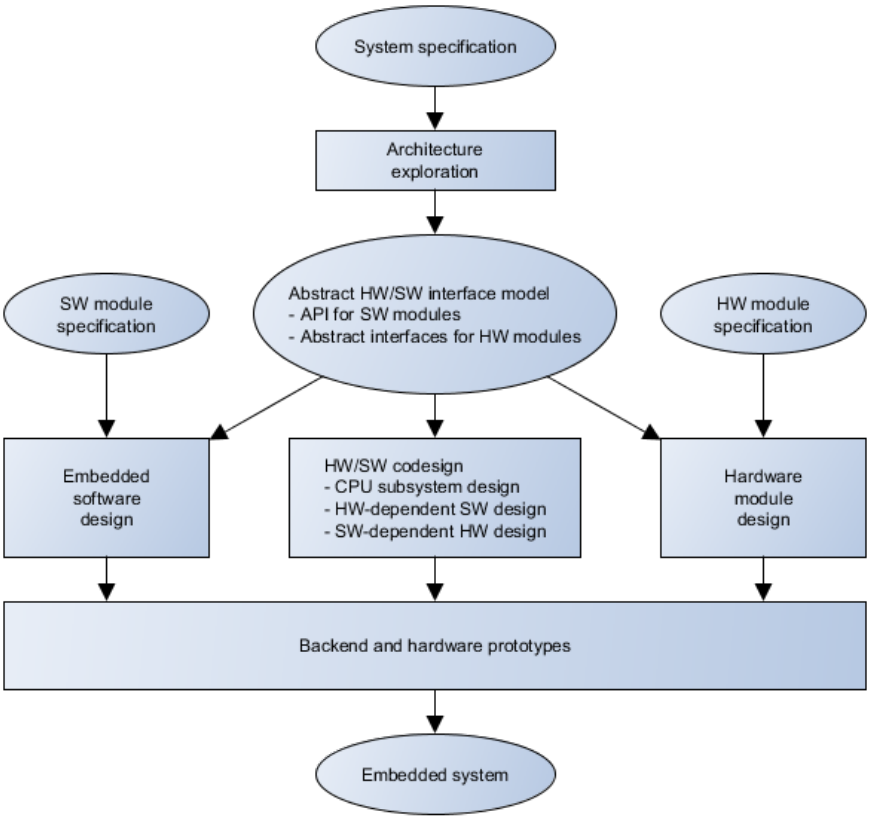

Figure 22: *The proposed approach. In this approach, system and architecture are specified and based on that, hardware and software interfaces are defined and both systems last developed, based on this codesign. This approach is proposed to be used with earlier defined ATDD approach with multiple iterations on software and hardware modules by defining test cases.* [69]

message location in SW and also in HW are defined and by which steps the message is processed between SW and HW.

3 *Synchronization:* In this phase, the communication is defined in deeper level. Thus, the actual implementation of software and hardware is defined. For example, how message is processed in flight software and how the message is processed inside the PCB, such as in Processing board. In software side, functions are written and detailed implementation of PCB is designed.

4 *Data transfer:* The actual communication architecture is defined, meaning how the data processed inside software and hardware is implemented and then passed on to OBC API. This phase encapsules the whole flight software-OBC API-hardware processing.

5 *Explicit interfaces:* All detailed memory locations and I/O architecture, which are highly hardware dependent.

However, this method also requires that inside Step 3, synchronization, the PCB

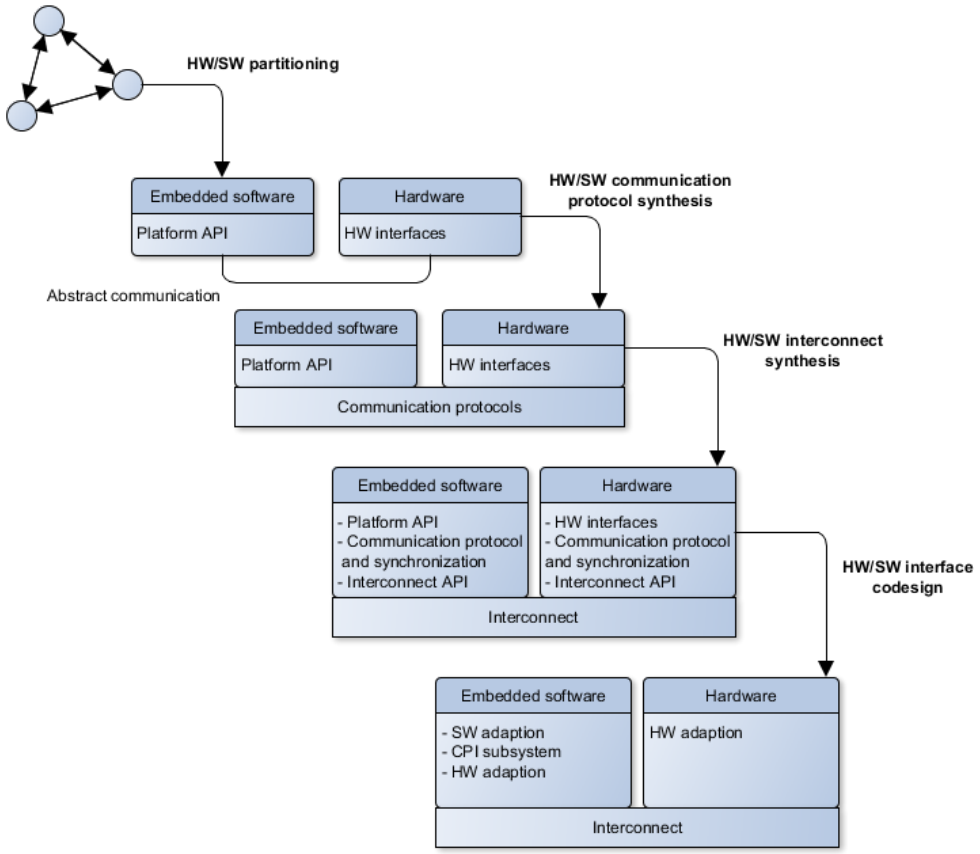

Figure 23: *Full HW/SW interface codesign scheme: explicit interfaces, data transfer, synchronization, communication, and partitioning. Based on illustration from Jerraya et al.* [69]

functionality is tested throughout with simple functions. It does not serve if OBC API is well designed but the actual board is not functional due to design failures. Therefore, before defining the whole data transfer process, it must be ensured that PCB fulfills its requirements. This can be simply used to call single functions inside current OBC API implementation, for example *turn Processing board on* and waiting for correct response. After that, the overall data transfer process can be defined and explicit interfaces designed until the end on whole system level.

4.5 Continuous integration and continuous delivery

Continuous Integration (CI) and continuous delivery (CD) refer to software development approaches, where developers integrate new software to shared a repository in short cycles. The new software is then verified with automated builds, automated tests and lastly deployed to production. The whole CI/CD process can be defined as the automated pipeline from writing code to new working software with increased development velocity.

The key benefit of using CI/CD is that developers can detect possible errors more frequently and fix them more easily. If the integrated new code change is small, the

defecting change can be pinpointed quickly. [73] In addition, coherent automated pipeline removes manual work while improving code quality [29]. CI and CD have developed the best practices of modern software development while they allow developers move faster and keep the high quality of standards with code [41] [66].

While CI and CD are closely tied together, they are slightly different terms. *Continuous Integration* refers to practice of integrating changes from different developers to mainline, possibly even several times a day. This means code does not divert greatly between different developers. *Continuous Delivery* itself refers to practice to keep codebase deployable at any point. Therefore, after automated tests, software can be configured and it is ready for deployment. Alongside continuous delivery, there is also a term *continuous deployment* which differs from continuous delivery. In continuous deployment, a deployment phase is also performed automatically. In this case, when we refer to CI/CD pipeline, we talk about continuous integration and continuous delivery.

While CI/CD pipeline is easy to implement in SaaS, problems emerge in embedded systems, such as spacecraft, especially on hardware. Even if software can easily be implemented on shared repository with the same practices as presented above, the presence of hardware arises difficulties. The most significant problem is related to testing phase, which is discussed in Chapter 4.6. In addition, "deployment" is not achieved similarly in SaaS as the final product operates in space. The common conventions were pointed out in Chapter 2.3.2. In addition, when PCBs are considered stable in the late development, the usefulness of CI/CD drops after new changes are not introduced the same way.

4.5.1 Automation tools

Jenkins is an open source automation server tool to provide CI and CD functionality. It allows building pipelines with different "jobs", which handle single phases of the pipeline. Jenkins functionality can easily be expanded with different plugins. Besides executing different jobs, Jenkins also monitors all executions and provides information of different executed phases for developers [74].

With Jenkins, it is possible to trigger automatic builds when a new change is commited to main branch. When software has been built, and on success status, all required checks, such as code quality and unit tests, are performed, and last, the final verified software build is deployed. Jenkins also provides concurrent builds connected to different slave machines, which for example have different toolchains to build embedded software and other systems to build ICEYE OS (GNU/Linux). All these slaves are connected to main server and Jenkins handles collaboration and merging of software.

Jenkins itself does not build or test anything, but handles the execution of different tools. Therefore, in order to build software, build server should have the corresponding building tools available which are configured to Jenkins. In addition, in case of code quality checks, tools for checking code quality must be configured in as a Jenkins build. For software tests, a testing framework is used and executed and files generated and reported. Most of the Jenkins plugins only allows easier integration of these tools, but the actual tools must have been installed on the server.

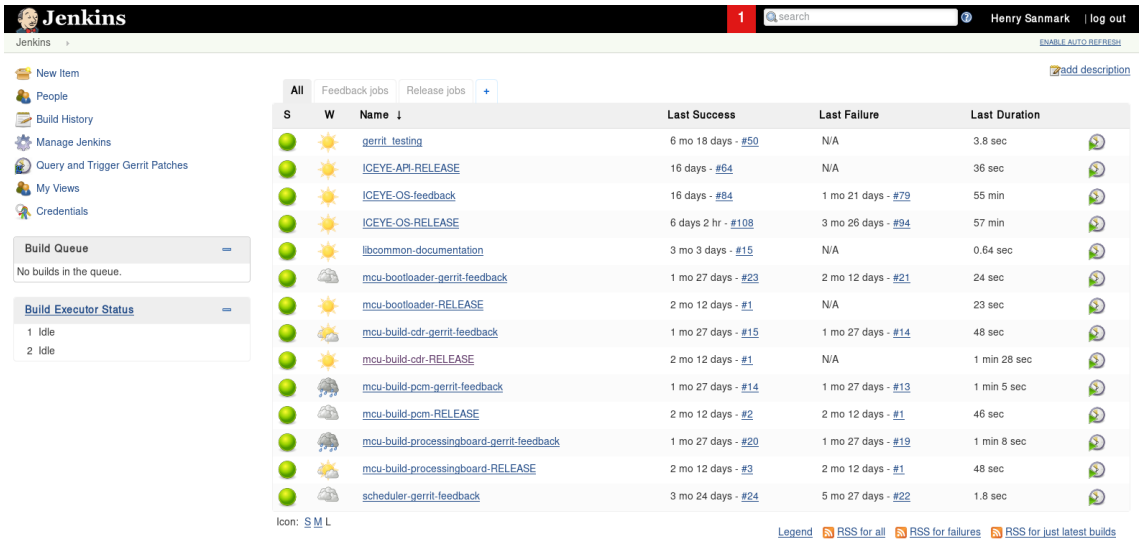

Figure 24: *The overview of Jenkins main window. The main window presents all jobs which are handled by Jenkins and their latest status. This view can be expanded with multiple plugins and other customization.*

4.5.2 CI/CD pipeline

ICEYE uses Jenkins for their CI/CD pipeline. Different jobs are used to build either ICEYE OS or MCU software. Jenkins is hosted at AWS (Amazon Web Services) which is considered as the master, which handles all job executions. Jenkins is connected to Gerrit. Jenkins is configured so that it triggers new builds when new changes are merged to repository main branch after the review process. Jenkins reports the feedback from the newest changes so developers can track if the they build and pass all postprocessing steps. After successful build and postprocessing steps, built software is published in a Jenkins Artifactory, which is a storage of latest succesful builds.

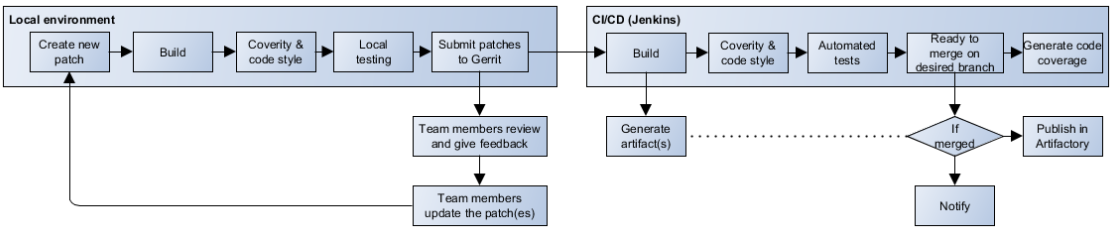

Figure 25: *ICEYE CI/CD pipeline. Pipeline presents how software is developed locally, and after review process with Gerrit, it is handled by Jenkins where automated jobs are executed and reported to the developers.*

There are two different build types in use called *feedback* and *RELEASE*. Feedback builds are considered as the build types which are triggered automatically when a new patch is implemented to main branch, and the results of the reports of latest build are reported. Release-builds are triggered manually when the newest software is considered

stable, and as the latest working version of software.

When analysing the whole pipeline, it can be noticed that all methods and tools which are presented in this thesis, are avalable in that pipeline which can be automated. The current Jenkins pipeline at ICEYE is defined only to the building phase and for ICEYE OS building, and does not take into account MCU software building, flashing and system level testing. Therefore, the proposed whole Jenkins pipeline can be summarized with the following:

1 Create a new piece of software.
2 Push software to Gerrit, and wait until it has been reviewed and therefore verified
3 Jenkins gets the trigger from Gerrit, and starts building the software using CMake configurations
4 After software is succesfully built, code integrity is verified and software analysed
5 Software is flashed to the HW
6 Run required system level tests to verify the integrity of software
7 Report the current status

This proposed pipeline is, however, optimistic considering the embedded software development. Because most of the PCBs are developed in-house and software is considered unstable in the development environment, it might require supervision so that hardware does not fragment in case of anomalies, which can be stopped, for example, by unplugging the cables. This was one of the fundamental problems during the analysis of automated tools inside ICEYE, because hardware related unstable executions could lead to massive damage.

In this case, some modification is required for the presented optimal workflow which ensures the safety during the development, even if it might decrease the development velocity. The CI/CD pipeline shall be divided into two sections:

1 **Building phase**, where new software is built and code verified after every approved review. (Phases 1-4)
2 **Testing phase**, which is triggered either manually or by verified scheduled times. In this case, software is flashed to PCBs and all required tests are run. The idea is that this phase is triggered only if testers can supervise it in case of anomalies, even if it is executed automatically. (Phases 5-7)

Especially Phase 5 should be investigated. Currently this method is done manually which also ensures that PCB flashing is done correctly. This does not, however, provide automatic system level tests during the new patches, and therefore, automated tests should only be executed after the new software is flashed to PCB. It is also possible to also automate this phase, but it was not perfomed in the scope of this thesis and could also lead to potential hardware related problems in case of anomalies.

Using the proposed approach with Jenkins alongside other activities presented in this thesis, and based on the earlier researches considering the benefits of CI/CD pipeline [3] [6] [29] [41] [66] [75], it is encouraged to improve the current Jenkins setups with proposed changes and taking full advantage of Jenkins jobs.

4.6 Mission critical system testing

Standards and common conventions promote testing to verify the functionality of the spacecraft. It must be ensured that all requirements are fulfilled and validation/verification steps are met. [76] As it was presented in Chapter 3.3, at ICEYE system level testing is used instead of unit level testing. This is because of limited resources and focusing on MVP.

The same approach is also used in software testing. Based on Preliminary Design Review from 2016 and Critical Design Review from 2017 [49] [77], it was emphasized that software needs unit testing. Overall in Preliminary Design Review, it was stated that testing should be focused more on, even during the active development process. The whole testing plan was absent. However, in Critical Design Review the preliminary testing plan was declared as well as approaches for it, and it was accepted that system level testing would be enough, even if unit level testing was encouraged but not mandatory in terms of current design scope.

Software system level tests are designed to be part of the continuous delivery pipeline. In this case, after a new software is built, tests are executed automatically inside Jenkins - see Figure 25 in Chapter 4.5. for more details on the pipeline. Alongside the automatic test execution, also manual testing is performed when required. Test cases are based on the current software and mission requirements analysis and HW/SW design. However, when designing the implementation of tests, two main questions were present:

1 How is it possible to integrate automatic tests to CI/CD workflow when tests are executed on real hardware?

2 How to execute test cases safely on hardware, especially when people are not present?

When testing software, hardware is also tested. Therefore, when in terms of this thesis, we talk about "software testing" it must be kept in mind that all software runs on real hardware. It is possible to run software on simulations, which is commonly used in multiple spacecraft projects [12, pp. 256] [76], but in terms of custom designed software and hardware, building a separate hardware simulator alongside constantly developing piece of hardware would take too much resources. Even if resources are limited in developing the satellite, the verification and validation requirements are not smaller, which results in significant share of total resources to be assigned into testing. This should be considered when calculating total resources of development. [76]. In cases where both real hardware and simulator can be developed concurrently, it might be encouraged to double-verify the design. However, in this case, a real hardware is used in testing.

The most significant is phase is to define, how these tests can also be implemented during the simulated missions. Thus, when hardware and software are developed far enough, the mission is simulated using appropriate simulators with simulated telemetry data and such, where automated sequences are run separately. Therefore the tests are not part of the CI/CD workflow, but instead works as real life scenarios on-flight, such as: "if Processing board turns suddenly off, does FDIR handle the system correctly"?

For question 1 the solution is presented in Chapter 4.6.1, which covers the tools required for the testing purposes and how to technically implement the testing procedure. The second question and testing arrangements are discussed in Chapter 4.6.2.

4.6.1 System level testing tools

For system level automatic testing, a test framework called Robot Framework is used. Robot Framework defines itself as following:

Robot Framework is a generic test automation framework for acceptance testing and acceptance test-driven development (ATDD). It has easy-to-use tabular test data syntax and it utilizes the keyword-driven testing approach. Its testing capabilities can be extended by test libraries implemented either with Python or Java, and users can create new higher-level keywords from existing ones using the same syntax that is used for creating test cases. [78]

The architecture of the Robot Framework in Figure 26 presents the basic approach for building tests. The modular architecture allows using Robot Framework in multiple different systems. Being also an open source project written with Python, there are only minimal limitations to run tests on modern computers which can also run Python.

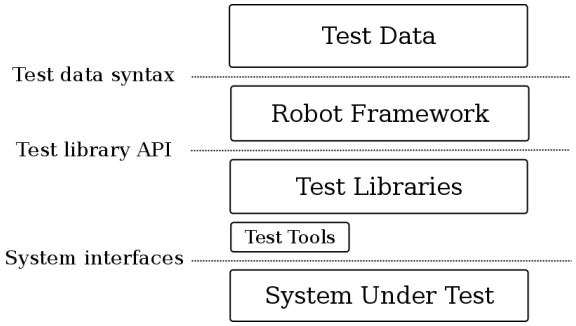

Figure 26: *A modular Robot Framework functional architecture, extracted from Robot Framework site. This Figure presents how Robot Framework is located inside the whole system and how system under testing is handled. [78]*

Writing test cases with Robot Framework is built to be as straightforward as possible. Clear syntax allows writing test cases almost as a "plain text" which also provides easier implementation to write a test case based on a functional story. After that, custom keywords can also be declared which themselves work with target platform's libraries and runs the function execution. In Figures 27 and 28, the basic syntax of test file alongside the user defined keywords and variables are presented. The example files are derived from Robot Framework WebDemo [79].

Robot Framework also provides human readable output of test results in command-line form or in generated logging files. In command line, the overall results of test cases are shown but in HTML-based logging files it is possible to follow every single detail of the test runs. The example outputs are presented in following output text and in Figure 29.

```
*** Settings ***
Documentation     A test suite with a single test for valid login.
...
...               This test has a workflow that is created using keywords in
...               the imported resource file.
Resource          resource.robot

*** Test Cases ***
Valid Login
    Open Browser To Login Page
    Input Username    demo
    Input Password    mode
    Submit Credentials
    Welcome Page Should Be Open
    [Teardown]    Close Browser

```

Figure 27: *A simple test case implemented in Robot Framework. This test case tests if user can login to page with corrent login and username.*

```
*** Settings ***
Documentation     A resource file with reusable keywords and variables.
...
...               The system specific keywords created here form our own
...               domain specific language. They utilize keywords provided
...               by the imported Selenium2Library.
Library           Selenium2Library

*** Variables ***
${SERVER}         localhost:7272
${BROWSER}        Firefox
${DELAY}          0
${VALID USER}     demo
${VALID PASSWORD}    mode
${LOGIN URL}      http://${SERVER}/
${WELCOME URL}    http://${SERVER}/welcome.html
${ERROR URL}      http://${SERVER}/error.html

*** Keywords ***
Open Browser To Login Page
    Open Browser    ${LOGIN URL}    ${BROWSER}
    Maximize Browser Window
    Set Selenium Speed    ${DELAY}
    Login Page Should Be Open

Login Page Should Be Open
    Title Should Be    Login Page

Go To Login Page
    Go To    ${LOGIN URL}
    Login Page Should Be Open

Input Username
    [Arguments]    ${username}
    Input Text    username_field    ${username}

Input Password
    [Arguments]    ${password}
    Input Text    password_field    ${password}

Submit Credentials
    Click Button    login_button

Welcome Page Should Be Open
    Location Should Be    ${WELCOME URL}
    Title Should Be    Welcome Page
```

Figure 28: *Custom keywords and variables, which are used in the test case above. Note that Selenium2Library is used to implement these custom keywords.*

Listing 1: *Shell output of sample Robot Framework test run.*

```
==============================================================================
Test Example :: Test suite for functional tests
==============================================================================
Hello World :: This test case presents the same functionality as h... | PASS |
------------------------------------------------------------------------------
Test Example :: Test suite for functional tests                       | PASS |
1 critical test, 1 passed, 0 failed
1 test total, 1 passed, 0 failed
==============================================================================
Output:  /some/path/iceyesw/tools/functional_tests/output.xml
Log:     /some/path/iceyesw/tools/functional_tests/log.html
Report:  /some/path/iceyesw/tools/functional_tests/report.html
```

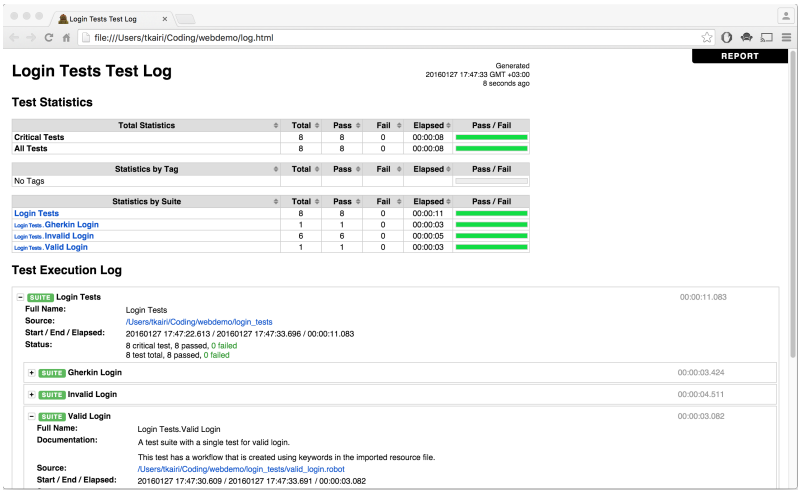

Figure 29: *An example log file of executed test, which is generated in HTML form after a finished test execution.*

Robot Framework is powerful in the way that it can be used for multiple purposes and for different systems. Only the required libraries should be included in the test library to write the tests. On OBC, the ICEYE OS is GNU/Linux based distribution with Python support and a working OBC API which allows executing payload functions from Python libraries. Even if running tests technically on ICEYE OS is possible, it is not as straightforward than in regular desktop environments. Therefore, when designing system level tests at ICEYE, three requirements were defined in software team meetings considering the capabilities of the framework and restrictions of platform.

Requirement 1: Test must be executed remotely, and test results shall be stored in test PC instead of OBC filesystem. **Requirement 2:** Tests shall not import any of the libraries locally. All libraries and functions under test are located remotely. **Requirement 3:** Tests can be run on multiple computers inside the network.

OBC and ICEYE OS are part of the whole satellite system, which should be kept as simple as possible. This is the base for requirement 1. Therefore installing any unnecessary tools which are not required in actual mission is discouraged. In addition, test result logfiles should also be stored in central place, which is not the system under testing, promoting the

ability to log history, easier access for all testers to read results. In addition, data is not lost if target platform breaks.

Robot Framework supports a library called `Remote`, which allows executing tests in another system where the Robot Framework itself is located. This could have been used in ICEYE test system, if this would not require installing an additional remote server on OBC, which is not allowed based on the description above.

Requirement 2 means that the run OBC API is located inside OBC, and it should not have any duplicates inside test computer. Therefore, it is not possible to remotely download libraries and then include them into a test suite, or concurrently update the same library file inside the test computer and OBC. This does not meet the actual execution of tests, and therefore, could lead to misleading results when library is located in another place than it actually should. Therefore, the test must ensure that OBC API and other required libraries are used completely remotely.

Requirement 3 allows us to run test sequences manually when needed just by typing the test command on your local development PC or running test automatically as, for example, Jenkins job inside the whole pipeline. However, it was declared that the computer inside the testing laboratory is mainly used for manual execution even if it is possible from every single computer.

Based on these requirements, a different approach than a regular use of Robot Framework was essential in order to use the benefits of Robot Framework in system level testing. Therefore, instead of using `Remote` library, tests are based on `SSHLibrary` which allows using SSH (Secure Shell) connection inside Robot Framework tests. Because ICEYE OS is GNU/Linux distribution and its development is dependent on SSH connection, SSH is included in the software. After connecting to OBC over SSH, which in this case is a radio connection, a Python shell is opened and every required command is passed to OBC inside Python shell. The returned values are stored as variables are read from Python shell, and then reported back to Robot Framework machine which reads the actual result. Briefly, the system is the following:

1 Run test suite on local test machine

2 Robot Framework connects to OBC using SSHlibrary

3 After connected, Python shell is opened and Python imports all required libraries inside the shell during the session

4 Every RF keyword is wrapped as a Python shell interaction with command writing and reading values from stdout/stderr

5 Test results are generated on local machine, and SSH connection is closed after tests have run

The Robot Framework files, which includes all resource files and actual test cases, have a guideline at ICEYE which was written during the thesis. The guideline is based on common conventions of writing Robot Framework tests, and therefore file structure shall be the following:

- test_suite.robot

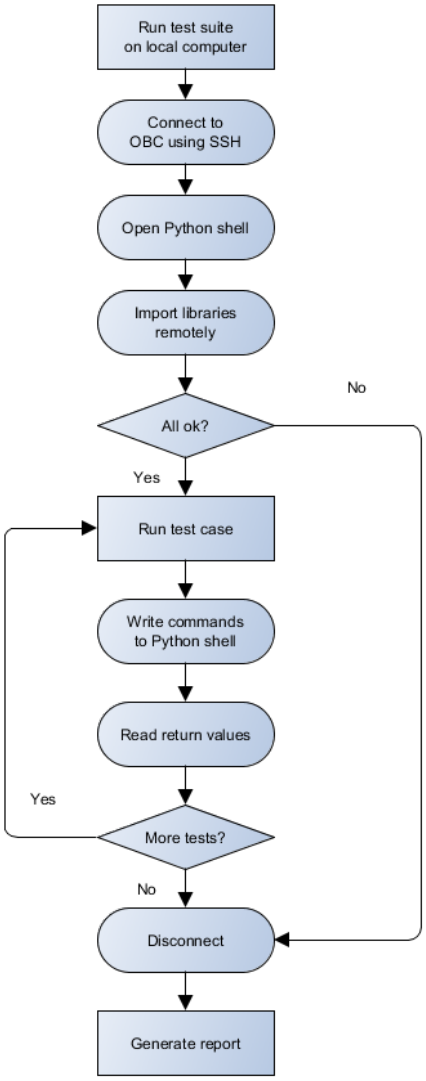

Figure 30: *The example test execution flow in ICEYE software using Robot Framework. This principle is used in multiple different tests, which differ significantly from the traditional approach.*

  - The test suite, which contains all test cases with simple keywords
  - Defines the test structure at the most abstract layer, and is easily readable even for non-developers
  - Multiple different test suite files exists for different testing procedures, such as imaging sequences.

- test_keywords.robot
  - Collection of keywords which wraps the Python shell commands
  - User defined keywords
  - Instead of wrapping directly OBC API keywords, this contains a single functionality required for test case

  - Test suite uses these keywords on its functionality.
- methods.robot
  - Almost the same as `test_keywords.robot`
  - One core file which contains wrappers for the single OBC API methods
- resources.robot
  - The core level of tests, which imports all required dependencies and introduces core keywords
  - Alongside suite and test case setups and teardowns, defines "Execute" and "Verify" functions which are the core wrapper functions for writing commands over SSH
  - The most complex part of the test suite

Alongside tests which are constructed in this way, it was also required that it is possible to run test Python scripts which are located on-board. These are, for example, scripts which are used in test procedures but also part of the final flight software. In these test files, it is only possible to get the overall success or failure status of the script without the detailed information, which is printed to shell output. Therefore, these test files

1 Run test suite on your local test machine
2 Robot Framework connects to OBC using SSHlibrary
3 After connected, locate to folder where the required script is
4 Run the script directly from command line
5 Read until the script has ended, and verify that it has not reported errors
6 Report results to Robot Framework

Using both of these methods, it is possible to either verify that a set of PCBs is responsive by calling their corresponding methods, as a health check, or run a desired sequence which is required to be tested.

4.6.2 Test plan

In Chapter 4.2, ATDD approach was promoted and a technical way to implement test cases was presented. While currently ATDD is not introduced in ICEYE software development, the preliminary work around it has been developed and system level tests written. Two different automated tests were developed during this thesis which are written with Robot Framework:

1 Healthcheck testing
2 Sequential system level testing
  - Imaging test
  - Downlinking test

Healthcheck used to check automatically that the hardware is responsive. This means, that using OBC API, every subsystem should be turned on and off, subsystems respond to test case ping commands and otherwise check that subsystems are responsive and if subsystems generate errors or not. This test case provides simple and fast healthchecks but does not track the actual functionality of subsystems. This one can be used in HW/SW development phase, where abstract level of *synchronization* is defined to ensure that a set of functionality is provided.

Sequential system level testing, such as imaging test and downlinking test, are examples of system level testing which was introduced in earlier Chapters. It follows the actual sequence on how software and hardware should perform and generate the results which testers can read from RF reports. Currently, the sequential system level testing is only implemented on imaging and downlinking sequence, which were developed during the thesis research. The same approach can be used in all other subsystems and system level sequences. It should be noted that the presented test plan is also developed for laboratory environment and does not completely fulfill the requirements of in-flight but rather provides automated data in development phase. Next, the test plan for imaging and downlinking sequence inside laboratory enviroment is presented.

When a tester wants to declare which parameters are used in a specific test case, a file `parameters.robot` is written which contains all test parameters. All test cases themselves are written as *test templates*. This allows required tests to be data-driven meaning that same test case can be run with different set of parameters. Thus, the same test sequence can be run automatically multiple times with different data. Test sequence is divided into two sections, imaging and downlinking, which both act as their own tests. Jenkins, however, handles both of these jobs in correct order as upstream and downstream jobs. While imaging test is executed, after its success the downlinking is executed. The sequence itself works as following:

1 With given parameters, initialize an imaging phase. This includes turning payloads on and configuring mandatory files.

2 Acquire an image and update current configuration.

3 Re-initialize the imaging with new parameters, meaning that a new picture shall be taken with different settings.

4 Acquire the new image and update configuration.

6 Shutdown the imaging phase, where payloads are shut down and image data is transferred to correct location.

7 Prepare system for downlinking with provided parameters, which turns on and conFigures required payloads.

8 Before downlinking, turn on the XBand downconventer in laboratory from the web interface so image data can be received and analysed. This phase uses `Selenium` to automate navigating web pages. This phase could also be done manually by pressing a button in a laboratory, but in case of automation, this can be achieved by web interface.

9 Start recording the XBand radio, so downlinked image is actually received. This phase is also performed from web interface using Selenium.

9 Downlink image by sending image data from RAM to XBand Radio.

10 Stop recording from web interface.

11 Turn off XBand downconventer from web interface.

12 Wait until laboratory system has processed the received file(s) and it appears on local web drive.

13 Verify that file data is correct and no errors are occured.

With simple test steps, it is possible to track which phase of the test did not work and report possible failures accordingly to help developers to find possible bugs in the code. In addition, data-driven tests are required while same sequence is used multiple times but with different settings which must be tested. The same approach can also be used in other system level sequences, and test "user story" is possible to be written in simple English. An example of the test files is attached as Appendix B, which presents how imaging phase is written. The actual downlinking phase uses the same approach, with additional keywords for other systems.

Even if the presented test plan is defined afterwards, it shows how tests should be defined first and software written afterwards in terms of ATDD principles. Based on hardware and software requirements, the test "user story" shall be defined the same way as described above. After that, the code shall be developed based on that sequence and validated through successful testing. With automated tests, this can be applied to the overall pipeline.

While the presented example of downlinking sequence is considered in the case performed in laboratory environment within the automated sequence inside CI/CD, this same approach can also be used in earlier defined mission simulations, where automated sequences are executed and supervised during the simulated mission. Therefore, when system has a simulation running, the test case is executed and results are reported to the tester.

5 Discussion

An approach of this thesis was to analyse multiple development phases and their correlation to each other. We have presented the phase-specific analysis in each Chapter and provided the basic information to understand motivation for these solutions. To understand the effect of analysis, every phase and their integration to each other shall be analysed which can be wrapped up as a improved development practices.

5.1 Integration of tools

We have presented multiple different tools during this thesis. To justify their usage, Table 7, based on the listing from Chapter 4.3, is constructed to present all tools in development and to verify their role. It should be ensured that the presented tools do not collide with each other or do not slow down any presented phase in pipeline. Most of the analytics is based on the current status of the development, and most of these tools were already in practice, so last question about *How long does it take to apply the tool in development?* is answered based on documentation.

As seen in the table, most of the current development tools are used in *create* phase which was derived from DevOps practices. This was based on previous analysis of the characteristics of embedded software development, where, for example, deployment must be considered differently and phases such as monitoring are used with hardware-related analytic tools rather than commonly used software metrics which were left out in this thesis.

Based on the analysis and the choice of tools, software development benefits from agile practices even if it must choose several compromises on how the tool is used in terms of embedded systems. This means that the whole process cannot be analysed purely from the software perspective, as hardware related development pays a major role and consists of completely different practices. Therefore, if we want to test software, for example, it must be done in terms of hardware requirements, and therefore, all practices cannot be executed the same way as in SaaS applications.

The detailed analysis of the hardware related development tools was left out being a completely different approach. For example, some events happening during flight cannot be tested purely from software perspective. This means that all PCBs must be connected to functional GS with mission simulator and analytics tools which are their own projects and not inside software development framework. This would be possible if there were comprehensive simulations of all electronics available, but in terms of resources, this cannot be achieved.

The one significant finding of this thesis was the importance of review process. It is highly promoted in other fields of technology and especially required during the mission-critical software development. ICEYE practices the actual reviewing between small number of team members, but the actual reviewing tool Gerrit is not properly used. While the tools in use are considered as a state-of-the-art, they cannot fix all problems of the process if there are not enough resources. Therefore, Gerrit should be considered in the future when the number of developers and reviewers increases but currently reviewing practices should be performed with discussion between team members. However, there is a potential danger

Table 7: *Results of integration of tools in ICEYE software development.*

	Eclipse-based IDE	**CMake**	**Git**	**Gerrit**	**Repo**	**Jenkins**	**Robot Framework**
In what development phase the tool is used?	Create	Create	Create	Create	Create	Create, Verify, Staging, Release, Configure	Verify
How the tool is used, who uses it?	Developing MCU software, and providing all toolchains for cross-compiling. Developers.	Building MCU software in CI pipeline with automatically triggered builds. Developers and CI.	Version control. Developers.	Code review. Developers are team leaders.	Manage multiple repositories, which are different projects for eg. different subsystems. Developers.	Automating the whole pipeline and providing CI/CD. Developers, team leaders, testers and system engineers.	System level testing. Developers, testers and system engineers.
What does it improve in development?	Toolchains and debugging with JTAG is mandatory, but building and writing code be outsourced	Allows building MCU software within CI	Allows version control, branching, distributed environments and pull requests	Allows reviewing code before merging it to the master branch, to peer-review patches for potential defects and design issues.	Allows controlling multiple repositories, which are divided into different subprojects such as subsystems codebase, thus removing the hassle between projects and improving integration of the system.	Allows integration of tools and automates the process.	Developers to automate the testing procedures and verifying the functional requirements are met. Also provides an easy way to execute sequences of software using OBC API.
What additional costs it provides to developers and company	Free software, requires accepting licence. Setting up is mandatory for development.	Open source software. Writing and maintaining CMake files requires additional work hours.	Open source software. Requires maintaining repository management server and knowledge from developers on how to use it.	Open source software. Requires constant review process and therefore provides only benefits if properly performed, meaning larger costs per work hour, depending on the amount and size of the commits.	Open source software, requires the usage of git and knowhow of the developers.	Open source software, costs with work hours by maintaining the jobs and Jenkins server (AWS)	Open source software, costs are relevant to training the developers for writing RF tests and thinking the requirements as test cases.
How long does it take to apply the tool in development?	1 hour for every developer	Initial setup required 15 workdays, mantaining CMakeFiles up to 1 hour per change.	Hours, but maintaining repositories have taken multiple days.	Hours, but maintaining projects have taken multiple days.	Hours.	Around 30 minutes, but maintaining individual jobs requires additional work depending on the job.	5 minutes of installing requirements, 30 workdays to develop the custom wrapper for OBC API.

of code ownership which should be monitored during the development by following the company guidelines and common conventions of software design.

The other finding was that integration should always be considered. During the thesis work, the MCU software building was rewritten with CMake which directly provided implementation of build process in Jenkins servers which did not exist before. In addition, using Robot Framework, it is possible to integrate testing sequences to CI which were executed manually beforehand. With relatively easily applied open source tools, they do not include enormous hours for setting up and use (see Table 7) and provide fast feedback and therefore improved software quality.

The findings considering the tools can be summed as following:

- Mission-critical software development benefits from the latest tools known from common industry.
- The key principle considering the tools is how they integrate with each other, meaning how they can be applied in the development pipeline.
- Every tool must be justified. It should not be applied to the development environment if it is not used properly.
- Favor open source tools, if possible. Their integration with each other is more guaranteed and do not introduce huge license fees or additional workhours to get interfaces working. Use proprietary software only if mandatory and if it clearly provides significant benefits.
- Always use tools, which can be applied in different environments without additional work. For example, building software with CMake instead of IDE's *Build* function is encouraged in order to verify it works on every developer's machine and CI servers with similar settings and prevents so called "works on my machine" cases.
- Always consider the choice of tools from hardware development perspective.
- Beyond all choice of tools and their integration, the culture of software development pays a major role.

5.2 Workflow and practices

Even if the right choice of tools provides efficient way to improve software development, the major role is in cultural change of the development and sophisticated workflow between development phases. As stated, DevOps principles provide efficient base for the development and operations, but in terms of embedded software development, all phases cannot be justified and compromises should be made.

Starting from the requirements definition, it should be defined how mission-critical definitions are formalized so it can be specified efficiently for hardware and software. In this case, ATDD was introduced as a base for HW/SW requirements which are derived from mission and systel level requirements. ATDD is also considered as a base for later automated system level testing procedures, which promotes fast feedback and therefore higher quality software. When defining hardware and software specific requirements, they should be defined as interface models which themselves consist of 5 different abstraction

Table 8: *Remaining DevOps toolchain phases in terms of embedded software development at ICEYE.*

Release	Flash new piece of software to laboratory environment PCBs, or if possible, in simulation. In this phase, the newest RELEASE-version of software is put into a system where the actual mission is executed. For example, all hardware are running the laboratory environment inside the FlatSat where functionality and integration of all PCBs are tested. Currently this is done manually by updating the built software to the newest hardware, but should be also considered that even the flashing phase could be automated, where CI/CD pipeline also covers the Continuous Deployment capabilities. Constructing this system was, however, out of the scope of this thesis.
Configure	FlatSat configuration, meaning verifying that all systems are powered up and functional. Also the possible mission simulation environment with simulated flybys and telemetry data is configured.
Monitor	During the mission simulation, all parsed telemetry from simulated mission is monitored with corresponding tools and data is gathered.

levels. Then, software is locally developed and verified if possible before submitting to Gerrit. When submitting a patch to Gerrit, the piece of code is reviewed. Upon acceptance, automatic feedback builds are triggered and code coverage and style is verified with, for example, *Cppcheck*. When verified, software is "released", meaning it is deployed to satellite ETB (Electrical Test Bench), commonly known as FlatSat (see "Release" in Table 8). Then, system level tests are executed with Robot Framework where the test cases are derived from requirements as "user stories". During all automated phases, notifications of execution are published in readable form. This can be achieved with, for example, Slack team collaboration tool [80] or mailing lists. The workflow can be expressed as following:

1. Mission operations specification and requirements
2. System specification and architecture
3. ATDD requirements
4. Abstract HW and SW interface models
5. Software design
6. Local testing
7. Submit to Git and/or Gerrit
8. Review process
9. Automated builds (CMake and others)
10. Code coverage and style check
11. Publish in Artifactory
12. Software release in FlatSat
13. Automated system level testing

Beyond that, DevOps "release", "configure" and "monitor" phases should be handled differently. It should be noted that due to spacecraft mission, the only way to gather enough data from both software and hardware is to simulate the actual mission after the release. Thus, it is preferred that the operations listed in Table 8 are performed.

While the workflow analysis was based on DevOps principles, it is clear that the workflow itself contains elements from traditional waterfall principle presented in Chapter 2.3. This flow can also be viewed as a waterfall point of view, where all phases are performed after the previous phase has ended. Waterfall is often preferred in mission-critical software design because of its robustness even though it lacks development velocity. However, if the development workflow is considered with the base of DevOps model, agile principles can be applied to these phases.

DevOps principles are used as an approach with the cultural change and promoting the choice of tools, emphasizing the continuous integration and automation. Then, all presented phases are agile themselves and should be built with small iterations. This means that, starting from the requirements, these are defined from top-level with short iterations, and the software is built based on the current definition. SW and HW interfaces are also defined with short iterations with abstraction levels. When it comes to actual development phase, more traditional agile principles in software are used to ensure that it is ready when a new PCB arrives. Then, system level testing in FlatSat can be performed and simulations run.

In case of defects, it is easy to go back to one phase and fix, for example, requirements or interfaces between HW and SW. The main idea is to get feedback from the working system as fast as possible, which is possible with clearly defined steps in development and automated feedback.

The most significant obstacle considering the workflow is the cultural change that software and hardware developers meet their common goals and stick to development plan with commonly agreed conventions regarding interfaces and design. While software is faster to change based on feedback than hardware, this model should also provide fast feedback to HW designers even if the new board is under development and software is already working with the latest changes.

6 Conclusions

The purpose of this thesis was to analyse mission-critical spacecraft software development, especially in terms of limited resources and tight development schedule. This was achieved by analysing the characteristics of spacecraft software and modern software development practices. Then, this information was reflected to ICEYE project which was used as a case-study and the in-depth analysis was performed in requirements definition, development tools, hardware and software codesign, continuous integration and testing practices.

First, the traditional characteristics of OBC and payload software was analysed. It was indicated that specific software development consists of multiple standards which promotes the role of robustness and fault tolerance. In addition, the overall software structure and architecture was presented and commonly used design practices introduced. Two possible architectures, static and dynamic, were presented for OBC, and for payloads a possible architecture approach called OSRA-P was introduced. In addition, an in-depth analysis of FDIR functionality was performed in terms of mission-criticalness.

In software development practices, the traditional waterfall principle was briefly presented. Waterfall is commonly used in spacecraft software development due to its ensured robust result, but lacks in flexibility could lead to complexity overload and encourages late integration. Modern agile practices were introduced as a solution for problems of waterfall, where especially test-driven development, early incremental code production and pair programming was emphasized as key solutions. Also DevOps practices were introduced which promotes automation, fast feedback and measurements in cultural change. However, several aspects of DevOps are not suitable for mission-critical embedded software because of differences from SaaS applications, and therefore, some compromises must be made especially in software deployment and measurements.

ICEYE software development itself is based on limited resources and tight schedule, where software and other systems are developed in short iterations in order to get MVP which meets the DevOps principles. Therefore, considering DevOps was taken as a base for software development analysis and previously presented limitations and architectural challenges were promoted.

Analysing requirements definition, it was noted that especially system level requirements should be trackable and plans for changing requirements should be defined. From that point of view and from lessons learned in agile studies, ATDD approach is promoted as a base for software development. In that practice, requirements are defined from mission perspective and then with several iterations specified in more details, where test cases are also defined and then later used in system level testing.

In case of choice of tools, modern software development tools such as Git, Gerrit and Repo were presented. While Git, alongside with Repo, are commonly encouraged in software development, and this was endorsed with multiple studies, the main focus was included in review phase analysis with Gerrit. The analysis showed that review process is highly encouraged, but in terms of ICEYE development using Gerrit as a reviewing tool lacks proper usage. Therefore, it is adviced to promote reviewing between single team members in the office and consider Gerrit only when number of developers increases in the future.

Considering the build process, the design of current ICEYE software building was

analysed and a new system for building embedded software was designed with proper toolchain, which can be integrated to the automated pipeline, as suggested in DevOps practices. Additionally the building ARM software was presented to get an understanding of the process. It was also suggested that open source tools should be used.

For hardware and software codesign, the fundamental problems considering information sharing and design cycles were outlined. The solution to improve this was to design hardware and software in different abstraction levels, where hardware and software interface models are generated based on ATDD practices and those itself from current system specifications.

DevOps practices embraces automation, and therefore CI/CD practices were introduced and latest trends relating to it analysed. All previous phases were integrated to CI pipeline, but the system was divided into 2 different phases: building and testing. This was essential because the lattest phases of DevOps toolchain are not directly compatible with hardware restrictions of system. Therefore, the build pipeline was automated, but automated tests are triggered only when hardware is verified and PCBs flashed with latest binaries. This could also be done automatically, but it was not in the scope of this thesis.

In system level testing, an acceptance testing framework called Robot Framework was used and its functionality presented. The designed test structure was presented, which used ICEYE OBC API and SSH connection, which enabled running tests remotely without additional servers installed on the OBC. In addition, a test plan with healthchecks and automated functional sequences was presented, which were derived from mission specification, but limited to laboratory environment. However, also testing during the simulated mission was briefly presented as a final testing practice which uses the same principle as automated tests in CI/CD workflow.

Last, it was noted even if waterfall is considered the robust approach for software development, using a DevOps-derived model with agile practices leads to functional results, which consist of short iterations with tight milestones. To achieve this, cultural changes to software and hardware developers should be considered so that proposed actions can be actualized.

When we analyse the original research problem *"How to improve embedded software development in mission critical satellite systems, when robustness and fast development cycle in the small "NewSpace" company must be taken into consideration?"*, we can sum the results as following:

> *To improve embedded software development in mission critical satellite systems with limited resources, use acceptance test-driven approaches with the trackable requirements from system level. This can be achieved with continuous integration, justified integration of open source tools, system level testing and interface driven approach with abstraction layers in hardware and software codesign. The final system should also be tested in real functional scenarios, which occurs during the actual mission. All these phases during the whole workflow should favor agile practices from requirements definition to system level testing.*

6.1 Suggestions for future work

In addition to proposed changes, several alternative thoughts should also be considered for improved software development.

- This thesis analysed the software development in multiple aspects to improve the overall quality and development velocity. However, every single phase of the development should be analysed separately with unique researches, which was not possible in terms of thesis scope and schedule.
- Automated scenarios for mission simulations should be constructed. This means that those system level tests which were introduced with Robot Framework, should also be included in simulation as real cases. The current design allows executing tests in laboratory environment and with scheduled sequences, but when the whole operation is functional, tests should be executed as random encounters and therefore testing the real capabilities of spacecraft.
- DevOps model for mission-critical software development requires additional research. Even if model was widely used in this thesis and used as a base for analysis, this thesis should not be considered as a complete DevOps guide for embedded software development. It is also possible to suggest an alternative models which takes into account the flaws of DevOps model in this field of industry.

6.2 Final thoughts

The proposed changes in software development which were based on literature reviews and in-house research, alongside the developed systems improves the software development. However, because of tight schedule and the size of this thesis, all results and developed systems did not provide enough final data to be analysed. This was mainly because some of these proposals were not taken into action yet, and the final results can be analysed only in an environment which takes all proposed changes in action and have been run for months, or even years. The designed systems, such as MCU software building and system level testing with Robot Framework are, however, currently part of the development workflow.

While this thesis was mainly considered as a list of proposals to improve software development at ICEYE, listed changes are completely usable in other companies and researches. Therefore, my personal thoughts about this thesis is that this should be considered as a starting point or reference for embedded software development in NewSpace applications. Even if many listed proposals require more research, the results can be used as a base for better software development practices.

References

[1] K. Könnölä, S. Suomi, T. Mäkilä, V. Rantala, and T. Lehtonen, "Can embedded space system development benefit from agile practices", *EURASPI Journal on Embedded Systems*, vol. 3, 2017. DOI: 10.1186/s13639-016-0040-z.

[2] K. Könnölä, S. Suomi, T. Mäkilä, V. Rantala, and T. Lehtonen, *Sulautettujen järjestelmien ketterän käsikirjan lisäosa: Ketteryys avaruusteollisuudessa*. University of Turku, Technology Research Center; Tekes, Oct. 2005, Digital publication, ISBN: 978-951-29-6283-9.

[3] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "DevOps", *IEEE Software*, vol. 33, no. 3, pp. 94–100, Jun. 2016, doi:10.1109/MS.2016.68, ISSN: 0740-7459.

[4] E. Ahmad, B. Raza, R. Feldt, and T. Nordebäck, "ECSS Standard Compliant Agile Software Development - An Industrial Case Study", in *In Proceedings of the National Conference for Software Engineering (NSEC 2010)*, 2010. [Online]. Available: http://www.cse.chalmers.se/~feldt/publications/ahmad_2010_nsec.html.

[5] B. Graaf, M. Lormans, and H. Toetenel, "Embedded software engineering: the state of the practice", *IEEE Software*, vol. 20, no. 6, pp. 61–69, Jul. 2003. DOI: 10.1109/MS.2003.1241368.

[6] J. Engblom, "Continuous Integration for Embedded Systems using Simulation", Wind River, Embedded World Congress, Nürnberg, Germany, Feb. 2015.

[7] K. Fowler, *Mission-critical and safety-critical systems handbook: Design and development for embedded applications*. Sharfus Draid, Inc., Amsterdam, Elsevier, 2010, ISBN: 978-0-7506-8567-2.

[8] J. Axelsson, E. Papatheocharous, and J. Andersson, "Characteristics of software ecosystems for Federated Embedded Systems: A case study", *Information and Software Technology, Special issue on Software Ecosystems*, vol. 56, no. 11, pp. 1457–1475, Nov. 2014.

[9] *On-board computers / On-board data handling*, http://www.esa.int/Our_Activities/Space_Engineering_Technology/Onboard_Computer_and_Data_Handling/Onboard_Computers, web page, accessed: 08.01.2017.

[10] *On-board software | ESC Aerospace*, http://www.esc-aerospace.com/?page_id=460, web page, accessed: 08.01.2017.

[11] M. Jones *et al.*, *Introducing ECSS Software-Engineering Standards within ESA, Practical approaches for space- and ground-segment software*, ECSS software-engineering standards, bulletin 111, Aug. 2002.

[12] J. Eickhoff, *Onboard Computers, Onboard Software and Satellite Operations: An Introduction*. Springer Heidelberg, Dordrecht, London, New York, 2012, ISBN: 978-3-642-25169-6.

[13] "ECSS-E-70-41A: Ground systems and operations - Telemetry and telecommand packet utilization", ESA Publications Division, ESTEC, P.O. Box 299, 2200 AG Noordwijk, The Netherlands, European Space Agency, ECSS, 2003.

[14] C. Honvault and G. Furano, *Towards a HW/SW reference architecture for payloads*, Material slides, European Space Research and Technology Centre (ESTEC), Noordwijk ZH, The Netherlands, Oct. 2014.

[15] I. Xabier, V. Balaji, O. Emre, and D. Shidhartha, "A Triple Core Lock-Step (TCLS) ARM Cortex-R5 Processor for Safety-Critical and Ultra-Reliable Applications", *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*, no. 46, pp. 246–249, Jun. 2016.

[16] C. Brand, "The Development of an ARM-based OBC for a Nanosatellite", Master's thesis, University of Stellenbosch, Department of Electrical Engineering, 2007.

[17] G. Dreijer, "The Evaluation of an ARM-based On-Board Computer For a Low Earth Orbit Satellite", Master's thesis, University of Stellenbosch, Dec. 2002.

[18] A. Kestilä, T. Tikka, P. Peitso, J. Rantanen, A. Näsilä, K. Nordling, H. Saari, R. Vainio, P. Janhunen, J. Praks, and M. Hallikainen, "Aalto-1 nanosatellite - technical description and mission objectives", *Geoscientific Instrumentation, Methods and Data Systems*, vol. 2, no. 1, pp. 121–130, 2013. DOI: 10.5194/gi-2-121-2013. [Online]. Available: http://www.geosci-instrum-method-data-syst.net/2/121/2013/.

[19] E. Razzaghi, A. Yanes, J. Praks, and M. Hallikainen, "Design of a reliable OBC for Aalto-1 nanosatellite mission", in *Proceedings of the 2nd IAA Conference on University Satellites Missions and CubeSat Workshop*, Feb. 2013, pp. 447–460.

[20] V. Pisacane, *Fundamentals of Space Systems*. Oxford University Press, Inc., New York, 2005, ISBN: 0-19-516205-6.

[21] P. Fortescue, G. Swinerd, and J. Stark, *Spacecraft Systems Engineering*, 4th ed. A John Wiley and Sons, Ltd., 2011, ISBN: 978-111-99710-1-6.

[22] V. Bos and A. Trcka, *On-board Software Reference Architecture for Payloads*, Contract No: 4000110034/13/NL/LvH, The Software Systems Division (TEC-SW) and Data Systems Division (TEC-ED) Final Presentation Days, ESA/ESTEC, The Netherlands, Dec. 2015.

[23] *Fault-Detection, Fault-Isolation and Recovery (FDIR) Techniques - Utilize FDIR Design Techniques to provide for Safe and Maintainable On-Orbit Systems - Technique DFE-7*, Technique DFE-7, NASA engineering practices.

[24] A. Guiotto, A. Martelli, and C. Paccagnini, "SMART-FDIR: use of Artificial Intelligence in the implementation of a Satellite FDIR", Alenia Spazio S.p.A., Software & Simulation Architectures, Tech. Rep., 2003.

[25] N. Holsti and M. Paakko, *Towards Advanced FDIR Components*, Technical report, 2001.

[26] *ESA - Software engineering and Standardisation- Fault Detection, Isolation and Recovery*, http://www.esa.int/TEC/Software_engineering_and_standardisation/TEC4WBUXBQE_0.html, Accessed: 10.01.2017.

[27] "ECSS-Q-ST-80C: Space product assurance - Software product assurance", ESA Publications Division, ESTEC, P.O. Box 299, 2200 AG Noordwijk, The Netherlands, European Space Agency, ECSS, 2009.

[28] "C and C++ Coding Standards", ESA Board for Software, Standardisation and Control, Tech. Rep., Mar. 2000.

[29] N. Seth and R. Khare, "ACI (Automated Continuous Integration) using Jenkins: Key for Successful Embedded Software Development", *2015 RAECS UIET Panjab University Chandigarh*, Dec. 2015, ISSN: 978-1-4673-8253-3/15.

[30] "ECSS-E-ST-10-02C: Space engineering - Verification", ESA-ESTEC, Requirements & Standards Division, Noordwijk, The Netherlands, Tech. Rep., Mar. 2009.

[31] S. H. VanderLeest and A. Butler, "Escape The Waterfall: Agile for Aerospace", *28th Digital Avionics Systems Conference*, 2009, ISSN: 978-1-4244-4078-8.

[32] C. Larman, "Agile & Iterative Development: A Manager's Guide", Addison-Wesley, Tech. Rep., 2004, pp. 58–62.

[33] "Project AGILE", Tech. Rep., 2007, http://www.agile-itea.org/public/papers/ITEA-AGILE-results_oct-07.pdf.

[34] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, *Manifesto for Agile Software Development*, 2001. [Online]. Available: http://www.agilemanifesto.org/.

[35] K. Sureshchandra and J. Shrinivasavadhani, "Adopting Agile in Distributed Development", *2008 IEEE International Conference on Global Software Engineering*, ISSN: 978-0-7695-3280-6/08.

[36] O. Salo and P. Abrahamsson, "Agile methods in European embedded software development organisations: a survey on the actual use and usefulness of Extreme Programming and Scrum", *IET Software*, vol. 2, no. 1, pp. 58–64, Feb. 2008, doi:10.1049/iet-sen:20070038.

[37] V. Rantala, K. Könnölä, S. Suomi, and M. Isomäki, "Guidelines For Small Embedded System Companies Aiming to Enter The Space Industry", Tech. Rep. 7, Aug. 2015.

[38] D. Nicoll, "Use of Agile Techniques in the Development of a Safety-Critical Rail Application", Agile Business Conference, London, Tech. Rep., Sep. 2008.

[39] M. Hüttermann, *DevOps for Developers*, 1st. Berkely, CA, USA: Apress, 2012, ISBN: 1430245697, 9781430245698.

[40] L. E. Lwakatare *et al.*, "Towards DevOps in the Embedded Systems Domain: Why is It so Hard?", *49th Hawaii International Conference on System Sciences*, 2016, ISSN: 1530-1605/16.

[41] D. Isla, "Interplanetary DevOps at NASA JPL", presentation slides, Boston, MA: USENIX Association, 2016.

[42] *ICEYE Ltd. official website*, http://iceye.fi, accessed 10th January 2017.

[43] "ICEYE official technical documents", internal pages, 2017.

[44] "ICEYE OBC specification", internal documentation, 2017.

[45] *The Buildroot User Manual*, https://buildroot.org/downloads/manual/manual.html, The Buildroot manual is written by the Buildroot developers. It is licensed under the GNU General Public License, version 2. Refer to the COPYING file in the Buildroot sources for the full text of this license.

[46] "ICEYE OS", internal documentation, 2017.

[47] "ICEYE MCU Software reference", internal documentation, 2017.

[48] W. Vesely, *Fault Tree Handbook with Aerospace Applications*, version 1.1, NASA publication, NASA Office of Safety and Mission Assurance, 2002.

[49] "Critical Design Review (CDR) meeting notes", internal notes, Jan. 2017.

[50] *NASA Lessons learned: Systems Requirement Flow*, https://llis.nasa.gov/lesson/1298, lesson number 1289, 2002.

[51] *NASA Lessons learned: Software Traceability*, https://llis.nasa.gov/lesson/18802, lesson number 18802, 2016.

[52] *NASA Lessons learned: Requirements Development*, https://llis.nasa.gov/lesson/1591, lesson number 1591, 2005.

[53] E. Hendrickson, *Driving Development with Tests: ATDD and TDD*, http://testobsessed.com/wp-content/uploads/2011/04/atddexample.pdf, STANZ 2008 and STARWest 2008, 2008.

[54] C. Manual and C. M. Teixeira, "Towards DevOps: Practices and Patterns from the Portuguese Startup Scene", Doctoral dissertation, University of Porto, Sep. 2016.

[55] A. Yau and C. Murphy, "Is a Rigorous Agile Methodology the Best Development Strategy for Small Scale Tech Startups?", Tech. Rep., Jan. 2013, University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-13-01.

[56] *Benefits of Using Open Source Software*, http://open-source.gbdirect.co.uk/migration/benefit.html, accessed 20th March 2017.

[57] J. Loeliger *et al.*, *Version Control with Git*, 1st ed., A. Oram, Ed. 1005 Gravenstein Highway North, Sebastopol: O'Reilly Media, Inc., May 2009, ISBN: 978-0-596-52012-0.

[58] *The Value And Importance Of Code Reviews*, A Forrester Consulting Thought Leadership Paper Comissioned By Klocwork, Mar. 2010.

[59] D. Spinellis, "Git", May 2012, ISSN: 0740-7459/12.

[60] *Why Git for your organization*, https://www.atlassian.com/git/tutorials/why-git, Atlassian Corporation Plc, accessed 5th April 2017.

[61] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The Impact of Code Review Coverage and Code Review Participation on Software Quality", *MSR'14*, May 2014, ISSN: 978-1-4503-2863-0/14/05.

[62] A. Bacchelli and C. Bird, "Expectations, Outcomes, and Challenges of Modern Code Review", *ICSE 2013, San Francisco, CA, USA*, 2013, ISSN: 978-1-4673-3076-3/13.

[63] *Repo command reference*, https://source.android.com/source/using-repo.html, accessed 20th March 2017.

[64] *Code, test and deploy together with GitLab open source git repo management software*, https://about.gitlab.com/, accessed 29th March 2017.

[65] S. Rifenbark, *Yocto Project Mega Manual*, http://www.yoctoproject.org/docs/2.2.1/mega-manual/mega-manual.html, revision 2.2.1, 2017.

[66] K. Baltzer and D. Büchi, *Continuous Integration for Embedded Systems*, presentation slides, 2010.

[67] *CMake Reference Documentation*, https://cmake.org/cmake/help/v3.8/, accessed 20th March 2017.

[68] *ARM Optimizing C/C++ Compiler*, www.ti.com/lit/pdf/spnu151, Dec. 2016.

[69] A. A. Jerraya and W. Wolf, "Hardware/Software Interface Codesign for Embedded Systems", Feb. 2005, ISSN: 0018-9162/05.

[70] *Imperas*, http://www.imperas.com/, accessed 19th April 2017.

[71] *SystemC*, http://accellera.org/downloads/standards/systemc, accessed 19th April 2017.

[72] *Wind River SimicS*, https://www.windriver.com/products/simics/simics/_po/_0520.pdf, product overview paper.

[73] H. L. Akshaya, S. N. Jagadish, J. Vidya, and K. Veena, "A Basic Introduction to DevOps Tools", *International Journal of Computer Science and Information Technologies*, vol. 6, no. 3, 2015, ISSN: 0985-9646.

[74] *Jenkins Documentation*, https://jenkins.io/doc/, accessed 20th March 2017.

[75] K. Hirvikoski, "Advances in Streamlining Software Delivery on the Web and its Relations to Embedded Systems", Master's thesis, University of Helsinki, Department of Computer Science, Apr. 2015.

[76] A. J. Stephen, "Survey of Verification and Validation Techniques for Small Satellite Software Development", Long Beach, CA, Tech. Rep., May 2015.

[77] "Preliminary Design Review (PDR) meeting notes", internal notes, May 2016.

[78] *Robot Framework User Guide v.3.0.2*. http://robotframework.org/robotframework/3.0.2./RobotFrameworkUserGuide.html.

[79] *Robot Framework Web Demo*, https://bitbucket.org/robotframework/webdemo, repository accessed 20th March 2017.

[80] *Slack: Where work happens*, https://slack.com/, accessed 13th March 2017.

A Appendix A: CMake files for MCU software

Main file

```
cmake_minimum_required(VERSION 2.8.12)

set(CMAKE_SYSTEM_NAME Generic)

set(CMAKE_C_COMPILER $ENV{TI_HERCULES_COMPILER}/armcl)
set(CMAKE_ASM_COMPILER $ENV{TI_HERCULES_COMPILER}/armcl)
set(CMAKE_AR $ENV{TI_HERCULES_COMPILER}/armar)

# We skip compiler and ABI tests due to the use of TI compiler
set(CMAKE_ASM_COMPILER_WORKS 1)
set(CMAKE_C_COMPILER_WORKS   1)
set(CMAKE_ASM_ABI_COMPILED 1)
set(CMAKE_C_ABI_COMPILED 1)

project (processingboard C ASM)

# Same flags that CCSv6 uses
set(CMAKE_C_FLAGS "-mv7R4 --code_state=32 --float_support=VFPv3D16 --abi=eabi
 -g --diag_wrap=off --display_error_number --diag_warning=225 -i
  $ENV{TI_HERCULES_COMPILER_LIB}  -i $ENV{TI_HERCULES_COMPILER_INCLUDE}")

# We use the same flags for assembly code
set(CMAKE_ASM_FLAGS "${CMAKE_C_FLAGS}")

# Set runtime directory to "bin"
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "bin/")

# Application-dependant directories and their subdirectories

# FreeRTOS
include_directories(freertos
  freertos/include
  freertos/source)
add_subdirectory(
  freertos/source
)

# commonLib
include_directories(lib
  lib/generic
  lib/generic/libADC
  lib/generic/libDummy
  lib/generic/libFDIR
  lib/generic/libFuncTab
  lib/generic/libGPIO
  lib/generic/libI2C
  lib/generic/libInclude
  lib/generic/libInterface/libInterfaceCAN
  lib/generic/libTelecommand
  lib/generic/libTelemetry
  lib/generic/libUtil
  lib/generic/libVarTab
)
add_subdirectory(lib/generic/libADC)
add_subdirectory(lib/generic/libDummy)
add_subdirectory(lib/generic/libFDIR)
add_subdirectory(lib/generic/libFuncTab)
add_subdirectory(lib/generic/libGPIO)
add_subdirectory(lib/generic/libI2C)
add_subdirectory(lib/generic/libInterface/libInterfaceCAN)
add_subdirectory(lib/generic/libTelecommand)
add_subdirectory(lib/generic/libTelemetry)
```

```
add_subdirectory(lib/generic/libVarTab)

# Application
include_directories(app
  app/application
  app/application/initialization
  app/application/interface
  app/application/meassure
  app/application/norMemory
  app/application/services
  app/application/spiAnlg
  app/application/telecommand
  app/application/telemetry
  app/codeHalcoGen
  app/include
  app/targetConfigs
)
add_subdirectory(app/application)
add_subdirectory(app/application/initialization)
add_subdirectory(app/application/interface)
add_subdirectory(app/application/meassure)
add_subdirectory(app/application/norMemory)
add_subdirectory(app/application/services)
add_subdirectory(app/application/spiAnlg)
add_subdirectory(app/application/telecommand)
add_subdirectory(app/codeHalcoGen)

# Define libraries to be built
add_library(freertosLib
  $<TARGET_OBJECTS:freertosSource>
)
SET_TARGET_PROPERTIES(freertosLib PROPERTIES PREFIX "")

add_library(commonLib
  $<TARGET_OBJECTS:libADC>
  $<TARGET_OBJECTS:libDummy>
  $<TARGET_OBJECTS:libFDIR>
  $<TARGET_OBJECTS:libFuncTab>
  $<TARGET_OBJECTS:libGPIO>
  $<TARGET_OBJECTS:libI2C>
  $<TARGET_OBJECTS:libInterfaceCAN>
  $<TARGET_OBJECTS:libTelecommand>
  $<TARGET_OBJECTS:libTelemetry>
  $<TARGET_OBJECTS:libVarTab>
)
SET_TARGET_PROPERTIES(commonLib PROPERTIES PREFIX "")

add_executable(processingboard
  $<TARGET_OBJECTS:application>
  $<TARGET_OBJECTS:initialization>
  $<TARGET_OBJECTS:interface>
  $<TARGET_OBJECTS:meassure>
  $<TARGET_OBJECTS:norMemory>
  $<TARGET_OBJECTS:services>
  $<TARGET_OBJECTS:spiAnlg>
  $<TARGET_OBJECTS:telecommand>
  $<TARGET_OBJECTS:codeHalcoGen>
)

target_link_libraries(freertosLib
 $ENV{TI_HERCULES_COMPILER_LIB}/rtsv7R4_T_be_v3D16_eabi.lib)
target_link_libraries(commonLib freertosLib)
target_link_libraries(processingboard commonLib)

set(CMAKE_C_LINK_EXECUTABLE "<CMAKE_C_COMPILER> --run_linker
 --output_file=<TARGET> --map_file=<TARGET>.map <CMAKE_C_LINK_FLAGS>
 <OBJECTS> <LINK_LIBRARIES> <LINK_FLAGS>")
```

Subfolder file

```
set (initializationSources
  initCAN.c
  initialization.c
  initializationFuncTab.c
  initializationTelecommand.c
  initRS485.c
  initVarTab.c
)

add_library(initialization OBJECT ${initializationSources})
SET_TARGET_PROPERTIES(initialization PROPERTIES PREFIX "")
```

B Appendix B: Robot Framework Imaging tests

Test suite

```
*** Settings ***
Documentation           Imaging functional test

Resource                imaging_keywords.robot
Resource                parameters.robot

Suite Setup             Open Connection And Log In
Suite Teardown          Close all connections
Test Setup              Start Shell And Import Libs
Test Teardown           Quit Shell


*** Test Cases ***

Single imaging sequence
    [Template]      Single imaging ${parameter_list}
    [Tags]          single
    ${SINGLE_TEST_1}

Full imaging sequence
    [Template]      Full imaging ${parameter_list}
    [Tags]          full
    ${FULL_TEST_2}

*** Keywords ***

# Test case templates

Single imaging ${parameter_list}

    Set Parameters For Single Imaging Test  ${parameter_list}

    Initialize imaging  ${idInput}  ${startAddress}  ${filename}  ${idSettings}  ${
        ↪ idFpga}  ${gainVGA}  ${idChainDivider}
    Acquire Image       ${start_time_100us}
    Shutdown Imaging

Full imaging ${parameter_list}

    Set Parameters For Full Imaging Test    ${parameter_list}

    Initialize imaging      ${idInput_1}  ${startAddress_1}  ${filename_1}  ${
        ↪ idSettings_1}  ${idFpga_1}  ${gainVGA_1}  ${idChainDivider_1}
    Acquire Image       ${start_time_100us_1}
    Re-initialize Imaging   ${idInput_2}  ${startAddress_2}  ${filename_2}  ${
        ↪ idSettings_2}  ${idFpga_2}  ${gainVGA_2}  ${idChainDivider_2}
    Acquire Image       ${start_time_100us_2}
    Shutdown Imaging

# Actual imaging sequences

Initialize imaging
    [Documentation]     This test tests if initialization is correct
    [Arguments]         ${idInput}  ${startAddress}  ${filename}  ${idSettings}  ${
        ↪ idFpga}  ${gainVGA}  ${idChainDivider}

    Initialize initializePayload    ${idInput}  ${startAddress}  ${filename}  ${
        ↪ idSettings}  ${idFpga}  ${gainVGA}  ${idChainDivider}

    Make sure ADCS is on
    Turn On Tranceiver
    Turn On Processing Board
```

```
    Initialize Processing Board
    Transfer Input File To FPGA                 ${IMAGING}
    Set Gates
    Get Output File Size
    Turn CDR Lite On Imaging
    Configure The Divider Board
    Turn On CDRs
    Create Config File                          ${IMAGING}

Acquire Image
    [Documentation]     Verify image acquiring
    [Arguments]         ${start_time_100us}

    Initialize acquireImage     ${start_time_100us}
    Imaging Step
    Update Current Imaging Configuration
    Update Configuration File
    Check RAM Progress

Shutdown Imaging
    [Documentation]     Shutdown payload after a successful imaging sequence

    Initialize shutdownPayload
    Power Down CDR Lite                         ${SHUTDOWN}
    Turn Off Tranceiver                         ${SHUTDOWN}
    Get Configuration File Info
    Send Image Metadata To Processing Board     ${SHUTDOWN}
    Transfer Data From RAM to Flash             ${SHUTDOWN}
    Create Image Data
    Power Down Processing Board                 ${SHUTDOWN}

Re-initialize Imaging
    [Documentation]     Reinitialize imaging if want to take more images within
        ↪ timeframe
    [Arguments]         ${idInput}  ${startAddress}  ${filename}  ${idSettings}  ${
        ↪ idFpga}  ${gainVGA}  ${idChainDivider}

    Initialize re-initialization    ${idInput}  ${startAddress}  ${filename}  ${
        ↪ idSettings}  ${idFpga}  ${gainVGA}  ${idChainDivider}
    Update Image Catalogue
    Send Image Metadata To Processingboard      ${REINIT}
    Transfer Data From RAM To Flash             ${REINIT}
    Write Image Catalogue
    Reinitialize Processing Board
    Transfer Input File To FPGA                 ${REINIT}
    Create Config File                          ${REINIT}

Reset Payload
    [Documentation]     Resets the imaging payload.
    [Arguments]         ${idInput}  ${startAddress}  ${filename}  ${idSettings}  ${
        ↪ idFpga}  ${gainVGA}  ${idChainDivider}

    Initialize resetPayload         ${idInput}  ${startAddress}  ${filename}  ${
        ↪ idSettings}  ${idFpga}  ${gainVGA}  ${idChainDivider}
    Reinitialize Processing Board               ${RESET}
    Transfer Input File To FPGA                 ${RESET}
    Create Configuration File                   ${RESET}

Abort Imaging
    [Documentation]     Aborts the imaging.

    Initialize aborting
    Power Down CDRs
    Power Down CDR Lite                         ${ABORT}
    Turn Off Tranceiver                         ${ABORT}
    Power Down Processing Board                 ${ABORT}

Compile Image Catalogue
```

```
    [Documentation]     Compile Image Catalogue

    Initialize image compiling
    Read Image Catalogue
    Read Flash Blocks
    Genererate Catalogue
    Print Image Catalogue
```

Keywords

```
*** Settings ***
Documentation           Imaging sequence related keywords

Resource                ../resources.robot
Resource                ../methods.robot

Library                 String

*** Keywords ***

############################
# Initialization keywords #
############################

Initialize initializePayload
    [Arguments]         ${idInput}  ${startAddress}  ${filename}  ${idSettings}  ${
        ↪ idFpga}  ${gainVGA}  ${idChainDivider}
    Write               import initializeLibrary
    ${stdout}=          Read                delay=1.0s
    Should not contain  ${stdout}           ImportError
    ${params}=          Catenate    SEPARATOR=,${SPACE}     ${idInput}  ${
        ↪ startAddress}  '${filename}'  ${idSettings}  ${idFpga}  ${gainVGA}  ${
        ↪ idChainDivider}
    ${command}=         Catenate    ${IMAGING}  =   initializeLibrary.
        ↪ initializeLibrary(${params})
    Write               ${command}
    ${stdout}=          Read                delay=1.0s
    Should not contain  ${stdout}           NameError

Initialize acquireImage
    [Arguments]         ${start_time_100us}
    Write               import acquireLibrary
    ${stdout}=          Read                delay=1.0s
    Should not contain  ${stdout}           ImportError

    ${command}=         Catenate    ${ACQUIRE}  =   acquireLibrary.acquireLibrary($
        ↪ {start_time_100us})
    Write               ${command}
    ${stdout}=          Read                delay=1.0s
    Should not contain  ${stdout}           NameError

Initialize shutdownPayload
    Write               import shutdownLibrary
    ${stdout}=          Read                delay=1.0s
    Should not contain  ${stdout}           ImportError

    ${command}=         Catenate    ${SHUTDOWN}  =  shutdownLibrary.shutdownLibrary
        ↪ ()
    Write               ${command}
    ${stdout}=          Read                delay=1.0s
    Should not contain  ${stdout}           NameError

Initialize re-initialization
    [Arguments]         ${idInput}  ${startAddress}  ${filename}  ${idSettings}  ${
        ↪ idFpga}  ${gainVGA}  ${idChainDivider}
```

```
    Write               import reinitializeLibrary
    ${stdout}=          Read                delay=1.0s
    Should not contain  ${stdout}           ImportError
    ${params}=          Catenate    SEPARATOR=,${SPACE}     ${idInput}  ${
        ↪ startAddress}  '${filename}'  ${idSettings}  ${idFpga}  ${gainVGA}  ${
        ↪ idChainDivider}
    ${command}=         Catenate    ${REINIT}   =   reinitializeLibrary.
        ↪ reinitializeLibrary(${params})
    Write               ${command}
    ${stdout}=          Read                delay=1.0s
    Should not contain  ${stdout}           NameError

Initialize resetPayload
    [Arguments]         ${idInput}  ${startAddress}  ${filename}  ${idSettings}  ${
        ↪ idFpga}  ${gainVGA}  ${idChainDivider}
    Write               import resetLibrary
    ${stdout}=          Read                delay=1.0s
    Should not contain  ${stdout}           ImportError
    ${params}=          Catenate    SEPARATOR=,${SPACE}     ${idInput}  ${
        ↪ startAddress}  '${filename}'  ${idSettings}  ${idFpga}  ${gainVGA}  ${
        ↪ idChainDivider}
    ${command}=         Catenate    ${RESET}   =   resetLibrary.resetLibrary(${
        ↪ params})
    Write               ${command}
    ${stdout}=          Read                delay=1.0s
    Should not contain  ${stdout}           NameError

Initialize aborting
    Write               import abortLibrary
    ${stdout}=          Read                delay=1.0s
    Should not contain  ${stdout}           ImportError

    ${command}=         Catenate    ${ABORT}   =   abortLibrary.abortLibrary()
    Write               ${command}
    ${stdout}=          Read                delay=1.0s
    Should not contain  ${stdout}           NameError

Initialize image compiling
    Write               import compileLibrary
    ${stdout}=          Read                delay=1.0s
    Should not contain  ${stdout}           ImportError

    ${command}=         Catenate    ${COMPILE}   =   compileLibrary.compileLibrary
        ↪ ()
    Write               ${command}
    ${stdout}=          Read                delay=1.0s
    Should not contain  ${stdout}           NameError

####################
# Wrapper keywords #
####################

Make sure ADCS is on
    Execute Command     ${IMAGING}          makeSureADCSisOn

Turn On Tranceiver
    Execute Command     ${IMAGING}          turnOnTranceiver

Turn On Processing Board
    Execute Command     ${IMAGING}          turnOnProcessingboard

Initialize Processing Board
    Execute Command     ${IMAGING}          initializeProcessingboard

Transfer Input File To FPGA
    [Arguments]         ${phase}
    Execute Command     ${phase}            transferInputFileToFPGA
```

```
Set Gates
    # Temporary, shall be deleted soon(ish)
    Write                   imaging.setGates()

Get Output File Size
    Execute Command     ${IMAGING}          getOutputFileSize

Turn CDR Lite On Imaging
    Execute Command     ${IMAGING}          turnOnCDRLite

Configure The Divider Board
    Execute Command     ${IMAGING}          configureTheDividerBoard

Turn On CDRs
    Execute Command     ${IMAGING}          turnOnCDRs

Create Config File
    [Arguments]         ${phase}
    Execute Command     ${phase}            createConfigFile

Imaging Step
    Execute Command     ${ACQUIRE}          imagingStep

Update Current Imaging Configuration
    Execute Command     ${ACQUIRE}          updateCurrentImagingConfiguration

Update Configuration File
    Execute Command     ${ACQUIRE}          updateConfigurationFile

Check RAM Progress
    Execute Command     ${ACQUIRE}          checkRAMprogress

Power Down CDR Lite
    [Arguments]         ${phase}
    Execute Command     ${phase}            powerDownCDRLite

Turn Off Tranceiver
    [Arguments]         ${phase}
    Execute Command     ${phase}            turnOffTranceiver

Get Configuration File Info
    Execute Command     ${SHUTDOWN}         getConfigurationFileInfo

Send Image Metadata To Processing Board
    [Arguments]         ${phase}
    Execute Command     ${phase}            sendImageMetadataToProcessingboard

Transfer Data From RAM to Flash
    [Arguments]         ${phase}
    Execute Command     ${phase}            transferDataFromRAMtoFlash

Create Image Data
    Execute Command     ${SHUTDOWN}         createImageData

Power Down Processing Board
    [Arguments]         ${phase}
    Execute Command     ${phase}            powerDownProcessingboard

Update Image Catalogue
    Execute Command     ${REINIT}           updateImageCatalogue

Write Image Catalogue
    Execute Command     ${REINIT}           writeImageCatalogue

Reinitialize Processing Board
    Execute Command     ${REINIT}           reinitializeProcessing

Power Down CDRs
```

```
    Execute Command     ${ABORT}            powerDownCDRs

Read Image Catalogue
    Execute Command     ${COMPILE}          readImageCatalogue

Read Flash Blocks
    Execute Command     ${COMPILE}          readFlashBlocks

Genererate Catalogue
    Execute Command     ${COMPILE}          generateNiceCatalogue

Print Image Catalogue
    Execute Command     ${COMPILE}          printImageCatalogue
```

Parameters

```
# Parameter file for tests
# Using this file, add test parameters which are run inside the test case
# and call the test sequence with the corresponding list
#
# SYNTAX: idInput, startAddress, filename, idSettings, idFpga, gainVGA,
    ↪ idChainDivider,  start_time_100us
# - repeat list how manu imaging sequences are performed
#
# Note that parameters are provided as a STRING, which is then
# parsed to corresponding parameter list inside tests.
#
# When adding new parameter lists, remember to add a corresponding
# test case with corresponding test template. See test suite for details.

*** Variables ***

${SINGLE_TEST_1}    48 1600 auto_sine_0dB 0 0 0 0 10
${FULL_TEST_2}      48 1750 auto_sine_10dB 0 0 10 0 10 41 1900 auto_chirp_10dB 0 0
    ↪ 10 0 10
```

Resources

```
*** Settings ***
Documentation           Resource file for SSH connection and
...                     Python shell interaction. The core functionality of
...                     this test suite.

Library                 SSHLibrary
Library                 String

*** Variables ***
# Default SSH connection parameters, override with command line options
${HOST}                 localhost
${USERNAME}             test
${PASSWORD}             test

# Evaluation values
${ALL_OK}               Success
${ALL_TURNED_OFF}       All off
${ERROR_OCCURED}        Error occured
${NOT_ALLOWED_RANGE}    Not in allowed range
${UNSUPPORTED}          Unsupported filetype

# obcAPI and cdrUtilities instance names
${API_INSTANCE}         api
${CDRUTILS}             cdrUtilities
```

```
# obcAPI classes - subsystems
${BATTERY}              battery
${PCM}                  pcm
${CDR_LITE}             cdrLite
${CDR}                  cdr
${PB}                   pb
${TRANS}                trans
${ADCS}                 adcs
${XBR}                  xbr
${TMY}                  telemetryGroundHandler

# Imaging sequence related variables for libraries
${IMAGING}              imaging
${ACQUIRE}              acquire
${SHUTDOWN}             shutdown
${REINIT}               reinit
${RESET}                reset
${ABORT}                abort
${COMPILE}              comp

# Downlinking sequence related variables for libraries
${DL_PREP}              prep
${DL}                   dl
${DL_END}               end

*** Keywords ***

Open Connection And Log In
    [Documentation]     Opens SSH connection and interactive Python shell
    Open Connection     ${HOST}
    Login               ${USERNAME}         ${PASSWORD}

Start Shell And Import Libs
    Write               export PYTHONIOENCODING=utf-8
    Write               python -i
    ${stdout}=          Read Until          >>>
    Should Contain      ${stdout}           >>>
    Set Client Configuration                prompt=>>>

    # Import required libraries
    Write               import sys
    ${stdout}=          Read                delay=1.0s
    Should not contain  ${stdout}           ImportError

    # Import ICEYE-related libraries
    Write               sys.path.insert(0, '/opt/iceye/lib/api')
    Write               sys.path.insert(0, '/opt/iceye/test_scripts/cdrScripts/')
    Write               sys.path.insert(0, '/opt/iceye/test_scripts/configFiles/')
    Write               sys.path.insert(0, '/opt/iceye/test_scripts/Imaging Scripts
        ↪ /')
    Write               sys.path.insert(0, '/opt/iceye/test_scripts/Imaging Scripts
        ↪ /libs/')
    Write               sys.path.insert(0, '/opt/iceye/test_scripts/Downlink
        ↪ Scripts/')
    Write               sys.path.insert(0, '/opt/iceye/test_scripts/Downlink
        ↪ Scripts/libs/')
    Write               import obcAPI
    ${stdout}=          Read                delay=1.0s
    Should not contain  ${stdout}           ImportError
    Write               ${API_INSTANCE} = obcAPI.obcAPI()
    ${stdout}=          Read                delay=1.0s
    Should contain      ${stdout}           ICEYE OBC API instance created
    #Write               import cdrUtilities
    #${stdout}=          Read                delay=1.0s
    #Should not contain  ${stdout}           ImportError
    #Write               import setCDR
    #${stdout}=          Read                delay=1.0s
```

```
    #Should not contain  ${stdout}           ImportError

Quit Shell
    Write               quit()

Verify value
    [Documentation]     This keyword excepts "dbus.Int32(VALUE)" messages
    [Arguments]         ${result}       ${expected}

    ${subval1}=         Get Lines Containing String     ${result}  dbus.Int32(
    ${subvalue}=        Strip string    ${subval1}
    ${truevalue}=       Set Variable If
    ...                 "${subvalue}" == "dbus.Int32(1)"    ${ALL_OK}
    ...                 "${subvalue}" == "dbus.Int32(10)"   ${ALL_OK}
    ...                 "${subvalue}" == "dbus.Int32(0)"    ${ALL_TURNED_OFF}
    ...                 "${subvalue}" == "dbus.Int32(-1)"   ${ERROR_OCCURED}
    ...                 "${subvalue}" == "dbus.Int32(-2)"    ${NOT_ALLOWED_RANGE}
    ...                 "${subvalue}" == "dbus.Int32(-3)"   ${UNSUPPORTED}

    Run Keyword If      "${truevalue}" == "None"
    ...                 Fail            ERROR: Invalid return value: ${subvalue}

    Should Be Equal     ${truevalue}        ${expected}

Execute
    [Documentation]     Executes function in subsystem and returns its value by
        ↪ reading it directly from Python shell.
    [Arguments]         ${subsystem}    ${cmd}      ${params}=${EMPTY}  ${timeout
        ↪ }=3min

    # Construct the API command in Python way (both subsystem and cdrutil)
    ${command}=         Run Keyword If      '${subsystem}' == '${CDRUTILS}'
    ...             Catenate    SEPARATOR=      ${CDRUTILS}     .   ${cmd}  (  ${params
        ↪ }  )
    ...             ELSE IF     '${subsystem}' == '${IMAGING}'
    ...             Catenate    SEPARATOR=      ${IMAGING}      .   ${cmd}  (  ${params
        ↪ }  )
    ...             ELSE IF     '${subsystem}' == '${ACQUIRE}'
    ...             Catenate    SEPARATOR=      ${ACQUIRE}      .   ${cmd}  (  ${params
        ↪ }  )
    ...             ELSE IF     '${subsystem}' == '${SHUTDOWN}'
    ...             Catenate    SEPARATOR=      ${SHUTDOWN}     .   ${cmd}  (  ${params
        ↪ }  )
    ...             ELSE IF     '${subsystem}' == '${REINIT}'
    ...             Catenate    SEPARATOR=      ${REINIT}       .   ${cmd}  (  ${params
        ↪ }  )
    ...             ELSE IF     '${subsystem}' == '${RESET}'
    ...             Catenate    SEPARATOR=      ${RESET}        .   ${cmd}  (  ${params
        ↪ }  )
    ...             ELSE IF     '${subsystem}' == '${ABORT}'
    ...             Catenate    SEPARATOR=      ${ABORT}        .   ${cmd}  (  ${params
        ↪ }  )
    ...             ELSE IF     '${subsystem}' == '${COMPILE}'
    ...             Catenate    SEPARATOR=      ${COMPILE}      .   ${cmd}  (  ${params
        ↪ }  )
    ...             ELSE IF     '${subsystem}' == '${DL_PREP}'
    ...             Catenate    SEPARATOR=      ${DL_PREP}      .   ${cmd}  (  ${params
        ↪ }  )
    ...             ELSE IF     '${subsystem}' == '${DL}'
    ...             Catenate    SEPARATOR=      ${DL}           .   ${cmd}  (  ${params
        ↪ }  )
    ...             ELSE IF     '${subsystem}' == '${DL_END}'
    ...             Catenate    SEPARATOR=      ${DL_END}       .   ${cmd}  (  ${params
        ↪ }  )
    ...             ELSE
    ...             Catenate    SEPARATOR=      ${API_INSTANCE}     .   ${subsystem}
        ↪     .   ${cmd}  (  ${params}  )
```

```
    ${written1}=        Write               ${command}

    # Check that execution does not rise and error
    ${ex_out}=          Wait Until Keyword Succeeds     ${timeout}    3sec
    ...                 Read Until Prompt
    Should Not Contain  ${ex_out}           Error:
    Should contain      ${ex_out}           dbus.Int32
    ${output}=          Set Variable        ${ex_out}


    # Return the proper output value
    [Return]            ${output}

Execute Command
    [Documentation]     Wrapper keyword for cases where "Execute" and "Verify"
    ...                 are called.
    [Arguments]         ${subsystem}  ${cmd}  ${params}=${EMPTY}  ${timeout}=3min
    ${retval}=          Execute         ${subsystem}  ${cmd}  ${params}  ${timeout}
    Verify Value        ${retval}       ${ALL_OK}

Set Value
    [Documentation]     Almost like "Set Variable", but uses Python shell in this
        ↪ test suite
    [Arguments]         ${variable}         ${retval}

    ${value_cmd}=       Catenate    ${retval}   =   ${variable}
    ${written1}=        Write               ${value_cmd}
    ${written2}=        Write               ${retval}
    ${output}=          Read Until Prompt
    Should Contain      ${variable}   ${output}

# Parameter set functions

Set Parameters For Single Imaging Test
    [Arguments]         ${parameter_list}
    @{params}=          Split string                ${parameter_list}

    Set Test Variable   ${idInput}                  @{params}[0]
    Set Test Variable   ${startAddress}             @{params}[1]
    Set Test Variable   ${filename}                 @{params}[2]
    Set Test Variable   ${idSettings}               @{params}[3]
    Set Test Variable   ${idFpga}                   @{params}[4]
    Set Test Variable   ${gainVGA}                  @{params}[5]
    Set Test Variable   ${idChainDivider}           @{params}[6]
    Set Test Variable   ${start_time_100us}         @{params}[7]

Set Parameters For Full Imaging Test
    [Arguments]         ${parameter_list}
    @{params}=          Split string                ${parameter_list}

    Set Test Variable   ${idInput_1}                @{params}[0]
    Set Test Variable   ${startAddress_1}           @{params}[1]
    Set Test Variable   ${filename_1}               @{params}[2]
    Set Test Variable   ${idSettings_1}             @{params}[3]
    Set Test Variable   ${idFpga_1}                 @{params}[4]
    Set Test Variable   ${gainVGA_1}                @{params}[5]
    Set Test Variable   ${idChainDivider_1}         @{params}[6]
    Set Test Variable   ${start_time_100us_1}       @{params}[7]
    Set Test Variable   ${idInput_2}                @{params}[8]
    Set Test Variable   ${startAddress_2}           @{params}[9]
    Set Test Variable   ${filename_2}               @{params}[10]
    Set Test Variable   ${idSettings_2}             @{params}[11]
    Set Test Variable   ${idFpga_2}                 @{params}[12]
    Set Test Variable   ${gainVGA_2}                @{params}[13]
    Set Test Variable   ${idChainDivider_2}         @{params}[14]
    Set Test Variable   ${start_time_100us_2}       @{params}[15]

Set Parameters For Downlinking
```

```
    [Arguments]          ${parameter_list}
    @{params}=           Split string                ${parameter_list}

    Set Test Variable    ${direct_flash_address}     @{params}[0]
    Set Test Variable    ${image_name}               @{params}[1].icf
    Set Test Variable    ${idFpga}                   @{params}[2]
    Set Test Variable    ${idSettings}               @{params}[3]
    Set Test Variable    ${start_address}            @{params}[4]
    Set Test Variable    ${stop_address}             @{params}[5]
```