NTNU
Norwegian University of
Science and Technology

Software Design of an Onboard Computer for a Nanosatellite

Magne Alver Normann

Master of Science in Cybernetics and Robotics	
Submission date:	July 2016
Supervisor:	Amund Skavhaug, ITK
Co-supervisor:	Roger Birkeland, IET

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Code as if whoever maintains your program is a violent psychopath who knows where you live..
-Anonyomous

Executive Summary

This thesis presents the development and implementation of a software architecture for an Onboard Computer for a nanosatellite. The process of developing the OBC software architecture is described in stages from software requirement analysis to test and verification of an implementation.

Based on the results from the testing, it is believed that the proposed solution, can satisfy the constraints imposed on the NUTS OBC software through the use of service-oriented architecture based on the use of CSP for internal as well as external communication. The architecture enables independent development of services through standardised interfaces. This can greatly ease system integration as well as the implementation and rerouting of redundant services. The computational overhead as well as the added latency on inter-thread communication is analysed and the solution considered cost efficient.

Parts of this thesis have been accepted for oral presentation and publication at the European Space Agency (ESA) 4S Symposium 2016. The ESA 4S submitted paper is included in Appendix E.

Sammendrag

Denne avhandlingen beskriver hvordan en programvarearkitektur ble utviklet og implementert for en Onboard computer (OBC) for en nanosatellitt. Utviklingsprosessen for programvarearkitekturen blir beskrevet i trinn, fra kravsanalyse til testing og verifisering av implementasjonen. Basert på resultatene fra testingen, postuleres det at den presenterte løsningen kan tilfredsstille kravene som er pålagt NUTS OBC programvaren ved å bruke tjenesteorientert progamvarearkitektur som muliggjøres gjennom bruk av Cubesat Space Protocol (CSP) for intern så vel som ekstern modulkommunikasjon. Arkitekturen legger til rette for uavhengig utvikling av tjenester gjennom standardiserte grensesnitt. Dette kan lette systemintegrasjon samt implementering og omruting av redundante tjenester. Den ekstra beregningskostnaden, så vel som den ekstra ventetiden som følger bruk av CSP for kommunikasjon mellom tråder analyseres og den endelige løsningen vureres som en kostnadseffektiv realisering av OBC promramvarearkitekturen.

Deler av denne avhandlingen har blitt akseptert for muntlig fremføring og publisering på Den europeiske romfartsorganisasjons (ESA) 4S symposium 2016. Den innsendte rapporten kan sees i Vedlegg E.

Acknowledgements

I would like to thank my supervisor Amund Skavhaug, and co-supervisor Roger Birkeland for all the help and advice you have offered throughout my project work and master thesis. I would also like to express my deep gratitude to my fiancee Martha, and my brother Morten for their endless support, understanding and rubber ducking.

I also want to thank the members of the NUTS team for all their individual contributions to this thesis, as well as the entertaining discussions, theories and stories which have contributed to making my last year at NTNU such an adventure. At last I want to thank all my friends and family for their encouraging motivation, and the support they have offered during this semester.

Problem

This work is part of the Norwegian University of Science and Technology(NTNU) Test Satellite (NUTS) project which aims to bring forth a double CubeSat through the work of master students at the Norwegian University of Science and Technology.

The task in this assignment is to research and design, given the existing hardware architecture of NUTS, a software architecture for the NUTS OBC system, with particular focus on developing a system that allows isolation of work tasks while maintaining easy integrability to facilitate further development of the NUTS Satellite computer system.

The OBC is one of the principal components of the satellite, and is able to control the rest of the system by granting or denying subsystems access to power and the databus. Other OBC tasks include logging of system parameters in addition to preparing and reading data transmitted to and from the communication systems.

The OBC must be designed to be reliable, as maintenance is impossible after launch. Challenges related to reliability, maintainability and resource constraints, as well as challenges related to student-driven development must be identified. In areas where mitigation of such problems is possible, solutions should be presented. Whether the solutions are to be implemented should be based on a cost/benefit analysis.

Key tasks for the student:

- The project should explore and determine the requirements for the OBC software in order for it to meet the requirements for the NUTS mission.
- The project should research different types of software architectures and compare their strengths and weaknesses, and present a justified choice for the NUTS mission.
- The project should also outline a more detailed design that can be supported by the software architecture.

In addition to the given tasks, the student is expected to participate in relevant group work. The NUTS project is a multi disciplinary project, which requires more involvement from the student than just the completion of the individual task and report.

Supervisor: Amund Skavhaug
Co-supervisor: Roger Birkeland

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, the reader will be introduced to the Norwegian University of Science and Technology (NTNU) Test Satellite project, as well as the scope of this thesis.

1.1 Project Background and Motivation

In 2006, Norwegian Centre for Space-related Education (NAROM), the Norwegian Space Centre (NSC) and Andøya Space Center (ASC) decided to initiate a student satellite program; The Norwegian Student Satellite Project (ANSAT). The goal for this initiative was to launch three CubeSats within 2014 [3]. Three projects were started, one at Høgskolen in Narvik (HiN), one at the University in Oslo (UiO) and one at the NTNU [4]. The satellite project at HiN, HiNCube, was completed and launched 21st of November 2013 [5]. Unfortunately, contact could not be established with the satellite after the launch [6]. The HiNCube is, to date, the only satellite to have been launched from the ANSAT project, and although the time frame of the initial initiative is done, a successful student satellite has yet to be launched.

1.2 The NTNU Test Satellite (NUTS)

The satellite is a double CubeSat, measuring 10 cm x 10 cm x 20 cm and weighing less than 2.66 kg, which conforms to the CubeSat Standard. The satellite will carry a high definition camera for earth observational purposes.

The satellite consists of:

- Electrical Power System (EPS)
- Altitude Determination and Control System (ADCS)
- Ultra High Frequency (UHF) Radio
- Very High Frequency (VHF) Radio
- On Board Computer (OBC)

- Payload Camera
- Communication subsystem
- Antenna
- Mechanical Structure

The EPS is connected to solar panels and batteries, and is responsible for supplying the power bus with enough electrical power to drive the satellite. The ADCS is entrusted with the de-tumbling and pointing of the satellite. A communication subsystem is implemented as a backplane and is used as a power and communication bus so that module cards easily can be plugged in and out without disassembling the whole satellite. The satellite is equipped with both a UHF radio and a VHF radio so that failures in one radio will not cause a mission failure. When it comes to the mechanical structure, this project has decided not to use the standard aluminium frame but rather aims to utilize composite materials (carbon fiber/epoxy). Although some minor components have been made of carbon fiber in the past, launching a CubeSat with an all-composite primary structure has not yet been done. As a gateway between the radios and the satellite bus, is the OBC.

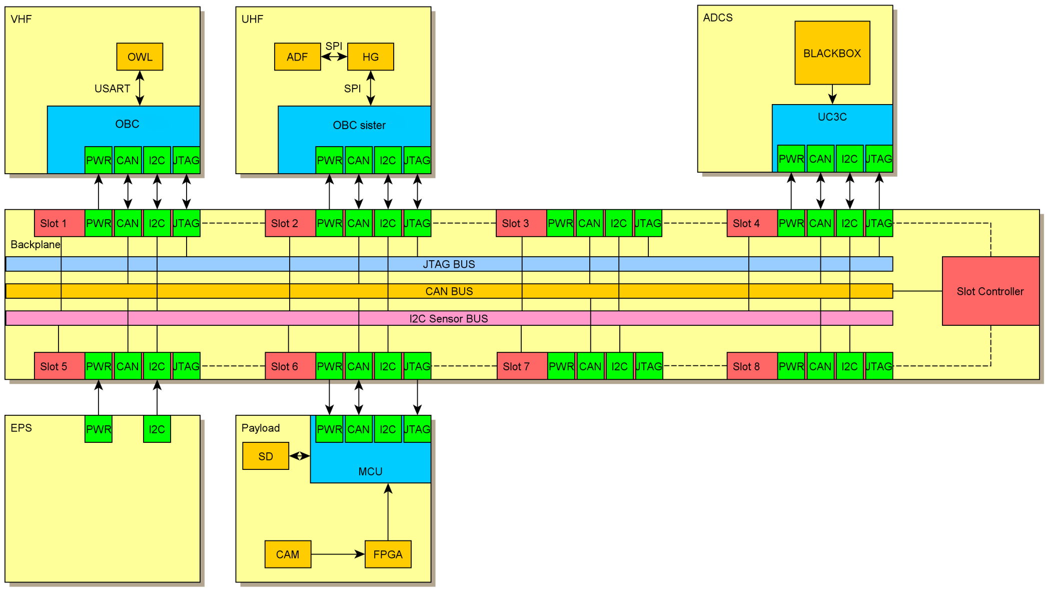


Figure 1.1: The NUTS modules

1.3 The Purpose of the Onboard Computer (OBC)

The main responsibility of the OBC is to monitor the health of the system and to take necessary actions when situations demand for it. It monitors the health of the satellite by periodically requesting health packages from software instances as well as polling sensors for the different modules' power consumption. It also monitors the satellite battery power level, and sets the satellite state appropriately. In addition to this the OBC also acts as a gateway between the satellite bus, and the radio link to ground station. The satellite is designed as a distributed system with redundant functionalities implemented in different modules. The OBC has a sister microcontroller at the UHF Radio module, with a very similar hardware and software setup. OBC functionalities are therefore also implemented in the UHF Radio module. The two modules rely on cold redundancy, meaning that only one of the modules will take the role as an OBC at a time. Both OBC microcontrollers are connected to a Joint Test Action Group (JTAG) standard bus, and have the ability to reprogram the other Microcontroller Unit (MCU) if necessary.

1.4 The NUTS OBC Hardware

The hardware of the OBC consists of the OBC microcontroller connected to various memories and communication interfaces. The module containing the UHF Radio is populated with the ATSAMV71Q21 microcontroller, while the module containing the VHF radio is populated with the AT32UC3C0512C. For mass-storage a flash bank is connected though SPI, for storing of critical data, a FRAM memory chip is connected to the External Bus Interface (EBI). Also connected to the EBI is the SRAM chip allowing the microcontroller a larger non-volatile memory space. To be able to communicate both with the ground station and the rest of the satellite, the OBC MCU is connected to a radio via USART and the internal sattelite bus though a CAN tranciever. For housekeeping purposes the OBC also has access to a I2C sensor bus for monitoring of the application, as well as a JTAG bus for in-orbit reprogramming of faulty modules. The setup of the OBC hardware can be seen in 1.2.

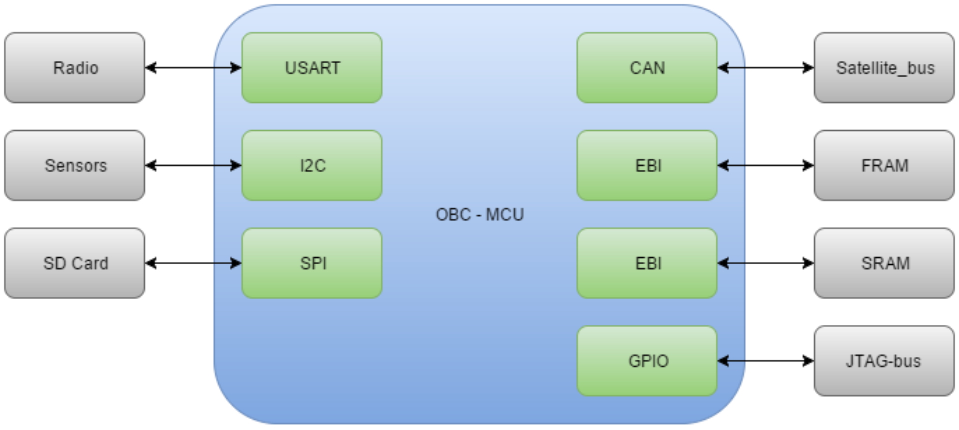


Figure 1.2: The NUTS OBC HW setup

1.5 The Approach Taken in this Thesis

This thesis has been structured very similar to the stages used when developing the software architecture. This has been done in order not to include too many aspects at once, but rather to present one stage at a time, starting with the motivation and environment around the problem. The paper has been sectioned into eight chapters as can be seen below:

- Introduction
- Background
- Software Requirements
- Software Architecture
- Detailed Design
- Implementation
- Testing and Verification
- Reflection

The first two chapters aim to give the reader an understanding of the situation surrounding this project as well as to define the terminology that will be used throughout the report. The thesis problem is then approached by analysing the requirements imposed on the OBC software and formulating a Software Requirement Specification. This SRS is then used as input for the next stage; Software Architecture. In this chapter, possibilities for satisfying the requirement specification are explored and a software architecture aiming at this, is presented. The Software design is discussed in more detail in the next chapter entitled ”Detailed Design”. This is done both to better convey how the system is envisioned as well as to facilitate the future implementation of parts of the system in isolation, without endangering the integrability of the system. After the detailed description of the design, the more practical issues and configurations for the implementation of the software architecture baseline are discussed and presented. Various functions of the implementation is then tested and the results discussed in the ”Testing and Verification” chapter. The motivation for this is two-folded, as the tests are used both to gain confidence in the current implementation as well as to uncover as many defects as early as possible. The thesis ends with a general discussion of the work that has been done, the lessons learnt, a conclusion and what the future might hold for this project.

Chapter 2

Background

This chapter aims at presenting some of the terms and definitions that will be used when discussing reliability and space radiation effects on electronics throughout this report. There is also included a short summery of the most notable contributions to the OBC, to give the reader an understanding of what work already has been done concerning the OBC. Much of this chapter was originally published in Normanns ”Hardware Review of an Onboard Controller”, however some sections have been edited to better fit the scope of Onboard computer software.

2.1 Reliability Theory

This section aims at presenting some of the terms and definitions relating to reliability theory.

Reliability is by Randell et al. defined to be a measure of the success with which a system conforms to some authoritative specification of its behavior [7]. For a satellite launched into orbit, reliability is of particular importance as repair after launch will not be feasible, or at least not economically within the budgets for the NUTS satellite. Before proceeding to how reliability may be achieved, it is important to define some further terms as to avoid any misconceptions or misunderstandings.

2.1.1 Reliability, Failures and Errors

In reliability theory one usually differentiates between faults, errors and failures. In this paper these terms will be used as they are defined in Reliable Computer Systems [1, p. 22]

- Fault is an incorrect state of hardware or software resulting from failures of components, physical interference from the environment, operator error, or incorrect design.
- Error is the manifestation of a fault within a program or data structure; errors can occur some distance from the fault sites.

- Failure occurs when the delivered service deviates from the specified service; failures are caused by errors.

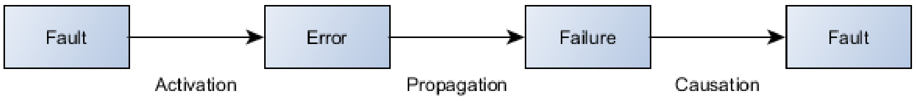


Figure 2.1: The fault, error, failure, fault chain

The relationship between fault, error and failure can be seen in Figure 2.1. In addition to these terms it is common to distinguish three types of faults; permanent, intermittent or transient [8]. These terms are defined as below:

- Permanent describes a failure or fault that is continuous and stable; in hardware, permanent failures reflect an irreversible physical change. An example is a bitflip in the program memory, a broken wire or a software design.
- Intermittent describes a fault that is only occasionally present due to unstable hardware or varying hardware or software states (e.g. as a function of load or activity)
- Transient describes a fault resulting from temporary environmental conditions. It only remains in the system for a limited period of time before disappearing. They can be dormant during all their lifetime (which means that they do not generate an error), or can activate at some point (inducing an error). An example of such faults is a bit-flip in a RAM memory.

The different categories of faults and their origins can be seen in Figure 2.2.

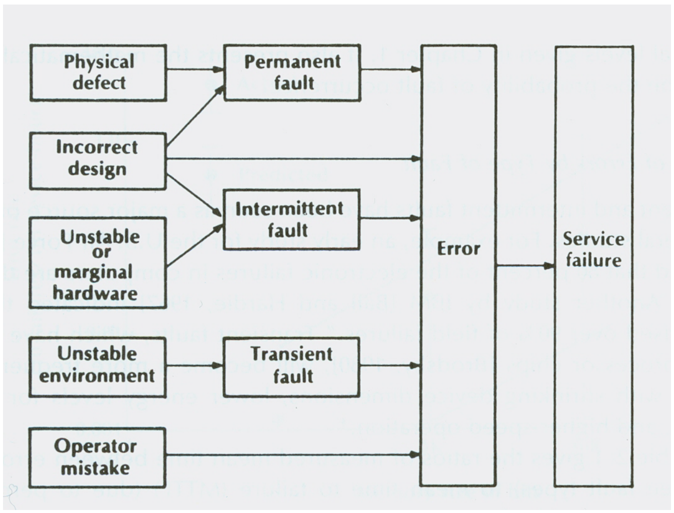


Figure 2.2: Sources of errors and service failures as presented in Reliable Computer Systems [1].

2.2 Space Environment

This section aims to give the reader a short introduction to some of the important concepts and definitions concerning space radiation and the effects space radiation can have on electronics.

2.2.1 Space Radiation

One of the aspects that makes the design of space technologies so challenging is the extreme constraints imposed by the harsh outer space environment [9]. A satellite has to be carefully designed to contend with void, extreme temperature variations, intense accelerations and space radiation. The NUTS satellite is to enter a Low Earth Orbits (LEO), and for most LEOs, the radiation environment is harsher compared to Earth's surface, but not as harsh as the higher orbits or deep space [10].

Radiation in space is produced by particles emitted from either the sun (solar radiation) or from outside of the solar system, Galactic Cosmic Rays (GCRs). Radiation effects from these solar and galactic emitted particles can not only cause degradation, but can also cause failure of the electronic and electrical systems in space vehicles or satellites.

Figure 2.3: NUTS satellite as imagined in LEO

One can by investigating the radiation toughness of components get an understanding of the failure rate one might expect. When categorizing components base on radiation tolerance, three categories are usually used; commercial, rad tolerant, and rad hard. The characteristics parented in the list below are all gathered from NASA [11].

Commercial:

- Process and Design limit the radiation hardness
- No lot radiation controls
- Hardness levels:
  - Total Dose: 2 to 10 krad (typical)
  - SEU Threshold LET: 5 MeV/mg/cm2
  - SEU Error Rate: 10^{-5} errors/bit-day (typical)

- Customer performs rad testing, and assumes all risk
- Customer evaluation and risk

NUTS, as most CubeSats mostly apply commercial of-the-shelf (COTS) components and thus faces some serious risks of various radiation induced errors. A short introduction to the most common effects will now be presented.

2.2.2 Total Ionizing Dose

Total ionizing dose (TID) is the accumulation of ionizing dose deposition over time. This occurs mainly as an effect of protons and electrons, and the ionization creates charges or electron-hole pairs in oxides. This could lead to circuit parameter changes and over time make the circuit ceases to function [12].

According to NASA the expected radiation levels in Low Earth Orbit for higher inclinations (20-85 degrees) is to be about $1 - 10krad(Si)/year$ [11], and typical total dose failure levels of microprocessors at 15-70 krad(Si) [10]

2.2.3 Single Event Effects

Electronic components are vulnerable to a number of effects when exposed to cosmic rays. The collective term for the different failure mode occurrences is Single Event Phenomena (SEP) or Single Event Effects (SEE). A brief overview of the most common SEEs can be seen below. The SEU threshold LET is described as the energy level per amount of

Name	**Effect**
Single Event Transient (SET)	Soft intermittent fault Propagating through circuit.
Single Event Upset (SEU)	Soft transient fault State change on latch or memory.
Single Event Latchup (SEL)	Apparent short circuit Can be mitigated with power cycling Can cause destructive thermal runaway.
Single Event Gate Rupture (SEGR)	Permanent failure.
Single Event Burnout, SEB	Permanent failure.

Table 2.1: Single Event Effects

material of the radiation that will trigger SEU events. The energy of most comic rays range between $100 - 10000MeV \cdot cm^2/mg$ [13], and as seen earlier the expected SEU error rate for COTS in LEO is by NASA estimated to be around 10^{-5} error/bit per day. This s of course a rough number as it doesn't even mention for what type of memory it is applicable. It does however give some understanding to in what magnitude errors can be expected. If we assume this error rate is correct, a 128 kB of RAM would for instance experience around 10 errors accumulated per day in orbit [10].

2.3 Earlier Work

This section will shortly present some of the theses that have contributed to the OBC in the the past.

Internal Data Bus of a Small Student Satellite - Marius Lind Volstad
Volstad's master thesis is the first and maybe the most substantiating contribution to the OBC. In this thesis Volstad designs the inital hardware both for the backplane as well as the OBC. Both the OBC and the backplane hardware are produced and tested to be mostly functional. Some test drivers for the OBC hardware was also implemented to verify the setup. Unfortunately as the thesis covers so much, many of the reasons and discussions for choosing the various solutions have not been included.

Memory management and error handling in FreeRTOS for a CubeSat project - Diaa Jadaan
Jadaan explains the different memory management schemes in FreeRTOS, as well as how stack overflows can be detected. The paper also investigates how an exception handling framework, Exceptions4c, can be used for exception handling in POSIX-based systems. Jadaan states that in order for this to be useful in FreeRTOS porting is needed, but that this is a complicated task that he was not able to finish within the timeframe given. The paper also states that RAID 4 techniques can be used for error detection for embedded systems, and a small demonstration is written in C++.

Implementing CSP over I2C for the new repository on the NTNU Test Satellite - Erlend Riis Jahren
Jahren imports the Cubesat space protocol library into the NUTS project and implements the NUTS Reliable Protocol on top of it. He also tests the various functionalities rather extensively and discusses the results. Jahren states in the report that ”almost all of the memory is already used in the FreeRTOS, CSP and NRP implementations, leaving only the leftover memory available to be used in testing” [14]. He attempted to solve this by trying to move the heap into an external SRAM, but it was never completed successfully. Despite this he was however able to prove that most of the NRP functionalities were succesfully implemented, where as the only exception was that of multiple concurrent streams, as a server was not able to receive concurrent streams from two clients.

Mission Event Planning & Error-Recovery for CubeSat Applications - Magnus Haglund Arnesen and Christian Elias Kiær
Arnesen & Kiær presents the most important system level mission event plans. Their paper investigates the battery charging and discharging and presents a power budget, and a design for an external watchdog is presented and tested.

Improvement in the Reliability of a Bi-Processing Unit Satellite Subject to Radiation-Induced Bit-Flips - Mayeul Marcadella
In his thesis Marcadella presents and implements the Resilient System Prototype (RSP). The RSP is a software project using the same microcontrollers as the OBC and the sister-OBC. It implements cold dynamic redundancy between the two modules, where one can

coop and/or reprogram the other in the event of a failure. The project implements a simple JTAG controller, a program memory corruption detection and correction facility, and a bit-flip injector for testing purposes. The setup was tested for 8 hours under a simulated bit-flip density 250 times higher than the expected on-orbit rate [9]. The downsides are that the project has not been tested on the actual OBC, the project uses a USART that is not implemented in the OBC design and there are also quite some work left as the project only emulates the use of external memories and doesn't actually interface any external memories itself.

Error Detection and Correction for Low-Cost Nano Satellites - Kjell Arne Ødegaard
Ødegaard evaluates low-cost measures for dependability and robust Error Detection and Correction for use in applications such as nano satellites. Different methods are evaluated, with the main result being the mitigation failures due to bit-flips in system memory by using BCH codes [15]. An implementation is then made, tested and the result is discussed.

As can be seen form the theses presented above, there has been written multiple papers on how reliability can be assured for the OBC through the use of various software techniques. This paper therefor focuses on the implementation design of the actual OBC, using the already documented research of the former theses as a starting point. Before an architecture can be decided upon, one has to create a requirement specification to ensure that one solves the correct problem. This will be presented in the next chapter.

Chapter 3

Software Requirements

In this chapter the requirements imposed on the OBC software will be analysed and discussed. The existing functional and non-functional requirement specifications will be updated and a specific software requirements specification in accordance to Institute of Electrical and Electronics Engineers (IEEE) Standard 830 Recommended Practice for Software Requirements Specification (SRS) will be created and presented.

3.1 Requirement Specifications and Documentation

By far the most common project risks in system development are poor requirements and poor project planning. Researchers at Hewlett-Packard, IBM, Hughes Aircraft, TRW, and other organizations have found that purging an error by the beginning of construction allows rework to be done 10 to 100 times less expensively than when it's done in the last part of the process, during system test or after release [16]. Requirements are therefor considered an important part of the development projects and well worth spending some time on refining.
A general requirement specification for the OBC was presented by Normann in [17]. This specification outlines the top-level requirements imposed on the OBC system and will be used as a starting point in which more detailed requirements may be added as they are explored.

Before diving into the requirement refining process, it is important that one considers the downstream consequences of one's action. Engineers and scientists often don't realize the downstream complicity (and cost) entailed by their local decisions [18]. Overly stringent requirements and simplistic hardware interfaces can complicate software. In fact NASA made a comprehensive inquiry into flight software complexity and published the result in the report "NASA Study on Flight Software Complexity" in 2006. In this report one of the major key lessons learnt was that unsubstantiated requirements have caused unnecessary complexity in software, either because the requirement was unnecessary or overly stringent [18]. It is therefor considered important not to propose requirements simply for the sake of deciding matters, but rather so that actual, real constraints imposed on the system can be identified and enunciated in an early stage of the development process.

Wertz and Larson states that analysis and designs are iterative, gradually refining both the requirements and methods of achieving them. Broad objectives and constraints are the key to this process. Procurement plans for space systems too often substitute detailed numerical requirements for broad mission objectives. While our overall objectives to communicate, navigate, or observe will generally remain the same, we may achieve these objectives differently as technology and our understanding of the process and problem evolve. For this reason it is important to avoid overly stringent or unnecessary requirement specifications [19].

3.2 Detailing the OBC Requirements

As this thesis focuses on the OBC software, the top-level OBC requirements will be used to generate a software requirements specification. Standardization is considered important, as it can serve as a reminder and helps the developer to consider the most important aspects that need to be addressed. However as very few developers with the NUTS project have any experience with established standards, it is not considered important that the standard is followed strictly, but rather that it can be adopted and modified for the NUTS project. For the NUTS project no formal standard was chosen for software or design documentation. The author of this thesis therefore proposes the use of the well established IEEE Standard 830 Recommended Practice for Software Requirements Specification and the IEEE 1016 Software Design Document for documentation of software requirements and design, respectively. The IEEE standards were chosen as they have been popular in the NTNU computer science course TDT4240 - Software Architecture, and thus most students having taken this course will already be familiar with these standards. It is however important to note that these standards should not be followed blindly. They are meant to be a tool to help the developer write good requirement and design documentation. The designer should use the standards as a reminder of what should be included and how it can be structured in the documentation. The developers are free to make deviations from the standard if they feel that this will better convey the requirements or design of the software.

3.2.1 Detailing the Functional Requirements

The functional requirements have not changed much since the original specification was published. There is however one addition to the list of requirements; in-orbit reprogramming. The reprogramming of satellite modules in-orbit is now considered a must after discussions within the NUTS project. This is much due to the lessons learnt, presented from other successful CubeSat projects such as the ESTCube-1 [20], GOMX-3 [21] and PolySat [22]. In the report ”ESTCube-1 In-Orbit Experience and Lessons Learnt” it is stated that a large portion of ESTCube-1 software was written after launch: ”We consider this bad practice because it relies on in-orbit software updates and the mission is deployed, increasing a risk of satellite failure before performing all the planned experiments. However, we think that functionality of in-orbit software updates of all active subsystems is critical for a CubeSat mission, especially for teams without prior experience. That functionality can, most importantly, save the mission and it also allows using the satellite for other purposes than initially planned." [20].] This functionality will have to be implemented in the OBC and thus the OBC must be able to reprogram itself as well as other modules. The additional requirements induced by this can be seen in Table A.6.

R00-OBC-PRG-000	PRG= PROGRAMMABILITY. The OBC must be able to reprogram itself as well as other modules.
R00-OBC-PRG-002	The OBC be able to receive and store at least one additional boot image from ground station

Table 3.1: OBC reprogramming requirements

As changes now have been made to the initial document, a document revision history section is added to the document to avoid any mix-ups between different versions.

Revision	Date	Author	Description
1.1	01.03.2016	Magne Normann	Included requirements of reprogramming, refined non-functional requirements.
1.0	05.10.2015	Magne Normann	Initial Document Release

Table 3.2: OBC reprogramming requirements

3.2.2 Detailing the Non-Functional Requirements

Non-functional requirements specify system properties [23]. These system properties can be imperative for mission success and must be analysed before any design can be made. The top level non-functional requirements for the OBC were presented along with the functional ones in the "Hardware Review for an Onboard Controller for a Cubesat" report by Normann [17]. The following section will now further elaborate on the non-functional requirements considered to be the most important for the NUTS mission. These include reliability and maintainability, as well as memory and power requirements that might affect the OBC software.

Maintainability Requirements
Managing complexity is the most important technical topic in software development, and once you understand that all other technical goals in software are secondary to managing complexity, many design considerations become straightforward [16].

The NUTS project is student driven, where most contributions come from project works and master theses. This means that there is a large turnover in the project and that personnel seldom stay for more than one or two semesters. This constitutes to a constant brain-drain and poses a serious challenge on the project management, as well as for the accepted amount of complexity in the project. It is imperative that an engineering student can understand, and contribute to the project within the timeline of one semester. This poses strict demands to the maintainability of the system, and the allowed complexity. It is therefor of utmost importance that the OBC software architecture is easy to use, maintain and further develop by new students joining the program.

Ease of further development is considered part of the maintainability of the project and with this comes requirements to version control, and ease of testing. The NUTS OBC software architecture therefore requires that tools for version control and debugging should be an inherent part of the software project, together with well documented code.

Reliability Requirements

R00-OBC-MAI-001	MAI= MAINTAINABILITY. All user written code must be documented in a coherent manner
R00-OBC-MAI-002	Version control shall be used for the code repository
R00-OBC-MAI-003	Software architecture must support tools for debugging

Table 3.3: OBC Maintainability requirements

The reliability requirements are especially important for deployed systems such as CubeSats, as manual maintenance after launch is impossible. The system must be able to recover from any failures that are likely to happen in the life span of the satellite. Care must be taken to find the simplest and overall most effective way of handling the errors that might occur. It is important to keep in mind that the enunciation of any strict requirements concerning the reliability of the system may have tremendous consequences for the complexity of the resulting system [18]. The main reliability requirement is R00-OBC-NON-REL-000 ”The OBC mean time to failure should be greater than the duration of the space mission.” This is a general, not too stringent and verifiable requirement. It is thus deemed adequate for this iteration. In requirement R00-OBC-NON-REL-002 however, it is stated that ”The OBC must be able to recover from failures”. This requirement is both ambiguous and unspecified and thus a poor requirement. The purpose of the requirement is to ensure that the system does not go down despite errors occurring. The most likely error sources are the coding bugs as well as radiation effects such as SEU and SEL. The coding bugs are not so easily handled by a faulty software, but the radiation induced effects are more often than not soft errors and can be handled by a simple reset of the OBC [10]. The requirement is therefor changed to: ”R00-OBC-NON-REL-002 The OBC must be able to recover from transient errors such as SEL and SEU, no matter where or when they might occur.” This may sound stringent, but as all soft errors can be resolved by a soft reset of the module, it is considered an important and cost-effective requirement. The aim for the system developer should not be to strive to find different solutions for all possible failure scenarios, but rather to implement the simplest, most general catch-all measures.

Power Requirements

Arnesen & Kiær estimated the normalized charging power to be 3.205 W, while the combined satellite power consumption was estimated to reach as high as 8.791 W [10]. A maximum power consumption for the OBC module (MCU plus memories) has been estimated to 550 mW [17], however the average power consumption must be much lower if

the batteries are not to be drained out. It is therefor imperative that power optimization techniques are integrated into system design and architecture. The satellite wide power budget for the NUTS satellite reserves 300mW for the average power consumption of the OBC and its memories.
The amount of power consumed by the microcontroller effects the heat dissipation. As the system will be deployed in vacuum, there will not be any air circulation to help conduct the heat away from the micrcontroller and heat issues must be addressed as well. As an example, Estcube-1 using an ARM Cortex-M3 based microcontroller reported that that microcontroller supported overclocking of up to 128 MHz while remaining stable at room temperature. However, instabilities of the overclocked microcontroller started to appear when the operational environment temperature exceeded 50 degrees Celsius. At nominal 72 MHz frequency the device was able to operate within the ranges set in the specifications: from –20 up to 80 degrees Celsius. The SAMV71 used for one of the OBC microcontrollers can be clocked as high as 300 MHz and has it's operation temperature range of -40 to 105 degrees Celsius [24]. The UC3C only support clock frequencies up to 66 MHz and has an operational temperature range between -40 and 85 degrees Celsius [25]. The temperatures of the microcontrollers should therefore be tested in vacuum at with different frequencies before a final processor speed is decided upon.

Memory Requirements
Empirically, initial software size and throughput estimates double from System Requirements Review to launch because early requirements are uncertain, and changes in software are easier to make than changes in hardware during late stages of spacecraft development [26]. These statistics are for regular spacecraft development, and may not be applicable for cubesat development. However memory constraints should not be taken lightly and to be on the safe side, this author recommends that the initial design aim to reserve 30% of the available on-chip non-volatile memory (ROM) for unforeseen expenses, in accordance with reccomandation from Hansen et al [26]. The SAMV71 embeds 2MB flash while the UC3C MCU embeds 512 KB Flash. This means that the OBC software should aim at using less than $512KB * 0.7 = 358.4$ KB for its applications.

One should not attempt to use all of the available volatile memory (RAM) or throughput either. Asynchronous processing, such as interrupt handlers, introduces a level of uncertainty in throughput. Costs also rise dramatically as we shoe-horn the software into existing memory [Boehm, 1981]. As a rule of thumb Hansen et al. recommends that spacecraft computer systems should use 70% or less of available throughput [26]. For the proposed OBC microcontrollers SAMV71 and UC32C the available onboard RAM is 384 KB and 64 KB, respectively. Due to the small size of the internal RAM of the UC3C, external RAM is incorporated into the current design of the OBC, yielding an additional 2 MB of memory space. The maximum allowed memory usage when reserving 30% then becomes $2064KB * 0.7 = 1444.8KB$. This may seem like an excessive amount, but one must remember that the OBC will have to work with both boot images as well as captured pictures from the camera. It is therefor important that this boundary is established and respected.

Timing Requirements
The OBC must handle incoming packages in a satisfactory fashion. Except for this, there are no hard timing demands for the OBC. The radio link uses a baudrate of 9600 bits per second. For the OBC connected to the UHF radio, it is important that the buffer on the radio microcontroller does not overflow with packages sent from the ground station. There are many factors, such as packet size and baudrate for the radio communication that have not yet been decided and thus an accurate estimate of the expected workload cannot be obtained. However using preliminary values to get an estimate can give an indication on the requirements posed on the OBC. The microcontroller in charge of controlling the ADF7021 UHF radio is able to queue up to 5 frames. If 266 bit frames are used at a baudrate of 9600 bits/second, this gives a timeslot of $(266 * 5)/9600 = 0.139s$. The OBC must therefor be able to receive at least 5 frames of 266 data bits from the UHF radio microcontroller every 139 ms.

When it comes to the internal data bus, baudrates up to 1Mb/s are supported. The specific data rate to be used for the internal bus has not yet been decided, but is likely to be slower than the maximum speed. It is however recommended that the CAN receiving is interrupt based with a high priority so that lower priority tasks can be preempted to be able to accommodate the data stream supplied by the internal satellite data bus. The CPU clock frequency must be considerable faster than the bus baudrate to properly receive the data. As was described above, Hansen et al. recommends that no more than 70 % of available throughput shall be used, this serves as a minimum requirement for the speed of the OBC. The maximum clock frequency for the UC3C microcontroller is 66 MHz, while for the SAMV71 the maximum frequency is 300 MHz for the CPU clock and 150 MHz for the internal bus matrix.

3.3 Defining Operational Modes

To define requirements for a computer system, it is convenient to develop a computer system state diagram [26]. Arnesen and Kiær proposed three different modes of operations, depending on the remaining battery capacity:

- Critical mode - less than 25% battery capacity
- Avoidance mode - between 25-50% battery capacity
- Normal mode - between 50-100 % battery capacity

By incorporating these power modes with the typical state transition diagram proposed by Hansen et al. [26], together with the cold redundancy scheme proposed by Marcadella in ”Improvement in the Reliability of a Bi-Processing Unit Satellite Subject to Radiation-Induced Bit-Flips” [9], one receives a state transition diagram for the Onboard Computer as depicted in Figure 3.1 and Figure 3.2.

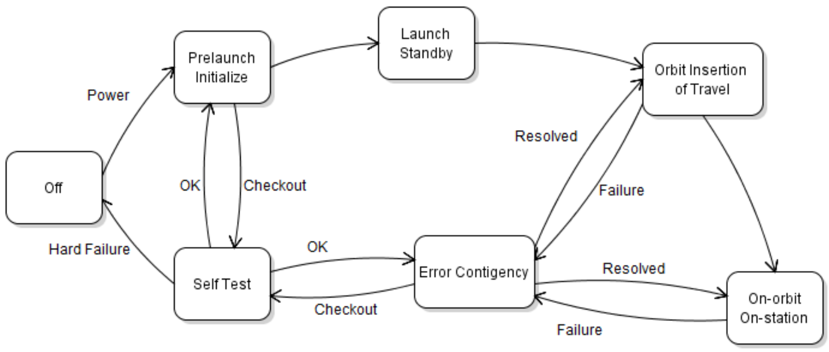


Figure 3.1: The OBC State Transition Diagram

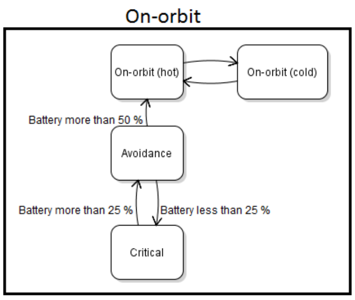


Figure 3.2: The OBC on-orbit internal state diagram

3.4 Summary

In this chapter the functional requirement specification for the OBC was taken as input, and a Software Requirement Specification in accordance with the IEEE standard 830 was outputted. Functional requirements concerning in-orbit reprogramming was added, and non-functional requirements for the OBC software were analysed and discussed in more detail. Due to the challenges related to the high turn-over of inexperienced personnel in the NUTS project, maintainability was considered one of the foremost important non-functional requirements of the software architecture to be created.

Chapter 4

Software Architecture

In this chapter, the various possibilities for satisfying the requirement specification presented in the previous chapter are explored and a software architecture is presented. The output of this section will be a design specification, create din accordance with the IEEE Standard 1016 Software Design Document.

Maintainability was determined to be one of the foremost important non-functional qualities, therefore this chapter will start by presenting the characteristics of good software design, to show what the design should strive for. Different approaches is then shortly discussed before one is chosen for this project. After defining an approach, the first stage of the design process is started as the OBC software is decomposed into software modules.

4.1 Attributes of Good Software

The goal for the OBC software is to satisfy the OBC software requirements specification. The possibilities of this are many, and it is therefore important to identify what separates a good design from a bad one. McConnel, the author of Code Complete, states that there are several general characteristics to high quality design. These can be seen in the list below:

- Minimal complexity
- Ease of maintenance
- Loose coupling/high cohesion
- Extensibility (Extensibility means that you can enhance a system without causing violence to the underlying structure. You can change a piece of a system without affecting other pieces. The most likely changes cause the system the least trauma.)
- Reusability (Reusability means designing the system so that you can reuse pieces of it in other systems.)
- Portability
- Leanness (Leanness means designing the system so that it has no extra parts (Wirth 1995, McConnell 1997).)

- Stratification (Stratification means trying to keep the levels of decomposition stratified so that you can view the system at any single level and get a consistent view. Design the system so that you can view it at one level without dipping into other levels)
- Standard techniques

Minimal complexity and ease of maintenance is considered to be the foremost important characteristics of a good design. This holds true for the NUTS project as well. Furthermore, emphasise is put on loose coupling between software modules, while having strong cohesion inside them, effectively minimizing dependencies between modules [16] [27]. This is important as it minimizes work during integration, testing and maintenance. When it comes to extensibility, reusability and portability these are not of major importance to the NUTS project as the mission is specific and it's considered unlikely that software will be reused for other missions, or that there will be any major changes in the mission goals. However Wertz & Larson states that even though the overall objectives to communicate, navigate, or observe will generally remain the same, we will achieve these objectives differently as technology and our understanding of the process and problem evolve [19]. A good design should therefor anticipate some change, though only changes that are likely to happen. The cost of preparing for these changes should also only be implemented if they do not add a considerable amount of complexity. For the NUTS OBC, the most likely changes are considered to be to hardware changes, such as microcontrollers and memories as the technology advancements are quite rapid in these areas. In addition it is expected that the system will have to support the adding of, and changes to, features and services. Much in the same manner as in-orbit reprogramming was added to the requirements in this iteration. McConnel also recommends the use of standard techniques. This is deemed especially important for the NUTS project due to the high turnover of personnel. The more a system relies on exotic pieces, the more intimidating it will be for someone trying to understand it for the first time. Trying to give the whole system a familiar feeling by using standardized, common approaches is therefor considered an important tool for managing the complexity and easing the maintainability of the system.

4.1.1 Platform Selection

There are many different ways of doing embedded system design. Sommerville states that platform selection is an activity that is often included in embedded design processes. In this activity, an execution platform (i.e. the hardware and the real-time operating system to be used) is chosen for the system. As the hardware already has been decided for the OBC, the consideration of programming language and operating system seems the natural next step, as these choices will affect what software mechanisms, and thus what designs are possible. One could of course argue that in an ideal world, a problem is first solved, and only then can the language and operating system optimal for the solution be selected. However, there may be constraints related to the choosing of language and operating system, invalidating all designs that does not incorporate these constraints. Therefor, a more practical approach is chosen, where the constraints and demands for the language and operating system is explored first, so that a design using the available tools can be made.

Real-time and embedded programs (such as CubeSat applications) have to control and interface with real-world entities (thrusters, switches, sensors, etc.) that are inherently parallel. Reflecting the parallel nature of the system in the structures of the program makes for a more readable, maintainable and reliable application [8]. This is one of the main motivations for writing concurrent programs. For this reason and to minimize dependencies between different services, concurrency was deemed appropriate for the NUTS OBC.

Some hard real-time systems are still sometimes programmed in assembly language so that tight deadlines can be met. However, systems-level languages, such as C, which allow efficient code to be generated are also widely used. The advantage of using a systems programming language like C is that it allows the development of very efficient programs, that allows for direct hardware access [28]. As there are no real-time constraints that demands for assembly implementation, the C language is considered the most appropriate for the NUTS OBC software. Unfortunately, the C language does not include constructs to support concurrency or the management of shared resources. Concurrency and resource management must therefore be implemented through calls to primitives provided by the real-time operating system, such as semaphores for mutual exclusion.

Up until the writing of this thesis, the real-time operating system FreeRTOS has been used for the OBC software, though no argumentation has been given for why this operating system was chosen. A short inquiry into this matter will therefore be made in the follow section.

A comparison between a number of CubeSat satellites was published in the paper ”Fault Tolerant and Flexible CubeSat Software Architecture” by Manyek, published in 2011. Of the twelve CubeSats that were investigated only two use a custom operating system. Most systems rely on two watchdogs to protect against hang-ups that can occur as a result of radiation events. Most are coded in C, though parts may be coded in assembly. The small CubeSats (in consideration to the processing power of the OBC) often rely on specialized solutions and code that is not reusable, while for the CubeSats running Linux OS, VxWorks or Windows, more independent programs with few or no inter-dependencies are being used. Unfortunately this author could not find any comparison that was more up to date, therefore a comparison of a handful of newer CubeSats was made and is presented below in Table 4.1.

CubeSat	Launch Date	OBC MCU	OS	Architecture
AAUSAT3	23.02.2013	AT90CAN128, ARM7	FreeRTOS	Distributed
ESTCube-1	07.05.2013	STM32F7 (32-bit)	FreeRTOS	Centralized
kySAT-2	19.11.2013	C8051 (8-bit)	SPARTOS	Distributed
delfi-n3xt	21.11.2013	msp430 (16-bit)	custom	N/A
GOMX-3	19.08.2015	UC3 and ARM A9 (32-bit)	Freertos & Linux	Distributed
AAUSAT5	19.08.2015	AT90CAN128, ARM7	FreeRTOS	Distributed
skCUBE	July 2016	msp430 (16-bit)	custom	N/A
Eye-Sat	end of 2017	ARM A9	FreeRTOS & Linux	Centralized
EstCube-2	end of 2018	STM32F7 (32-bit)	FreeRTOS	Centralized

Table 4.1: Cubesat Operating System Comparison

As can be seen from the Table 4.1, Linux, FreeRTOS and custom operating systems are still popular for CubeSats. However all of these may not be possible for the NUTS hardware. The choice of OS is often dependent on what can be supported by hardware. The AVR UC3 devices do not embed a Managment Unit (MMU), which is a prerequisite for porting most Linuxes on MCUs. However uClinux, a derivative of Linux 2.0 kernel, is a distribution intended for microcontrollers without Memory Management Units (MMUs) and is ported to some Atmel SAM microcontrollers through the at91 project, but no official ports exist for the samv71 microcontroller [29], nor for the UC3. A master thesis was written at NTNU in 2008 which ported Linux to the UC3A [30], but this has unfortunately not been merged into the mainline kernel. Care should be taken before choosing unofficial distributions as very little can be said about their user community, maintainability and guarantee of function. Linux is therefore considered unfit for the current OBC hardware, due to the lack of any official ports. Using home grown RTOS is not recommendable due to the complexity of developing it, using it, and maintaining it. Getting RTOS right can be difficult, and even if one gets it right, it can be a lot of work [31]. FreeRTOS thus seems like a good choice for the NUTS OBC.

FreeRTOS is available as a library of types and functions to build real-time, multi-tasking, embedded software applications. The Scheduling can be preemptive, cooperative or a hybrid configuration. It is mainly written in C, with some parts in assembly. The memory footprint of FreeRTOS depends on application and hardware, but Real Time Engineers Ltd, the developers of FreeRTOS, claim the following memory footprint for a fully optimized IAR STR71x ARM7 port with minimal configuration and four priorities [32].

Item	Bytes Used
Scheduler	236 bytes
For each task	64 bytes + stack size
For each queues	76 bytes + storage area

Table 4.2: FreeRTOS memory footprint

The ROM space needed for the kernel itself was reported to be between 5 to 10 KBytes. EstCube-1 reported that for their OBC, FreeRTOS takes about 11 KB of Flash and 64 KB of RAM, most of which is reserved for the heap of dynamic memory management [20].

One can see from this that FreeRTOS will have no problem to fit inside neither of the proposed microcontrollers and it should only claim a minor part of the system memory. The FreeRTOS is therefor chosen as the OBC real time operating system, due to it's simplicity as well as its large user community both in the field of CubeSats as well as other embedded systems.

Debugging a real time application can be a complex exercise due to multiple task management and kernel objects [33]. For this reason, as well as requirement R00-OBC-MAI-003: "The OBC software architecture must support tools for debugging", a tool for debugging FreeRTOS is deemed necessary.

There are multiple ways of debugging FreeRTOS through Atmel Studio. Methods readily available though Atmel Studio include FreeRTOS Viewer and Percepio Streaming Recorder. Due to its comprehensiveness Percepio Tracelizer (Snapshot) library is chosen as it in addition to task view allows for analysis of the following [34]:

- CPU Load
- Timing Variation
- Communication Flow
- Synchronized view
- Kernel Object History

One should be aware however that this is an intrusive method, and the timing variations thus may be different when the trace is disabled. For time critical analyses, Percapio Stream Recorder can be used. This is based on the J-Link's Real-Time Transfer feature (RTT) that allows for transferring data between host and target at high speeds in a non-intrusive manner [35]. However, one should be aware that the method is not truly non-intrusive as the internal bus matrix is used for the transfer, and thus the program execution may be effected by having to wait for bus access. Another possibility is to use the Coresight features embedded in the ARMv7 architecture, though these traces are not aware of kernel objects and may be difficult to interpret. If the importance of doing the time critical analysis outweighs the time effort, this is however a method that could be used. The Percepio Trace Snapshot Recorder seems to implement the required functions and is considered sufficient for most development debugging purposes, and thus chosen for implementation for the OBC Software.

For communication between the satellite subsystems a CAN-bus is used. As CAN is a multi-master serial bus, this means that hardware supports a distributed massage based architecture, where any node may initiate interactions. As a communication protocol for the communications between satellite subsystems (and the ground station) NUTS uses a small protocol stack called the CubeSat Space Protocol (CSP). The small protocol stack was formerly known as CAN Space Protocol and has proven flight heritage with satellites such as AUSAAT3, AAUSAT4 and GOMX-3 [36]. It is written in C and its design follows the TCP/IP model and includes a transport protocol, a routing protocol and several

MAC-layer interfaces [37].

4.1.2 Architecture Selection

Somerville states that the most common architectures for embedded systems is the master-slave architecture, while for distributed systems it's a more service-oriented server-client architecture [28]. Somerville states that master-slave architecture is used in real-time systems in which guaranteed interaction response times are required. For the NUTS satellite there are no such hard real-time demands, and thus a distributed architecture may be a viable option.

As mentioned earlier, NUTS already uses CSP for communication between satellite modules. Though originally intended for cross-module communication, the protocol does support loopback mode. Meaning that messages could be sent from one thread to another running on the same microcontroller via CSP. This facilitates the possibility of a software component-based architecture without having to implement or add any new middleware libraries.

Component-Based and Service-Oriented Software

As with most systems in the NUTS satellite, the OBC is being realized through the work of multiple student developers, working with the NUTS initiative at different times throughout the satellite project timeline. It is therefor of great importance that software modules are as independent, or loosely coupled, as possible in order to secure good extensibility and maintainability. Independent components that are completely specified by their interfaces, would greatly simplify the development and maintenance of the software as each module could be designed, implemented, tested and if necessary replaced without effecting the rest of the system. This idea, of building a "legobox" of interoperable, reusable services has been a popular goal in software engineering. Object-oriented software engineering, Component-Based Software Engineering (CBSE) and now Service-oriented Architecture (SOA) engineering all strive towards this goal in slightly different ways [38]. CBSE seems to be a popular way of constructing software for small satellites and has been reported as the main software architecture of projects such as NASAs GSFC Open Source Software Core Flight Executive (cFE) [39], ESAs Space Avionics Open Interface aRchitecture (SAVOIR) project [40], as well as by private actors such as in Bright Ascension's GenrationOne Onboard Software product, which was used on UKube-1 [41].

The SOA philosophy is very close to that of CBSE. The SOA term is mostly used for web services, and has multiple definitions, the one given by Somerville states that a service is defined as: ”A loosely-coupled, resusable software component that encapsulates discrete functionality, which may be distributed and programmatically accessed." [28]. Petritsch states that in an SOA, resources are made available to other participants in the network as independent services that are accessed in a standardized way. Most definitions of SOA identify the use of web services (using SOAP, WSDL and UDDI) in its implementation; however it is possible to implement SOA using any service-based technology [38].

Both component-based and service-oriented software can greatly ease the design, implementation and maintenance of software modules. It alleviates the intricacies of inter module communication by abstracting the responsibility of gluing modules together onto the middleware. Implementing this handling in the middleware may however prove to be a challenging and comprehensive task. Therefor existing solutions should be explored. The advantages of component-based or service oriented software is desirable, but not deemed cost-effective if NUTS have to develop the middleware handling of communication from scratch.

Both NASAs cFE and ESAs SAVOIR could be fun and maybe advantageous to use for the NUTS satellite, but after further investigation it was discovered that neither of these systems had official support for the target operating system (FreeRTOS)[42]. Also both of these projects are quite large and may add a level of complexity that simply isn't required for the NUTS mission. The architecture of layered component-based or service-oriented design does however seem to exhibit many of the characteristics needed for the OBC software. The next sections have therefor been reserved for the investigating of pros and cons of using the already present CSP library to facilitate a service oriented, or component based architecture for the NUTS satellite.

The Perks of CSP for Inter-Thread Communication

The main arguments for using CSP as a unified interface for both internal and external services are that it may minimize system complexity and ease error checking and handling. CSP uses a service-oriented topology with a Berkley/POSIX socket-like Application Programmer Interface (API). Most computer developers have at least some experience with socket programming and are thus already familiar with the basics of how the CSP API is used. Using this service-oriented architecture also encourages developers to minimize dependencies to other sub-modules and uphold a standard way of communication throughout the satellite.

If CSP is used not only for inter-module communication but also for intra-module communication, every internal service becomes (if wanted) accessible to other modules as well. This means that if a service provider permanently fails, the service can be requested from any other service provider with similar request-handling, simply by changing the receiver address. An example of this could be if the logging service of a module goes down due to some permanent hardware failure. The module would then be able to use the logging service implemented in another module, by changing the receiver address it sends its log messages to.

If a housekeeping entity is being used to manage the health of the satellite, then it can be used to update what sockets should be used in case of any permanent submodule failure, abstracting any satellite-system error handling away from the submodules. This means that a submodule developer does not need to check for or implement handling in case of any failures in other submodules. If other modules go down, the housekeeper will change the addresses appropriately and the submodule application developer is only responsible of implemented error handling for its own services.

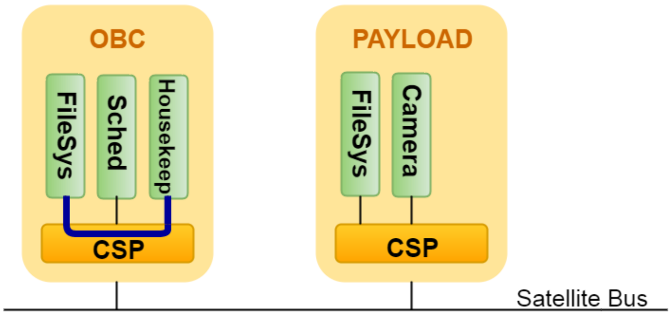


Figure 4.1: Service supplied from on-chip provider

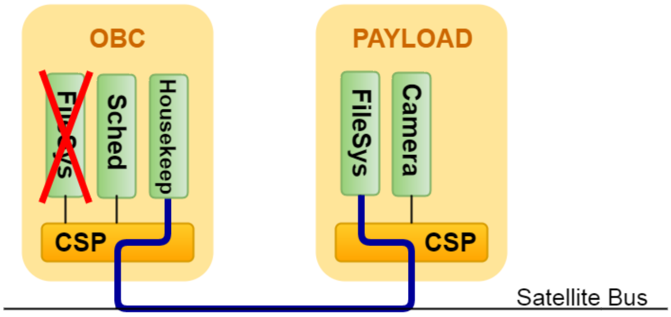


Figure 4.2: Service rerouted to off-chip provider

Concerning congestion control and error checking, CSP supports a handful of Internet Control Message Protocol (ICMP) calls, as well as services such as requesting memory and buffer status. It also has support for getting and setting time, retrieving the up-time of another module, and retrieving a list of the running threads as well as their statuses. In addition to this, the CSP router task supports Quality of Service (QoS) and can evaluate its own performance. If CSP is used for inter-thread communication, all these services become available for submodules as well and may ease the design and implementation of certain housekeeping functions as the responsiveness and healthiness of any submodule connected to the network can be investigated through standard CSP-calls.

The Trade-Offs of CSP for Inter-Thread Communication

The main trade-offs for using CSP for inter-thread communication is considered to be the added overhead resulting in slower execution and more memory needed, and the fact that the CSP router-task poses a single point of failure for the communication both internally as well as externally. CSP may also be considered to be an ”exotic piece” an thus can be intimidating for new developers. Especially so if they are not familiar with socket programming.

Using CSP for both internal and external communication causes all packages to be handled by the router task. This makes the router task a single point of failure for the entire module. If the router task suffers a failure, no communication, not even internally to the module is possible. If one considers the alternative of using native FreeRTOS synchronization techniques for inter-task communication. The module may continue to function with some degraded performance. However it would be much harder to investigate and verify the healthiness of the internal communication. It can also be argued that as there may only be limited value in the internal healthiness of a module that cannot communicate with the rest of the system, there is only limited value in separating the two communication schemes as both must work for the satellite to be functional. It may thus be worth the payoff to join these two single points of failures into one, as this greatly simplifies error detection and handling.

CSP uses a zero-copy buffer and queue system, meaning that data isn't actually copied and moved around, but rather only the address reference is handed from one handler to another. This results in efficient use of memory, but as with most message passing systems, deciding the buffer size of each queue does pose a challenge. All buffer sizes must be declared to their maximum capacity. As the maximum capacity is much larger than what is normally needed, a large portion of the memory will be occupied but very seldom used. If CSP weren't used, but instead native FreeRTOS queues directly, each queue would have to be declared to its maximum capacity. Substituting this with one router task that handles all communication may then be advantageous as it is unlikely that maximum capacity is needed for all communication interfaces at the same time. Thus a smaller memory buffer may be allocated for the router task than if multiple buffers were created separately.

To estimate the additional overhead that comes from using CSP for thread communication a simple test system was set up and analysed. The tests as well as the results can be seen under Chapter 7. After preliminary testing proved the feasibility of using CSP for both inter- and intra-module communication, the next step was to decompose the system into modules.

4.2 Decomposing the System into Modules

This step is what is often called decomposing or modularizing in traditional software literature [16], and is considered one of the primary weapons against software complexity [27].

When decomposing the system one must base every design and decision task on requirements, and trade studies at any level must take into account all related requirements, while considering the impact of changes throughout the project and system architecture [43]. There are many different methodologies on how decomposing should be done. By using the popular top-down approach where one starts with the general problem and then breaks it down to manageable pieces, one may obtain an adequate design. The opposite approach is to compose a system by starting with manageable parts and building the system bottom-up. These methods both have their strengths and weaknesses and aren't necessary competing strategies, but can be mutually beneficial [16]. The OBC software must necessarily run on the OBC hardware and thus cannot be completely decoupled from the underlying hardware. The software design for the NUTS OBC is therefor approached starting at both top and bottom, iterating towards an compliance.

To ensure that the decomposition is true to the OBC software requirements one may try to create modules mimicking the requirement partitioning. This would constitute to the following modules:

- Satellite Initialization
  - Antenna deployment
  - Radio configuration
  - ADCS detumble
- Housekeeping
  - Telemetry acquisition
  - Telemetry logging
- OBC Executables
  - Command Handling
  - Scheduling of Commands
  - Time and alarm management
  - Programming
- Satellite Executables
  - Payload control
  - ADCS control
  - EPS control
- Error Handling

The Satellite Initialization differs from the others as it is one-time executable. After it has been run, it should never be run again and its lifespan is thus very short, and ends as the others begin. It is therefor not considered a software module like the others, but rather just an initialization procedure.

If one groups modules together with concern to functionality and underlying drivers rather than the top level services that must be provided one could make the following partitioning:

- Time management (Scheduling):
  - RTC
- Memory management (Logging and Programming):
  - Flash driver
- Health management (Housekeeping)
  - Sensors
  - I2C
- Communication management
  - Routing commands from ground station to satellite subsystems

- Failure management
    - Error handling procedures

By looking at the second list it may be easier to understand how the actual system can be constructed, as the partitioning is closer to what a developer might think a system is comprised of. From this list one also sees that logging and programming have similar functionalists as they both need to store and retrieve files from non-volatile memory. In this list there are no shared drivers between the modules and each module may be considered more like a software service. This may also be great for failure containment as if a driver fails, only the module implementing it will go down, not the others. The system may be able to continue operation if the failed driver functionality can be provided from elsewhere in the satellite, as was shown earlier.

By investigating the two lists the final modularization was obtained by discussion in the NUTS group and can be seen below:

- Timekeeper
- Housekeeper
- Memory manager
- Event manager

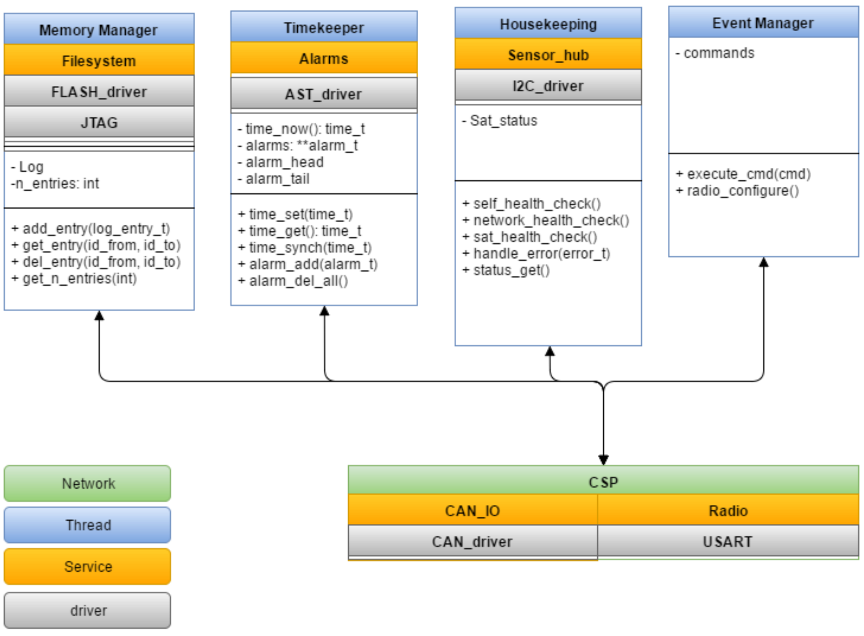


Figure 4.3: The Proposed Software Architecture

Communications are handled by CSP middleware though the CSP router task. This modularization was chosen as it inherently supports service rerouting while having minimal shared drivers, which in turn simplifies error confinement as well as multiple concurrency related issues. The architecture implements memory management as a service. This makes the memory interface available for calls from ground station, as well as enabling for easy service rerouting internally in the satellite in the event of flash failures.

Other alternatives were evaluated as well, and the two strongest candidates differed mainly in the amount of concurrency. One alternative suggested the use of shared memory methods for drivers as shown in Figure 4.4. This has the advantages of good extendability and good stratification. However as drivers are shared between services, they must incorporate mutual exclusion mechanisms. This concurrency can complicate the system design. Also in the event of hardware failure, all services that use the affected driver will go down, as driver rerouting is not easily supported. The other alternative represents a simpler architecture, where all OBC functions are implemented in one thread. This approach includes the least amount of concurrency and thus enables easy ”sharing” of information internally to the OBC. This may simplify how drivers and services are implemented, and may be the easiest way to structure the OBC. This implementation would contain one big switch case on incoming packages along with some interrupt handlers. The execution flow would be easy to comprehend and easy to test. However, there is an issue of throughput and prioritization. If there is only one ”consumer” thread receiving the messages from the the radio and the satellite bus, the buffers may overflow as the thread can only handle packages in between other procedures. If the OBC is busy doing a low priority but time demanding procedure, there are no easy ways of notifying the task that it needs to stop doing what it’s doing and handle the incoming packages first. This can lead to some serious issues especially considering that whole boot images may be sent to the OBC. Package drop on these are not acceptable as programming with a faulty boot image will render a module non-functional. This implementation is also vulnerable to single errors as any error appearing anywhere in the system may bring the entire OBC and all its functions to a grinding halt. For this reason, the service-oriented architecture is chosen as a middle point between the shared rivers and the single thread architecture.

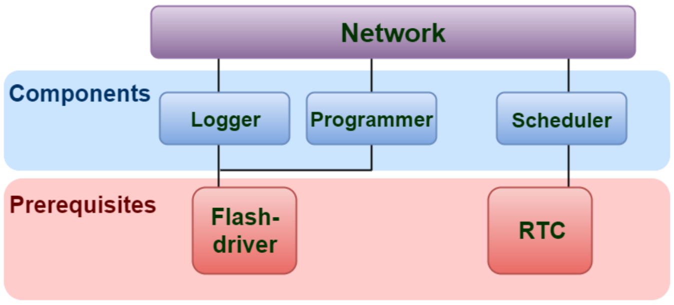


Figure 4.4: Shared driver flash driver between two software components

4.3 Methods for Maintainability

The suggested architecture uses a service-oriented architecture to enforce loose coupling and good reusability. No remote function calls are used, and thus every module can easily be isolated for development and testing. The establishing of service contracts before implementation aims to ease system integration.

To maximize extensibility as well as portability, emphasise is put on stratification through the use of hardware abstraction layers. This hides the underlying implementations from the application developer and eases reuse of code between different microcontrollers, such as the two used for the OBCs. The implementation is both hardware and operating system independent and can if wanted, be simulated on a Linux desktop if dummy drivers are implemented. Maybe more important, if future development calls for a shift to a Linux OS, the application code doesn't need to change, only the operating system. This applies for swapping out or restructuring hardware drivers as well. The layered design can be seen in Figure 4.5.

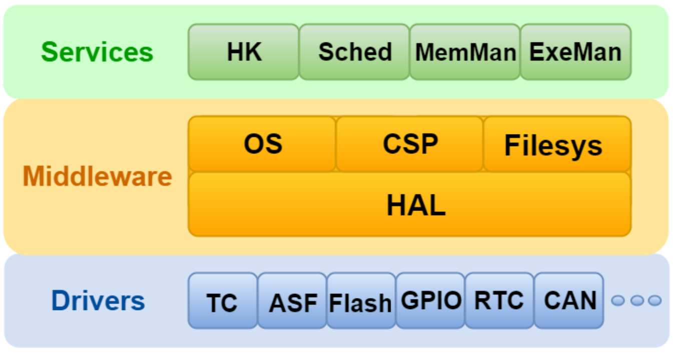


Figure 4.5: Architecture layers

4.4 Methods for Reliability

As discussed in the requirements section of this thesis, reliability is important for the OBC. Reliability is mainly supported in the architecture through easy implementation and rerouting of service redundancy. All OBC services are implemented in both OBCs. In the event of permanent failure in flash hardware or file system software, services can be routed from elsewhere in the system by changing the CSP routing tables, this can be done either automatically by the OBC or manually from the ground station. There are multiple other reliability measures chosen for the OBC such as the use of multiple watchdog timers. This will be further discussed in the detailed design section.

4.5 Methods for Power Optimization

It is common to reduce the power consumed by the microcontroller on which RTOS is running by using the idle task hook to place the microcontroller into a low power state. As the OBC will spend most of it's time inactive, the power consumption in this inactive state will dominate the overall power consumption and is therefor of great importance to minimize the power consumption in such a state.

Both the UC3C and SAMV71 support various low power modes. When deciding upon an appropriate low power mode, one must consider wake-up time and wake-up source. For the OBC there are considered to be three wake-up sources that must be able to wake the microcontroller from sleep. These are:

- Wake on internal timer (scheduling event)
- Wake on received message from radio
- Wake on received message from satellite bus

As different low power modes can be used for different solutions for the radio or the satellite bus communication, it is important that the choice of sleep mode is justified to the current design of other modules. A robust implementation of sleep modes should therefor support entering and exiting all viable sleep modes in order not to force later OBC designers to choose bad solutions for how bus or radio communication could be structured.

When using an RTOS, the power saving that can be achieved by this simple sleep method is limited by the necessity to periodically exit and then re-enter the low power state to process tick interrupts. Further, if the frequency of the tick interrupt is too high, the energy and time consumed entering and then exiting a low power state for every tick will outweigh any potential power saving gains for all but the lightest power saving modes [2]. To mitigate this issue and further optimizing the power consumption, tickless idle mode is implemented. The FreeRTOS tickless idle mode stops the periodic tick interrupt during idle periods, then makes a correcting adjustment to the RTOS tick count value when the tick interrupt is restarted.

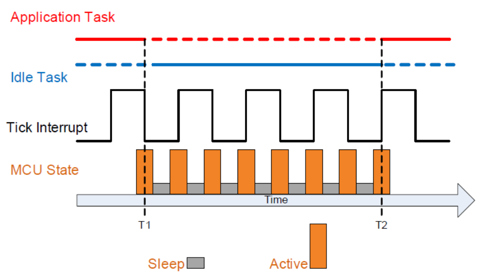


Figure 4.6: Common way for RTOS to do low power design [2]

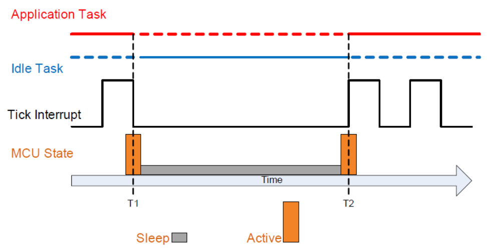


Figure 4.7: FreeRTOS Tickless Feature to do Low Power Design [2]

4.6 Summary

In this chapter, an architectural baseline was obtained through analysis and discussion. The resulting architecture is a layered, service-oriented architecture that aims to support the development and testing of services in complete isolation from each other. This was decided to facilitate a sustainable development environment that addresses the challenges imposed on projects with high turn-over of inexperienced personnel, such as student-driven CubeSat development projects. This chapter explored how the CSP library, already present in the OBC software repository, could be used to support this architecture, thus extinguishing the need for any additional third-party software libraries or new in-house developed frameworks. A decomposition of the system into four modules was then proposed, and the chapter ended by reviewing to what extent the non-functional requirements set forth thought the SRS could be satisfied through the given architecture. The outputs of this chapter forms the first part of the Software Design Document which adheres to the IEEE Standard 1016 Software Design Document and can be seen in Appendix D. The second part of this SDD comprises the detailed design for the architecture, which will be discussed in the next chapter; Detailed Design.

Chapter 5

Detailed Design

This chapter presents discussions and decisions made for the design of each of the OBC software modules. The conclusions as well as a short explanation form the module sections in the software design document that can be found in Appendix D.

5.1 Housekeeper

The housekeeper module is responsible for managing the health of the satellite. It should periodically issue tests to verify the correctness and liveliness of the satellite. The housekeeping tests can be made rather advanced, with individually testing each software module from multiple other modules. However an effort has been made to keep the error states as few and general as possible to avoid further complicating the system. Whenever there is an error discovered inside a sub-module, the entire module will be reset. This is done to avoid additional errors and complicated failure modes that might follow a single submodule reset/re-initialization. This means that although the system design supports intricate tests, where single software submodules may be tested with customized tests issued from various other modules, and the following results discussed among the other modules, the author emphasizes simplicity and only tests deemed absolutely necessary are to be implemented.

5.1.1 Error Recovery

As there are no hard deadlines, the added implementation and performance cost of static redundancy was not deemed cost-effective and thus dynamic redundancy is being used for fault tolerance. Anderson and Lee states there should be four constituent phases to dynamic redundancy. These are (1) error detection, (2) damage confinement and assessment, (3) error recovery, and (4) fault treatment and continued service [8]. Arnesen & Kiær presented a flowchart for high-level error checking and handling for the NUTS satellite in their thesis ”Mission Event Planning & Error-Recovery for CubeSat Application” [10]. Although this may be a competent way of verifying the module health in most cases, there may be some unintended behaviors as well. As can be seen in the flow chart below, when an error is detected, the response is to directly go into error recovery by resetting the module. The second stage in Anderson and Lee's model is skipped, as no attempt at damage confinement or assessment is attempted. By skipping this crucial second step, the procedure becomes vulnerable to errors originating outside the module, and will not be able so tell propagated error symptoms from the actual error cause. If for instance there is a network error, this procedure will try to communicate with a module, detect that it is not responding and try to reset the module, even though the module is perfectly fine. This will then happen for all the modules as the OBC tests them all individually. To avoid this, each test would have to be able to decide the proper follow-up tests whenever an error is discovered, in order to assess and confine it. This quickly becomes complicated as one must consider how these tests may also fail or warrant further follow-up testing. A better solution might be for these tests to only report errors, so that another entity, to which all errors are reported, can make the assessment and initiate any error recovery procedures. As the housekeeper is concerned with the health of the satellite, it seems fitting that such error handling is to be implemented here. With the argumentation seen above, it is decided that high-level error handling shall incorporate the three first stages suggested by Anderson and Lee. Procedures trying to unearth errors will run periodically, and report any errors when detected so that further investigations can be launched before any measures are taken to recover from the error.

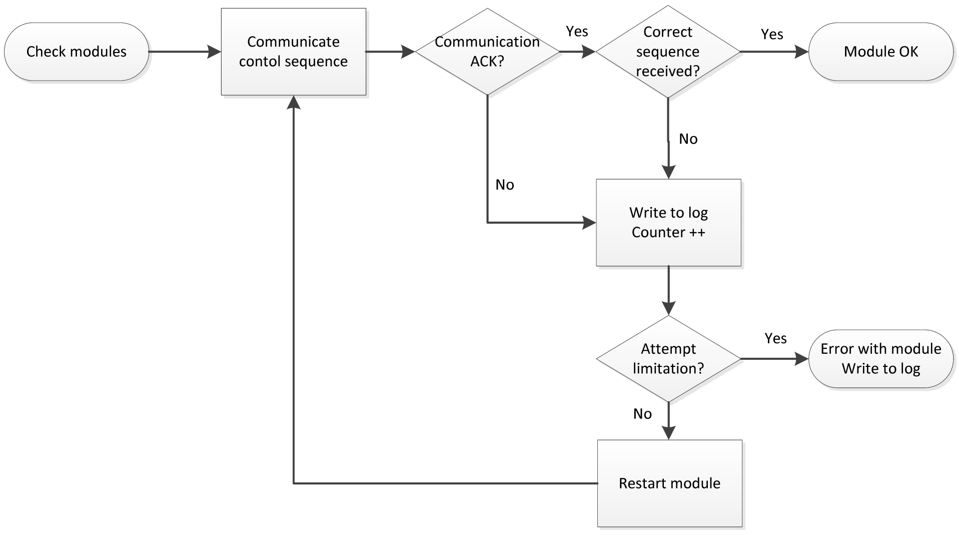


Figure 5.1: Previously suggested module check

5.1.2 Managing the Satellite State

In order to make the right decisions, the housekeeper will need to at all times know the current state of the satellite, as well as to have an understanding of previous events that have occurred in the system. This data must survive any module resets and should therefor be stored in non-volatile memory. Care must be taken however that the system still works if this non-volatile memory should fail. One solution might be to keep shadow-copies in various memories in order to have multiple places to get the information if one specific memory should fail permanently. The implementation and performance costs must be considered. The author proposes a scheme where a volatile working copy is kept in non-volatile memory, and is shadowed in MRAM, as well as being logged to non-volatile memory in the logging module. This way the system should function properly if one of the non-volatile memories fail permanently, and should be able to continue with a partial degradation in functionalities even if all external memories are broken. One could also synchronize the satellite's state between the OBC and its sister module, however when further investigating this method one discovers that this may be quite risky. Whenever a permanent error is detected in the OBC or its sister module, the one which can be considered the healthiest will be promoted to active state as the other goes into a passive state [9]. This means that if an OBC has a faulty memory and is still active, its sister module must be even worse off, and care should be taken before relying on services from the passive sister module.

5.1.3 Acquiring the Satellite State

All errors discovered through normal use shall of course be signaled to the housekeeper. In addition to this, the housekeeper shall periodically test the different parts of the satellite in order to uncover any silently failed modules. This periodic checkup should start by verifying the integrity of the OBC in order to assure that the most vital components are intact, as well as confirming that the testing facility is indeed functional before starting to test external modules. If no errors are detected the network should be tested, before finally, the other modules of the satellite can be tested. It is important that the network check is done before any external modules are checked as external modules cannot be tested without the interface being functional.

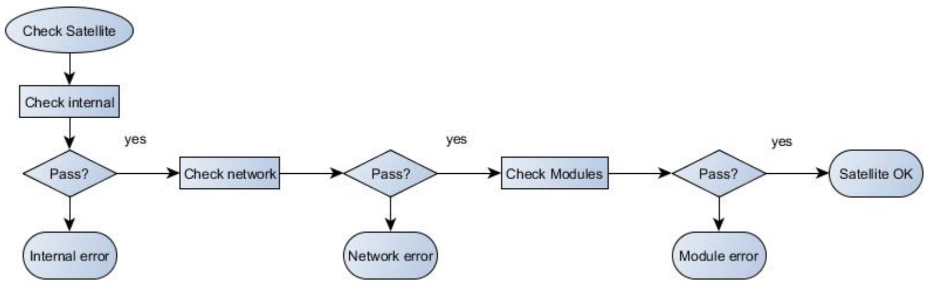


Figure 5.2: Satellite check

5.1.4 Internal Integrity Check

Burns and Wellings states that the major application detection techniques are replication checks, timing checks, reversal checks, coding checks, reasonableness checks, structural checks, and dynamic reasonableness checks [8]. Most suited for detecting memory corruption is the coding checks, and thus this is what shall be used to detect whether the program data has been corrupted. When it comes to testing the external memories, both timing checks, as well as code checks could be useful. The timing checks should be made on the read and write commands, to ensure that an error does not block eternally, but rather times out and reports the failure. If the timing however checks out one could use coding checks to verify that not only the access is functional, but also that the contents are intact.

After the various memories have been tested, the different internal software modules can be tested. These tests can consist of anything from a simple ping-pong routine, to including just about every method suggested by Burns and Wellings. The author of this thesis suggests to not go all overboard with this but only implement simple functional tests. The memory manager facility can be tested by requesting a write to log, and then a read-back of the same entry. The timekeeper facility can be tested by adding an alarm that should be triggered in the immediate future and do a timing check on the response. The event handler facility can be tested with a simple ping-pong routine.

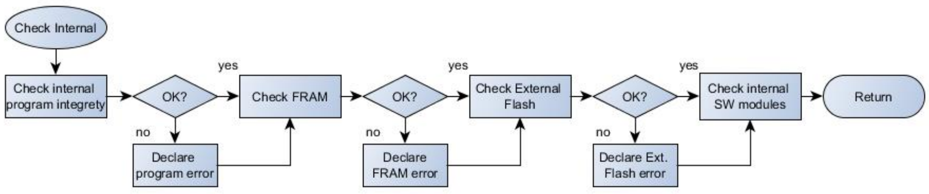


Figure 5.3: Internal check

5.1.5 Satellite Network Check

The satellite network check can be made by simply pinging all other modules, and wait for responses for a limited time. If all modules respond within the time limit, the network is assumed functional. If some modules respond, but not all, the network is considered partially functional and an exception is thrown. If no responses are received, the network is assumed to be non-functional.

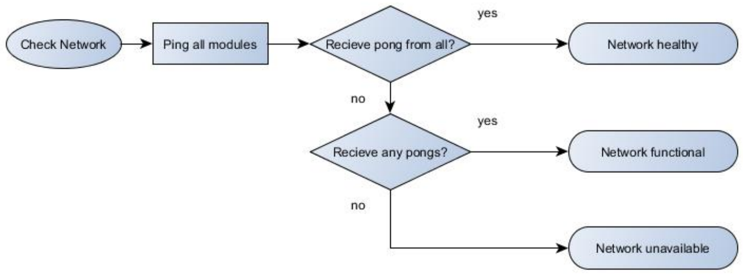


Figure 5.4: Network check

5.1.6 Module Check

As all modules are interfaced through the CSP sockets, performing functional timing tests are easily implemented as each call to socket receive can be given a timeout period, specifying how long the socket should block and wait for an incoming message. In fact, one OBC can perform exactly the same functional tests on the other OBC as it did on itself by simply addressing the respective socket on the other OBC instead of the one implemented in itself.

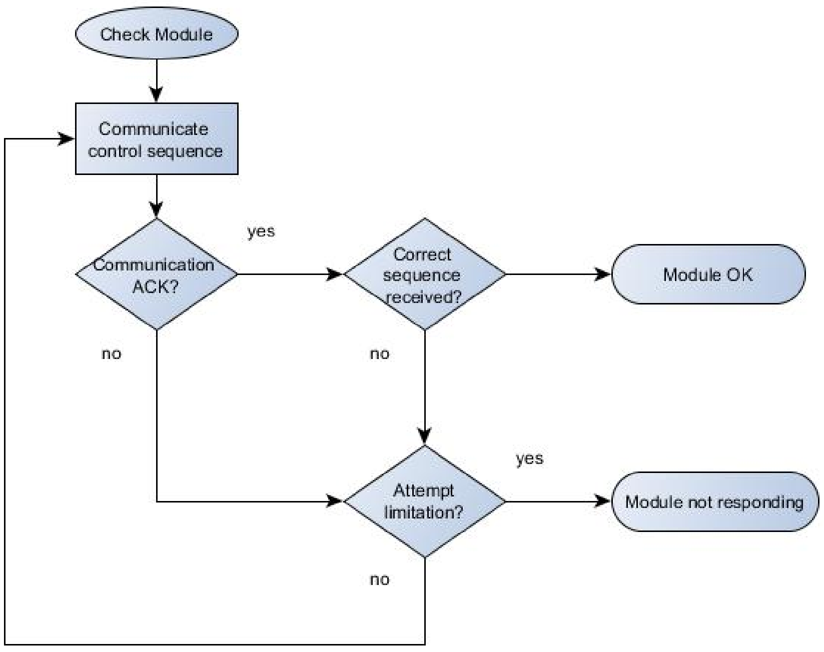


Figure 5.5: Module check

For example, to verify the memory manager on the OBC-sister the OBC can request to write to and read from the log, but instead of using addressing the requests to its own memory management socket, it can send them to the socket of the memory manager at the other OBC.

5.1.7 The Error Handler

As error handling code in itself can be a source of errors, the complexity of error handling code should be minimized if a maintainable system is to be constructed. It is therefore recommended to implement basic catch-all measures rather than specialized handlers for complicates error states. As a result of this the author recommends that a complete module reset is a better response than single submodule repair in the face of soft errors. One of the main arguments for this is that it makes for easier maintainable code as well as error propagation may be minimized.

Upon reset, the housekeeper checks the battery level to see if the satellite can enter the test mode. If the battery power level is sufficient the housekeeper starts a health check. First it checks its own host-module. The non-volatile memory is checked for any status information, as well as running an integrity check for the rest of the non-volatile stored data. If any errors are unearthed in the self test, these must be dealt with before testing the rest of the system. However, the internal test is still to be completed, even if a previous step uncovered an error. This is done to get an overview of the situation before making any decision on what the appropriate counter measures are. This is in turn done to ensure that multiple errors are registered and handled correctly. If the self test is positive, the other satellite modules are tested as the flowchart below depicts. Any failed tests, result in a limited amount of module resets before a module is declared dead.

5.1.8 Interface for the Housekeeper

Call	**Parameters**	**Reaction**
PING	RSVP_SOCKET	Send PONG to RSVP_SOCKET
PONG	NONE	Register response
RUN_HEALTH_TEST	NONE	Run a full satellite test
ERROR_REPORT	error_t error	Do necessary actions, such as logging the error and saving the state, before resetting the module
SYNC_TIME	time_t time	Synchronize all RTCs in the satellite with the time given, if no time is given synchronize using the timekeeper time.
GET_SAT_STATE	RSVP_SOCKET	Retrieve the current satellite state and send it to RSVP_SOCKET.

5.2 Timekeeper

The OBC specification demands that the OBC needs to keep track of time cf. requirement R01-OBC-EXE-TIM-001. This can be done in multiple ways, some important design choices are considered to be whether the OBC should keep absolute time or relative time, and if the timekeeping should be implemented internally or externally to the OBC.

5.2.1 Absolute Time vs Relative Time

Absolute time refers to using a universal time, for instance, the satellite could adhere to the Greenwich Mean Time (GMT). This would make it easy to compare timelines from the satellite to that of earth. The downside is that this time would have to periodically be synchronized via communication to make up for drifting due to inaccurate hardware components. The alternative is to use relative time, meaning that the time counting in the satellite is decoupled from that on earth. All that is important is the relative time between events. For instance if a picture needs to be taken 42 minutes in the future, with relative time the only thing that matters is that there is 42 minutes between now and the time the image will be taken. The satellite does not need to know that the time now is 12:55, and that the picture needs to be taken at 13:37. The advantage of using relative time is that there is no need to synchronize the clock as the short drift experienced from an event is planned to the event needs to be executed most likely will be negligible. However, although using relative time at first glance may seem simple, it quickly becomes more difficult when considering resets and synchronizing time across the satellite. When a module is reset, it is not trivial for that module to know how to reset alarms. If the reset was system wide, no part of the satellite can know when to reset the alarms. It thus seems that no matter if the satellite uses absolute time or relative time, it will need to compare time with earth. As this renders the main advantage of using relative time moot, absolute time is chosen for simplicity of implementation and synchronization. When a system wide reset occurs the time in the satellite will be reset to a preset time and will be out of synchronization with earth until it is synchronized from the ground station.

5.2.2 External or Internal timer

The main advantages of using an external Real Time Clock (RTC) is considered to be the added accuracy that might be obtained, and the fact that the RTC doesn't necessarily need to be reset when the OBC is reset. However the RTC will have to be reset when the module it's situated on resets, extra batteries are not acceptable as it must be able to disconnect the RTC from power (in case of latch-ups and other soft errors) and this adds more complexity to the system than what is warranted for by the payoffs.

The timer will mostly be used for scheduling orientation changes or image capturing, neither which need to be very accurate in time, a couple of seconds drift are unlikely to affect the result. High accuracy is thus not considered to be imperative to the satellite mission. The UC3C microcontroller is equipped with an Asynchronous Timer (AST), and the SAMV71 with a Real Time Clock (RTC) that can be driven from different clock sources, according the different needs of power consumption and accuracy. If accuracy is emphasized one can invest into a high performance external 32KHz crystal oscillator, care must be taken however as temperature effects the frequency and most crystal oscillators are optimized for room temperature. In any way, the AST/RTC is deemed fit concerning the accuracy, as a little drift is acceptable, but also can be minimized by using an accurate external crystal.

Having discussed accuracy, an external RTC still has the advantage that it can be situated at another module and thus keep time despite OBC resets. However this also holds true for using the AST implemented in the sister-OBC for timekeeping. As this needs no extra external components or drivers it may be the more elegant solution. A counter argument is that the chances of more advanced modules failing is much greater than that of small simplistic ones such as a stand-alone RTC. If however, the internal AST is used for timekeeping, and when possible and appropriate, synchronizes with external timing devices such as the AST/RTC in the sister device, one may get a more robust solution.

As the philosophy of this software design is that as much as possible should function under the least amount of assumption with the least amount of implementational effort, the internal AST/RTC will be used to keep time internal to the module, as it is much more likely that any connection with and external timekeeping device will go down, than that the internal AST/RTC should malfunction. However the internal AST/RTC should be synchronized with other timekeeping modules as often as appropriate, but it is important that care is taken so that the time service of the OBC does not seize to function in the event of lost connection to external timekeeping devices.

5.2.3 Design of the Timekeeper

The internal AST/RCT will be used to keep absolute time, however the OBC specification demands not only that the OBC needs to keep track of time, but also that it must be able to set alarms to schedule future executions cf. requirement R01-OBC-EXE-TIM-002. The AST/RTC only supports two alarms; ALARM0 and ALARM1. Any scheduling thus needs to reuse alarms. This can be solved by keeping a sorted array of set alarms where ALARM1 is set to the first one in the array, which will always be the one closest in time.

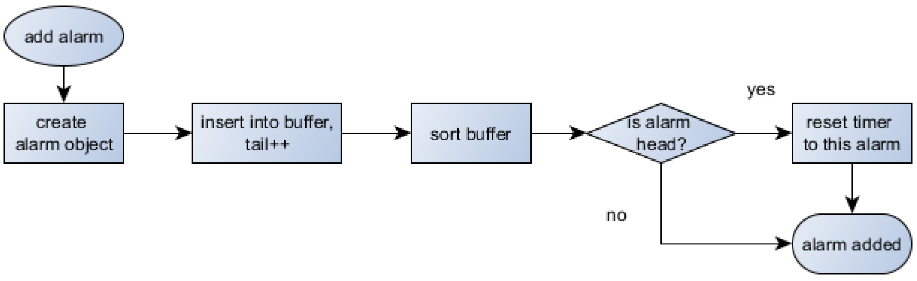


Figure 5.6: Adding new alarms

When the alarm is triggered it resets itself to the next alarm in the array. For memory purposes, the alarm is implemented as a ringbuffer of a fixed size, and new alarms are added using bubble sort (from back to front). Bubble sort is chosen for its simple implementation and the fact that if one makes the assumption that most new alarm times are likely to be further into the future than the ones already present in the buffer, this minimizes sorting operations. A faster alternative could be to use a dynamically allocated linked list, instead of the ring buffer and sorting, as this would reduce both the time and memory usage. However due to the added complexity and the difficulties related to debugging problems linked to dynamically allocation, the sorted ringbuffer-method is deemed safer. Alarms should be shadowed in non-volatile memory in case of any resets. The time can be set externally (the housekeeper is responsible of synchronizing time between the modules.) The timekeeper module itself should be as simple as possible, and no self-synchronizing should be implemented. This is done to avoid complex and/or uncontrolled behaviors in the OBC. All time synchronization should happen on the initiatives of the housekeeper or the event handler as these modules have a much better image of the satellites situation.

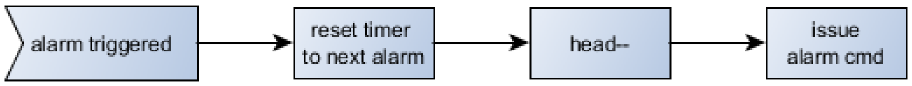


Figure 5.7: Alarm triggered

5.2.4 Interface of the Timekeeper

The Timekeeper is to be implemented with one blocking listening socket TIMEKEEPER_SOCKET_RX that shall receive and react to calls concerning time and alarms. The calls it shall respond to can be seen in the table below.

Call	**Parameters**	**Reaction**
PING	RSVP_SOCKET	Send PONG to RSVP_SOCKET
PONG	NONE	Register response
TIME_GET	RSVP_SOCKET	Send current time to the RSVP_SOCKET
TIME_SET	time_t time	Update the current time to the one received
ALARM_SET	time_t time, action_t action	Add alarm to execute ”action” at ”time”
ALARM_DEL_ALL	NONE	Delete all alarms

The action type received as parameter in the ALARM_SET call is to be from the same action set that is implemented within the event handler.

In addition to the TIMEKEEPER_SOCKET_RX the timekeeper shall also use an outgoing socket named TIMEKEEPER_SOCKET_TX, which is to be used when responding to time requests and triggered alarms.

5.3 Memory Manager

The memory manager is the software service responsible for storing and retrieving log entries, boot images, as well as reprogramming other modules. It is implemented as a single thread with one listening socket, MEMMAN_SOCKET_RX. The memory manager will be the only task that has direct access to the hardware keeping the log (i.e. flash memory) and will thus need to act as a server, responding to the logging requests of the rest of the system.

5.3.1 Timestamp Supplier

All log entries should be timestamped in order to construct time lines of event. One question that arises is then; should the time be supplied by the caller, or by the memory manager? It is not unlikely that some drift will occur between the different clocks in the satellite, despite valiant efforts of synchronizing them. Would it then make more sense to use the time from the local timekeeper, assuring that all entries are stamped chronologically to when they are received, or should the caller stamp the log message before sending, assuring that no delays due to packet loss or resending affects the time stamping? One solution might be to use both, this would effectively mark each log entry both when it was sent, and when it was received. Making it easier to investigate synchronization or communication issues in the satellite.

5.3.2 File System

The memory manager should support some sort of file system to categorize different entries. When choosing a file system care must be taken both to the memory footprint of the system as well as its reliability. It is not unlikely that power outs or outside of spec voltages may occur. It is therefore important that the memory manager does not corrupt the memory, even if operations are aborted half finished. For this reason the developer may want to use a journaling file system, as such file systems can be brought back online more quickly with lower likelihood of becoming corrupted after power failures [44] [45]. The developer may also want to investigate whether the Reliance Edge fail-safe file system included in FreeRTOS+ Ecosystem [46] is an appropriate file system or not.

5.3.3 In-Orbit Programming

Due to R00-OBC-PRG-001 requirement, the OBC must be able to program other modules after launch. This firmware update can be done either by the microcontroller itself or by another microcontroller. Priority is given to programming of one microcontroller by another, as this can be forced upon a microcontroller with a faulty firmware. Marcedella implemented firmware update using the JTAG protocol. However his method is target specific and does not easily port to other microcontrollers or boot images. For this reason a new programming scheme needs to be implemented, though it is recommended to use Marcadellas work as guiding.

5.3.4 Interface for the Memory Manager

Call	**Parameters**	**Reaction**
PING	RSVP_SOCKET	Send PONG to RSVP_SOCKET
PONG	NONE	Register response
LOG_ADD	log_t log, time_t time	Write the log entry into the log.
LOG_GET	RSVP_SOCKET, int N	Send the N last log entries to RSVP_SOCKET.
LOG_DEL_N_LAST	NONE	Delete the N last log entries.
LOG_INIT	NONE	Initialize the LOG, any old log entries will be deleted.
BOOT_ADD	boot_image[]	Store boot image in flash
BOOT_GET	RSVP_SOCKET, boot_image_id	Retrieve boot image from flash
PROG_FIRM	Module, boot_image_id	Reprogram module with firmware boot_image_id

5.4 Event Manager

The main responsibilities of the event handler is to execute composed commands, enabling simple commands from the base station to result in multiple commands to be issued inside the satellite. Originally it was planned that all messages would be unpacked and inspected and then routed to the correct module by the event handler, however as the use of CSP enables routing of packages directly to the correct module, this is alleviated from the event handler. This means that though all actions could be performed by sending an array of commands to be executed from the ground station, it is considered to be preferable if these action sequences were grouped together and triggered by a simple command. This forms an easier interface where the exact sub-commands and sequences are abstracted from the user interface presented to the ground station.

Interface for the Event Manager

The Event Handler is implemented as a single thread with one listening socket, EVENT_SOCKET_RX. The Event handler will in respond to events or commands that entails complex or composite actions to be executed. The list is to be made.

5.5 Summary

In this chapter a more detailed discussion for the implementation of each of the OBC software modules were made. The discussions aims at presenting the developers with a plan for how the software should behave and what methods that could be used. Some sections, such as those describing the behavior of the housekeeper are detailed with flowcharts and activity diagrams to better convey the intended behavior in the face of errors. Other sections such as the ones describing the logger and the event manager are not as detailed and leaves the developer free to make choices onto how the needed behavior is implemented. This reflects the level of freedom the developer has in accordance with the rest of the satellite. While decision on how time should be described will affect the whole satellite, the choice of implementation behind the file system will be abstracted behind a socket interface, thus granting the developer greater freedom. This chapter forms the second part of the SDD, that has been made to comply with the IEEE Standard 1016 Software Design Document and can be seen in Appendix D.

Chapter 6

Implementation

To be able to support the development of each of the OBC services, the framework of the architecture should be implemented. This chapter explains how the operating system and the communication framework for the OBC were configured and installed, as well as the tools for RTOS trace debugging and power optimization. The output of this section is an implementation of the proposed software architecture of Chapter 4. The implementation aims to provide future NUTS OBC developers with a framework for the isolated development, debugging and testing of OBC services. Heavy emphasise is made on the maintainability and ease of use for future developers.

6.1 Setting up FreeRTOS for SAM V71

In Chapter 3 it was decided that the OBC should sport an RTOS. In Chapter 3, FreeRTOS was chosen as this RTOS. In this chapter the setup and configuration of the OBC RTOS will be further elaborated upon, starting with choosing the FreeRTOS version, before discussing the configuration.

6.1.1 Choosing FreeRTOS version

FreeRTOS source code is available from Sourceforge (https://sourceforge.net). The latest release candidate is v9.0.0rc2, where the RTOS kernel updates compared to the latest official released version (v8.2.3) can be seen in the list below [47]:

- Major new feature - tasks, semaphores, queues, timers and event groups can now be created using statically allocated memory, so without any calls to pvPortMalloc().
- Major new features - Added the xTaskAbortDelay() API function which allows one task to force another task to immediately leave the Blocked state, even if the event the blocked task is waiting for has not occurred, or the blocked task's timeout has not expired.
- Updates necessary to allow FreeRTOS to run on 64-bit architectures.

- Added vApplicationDaemonTaskStartupHook() which executes when the RTOS daemon task (which used to be called the timer service task) starts running. This is useful if the application includes initialisation code that would benefit from executing after the scheduler has been started.
- Added the xTaskGetTaskHandle() API function, which obtains a task handle from the task's name. xTaskGetTaskHandle() uses multiple string compare operations, so it is recommended that it is called only once per task. The handle returned by xTaskGetTaskHandle() can then be stored locally for later re-use.
- Added the pcQueueGetQueueName() API function, which obtains the name of a queue from the queue's handle.
- Tickless idling (for low power applications) can now also be used when configUSE_PREEMPTION is 0.
- If one task deletes another task, then the stack and TCB of the deleted task is now freed immediately. If a task deletes itself, then the stack and TCB of the deleted task are freed by the Idle task as before.
- If a task notification is used to unblock a task from an ISR, but the xHigherPriorityTaskWoken parameter is not used, then pend a context switch that will then occur during the next tick interrupt.
- Heap_1.c and Heap_2.c now use the configAPPLICATION_ALLOCATED_HEAP settings, which previously was only used by heap_4.c. This allows the application writer to declare the array that will be used as the FreeRTOS heap, and in-so-doing, place the heap at a specific memory location.
- TaskStatus_t structures are used to obtain details of a task. TaskStatus_t now includes the bae address of the task's stack.
- Added the vTaskGetTaskInfo() API function, which returns a TaskStatus_t structure that contains information about a single task. Previously this information could only be obtained for all the tasks at once, as an array of TaskStatus_t structures.
- Added the uxSemaphoreGetCount() API function.
- Replicate previous Cortex-M4F and Cortex-M7 optimisations in some Cortex-M3 port layers.

FreeRTOS v9.0.0rc2 is a release candidate, meaning it is a beta version with potential to be a final product, which is ready to release unless significant bugs emerge. In this stage of product stabilization, all product features have been designed, coded and tested through one or more beta cycles with no known showstopper-class bug [48]. However the developers of FreeRTOS Real Time Engineers ltd. state that testing is still not finished [47], and thus, unearthed bugs may still exist and features may still be added or removed. This makes FreeRTOS v9 a poor choice for high reliability space applications. Luckily FreeRTOS V9.x.x is drop-in compatible with FreeRTOS V8.x.x [47], meaning that even if the older, safer official release (v8.2.3) is used, it should be a smooth transition if migration to FreeRTOS v9.x.x is warranted after its official release. FreeRTOS v8.2.3 is therefor chosen as the operating system for the OBC as it is the safest option, but also does not complicate potential future version updates.

6.1.2 Choosing Memory Scheme

A real time operating system kernel has to allocate RAM dynamically each time a task, queue, or semaphore is created [49]. The use of the standard malloc() and free() library functions can by accompanied with some undesirable side effects for one or more of the following reasons [50]:

- Their implementation can be relatively large, taking up valuable code space.
- They are rarely thread-safe.
- They are not deterministic; the amount of time taken to execute the functions will differ from call to call.
- They can suffer from memory fragmentation.
- They can complicate the linker configuration.

To be able to best satisfy the different requirements of different applications, FreeRTOS allocates memory by calling pvPortMalloc() and frees it by calling vPortFree(), where the implementations can be decided by the programmer. FreeRTOS v8.x.x contains five implementations of pvPortMalloc() and vPortFree, each designed to address different application requirements. A short comparison of the implementations made by Diaa Jadaan can be seen in Table 6.1 [49].

	Deterministic	**Code size**	**Segmentation**
heap_1	Yes	Small	No
heap_2	No	Small	High
heap_3	No	Large	Target dependent
heap_4	No	Moderate	Moderate
heap_5	No	Moderate	Low

Table 6.1: FreeRTOS malloc implementations, as presented by Jadaan

Generally speaking, it is not recommended to use dynamic memory in embedded systems, for such systems the heap_1 implementation would be the most appropriate. However as NUTS communication system needs to dynamically allocate queues and buffers depending on the various communication channels that are opened and closed, a more sophisticated memory scheme is needed. In the paper ”Memory management and error handling in FreeRTOS for a CubeSat project” Diaa Jadaan evaluates the different memory schemes with the NUTS project in mind. He concludes that heap_4 is the best fit for its compromise between the required overhead and segmentation level. He adds that heap_5 might be desired in some cases for its additional support of heap spanning multiple non-contagious memory regions, but that the additional code size and operation overhead should be evaluated against the actual usability. By inspecting the source code for the different implementations one finds that for FreeRTOS v8.x.x heap_4 contains 473 code lines in its C implementation file, while heap_5 contains 522. However if lines only containing comments are removed, one is left with 246 and 254 lines, respectively. Though this is not an exact metric for the code size, there are no additional calls to external functions in heap_5 compared to those in heap_4, and thus the code size differences are considered to only be minimal. As the NUTS satellite will spend most of its time sleeping, and dynamic memory is only used when justified, calls to allocate and free memory are considered to occur quite seldom. It thus seems unlikely that the small computational differences in heap_4 and heap_5 will have much of an impact on the system as a whole. However, since heap_4 is the only heap implementation for v8.2.4 that supports the use of configAPPLICATION_ALLOCATED_HEAP, it is considered the best alternative as this is a prerequisite for placing the heap on external RAM.

6.1.3 Configuring FreeRTOS

FreeRTOS is customised using a configuration file called FreeRTOSConfig.h. Every FreeRTOS application must have a FreeRTOSConfig.h header file in its pre-processor include path [51]. For the OBC application the configuration file is included into the config folder along with all the other configuration files. There are many settings one can configure for FreeRTOS, the most important will be shortly discussed below.

Scheduling
Preemptive scheduling is being used as this is the only mode that supports tickless sleep for FreeRTOS v8.2.3.

Heap Size
FreeRTOS heap is configured to be $384Kb * 50\% = 198KB$. This way, whenever an application developer violates the memory constraint, the malloc fail will be caught in the respective FreeRTOS hook and notify the programmer of the violation. If this happens due to some last minute fix, the heap size can safely be increased. However if it happens in normal design situation, the developer should further investigate the memory usage of the application and try to optimize the memory usage before proceeding.

Memory Violation and Debugging
FreeRTOS has support for stack overflow checking. This can be done in two ways. The first method (method one) is the fastest, but only checks that the processor stack pointer is within legal range when a task is switched out of the running state. To catch past overflows method two can be used, which includes the same check as method one, but also checks the last 16 bytes of the valid stack range, to see if they have been overwritten. This last method is very similar to that of using stack canaries. If a stack overflow is detected, the stack overflow hook is fired. Malloc fail hook is also enabled. As well as the use of trace facility (needed for hooks used by Percepio Trace Snapshot library).

FreeRTOS does include two ports for ARM Cortex-M3 microcontrollers and two ports for ARM Cortex-M4F microcontrollers - the standard FreeRTOS port and FreeRTOS-MPU. FreeRTOS-MPU includes integrated memory protection. As both OBC microcontrollers embeds a Memory Protection Unit (MPU), finding a FreeRTOS port that supports this would be preferable. Unfortunately FreeRTOS does currently not include any such port for the OBC MCUs, and thus the regular port is used [52].

6.2 Installing FreeRTOS Trace Tool

In the architecture chapter it was decided that Percepio Trace Recorder should be implemented as part of the OBC software architecture. Percepio Trace Recorder consists of two components - a PC application with a Graphical User Interface (GUI) and a trace logging library. The trace logging library is provided as C source code. The library uses the standard FreeRTOS trace macros, which are empty macros that application can redefine for purpose of providing application specific trace facilities. It has an adjustable RAM footprint (uses a ring buffer) 5-10 KB gives about 50-200 ms at normal rates for an ARM-M [53]. For the NUTS application a 10KB buffer was allocated, this size was chosen somewhat arbitrary and any future developers should feel free to increase or decrease its size. It should be remembered though, that the trace library is intrusive and that it shall not be used in-orbit. Timings of all operating system calls will be different when trace is enabled. It is therefore important that it is used as a tool for debugging, but should be switched off for testing.

6.3 Porting CSP for SAMV71 and UC3C

The existing NUTS repository already contained a compiled version of the CSP library. However this library was compiled for the UC3A3256 microntroller, with I2C support and no additional CSP features enabled. As the NUTS project has upgraded the satellite bus from I2C to CAN, as well as the OBC MCUs from two UC3A3, to one UC3C and one SAMV71, the library needed to be recompiled.

There isn't much documentation available for the public concerning CSP. Getting started can therefor be challenging, as for the most part, the only documentation you have is the source code. However, the source code is not compilable as is, and needs to be configured before it can be built and installed. GomSpace uses a Python-based framework called waf for configuring, compiling and installing CSP applications [37]. Jahren focused his project work and master thesis on getting CSP and reliable message passing to work on the UC3A3 microcontroller. He states that the waf system is not very intuitive, and there is not much useful information about how to use it for the CSP system online either [14]. He therefor made a step-by-step guide on how to install CSP for the UC3A3256 microcontroller. However this guide does not describe why or how the different commands were determined, but rather just lists a series of steps and specific configuration commands used to build CSP for this specific microcontroller and application. In addition to this, there has been made some update changes to the waf library. These changes seem to change how the toolchain input string is tried executed as a program. Trying to build CSP using the procedures suggested by GomSpace (not updated since 2011) or by Jahren does not work. Using the old waf system does work, but only for the UC3A3256 microcontroller and with older versions of CSP.

Since all the source code for both CSP and waf are publicly available, it was decided that it should be possible to manually build and install CSP without using waf. After investigating the source code behind the configuration script, CSP and waf, the CSP source code was downloaded and a manual configuration process started. The result is a CSP distribution that is not prebuilt, and thus easily reconfigurable between each build of the application project. A header file, including defines for the different CSP options was created and named conf_csp.h. The csp library can thus be configured without having to perform the whole step-by-step building procedure, but rather by changing a couple of defines in the conf_csp.h file. This includes settings such as endianess and freeRTOS version, thus making the source code easily portable to different microcontroller projects, including SAMV71 and UC3C.

As all satellite subsystems communicate through the use of CSP, it would be of great advantage if each module ran the same CSP code to avoid any version mismatch. A separate git repository was therefore made to track changes to the CSP code. All CSP files are included in this repository, the only exception is the conf_csp.h file as this may need to be configured differently for different microcontrollers. Having a git repository tracking the changes made to CSP files, makes it easy to integrate any CSP features on a satellite-wide basis. If for instance a developer working on the OBC software defines a new packet structure for a specific service and needs all other modules to recognize this structure, he'd need only to implement support into the CSP code used by the OBC, verify it, and then push the changes to the remote CSP repository. The next time any developer of any other module would pull the changes from CSP remote, his module would automatically incorporate the changes implemented by the OBC developer. Also, if any coding errors were detected, all submodules could revert to an older safer commit.

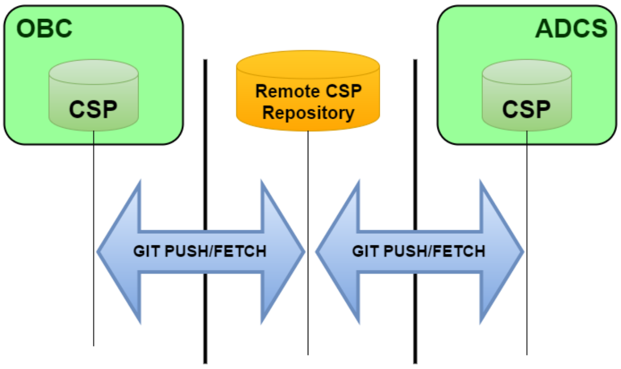


Figure 6.1: CSP Repository workflow)

Another advantage of not precompiling the CSP library and keeping it in a separate git repository, is that changes to the original GomSpace repository can be pulled into the NUTS CSP repository if needed. This is especially important when errors are detected and fixes implemented in the original repository. These bugfixes may need to be implemented for the NUTS project and thus, pulling these commits is a desirable feature. One should however be careful when pulling commits from the GomSpace repository, as all updates may not be applicable to the nuts project. Care should therefor be taken before pulling changes from the original repository.

After comparing the manual configuration with the step-by-step guide made by Jahren it became clear that the build method for the UC3A3256 microcontroller did not enable for resending of packages. The option of enabling this has now been included in the conf_csp.h file, and testing will show if the new CSP configuration supports detection and resending of lost packages or not. If this feature proves functional, the added layer formerly called NUTS Realiable Protocol (NRP), will be redundant and can be excluded, freeing up precious memory, and minimizing complexity.

The only downside of not precompiling the CSP library is considered to be the added warnings related to CSP source code, that now are displayed upon building of the submodule application. The warnings are considered to be harmless, as most of them are due to the use of ”packed” keyword for various CSP package structs. The keyword packed tells the compiler to not use any padding between the variables, thus mitigating any problems related to compilers choosing different amount of padding for different architectures. This struct layout does however cause inefficient memory alignment, which produces a compiler warning. The keyword is kept as it serves a purpose, but having 150 compiler warnings is not ideal, especially not for high reliability applications. It is however considered to be an acceptable price to pay for the various advantages described above.

6.4 Implementing Sleep Modes and Tickless Idle

The sleep modes and Tickless Idle is implemented in the middleware folder, as they are considered an extension of the operating system. A configuration file conf_sleepmgr.h is used to choose what sleep modes should be used, as well as some other configuration settings such as shortest allowable sleep time. A timer/counter must be used in order for the CPU to wake up on scheduling events. The timer used for the FreeRTOS tick is the SYSTICK timer embedded in the ARM Cortex. This timer is however unpowered in all low power modes except the lightest one. Therefor an additional timer must be used to allow for deep sleep. For the UC3C the AST timer can be used as a wake-up source from all sleep modes, the same goes for the RTC and the RTT implemented in SAMV71. Unfortunately both OBCs will run their AST/RTC in calender mode, meaning that the smallest time step for sleep is one second. For the UC3C OBC this cannot be avoided, but for the SAMV71 OBC, the RTT can be used in free running mode, resulting in a better precision. As prescaler values 1 and 2 are forbidden prescaling value of 3 is used. As the input clock is 32768 Hz, this results in a $32768/3 = 10922Hz$ frequency. This gives a time step of 0.092 ms, far more accurate than needed for the 1 ms tick OS. The longest possible sleep with this setting is $(2^{32}/(32768/3))/(60 * 60) = 109hours$ which is considered to be longer than needed. The settings are therefor considered to adequately support both short and long periods of sleep.

The power consumption analysis with the results for different power modes can be found in the testing chapter.

6.5 Implementing Skeleton Code and Tests

The command and data handling subsystem (C&DH, OBC) is often one of the last on the spacecraft to be defined. It is a tool used to configure control, or program the payload and other spacecraft subsystems. C&DH equipment cannot be completely defined until the requirements of other systems have been established [19]. This is true for the NUTS OBC as well, neither of the other modules are at a point in development that they are connected to the databus or that they have detailed requirements for their interfaces. This makes it highly impractical to implement specific functionality in the OBC as the interface and service contracts of other modules may still change. However to facilitate further development of the OBC software, a series of steps have been made through this thesis. As part of this, skeleton code was structured to convey the design philosophy as well as to give new developers a quick understanding of how the code should be structured when the process allows for it. The purpose of this is to supply later developers with a finished framework for what needs to be done, and how it should be done. The skeleton code implements the different modules and hopefully makes it easier for later developers to focus on their specific task rather than the whole OBC system.

It takes significant effort and awareness to write a software component that is effectively reusable. The component needs to be [54]:

- Fully documented
- Thoroughly tested
- Robust - with comprehensive input-validity checking able to pass back appropriate error messages or return codes
- Designed with an awareness that it will be put to unforeseen uses

The skeleton code implements the different services and shows how they can use sockets to communicate. In addition to this, the folder structure is implemented, with specific readme files in each folder which are to describe the purpose of its contents. When no implementation files have been made yet, the folder and its readme is still created to ensure that the layered approach is upheld. The folder structure can be seen below:

- docs/
  - obc_sw.txt
  - service_contracts.txt
  - todo.txt
  - README.txt
- lib/
- include/
- src/
  - services/
  - middleware/
  - drivers/
  - ASF/
  - config/
  - main.c
  - README.txt

- obc_v71.cproj
- README.txt

6.6 Refactoring

As the final step of any software implementation, the code should be reviewed and restructured for eased maintainability. Passing tests show correct behavior, but there's more to software than correct behavior. Code has to be kept clean and well structured, showing professional pride in workmanship and an investment in future ease of modification [16].In Martin Fowler's book Refactoring: Improving the Design of Existing Code [55], he describes refactoring like this: refactoring is the activity of changing a program's structure without changing its behavior. The purpose is to make less work by creating code that is easy to understand, easy to evolve, and easy to maintain by others and ourselves. This refactoring was the final step of the implementation section, and is a recommended practice for all later OBC developers.

Chapter 7

Testing and Verification

This chapter presents the performance tests that were made for the OBC software architecture. Each test has been designed to investigate a specific matter or function of the architecture framework, and will be presented along with the results of the test. Each test and result section includes a discussion on what can and cannot be deducted from the acquired results.

7.1 What is Testing All About?

Testing accounts for more than half of the time spent on projects. The reward for finding a defect early in the process is at least a tenfold saving compared to finding it at integration time, or worse, after delivery. Consequently, one must test early and often." [27].

It is important that what has been implemented reflects what was stated in the software requirements. From the discussions in Chapter 3, one could see that the main non-functional requirements were related to:

- Maintainability
- Reliability
- Power consumption
- Memory constraints

To investigate the implementation, tests for each of these non-functional requirements were performed. It is important to note that the purpose of testing is not to show that the application is satisfactory, but to vigorously determine where the application is not satisfactory" [27].

7.2 Verifying FreeRTOS and Tracelizer

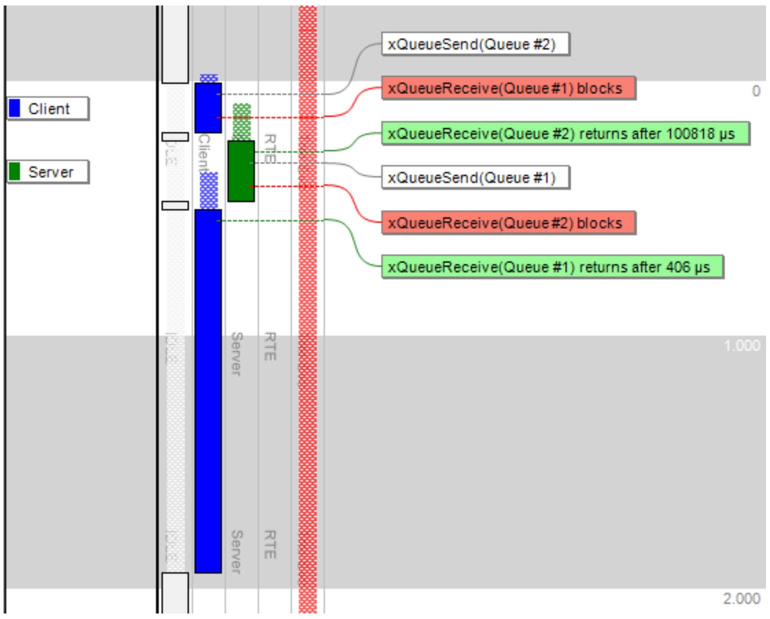
Maintainability is considered one of the foremost important non-functional requirements for the NUTS OBC. As means to facilitate a maintainable architecture, it was decided that concurrency as well as debugging tools should be an inherent part of the architecture. FreeRTOS was chosen as OS and Percepio Trace as the OS debugging tool. To investigate the implementation and installment of these, a test application was made.

7.2.1 Method

The application consists of two threads communicating through two queues. The implementation toggled a LED, as well as printing status massages to debug USART connected to a desktop to give visible signs of workflow. To further investigate whether the FreeRTOS implementation actually allowed multiple threads to run, as well as to evaluate the functioning of the trace debugging tool, the trace was recorded and transferred to the desktop for inspection.

7.2.2 Result

A part of the resulting trace can be seen in Figure 7.1. The periodic blinking of the LED,



xQueueSend(Queue #2)
xQueueReceive(Queue #1) blocks
xQueueReceive(Queue #2) returns after 100818 μs
xQueueSend(Queue #1)
xQueueReceive(Queue #2) blocks
xQueueReceive(Queue #1) returns after 406 μs

Figure 7.1: Trace from FreeRTOS test application

as well as the debug messages printed to screen could be observed as the code was running on the microcontroller. This proved that the implementation and installment exhibited the characteristics that were expected from the test setup. This should give some confidence in the installment of FreeRTOS and Percepio Trace, though the test does not in any way prove that they are free from defects, it does however prove that at a subset of FreeRTOS and Percepio Trace functionlities, (i.e. those tested in the test setup) is supported by the application.

7.3 Memory Footprint

Jahren reports encountering issues due to running out of internal memory of the UC3A3 while testing the former CSP implementation [14]. Though his test setup was not optimized for size, it is not unlikely that memory constraints may be challenging in this implementation as well. An inquiry was therefor made into the memory usage of the architecture proposed in this thesis.

7.3.1 Method

The GCC build process reports the memory use after building the project, this can be used to obtain the ROM needed for the program code. The data memory is also reported and can be used to investigate further into the memory usage.

7.3.2 Results

The memory usage reported from the build process of the OBC can be seen in Table 7.1.

Communication form	Program Memory Usage	Data Memory Usage
OBC	110176 bytes (5.3%)	2539404 bytes (64.6%)

Table 7.1: Memory usage for the OBC application

As all task stack sizes and queue buffer lengths can be configured by the application designer, the memory footprint can be highly optimizable. The reason for the rather large program memory usage is the fact that the total FreeRTOS heap size is configured to use half of the available memory (198KB).

7.4 Investigating CSP Functionality

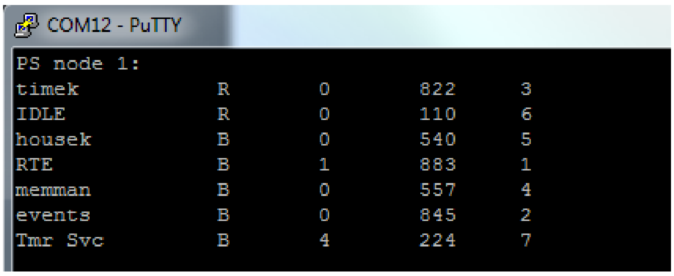
To investigate the functioning of CSP, multiple tests were implemented. All of the following tests were performed using the skeleton code for the OBC. The test setup thus consisted of an ARM M7 microcontroller (ATSAMV71-Xult, situated on a SAMv71 Xplained Ultra Evaluation Kit) running the currently last official release of FreeRTOS (version 8.2.3) with a system frequency (CPU and bus) of 75 MHZ and 1 ms RTOS ticks. The setup was tested using preemptive scheduling, with stack overflow checking enabled. The application was the OBC skeleton code, consisting of four user threads communicating through CSP. The following procedures were tested by sending a request to the OBC CSP node from the housekeeper task:

- Send a single ping/echo packet
- Request process list
- Request amount of free memory
- Request number of free buffer elements
- Reboot subsystem
- Request subsystem uptime

Each of these tests are now presented in more detail in the following sections.

7.4.1 Request Process List Test

CSP implements a specific function for requesting a list of running processes on any microcontroller or computer connected to the CSP network. As there is an undergoing change from using I2C to using CAN for the internal satellite bus, the request function was routed for the OBC (the same microcontroller) effectively returning information on the threads running on the same microcontroller. However, when the CAN bus drivers have been implemented, the call would look exactly the same from the application programmer viewpoint. The routing of the request to another microcontroller and back would be handled by the CSP middleware.



```
PS node 1:
timek          R       0       822      3
IDLE           R       0       110      6
housek         B       0       540      5
RTE            B       1       883      1
memman         B       0       557      4
events         B       0       845      2
Tmr Svc        B       4       224      7
```

Figure 7.2: Task list as reported from CSP

The retrieved information can be seen in Figure 7.2, while a small trace obtained by the Percepio trace tool for the same application can be seen in Figure 7.3.

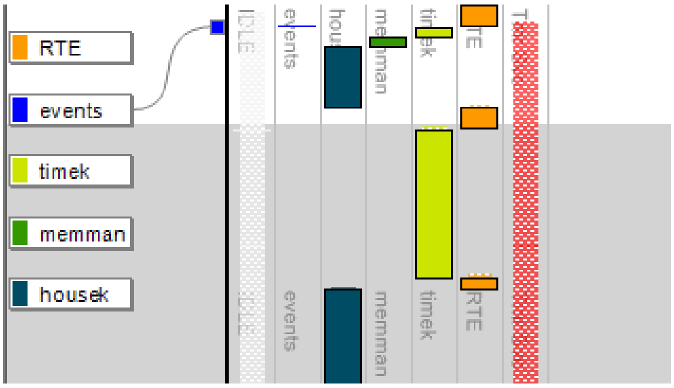


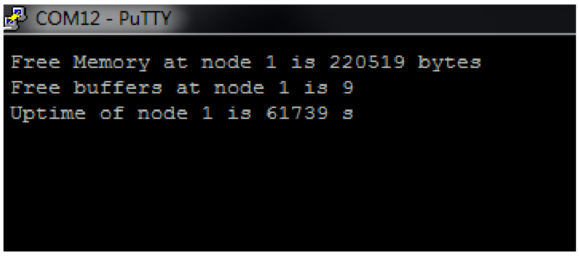
Figure 7.3: Trace showing the existence of FreeRTOS tasks

CSP Memory Leak When Receiving csp_ps

The csp_read() function allocates buffers for packets, the memory is then handed over to the caller through the returned pointer. This means that the caller is responsible of freeing the buffer. This was not handled correctly in csp_ps() implemented in csp_services.c. There is a clean up section at the bottom of the function. However this only frees packets if their pointers are non-NULL. As the last call to csp_read() always returns a NULL pointer when there is no more to receive, the packet buffer fails the if(packet != NULL) test and is never freed. A workaround was therefor constructed and the issue reported to GomSpace. This workaround has now been implemented in the official GomSpace repository as well.

7.4.2 Request Amount of Free Memory Test

The initial test failed due to an assert forced by the vApplicationMallocFailedHook. Upon further investigation on how exactly the allocation failed, it was discovered that the amount of free memory is determined through trial and error with how much memory malloc can allocate. The procedure starts by trying to allocate 10 KB, when this fails, it tries to allocate half of that (5 KB), and so on. It basically does a binary search by trial and error through repeated calls to dynamically allocate different sized memory blocks. As the OBC application implements freeRTOS hooks for detection of malloc fails and stack overflows, this procedure resulted in a forced assert from the vApplicationMallocFailedHook. However when removing the forced assertion from the vApplicationMallocFailedHook, the procedure successfully returned the available amount of memory, as shown in Figure 7.4



```
Free Memory at node 1 is 220519 bytes
Free buffers at node 1 is 9
Uptime of node 1 is 61739 s
```

Figure 7.4: CSP Memory and uptime request result

7.4.3 Request Number of Free Buffer Elements

When CSP buffer system is initiated with 10 packets, the call to buffer left will return 9, due to the one buffer in use for the transaction. When the buffer system is initiated with 20 packets, the reported number of free buffers change to 19. To further verify the buffer request procedure, it was combined with the erroneous implementation of process list request discussed above. The procedure for obtaining the number of free buffers then successfully reported the decreasing number of buffers available until there were none left and no more requests could be sent, the procedure then reported network error.

7.5 Analysing CSP Timing

To investigate the functioning of the CSP middleware as well as to estimate the additional overhead that comes from using CSP for thread communication, a simple test system was set up and analysed. The test system consists of three threads; two user threads (one client and one server) and the CSP router task (RTE). To test the CPU overhead the client constructs and sends a packet with a payload of 100 bytes to the server. The server does any necessary check for the integrity of the package before sending it back to the client. The client then verifies the content of the package and records the time the transaction took. The average execution time is obtained by calculating the average over 100 iterations. Each iteration is measured by reading the ARM SYSTICK timer, effectively yielding how many CPU cycles were spent on the procedure.

7.5.1 Method

The test setup consisted of an ARM M7 microcontroller (ATSAMV71-Xult, situated on a SAMv71 Xplained Ultra Evaluation Kit) running the currently last official release of FreeRTOS (version 8.2.3) with a system frequency (CPU and bus) of 75 MHZ and 1 ms RTOS ticks. The setup was tested using preemptive scheduling, with stack overflow checking enabled. The test was performed for various optional CSP features enabled such as the Reliable Data Protocol (RDP), check-sums (CRC32), encryption (XTEA) and authentication (HMAC-SHA1). The trace from one of these tests can be seen in Figure 7.5

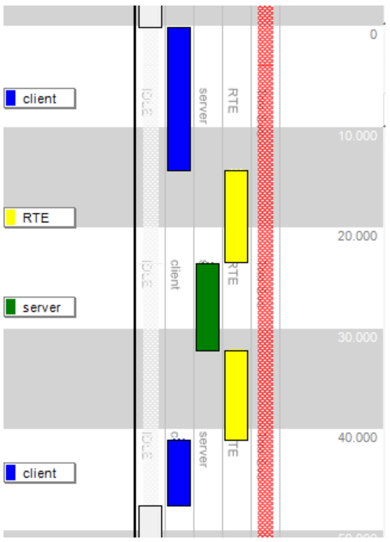


Figure 7.5: Trace from CSP ping test application (NB: not used for the actual timing, as the trace tool is intrusive)

First a small application using only FreeRTOS queues to communicate directly from the client task to the server and back was tested. The application was compiled with GCC and uploaded to the microcontroller using an Atmel-Ice debugger. The application ran for 100 iterations and the average execution time was calculated from these 100 samples. This was done to be able to compare the CSP latency, to that of using FreeRTOS queues directly.

7.5.2 Results and Discussion

The result can be seen in Table 7.2.

Enabled Features	**Duration [ticks]**	**Duration at 75 MHz [ms]**
No features	30 231	0.403

Table 7.2: Latency for for native FreeRTOS queue test application

Communication form	**Program Memory Usage**	**Data Memory Usage**
CSP sockets	89440 bytes (4.3%)	62040 bytes (15.8%)

Table 7.3: Memory usage for CSP socket test application

After testing the regular FreeRTOS queues, a small application running CSP was tested with various features enabled. The memory usage can be seen in table 7.3, while the duration for the setup and communication of each iteration can be seen in table 7.4. For

Enabled Features	Duration [ticks]	Duration at 75 MHz [ms]
No features	134 586	1.795
CRC32	149 374	1.992
RDP	297 681	3.969
HMAC-SHA1	251 271	3.350
XTEA	291 829	3.891
RDP + CRC32	316 042	4.214
ALL	785 040	10.467

Table 7.4: Latency for CSP test application

connection-less communication with no CRC or encryption there is a 345.19% workload increase, which at 75 MHz corresponds to an additional 1.392 milliseconds of execution time. If connection oriented communication with resending of lost packages is being used, the overhead is an additional 163095 cycles compared to the connection-less option. This corresponds to about 2.174 ms extra computation time for the round-trip when running at 75 MHz.

As a comparison; The Polysat project analysed the latency of UDP for inter-process communication in their CubeSat application. They were running Linux at 400MHz on a AT91SAM9G20 microcontroller and reported average time of 7.2 ms one way [56]. If one makes the assumption there are no wait-states used for this procedure, the 7.2 ms oneway-trip at 400MHz would correspond to 76.8 ms round-trip at 75 MHz. Compared to using FreeRTOS and CSP that is more than 40 times as long!

7.6 Analysing Power Consumption

As was discussed in Chapter 3, it is imperative that the power consumption of the OBC is sustainable by the satellite power budget. To investigate the power consumption of the OBC SAMV71 microcontroller, the skeleton code mimicking the activity of the OBC was uploaded and the power consumption of the application measured.

7.6.1 Method

The power consumption of the SAMV71 microcontroller was obtained by measuring the power consumption of the SAMV71 microcontroller situated on an Atmel SAM V71 Xplained Ultra kit, while it ran the OBC skeleton example code. Measurements were made for different power optimization techniques as to inspect the effect of each of the techniques. The power consumption was measured using the Atmel Power Debugger and plotted through the Atmel Studio extension Data Visualizer.

7.6.2 Results and Discussion

The unoptimized power consumption of the OBC microcontroller was measured to be around 73 mA at 3.3V, as can be seen in Figure 7.6

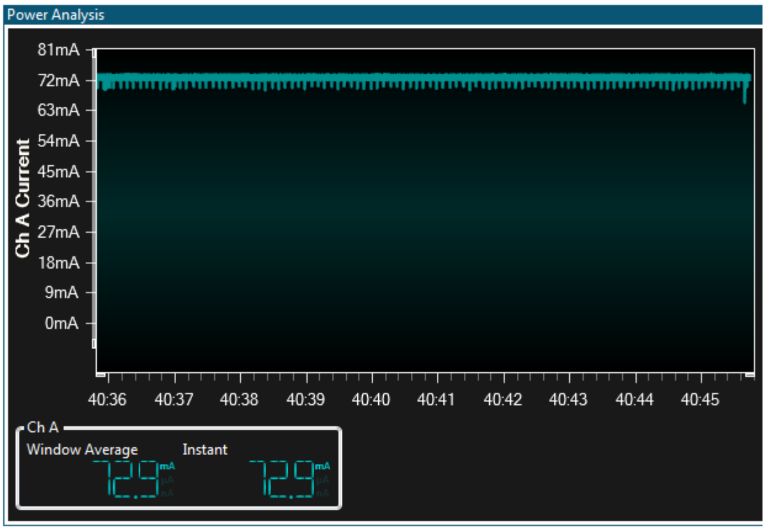


Figure 7.6: Power Consumption of the OBC MCU at 300MHz

The exact execution speed for OBC MCUs will be decided through later tests and analysis of power consumption, heat dissipation and throughput needed. As the SAMV71 processor is the only part of the microcontroller (or satellite for that matter) which supports a 300 MHz frequency it may be induced that it may not be efficient to run at this frequency as wait states must be included for all communication. This includes memory access for the internal RAM as well as the peripherals and external memory. USART and CAN communications may also result in unnecessary high power consumption as their baudrates will be much slower than the clock of the CPU. Because of this, a slower speed is chosen for the test setup. The exact speed should however be decided through extensive testing and analyses when the system is more mature. The clock frequency for the SAMV71 OBC was in this setup divided by 4 to obtain the frequency of 75 MHz, a frequency more relatable to the 66 MHz of the UC3C OBC. This reduction of frequency reduced the power consumption by about half, as can be seen in Figure 7.7.

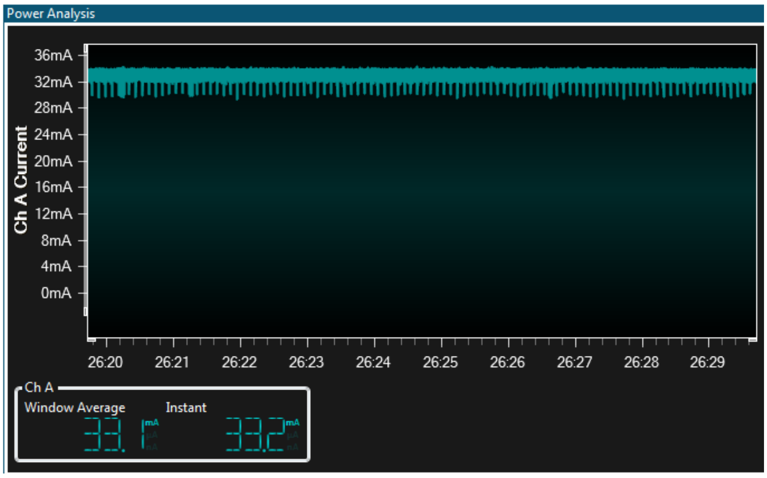


Figure 7.7: Power Consumption of the OBC MCU at 75MHz

By enabling sleep, the power consumption could be further reduced by about 31% as can be seen in Figure 7.8.

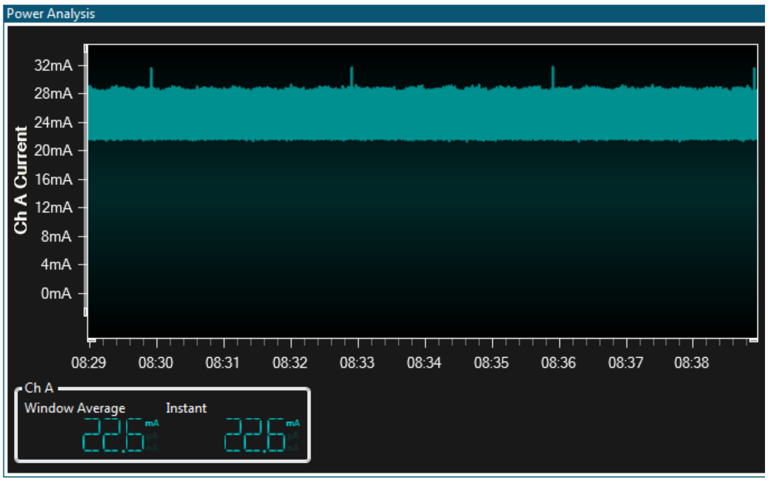


Figure 7.8: Power Consumption with Active Sleep Mode

As the SYSTICK timer of the ARM Processor is used to generate the RTOS tick, the system would only support active sleep mode. To support deeper sleep modes Tickless Idle had to be implemented using an asynchronous timer to keep track on time while the CPU sleeps. This reduced the power consumption substantially, as can be seen in Figure 7.9. Connection oriented communication is implemented in CSP in such a way that the router task repeatedly wakes up to check whether any of the connections have timed out. This causes the router task to wake up every 100 ms (the default timeout period) to check

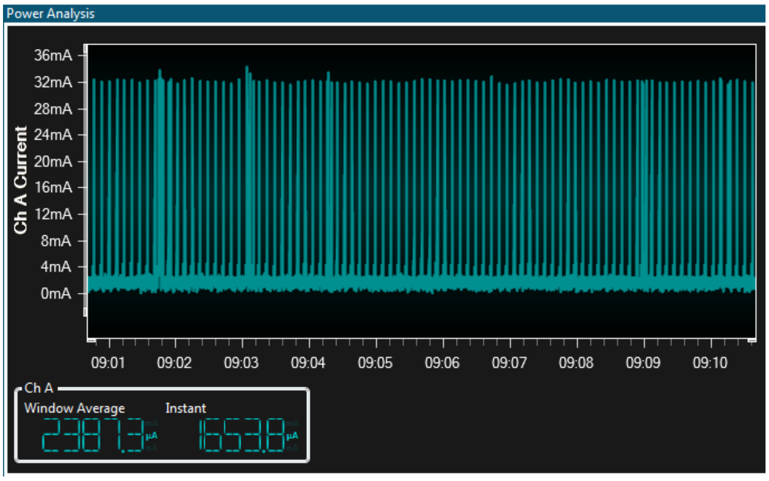


Figure 7.9: Power Consumption with Wait Mode and Tickless Idle

for timeouts even when there are no active connections. By extending the timeout of the connections to 10 seconds, an even smaller power consumption can be obtained, as seen in Figure 7.10. However as this may make the system more unstable and only account for the saving of less than one milliampere, the timeout was reset to 100 ms after the measurements were made.

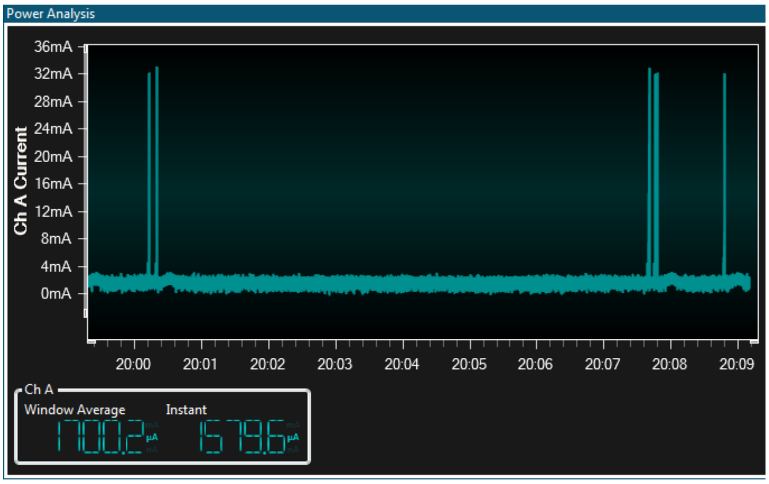


Figure 7.10: Power Consumption with Wait Mode, Tickless Idle and extended connection timeout

The deepest sleep mode supported by the SAMV71 is the backup mode, however in this mode the internal SRAM is unpowered. Unless the internal backup RAM is used, exiting backup mode will be similar to a soft reset. The wake-up time from this sleep mode can be up to 2 ms [?], and any states not stored and retrieved form non-volatile memory will be reset. Using this mode was attempted without the use of intern backup RAM and can be seen in Figure 7.11. The constant reseting of the MCU proves power inefficient compared to using the wait mode.

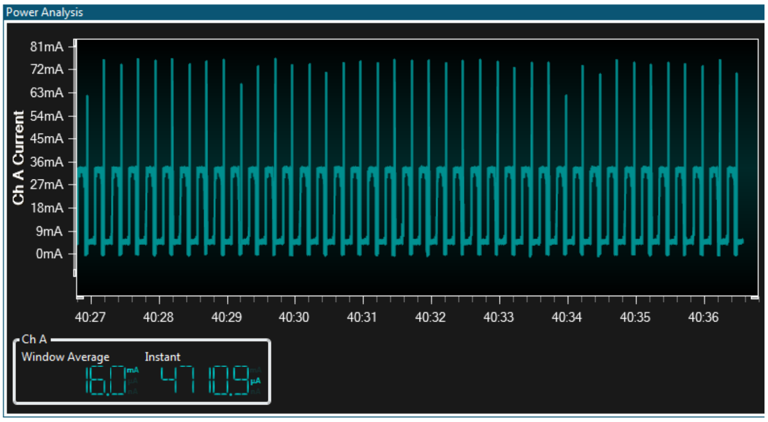


Figure 7.11: Inefficient use of backup sleep mode causes constant MCU soft reset

The tickless idle sleep using wait mode with a CPU frequency of 75MHz as seen in Figure 7.12 was also compared to that of the same setup running at 300 MHz, as can be seen in Figure 7.13. It can then be seen at with this setup the system running at 75 MHz was indeed more power efficient than that running at 300 MHz, despite the shorter bursts of workload that is needed at the faster execution speed.

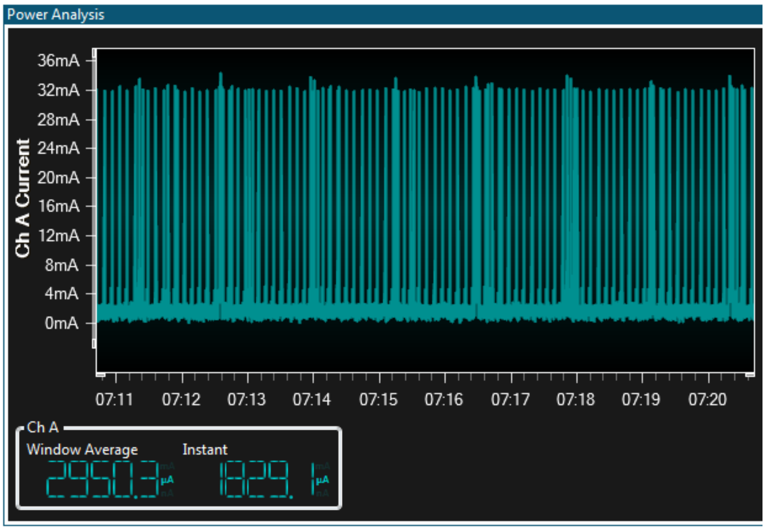


Figure 7.12: Power consumption with CPU clocked at 75 MHz

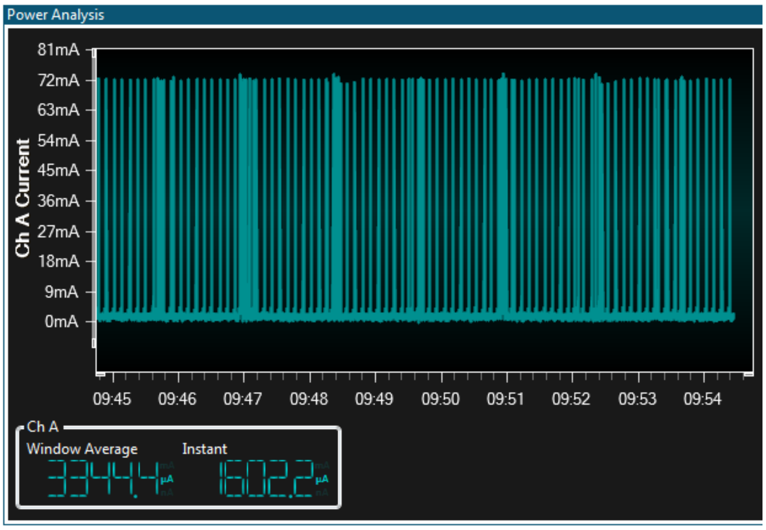


Figure 7.13: Power consumption with CPU clocked at 300 MHz

7.6.3 Power Consumption Summary

In this section different techniques for power optimization were presented and their resulting power consumptions were plotted and compared. It was discovered that for the current setup, the average power consumption was reduced when running at 75 MHz compared to running at 300 MHz. It was also discovered that the implementation of connection time out may not be ideal, however as the energy savings for implementing a fix for this is unlikely to be more than 1 mA, it is at this time not deemed to be a cost efficient improvement. The results obtained in this chapter are summarised in Table 7.5

Method	Energy consumption	Reduction from previous method (%)	Reduction from before optimization (%)
300 MHz	72.9 mA	0	0
75 MHz	33.1 mA	54.6	54.6
Active Sleep	22.6 mA	31.7	69.0
Tickless Idle	2387.3 μA	89.4	96.8
Extended timeout	1700.2 μA	28.8	97.7

Table 7.5: Power optimization techniques and improvements

7.7 Summary and Discussion

In this chapter, different aspects of the architecture implementation has been tested and analysed. Many functions were observed to behave as expected, however some bugs and undesired features in the CSP code were encountered as well. One of the heavier arguments of using CSP in the core of the architecture, was the fact that it had flight heritage with multiple CubeSats, and was thus considered safer than having a student made implementation. The testing of the CSP functions for retrieving the list of threads running on a CSP node, did however unearth a bug in the software that resulted in a buffer leak and network failure. The fact that this bug existed proves that despite the library's flight heritage, one cannot assume that it is bug free. The successful use of CSP in other satellite missions, does suggest that most parts of CSP are tested and considered safe, but the functioning of the complete system should not be assumed.

As mentioned in the beginning of this chapter it is important to remember that tests cannot prove that an application is free of defects, as proofs of correctness can. Testing can only show the presence of defects." [27]. Despite the tests proving the framework functional, they do not prove that it is defect-free. Ideally, the testing of code should be performed by people other than those who developed it. When an engineer develops code, he forms a vision of what the code is meant to do, and, at the same time, he develops typical circumstances in which the code must execute. It is safe to assume that the code shows few problems in those particular circumstances. Consciously or not, these circumstances form the developer's test cases. Thus, when an individual tests his own code he tends to hide the very defects that need uncovering." [27]. As no bugs were detected related to the installment of FreeRTOS, Percepio Trace, CSP, and power optimization techniques, all of which this author was responsible for, the framework of the software architecture is considered functional, but not necessarily optimal. It is recommended that the testing of the different architectural features are continued through the next iteration in order to try to uncover any defects as early as possible.

Chapter 8

Reflections

In this chapter the work done in this thesis is reviewed and discussed.

8.1 Discussion

Through this thesis, a software architecture for the NUTS OBC has been proposed. To arrive at the software architecture, the software requirements for the OBC were analysed, features of good software design were discussed and an architecture as well as its implementation was suggested. Details of the intended design were further elaborated, to outline how they could be realised. The implementation of the OBC was begun through the construction of the OS and communication frameworks and debugging tools. The setup was then tested to verify the implementation, as well as to try and uncover any defects. The results of these tests were then presented.

Discussions have been made under each chapter, where relevant data has been presented and compared. The OBC is entrusted with many tasks and is dependent on many factors. This makes for a complex system that is not easily comprehensible and the author therefor believed it to be necessary to structure the thesis so that individual discussions are made for each problem, rather than one huge discussion section at the end of the thesis.

8.1.1 Review of the initial Problem and the Contributions Made

As there are multiple development steps between the implementation and the mission goals that were to be satisfied (where each stage is based on the one before), it may be advantageous to review the result in light of the initial problem.

In this thesis, the OBC hardware, and an OBC functional and non-functional requirement specification were put in, and the assignment was to identify software requirements for the system as well as to research and design a software architecture for the NUTS OBC system, with particular focus on developing a system that allows isolation of work tasks while maintaining easy integrability to facilitate further development of the NUTS Satellite computer system.

The output of the work in this thesis has been an implementation of a service-oriented software architecture, including middleware for communication handling and trace debugging. As steps towards this architecture both a SRS and a SDD were created in accordance with IEEE recommended practises. The porting of CSP from a library that had to be reconfigured and built for a target specific application, to that of being a more portable, fully featured, reconfigurable library that is configurable through regular precompiler defines in each application program, is considered to be a major contribution of this thesis. CSP is used on all nodes connected to the satellite bus, including both OBCs, the ADCS and the Payload camera module, in addition to also being used on the ground station. Having a functional, easy reconfigurable, fully featured and extensible port for CSP on each of these satellite submodules is considered important for the successful integration of the satellite system. A list of the most important contributions can be seen below.

- Port of CSP with all features enabled supported for all NUTS microcontrollers
- Software Architecture discussion and design
- SRS
- SDD
- FreeRTOS project with Percepio Trace library implementation
- Sleep and tickless idle implementation
- Setup of repositories for application and CSP with satellite-wide integration in mind
- CSP timing analysis for inter thread communication.
- Port of Marcedellas RSP to the UC3C microcontroller

Testing was done both to verify some functions as well as to uncover defects in the implementation. Many CSP functions were tested functional, while some undesirable features of the implementation were discovered as well. A bug resulting in a buffer leak and network failure as well as a power-inefficient implementation of connection-oriented communication was unearthed and discussed.

8.1.2 Further Work for the OBC Software

As the goal for this thesis was to present a software architecture for the NUTS OBC, no drivers except for the ones needed for the architectural frameworks have been implemented. If the contributions in this these are accepted into the NUTS project, the next natural steps would be to start the development and implementation of the submodules described in the detailed design section of this thesis. Further specifying of interfaces and service contracts (i.e. deciding upon he design of each data structure that is passed) should be done before any implementation is begun as to facilitate later integration. It is also suggested that as soon as a prototype for the new CAN-based backplane is constructed, CAN drivers should be made and connected to the CSP library to start the testing of the communication between the satellite modules to discover any defects as early as possible.

8.2 Conclusion

Based on the results from the testing, it is believed that a service-oriented architecture, satisfying the constraints imposed on the NUTS OBC software can be realised through the use of CSP for internal as well as external communication. Such an architecture is considered to be ideal for the development of a university CubeSat such as NUTS, as SOA enables the independent development of services through standardised interfaces. This can greatly ease system integration as well as the implementation and rerouting of redundant services. The computational overhead as well as the added latency on inter-thread communication is analysed and the solution considered cost efficient. The architecture proposed in this thesis is therefor recommended for further implementation and use in the NUTS project.

Bibliography

[1] Daniel Siewiorek and Robert Swarz. *Reliable Computer Systems: Design and Evaluatuion*. Digital Press, 1992.

[2] Atmel Corp. Atmel at03289: Sam4l low power design with freertos. http://www.atmel.com/Images/Atmel-42204-SAM4L-Low-Power-Design-with-FreeRTOS_AP-Note_AT03289.pdf, 2013.

[3] The Norwegian Student Satellite Program ansat. http://andoyaspace.no/?page_id=254. Accessed: 2015-28-10.

[4] Nasjonalt studentsatellittprogram narom. https://www.narom.no/artikkel.php?aid=2&bid=56&oid=813. Accessed: 2015-28-10.

[5] Space SkyRocket gunters space page. http://space.skyrocket.de/doc_sdat/hincube.htm. Accessed: 2015-28-10.

[6] HiNCubesat is it up there? http://hincube.cubesat.no/wp. Accessed: 2015-28-10.

[7] B. Randell, P. Lee, and P. C. Treleaven. Reliability issues in computing system design. *ACM Comput. Surv.*, 10(2):123–165, June 1978.

[8] Wellings Andy Burns, Alan. *Real-Time Systems and Programming Languages*. Pearson Education Limited, 2009, 4. edition, 2009.

[9] Mayeul Marcadella. Improvement in the reliability of a bi-processing unit satellite subject to radiation-induced bit-flips. 2014.

[10] Magnus Haglund Arnesen and Christian Elias Kiær. Mission event planning & error-recovery for cubesat applications. 2014.

[11] Space Radiation Effects on Electronic Components in Low-Earth Orbit nasa. http://llis.nasa.gov/lesson/824. Accessed: 2015-23-11.

[12] Radiation effects on space electronics uio. http://www.uio.no/studier/emner/matnat/fys/FYS4220/h11/undervisningsmateriale/forelesninger-vhdl/Radiation%20effects%20on%20space%20electronics.pdf. Accessed: 2015-24-11.

[13] Cosmic Rays cosmic rays. http://www.srl.caltech.edu/personnel/dick/cos_encyc.html, 1996. Accessed: 2015-16-12.

[14] Erlend Riis Jahren. Design and implementation of a reliable transport layer protocol for nuts. 2015.

[15] Kjell Arne Odegaard. Error detection and correction for low-cost nano satellites. 2013.

[16] Steven McConnell. *Code Complete*. Microsoft Press, 2004, 2. edition, 2004.

[17] Magne Normann. Hardware review of an on board controller for a cubesat. 2015.

[18] Daniel L Dvorak et al. Nasa study on flight software complexity. 2009.

[19] Wiley J Larson and James Richard Wertz. Space mission analysis and design. Technical report, Microcosm, Inc., Torrance, CA (US), 1992.

[20] Andris Slavinskis et al. Estcube-1 in-orbit experience and lessons learned. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=7286959&tag=1, 2015.

[21] David Gerhardt et al. Rapid results: The gomx-3 cubesat path to orbit. 2016.

[22] Greg Manyak. Fault tolerant and flexible cubesat software architecture. http://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1600&context=theses, 2011.

[23] Ralph Rowland Young. *The requirements engineering handbook*. Artech House, 2004.

[24] Atmel. Sam v71, smart arm-based flash mcu. http://www.atmel.com/images/Atmel-44003-32-bit-Cortex-M7-Microcontroller-SAM-V71Q-SAM-V71N-SAM-V71J_Datasheet.pdf, 2016.

[25] Atmel Corp. Atmel avr uc3c 32-bit flash microcontrollers. http://www.atmel.com/Images/32187B_AVR%20UC3C_E_US-0912_LR.pdf, 2012.

[26] L Jane Hansen and Robert W Hosken. Spacecraft computer systems. In Wiley J Larson and James Richard Wertz, editors, *Space mission analysis and design*, chapter 16, pages 645–684. Microcosm, Inc., Torrance, CA (US), 1992.

[27] Eric J Braude. *Software engineering: an object-oriented perspective*. John Wiley & Sons, Inc., 2000.

[28] Ian Sommerville. Software engineering. international computer science series. *ed: Addison Wesley*, 2004.

[29] Embedded linux/microcontroller project. http://www.uclinux.org/ports/. Accessed: 2016-24-02.

[30] Linux support for avr32 uc3a. https://daim.idi.ntnu.no/masteroppgaver/004/4637/masteroppgave.pdf. Accessed: 2016-07-04.

[31] Philip Koopman. *Better embedded system software*. Drumnadrochit Education, 2010.

[32] Freertos faq - memory usage, boot times & context switch times. http://www.freertos.org/FAQMem.html. Accessed: 2016-01-04.

[33] Atmel Corp. At04056: Getting started with freertos on atmel sam flash mcus. http://www.atmel.com/Images/Atmel-42382-Getting-Started-with-FreeRTOS-on-Atmel-SAM-Flash-MCUs_ApplicationNote_AT04056.pdf.

[34] Tracealyzer for freertos. http://percepio.com/docs/FreeRTOS/manual/index.html#Tracealyzer_for_FreeRTOS.

[35] Improved J-Link trace streaming, percepio. http://percepio.com/2015/10/27/improved-j-link-trace-streaming/.

[36] GomSpace CubeSat Space Protocol(CSP) gomspace. http://www.gomspace.com/documents/GS-CSP-1.1.pdf. Accessed: 2015-22-11.

[37] GomSpace libcsp github. https://github.com/GomSpace/libcsp. Accessed: 2015-22-11.

[38] Helmut Petritsch. Service-oriented architecture (soa) vs. component based architecture. *Vienna University of Technology, Vienna*, 2006.

[39] Core Flight Executive nasa. http://opensource.gsfc.nasa.gov/projects/cfe/index.php. Accessed: 2016-08-03.

[40] Space avionics open interface architecture. http://savoir.estec.esa.int/.

[41] Generationone onboard software. http://www.brightascension.com/products/generation1/.

[42] Operating system abstraction layer. https://github.com/nasa/osal. Accessed: 2016-12-03.

[43] I Stanley Weiss and Michael S Williams. Requirements definition. In Wiley J Larson and James Richard Wertz, editors, *Space mission analysis and design*, chapter 4, pages 73–94. Microcosm, Inc., Torrance, CA (US), 1992.

[44] Anatomy of Linux journaling file systems, ibm. http://www.ibm.com/developerworks/library/l-journaling-filesystems/index.html. Accessed: 2016-17-02.

[45] Crash consistency: Fsck and journaling. http://pages.cs.wisc.edu/~remzi/OSTEP/file-journaling.pdf. Accessed: 2016-17-02.

[46] Reliance Edge Fail-Safe File System freertos. http://www.freertos.org/FreeRTOS-Plus/Fail_Safe_File_System/Reliance_Edge_Fail_Safe_File_System.shtml. Accessed: 2016-01-02.

[47] Freertos version 9. http://www.freertos.org/FreeRTOS-V9.html. Accessed: 2016-01-02.

[48] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010.

[49] Diaa Jadaan. Memory management and error handling in freertos for a cubesat project. 2013.

[50] Richard Barry. *Using the FreeRTOS real time kernel: a practical guide*. Real Time Engineers, 2010.

[51] The freertos reference manual. http://www.freertos.org/a00110.html. Accessed: 2016-01-03.

[52] Memory protection unit (mpu) support. http://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html. Accessed: 2016-01-03.

[53] Percepio faq. http://percepio.com/tz/faq/. Accessed: 2016-20-02.

[54] Component-based software engineering wikipedia. https://en.wikipedia.org/wiki/Component-based_software_engineering. Accessed: 2016-08-03.

[55] Martin Fowler. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland*, 1997.

[56] Fault tolerant and flexible cubesat software architecture. http://digitalcommons.calpoly.edu/theses/550/. Accessed: 2016-10-06.

Appendices

Appendix A

OBC Requirement Specification

A.1 OBC Functional Requirements

A.1.1 Satellite Initialization

R00-OBC-INI-000	INI=INITIALIZATION. The OBC must run an initiation of the satellite upon deployment
R00-OBC-INI-001	The OBC must issue an antenna deployment signal
R00-OBC-INI-002	The OBC must start beacon transmission from the radios
R00-OBC-INI-003	The OBC must issue a de-tumble command to the ADCS upon initial deployment
R00-OBC-INI-004	The OBC must enter normal operation mode

Table A.1: OBC initialization requirements

A.1.2 Satellite Housekeeping

R04-OBC-HKP-000	HKP=HOUSEKEEPING. The OBC must monitor and maintain the health of the satellite.
R00-OBC-HKP-001	The OBC must periodically check and act upon its own integrity.
R00-OBC-HKP-002	The OBC must periodically check and act upon the health of other modules.
R04-OBC-HKP-003	The OBC must keep a log of the satellite's health

Table A.2: OBC Housekeeping requirements

A.1.3 OBC Executables

R01-OBC-EXE-000	EXE=EXECUTABLES. The OBC must be able to execute commands on demand from ground station
R01-OBC-EXE-CMD-000	CMD=COMMANDS. The OBC must support a set of commands.
R00-OBC-EXE-CMD-001	The OBC must be able to perform a full, and/or partial reset of the satellite.
R01-OBC-EXE-TIM-000	The OBC must be able to schedule commands to be executed later.
R01-OBC-EXE-TIM-001	The OBC must keep track of time.
R01-OBC-EXE-TIM-002	The OBC must be able to set alarms to schedule future executions.
R00-OBC-EXE-PRG-000	PRG= PROGRAMMABILITY. The OBC must be able to reprogram itself as well as other modules.
R00-OBC-EXE-PRG-002	The OBC be able to receive and store at least one additional boot image from ground station

Table A.3: OBC general requirements

A.1.4 Satellite Executables

R04-OBC-EXT-000	EXT=EXTERNAL. The OBC must interface with the other modules
R00-OBC-EXT-PRG-000	PRG= PROGRAMMABILITY. The OBC must be able to reprogram itself as well as other modules.
R00-OBC-EXT-PRG-002	The OBC be able to receive and store at least one additional boot image from ground station
R04-OBC-EXT-RAD-000	The OBC must interface with the on-board radio
R03-OBC-EXT-ADC-000	ADC=Altitude Determination and Control System. The OBC must be able to configure and control the ADCS.
R03-OBC-EXT-ADC-001	Module reset, module power down.
R03-OBC-EXT-ADC-002	Start de-tumbling.
R03-OBC-EXT-ADC-003	Stop de-tumbling.
R03-OBC-EXT-ADC-004	Point in a given direction.
R07-OBC-EXT-CAM-000	CAM=CAMERA PAYLOAD. The OBC must be able to configure and control the camera.
R07-OBC-EXT-CAM-001	Module reset, module power down/up, request statistics.
R07-OBC-EXT-CAM-002	Schedule image capturing
R07-OBC-EXT-CAM-003	Retrieve images, and send to ground station
R08-OBC-EXT-CAM-004	Retrieve thumbnail images and send to ground station
R08-OBC-EXT-CAM-005	Delete images
R07-OBC-EXT-CAM-006	Set; gain, frame rate, frame size, and exposure time.

Table A.4: OBC External interface requirements

A.2 OBC Non-Functional Requirements

ID	Description
R00-OBC-NON-DUR-000	DUR=DURATION Mission operation is 3 months.
R00-OBC-NON-AVA-000	AVA=AVAILABILITY The mission capable rate should be ass high as possible, and should not be less than 50%
R00-OBC-NON-AVA-001	The OBC must be able to handle multiple concurrent requests.
R00-OBC-NON-SUR-000	SURVIVABILITY The OBC must survive its natural environment (space)
R00-OBC-NON-REL-000	REL=RELIABILITY The OBC mean time to failure (MTTF) should be greater than the duration of the space mission.
R00-OBC-NON-REL-001	No single failing task or component shall end the satellite mission.
R00-OBC-NON-REL-002	The OBC must be able to recover from transient errors such as SEL and SEU, no matter where or when they might occur.
R00-OBC-NON-REL-003	Execution of less-important tasks shall not affect the timeliness of higher-prioritised tasks.
R00-OBC-NON-TES-000	TES=TESTABILITY A formal test procedure with measurable results must be enunciated and performed.
R00-OBC-NON-MAI-000	MAI=MAINTAINABILITY Code and documentation must be made easy to understand
R00-OBC-NON-MAI-001	All user written code must be documented in a coherent manner
R00-OBC-NON-MAI-002	Version control shall be used for the code repository
R00-OBC-NON-MAI-003	Software architecture must support tools for debugging

Table A.5: OBC non-functional requirements

R00-OBC-PRG-000	PRG= PROGRAMMABILITY. The OBC must be able to reprogram itself as well as other modules.
R00-OBC-PRG-002	The OBC be able to receive and store at least one additional boot image from ground station

Table A.6: OBC reprogramming requirements

A.3 Document Version History

Revision	Date	Author	Description
1.1	01.03.2016	Magne Normann	Included requirements of reprogramming, refined non-functional requirements.
1.0	05.10.2015	Magne Normann	Initial Document Release

Table A.7: OBC reprogramming requirements

Appendix B

OBC Service Contracts

Interface of the Timekeeper
The Timekeeper is to be implemented with one blocking listening socket TIMEKEEPER_SOCKET_RX that shall receive and react to calls concerning time and alarms. The calls it shall respond to can be seen in the table below.

Call	**Parameters**	**Reaction**
PING	RSVP_SOCKET	Send PONG to RSVP_SOCKET
PONG	NONE	Register response
TIME_GET	RSVP_SOCKET	Send current time to the RSVP_SOCKET
TIME_SET	time_t time	Update the current time to the one received
ALARM_SET	time_t time, action_t action	Add alarm to execute ”action” at ”time”
ALARM_DEL_ALL	NONE	Delete all alarms

Interface for the Housekeeper
The housekeeper module is responsible for managing the health of the satellite. It is to be implemented with one blocking listening socket HOUSEKEEPER_SOCKET_RX that shall receive and react to calls concerning satellite health and telemetry. The calls it shall respond to can be seen in the table below.

Call	Parameters	Reaction
PING	RSVP_SOCKET	Send PONG to RSVP_SOCKET
PONG	NONE	Register response
RUN_HEALTH_TEST	NONE	Run a full satellite test
ERROR_REPORT	error_t error	Do necessary actions, such as logging the error and saving the state, before resetting the module
SYNC_TIME	time_t time	Synchronize all RTCs in the satellite with the time given, if no time is given synchronize using the timekeeper time.
GET_SAT_STATE	RSVP_SOCKET	Retrieve the current satellite state and send it to RSVP_SOCKET.

Interface for the Logger
The logger is the software service responsible for maintaining the log. It is implemented as a single thread with one listening socket, LOGGER_SOCKET_RX. The logger will be the only task that has direct access to the hardware keeping the log (i.e. flash memory) and will act as a server, responding to logging requests of the rest of the system. The calls it shall respond to can be seen in the table below.

Call	Parameters	Reaction
PING	RSVP_SOCKET	Send PONG to RSVP_SOCKET
PONG	NONE	Register response
LOG_ADD	log_t log, time_t time	Write the log entry into the log.
LOG_GET	RSVP_SOCKET, int N	Send the N last log entries to RSVP_SOCKET.
LOG_DEL_N_LAST	NONE	Delete the N last log entries.
LOG_INIT	NONE	Initialize the LOG, any old log entries will be deleted.

Interface for the Event handler
The main responsibilities of the event handler is to execute composed commands, enabling simple commands from the base station to result in multiple commands to be issued inside the satellite. It is implemented as a single thread with one listening socket, EVENT_SOCKET_RX. The Event handler will in general respond to events or commands that entails complex or composite actions to be executed. The list is to be made.

Appendix C

OBC Software Requirements Specification

1 Introduction

The OBC is one of the principal components of the satellite, and is able to control the rest of the system by granting or denying subsystems access to power and the databus. Other OBC tasks include logging of system parameters in addition to preparing and reading data transmitted to and from the communication systems.

The OBC must be designed to be reliable, as maintenance is impossible after launch.

1.1 Purpose

This document provides all the requirements for the On Board Computer for the NUTS satellite. Parts 1 and 2 are intended primarily for users of the OBC (other satellite modules), but will also be of interest to software engineers maintaining the OBC software. Part 3 is intended primarily for software engineers of the OBC, but will also be of interest to users.

1.2 Scope

This document covers the requirements that must be met by the on board computer software. The purpose of this is to guide developers in selecting a design that will be able to accommodate the full-scale NUTS OBC requirements.

1.3 Definitions, acronyms, and abbreviations

1.4 References

NUTS mission goals NUTS Power Budget 2015 OBC functional requirements OBC non-functional requirements

1.5 Overview

The OBC is one of the principal components of the satellite, and is able to control the rest of the system by granting or denying subsystems access to power and the databus. Other OBC tasks include logging of system parameters in addition to preparing and reading data transmitted to and from the communication systems.

2 Overall description

The main responsibility of the OBC is to monitor the health of the system and to take necessary actions when situations demand for it. It monitors the health of the satellite by periodically requesting health packages from software instances as well as polling sensors for the different modules' power consumption. It also monitors the satellite battery power level, and sets the satellite state appropriately. In addition to this the OBC also acts as a gateway between the satellite bus, and the radio link to ground station. The satellite is designed as a distributed system with redundant functionalities implemented in different modules. The OBC has a sister microcontroller at the UHF Radio module, with a very similar hardware and software setup. OBC functionalities are therefore also implemented in the UHF Radio module.

2.1 Product perspective

The NUTS satellite is intended to be an earth observational satellite. It's main goal however is to raise awareness and build competence related to space development with students enrolled in studies at NTNU. There are many other observational satellites that have had similar mission goals as the NUTS satellite, among these are ESTCube-1.

System interfaces

User interface Concepts

The user interface for the NUTS OBC software are reachable though the ground station GUI. All requests must conform to the Cubesat Space Protocol.

Hardware interfaces

The OBC software runs on the main OBC microcontroller. A sketch of the OBC hardware setup can be seen in figure C.1. A detailed pinout description can be found in the OBC Hardware Schematics hosted on the internal wiki.

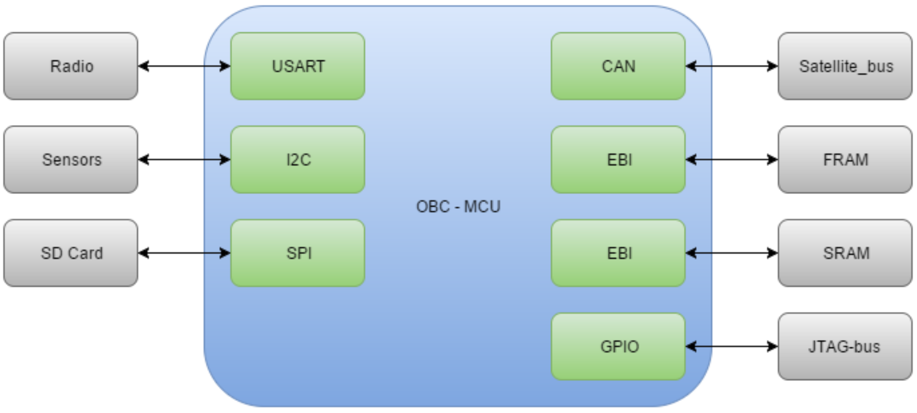


Figure C.1: OBC Hardware Interfaces

Software interfaces

All Communication with the OBC are expected to conform to the CubeSat Space Protocol.

Communications interfaces

The OBC must be able to communicate with all satellite subsystems, these entail:

- ADCS
- Payload
- Radio (ground station)
- Backplane
- Sister-OBC

Memory constraints

The OBC software design shall aim at using no more than 70 % of the available system memory and throughput.

Operations

The OBC software shall be constructed for operation in space, but must also support running software in the lab.

Site adaption requirements

The OBC software shall be easily portable to other 32-bit ARM-based microcontrollers. In particular, the software must be able to run on both the UC3C and the SAMV71 micro-contollers from Atmel.

2.2 Product function

The functional requirements of the OBC is captured in the OBC Function Requirement Specification.

2.3 User characteristics

The system will be further developed, maintained and used by master students enrolled in engineering sciences at NTNU. In particular, student related enrolled in electronics, cybernetics, computer sciences and communication sciences are expected to be the users of this system. The user is thus expected to be technically proficient, but inexperienced.

2.4 Constraints

The software must be able to run on the OBC hardware. The currently used microcontrollers and their internal specifications can be seen in Table C.1, while their external memories can be seen in the list below:

- 4 Mbit FeRAM B85R4001
- 16 Mbit SRAM IS61wv102416BLL-10TLI
- 32 GB NAND FLASH (MT29F16G08ABACAWP-ITZ:C)

Value	**UC3C**	**SAM V71**
Frequency $[MHz]$	66	300
Flash $[MHz]$	512	2048
SRAM $[KB]$	64	384
Power Consumption Active $[mA]$	40	100
Power Consumption Sleep $[\mu A]$	31-100	5.8-24000

Table C.1: UC3C and SAM V71 specification

The OBC is on a power budget. Its power consumption should therefor not exceed the set limit of 300 mW average power consumption.

2.5 Assumptions and dependencies

It is to be expected that the hardware as well as the software may be subjected to change and thus measures must be taken so that change can be handled as elegantly as possible.

This documents assumes the NUTS hardware version 4/Engineering model 2. The use of CSP for communication between satellite systems is also assumed.

2.6 Apportioning of requirements

The requirements described in Sections 1 and 2 of this document are referred to as ”C-requirements”; thise in Section 3 are referred to as ”D-requirements.” The primary audience for C-requirements is the user community (non-OBC developers), and the secondary audiance is the OBC developer community. The reverse is true for the D-requirements. These two levels of requirements are intended to be consistent. Inconsistencies are to be logged as defects. In the event that a requirement is stated within both the C-requirements and the D-requirements, the applications shall be built from the D-requiremen version since it is more detailed. ”Essential” requirements (referred to in section 2) are to be implemented for this version of the OBC. ”Desirable” requirements are to be implemented in thir release if possible, but not committed to by the developers. It is anticipated that they will be part of a future release. ”Optional” requirements will be implemented at the discretion of the developers.

2.7 External interface requirements

The OBC user interface consists of CSP packets arriving on either the USART link from the Radio microcontroller or the CAN tranciever connected to the Satellite communication bus.

User interfaces

All communication with the OBC is done through CSP, which uses a Berkley socket-like API.

Hardware interfaces

The OBC hardware setup can be seen in figure C.1. The OBC microcontroller interfaces with the UHF/VHF radio via USART. External flash bank via SPI. External SRAM and FRAM connected to the External Memory Bus of the microcontroller. The OBC is also connected to a JTAG bus, I2C based sensor bus and a CAN transceiver.

Software interfaces

All software interfaces uses CSP. All communication is handled by the CSP middleware.

Communications interfaces

All communication with the OBC must adhere to the service contract described in the OBC Service Contracts.

2.8 Classes/Objects

The OBC is considered to comprise of four software modules:

- Scheduler
- Housekeeper
- Memory manager
- Event manager

2.9 Performance requirements

The OBC must handle incoming packages in a satisfactory fashion. Except for this, there are no hard timing demands for the OBC.

2.10 Design constraints

The NUTS project is student driven with a high turn-over of inexperienced personnel. It is therefor imperative that the software is easy maintainable and user friendly. An engineering student should be able to understand what is necessary to contribute to the project in less than one semester.

2.11 Software system attributes

McConnel, the author of Code Complete, states that there are several general characteristics to high quality design. These can be seen in the list below:

- Minimal complexity
- Ease of maintenance
- Loose coupling/high cohesion
- Extensibility (Extensibility means that you can enhance a system without causing violence to the underlying structure. You can change a piece of a system without affecting other pieces. The most likely changes cause the system the least trauma.)
- Reusability (Reusability means designing the system so that you can reuse pieces of it in other systems)s
- Portability
- Leanness (Leanness means designing the system so that it has no extra parts).
- Stratification (Stratification means trying to keep the levels of decomposition stratified so that you can view the system at any single level and get a consistent view. Design the system so that you can view it at one level without dipping into other levels)
- Standard techniques

3 Supporting information

OBC pages on NUTS' internal Wiki pages.

4 Document Version History

Revision	Date	Author	Description
1.0	05.04.2016	Magne Normann	Initial Document Release

Table C.2: OBC SRS Document history

Appendix D

OBC Software Design Document

1 Introduction

The OBC is one of the principal components of the satellite, and is able to control the rest of the system by granting or denying subsystems access to power and the databus. Other OBC tasks include logging of system parameters in addition to preparing and reading data transmitted to and from the communication systems.

The OBC must be designed to be reliable, as maintenance is impossible after launch.

1.1 Purpose

This document describes the NTNU Test Satellite On Board Computer software design architecture.

1.2 Scope

This design is the first full scale design for the OBC. It describes the software architecture of the OBC as well as the detailed design of each of the modules.

1.3 Definitions, acronyms, and abbreviations

OBC - Onboard Computer
RTC - Real Time Clock
AST - Asynchronous Timer
CSP - Cubesat Space Protocol

2 References

”Software Design of an Onboard Computer for a Nanosatellite”, by Magne Normann

3 Decomposition description

The decomposition of the system has been obtained by discussion in the NUTS group and can be seen below:

- Timekeeper
- Housekeeper
- Memory manager
- Event manager

Communications are handled by CSP middleware though the CSP router task. This modularization was chosen as it inherently supports service rerouting while having minimal shared drivers, which in turn simplifies error confinement as well as multiple concurrency related issues. The architecture implements memory management as a service. This makes the memory interface available for calls from ground station, as well as enabling for easy service rerouting internally in the satellite in the event of flash failures.

3.1 Module decomposition

This section describes each of the modules that make up the OBC software architecture.

Timekeeper

The timekeeper module is responsible for keeping track of time and the scheduling of events to be initiated by the passing of time. The timekeeper is the only module using the RTC/AST driver.

Housekeeper

The housekeeper module is responsible for managing the health of the satellite. It should periodically issue tests to verify the correctness and liveliness of the satellite. The housekeeping tests can be made rather advanced, with individually testing each software module from multiple other modules. However an effort has been made to keep the error states as few and general as possible to avoid further complicating the system. Whenever there is an error discovered inside a sub-module, the entire module will be reset. This is done to avoid additional errors and complicated failure modes that might follow a single submodule reset/re-initialization. This means that although the system design supports intricate tests, where single software submodules may be tested with customized tests issued from various other modules, and the following results discussed among the other modules, the author emphasizes simplicity and only tests deemed absolutely necessary are to be implemented.

Memory Manager

The memory manager is the software service responsible for storing and retrieving log entries, boot images, as well as reprogramming other modules. It is implemented as a single thread with one listening socket, MEMMAN_SOCKET_RX. The memory manager will be the only task that has direct access to the hardware keeping the log (i.e. flash memory) and will thus need to act as a server, responding to the logging requests of the rest of the system.

Event Manager

The main responsibilities of the event handler is to execute composed commands, enabling simple commands from the base station to result in multiple commands to be issued inside the satellite. Originally it was planned that all messages would be unpacked and inspected and then routed to the correct module by the event handler, however as the use of CSP enables routing of packages directly to the correct module, this is alleviated from the event handler. This means that though all actions could be performed by sending an array of commands to be executed from the ground station, it is considered to be preferable if these action sequences were grouped together and triggered by a simple command. This forms an easier interface where the exact sub-commands and sequences are abstracted from the user interface presented to the ground station.

4 Dependency description

The implementation must be made on an OS that supports CSP such as Linux or FreeRTOS.

4.1 Concurrent process

All the four modules are implemented as concurrent processes, effectively functioning as servers in the satellite network. In additino to the four OBC threads there is a fifth tread implemented by csp, named the router task. This task is responsible for the delivering of CSP packages between modules and threads.

5 Interface description

The following section describes the interface for each of the OBC software modules.

5.1 Interface for the Timekeeper

The Timekeeper is to be implemented with one blocking listening socket TIMEKEEPER_SOCKET_RX that shall receive and react to calls concerning time and alarms. The calls it shall respond to can be seen in the table below.

Call	Parameters	Reaction
PING	RSVP_SOCKET	Send PONG to RSVP_SOCKET
PONG	NONE	Register response
TIME_GET	RSVP_SOCKET	Send current time to the RSVP_SOCKET
TIME_SET	time_t time	Update the current time to the one received
ALARM_SET	time_t time, action_t action	Add alarm to execute ”action” at ”time”
ALARM_DEL_ALL	NONE	Delete all alarms

The action type received as parameter in the ALARM_SET call is to be from the same action set that is implemented within the event manager.

In addition to the TIMEKEEPER_SOCKET_RX the timekeeper shall also use an outgoing socket named TIMEKEEPER_SOCKET_TX, which is to be used when responding to time requests and triggered alarms.

5.2 Interface for the Housekeeper

Call	Parameters	Reaction
PING	RSVP_SOCKET	Send PONG to RSVP_SOCKET
PONG	NONE	Register response
RUN_HEALTH_TEST	NONE	Run a full satellite test
ERROR_REPORT	error_t error	Do necessary actions, such as logging the error and saving the state, before resetting the module
SYNC_TIME	time_t time	Synchronize all RTCs in the satellite with the time given, if no time is given synchronize using the timekeeper time.
GET_SAT_STATE	RSVP_SOCKET	Retrieve the current satellite state and send it to RSVP_SOCKET.

5.3 Interface for the Memory Manager

Call	Parameters	Reaction
PING	RSVP_SOCKET	Send PONG to RSVP_SOCKET
PONG	NONE	Register response
LOG_ADD	log_t log, time_t time	Write the log entry into the log.
LOG_GET	RSVP_SOCKET, int N	Send the N last log entries to RSVP_SOCKET.
LOG_DEL_N_LAST	NONE	Delete the N last log entries.
LOG_INIT	NONE	Initialize the LOG, any old log entries will be deleted.
BOOT_ADD	boot_image[]	Store boot image in flash
BOOT_GET	RSVP_SOCKET, boot_image_id	Retrieve boot image from flash
PROG_FIRM	Module, boot_image_id	Reprogram module with firmware boot_image_id

5.4 Interface for the Event Manager

The Event Handler is implemented as a single thread with one listening socket, EVENT_SOCKET_RX. The Event handler will in respond to events or commands that entails complex or composite actions to be executed. The list is to be made.

5.5 Data decomposition

TBD.

6 Detailed design

6.1 Timekeeper

The internal AST/RCT will be used to keep absolute time, however the OBC specification demands not only that the OBC needs to keep track of time, but also that it must be able to set alarms to schedule future executions cf. requirement R01-OBC-EXE-TIM-002. The AST/RTC only supports two alarms; ALARM0 and ALARM1. Any scheduling thus needs to reuse alarms. This can be solved by keeping a sorted array of set alarms where ALARM1 is set to the first one in the array, which will always be the one closest in time. When the alarm is triggered it resets itself to the next alarm in the array. For memory purposes, the alarm is implemented as a ringbuffer of a fixed size, and new alarms are added using bubble sort (from back to front). Bubble sort is chosen for its simple implementation and the fact that if one makes the assumption that most new alarm times are likely to be further into the future than the ones already present in the buffer, this minimizes sorting operations. A faster alternative could be to use a dynamically allocated linked list, instead of the ring buffer and sorting, as this would reduce both the time and memory usage. However due to the added complexity and the difficulties related to debugging problems linked to dynamically allocation, the sorted ringbuffer-method is deemed safer. Alarms should be shadowed in non-volatile memory in case of any resets. The time can be set externally (the housekeeper is responsible of synchronizing time between the modules.) The timekeeper module itself should be as simple as possible, and no self-synchronizing should be implemented. This is done to avoid complex and/or uncontrolled behaviors in the OBC. All time synchronization should happen on the initiatives of the housekeeper or the event handler as these modules have a much better image of the satellites situation.

6.2 Housekeeper

Error Recovery

As there are no hard deadlines, the added implementation and performance cost of static redundancy was not deemed cost-effective and thus dynamic redundancy is being used for fault tolerance. Anderson and Lee states there should be four constituent phases to dynamic redundancy. These are (1) error detection, (2) damage confinement and assessment, (3) error recovery, and (4) fault treatment and continued service [8]. Tests should only report errors, so that another entity, to which all errors are reported, can make the assessment and initiate any error recovery procedures. As the housekeeper is concerned with the health of the satellite, it seems fitting that such error handling is to be implemented here.It is decided that high-level error handling shall incorporate the three first stages suggested by Anderson and Lee. Procedures trying to unearth errors will run periodically, and report any errors when detected so that further investigations can be launched before any measures are taken to recover from the error.

Managing the Satellite State

In order to make the right decisions, the housekeeper will need to at all times know the current state of the satellite, as well as to have an understanding of previous events that have occurred in the system. This data must survive any module resets and should therefor be stored in non-volatile memory. Care must be taken however that the system still works if this non-volatile memory should fail. One solution might be to keep shadow-copies in various memories in order to have multiple places to get the information if one specific memory should fail permanently. The implementation and performance costs must be considered. The author proposes a scheme where a volatile working copy is kept in non-volatile memory, and is shadowed in MRAM, as well as being logged to non-volatile memory in the logging module. This way the system should function properly if one of the non-volatile memories fail permanently, and should be able to continue with a partial degradation in functionalities even if all external memories are broken. One could also synchronize the satellite's state between the OBC and its sister module, however when further investigating this method one discovers that this may be quite risky. Whenever a permanent error is detected in the OBC or its sister module, the one which can be considered the healthiest will be promoted to active state as the other goes into a passive state [9]. This means that if an OBC has a faulty memory and is still active, its sister module must be even worse off, so care should be taken before relying on services from the passive sister

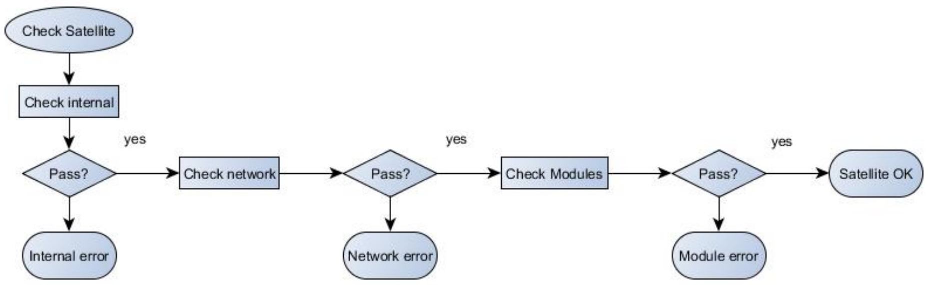


Figure D.1: Satellite check

module.

Acquiring the Satellite State

All errors discovered through normal use shall of course be signaled to the housekeeper. In addition to this, the housekeeper shall periodically test the different parts of the satellite in order to uncover any silently failed modules. This periodic checkup should start by verifying the integrity of the OBC in order to assure that the most vital components are intact, as well as confirming that the testing facility is indeed functional before starting to test external modules. If no errors are detected the network should be tested, before finally, the other modules of the satellite can be tested. It is important that the network check is done before any external modules are checked as external modules cannot be tested without the interface being functional.

Internal Integrity Check

Burns and Wellings states that the major application detection techniques are replication checks, timing checks, reversal checks, coding checks, reasonableness checks, structural checks, and dynamic reasonableness checks [8]. Most suited for detecting memory corruption is the coding checks, and thus this is what shall be used to detect whether the program data has been corrupted. When it comes to testing the external memories, both timing checks, as well as code checks could be useful. The timing checks should be made on the read and write commands, to ensure that an error does not block eternally, but rather times out and reports the failure. If the timing however checks out one could use coding checks to verify that not only the access is functional, but also that the contents are intact.

After the various memories have been tested, the different internal software modules can be tested. These tests can consist of anything from a simple ping-pong routine, to including just about every method suggested by Burns and Wellings. The author of this thesis suggests to not go all overboard with this but only implement simple functional tests. The memory manager facility can be tested by requesting a write to log, and then a read-back of the same entry. The timekeeper facility can be tested by adding an alarm that should

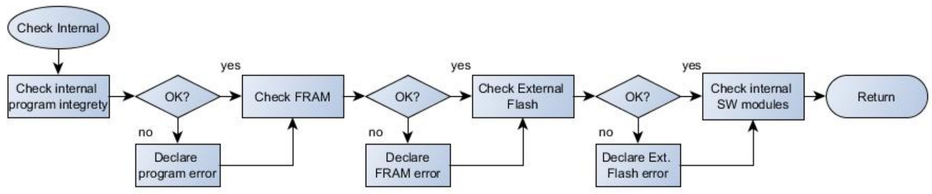


Figure D.2: Internal check

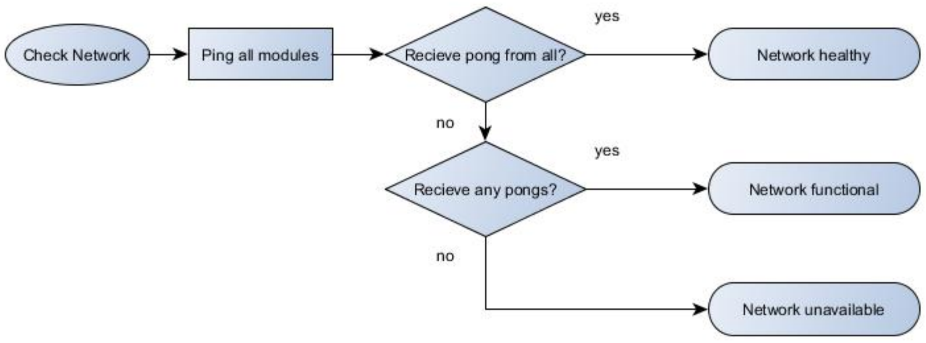


Figure D.3: Network check

be triggered in the immediate future and do a timing check on the response. The event handler facility can be tested with a simple ping-pong routine.

Satellite Network Check

The satellite network check can be made by simply pinging all other modules, and wait for responses for a limited time. If all modules respond within the time limit, the network is assumed functional. If some modules respond, but not all, the network is considered partially functional and an exception is thrown. If no responses are received, the network is assumed to be non-functional.

Module Check

As all modules are interfaced through the CSP sockets, performing functional timing tests are easily implemented as each call to socket receive can be given a timeout period, specifying how long the socket should block and wait for an incoming message. In fact, one OBC can perform exactly the same functional tests on the other OBC as it did on itself by simply addressing the respective socket on the other OBC instead of the one implemented in itself. For example, to verify the memory manager on the OBC-sister the OBC can request to write to and read from the log, but instead of using addressing the requests to its own memory management socket, it can send them to the socket of the memory manager

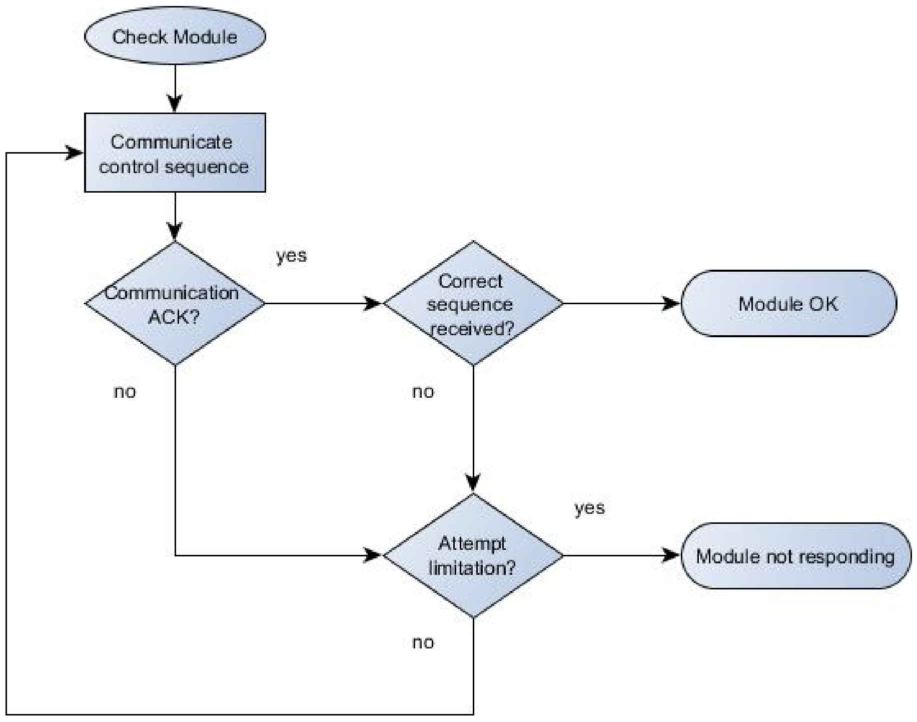


Figure D.4: Module check

at the other OBC.

The Error Handler

As error handling code in itself can be a source of errors, the complexity of error handling code should be minimized if a maintainable system is to be constructed. It is therefore recommended to implement basic catch-all measures rather than specialized handlers for complicates error states. As a result of this the author recommends that a complete module reset is a better response than single submodule repair in the face of soft errors. One of the main arguments for this is that it makes for easier maintainable code as well as error propagation may be minimized.

Upon reset, the housekeeper checks the battery level to see if the satellite can enter the test mode. If the battery power level is sufficient the housekeeper starts a health check. First it checks its own host-module. The non-volatile memory is checked for any status information, as well as running an integrity check for the rest of the non-volatile stored data. If any errors are unearthed in the self test, these must be dealt with before testing the rest of the system. However, the internal test is still to be completed, even if a previous step uncovered an error. This is done to get an overview of the situation before making any decision on what the appropriate counter measures are. This is in turn done to ensure that multiple errors are registered and handled correctly. If the self test is positive, the other satellite modules are tested as the flowchart below depicts. Any failed tests, result in a limited amount of module resets before a module is declared dead.

6.3 Memory Manager

The memory manager is the software service responsible for storing and retrieving log entries, boot images, as well as reprogramming other modules. It is implemented as a single thread with one listening socket, MEMMAN_SOCKET_RX. The memory manager will be the only task that has direct access to the hardware keeping the log (i.e. flash memory) and will thus need to act as a server, responding to the logging requests of the rest of the system.

Timestamp Supplier

All log entries should be timestamped in order to construct time lines of event. One question that arises is then; should the time be supplied by the caller, or by the memory manager? It is not unlikely that some drift will occur between the different clocks in the satellite, despite valiant efforts of synchronizing them. Would it then make more sense to use the time from the local timekeeper, assuring that all entries are stamped chronologically to when they are received, or should the caller stamp the log message before sending, assuring that no delays due to packet loss or resending affects the time stamping? One solution might be to use both, this would effectively mark each log entry both when it was sent, and when it was received. Making it easier to investigate synchronization or communication issues in the satellite.

File System

The memory manager should support some sort of file system to categorize different entries. When choosing a file system care must be taken both to the memory footprint of the system as well as its reliability. It is not unlikely that power outs or outside of spec voltages may occur. It is therefore important that the memory manager does not corrupt the memory, even if operations are aborted half finished. For this reason the developer may want to use a journaling file system, as such file systems can be brought back online more quickly with lower likelihood of becoming corrupted after power failures [44] [45]. The developer may also want to investigate whether the Reliance Edge fail-safe file system included in FreeRTOS+ Ecosystem [46] is an appropriate file system or not.

In-Orbit Programming

Due to R00-OBC-PRG-001 requirement, the OBC must be able to program other modules after launch. This firmware update can be done either by the microcontroller itself or by another microcontroller. Priority is given to programming of one microcontroller by another, as this can be forced upon a microcontroller with a faulty firmware. Marcedella implemented firmware update using the JTAG protocol. However his method is target specific and does not easily port to other microcontrollers or boot images. For this reason a new programming scheme needs to be implemented, though it is recommended to use Marcadellas work as guiding.

7 Document Version History

Revision	**Date**	**Author**	**Description**
1.0	30.04.2016	Magne Normann	Initial Document Release

Table D.1: OBC SDD Document history

Appendix E

Cubesat Space Protocol for intra-Module Communication

Cubesat Space Protocol for intra-Module Communication

Magne Normann

M.Sc Student, Department of Cybernetics and Robotics
NTNU, Norwegian University of Science and Technology
O.S. Bragstad plass 2D, N-7491 Trondheim - Norway
magnealv@stud.ntnu.no
Phone: +47 97005173

Abstract

When designing for embedded systems, the communication model used for internal communication plays a vital role to the system complexity. A suited communication model assures modularization and greatly eases system development and maintainability. While an unsuited communication model can make system development challenging, often forcing the developer to write complicated code, resulting in added complexity and lowered maintainability. This paper suggests the Cubesat Space Protocol (CSP) as a viable solution for both inter- and intra-satellite communication. This enables for a seamless distributed system, where submodules can communicate with other submodules in a unified manner, no matter if the other submodules are implemented inside the same microcontroller or at the ground station. There are some unnecessary added overhead, as not all communications need checksums or congestion control, but if the added overhead and communication time is acceptable, the CSP layers can abstract the underlying interfaces and routing, thus easing system design, as subsystem developers only need to think about defining a service-contract, and a set of port-numbers her/his system will be responding to. In this paper the advantages and disadvantages of using CSP for inter-task communication are discussed. A small test application is analysed to measure the computational overhead of different CSP features in order to evaluate it for satellite applications. It is concluded that using CSP can be very advantageous for non-time critical communication but is not recommendable where synchronization is highly time sensitive.

1 Introduction

Cubesats are becoming more and more popular, yet failure rates are very high with less than 50% success rate [1]. One of the many challenges of cubesat development is keeping the system complexity at a manageable level. Real-time and embedded programs (such as cubesat applications) have to control and interface with real-world entities (thrusters, switches, sensors, etc.) that are inherently parallel. Reflecting the parallel nature of the system in the structures of the program makes for a more readable, maintainable and reliable application. This is one of the main motivations for writing concurrent programs [2]. However, communication and synchronization in concurrent programs are far from trivial and can be really difficult to understand for humans. There may be the risk of oversights, even after very rigorous code inspections [3]. For this reason, software with proven flight heritage is often preferred. The Cubesat Space Protocol (CSP), formerly known as CAN Space Protocol, is a small protocol stack with proven flight heritage. It is written in C and its design follows the TCP/IP model and includes a transport protocol, a routing protocol and several MAC-layer interfaces [4]. It was developed at Aalborg University in 2008, and is now maintained by GomSpace. An implementation of the basic CSP functionality is released by GomSpace as a software library under GNU Lesser General Public Licence (LGPL), allowing the material to be copied and used by the public. The protocol supports up to 16 modules, whereas each module can have up to 64 ports. The protocols supports next-hop routing as well as loopback mode. According to GomSpace, known satellites or organizations that uses CSP include GomSpace GATOSS GOMX-1, AAUSAT-3, EgyCubeSat, EuroLuna, Hawaiian Space Flight Laboratory, GomSpace GOMX-3, and NUTS-1.

Though originally intended for cross-module communication, the protocol does support loopback mode. Meaning that messages could be sent from one thread to another running on the same microcontroller via CSP. This report will discuss and test the possibility of using CSP as a unified communication interface for satellite services regardless of whether they are implemented inside the same module or not.

2 The perks of CSP for inter-thread communication

The main arguments for using CSP as a unified interface for both internal and external services are that it may minimize system complexity and ease error checking and handling. CSP uses a service oriented topology with a Berkley/POSIX socket-like Application Programmer Interface (API). Most computer developers have at least some experience with socket programming and are thus already familiar with the basics of how the CSP API is used. Using this service oriented architecture also encourages developers to minimize dependencies to other sub-modules and uphold a standard way of communication throughout the satellite.

If CSP is used not only for inter-module communication but also for intra-module communication, every internal service becomes (if wanted) accessible to other modules as well. This means that if a service provider permanently fails, the service can be requested from any other service provider with similar request-handling, simply by changing the receiver address. An example of this could be if the logging service of a module goes down due to some permanent hardware failure. The module would then be able to use the logging service implemented in another module, by changing the receiver address it sends its log messages to.

If a housekeeping entity is being used to manage the health of the satellite, then it can be used to update what sockets should be used in case of any permanent submodule failure, abstracting any satellite-system error handling away from the submodules. This means that a submodule developer does not need to check for or implement handling in case of any failures in other submodules. If other modules goes down, the housekeeper will change the addresses appropriately and the submodule application developer is only responsible of implemented error handling for its own services.

Concerning congestion control and error checking, CSP supports a handful of Internet Control Message Protocol (ICMP) calls, as well as services such as requesting memory and buffer status. It also has support for getting and setting time, retrieving the up-time of another module, and retrieving a list of the running threads as well as their statuses. In addition to this, the CSP router task supports Quality of Service (QoS) and can evaluate it's own performance. If CSP is used for inter-thread communication, all these services becomes available for submodules as well and may ease the design and implementation of certain housekeeping functionalities as the responsiveness and healthiness of any submodule connected to the network can be investigated thought standard CSP-calls.

3 The trade-offs of CSP for inter-thread communication

The main trade offs for using CSP for inter-thread communication is considered to be the added overhead resulting in slower execution and more memory needed, and the fact that the CSP router-task poses a single point of failure for the communication both internally as well as externally.

Using CSP for both internal and external communication causes all packages to be handled by the router task. This makes the router task a single point of failure for the entire module. If the router task suffers a failure, no communication, not even internally to the module is possible. If one considers the alternative of using native FreeRTOS synchronization techniques for inter-task communication. The module may continue to function with some degraded performance. However it would be much harder to investigate and verify the healthiness of the internal communication. It can also be argued that as there may only be limited value in the internal healthiness of a module that cannot communicate with the rest of the system, there is only limited value in separating the two communication schemes as both must work for the satellite to be functional. It may thus be worth the payoff to join these two single point of failures into one, as this greatly simplifies error detection and handling.

CSP conforms to the TCP/IP model with multiple layers. This means that the underlying drivers and low level details are abstracted away for the application designer. This simplifies design and implementation, but comes at a cost. Added abstraction means added overhead. Whether or not the additional computational overhead of using CSP is acceptable or not depends on the application requirements. To investigate how much overhead computation CSP imposes on the application a test bench was created. The test bench, the results from the tests and a discussion of the results will be presented in the following sections.

4 Testing the build

To estimate the additional overhead that comes from using CSP for thread communication a simple test system was set up and analyzed. The test system consists of three threads; two user threads (one client and one server) and the CSP router task (RTE). To test the CPU overhead the client constructs and sends a packet with a payload of 100 bytes to the server. The server does any necessary check for the integrity of the package before sending it back to the client. The client then verifies the content of the package and records the time the transaction took. The average execution time is obtained by calculating the average over 100 iterations. Each iteration is measured by reading the ARM SYSTICK timer, effectively giving how many CPU cycles was spent on the procedure.

The test setup consisted of an ARM M7 microcontroller (ATSAMV71-Xult, situated on a SAMv71 Xplained Ultra Evaluation Kit) running the currently last official release of FreeRTOS (version 8.2.3) with a system frequency (CPU and bus) of 75 MHZ and 1 ms RTOS ticks.The setup was tested using preemptive scheduling, with stack overflow checking enabled. The test was performed for various optional CSP features enabled such as the Reliable Data Protocol (RDP), check-sums (CRC32), encryption (XTEA) and authentication (HMAC-SHA1).

5 Results

First a small application using only FreeRTOS queues to communicate directly from the client task to the server and back was tested. The application was compiled with GCC and uploaded to the microcontroller using an Atmel-Ice debugger. The memory usage reported from the build process can be seen in table 1. The application ran for 100 iteration and the average execution time was calculated from these 100 samples. The result can be seen in table 2.

Communication form	**Program Memory Usage**	**Data Memory Usage**
FreeRTOS queues	37572 bytes (1.8%)	60304 bytes (15.8%)

Table 1: Memory usage for native FreeRTOS queue test application

Enabled Features	**Duration [ticks]**	**Duration at 75 MHz [ms]**
No features	30 231	0.403

Table 2: Latency for for native FreeRTOS queue test application

After testing the regular FreeRTOS queues, a small application running CSP was tested with various features enabled. The memory usage can be seen in table 3, while the duration for the setup and communication of each iteration can be seen in table 4.

Communication form	**Program Memory Usage**	**Data Memory Usage**
CSP sockets	89440 bytes (4.3%)	62040 bytes (15.8%)

Table 3: Memory usage for CSP socket test application

Enabled Features	Duration [ticks]	Duration at 75 MHz [ms]
No features	134 586	1.795
CRC32	149 374	1.992
RDP	297 681	3.969
HMAC-SHA1	251 271	3.350
XTEA	291 829	3.891
RDP + CRC32	316 042	4.214
ALL	785 040	10.467

Table 4: Latency for CSP test application

For connection-less communication with no CRC or encryption there is a 345.19 % workload increase, which at 75 MHz corresponds to an additional 1.392 milliseconds of execution time. If connection oriented communication with resending of lost packages is being used, the overhead is an additional 163095 cycles compared to the connection-less option. This corresponds to about 2.174 ms extra computation time for the round-trip when running at 75 MHz.

6 Discussion

From the results one can see that when comparing using CSP with no features enabled, it uses more than 1000 execution cycles more than the native FreeRTOS queue setup. For a microcontroller running at 75 MHz this means an additional 1.4 millisecond of execution time. For many application this may be affordable, but for others it may prove CSP unsuitable. CSP packets can be prioritized and thus important packages will be prioritized in front of lower priority ones, however on a system wide level they will still only be handled when the router task is allowed to run. If any higher prioritized thread is running, additional delay for the package handling must be expected. For native FreeRTOS queues, package prioritization is not supported, but it is rather the priority of the receiver task that dictates whether or not the package handling gets precedence. Neither one of these methods are ideal for urgent synchronization, and may suffer from extra delays if the system is not constructed properly. However as CSP communicates though FreeRTOS queues (when running on FreeRTOS), the added delay can be considered as a trade-off for the prioritized handling, and may in some cases lead to faster execution with CSP if the workload is large.

With additional features enabled, the CPU load increases. CSP does support any combination of the features mentioned above for each socket separately. The handling of resending of missed packets proved functional, though time characterization for this was not attempted as both timeout and number of resending is user configurable. The different CSP calls for ping, thread listing (ps) requesting of up-time and memory statuses were verified functional as well, and can ease monitoring and error handling in the communication system.

The memory usage of a CSP application is highly configurable through enabling and disabling features as well as customizing the amount of memory allocated for both tasks and queues. The memory usage of the system using CSP sockets and the system using native FreeRTOS queues directly can be seen in table 1 and 3. These memory consumptions are only meant to be indicative as neither one is optimized for size as this is a question of maximum capacity and load handling.

CSP uses a zero-copy buffer and queue system, meaning that data isn't actually copied and moved around, but rather only the address reference is handed from one handler to another. This results in efficient use of memory, but as with most message passing systems, deciding the buffer size of each queue does pose a challenge. All buffer sizes must be declared to their maximum capacity. As the maximum capacity is much larger than what is normally needed, a large portion of the memory will be occupied but very seldom used. If CSP weren't used, but instead native FreeRTOS queues directly, each queue would have to be declared to its maximum capacity. Substituting this with one router task that handles all communication may then be advantageous as it is unlikely that maximum capacity is needed for all communication interfaces at the same time, thus a smaller memory buffer may be allocated for the router task than if multiple buffers were created separately.

7 Conclusion

In this report CSP has been evaluated for inter-thread communication. A test application was developed to test the computational overhead that comes from using CSP. The results were then discussed and when applicable compared to that of using FreeRTOS queues directly. It was concluded that CSP can be advantageous, but only if speed is not of critical importance. The latency of CSP calls may be non-deterministic and thus unsuited for hard real-time application such as time sensitive calculations.

CSP does however enable a seamless distributed system architecture, though the use of sockets, where submodules can communicate with other submodules in a unified manner, no matter if the other submodules are implemented inside the same microcontroller or at the ground station. CSP also includes tools for congestion control and common network requests such as ping, buffer status, getting and setting of time as well as listing of running tasks and their statuses. This makes CSP an attractive alternative for both internal and external module communication as it greatly eases the monitoring and management of the entire satellite communication.

References

[1] M. Swartwout. Cubesat database. `https://sites.google.com/a/slu.edu/swartwout/home/cubesat-database#plots`, 2016.

[2] Wellings Andy Burns, Alan. *Real-Time Systems and Programming Languages*. Pearson Education Limited, 2009, 4. edition, 2009.

[3] Peter Marwedel. *Embedded system design: Embedded systems foundations of cyber-physical systems*. Springer Science & Business Media, 2010.

[4] GomSpace libcsp github. `https://github.com/GomSpace/libcsp`. Accessed: 2015-22-11.