Czech Technical University
Faculty of Electrical Engineering
Department of Control Engineering

DIPLOMA THESIS

Flight Software Development for Demise Observation Capsule

Prague, 2017 **Jakub Zamouřil**

Author's Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

.....	.....
Place, date	Signature

esc Aerospace s.r.o. Responsible Person Declaration

I have reviewed the presented work on behalf of esc Aerospace s.r.o. and by signing below I declare that it can be unconditionally released to public in its entirety.

.....	.....
Name	Position
.....	.....
Place, date	Signature

S[&]T b.v. Responsible Person Declaration

I have reviewed the presented work on behalf of S[&]T b.v. and by signing below I declare that it can be unconditionally released to public in its entirety.

.....	.....
Name	Position
.....	.....
Place, date	Signature

Anotace a klíčová slova

Tato práce popisuje proces návrhu letového softwaru pro zařízení kvalifikované pro provoz ve vesmíru, ukazuje jeho vývoj a testování, a poskytuje popis hotového produktu. Letový software popisovaný v této práci byl vyvinut pro projekt Demise Observation Capsule (DOC). DOC je zařízení, které bude instalováno do druhých a vyšších stupňů nosných raket, a slouží k pozorování jejich rozpadu při vstupu do atmosféry na konci jejich mise. Vzledem k omezenému času komunikace v průběhu mise a vysokým nárokům na přenášená data byl také vytvořen vlastní protokol pro přenos dat.

Klíčová slova: letový software, software pro vesmírná zařízení, software pro embedded zařízení, architektura softwaru, návrh softwaru, návrh komunikačního rozhraní, Demise Observation Capsule

Abstract and keywords

This work describes the process of the design of a flight software for a space-qualified device, outlines the development and testing of the SW, and provides a description of the final product. The flight software described in this work has been developed for the project Demise Observation Capsule (DOC). DOC is a device planned to be attached to an upper stage of a launch vehicle and observe its demise during atmospheric re-entry at the end of its mission. Due to constraint on communication time during the mission and the need to maximize the amount of transferred data, a custom communication protocol has been developed.

Keywords: flight software, space-qualified software, safety critical software, embedded software, software architecture, software design, communication protocol design, Demise Observation Capsule

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Control Engineering

DIPLOMA THESIS ASSIGNMENT

Student: **Zamouřil Jakub**

Study programme: Cybernetics and Robotics
Specialisation: Cybernetics and Robotics

Title of Diploma Thesis: **Flight Software Design for Demise Observation Capsule**

Guidelines:

1. Study and describe the general process of flight software design, including needed documents, standards, tools and processes
2. Describe the Demise Observation Capsule mission and goals, and deduce the necessary functions which the software will have to perform
3. From a given set of requirements design flight software architecture and software interfaces
4. Document the software architecture and interfaces, including justification of all crucial decisions
5. Cooperate with your colleagues on development and testing of the software
6. Describe the developed software, results of its testing, and lessons learned

Bibliography/Sources:

[1] ECSS-E-ST-40C SPACE ENGINEERING: SOFTWARE, ECSS standard, 2009.
Koopman, P.: Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks, DSN-2004.
[2] Richard Barry: Mastering the FreeRTOS? Real Time Kernel A Hands On Tutorial Guide, online, http://www.freertos.org/Documentation/RTOS_book.html

Diploma Thesis Supervisor: Ing. Michal Sojka, Ph.D.

Valid until the summer semester 2017/2018

L.S.

prof. Ing. Michael Šebek, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 26, 2017

Acknowledgment

I would like to give thanks to:

Michal Sojka from FEE CTU, my thesis supervisor; for agreeing to supervise the thesis, providing his comments and opinion, and helping to improve the text;

Petr Suchánek from esc Aerospace, my project supervisor; for his tutoring and help with the project itself, as well as support in preparation and writing of this thesis;

Ludo Visser and Trevor Watts from S[&]T; for reading the thesis, providing their comments, and supporting me in the publication of this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The presented work describes a process of software design and development for Demise Observation Capsule project, describing the followed regulation, justifying the design decisions and presenting the result.

The ESA's Demise Observation Capsule (DOC) project aims to improve the understanding of the disintegration process of launch vehicle upper stages during atmospheric re-entry. The DOC is a small device which will be mounted to the host vehicle (HV), collect data and observe the demise of the HV while surviving the atmospheric re-entry, and transmit the collected data to ground segment.

The DOC project was kicked off in 2014. The author of this work joined the project in June 2016, when the project was past preliminary design review and aiming for critical design review (project lifecycle is explained in section 2.5.1). After joining the author took over the software (SW) design and development after previously responsible person and became the lead application software (ASW) developer for the project. The requirements and preliminary design approved by the preliminary design review were developed into a mature version by the author, and the ASW development was started. As this SW project is too large for one person to finish, the SW design was consulted and iterated with author's project supervisor as well as third parties, and the SW development was done in cooperation with the DOC ASW development team at esc Aerospace s.r.o., under author's lead.

A theoretical background on flight software development is given in chapter 2. As general description of software development process is too broad of a topic and is covered by many books already[1][2][3], only a few topics were selected for discussion in chapter 2, mainly describing necessary inputs and tools of the flight software developer, and focusing on differences between development of non-critical software and flight software development.

The Demise Observation Capsule project background, overview of its hardware and interfaces, and requirements on the SW are discussed in chapter 3.

Chapter 4 covers the author's proposed SW architecture. The preliminary architecture which was reviewed during the preliminary design review was expanded and finalized. The chapter describes static and dynamic architecture design, including SW behavior. The design of the

interface between the capsule and the ground segment is also described.

In chapter 5 a description of software development and testing is given. Changes in the design which were made during the software development are also discussed. The SW development and testing was done in cooperation with multiple members of the DOC development team.

Finally, results, assessment of the project, and lessons learned are presented in chapter 6.

As there are many abbreviations used throughout this work, their list is given on page 65.

A secondary goal of this thesis is to summarize the knowledge gained during the work on this project and provide it in a clear and concise form to the reader, so that the text of this work can be used as a training material for software developers new to the aerospace field. Therefore a secondary focus is given to explanation of the theoretical background in chapter 2 in context of the work to be done.

Chapter 2

Flight Software Design Process

The term flight software in the context of this work describes a SW operating on a space-qualified device. A definition and introduction to flight software (FSW) is given in section 2.1.

The work on space projects is, up to a point, standardized. The most important standard for a flight software developer to follow is the ECSS-E-ST-40C (Space Engineering – Software) standard[4], which is developed and maintained by the European Cooperation for Space Standardization (ECSS). The ECSS standards are described in section 2.2.

A flight software developer needs to obtain many inputs from the customer, most prominently the system requirements, description of the hardware and documentation to the hardware or any used SW libraries. These are discussed in section 2.3.

In section 2.4, tools which can be used for FSW development are discussed.

In section 2.5, an overview of a space-qualified SW project lifecycle is given.

Finally, the general process of FSW development is described in section 2.6, including the quality assurance process.

2.1 Flight Software

Every space project needs software to enable the device to perform its mission. A term flight software (FSW) encompasses all different kind of software, which can be present on a space-qualified device; for example boot SW, device drivers, SW on dedicated micro-controllers on HW peripheries, or the application software (ASW), which is the SW controlling the device. This work describes a development of an ASW and its integration into other third-party SW to create a FSW for Demise Observation Capsule.

Flight software (or application software) design process is fundamentally different from what is a usual software design process in the industry. While normally in the industry (when not developing a safety critical software) an emphasis is made on fast and cost-efficient development, flight software needs to follow very strict quality and safety standards, and if it does not meet the required qualification or is not thoroughly tested and bug-free, it will not be accepted by the customer.

In addition to FSW, there is usually a need to develop ground segment SW for communication with the flight segment of the project. Although the ground SW is not as safety critical as the FSW, the ECSS standards (discussed in the next section) also apply to it.

2.2 ECSS standards

To enable effective cooperation on large space projects and address standardization of work on such projects, the European Cooperation for Space Standardization (ECSS) has developed a series of standards providing requirements and specifications for management, engineering and product assurance. For flight software development, there are three most relevant standards:

- ECSS-E-ST-40C (Space Engineering – Software)[4]
- ECSS-Q-ST-80C (Space Product Assurance – Software)[5]
- ECSS-M-ST-10C (Space project management – Project planning and implementation)[6]

with the ECSS-E-ST-40C standard being the most relevant to the work of a software engineer.

These standards are in essence collections of requirements on the flight software development process (or project management in general). The requirements in these standards are given in terms of what shall be accomplished, e.g. the work packages to be performed and documentation to be written; the specification of methods how the necessary work will be performed is then left to the customer. To help the customer and developer in defining the specifics, ESA published a handbook[7], which explains and clarifies the ECSS-E-ST-40C standard rule-by-rule. Due to its comprehensiveness, this handbook is the primary source for this chapter.

2.2.1 ECSS-E-ST-40C Overview

The ECSS-E-ST-40C standard[4] describing space software development is applicable to all SW developed during the project and applies to all segments of a space system, including the ground segment. It also applies to development of SW which is not delivered, but affects the quality of the product. The standard also describes the interfaces of SW development with project management (covered by ECSS-M-ST-10C) and product assurance (covered by ECSS-Q-ST-80C).

Sections 1-3 of the standard give an introduction and overview of used terms and definitions. Section 4 of the standard briefly introduces the framework and organization of the standard and explains each step of the SW development process (the development process is further described in section 2.6 of this work). The standard's section 5 then lists the requirements on each step of the development process, that is:

- SW system requirement process
- SW project management

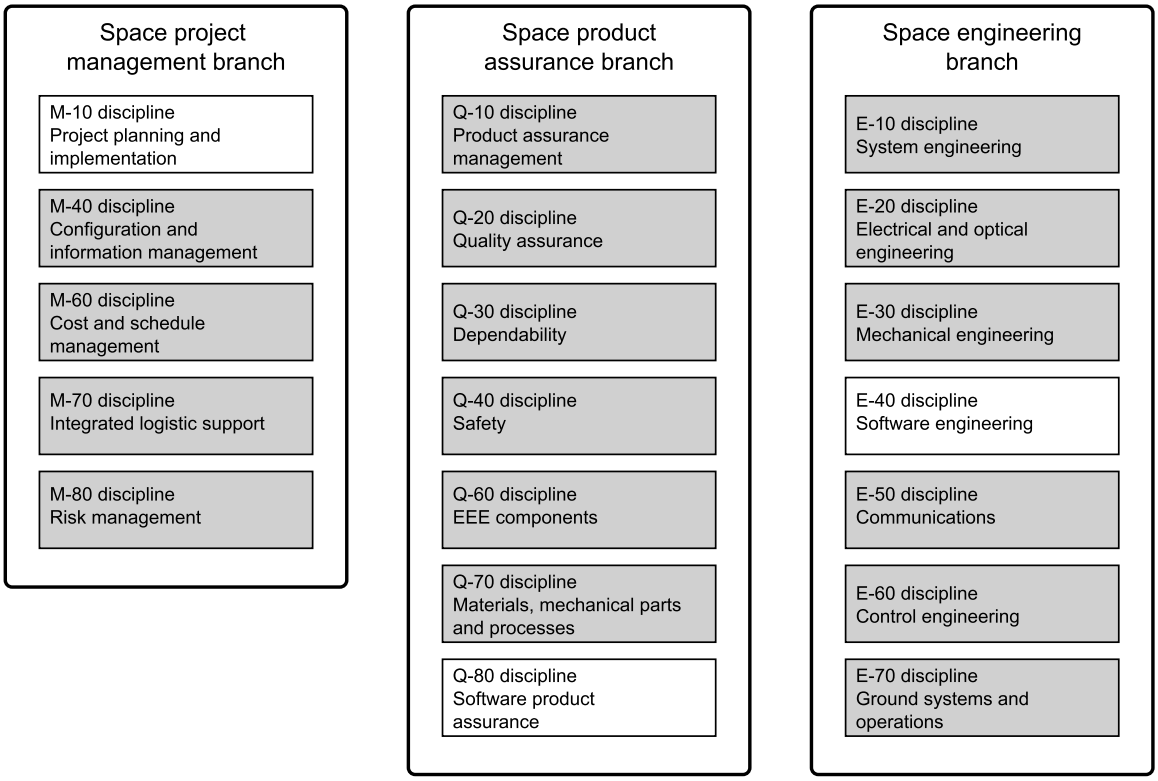

Figure 2.1: ECSS standards relevant for ASW development[7] (standards relevant for ASW development highlighted)

- SW requirements and architecture engineering process
- SW design and implementation process
- SW validation
- SW delivery and acceptance
- SW verification
- SW operation
- SW maintenance.

The standard also contains a number of annexes, which each describe the expected contents of each one document introduced in one of the requirements from standard section 5. The overview of most important expected documentation is given in section 2.6.4 of this work.

2.2.2 Standard Tailoring and Software Criticality

Not all requirements given by the ECSS-E-ST-40C standard have to be implemented – the customer can specify that some of the non-essential requirements can be omitted in the project,

saving time and budget. This practice is known as *standard tailoring* or more precisely *tailoring of the standard to the needs of the project*. The standard itself encourages this practice and in Annex S[4] gives recommendations on its tailoring.

The main tool to help the customer tailor the standard is the concept of *software criticality*, defined by ECSS-Q-ST-80C standard (Space Product Assurance – Software[5] As given by the standard, any flight software can be categorized into one of the four criticality categories denoted by letters A,B,C,D, with A being the highest criticality and D the lowest. Definition of the categories is given in table 2.1.

Category	Definition
A	Software, whose failure can have *catastrophic* consequences
B	Software, whose failure can have *critical* consequences
C	Software, whose failure can have *major* consequences
D	Software, whose failure can have *minor or negligible* consequences

Table 2.1: Definition of software criticality categories[5]

The definition of severity of the consequences of a failure is given below:[5]

1. A failure is classified as *catastrophic* if it can cause any of the following effects:
   - Loss of life, life-threatening or permanent disabling injury or occupational illness
   - Loss of system
   - Loss of an interfacing manned flight system
   - Loss of launch site facilities
   - Severe detrimental environmental effects
2. A failure is classified as *critical* if it can cause any of the following effects:
   - Temporarily disabling but not life-threatening injury, or temporary occupational illness
   - Major damage to interfacing flight system
   - Major damage to ground facilities
   - Major damage to public or private property
   - Major detrimental environmental effects
3. A failure is classified as *major* if it can cause major degradation of the mission, e.g. some of the mission objectives will not be fulfilled
4. A failure is classified as *minor* if it can cause minor degradation of the mission, e.g. all mission objectives will be fulfilled, but some data will be lost

With the software criticality category known, the customer can then use a recommended tailoring of the ECSS-E-ST-40C standard given in its Annex R[4], which can be further modified

to the needs of the project. To give an example of tailoring; for software of criticality A or B the standard recommends no tailoring to be done, e.g. all of the standard requirements and specifications are to be fulfilled. For software of criticality C, the standard recommends omitting some design aspects (e.g. the software logical model) and some documentation (e.g. software behaviour verification or unit testing verification report). For software of criticality D the standard further recommends to omit traceability matrices (see section 2.3.1) and more verification reports.

2.3 Inputs for ASW development

This section lists necessary items for the ASW developer to begin his work.

2.3.1 System requirements

It is a usual practice in programming business that the customer sets requirements on the produced software. This serves to prevent ambiguity and makes any changes to the requirements easily tracked[1]. In flight software development, this practice plays central role to projects and is very closely tracked and scrutinized.

In accordance with the standard[4], four documents need to be produced during lifetime of the project to track the requirements and their fulfillment to a sufficient level. These are:

Software System Specification (SSS), which contains the requirement specification of the customer.

Software Requirement Specification (SRS), which contains more detailed requirements on software derived from the system specification (SSS). This document is usually prepared by the software developer and approved by the customer.

Software Verification Plan (SVerP), which contains a work plan, identifies development risks, and presents a plan of software verification, i.e. how it will be proved that all requirements are fulfilled. This document is usually prepared by management (or PA) of the software developer.

Software Verification Report (SVR), which is the final document produced at the end of software development, and present results of the software verification activities as defined by the SVerP.

From point of view of the ASW developer, the SSS document is the input describing the requested functionality of the SW. The developer then simultaneously designs the overall software architecture, and writes the SRS document, which describes the ASW being produced as a set of requirements. This informs the customer about the structure and operation of the developed ASW, sets up framework for later verification and validation process, and protects the ASW developer from changes in the requirement specification (which then need to be formally

processed and approved as extrawork). The requirement derivation process is further described in section 2.5.3.

To trace the requirements and ensure their fulfillment, the so-called traceability matrices are developed. These matrices are part of design documentation and assign each requirement to the design part (SW module, function, document) which fulfills the requirement. These matrices are subject to a review by the customer.

2.3.2 Hardware

To develop software exactly to the needs of the project, a complete hardware specification needs to be provided to the SW developer. However this is not always the case in large projects, where hardware may be simultaneously developed by other development team than the software, and often the SW development starts before the HW is finalized. In such a case, the SW developer needs to write the software in such way that it is independent on the HW (as is a good programming practice anyway)[1], and define an interface on the so-called *Hardware abstraction layer* (HAL). The bodies of the functions on the HAL can then be implemented later, once HW specification is provided.

This allows for unit testing of SW modules interacting with the HW, but doesn't allow for integration testing, which is often also required. To properly perform integration tests of the ASW (once HW specification is known), the SW developer should create emulators for every piece of HW the ASW interacts with based on provided documentation. The emulator itself has to be tested first to see if it produces output exactly as given in the provided documentation, and then it can be used to integration test the ASW.

2.3.3 External SW libraries

When external SW libraries are to be included into the project, such as an operating system or hardware drivers, it is necessary to verify they have been developed to the same standard as the ASW being developed. This means that extensive documentation including dynamical behavior (timings) and unit test report should be provided along with them. If the libraries contain any tasks (threads, processes), a schedulability analysis has to be provided in addition, so that the ASW developer can then include this information into the ASW schedulability analysis.

This is also a reason why during flight software development it is refrained from the use of C language standard libraries – so that it's not necessary to prove their compliance with the ECSS standard[4], and instead the necessary standard functions are rewritten and tested by the developer.

2.3.4 Software Development Plan

As described in ECSS-E-ST-40C Annex O, the SDP contains description of SW management and development activities and aims to present a work plan in accordance with customer requirements. This document is usually prepared by the SW project manager and contains among

others:

- Master schedule for the SW development
- Assumptions, dependencies and constraints of the master schedule, including foreseen technical issues
- Work breakdown structure and staffing plan
- Monitoring and controlling (QA) mechanisms
- Software development approach

The software development approach is to be planned and presented in the SDP, including software development lifecycle identification and its relationship with the overall system lifecycle (for system lifecycle description see section 2.5.1), SW development standards and practices, used tools (see section 2.4), the development and testing environment, and the documentation plan (for overview of expected documentation output see section 2.6.4).

2.3.5 Coding Standard

A coding standard is a document, usually prepared by the SW developer and often unique to the project, which describes the way the ASW shall be written. The coding standard can cover among others:

- naming conventions
- function header conventions
- rules for writing comments
- rules for encapsulation or modularization of the code
- rules for complexity of the code

There are several reasons to develop and adhere to a coding standard:

1. writing code in a unified way will enable SW developers to easily review and test code of other team members
2. common mistakes can be avoided (such as accidental assignment in if clause)
3. it can force proper documentation of the code
4. it enables QA of the SW developer or ISVV (see section 2.6.3) to easily check SW quality
5. if the standard is developed uniquely for the project, and the code adheres to the coding standard, it can be concluded that the software meets its quality requirements

Due to its importance in SW development, the project coding standard can be a required input for the ISVV and a part of the overall documentation delivery.

2.4 Tools used for SW development

There are many commercial and open-source tools to help SW developers in their work. This section presents a brief overview of the most commonly used ones. While this is a more general topic, the content of this section are still focused mostly on tools useful for flight software development in C language, and tools for other languages may be omitted.

2.4.1 Software Configuration Management

Version control systems, also known as software configuration management (SCM) are systems used to manage changes in files over time. These are essential to SW developments, as they allow to:

- track changes, enabling the developer to revert to an older revision
- merging changes from multiple developers, who can then edit one file simultaneously
- save entire codebase to one central location, which can then be automatically backed up
- branch or merge the codebase, enable multiple developers to work on different functionalities of the same software independently
- track which developer last edited each line of code (the so-called *blame* function)

Due to these benefits, it is recommended to always use a version control system, and include not only the codebase, but all files and documentation into the repository[1]. In flight software development it can even be a requirement on the software, as the usage of SCM is necessary to ensure sufficient software quality and effectiveness of its development. It also enables the developer to uniquely reference a version of software which is used for verification and validation, preventing ambiguity in exactly which version of SW was used for testing and review.

In addition to the benefits described above, having the codebase always up-to-date in one location allows the developer to run automated scripts each day, which can automatically format the code or offer updated insight into the SW quality such as:

- number of compile errors and warnings
- number of lines of code
- number of TODOs in the codebase
- number of coding standard violations (not all can be detected automatically, but e.g. forbidden constructs such as usage of *todo* can be detected by using a script)

Version control systems can be centralized (with one central repository server) or distributed. Commonly used centralized systems are *Concurrent Versions System (CVS)* or *Subversion (SVN)*, an example of a distributed system is the currently trending *Git* or *SVK*. While a

centralized system allows for better overview over versioning (as each commit has a sequentially incremented number), distributed system allow for greater flexibility, offline work, no dependance on a central hub (which may fail) and, in the case of *Git*, offer easy ways to perform common tasks such as branching or merging the repository (which can be more complicated in e.g. SVN system).

2.4.2 Debugging tools

Computer software developed for standard OS environment or off-the-shelf embedded devices can usually be debugged using a debugger integrated into an IDE, such as *Eclipse* or *Microsoft Visual Studio*. This is due to the fact that the target architecture is the same or similar to the development architecture. However since flight software usually runs on a custom embedded device, and running software directly on the target is preferable, a suitable debugger able to communicate remotely with embedded devices has to be selected. The most popular of these is the *GNU Debugger (GDB)*.

For GDB to function properly, it has to be compiled with the same version of compiler as the ASW. Then it needs to be provided with the binary file and map file of ASW (to resolve variable and function names), and a way to remotely communicate with the embedded device by either connecting to a GDB server or utilizing a GDB proxy connection to a proprietary MCU debugger.

A debugging console can also be used to print debug information out from the embedded device. Although useful, this technique is advised against, as running a debugging console presents an unpredictable load for the CPU and the used communication interface, and may interfere with correct function of the ASW. Therefore if the developer wants to use the debugging console, it is necessary to include it in the schedulability analysis and approach it as any other part of the space-qualified SW.

2.4.3 Test frameworks

In comparison to for example Java language, where creating unit tests with *JUnit* test framework is fairly straightforward and easy process, writing unit tests for C language modules is more complicated. The common issue is to break dependencies on other modules, which can be achieved by various means, function stubbing being the most common. Since test of units, most commonly single functions, can be done independently on architecture, these tests are usually run in the development environment as opposed to production environment.

The most common functionalities offered by unit test frameworks are the following:

- automatic function stub generation
- automatic test case generation
- running code in a separate process to allow for catching system signals (e.g. segfault or failed assert)

Some examples of unit test frameworks for C language are *Unity*, *Check* or *Google Test Framework*.

For integration or end-to-end (EtE) testing it also possible to use functions of a test framework; however usually these have to be developed specifically for the needs of the project (e.g. by defining project interfaces and communication protocols) and cannot be easily reused between projects.

2.4.4 Ticketing systems

Even in smaller projects, the need to organize tasks assigned to each team member arises fairly quickly, as using just e-mails to assign work proves to be very ineffective. This gave rise to various ticketing systems, which offer comprehensive overview of tasks (*tickets*) generated during the entire life of the project.

The tickets can be used to either request new features and assign work by the SW project manager, or they can be used to report bugs by any member of the team. The overview of tickets with highest severity to the project may then be included in monthly reports to judge the project progress and in SRelD of each released SW version (see section 2.6.4) to describe known issues with the release.

There is a large number of available ticketing systems, both commercial and free, and the selection of one system is entirely up to the SW project manager; some commonly used open-source examples of ticketing systems are *BugZilla*, *Mantis Bug Tracker*, *OsTicket* or *FlySpray*.

2.4.5 Code editors

Before integrated development environment editors (IDEs) became widespread, all code had to be written manually using a text editor. While this allowed the developer to rapidly code in a straightforward and understandable environment, it didn't offer the same amount of overview and automated tools IDEs offer. Some of the advantages IDEs offer against text editors include:

- automated code formatting
- context highlighting
- header file generation
- documentation generation
- integrated compiler (or interpreter)

While some people still prefer to write code in a text editor such as *vim*, *Emacs*, *Notepad++* or more modern *Sublime text*, many people prefer to use IDEs such as *Eclipse*, *NetBeans*, *Code-Blocks* or *Geany*. No matter what the choice of development environment, for rapid and effective work it is strongly recommended to properly learn one environment along with all of its keyboard shortcuts and then use it without the need to reach for the mouse[1].

2.5 SW project lifecycle

This section describes a project life cycle and the development process from a systematic point of view.

2.5.1 ESA project phases

The lifecycle of ESA space projects is typically divided into 7 distinctive phases. Each of the phases described below is usually ended with a project-wide review, which determines the maturity of the project and its readiness to progress into the next phase[6]. Five of the major reviews, which are relevant to the work of a flight software engineer, are then detailed in the next section.

Phase 0 – Mission analysis: The goal of this phase is to prepare the mission statement by identifying the project goals, needs and expected performance. Possible mission concepts are then developed in accordance with the mission statement, and a preliminary requirement specification is produced. This phase is ended with Mission Definition Review (MDR).

Phase A – Feasibility: During this phase the mission concepts defined in Phase 0 are examined for their feasibility. This is done by elaborating possible system architectures and comparing them to the identified project needs, assessing constraints related to implementation, cost, schedule, etc. and examining readiness of the technology required for the project. At the end of this phase, one or more feasible mission concepts are selected for further examination, and a technical requirement specification is produced. This is then reviewed in Preliminary Requirements Review (PRR).

Phase B – Preliminary definition: At the beginning of this phase, trade-off studies are performed, and the preferred mission concept is selected. A preliminary design definition is then prepared for the selected mission concept, along with interface, schedule and budget plans. During this phase, a complete technical requirement specification is prepared, and then reviewed in System Requirements Review (SRR). At the end of this phase, a Preliminary Design Review (PDR) is held, which judges the project readiness to progress to the next phase.

Phase C – Detailed definition: During this phase, the detailed mission design is finalized and the production, development and testing of critical system parts is performed. At the end of this phase, all critical parts of the project (hardware, software etc.) shall be produced, tested, and ready for qualification. This is verified by the Critical Design Review (CDR).

Phase D – Qualification and Production: The major tasks performed during this phase are complete manufacturing and qualification testing of all project parts, their final assembly, and End-To-End (EtE) testing of the space and ground segment of the mission.

The outcome of the verification, validation and testing performed in this phase is examined in Qualification Review (QR), and at the end of the phase the project is accepted by the customer during Acceptance Review (AR).

Phase E – Utilization: In this phase the mission is launched and operated. All activities at space and ground segment are performed to complete the mission objectives. There are four reviews planned for this phase: Flight Readiness Review, held before the launch; Launch Readiness Review, held immediately before the launch; Commissioning Result Review, held after completion of the on-orbit commissioning activities, and finally End-Of-Life Review, which advances the project into the last phase.

Phase F – Disposal: During this phase the mission disposal plan is executed and Mission Close-Out Review is held.

2.5.2 Software project reviews

There are five main project reviews identified in ECSS-E-ST-40C[4] which shall be included in every project. These reviews are under the responsibility of the customer and aim to evaluate the project progress and status of activities (as described by section above). In this section they are examined from the point of view of the ASW developer.

The mentioned reviews are:

System requirement review (SRR), whose purpose is to consolidate a system requirement baseline and reach its approval by all stakeholders. Expected output of this review is a Software System Specifiaction (SSS). Although this can include a review of software requirement specification (SRS), which is derived from SSS, SRS often undergoes changes and refinement as the software design progresses.

Preliminary design review (PDR), which reviews technical specification (including its compliance with SSS), software architecture and interface design, and verification and validation plans. After this review the overall design of the ASW should remain fixed and software coding can begin.

Critical design review (CDR), which takes place once all planned software activities are finished, i.e. the software is coded and tested on both unit and integration level. Its aim is to review the software design definition file (SDD), software justification file (can be part of SDD) and user manual, and ensure their compliance with SSS and SRS. The code itself and the completeness of its unit and integration testing is also reviewed. For software developer this review usually marks an end to development of the software, although an effort to fix identified problems may continue until the next review.

Qualification review (QR) reviews verification activities of the software and validation of the SW against the requirements baseline. After this review only SW delivery and installation to operational environment ensue.

Acceptance review (AR) marks an official end to the project development. It is a formal review where the customer officially accepts the completed and installed SW. After this review the project moves to support and maintenance phase from the SW developer point of view.

Any discrepancies discovered during any of these reviews are formally recorded as Review Item Discrepancies (RIDs). These can be classified as major (impacting mission objectives) or minor (considered part of normal workflow). All RIDs have to be followed-up on and should be addressed in time before the next review. The customer monitors actions taken on each RID and decides on its closing when its issue is resolved[6].

2.5.3 Software requirements derivation

As described in section 2.3.1, at the beginning of SW development the developer is provided with SSS, from which he needs to derive software requirements specification.

First, the system requirements which directly affect the ASW, and would thus become *parent requirements*, are identified. These are the requirements for the subsystem, which is a parent of ASW subsystem in the product tree (usually OBC or avionics subsystem). Only the requirements directly for the parent subsystem are taken in account; higher requirements in the requirement hierarchy should be already traced down to the identified parent requirement, and thus don't have to be regarded.

Once the set of parent requirements is identified, the requirement derivation process begins. During this process, a set of requirements describing the ASW is created, where each requirement has a parent from the set of parent requirements, and inversely each requirement from the set of parent requirements needs to have at least one child requirement (i.e. all relevant requirements have to be traced down). One parent can have multiple child requirements, however each child requirement should have only one parent, so that a tree structure of the requirement specification is maintained.

The child requirements generally should be more specific and go more into detail than their parent requirements. For example, where the parent requirement states that an OS with thread support shall be used, the child requirement may specify which specific OS shall be used; or where parent requirement states that the ASW shall be state-driven, the child requirements may describe all ASW states and the actions performed in them.

When regarded individually, all requirements have to be written such that it is possible to objectively assess or test their completion. For example, sentences like *"The communication overhead should be as low as possible"* should be avoided, and the requirement should be stated explicitly like *"The communication overhead shall not exceed 5%"*.

When regarded as a whole, the requirement set has to be complete, i.e. sufficiently cover all requirements imposed to the parent subsystem, and detailed enough so that ASW can be designed based on it. All critical decisions, which would significantly change the ASW design should be written down as requirements (so that changing them would be traceable); while less

important design decisions can be left up to the design and development process.

2.5.4 Technical budget and margins

Technical budget of the project is a document describing allocation of physical resources (mass, power consumption, link budget etc.) and the constrains put on them. From ASW developer point of view it servers only as an informative resource, although it can become necessary when the developer needs for example to design a communication link protocol.

Preparation of the technical budget is a responsibility of a systems engineer. Systems engineering is an inter-disciplinary field, which focuses on project management from engineering point of view, e.g. allocation of physical resources to each company, department or team working on the project. In project definition phase, a systems engineer must cooperate with each team (e.g. structure, avionics, software etc.) and prepare a realistic technical budget satisfying all technical requirements and the limits of available technology. The technical budget is not however fixed, as during the project development some of the teams may encounter unexpected constraints and may require change in the technical budget. The systems engineer must then update the budget and at the same time make sure not to disrupt work of other teams in the project.

Relevant to this topic is the idea of technical ("safety") margins. It states that at various phases (or more accurately at time of major reviews) the system under development shall use (or plan to use) only a certain percentage of the resources allowed by the technical budget. As time progresses and the project comes closer to completion, the margins decrease, i.e. the allowed percentage of resource usage increases. This approach ensures that constraints imposed by the technical budget are held, but also allows some room for errors and unexpected circumstances, which may increase the resource usage.

To illustrate with an example, the typical memory budget margin as given by the ECSS Handbook[7] is 50% at PDR, 35% at CDR and 25% at QR. If the margins are not met at any of the reviews, the developer must provide an explanation and a corrective action must be proposed.

2.6 ASW design and development process

2.6.1 Software static architecture design

The SW static architecture design goes hand-in-hand with the SW requirements baseline design, or more precisely the SRS should be a description of the overall SW architecture (as described in section 2.3.1). Therefore the SRS can be finalized only when the overall SW structure, operation and mode/state sequence is established. After this, the detailed software design process can begin.

During the static architecture design it is necessary to keep in mind the available sensors, actuators, HW peripheries and mission objectives; therefore, the SW design will be different for

each mission. Most SW architectures for use in space segment however are on a similar basis as the tasks needed to be performed can be generalized into four areas:

- Reading data from sensors
- Performing calculations or decisions
- Operating actuators
- Communicating (sending telemetry and accepting telecommands)

For sensor readout and actuator operation, an interface needs to be developed; either to provided SW libraries or directly to hardware items. For communication with the ground segment, a communication protocol has to be developed, along with means to process and parse incoming messages and send outgoing messages. Finally, there needs to be a system connecting these three areas, which performs the necessary calculations, takes decisions and performs other operations.

One of the important basics of good SW architecture is SW modularity. Therefore one of the first steps of SW architecture design is preparation of division of the necessary tasks to be performed into SW modules or packages (comprising of several modules). The modules or packages defined should have a clear (ideally single) purpose, defined and well-documented interface to other modules and should hide as much as possible of their inner functionality from the rest of the SW (this is known as *encapsulation*). An oriented graph describing the interaction of modules should also be prepared; ideally, the graph should not contain any cycles and be structured in a layered manner, with the bottom layer being the interface to SW libraries or HW (the so-called Hardware Abstraction Layer (HAL)), and each layer above only using functions of itself or the layer directly below[1].

When designing flight software, it is important to avoid unnecessary complexity, even more than with design of other SW. Simple and understandable solutions are heavily encouraged, because simple SW can be easily reviewed, tested, simulated and operated. Some complexity is of course unavoidable (arising from the project needs), but it is necessary to avoid adding more complexity, not only at the time of SW design (using SW modularity, known design patterns, and designing fault protection early) but also at the future (making the SW design robust against changes). ECSS recommends to reduce code complexity even at the cost of code performance[7].

Another necessary part of static SW design is the design of external interfaces. Although many of these will be given an unchangeable (when off-the-shelf or third-party HW items and SW libraries are used), usually some interfaces have to be designed (e.g. if the SW libraries are a subject of parallel development to the ASW). In this case it is necessary to design the interface to be robust to future changes. If it is also necessary to develop communication protocols for configuration or in-flight communication, the ASW developer needs to cooperate with the ground segment team and adhere to the technical budget.

Since any dynamic memory allocation is forbidden (that is the state-of-the-art in flight software development), memory budget is also part of the static SW design. The memory budget

should be prepared with regards to given technical budget (how much data shall be acquired or recorded, size of configuration data or pre-calculated look-up tables, etc.) and technical margins for the current project stage also have to be taken into consideration (see section 2.5.4).

Since most of the space segment software has similar needs, there has been a trend to develop *reference architectures*, which can be reused between projects. These architectures can contain for example reusable software for communication with the avionics, real-time scheduling or memory management, which results in reduced development time and cost, due to "not having to reinvent the wheel". With usage of such reference architecture, the ASW developer chooses its parts applicable to the project, and than has to only implement application specific and hardware specific code (see figure 2.2). Examples of such architectures are On-board Software Reference Architecture for Payloads (OSRAP)[8] or Space Avionics Open Interface Architecture (SAVOIR)[7].

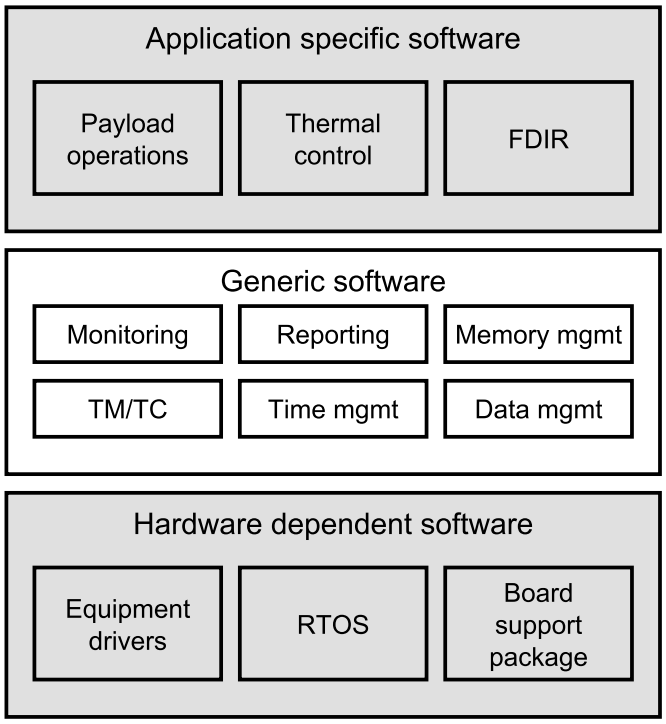

Figure 2.2: SW layer overview with usage of a reference architecture[8]

2.6.2 Software dynamic architecture design

The SW dynamic architecture describes the time-dependent behavior of the SW and dynamic aspects of the design, such as:

- Threads or tasks and their synchronization
- Shared resources and mechanisms of their protection

- SW behavior, state transitions, etc.

As with the static architecture design, the dynamic design is specific to the needs of each mission (although there are again similarities between various flight software projects). The aim of dynamic design should be to minimize the total number of tasks, while at the same time making sure each task performs a clearly defined operation and ends as soon as possible (doesn't run unnecessarily long).

The dynamic design also includes the actual scheduling of the tasks. The tasks can be woken either by an interrupt or periodically by a scheduler. The periodically woken tasks should then be scheduled in a deterministic way, enabling complete deterministic analysis of the task timing.

The most important part of dynamic architecture design is the schedulability analysis. During this analysis all tasks are examined for their priority, periodicity/period and worst case execution time (WCET). During design phase, the WCET is estimated based on the currently designed load. During development and later, the WCET is measured on the target architecture, with maximum possible load and worst-case execution path. The output of the schedulability analysis is then the average CPU load (which should observe technical margins for current project phase) and observation if all deadlines (hard real-time or soft, set by SRS) in the SW are observed.

In case multiple tasks are used, the dynamic design should also contain prevention (or contingency) of common problems associated with task switching, such as deadlocks or priority inversion problem. Most of these problems are however solved or their impact is reduced with usage of a RTOS such as FreeRTOS[9], which implements its own scheduler and protection mechanisms.

The SW dynamic design also includes a definition of SW behavior. If the SW behaves as a state machine, the states and the transitions between them should be clearly defined. A graph description of the state sequence is usually provided for clarity.

2.6.3 QA and ISVV

Based on the criticality of the SW, the quality control of the SW can be as important as development itself. Usually there are requirements on SW quality specified in the SRS, and the task of Quality Assurance (QA) personnel is (among others) to make sure these requirements are observed. While QA is usually part of the ASW development team, the Independent Software Verification and Validation (ISVV) personnel are external to the ASW developer and serve as an independent entity.

The QA personnel and subsequently ISVV auditors perform:

- Code audits, which ensure that the code is well written and documented, that the SW quality requirements are being observed and that the design documentation accurately describes the actual code
- Design documentation review, which ensures the design is complete, well structured and justified, and contains no errors or inconsistencies

- SW verification, which is a broad term encompassing the review of correct functionality of the SW; this may include review of unit/integration test plans and reports

- SW validation, which ensures that the produced SW fulfills its requirement specification.

A correct version control of ASW and all of its documentation is required, so that in each review it is clear which version of the SW or document is being reviewed, and in turn additional work in reviewing of issues which have been already resolved is minimized.

2.6.4 Expected output

Apart from the ASW itself, the SW developer is expected to deliver a number of documents to the customer. The expected documentation and its contents are prescribed by ECSS-E-ST-40C (in its Annexes), although this can be adjusted by the customer to the specific needs of the project (see section 2.2.2). The most important and usually required documents are:

Software Requirements Specification (SRS): This document was described in detail in sections 2.3.1 and 2.5.3. Requirements on this document are given in ECSS-E-ST-40C Annex D.

Interface Control Document (ICD): This document lists and describes all external ASW interfaces, including interfaces to SW libraries, HW items and communication buses and human-machine interfaces. It shall also have a complete overview of all communication protocols used. The SW ICD is used by other teams cooperation on the project (e.g. developing ground segment software) to develop modules for communication with the ASW or to verify the ASW uses the given interface correctly (in case of HW interfaces). Requirements on this document are given in ECSS-E-ST-40C Annex E.

Software Design Document (SDD): This document describes the ASW static and dynamic architecture. It shall contain, among others, a detailed description of the software, its internal interfaces, modes or states in which ASW operates, mission and configuration data, ASW behavior, and memory/CPU budgets. It should also contain a reference manual with description of each ASW function, its parameters and their validity ranges; this part of the SDD can be automatically generated from the commented code using dedicated tools (e.g. Doxygen). Requirements on this document are given in ECSS-E-ST-40C Annex F.

Software Release Document (SRelD): The purpose of this document is to describe each software release version in terms of problems or limitations with respect to the approved technical baseline (described in SRS and ICD documents). It also contains a changelog from previous released versions, a description of ongoing changes in the ASW, and advice for usage of the released ASW version. Requirements on this document are given in ECSS-E-ST-40C Annex G.

Software Configuration File (SCF): This document is released along with the SRelD during each software version release. It lists the contents of the delivery (code, documentation, other files), provides a checksum for each file (or just the entire delivery) and describes steps necessary to install, modify or run the software. It can also list the necessary tools which were not part of the delivery, such as required compiler or SW libraries. Requirements on this document are given in ECSS-M-ST-40C Annex E.

Software User Manual (SUM): This document provides instructions for users of the ASW. It shall summarize the SDD and ICD in a concise way (so that the user doesn't have to read through all of the technical documentation), i.e. describe the software purpose, configuration and nominal operation. It should also guide the user through set-up and initialization of the ASW, describe useful commands and operations and provide a tutorial, including several use cases. Requirements on this document are given in ECSS-E-ST-40C Annex H.

Software Verification Plan (SverP): The purpose of this document is to describe planned software verification activities. This includes the verification methodology for each requirement and software item, along with overall organizational aspects such as schedule, resource allocation, responsibilities, control procedures, etc. Requirements on this document are given in ECSS-E-ST-40C Annex I.

Software Unit/Integration Test Plan (SUITP): This document provides detailed description of planned unit and integration tests. It provides the setting, inputs, procedure and expected output for each planned test. The results of testing are then usually delivered separately, in Software Unit (Integration) Test Report document. Requirements on the SUITP are given in ECSS-E-ST-40C Annex K.

The list above is only a selection of deliverable documents. A usual ASW release will contain a number of additional documents, regarding for example software project management or software verification and validation processes and their results. For a complete overview of project documentation see the referenced ECSS standards[4][6].

Chapter 3

Demise Observation Capsule

The Demise Observation Capsule project, which is a part of ESA's Future Launchers Preparatory Programme (FLPP), aims to improve the understanding of the disintegration process of launch vehicle upper stages during atmospheric re-entry, and by doing so helps to develop stages for safer deorbitation maneuvers[10][11].

DOC's function can be likened to the function of an airplane's black box. It is a small device which will be mounted to the host vehicle (HV; a launch vehicle upper stage), collect data and observe the demise of the HV while surviving the atmospheric re-entry, and transmit the collected data to ground segment.

3.1 Mission objectives

The mission objectives of the DOC from systematic point of view are:

- Activate itself once the deorbitation maneuver of the host vehicle (HV) begins
- Observe the atmospheric re-entry by taking temperature, pressure, and IMU measurements
- At the right time release itself from the host vehicle
- Observe the disintegration of the host vehicle using two on-board cameras
- Survive the atmospheric re-entry
- Follow a ballistic trajectory with host vehicle debris, thus measuring their trajectory
- Transmit measured data and photos to the DRS.

As the DOC is expected to fall into the ocean and survival of the landing is not foreseen, it is necessary to transmit all essential data before the surface impact.

3.2 Project overview

The project consists of a space (flight) segment, which is the Demise Observation Capsule itself, and a ground segment, which contains an Electrical Ground Support Equipment (EGSE) for pre-flight configuration and checkout, and Data Reception System (DRS) for in-flight data download.

The flight segment is then further divided into subsystems, such as the power subsystem, structure subsystem, avionics subsystem etc. The avionics are then composed of sensors, actuators, communication buses and the on-board computer, on which the ASW runs.

The simplified overview of the project hierarchy is presented on figure 3.1.

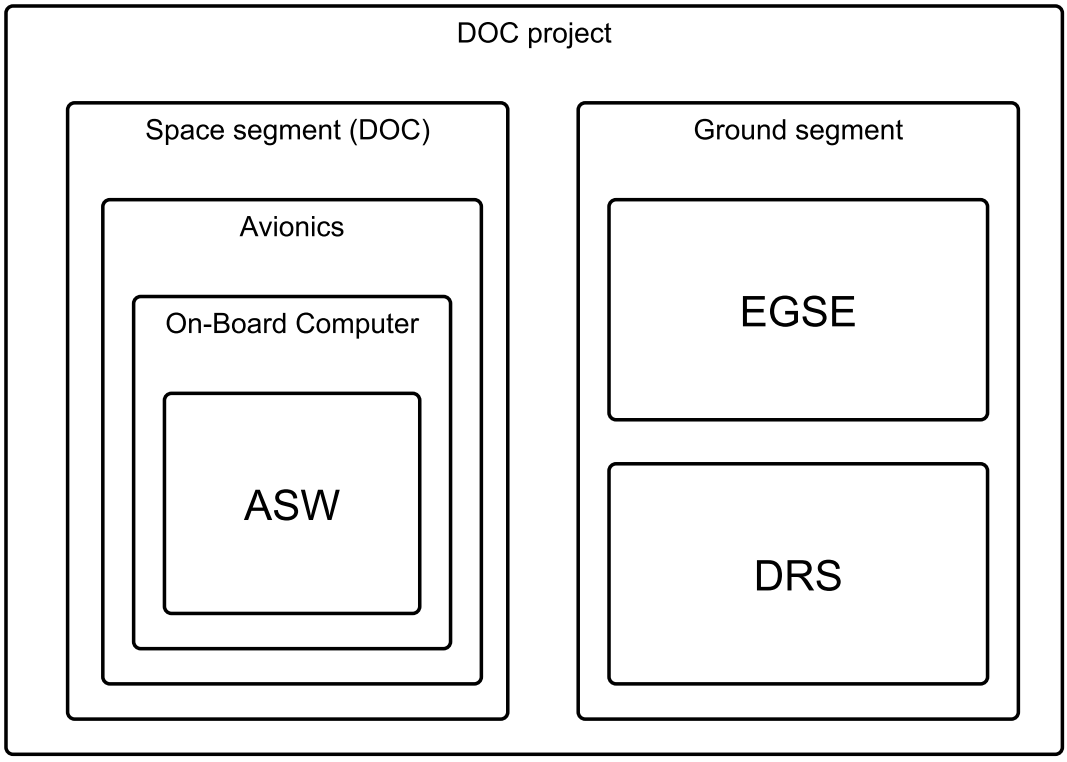

Figure 3.1: Overview of the DOC project (simplified, only relevant subsystems shown)

3.2.1 DOC HW overview

The main part of the DOC avionics is the On-Board Computer, on which the ASW is loaded. The OBC contains a processor, a real-time clock, SDRAM, FRAM and Flash memories and UART, CAN and I^2C buses for communication with peripheries. The OBC is powered off using Activation Inhibit (AI) circuit, which is an active-high circuit controlled by the HV. By turning off the AI, the HV can activate (turn on) the DOC.

To facilitate the mission, the DOC contains a number of sensors measuring various physical quantities:

- Temperature sensors (external and internal)
- Pressure sensors (external and internal)
- IMU for measurement of acceleration, angular rate and magnetic field strength
- High-acceleration accelerometer, which provides a 3-axis acceleration measurements in times when IMU accelerometer gets saturated
- GNSS device to provide GPS location and accurate time.

To reduce HW complexity, the majority of the sensors have been placed on a dedicated sensor board, which has its own micro-controller and is connected to the OBC via CAN bus. The sensor board works independently to the OBC, and is programmed to sample the sensors every 100ms; in case the sensor measurements cannot be read out immediately, it contains a ring buffer which can contain up to 10 measurement sets (i.e. 1 second of measurement data). The measurements then need to be downloaded to OBC using provided SW library including a CAN driver.

The only sensor/device which provides payload data and is not integrated in the sensor board is the GNSS device, which is powered and operated independently. The communication with the device is conducted over an UART interface. For this device no SW libraries are provided, and therefore the communication with it needs to be implemented in ASW from scratch.

In addition to the sensors listed above, the DOC contains two cameras; one is placed on the back shield of the capsule (in this work will be referred as *internal* camera) and the second one is placed on the HV and connected to DOC using a communication harness (in this work will be referred as *external* camera. Both cameras are identical as to manufacturer/make. Each camera has its own micro-controller, which is able to automatically take, process and save photos to the camera internal memory. Therefore only needed action from the ASW is to send a set of configuration to the camera, at the right time order the camera to take pictures, and then download the pictures to OBC memory. The communication with the cameras is over a CAN interface, with the CAN driver being included in a provided SW library.

The only actuator the ASW can operate is the HV release mechanism. The mechanism consists of two redundant release switches (the release mechanism will fire when both are activated at the same time), two redundant release sensors (to detect if the DOC is still attached to HV) and two redundant sensing straps (to detect if the DOC received permission to release from the HV). The release mechanism is disarmed by a Release Inhibit circuit, which is analogous to the AI circuit and is also controlled by the HV (to prevent premature release). The interface to the release mechanism is integrated into the sensor board and is operating using the same interface as the SB; that is CAN bus using SW library wrapper.

For communication with ground segment during flight, the DOC contains an Iridium modem, which is connected to the OBC using an UART interface. For configuration and operation of the modem, standard AT commands are used (no SW library provided). On ground, a Data Reception System (DRS) is used as the other side of the data link (see section 3.2.2).

The ASW can turn on/off various DOC HW items using the Electric Power Supply (EPS) device. The EPS has 6 power channels, and thus can independently toggle power to the OBC (OBC is running the ASW, and thus this channel shall stay powered on during the complete course of the mission), the GNSS device, internal camera, external camera, and the sensor board including the modem (together they use two power channels).

Finally, for configuration of the DOC before mission, an EGSE is being developed (see section 3.2.3. The EGSE can be connected to the DOC using an UART interface.

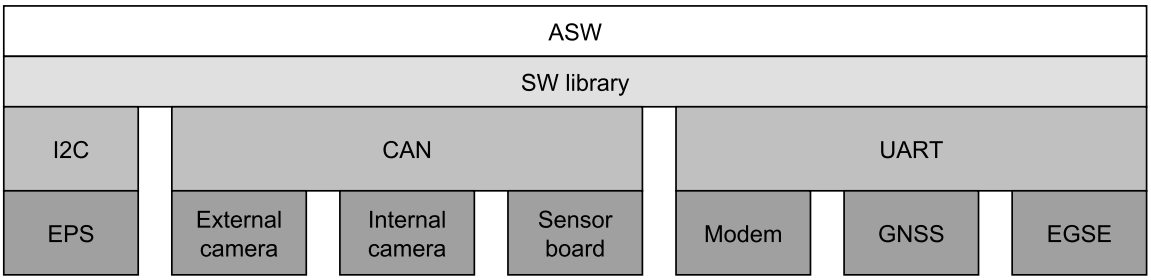

Figure 3.2: Overview of the DOC HW peripheries

3.2.2 DRS

The Data Reception System is a ground segment application used to communicate with the DOC, receive packages containing payload data, and generate acknowledges on them (which are sent back to the DOC). The Iridium network is used as the communication provider, with DOC using a wireless satellite connection and DRS using a TCP connection integrated into the Iridium network (for a diagram see figure 3.3).

Since during the atmospheric re-entry an ionization barrier forms around the HV and DOC preventing any kind of communication, it will be possible to obtain the Iridium data link only in the lower part of the atmosphere, when the free-fall speed is sufficiently decreased. Therefore the expected communication time is only about 250 seconds. Since the expected communication rate is 2400 bit/s, the expected total amount of data, which can be transferred to the DRS, is about 75 kBytes.

3.2.3 EGSE

An Electrical Ground Support Equipment (EGSE), which consists of EGSE HW and EGSE SW, is developed in parallel to the ASW, and is planned to be used for pre-flight check-out and configuration. The EGSE HW constists of:

- EGSE computer, which runs the EGSE SW
- Ethernet connection between EGSE computer and EGSE box

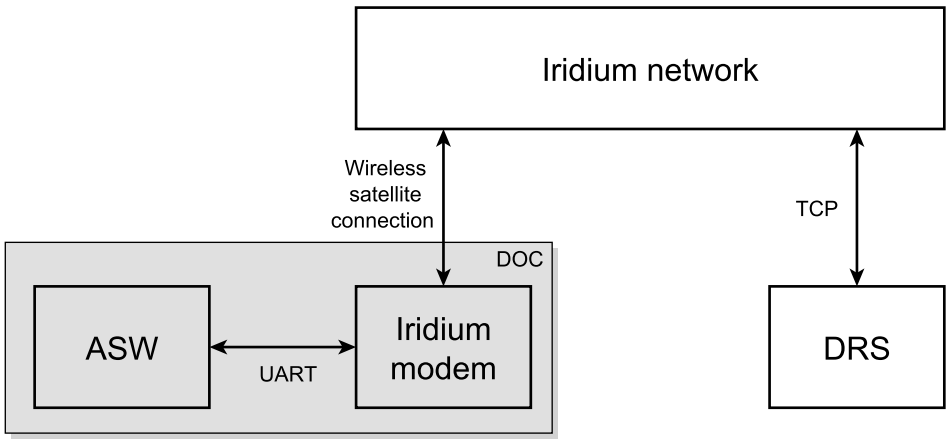

Figure 3.3: Diagram of ASW/DRS communication

- EGSE box, which serves as a TCP/UART adapter and contains means to power on/off the DOC (the activation inhibit (AI))
- Serial link between EGSE box and DOC.

For a visualization of this communication link, see figure 3.4.

During mission simulation and testing, the EGSE box can simulate the signals of the HV, i.e. turn on/off the DOC by disabling/enabling AI circuit and arm/disarm the release mechanism by disabling/enabling RI circuit. Regarding communication, the EGSE box is transparent both from EGSE computer and DOC side.

The EGSE SW contains a GUI which enables the user to perform a pre-flight checkout and health check of the DOC. The user can also upload various mission-dependent setting and configuration to DOC. Secondary purpose of the EGSE SW is to support the testing campaign, e.g. by being able to upload simulation data to DOC or perform memory dump/load. The design of ASW/EGSE interface is further discussed in section 4.1.2.

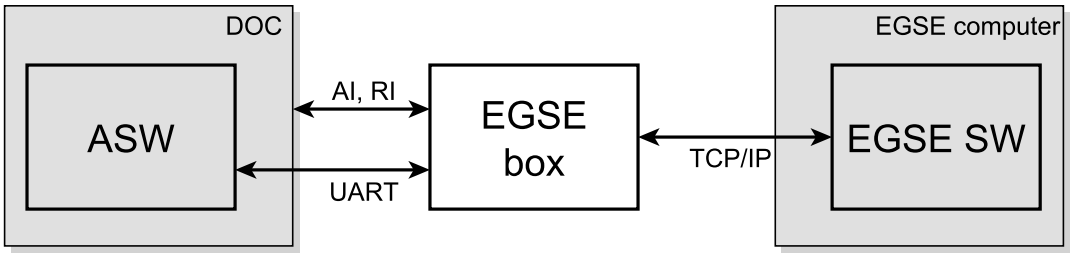

Figure 3.4: Diagram of ASW/EGSE communication

3.3 Required ASW operation

From the systematic point of view, four distinct phases of the mission have been identified:

1. **Pre-launch phase:** Entails DOC health checkout, configuration for mission and installation into HV
2. **Launch and Ascent phase:** Starts at HV launch. DOC is turned off or in stand-by. When orbit is reached, DOC activates and prepares for re-entry.
3. **Re-entry phase:** In this phase the most important measurements and pictures are taken. The DOC separates from HV during atmospheric re-entry and observes its demise.
4. **Descent phase:** The main goal of this phase is to acquire data link with the DRS and transmit as much data as possible before surface impact.

This translates to the following required sequence of operations for the ASW:

1. At startup perform self-checks and go into stand-by, conserving battery power
2. Allow EGSE connection during the stand-by. If the EGSE connects, switch into a dedicated maintenance mode. Once the EGSE disconnects, go back to stand-by.
3. By measuring acceleration detect the launch, or by measuring pressure detect in-orbit environment
4. After configurable amount of time since launch or since orbit was reached (depending on mission configuration), wake up from stand-by mode, initialize all HW peripheries, and prepare for measurement of data and picture acquiring.
5. When initialization of all peripheries is complete, start measuring data. After configurable amount of time since the wake-up, take pictures using the external camera and download them to OBC; then release from HV and take pictures using the internal camera.
6. While continuing to measure data, start trying to acquire the Iridium data link
7. Once the data link is open, transmit as much data as possible in configurable order (different data will have different priority assigned during mission configuration)

Because the DOC doesn't have any actuators or active surfaces except for the release mechanism, and the release mechanism is protected from accidental or premature firing by the release inhibit operated by the HV, the ASW is of criticality category C (see section 2.2.2).

3.4 Requirements on payload data flow

It is planned (and required) that all sensor data will be acquired with a 10Hz frequency. This also includes the measurements from the GNSS device.

A definition of the range and resolution of each measurement is part of the SSS; in total, measurements from all sensors occupy 66B of data. The acquired raw measurements are converted to the required range and resolution by the provided SW library, and the ASW doesn't do any post-processing of the data.

Since each measurement set (measurements from all sensors acquired at one point of time) is 66 bytes large, and the total amount of data which can be sent is only 75 kBytes, the size of the sent measurements needs to be reduced (otherwise only 113 seconds of data could be transmitted, with no space for camera pictures). Therefore it has been decided to divide the acquired data into two categories: essential and non-essential. The essential data would be a result of decimation and averaging of data points, while the non-essential data would contain a more complete data set. After the data points are decimated and averaged, they are further required to be compressed using a compression algorithm. This approach allows to transmit all essential data to ground and still have space left in the communication budget for camera pictures.

The data acquisition settings (decimation rate, averaging and transmission priority) of each measurement are part of the mission configuration and can be configured before each flight using the EGSE.

For testing purposes, it is also required that all measurements acquired shall be saved into the on-board flash memory. Due to this, the complete set of measurements can be obtained after simulated mission run using memory readout.

To enable a health check of the DOC pre-flight and during the mission, housekeeping data shall also be acquired independently to the payload measurements. The housekeeping data are only infomative and can be acquired with low frequency (configurable by user).

Chapter 4

DOC Software Design

This chapter describes the proposed ASW static and dynamic architecture, as designed before the start of SW development. A special focus is given to the design of the ASW/DRS communication protocol in section 4.3.

As mentioned in chapter 1, the author of this work joined the project when it was past PDR and had a requirement baseline and preliminary design approved. As the requirement baseline (see section 2.3.1) was not thorough or mature enough at that time, it was reworked and extended by the author. The SW design was then completed and brought to maturity under the lead of the author, with supervision and quality control done by author's supervisor and the customer. The comparison of the finalized design to the SW design approved in PDR is given at the end of each section 4.1, 4.2, 4.3.

The design presented in this chapter went through slight changes and improvements during the SW development. These changes are described in section 5.

To limit the extent of this thesis, this chapter explains only the most essential parts of the design. The complete description of the design is in the project SW design documentation (described in section 5.4) also mostly written by the author; however it cannot be included in this thesis due to its size (about 200 pages total), even as an annex.

4.1 Initial static architecture design

4.1.1 ASW packages identification

Based on the analysis described in chapter 3, five overall SW packages were identified. Each package is responsible for one distinct ASW functionality and consists of multiple SW modules, which are described in section 4.1.2. The identifies overall SW packages are:

Operational Manager, which is responsible for the overall ASW behavior and contains the mission mode sequence and decision logic

Data Handling, which contains modules responsible for readout of sensors and processing of the acquired data

Modem Communication, which is responsible for opening the Iridium data link and sending the measured data to DRS

EGSE Communication, which enables the EGSE to perform pre-flight health check and configuration of ASW

Device Drivers, which facilitate communication between ASW and all HW peripheries (containing both device operation logic and lower-level communication drivers).

The diagram of the described SW packages is depicted in figure 4.1. In addition to above, a utilities SW package is identified, which contains functions commonly used throughout the ASW. To preserve simplicity, this package and its modules are not depicted in architecture diagrams in this work, as they interact with most of the remaining ASW modules.

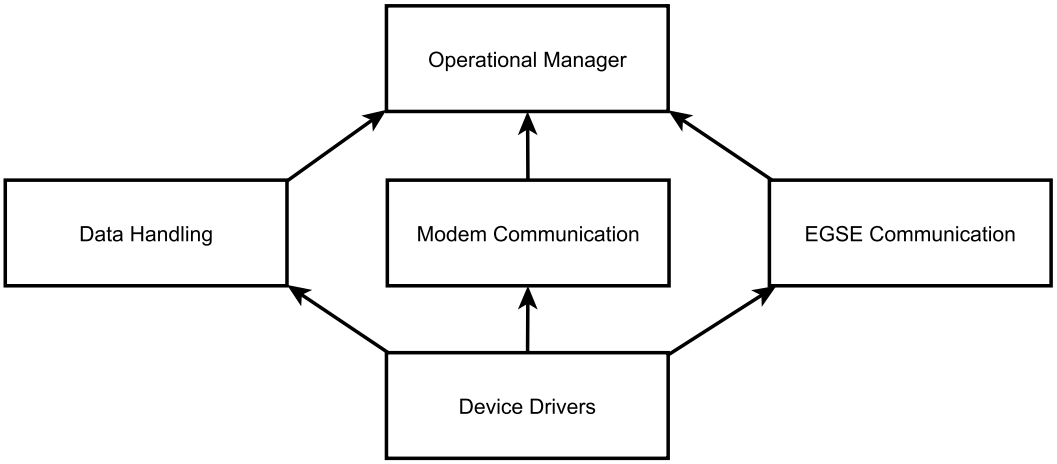

Figure 4.1: Overview of ASW overall SW packages

4.1.2 ASW modules identification

In the next step, the SW packages were divided into SW modules with defined (mostly singular) purpose.

Operational Manager modules

Operational manager package was divided into 6 cooperating SW modules, which controlled the overall ASW functionality. Since the ASW was developed as a state machine (see section 4.2.2), each mode behavior and transition would be operated by this SW package:

Mission Management: The main module of ASW. Contains the control task, which operates the rest of ASW based on the current ASW mode and mission configuration. This module would decide when to turn off/on HW peripheries, when to take photos with cameras, when to release from HV etc.

Mode Management: This module would closely cooperate with Mission Management module. While the responsibility of Mission Management is to control the rest of ASW based on the current mode, the responsibility of this module would be to decide what mode the ASW is currently in, and decide on mode transitions.

Release Management: As this is a critical functionality, it was decided to extract it into a dedicated module. While the Mission Management module decides on the timing of the release, this module would perform checks if all necessary conditions for release have been met (to prevent premature release) and then arm and fire the release mechanism.

Fault Detection, Isolation, and Recovery (FDIR): As is usual in most space projects, the ASW would contain a dedicated FDIR task, which communicates with all ASW modules, detects any faults or errors, and works to isolate them in the source module or recover them entirely. This is usually a complex functionality, and it is recommended to plan with an FDIR module since the beginning.[7]

Time: Since the DOC will not have a measure of real time during the mission (although it has a RTC on board, their time does not persist when the DOC is turned off), the measurements will be timestamped with a mission time, which is defined as zero on wake-up and increments every 100ms. This module would then be used for time-keeping of the mission time and associated functionality.

Scheduler: All ASW tasks are handled (initialized and started) by this module. The actual scheduling is done by the built-in task scheduler in FreeRTOS (which the ASW uses).

Data handling modules

Next, the data handling SW package was divided into SW modules based on the planned data flow. The requirements on data flow were described in section 3.4.

As the first step in the data flow, the measurements would be collected from all sources and processed into essential and non-essential data by **sensor readout** module. The raw (unprocessed) data would be sent to **data storage** module to be saved into the flash memory (this is useful for integration testing, as they can be read out after a mission simulation using the EGSE). The processed essential and non-essential data would on the other hand be sent to **data prioritizer** module which would assign priorities to them based on the mission configuration, and then to **data compression** module which would perform data compression. Finally, the data would be prepared into packets by a **data packetizer** module, and the resulting packets would be saved into memory by data storage. This data flow is also visualized in figure 4.2.

Modem communication modules

The task of *modem communication* SW package would then be to load the data packets (prepared by *data handling* package) from memory, at the right time open the Iridium data link, and send the packets one-by-one, always selecting the packet with the most priority.

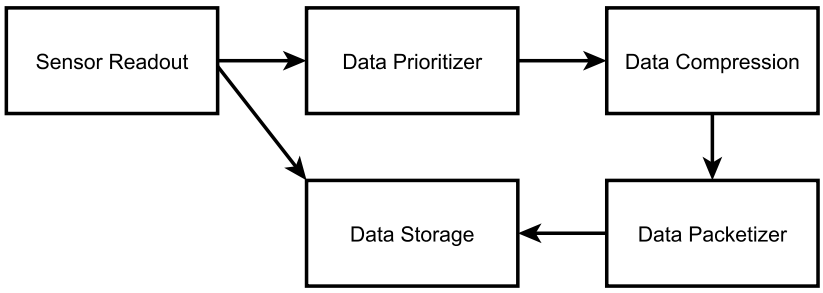

Figure 4.2: Overview of ASW payload data flow through data handling modules

The following modules were identified in this SW package:

Modem Control, which would load the packets and decide which packet to send next. It would also process acknowledges received from the DRS

Modem Connection, which would handle initialization of the modem and opening of the data link

Modem Protocol, which would encode the packets being sent into the ASW/DRS protocol form, and also decode the received acknowledges.

The data flow inside this SW package is pictured in figure 4.3. Description of the design of the ASW/DRS interface is given in section 4.3.

EGSE communication modules

The modules for communication with EGSE were designed in the same way as the modem communication modules, i.e. the three modules identified are:

EGSE control, which performs actions requested by incoming telecommands and generates telemetry packets

EGSE connection, which handles sending and receiving of packets

EGSE protocol, which encodes the telemetry packets into format used by the ASW/EGSE communication and decodes the telecommand packets.

The protocol chosen for this interface is the *CCSDS protocol*, which is a standardized protocol for communication between space segment and ground segment devices. This protocol is defined in ECSS-E-70-41A standard (Space Engineering – Ground systems and operations – Telemetry and telecommand packet utilization)[12] and has been chosen for use on this interface due to its widespread use and familiarity; as it is commonly used in space projects, some ASW functions and ASW/EGSE integration testing platform could be reused from other projects.

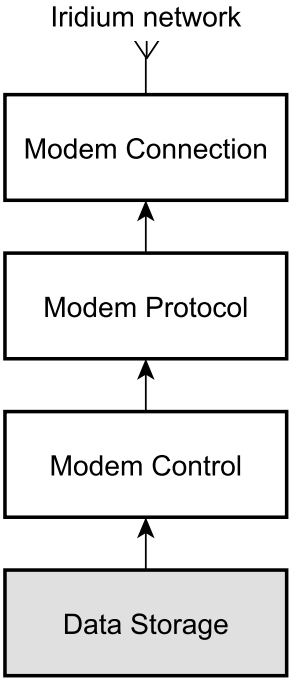

Figure 4.3: ASW payload data (in form of packets) flow through modem communication modules

Device drivers

This SW package would consist of two types of modules: device managers and Hardware Abstraction Layer (HAL) modules.

There would be a device management module for each HW periphery, i.e. for the sensor board, the GNSS device, the EPS and cameras. These modules would be responsible for handling the device initialization and operation (much like the *modem connection* module for modem), and for download of data from them. The data from each device would then get passed on to the *sensor readout* module (see figure 4.4).

The HAL modules would be created for each ASW interface, i.e. for the UART (the only communication bus the ASW interfaces on low level; other buses are accessed through protocols in the SW library) and for different parts of the SW library dedicated to communication with HW peripheries. Using this approach would effectively shield the ASW from changes in the SW library or in HW composition of the DOC, and thus any changes on the external interfaces would require changing only modules in this SW package while the rest of ASW could remain unchanged.

4.1.3 Comparison to PDR design

The preliminary static architecture design was composed only of three SW packages: *Operational manager*, *Data handling* and *Device drivers*. It defined 18 SW modules in total, with each SW package containing 6 modules. Furthermore each module was described only in terms of what

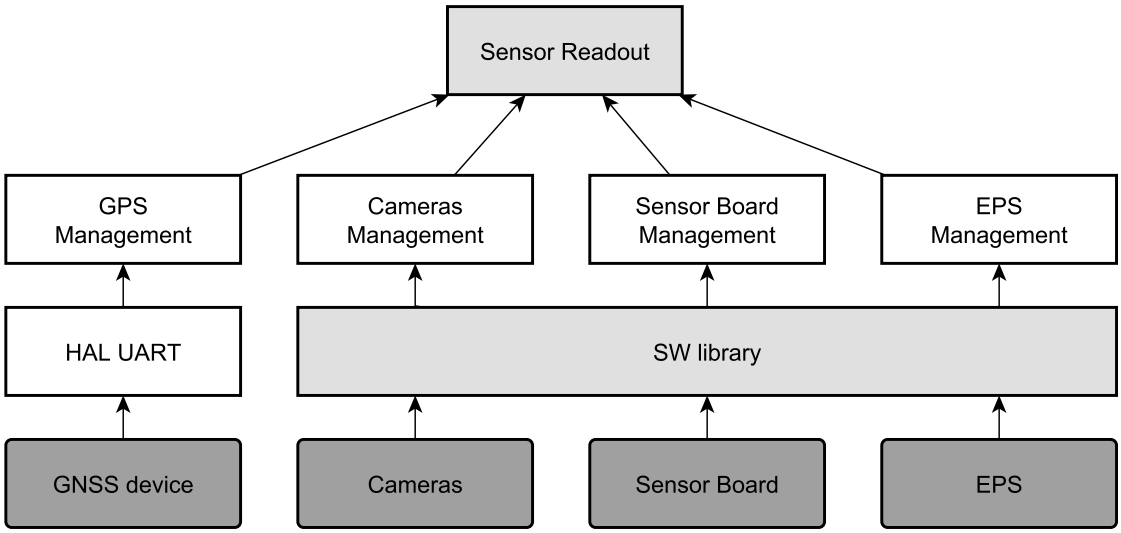

Figure 4.4: ASW payload data flow through device driver modules (HAL modules on the interfaces between device management modules and SW library not pictured)

it shall accomplish; the means of implementation were missing.

Under the responsibility of the author of this thesis, the SW static design was extended to cover 22 modules grouped into 5 SW packages, with implementation method and behavior of each module described.

4.2 Initial dynamic architecture design

4.2.1 ASW tasks

In total, 8 tasks were proposed to be implemented in the ASW. The task cycle repeat period was chosen to be 100 ms to match the requirement on measurement readout frequency; therefore the real-time deadline for all tasks to be finished is 100 ms. After this, new task cycle begins.

The proposed tasks were:

Mission management task, which would reside in the *mission management* module and would control the mode sequence and overall operation of the ASW. It would run at the beginning of the 100 ms cycles and decide on mode transitions and associated actions of the ASW

Sensor readout task, which would run after the mission management task, read measurements from all sensors and save them to internal buffer

Data processing task, which would run after the sensor readout task is finished, and would process the measured data (extract essential/non-essential data, packetize them, and pass them to data storage task). The mechanism chosen to pass measurements between tasks is queues.

Data storage task, which would be used for access to slower memories (especially the flash memory). It was foreseen that data to be saved into a memory would be passed to this task via a queue, which would be read out every 100 ms and the memory write process would begin. The memory read operation would then be asynchronous (when requested by any task) and blocking (as reading is much faster than writing)

Modem control task, which would perform the selection of packets to be sent and prepare them for sending

Modem connection task, which would be used for communication with the modem, its initialization, and subsequently readout and writing to the modem UART

EGSE control task, which would perform actions required by the incoming telecommands and prepare responses. During flight this task would still be scheduled (to avoid changing number of running tasks and thus the ASW schedulability during a run), however there would be nothing to do for the task and thus it would go to sleep immediately

EGSE connection task, which would be used for sending and receiving data on the EGSE communication interface. As with the EGSE control task, there would be nothing to do for this task during the mission.

Having separate modem and EGSE communication tasks was necessary, as the SW library offered only a blocking function for sending messages into each UART, i.e. the function would return only after the whole message was sent. Combined with the fact that the UARTs would be set to low baudrate to reduce the number of errors during the communication, this would slow the calling task considerably, and was therefore necessary to be moved to a dedicated sending task.

To prevent a deadlock or priority inversion problems, the tasks were analyzed and a priority was assigned to them in such a way that minimizes these risks, e.g. so that a higher priority task doesn't need a lower priority task to finish before it can function; also by increasing priority of tasks which take only a small amount of CPU time. Furthermore, all waits for shared resources have a (rather short) timeout, causing the task to skip its cycle and go to sleep if it fails to acquire the shared resource before timeout. Although this should never happen if the schedulability analysis is performed correctly, it adds additional confidence in the ASW robustness.

For real-time scheduling of the tasks and for means of their synchronization and communication, the FreeRTOS operating system is used. FreeRTOS is a flexible OS working on many embedded real-time architectures. It offers multiple means of task synchronization (such as semaphores and mutexes) and task communication (e.g. queues, task messages). For time-keeping the FreeRTOS uses ticks, which start at zero on system start-up and increment every millisecond. On each tick, the OS scheduler runs and reschedules tasks. Therefore the minimum granularity with which the schedulability can be planned is 1 millisecond.[9]

During design phase, the schedulability and WCET of tasks could only be estimated. Based on the available information it was estimated that the CPU load would not exceed 50% in any

mode, satisfying the requirements on CPU budget margins.

4.2.2 Mode sequence

Based on the mission profile described in section 3.3, it was decided to develop the ASW as a state machine progressing through a linear sequence of distinct operational modes. During design, there were 5 operational modes identified:

Stand-by mode, which would be the default mode activated after ASW start-up. Since it is possible the DOC would be in this mode for a long time waiting for the launch and would need to sustain on battery power, most HW peripheries (except the ones needed to detect the launch, i.e. the accelerometer and pressure sensors) would be powered off. There are two possible transitions into the next mode, based on whether the launch detection is enabled in mission configuration. Nominally, the launch detection is disabled and the DOC is turned on by the HV already in orbit; then the transition to next mode is governed by detection of low ambient pressure. If the launch detection is enabled, the DOC is on during the launch and detects it using acceleration measurements. After a launch (or in-orbit environment) is detected, a timer of configurable length would be started, after whose expiration the ASW would transition to *wake-up* mode.

Wake-up mode, during which the ASW would turn on all HW peripheries, initialize them and prepare for measurement and data transmission. After the preparation is finished, the ASW would transition to the next mode; either immediately or after an expiration of a configurable timer (based on the mission configuration).

Re-entry mode, the mode during which the measurement of data, capture of camera photos, and release from the HV would take place. All of the actions would be based only on configurable timers. Once the DOC is released from the HV (and thus its antenna would have an unobstructed view of the sky), the ASW would continually command the Iridium modem to open the data link. After the data link would be successfully established for the first time, the ASW would transition to the last mode.

Descent mode, during which the the most important operation would be to transmit all acquired data to the DRS. Measurement of new data would still take place, but it is foreseen that the data acquired during this mode would have lower priority than the data acquired during re-entry. This mode would then be active forever (or more specifically until the surface impact). Although it is unforeseen, if it would happen that all packets have been successfully sent, it is planned that they would be sent again; essentially, if it were not for the surface impact, the data transmission would never stop.

Maintenance mode, which would be activated once the EGSE connected to the ASW. In this mode the ASW would allow for reception of telecommands and generate responses in form of telemetry. All HW peripheries would be powered on to allow for overall DOC health

check. Since the ASW has no means to detect the physical connection of the EGSE (on the UART interface), it was decided to use the reception of a valid ping packet as the signal to switch into maintenance mode. During their communication, the EGSE would periodically generate keep-alive ping packets to maintain the connection. Once no valid packets arrived from EGSE for more than 3 seconds, the ASW would switch back into *stand-by* mode.

During testing it may be useful to abort a mission run during a specific mode and examine the contents of the on-board memories. Therefore the ASW listens on the EGSE port during the entire mission, and on reception of a valid ping packet aborts the mission and transitions to maintenance mode.

4.2.3 Recovery from reset

One of the requirements on the ASW states that the ASW shall recover pre-reset state and operations if it resets during the mission for any reason (CPU error, brownout, etc.). Although this is unlikely to happen, the ASW should have a contingency prepared.

To facilitate the recovery, at the beginning of each new mode, a current FreeRTOS tick count (starts at ASW startup and increments each 1 ms) and current RTC time measurement are saved into a persistent memory. After a reset, the memory location is checked for valid saved data, and if any are found, the tick count is recovered by comparing the saved RTC time with a current RTC time. In the case when RTC also lose power and thus the measure of real time is lost, the recovery is not possible.

Along with the timing information being saved into persistent memory, each module's information is preserved by allocating important variables and buffers into the on-board SDRAM memory. Although volatile, the contents of the SDRAM persist through CPU reset and are corrupted only when the entire board loses power (same as the RTC). This allows for a complete ASW recovery from reset.

4.2.4 Comparison to PDR design

The preliminary dynamic SW design contained 10 tasks in total, some of which were woken by an interrupt and some by a timer. This was simplified in the finalized design to use only 8 tasks, all of which are woken by a timer.

The mode sequence in the preliminary design contained a post-impact mode, which would be activated in case the capsule survived a landing on land. This was decided to be dropped to simplify the ASW design since the survival of the landing was not foreseen.

The recovery from reset was not planned during the preliminary design and was added during the process of design finalization.

4.3 ASW/DRS communication protocol design

Although the goal of the mission is to measure data, take pictures, and observe the demise of the HV in general, it would be of no use if no data were transmitted to the ground. As the communication time is short and the expected transmission speed very low, high focus was placed on design of a data transfer protocol, which would manage to transfer the largest amount of data to ground in the allocated time. This requires to have a very low overhead, while still enabling for packet identification and packet error control (PEC). This section describes the design of this protocol in detail by each part of the packet.

When designing a communication protocol in a space application, where there is a relatively high probability of packets getting corrupted or lost, it is important to ensure that the packets are independent on each other, i.e. no packet should need any data from another packet to be successfully parsed. This was also one of the requirements on the communication protocol design.

In general, packets are composed of:

Packet header, which uniquely identifies the packet, the type of its contents and can also provide their description

Packet data field, which contains the payload data, and can also contain their metadata

Packet error control (PEC), which enables detection and possibly also correction of transmission errors at the cost of larger overhead.

The overhead of the packet is defined as the amount of bytes which were added to the packet because of the transmission and would not be needed otherwise, i.e. the packet header, PEC, and other metadata. The packet overhead is usually expressed as a percentage of the total packet length.

There are two types of packets: telemetry packets (going from ASW to DRS, described below) and telecommand packets (going from DRS to ASW, described in section 4.3.4).

4.3.1 Packet header design

The purpose of the packet header is to enable the identification of the packet in the data stream, to provide an unique identifier to the packet (sequence number) which would be re-used in case of re-transmissions, and to provide a description of its contents so that they can be easily parsed. In case of ASW/DRS protocol, all fields should be as small as possible.

The most common approach to identify a packet in the data stream is to use a unique starting sequence (and possibly also ending sequence) and encode the rest of the packet so that there is no possibility of it containing the reserved sequences. As decreasing the number of available characters inevitably leads to more strain on the data budget (more bytes are needed to transfer the same information), this option was ruled out for ASW/DRS interface. Instead it was decided that a combination of packet length located at the start of the packet and PEC

located at the end of the packet would be used for packet identification. This means that each byte of the data stream would be examined as a possible packet start, the packet length would be extracted, and a PEC sequence would be located at the suspected end of the packet. If the PEC sequence would match, the packet would be considered valid.

To limit the size of the packet length field in the packet header, it was decided to limit the total packet size to 255 bytes. Due to this, 8 bits (1 byte) were allocated to packet size. This would also ease the parsing of packets from data stream, as the first byte of each packet describes its length.

To calculate the necessary size of sequence number field, it is first necessary to know the amount of packets which can be transferred over the mission lifetime, and therefore the minimum expected packet length. Assuming that on average, the packets will be at least 50 bytes large (actual expected average size of packets is 150 - 200 bytes), the link speed will be 2400 bit/s and communication time 250 seconds, it can be calculated that (at most) about 1500 packets will be sent over the mission lifetime. Therefore, the sequence number field needs to be at least 11 bits long.

To avoid transmitting empty (spare) fields in headers of some packets, only fields which are necessary to be included in every packet are included in the packet header. These are two pieces of information in case of ASW/DRS interface: the type of contents in the data field and the timestamp of the data. There are four possible types of packet data contents (see section 4.3.2), which can therefore be encoded in two bits of data. The timestamp was chosen to be provided as mission time, because it is possible that real time will not be available for the entire duration of the mission (due to e.g. GNSS device failure), and the mission time will also take less space in the packet header as it has smaller resolution. Since the maximum length of the mission (with start defined on transition to wake-up mode) is calculated to be about 12 000 seconds, and mission time has resolution of 100 ms, 17 bits (maximum value 131 072) are necessary to store mission time information uniquely.

Given the above design, the sum of all necessary packet header fields was 38 bits. Since this is two bits short of 5 full bytes and there is an uncertainty in the necessary sequence number (it was calculated based on several assumptions), two bits were added to the sequence number field for a total of 13 bits. This allows for sending of up to 8192 unique packets, or given the same assumed link speed and communication time an average packet size of 10 bytes.

The overview of final packet header design is in table 4.1.

4.3.2 Packet data field design

There are four possible types of data being transferred to the DRS: event reports, housekeeping reports, payload measurement data and camera pictures. Every packet can contain only one of these types of data. The data fields of each of these packet types are described in sections below.

Bytes	Bits	Name	Description
1	8	Packet Length	Total packet length in bytes.
4	2	Packet Type	See section 4.3.2.
	13	Sequence count	Unique for each packet; in case of re-transmission, the original sequence number is used
	17	Mission time	Mission time of when packet contents were captured (not when packet was sent). Defined as zero on change into Wake-up mode. Increments each 100 milliseconds.
variable	variable	Data field	Described in section 4.3.2.

Table 4.1: ASW/DRS telemetry packet header

Event packet data field

Four types of events are defined by the CCSDS protocol: normal/progress, low severity error, medium severity error and high severity error. It was decided to reuse this scheme in the ASW, with events of each category containing no additional information. This allows to represent each event with one byte – 2 bits for description of the severity type and 6 bits for identification of the event of the given severity type (there are less than 2^6 possible events in each category).

To buffer events into an event packet, each event needs to be timestamped with mission time. To conserve packet space, it was decided that to use the following packet structure:

Bytes	Bits	Name	Description
1	8	Event 1	Identification of event 1 (using its severity and enumeration). The mission time of generation of this event is in the packet header.
3	2	Event 2 time offset	Mission time offset of event 2 generation from the mission time in the packet header
	1	Event 2	Identification of event 2 (using its severity and enumeration)
...	...	...	...
3	2	Event N time offset	Mission time offset of event N generation from the mission time in the packet header
	1	Event N	Identification of event N (using its severity and enumeration)

Table 4.2: ASW/DRS event packet data field (containing N events)

Since the size of the mission time offset is 16 bits, two events happening more than 6553.5 seconds apart cannot be contained in the same packet.

Housekeeping packet data field

This type of packet aggregates housekeeping reports (each being 22 bytes long). Since they are generated at a fixed period throughout the mission and the generation time of the first report is known (provided in the packet header), time stamping of each report is not necessary

(the reports will be buffered sequentially into the packet). The structure of the packet for N housekeeping reports is in table 4.3.

Bytes	Bits	Name	Description
22	8*22	Housekeeping report 1	Housekeeping report generated at the mission time given in the packet header.
22	8*22	Housekeeping report 2	Housekeeping report generated fixed (configurable) time after the first report
...	...	...	...
22	8*22	Housekeeping report N	Housekeeping report generated fixed (configurable) time after the previous report

Table 4.3: ASW/DRS housekeeping packet data field (containing N housekeeping reports)

Measurements packet data field

The packet of this type would hold buffered measurements of only one type (data from only one sensor). The identification of the sensor, from which the contained data originated, would be a part of the data field. Since the mission configuration data (including frequency of acquisition of essential/non-essential data points) will be available to the DRS, from the timestamp of the first sample (which is given in the packet header) it will be possible to reconstruct timestamps of all samples contained in the packet (without the timestamps being part of the packet).

Since it is reasonable for vector data (from accelerometer etc.) to be sent together in one packet, data from all three axes taken in one point of time will be buffered together as one measurement.

The measurement data will be compressed to reduce the amount of data being sent. Information about the compression will be saved into one byte in front of the compressed data, to enable their decompression.

Bytes	Bits	Name	Description
1	2	Mission mode	Mission mode in which the measurements were acquired (wake-up, re-entry or descent)
	1	Essential	Whether this packet contains essential or non-essential data
	5	Measurement ID	Identification of the type of measurements contained in the packet
1	8	Compression metadata	Used to successfully decompress the measurements
variable	variable	Compressed measurements	Measurements compressed using one of the compression methods

Table 4.4: ASW/DRS measurement packet data field (containing measurements of the same type)

Camera picture packet data field

A packet of this type contains a photo or part of a photo. It is expected that photos will be too big to fit in one packet (their expected size is about 18 kB)and thus they will be split into multiple packets (each packet can contain up to 247 bytes of camera data).

Each of the packets will contains two 1-byte fields: total number of parts into which the photo has been split, and photo part counter identifying the sequence of the given packet in context of the whole photo. The packets containing one photo will then have the same mission time in the header (time when the photo was taken), the same total number of photo parts, and they will be sequentially numbered (starting with 1) in their photo part counter field. The somewhat redundant data (total number of parts, mission time of photo capture) are sent in each packet to make sure that even if some packets are lost, the photo can still be partially reconstructed.

Since the each packet carrying one photo (apart from the last one) contains 247 bytes of data and maximum number of photo parts is 255, the theoretical maximum size of a photo that can be transferred is 62 985 bytes.

As the photos are already compressed when downloaded from the camera, no further compression is done in the ASW.

Bytes	Bits	Name	Description
1	8	Photo part counter	Sequential number identifying which part of photo the packet contains. Minimum value is 1 and maximum is the total number of photo parts (the next field)
1	8	Total number of photo parts	Number of parts into which the photo has been split (1-255)
variable	variable	Part of photo	This field will be either 247 bytes long (for all photo parts except for the last one) or of variable length (for the last part of the photo)

Table 4.5: ASW/DRS camera picture packet data field

4.3.3 Packet error control design

In general, two approaches can be taken to PEC; either the errors are only detected (e.g. by using well-known CRC algorithms) or Forward Error Correction (FEC) algorithms can be used to repair the errors – at the cost of higher packet overhead.

During Iridium modem testing it came forward that the Bit Error Ratio (BER) of the communication may be up to 3.5×10^{-5}. Assuming average packet size of 200 Bytes, this would mean that on average 5.5% of packets would contain at least one bit error. If a CRC Packet Error Control is used, i.e. no errors can be repaired, 5.5% of packets would have to be retransmitted.

To decrease the number of necessary retransmissions, FEC codes may be used. There is a variety of FEC codes in existence. Commonly used, and well performing FEC codes are block codes, such as the Reed-Solomon codes. These however require fixed packet size, which is not the case with ASW/DRS protocol. Another possibility are convolutional FEC codes, which can handle packets with variable size, but have much larger overhead than block FEC codes (typically 20-50%). Since usage of CRC algorithms would increase the amount of data to be transmitted only by about 5.5% (by packet retransmittings), it was chosen to use CRC as PEC.

Next, an appropriate CRC algorithm was to be chosen. Generally, CRCs are defined by their length and generating polynomial. Both of these values affect the ability of the CRC to detect bit errors. CRC algorithms with the length of 1 byte have the ability to detect 1 bit error at any place in the packet. It is possible they will be able also detect multiple bit errors, however it is not guaranteed in general. CRC algorithms with length of 2 bytes are guaranteed to detect 3 independent bit errors (but at the cost of additional byte of overhead). To simplify the protocol, only algorithms with length in units of bytes were considered.

Given the average packet size of 200 bytes and BER of 3.5×10^{-5}, table 4.6 presents an overview of the probability that the average packet will contain a number of bit errors (up to 3 bit errors). It can be seen that there is only a 0.151% chance of a packet containing more than 1 bit error. Due to the above analysis, CRC algorithm with length of 1 byte (CRC8) was chosen.

A CRC8 polynomial 0xA6 was chosen to be employed based on a recommendation in *Koopman, P.*[13], which states that it is the best possible CRC8 polynomial for packets from 30 B to 256 B in size. For 256 B packets it detects all 1-bit errors, 99.7% of 2-bit errors and 99.6% of 3-bit errors. For 20 B packets it detects all 1-bit errors, all 2-bit errors and 99.5% of 3-bit errors.

Number of bit errors	Probability
0	94.5538%
1	5.2952%
2	0.2409%
3	0.0057%

Table 4.6: Probability of a 200B packet to contain a number of bit errors

4.3.4 Telecommand packet design

There are only two possible types of telecommand: ping or acknowledge. To preserve symmetry with the telemetry packets (and allow the reuse of parsing code), the packet header of telecommands has the same length and structure as the packet header of telemetry. This approach leads to some unused fields being sent over the data link, however this is not critical in this direction (DRS to ASW), as there will not be much data sent (basically only acknowledges of received packets, which are relatively small in size).

Bytes	Bits	Name	Description
1	8	Packet Length	Total packet length in bytes.
4	1	TC packet type	Ping or packet acknowledge
	1	Spare	0
	13	Sequence count	DRS packet sequence count
	17	Spare	0
variable	variable	TC Data field	For ping is empty (0 bytes), for acknowledge contains the sequence count and the mission time of the packet being acknowledged (4 bytes)
1	8	CRC	CRC8 using polynomial 0xA6

Table 4.7: ASW/DRS telecommand packet structure

4.3.5 Packet encoding

As mentioned in section 4.3.1, to minimize the amount of data being sent, no encoding can be done for telemetry packet (in direction from ASW to DRS).

For telecommands (in direction from DRS to ASW) this however may be beneficial, due to the fact that the Iridium modem sends messages to ASW over the same UART as received data from DRS, and thus these two data streams are joined. To enable the ASW to easily recognize and parse modem messages, which are always in the ASCII basic character range (1 - 128), it was decided to encode the telecommand packets so that they don't contain these characters. This is done by splitting each byte of the telecommand into two halves which will be transmitted in two consecutive bytes as lower halves of the bytes, where the top halves will have a value of 0xF0. To illustrate with an example, 0xAB byte will be transmitted as 0xFA followed by 0xFB.

4.3.6 Packet re-transmission strategy

Since it is expected that a number of packets will have to be re-transmitted, a suitable strategy for packet re-transmission was necessary to be prepared.

During the mission configuration, the user will be advised to configure the data acquisition in such a way, that the total amount of acquired essential data will be less than 75 kBytes, with a sufficient margin for packet re-transmissions.

During the mission, the strategy for sending out packets was chosen to be the following:

1. Start sending packets based on their priority. Essential packets are scheduled for sending before non-essential packets. Packets with higher priority get sent before packets with lower priority. In case of equal priority, the packet with lower mission time in header (containing earlier acquired data) will be sent first.

2. Receive and parse packet acknowledges received from the DRS. If some packet is not acknowledged 10 seconds after its sending, schedule it again to be sent based on its priority and mission time.

3. In case all packets have been sent and there are no unacknowledged packets to be resent

at the moment (not expected during mission), all packets (including the unacknowledged ones) will be scheduled for sending again; essentially starting the transmission again from the beginning.

4.3.7 Comparison to PDR design

Originally, a usage of CCSDS protocol was considered to be used on the ASW/DRS interface due to its widespread use. However during the design it was decided to design a custom protocol instead, as the CCSDS protocol has too high overhead (6%) for packets of size up to 255 bytes. Although a simple custom protocol has been proposed before the author took the responsibility for its design, it contained only a description of the packet header with little justification; it also expected a packet retransmit request mechanism instead of acknowledge mechanism. The design has been reworked by the author of this thesis including justification of sizes of all fields, the data field design, the choice of PEC algorithm and a packet acknowledge strategy.

Chapter 5

DOC Software Development

This chapter describes the development of the ASW, which was done in cooperation with multiple members of the ASW development team. The changes made to the design during the development are described in section 5.3. The documentation produced by the author is then described in section 5.4.

5.1 ASW development process

The first step in the ASW development process was to derive SW requirements from system requirements (which are outlined in section 3.3) and design the static and dynamic architecture (described in chapter 4). The SRS was subject to a PDR, after which the ASW development could start.

At the start of development, the external SW libraries including FreeRTOS and hardware drivers were provided to the ASW development team. The provided external SW library was already flight-proven; however it lacked a proper interface description (an ICD document). This proved to be problematic later in the development (this is described in the following sections).

Although SW development is possible without having access to the actual flight hardware, it is very beneficial to the development process if access to at least a reduced version of the flight hardware is provided; then a reduced integration testing can be performed throughout the ASW development. Due to this, a development board identical to the one which will be used in the flight hardware (without any HW peripheries attached) was also provided.

5.1.1 First stages of ASW development

The actual ASW development was started by work on the EGSE interface. By enabling and testing UART communication, it became possible to communicate with ASW from the outside, and it also enabled the usage of debug console. Although it is advised against a usage of a debugging console (and later in the development the console indeed caused problems with scheduling), it proved itself as a valuable tool. Although SW crashes could be examined via a debugger, the debugging console (whose code was reused from other projects) enabled the

development team to discover problems in the ASW which didn't crash the software or produce events.

The actual communication with the EGSE would be based on the CCSDS protocol (as mentioned in section 3.2.3). Although at first the communication protocol was being written from scratch to enable better understanding of its function, this has proven to be ineffective and prone to bugs; therefore a code from other projects using CCSDS protocol has been finally adopted. Several changes were necessary in the reused code to make it comply to ASW coding standard and the planned extent of service usage in the ASW/DRS communication.

After the UART interface and basic CCSDS protocol communication with the EGSE (which was developed in parallel) was established, the readout of data from the sensor board and the mechanism to provide them to the EGSE via telemetry was developed next. First problems with the SW library were encountered here as it was used to download the measurements from the sensor board; however it was unclear what format and range/resolution were the data passed on to the ASW (due to lacking documentation). This was resolved by contacting the SW library provider, which then provided an update to the library resolving this issue.

With EGSE interface and sensor readout in place, the *operational manager* SW package could be developed. This included *mode management* module which decided on mode transitions based on measurements from the sensor board and the status of EGSE connection; *mission management* which initialized the ASW on startup and then operated the ASW during mission (at this point in time of the development there was nothing to do apart from the mode switching); *time* module providing timing information and mission time keeping; and *scheduler* defining and starting ASW tasks.

At this point of time it was possible to connect the EGSE and receive and display the measured data from sensor board. The ASW was able to receive other telecommands too, however it lacked the functionality to fulfill their requests (such as a memory dump). ASW was also able to transition through modes; however lacked the functionality to perform required actions on mode transitions.

5.1.2 Middle stages of ASW development

In next stages the ASW development followed the data flow, starting from the already implemented sensor board readout. During this stage, the target flight hardware including all HW peripheries was available for remote connection and programming. This enabled reduced integration testing for nominal functionality during the development.

In the *sensor readout* module, which was already partly developed, the mechanism for synchronization of data from multiple sources and functions for extraction of essential and non-essential data were developed. These data point were then passed to a queue, which would be read out by *data packetizer*.

The extraction of essential and non-essential data requires data acquisition settings as an input, and thus the mechanism for mission configuration was added to ASW; namely to the EGSE interface, to *data storage* module, which would save and load the configuration from

memory, and to *mission management* module, which would provide the configuration to the *sensor readout* module based on the current operational mode.

The configuration and readout of data from the GNSS device was implemented based on the manual provided by the device manufacturer. The management module for the last device providing (housekeeping) data, the EPS, was also implemented. Here a problem was encountered where data packets coming from the EPS would get lost or corrupted, which was eventually linked to the debugging console interrupting the packet reception process for too long and thus corrupting its reception.

Next, the *data packetizer* and *data prioritizer* modules were developed, and *data storage* module was extended to allow for writing and reading data to/from the flash memory. Memory dump/load services were also implemented in the EGSE communication.

Apart from payload measurement data, housekeeping data are also acquired and collected from HW peripheries (SB, EPS, GNSS) and some ASW modules (data storage, modem control, mission management). A mechanism collecting and synchronizing these data was also added to *sensor readout* module. Since now the *sensor readout* module contained too many functions, the function related to housekeeping were moved to a dedicated module *housekeeping acquisition*.

The *release management* module was also implemented. Although originally the release function just checked if all necessary conditions for release are met and then fired the release mechanism, later it was changed to contain *arm and fire* logic, i.e. it is not possible to fire the release mechanism if it has not been previously armed in an independent operation.

At this stage, the ASW was able to go through the mode sequence, turn off/on the on-board HW peripheries, measure data based on the mission configuration, packetize them and save them to memory. It was also possible to use the EGSE for a HW health check (by observing housekeeping data) and configure the ASW.

During this stage, a first formal delivery of the ASW was performed as it was requested by the customer. Starting at this time, weekly snapshots of the ASW codebase were started to also be provided to the customer at the end of each week, tracking the development process.

5.1.3 Final stages of ASW development

With the core ASW functionality in place, the modem communication was next to be implemented. First, the initialization procedure of the Iridium modem using AT commands was developed. This was done using a state machine, where in each state, a new command would be sent to the modem. On reception of *OK* answer, the initialization would move to the next state. The final step in the initialization then was the dialing of the remote (DRS) number and thus opening the data link. Second, the functions for loading packets from memory, preparing them for sending and selecting the highest priority packet were developed. Finally, these functionalities were connected in the *modem control* module by decision logic and an algorithm to parse and process acknowledges.

The implementation of *camera management* and *HAL camera* modules has proven to be problematic. During development it was discovered, that the SW library allows for download

of camera pictures only with a usage of a filesystem on the client system (which the ASW didn't use) and doesn't offer any functions for download directly to memory. Therefore it was necessary to implement a file transfer protocol for camera pictures. Although an example code was provided by the SW library developer, it had to be significantly changed to the needs of ASW. Additional delays in development were caused by the insufficient documentation of the SW library.

Another problem which surfaced during this time of the development was that the SW library offered only blocking (not interrupt driven) function for sending messages to UART. This has blocked the schedulability analysis, as during descent mode, when packets were being sent to the modem, the UART imposed a high and unpredictable load to the CPU. This problem was resolved by contacting the SW library provider, which included an interrupt-driven function in the next library release.

The only two missing functionalities at this point of time were the data compression and reset recovery algorithms, as they were not mission critical. It was chosen to use only several simple compression algorithms (such as *common MSB* or *min + delta* compression); each packet would then be compressed by the algorithm producing the compressed packet with smallest size. Data recovery was implemented as is described in section 4.2.3.

At the end of the development it was necessary to refactor most modules to improve their readability and compliance with coding standard. To make unit testing easier, large functions were split into smaller ones. Several modules were also split (see section 5.3).

At the time of writing of this work the ASW is completed functionality-wise, under unit testing (see next section) and heading towards the CDR.

5.2 ASW testing

Two types of testing were done during the ASW development: unit testing and integration testing.

Unit testing in this context is defined as testing of each ASW function. It is aimed for 100% line coverage and also 100% branch coverage (each clause in an if statement is to be tested, not relying on short-circuit evaluation). For unit testing, a custom platform developed in-house was used. It allowed the development team to automatically generate function stubs, implement the tests in a comprehensive manner, and to auto-generate a complete unit test report as a .csv document. Although unit testing should be a part of the development process since the beginning, to avoid having to change the unit tests due to ASW refactoring, the unit testing was postponed to the end of the development.

A reduced integration testing was done throughout the ASW development – testing for nominal functionality on the target hardware. This was done to make sure that the developed code works on the target HW and performs the required operations correctly. A complete integration testing including negative test (intentionally triggering error states to see if they are handled correctly) was performed at the end of ASW development. To enable these tests, the

ASW binary was modified and therefore they couldn't be considered for SW qualification; they were only used to increase confidence in the ASW and catch bugs early on.

For SW qualification, only unit tests and end-to-end (EtE) test are considered. The EtE tests will be done with the final ASW binary on the target hardware, and will test the ASW operation using only external interfaces, i.e. consider the ASW to be a black-box. Using the EtE tests, the ASW will be validated against the SRS. The EtE tests are planned to take place after the publication of this work.

5.3 Architecture changes during ASW development

5.3.1 Static architecture changes

During development the static architecture didn't see many changes from architectural point of view, however many modules were split into smaller ones to improve the readability and modularity of the ASW. The modified modules are:

- *EGSE protocol*, which got split into three modules adapted from other projects
- *EGSE control,* which got split into itself plus 9 more modules (one module for each CCSDS service). The *EGSE control* module itself now contains a switch, which will pass the packet to the appropriate module for processing based on its service number
- *Data Storage,* which was split into itself (now only performing top-level logic, SDRAM operations, and containing a memory write task), *config* module which stores mission configuration, *packet storage* which specializes in storing packets to be sent to DRS, and *simulation* module containing mission simulation data and functions to access them
- *Data compression,* which now only selects the best performing compression algorithm for each packet; the algorithms themselves are in *compression method* module
- *Modem control,* which kept only the overall logic; the functions for sending packets were moved to *modem send* module and the functions for receiving and parsing incoming packets from DRS were moved to *modem receive* module

Two new device management modules (and their corresponding HAL modules) were added: *Flash management* and *FRAM management*. These modules hold information about structure of data in their corresponding memories and call the HAL functions for memory access.

The *FDIR* module, which was planned to be included in *operational management* SW package has been dropped; it was decided that FDIR functionality would not be helpful in context of DOC mission (short mission duration, almost no redundancy) and would only unnecessarily increase ASW complexity.

5.3.2 Dynamic architecture changes

Based on recommendations from a SDD review, the mode sequence was divided into more granular system states. This allows better control over the ASW behavior and makes testing easier, as each system state has a clearly defined behavior. There are 10 system states identified in the final design, through which the ASW passes in a linear manner:

1. Start-up: during this state the ASW goes through initialization; self-tests are performed and all ASW tasks are started
2. Ready: in this state the ASW is conserving power and waiting for launch detection
3. Launch detected: if launch detection is on, the ASW goes into this state on detection of launch acceleration. If it is off, the ASW transitions into this state on detection of in-orbit pressure
4. In-orbit: the ASW progresses to this state on detection of in-orbit pressure. HW peripheries are still turned off and conserving power
5. Wake-up: during this state the ASW turns on and initializes HW peripheries
6. Re-entry attached: in this state the ASW measures data and acquires pictures using the external camera
7. Departure: this state is activated at the moment of release from HV. Internal camera pictures are taken
8. Re-entry detached: after the separation and departure from HV, ASW starts trying to acquire data link with DRS. Data are continued to be measured
9. Descent: the ASW transitions to this state once the data link is open. This state focuses on data transmission; however data are still continued to be measured. There is no nominal transition from this state.
10. Maintenance: activated once EGSE connects. Can be activated from any system state. On EGSE disconnect, the ASW transitions to Ready system state.

The overview of system states is depicted in figure 5.1.

In comparison with the planned list of ASW tasks given in section 4.2.1, it was decided to increase the number of tasks in the data acquisition and processing modules, to allow for different data readout mechanisms and frequencies. In result, the *sensor readout* task was divided into six tasks; one for each data source, i.e. *sensor board task*, *GPS task*, *EPS task*, *Housekeeping acquisition task*, and *Cameras task*, and finally itself, i.e. the *sensor readout task* was preserved to serve as a central collection point of data from all other data acquisition tasks, and to perform the essential/non-essential data extraction. Although this increased the number of total tasks by five, it allowed for better modularization of ASW, making each part of the data acquisition process independent on each other.

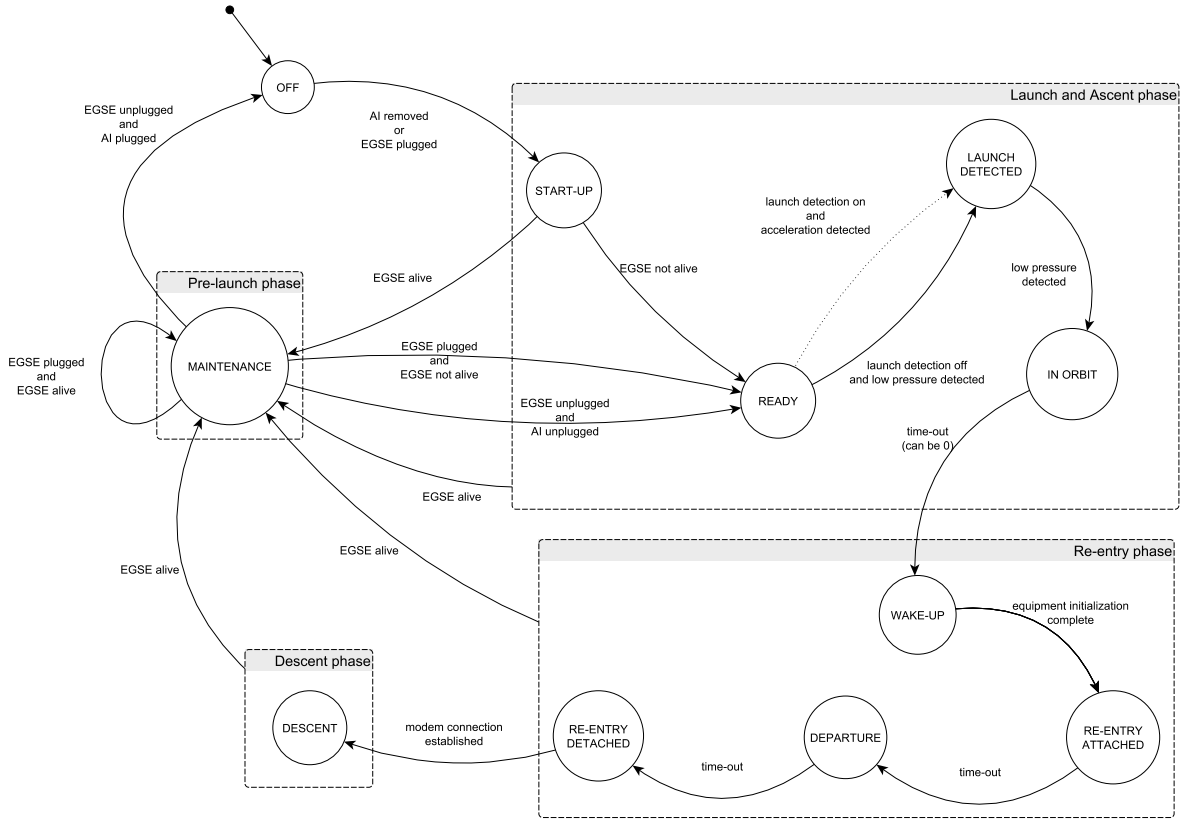

Figure 5.1: Overview of ASW states

5.4 Produced documentation

The expected documentation accompanying an ASW release was outlined in section 2.6.4. As of writing of this work, the following documents have been produced:

- SDD, describing the ASW static and dynamic architecture. A code documentation generated by Doxygen tool is attached to this document as an annex
- ICD, describing all external ASW interfaces
- SW justification file, which describes trade-offs and decisions taken while designing the ASW and its external interfaces
- SUM, describing the ASW and EGSE operation procedures
- SRelD and SCF, updated and released along with each official release of the ASW
- SRS, containing the derived requirements.

The documents listed above are only the technical documentation describing ASW, which has been prepared by the author of this thesis. In addition to the listed documents, many more management documents, test plans and reports, technical notes etc. have been produced;

however most of these serve only for informational purposes and only a subset of the most important documents is sent to the customer for CDR.

Chapter 6

Results

6.1 Final ASW description

In final version, the ASW is able to measure payload data, collect housekeeping, take pictures with on-board cameras, and send collected data over the Iridium link based on the configured priority. It can be configured by an EGSE and can recover from unexpected reset during the mission. It fulfills all customer requirements.

During mission, the ASW transitions through a linear sequence of states (i.e. works as a state machine). Each state has a defined ASW behavior and transition to the next state. The description of the states was given in section 5.3.

The static architecture is, from an overview, still the same as depicted in figure 4.1, i.e. *operational manager* SW package handles the ASW operation and decision making, *data handling* SW package acquires and processes payload data, *modem communication* package sends the acquired data over Iridium link, *EGSE communication* package allows for ASW checkout and configuration, and *device drivers* allow for separation of ASW from HW and SW library. Each SW package is composed of multiple modules; although in final version there is larger number of more granular modules, the initial design was more or less close to what the final design achieves.

As for the dynamic architecture; the number of ASW tasks also saw an increase (granularization) from the initial design, but otherwise the initial design was adhered to. In final version the ASW contains 12 tasks; however each performs only a small, clearly defined operation and goes to sleep in a short time, allowing for schedulability of ASW.

An overview of the ASW architecture (static and dynamic) is given in figure 6.1. It depicts all ASW modules grouped into SW packages. As picturing the dependencies of all modules would result in a visual clutter, only most important data flow dependencies are shown (payload data flow as pictured in figure 4.2 in red). The modules depicted in blue contain their own dedicated task.

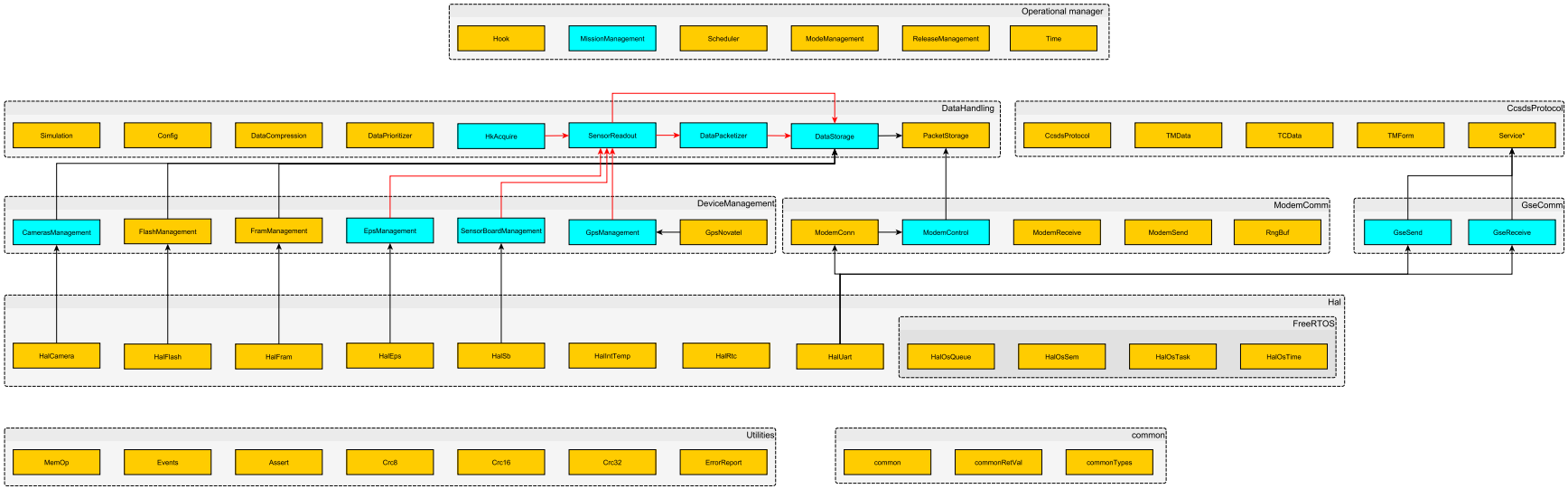

Figure 6.1: Overview of final ASW architecture

The protocol for sending payload data from ASW to DRS via Iridium was designed to have only 6 bytes of overhead for each packet. Given the maximum packet size of 255 bytes, this makes the overhead 2.4% at the best case and 3% at nominal case (average packet size of 200 bytes). This allows the ASW to send a larger total amount of data to the DRS; at the cost of harder packet detection at DRS and a possibility of false positive packet detection (described in the next section).

6.2 Results of ASW testing

As was described in section 5.2, there are three kinds of testing being done on the ASW:

- Unit testing, which is testing each function at full line and branch coverage
- Integration testing, which tests larger logical parts of the ASW and their interaction on the target hardware
- End-to-end testing, which tests the ASW for its behavior on its external interfaces.

As of writing of this thesis, the unit testing is still in progress. It has shown itself as a handy tool not only for identification of bugs in functions (which are subsequently fixed), but also for identification of poorly written code. Even though in theory in order to unit test a function, only its header and documentation shall be seen, in practice an insight into the function is necessary to properly test it for all possibilities and full branch coverage. Since understanding of the function is necessary in order to unit test it, coding standard violations and unnecessarily complex constructs are quickly identified, and the functions are refactored, simplified and clarified as a part of the unit testing process. This results in a readable and robust code.

As part of the ASW development, integration tests were gradually developed to test functionality of various parts of ASW as they were being developed. This includes testing of parts of the data flow, such as the readout of the sensor board, the packetization of measurements, the sending of packets over Iridium link etc. Preparing a complete set of integration tests allowed to make sure the code works functionality-wise, and re-testing after each change in the relevant part of code allowed for quick catching of newly introduced bugs. As of writing of this thesis, the ASW was complete functionality-wise, and all integration tests are being passed.

From customer's point of view the most important kind of tests are the EtE tests, which serve to prove the ASW compliance with requirements on its behavior, and the ASW qualification. Since the ASW can be commanded only via the EGSE interface, there are basically only two possible kinds of EtE tests: EGSE interaction tests and mission simulation tests.

For testing ASW/EGSE interaction, a custom platform for CCSDS protocol communication testing was used. Using this platform, telecommands of all possible CCSDS services and subservices were generated, including invalid packets to test general packet reception and identification. The telemetry packets received back from the ASW are then compared to expected responses.

Since memory dump and memory load services are implemented, almost all of the ASW functionality can be tested by manipulating and examining the on-board memories. The complete set of tests of all possible packet types takes about five minutes on the target architecture. The ASW passes all of the prepared ASW/EGSE tests.

Due to an implementation of a mission simulation mode, it is possible to upload mission simulation data to the ASW via the EGSE and start the mission simulation. During it the measurements are not read from the sensors and HW peripheries, but from the on-board memory instead. This allows to test the mode sequence, packetization of payload data and the Iridium data link in the same manner as they will be used in a real mission. The mission simulation will be used during the EtE testing campaign, which will take place before the CDR delivery.

During testing of the ASW/DRS communication it was discovered that the Iridium network injects a number of bytes between each two packets. This results in some false positive packets being detected by the DRS, and in turn acknowledges with nonsensical data being sent back to the ASW. This is however not an issue, as the DRS contains a mechanism to detect possible false positive packets based on expected data in packet header, and the ASW can detect and ignore acknowledges for non-existent packets.

6.3 Results of QA/ISVV reviews

At all times during the development, the design documentation, development process and the code itself was under the supervision of a QA manager. His work ensured that the code is designed and developed following the ECSS standards and coding standard, as well as other quality milestones. The (automatically, by a script) tracked SW quality quantities were for example:

- Number of compile-time warnings (with the compiler set to be very strict)
- Number of unwanted constructs (such as gotos, forbidden variable types, coding standard violations etc.)
- Number of todos in code
- Number of lines of code, comments and blank lines
- Number of warnings and infos (suspicious places in code) produced by an automated code analyzing tool *pclint*

In addition, the code underwent manual visual inspection ensuring it is well commented and easy to understand. The fixing of the issues raised by the QA manager was a common activity during the development and a part of a normal workflow. At the end of the development process, the ASW produces no compile warnings, contains no todos (except on testing) or unwanted constructs, and the warnings and infos produced by the *pclint* tool are in a process of being resolved (the goal is to inspect and/or fix all suspicious places in the code).

During selected milestones, the design documentation and the ASW code at its current state were sent to the ISVV contractor for a design and code review. During early stages of the development, the ISVV provided feedback on the planned architecture and the documentation itself, which prompted their fixing and/or improvement. At about the middle of the development, when first formal release of the ASW was performed, the first code review took place. It's results were unfavorable to the ASW, as at the time there wasn't much effort put into clarity of the code and the documentation or adhering to the coding standard. The results of this review prompted the development team to put more focus on coding and documentation quality, resulting in a positive ISVV design and code review near the end of the development.

6.4 Lessons learned

This section sums up lessons learned during the design and development of the DOC ASW and provides them to the reader as an advice for their future space-qualified SW projects.

1. During early stages of the project, take time to make a proper, robust, and detailed design. This will save time later in the project (already written code will not have to be changed).

2. Similar to the advice above, write a complete and detailed coding standard at the beginning of the project (adopting some widely used coding standard is preferable). Changes to the coding standard during the development are costly and time-consuming.

3. If you use external SW libraries and HW drivers, make sure they have been thoroughly tested and that they are completely documented. We have spent a lot of time going back and forth with the SW library provider on various issues, only because the library wasn't properly documented. Don't build your space-qualified house on sand.

4. Insist on having a functional copy of the flight hardware in-house for development use. HW providers prefer to have only one model and allow the SW developers remote access to it; however the remote access can be available only during some days/times, there might be problems in the HW the SW developer is not aware of, there might be conflicts when two people try to access the HW at the same time etc. Having the HW on a table is always better for development.

5. Make sure the SW requirements and design are fixed and approved by the customer before you start the development.

6. Use ticketing systems to track every issue, including todos in code, discovered bugs that cannot be fixed immediately, but also issues which you reported to an external party (customer, other SW provider) and are waiting for them to fix. It is much easier to have these issues in one place than to sift through sent e-mails and searching for them.

7. Have one person in the development team who has an overview of the entire project. This helps to avoid creation of modules and function which are useless to the project,

and avoids writing of the same code by two developers independently (for example in our project both the person who wrote modem communication and the person who wrote GNSS communication wrote a function which searches for a pattern in a buffer).

Chapter 7

Conclusion

The goal of this thesis was to design a space-qualified flight software, which would follow ECSS-E-ST-40C standard on space software engineering and customer requirements. The adherence to the standard and the requirements was under supervision by in-house QA and an ISVV project consortium member.

When the author of this thesis joined the project in June 2016, only preliminary ASW design was produced. Under the responsibility of the author the design was completed and finalized and the ASW was developed (with significant part developed by the thesis author) and tested on target hardware.

The static architecture of the ASW follows a fairly standard layered architecture, with decision logic being in the top layer, data handling and communication modules in the middle layer and device drivers in the bottom layer. The payload data flow in a linear way through the data handling modules, are compressed into packets, and passed to the communication modules for data transmission. In total, the ASW is composed of 62 modules.

The dynamic architecture design followed widely known rules for robust real-time SW design. In total, 14 tasks are being run on the ASW; most of these have a singular purpose and finish their operation quickly. FreeRTOS has been used for task switching and synchronization means.

The ASW is designed as a state machine, going through a linear sequence of 9 states during the mission. An additional 10th state (maintenance) is activated upon connection of the EGSE, which allows for ASW health check and configuration.

For transmission of payload data to the ground segment a custom protocol has been developed. It adds only 6 bytes of overhead to each packet, with expected packet size of about 200 bytes.

Even though at the time of the author joining the project the ASW design and development was delayed with respect to the project schedule, at the time of writing of this thesis the ASW is functionally finished and will be delivered on time for CDR. The ASW was presented to the customer (S[&]T b.v. and ESA) during a pre-delivery check, where the customer expressed their appreciation for the development progress. The DOC is expected to perform its first mission in late 2017 or early 2018.

Used Abbreviations

Abbreviation	Meaning
ASW	Application Software
AR	Acceptance Review
BER	Bit Error Ratio
CAN	Controller Area Network
CCSDS	Consultative Committee for Space Data Systems
CDR	Critical Design Review
CPU	Central Processing Unit
CRC	Cyclic Redundancy Code
DOC	Demise Observation Capsule
ECSS	European Cooperation for Space Standardization
EGSE	Electrical Ground Support Equipment
EPS	Electric Power Supply
ESA	European Space Agency
EtE	End-to-End (testing)
FDIR	Fault Detection, Isolation, and Recovery
FLPP	Future Launchers Preparatory Programme
FSW	Flight Software
GDB	GNU Debugger
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
HAL	Hardware Abstraction Layer
HV	Host Vehicle
HW	Hardware
ICD	Interface Control Document
IMU	Inertial Measurement Unit
IDE	Integrated Development Environment
ISVV	Independent Software Verification and Validation
MCU	Micro-Controller Unit
MDR	Mission Definition Review
MSB	Most Significant Byte

Abbreviation	Meaning
OBC	On-Board Computer
OS	Operating System
OSRAP	On-board Software Reference Architecture for Payloads
PA	Product Assurance
PDR	Preliminary Design Review
PEC	Packet Error Control
PRR	Preliminary Requirements Review
QA	Quality Assurance
QR	Qualification Review
RID	Review Item Discrepancy
RTOS	Real Time Operating System
SAVOIR	Space Avionics Open Interface Architecture
SB	Sensor Board
SCF	Software Configuration File
SCM	Software Configuration Management
SDD	Software Design Document
SRelD	Software Release Document
SRR	System Requirement Review
SRS	Software Requirements Specification
SSS	Software System Specification
SUITP	Software Unit/Integration Test Plan
SUM	Software User Manual
SVerP	Software Verification Plan
SVR	Software Verification Report
SW	Software
TC	Telecommand
TM	Telemetry
WCET	Worst Case Execution Time

References

[1] Andrew HUNT and David THOMAS. *The Pragmatic Programmer*. 1st ed. Addison Wesley, 1999. ISBN: 0-201-61622-X.

[2] Richard N. TAYLOR et. al. *Software Architecture*. Foundations, Theory and Practice. John Wiley & Sons, Inc., 2010. ISBN: 978-0470-16744-8.

[3] Len BASS et. al. *Software Architecture in Practice*. 3rd ed. Pearson Education, Inc., 2013. ISBN: 978-0321-81573-6.

[4] *ECSS-E-ST-40C*. Space Engineering - Software. Mar. 2009. URL: `www.ecss.nl`.

[5] *ECSS-Q-ST-80C*. Space Product Assurance - Software. Mar. 2009. URL: `www.ecss.nl`.

[6] *ECSS-M-ST-10C*. Space project management - Project planning and implementation. Mar. 2009. URL: `www.ecss.nl`.

[7] *Space Engineering*. Software Engineering Handbook. ESA Requirements and Standards Division, 2013.

[8] Viktor BOS. "On-board Software Reference Architecture for Payloads". In: esc Aerospace s.r.o., 2015.

[9] Richard BARRY. *Mastering the FreeRTOS Real Time Kernel - a Hands On Tutorial Guide*. Real Time Engineers Ltd., 2016.

[10] *ESA - Launcher systems and technologies*. URL: `http://www.esa.int/Our_Activities/Launchers/New_Technologies/Launcher_systems_and_technologies` (visited on 03/05/2016).

[11] *S&T - Demise Observation Capsule (DOC) for re-entry science and safety*. URL: `https://www.stcorp.nl/highlights/demise-observation-capsule-doc-re-entry-science-and-safety/` (visited on 03/05/2016).

[12] *ECSS-E-70-41A*. Space Engineering - Ground systems and operations - Telemetry and telecommand packet utilization. Jan. 2003. URL: `www.ecss.nl`.

[13] P. KOOPMAN. "Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks". In: DSN-2004.