Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Joonas Javanainen

Reliability evaluation of Aalto-1 nanosatellite software architecture

Master's Thesis
Espoo, January 26, 2016

Supervisor: Associate Professor Keijo Heljanko
Advisors: Assistant Professor Jaan Praks, Aalto University
M.Sc. (Tech.) Timo Metsälä, Reaktor

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Joonas Javanainen

Title:
Reliability evaluation of Aalto-1 nanosatellite software architecture

Date:	January 26, 2016	**Pages:**	62
Major:	Software Technology	**Code:**	T-106
Supervisor:	Associate Professor Keijo Heljanko		
Advisors:	Assistant Professor Jaan Praks M.Sc. (Tech.) Timo Metsälä		

Nanosatellite research projects are increasingly popular in universities all over the world. These projects offer interesting challenges, and reliability must be considered in architecture design to avoid mission failure.

Aalto-1 is a student nanosatellite project at Aalto University that has been under development for around five years. The satellite is based on the CubeSat specification, and its scientific mission includes hyperspectral imaging, radiation monitoring, and testing of an experimental de-orbiting device.

The goal of this thesis is to evaluate the reliability of Aalto-1 nanosatellite software architecture before launch. In addition to evaluation, design improvements were made to the system boot procedure, watchdog mechanisms, and internal communication.

This thesis confirms that software reliability has been considered in the design of Aalto-1, and the satellite can recover from many failure scenarios. However, the architecture includes some complexity that could be avoided, and further research could be used to validate the correctness of the custom protocols and important recovery logic in the architecture.

Keywords: Aalto-1, Nanosatellite, CubeSat, Software reliability, Redundancy, Watchdog

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Joonas Javanainen		
Työn nimi:	Aalto-1 nanosatelliitin ohjelmistoarkkitehtuurin luotettavuus		
Päiväys:	26. tammikuuta 2016	**Sivumäärä:**	62
Pääaine:	Ohjelmistotekniikka	**Koodi:**	T-106
Valvoja:	Professori Keijo Heljanko		
Ohjaajat:	Apulaisprofessori Jaan Praks FM Timo Metsälä		

CubeSat-määrittelyyn pohjautuvien satelliittien tutkimusprojektit ovat yhä suositumpia yliopistoissa ympäri maailman. Nanosatelliittiprojektit tarjoavat mielenkiintoisia haasteita, ja satelliitin luotettavuus tulee huomioida arkkitehtuurin suunnittelussa, jotta voidaan välttää satelliitin tehtävän epäonnistuminen.

Aalto-1 -nanosatelliitti on Aalto-yliopiston opiskelijaprojekti, joka alkoi noin viisi vuotta sitten. Satelliitti pohjautuu CubeSat-määrittelyyn, ja sen tieteelliset tehtävät ovat monispektrikuvantaminen, säteilymittaukset, ja kokeellisen sähköpurjeen testaaminen.

Tämän diplomityön tavoitteena on arvioida Aalto-1-nanosatelliitin ohjelmistoarkkitehtuurin luotettavuutta ennen laukaisua. Arvionnin lisäksi työssä tehtiin parannuksia järjestelmän käynnistykseen, vahtikoiramekanismeihin, ja sisäiseen kommunikaatioon.

Tämä diplomityö vahvistaa, että ohjelmiston luotettavuus on huomioitu monilla tavoin Aalto-1-nanosatelliitin suunnittelussa, ja satelliitti voi selvitä useista virhetilanteista. Sen arkkitehtuurissa on kuitenkin monimutkaisuutta mikä voitaisiin välttää, ja mahdollisilla jatkotutkimuksilla voitaisiin varmistaa omien protokollien ja tärkeiden palautumisproseduurien oikeellisuus.

Asiasanat:	Aalto-1, nanosatelliitti, CubeSat, ohjelmiston luotettavuus, kahdennus, vahtikoira
Kieli:	Englanti

Acknowledgements

I would like to thank Juha-Matti Liukkonen from Reaktor for introducing the Aalto-1 project to me, and Assistant Professor Jaan Praks for inviting me to the project, and giving valuable feedback and comments about the thesis.

I would also like to thank my supervisor, Associate Professor Keijo Heljanko for his support and for providing several useful references and ideas for this thesis.

Finally, I would like to thank Erica for supporting and encouraging me throughout the project.

Espoo, January 26, 2016 Joonas Javanainen

Abbreviations and Acronyms

AaSI	Aalto Spectral Imager
ADCS	Attitude Determination and Control System
CAN	Controller Area Network
COTS	Commercial-Off-The-Shelf
ECC	Error-Correcting Code
EPB	Electrostatic Plasma Brake
EPS	Electrical Power System
FDIR	Failure Detection, Isolation, Recovery
FPGA	Field-Programmable Gate Array
FRAM	Ferroelectric Random-Access Memory
HMAC	Keyed-Hash Message Authentication Code
GPIO	General-Purpose Input/Output
I^2C	Inter-Integrated Circuit
OBC	On-Board Computer
RADMON	Radiation Monitor
RTC	Real-Time Clock
SHA	Secure Hash Algorithm
SPI	Serial Peripheral Interface
UART	Universal Asynchronous Receiver/Transmitter
UBIFS	Unsorted Block Image File System
UHF	Ultra-High Frequency

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A student nanosatellite project provides exciting design and implementation challenges from both hardware and software points of view. The satellite hardware must be able to function properly in space, which is a much harsher environment than what is typically expected in projects with off-the-shelf hardware. The satellite software must have at least basic recovery capabilities from failures, and the limited debugging possibilities must be considered when designing the system.

Small satellites are typically categorized based on their mass. A satellite with mass between 1 and 10 kg is categorized as a nanosatellite, and a satellite with mass between 0.1 and 1 kg is a picosatellite [17]. CubeSat [9] is a popular specification for small satellites, such as pico- or nanosatellites, consisting of one or more 10 cm cubes with mass less than 1.33 kg. The individual cubes are connected to each other along a single axis, so a 3U CubeSat is 30 cm long and is typically categorized as a nanosatellite.

Aalto-1 [35] is a student nanosatellite project expected to launch in the beginning of 2016. The satellite follows the CubeSat specification and in addition to the main platform includes three payloads with custom hardware and software. Its scientific goals include spectral imaging, radiation monitoring, and testing of an experimental de-orbiting device. The target duration of the mission is two years.

1.1 Problem statement

This thesis evaluates the reliability of Aalto-1 On-Board Computer (OBC) [38] software, and focuses especially on error detection and recovery mechanisms and both internal and external communication systems. Since many fundamental software design decisions were set before this thesis was written,

the main focus is on evaluation and analysis, not fixing all flaws or preventing all potential error conditions. Aalto-1 employs a wide variety of software, so choosing the most important targets for evaluation is challenging. Issues in any part of the system can lead to a mission failure. However, the majority of the software components are existing well-tested software, so we focus mostly on custom critical software components built by the Aalto-1 team.

Some of the most important parts of a nanosatellite architecture are the components that contribute to having working communication with a ground station. Communication is vital for downlinking telemetry and scientific data, which are fundamental for a successful mission. While bandwidth and time limitations greatly complicate debugging and maintenance operations, working communication also makes it possible to manually recover from some error situations which the satellite can not handle automatically by itself. Therefore the main focus of recovery mechanisms should be to maintain a working communication link with the ground station.

1.2 Structure of the thesis

Chapter 2 explains the core challenges in implementing reliable software in a CubeSat satellite. Chapter 3 describes the architecture of Aalto-1. Chapter 4 describes the multiple watchdog mechanisms and the boot procedure used in Aalto-1. Chapter 5 evaluates the reliability of the I^2C protocol and shared library used in the project. Chapter 6 evaluates important aspects of reliable communication with the ground station. Chapter 7 includes discussion about potential future research topics and architectural improvements. Chapter 8 concludes the thesis.

The main contribution of this thesis is analysis and documentation of software designed and developed by other people. However, Chapter 4 and 5 include subsections that describe improvements to the original design of Aalto-1 that were developed as part of this thesis.

Chapter 2

Nanosatellite software challenges

Like many other recent research satellites, Aalto-1 follows the CubeSat specification [9], which specifies physical constraints, and does not mandate any specific hardware or software architecture for the on-board computer or other components. There are constraints for physical characteristics such as mass and volume, but the specification leaves a lot of flexibility in On-Board Computer (OBC) hardware design.

Unlike in avionics or automobile systems, nanosatellite failure scenarios will most likely not result in human casualties. However, reliability should still be considered in the design, because the environment makes many failure scenarios much more probable than on Earth, and a mission failure can have a significant cost in both time and money.

2.1 Physical environment

Space is a harsh environment, where hardware components face much more ionizing radiation than on Earth. This increases the probability of both hard errors, such as permanent damage to a memory cell, and soft errors, such as bit flips in dynamic memory. The satellite must also survive temperature changes and vibration during launch. [17]

A soft error is often called a Single Event Upset (SEU), which is a non-destructive change of state in memory or logic element. Typically the change involves just a single bit, but high-energy radiation can cause changes in multiple bits at the same time. A SEU could for example change the contents of a memory cell, or disrupt the internal logic of a processor. A change of a single bit in the internal L1 cache of a processor or a logic gate can lead to unexpected software behaviour and potentially to disastrous results. [4]

Integrated circuits based on Complementary Metal-Oxide Semiconduc-

tor (CMOS) technology are suspectible to Single Event Latchups (SELs). A high-energy particle may inject a small current into a positive feedback loop structure, which causes a high-current state that may cause permanent damage to the chip. [30]

Soft errors can be mitigated by the use of error-correcting codes (ECC). A RAM chip with ECC can detect and correct some errors caused by SEUs without any impact to the software. Memory must be scrubbed periodically, or soft errors might accumulate undetected in less often used areas of memory. Scrubbing involves reading all memory contents, correcting bit errors if possible, and writing data back into the memory.

A hard error involves permanent damage to the affected component. The risk of a mission failure due to hard errors can be reduced by including radiation-resistant components or redundancy in the hardware design. More advanced components may have additional latch-up protection circuitry.

2.2 Hardware and software choices

Early CubeSats such as CUTE-I [34] used a Microcontroller Unit (MCU) device as the main processor in the OBC architecture. MCU devices are still popular, but recent CubeSats often use more powerful processors. For example, 16-bit Texas Instruments MSP430 microcontrollers have been used in many CubeSats such as Delfi-C3 [54], Delfi-n3Xt [12], the Vermont Tech Cubesat [8] and WinCube [42]. 32-bit ARM processors offer even more computing power and flexibility, while still keeping power requirements low.

The processor choice greatly affects the software architecture. A simple 16-bit microcontroller might use custom bare-metal software for everything or an operating system suitable for MCUs such as FreeRTOS [3]. A more powerful processor can even use a full embedded Linux operating system, which increases system complexity as more software is needed, but it makes a satellite more flexible and approachable to people less familiar with low-level MCU software architectures. There is also a significant amount of existing software for Linux that can be used in the satellite, which can lower the development costs.

FreeRTOS was used for example in the ESTCube-1 satellite. The operating system was chosen based on its low flash and RAM requirements, deterministic scheduling, and simple support for keeping an idle MCU in sleep to reduce power consumption. [48]

California Polytechnic State University started with 8-bit microcontrollers in their CP-series CubeSats, but eventually switched to 32-bit ARM processors and Linux. The use of existing kernel code, and having support for

modern languages such as Python were considered important benefits in the operating system choice. [29]

2.3 Upgradability and configurability

The possibility to update the satellite software is desirable, because software updates can be used to fix problems found after the launch and they can also make it possible to reconfigure the satellite if necessary. Directly reflashing software components stored in nonvolatile memory is a simple but potentially risky way of performing updates. If the update is interrupted, or there is any kind of data corruption in memory, the update might result in a mission failure unless some safety features are included in the update process. If hardware redundancy is included in the hardware architecture, it might be possible to recover by using previous software stored in secondary memory.

A field-programmable gate array (FPGA) offers additional configurability compared to simple microcontrollers and software updates performed by reflashing. The programmable logic blocks of an FPGA can be reprogrammed to perform different functions depending on mission requirements. FPGA devices can also have specialized blocks such as Digital Signal Processing (DSP) blocks that make such devices very flexible and powerful for many applications. [38]

N. Bergmann and A. Dawood suggest that the reprogrammability of FPGAs enables repairing of many errors in flight. However, a reprogrammable FPGA such as one based on SRAM technology, is very suspectible to radiation-induced errors, and fault tolerance needs to be considered in the architecture design. [6]

2.4 Fault Detection, Isolation, and Recovery

Failure Detection, Isolation and Recovery (FDIR) is an important part of nanosatellite architecture. Failure scenarios are likely to happen during the mission of a satellite, and the satellite must be able to recover from many problems autonomously, because a human operator cannot necessarily perform recovery unless communication systems are still operational. All failures can not be detected, and not all of detected failures can be recovered from.

Failure detection mechanisms can include watchdog systems, ECC, and checksums. Some failures can be isolated, for example, by switching to a secondary component if hardware redundancy is present. The main goal of failure recovery is to get the system back to an operational state. Recovery

action can be a small correction such as fixing a single bit error, or a more drastic action such as rebooting the entire system or isolating a misbehaving component completely from the rest of the system. [25]

2.5 Communication with the satellite

Communication with the satellite is usually performed by one or more ground stations that are located on Earth at positions suitable to the orbit of the satellite. The number of available ground stations and the orbit affect the time that is available for communication. Ground stations may monitor and control the satellite by sending telecommand (TC) packets and receiving telemetry (TM) packets. Communication between a nanosatellite and a ground station is often fundamentally asymmetrical, because the uplink and downlink bandwidths can differ by more than one order of magnitude. The uplink can be several orders of magnitude slower than the downlink, so TC packets have much less bandwidth available than TM packets. [17]

2.6 Software processes and tools

CubeSat projects are often student projects with educational goals, so the software processes, programming languages, software libraries, and other tools need to be chosen so that they are accessible to students and do not complicate or slow down the project too much. Expert teams in commercial projects can use the best possible tools available, but a Master's degree student cannot be expected to be productive with very complicated industrial tools. [49]

The C programming language is widely used in embedded software projects, so it is also a natural choice for most software in a nanosatellite. However, C requires significant programmer discipline and its many low-level characteristics make it difficult to write error-free code. Several safety-oriented C programming language subsets and programming standards have been published to alleviate the challenges involved in writing correct C code.

The CERT C coding standard [43] consists of rules and recommendations about code style, programming patterns, language features, and commonly used platforms. It focuses on program reliability and security, but many of the rules might not be very relevant in embedded projects. A more relevant alternative could be MISRA C [1], which is used in the automotive industry and in safety-critical projects. L. Hatton [19] compared safe C subsets, and found some fundamental problems in MISRA C, but considered it to be an

improvement over other unnamed C standards. If the software architecture and hardware capabilities permit the use of other programming languages, C might not be the best choice for all nanosatellite software.

C. Brandon suggests that using the Ada programming language could improve the reliability of nanosatellite software. The Vermont Tech CubeSat uses an MSP430 processor, and the satellite software is programmed with SPARK, which is a subset of Ada. Using Ada also offers pedagogical advantages, because students get valuable experience of development of high-integrity software. [8]

The software development process can also significantly affect the reliability of the resulting software. In their 2009 paper, C. Ebert and C. Jones [13] describe several characteristics of embedded software projects. They suggest that 30 to 40 percent of resources in embedded software projects are spent on testing, and this effort could be reduced by detecting defects earlier. For example, a combination of automated testing, code analysis, and code reviews can help reduce the total amount of defects and the testing effort.

2.7 Surveys of previous CubeSat projects

J. Guo and J. Bouwmeester collected data of 94 pico- and nanosatellites in their 2010 paper [7]. The paper suggests that at the time Microchip PIC and Texas Instruments MSP430 microcontrollers were most common in OBCs, but ARM devices were also starting to become more popular. Almost half of CubeSat missions ended in failure, and around one third of the satellites did not launch successfully.

M. Swartwout has collected statistics about CubeSat projects in several papers [49–52], which have been published approximately yearly. His latest paper from 2013 [52] presents some interesting statistics about CubeSat mission failures. Firstly, the average failure rate in university CubeSat projects is almost 50%. Secondly, the total number of yearly failing CubeSats has not diminished over the years during which statistics have been collected. Thirdly, the most common failure is no contact at all with the satellite after launch, followed by problems in the communication system, power subsystem, and finally the main processor.

Chapter 3

The architecture of Aalto-1 satellite

3.1 Hardware overview

From a software reliability point of view, the most important architectural components of the Aalto-1 nanosatellite are the OBC, the Arbiter, communication radios, and the payload components. The satellite hardware has been tested with rigorous physical tests according to ECSS standards. This includes radiation and thermal tests. [23]

The components used in the OBC and connected systems are mostly commercial-off-the-shelf (COTS) components, which are not specifically designed for use in space. In order to decrease the probability of mission failure and to make the goal of a two year mission more realistic, the hardware includes some cold-redundant components to achieve some hardware fault tolerance. [38]

The system includes an Inter-Integrated Circuit (I^2C) and an Serial Peripheral Interface (SPI) bus that are used to communicate between devices. Some devices that use the I^2C bus use the custom Aalto-1 I^2C protocol, which is described in Chapter 5. The OBC also has two Universal Asynchronous Receiver/Transmitter (UART) connections to devices. Figure 3.1 shows a simplified connectivity diagram with the main focus on critical components required for functional communication. [22, 46]

Concerning the buses in the satellite, this thesis evaluates the reliability of only the I^2C bus, but other internal communication mechanisms also contribute to the reliability of the entire system, and could be potential targets for further research.

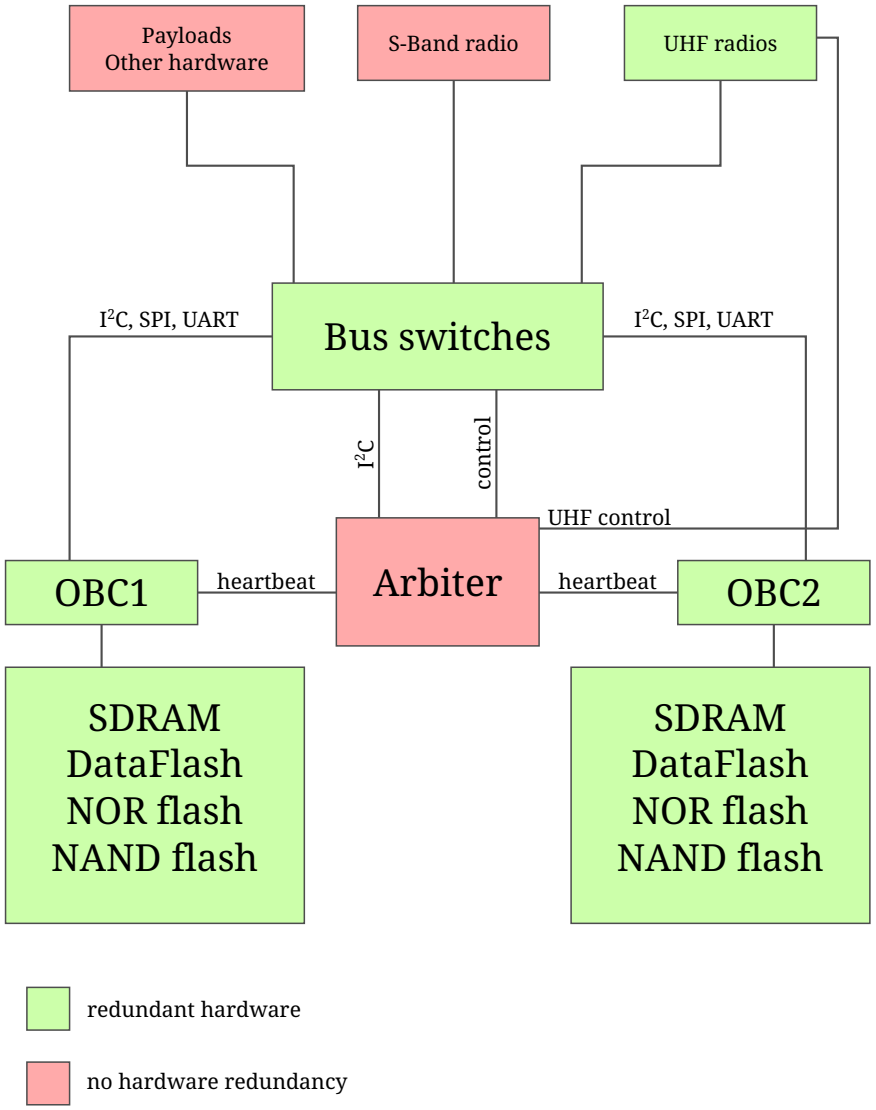

Figure 3.1: Simplified connectivity diagram

3.1.1 The OBC

The OBC CPU is Atmel AT91RM9200 [2], which is a microprocessor based on an ARM920T core. The hardware architecture includes two separate OBCs, so a hard failure in just one OBC does not cause mission failure. Only one OBC is active at any time, and a full reboot is required to switch the active OBC. Both OBCs include non-volatile NAND and NOR flash memories, and have an additional DataFlash memory connected to the SPI bus. In order to boot the system successfully, an OBC must have access to working boot and

root memories. The system supports booting from NOR flash or DataFlash, while the root memory can be mounted from NOR flash or NAND flash. The NAND flash is additionally divided into a larger primary filesystem and a smaller recovery filesystem. Therefore there is a total of 12 possible boot configurations, where each configuration includes the selected OBC, boot memory, and root memory. The root filesystems use the Unsorted Block Image File System (UBIFS), which is a filesystem intended for raw flash, and includes support for wear leveling. [27, 39]

The OBC hardware is not radiation hardened, although the NAND flash includes support for an error-correcting code (ECC). Therefore soft and hard errors are expected, and error detection and recovery need to be considered in the system design.

3.1.2 The Arbiter

A crucial component in enabling fault tolerance is the Arbiter, which is a MSP430-series microcontroller that selects which of the redundant components are in use. MSP430 [53] was chosen because it has already been used in Cubesats such as Delfi-C3 [54], Delfi-n3Xt [12], the Vermont Tech Cubesat [8] and WinCube [42]. The Arbiter is a single point of failure, but the model used in the OBC uses non-volatile ferroelectric RAM (FRAM) memory, which also supports an error-correcting code (ECC). Empirical tests [55, 56] indicate FRAM components perform well in radiation-harsh environments, and can be expected to be much more reliable than other RAM or flash components.

The Arbiter controls redundant hardware components by using General-purpose input/output (GPIO) lines to control which components are in use at any given time. GPIO lines with heartbeat signals from both OBCs are connected to the Arbiter, which monitors the OBCs and reacts if the current OBC heartbeat stops or is never activated after boot. [27]

3.1.2.1 Switching logic

The most important part of the Arbiter software is the switching logic, which determines whether the OBC boot configuration should be changed, resets the OBCs, and starts the selected OBC with the chosen configuration. Configuration parameters include OBC selection, boot memory, root memory, and UHF radio selection. The Arbiter stores the configuration and statistics of each OBC in a structure residing in non-volatile memory, so the Arbiter retains its state even if it is powered off. Therefore the switching logic will

not be reset even if the entire satellite is powered off by the Electrical Power System (EPS).

The switching procedure is always executed when the Arbiter powers up, and whenever the active OBC heartbeat stops. There is no way of commanding the Arbiter to run the switching procedure, but the OBC may request an OBC or UHF radio change directly. It is however possible to indirectly force the Arbiter to run the switching procedure by preventing the OBC heartbeat from starting at boot time.

When the switching procedure is executed, the Arbiter increases the failed boot counter by one, and clears the current heartbeat statistics. If the number of failed boots passes a large threshold, the Arbiter switches to the other OBC and its previous boot and root memory configuration. If the threshold is not reached, and the number of failed boots is low, the Arbiter will attempt to boot the same configuration without any changes. However, after a total of four failed boots, the Arbiter starts switching boot and root memories. Figure 3.2 summarizes the Arbiter switching logic.[1]

When booting the system, the Arbiter waits for at least 500 heartbeats before the system is considered to be working and the failed boot counter is cleared. Therefore even a successful boot will be considered a failure unless the system stays up for long enough to accumulate the required number of heartbeats.

3.1.3 Other important components

Another important component from the reliability perspective is the Electrical Power System (EPS), which has its own mechanisms that may power off the system in certain cases. For example, if the battery level decreases below a certain point, the EPS will shut down most systems until the battery has been charged back to nominal level. [16]

M. Siddique describes the power budget of Aalto-1 during different mission phases. The estimated power usage varies depending on which subsystems are in use, and power generation depends on several factors, including the position of the satellite relative to the sun, and the health of the solar cells. [44]

The OBC also includes two UHF radios, and the active radio can be switched by the Arbiter. Software running in the OBC can request a switch at any time by commanding the Arbiter.

The scientific payloads are not important from a reliability point of view, but are crucial for the mission objectives. Aalto-1 includes the Aalto Spectral

[1] Reference: Aalto-1 Arbiter source code, 2.1.2016

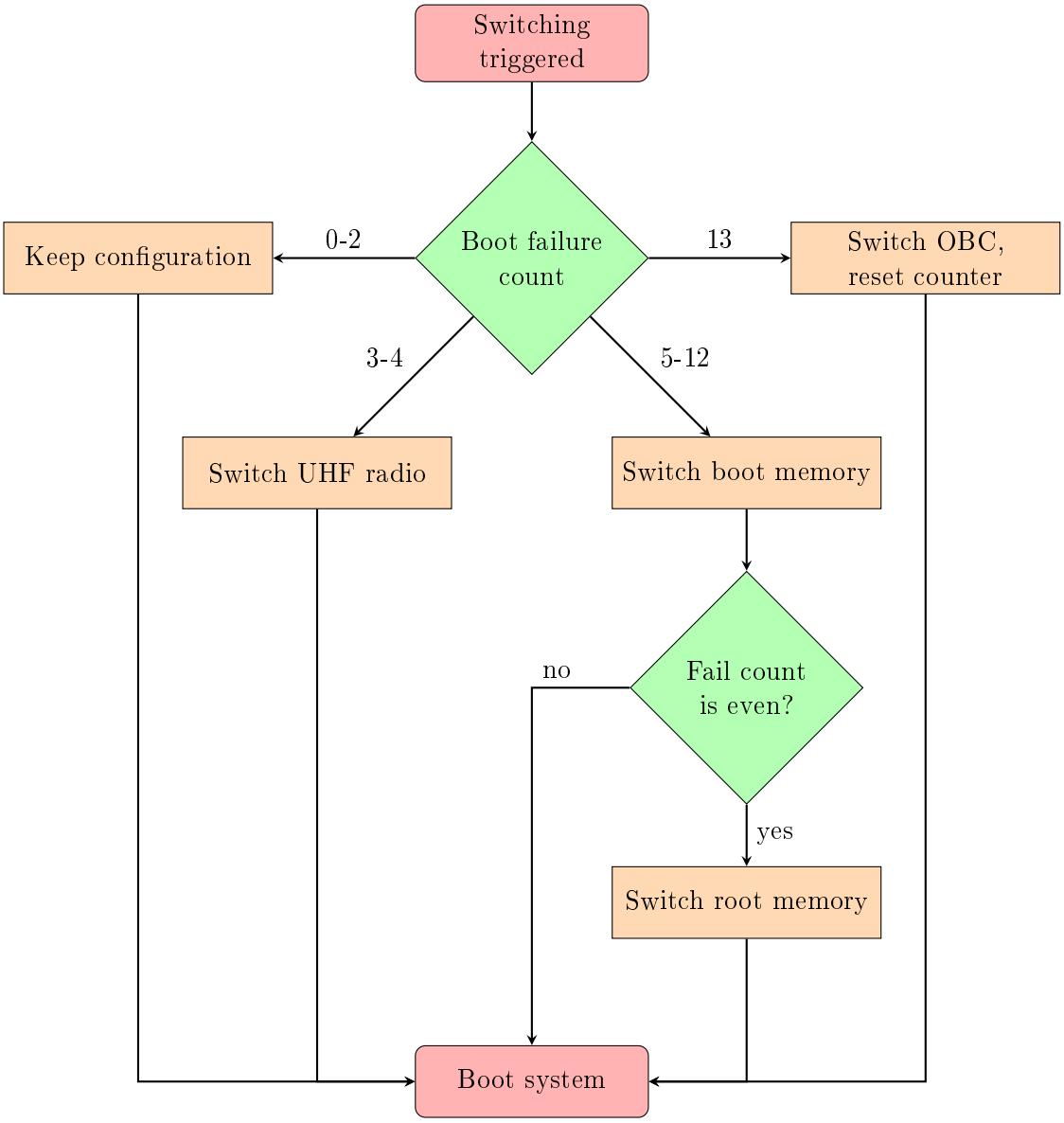

Figure 3.2: Arbiter switching logic

Imager (AaSI), Electrostatic Plasma Break (EPB), and Radiation Monitor (RADMON). [23]

3.2 Software platform

The OBC software platform is an embedded Linux platform based on Buildroot, which makes it easy to cross-compile a full embedded Linux system. Aalto-1 does not have hard realtime requirements, so a standard Linux kernel is used. Using Linux instead of other popular embedded operating systems, such as FreeRTOS, has several benefits. [40]

Firstly, Linux is widely supported in embedded contexts, so the availability of device drivers is good, and a plethora of existing software can be used instead of writing everything from scratch. Secondly, students are typically already familiar with Linux, so working with the development and production environments takes much less effort than with less commonly used operating systems. H. Sanmark [40] suggests that the use of simple UNIX commands reduces the need to create custom software for simple file handling tasks.

The OBC uses a standard Linux 3.4.106 kernel with several patches. For example, the kernel includes driver customizations to support the variable-length packets of the Aalto-1 I^2C protocol. This support can be toggled on and off by the userland. A watchdog driver for the AT91RM9200 board is also included in the kernel.

The system uses uClibc, which is a C standard library intended for embedded systems. It lacks several features present in more commonly used standard libraries such as glibc, but works well in an embedded context. [40]

Most standard Linux command-line tools are implemented by BusyBox, which is a single binary that implements minimal versions of several standard tools. The BusyBox binary is symbolically linked to several executable file names, and chooses the right behaviour depending on how it was invoked.

3.2.1 Fundamental custom software components

The most fundamental software component for the scientific mission is the communication software, because if communication is impossible, the satellite is no longer useful. The communication software is started on every boot, and is watched by a watchdog. It has exclusive access to the radio devices, and will request a UHF radio switch from the Arbiter if no communication has been observed in a certain time period. The communication software is described in detail in Chapter 6.

The system includes three separate watchdogs, and all of them need to be reset periodically, or the system will be reset. The watchdog mechanisms are described in detail in Chapter 4.

The OBC includes a scheduler, which is used to execute payload pro-

grams and other programs that need to be invoked based on a schedule. The scheduler is not analyzed in detail in this thesis, because it is not required for maintaining communication with the ground station. [45]

3.2.2 System logging and housekeeping

Logging and housekeeping facilities are fundamental in monitoring the health of the satellite. Most components of Aalto-1 use POSIX syslog facilities, which makes central collection of logs simple. Logs will be downlinked by the communication software when requested by the ground station.A dedicated housekeeping daemon program periodically collects telemetry data from the Arbiter, the EPS, the Attitude Determination and Control System (ADCS), and communication software.

Short downlinking opportunities and slow speeds limit how much logging and housekeeping data can be transferred per day. [23]

3.2.3 Software upgrades

Upgrading software components in space is risky, but it might be desirable if bug fixes or miscellaneous upgrades are needed. Instead of flashing entire filesystems, the plan in the Aalto-1 project is to transfer individual files, and to replace old files by renaming. Since there are several file systems in the satellite hardware, software upgrades might need to be done on all file systems.

3.3 Potential failure scenarios and design issues

3.3.1 Non-volatile memory corruption

Soft errors in non-volatile memories can cause corruption of files. The NAND memories in the OBCs support ECC, but they must be scrubbed periodically.

Since memories are not shared between OBCs, corruption in the cold-redundant OBCs memories cannot be detected or corrected without switching to the other OBC. This suggests that periodic switching might be necessary and should be scheduled during the mission lifetime.

Several software components in the system cause a significant amount of writes to the file system. Without proper wear leveling actively used areas in the non-volatile memory could become corrupted due to wear after excessive writing. However, Aalto-1 uses UBIFS which includes a wear-leveling layer

that spreads writes across the device, which alleviates this issue transparently to the main software running on the OBC.

3.3.2 Infinite loop in a non-critical program

An infinite loop can cause problems even if it happens in a non-critical program such as a payload program. The program might block others from using a shared resource, and would consume a significant amount of power and CPU time, which can result in drastic reduction in battery levels and eventually the EPS would power down the system. Taking significant CPU time can also distort system process scheduling times, which could in extreme cases cause timeouts or other failures in parts of the system which depend on timing.

3.3.3 Rogue system partition writes

The OBC uses a single root partition that is always writable. A software fault could cause a program to do unintended file writes or deletions. Some embedded systems prefer separate system and data partitions to reduce the risk of corrupting system files. Aalto-1 uses many shell scripts to perform tasks, and they must be thoroughly checked for errors. For example, a typical shell script error is removing a file and mistyping a variable name that is used as a prefix. Unless the system shell is configured to check variable names, running the command

```
rm -rf "$DIRCTORY"/
```

might attempt to remove everything in the root file system if the correct command was supposed to be

```
rm -rf "$DIRECTORY"/
```

Having multiple root file systems greatly reduces the impact of even the most catastrophic system partition writes. If the FDIR system works as intended, the system would eventually boot with another working file system if one exists.

3.3.4 Arbiter failures

While the Arbiter is resilient to radiation, it is still not impossible to have bitflips or hard errors. Also, the Arbiter software has been thoroughly tested but we cannot be fully sure that there will not be any software faults. The Arbiter is the single point of failure in the FDIR design, so issues with it can have serious consequences.

3.3.5 Full root file system

The OBC uses a single root file system, so it can easily become full. For example, a program might write a huge temporary file to /tmp, and fail to clean it up. The system must be able to boot at least a minimal communication loop even if the root file system is full. The boot script handles this scenario by using a pre-allocated file, so free disk space is not needed. If the file system is full, certain features in the system cease to work. For example, system logging will fail to append log messages to the log file.

3.3.6 Power loss

The EPS may power down the system at any point if the battery level goes too low. The Arbiter will be reset, but should not lose any state since all state is stored in non-volatile memory. The UBIFS filesystems in the OBC root memories use journals and are tolerant to power loss. Therefore a power loss should not cause any major issues in the system. If the cause for power loss was low battery levels, the EPS will automatically power on the system once the batteries have accumulated enough charge.

Chapter 4

Watchdogs and the boot procedure

The Aalto-1 satellite includes several system components that can be classified as watchdogs. A watchdog contains a timer, which is periodically reset by some other system component. If the timer is not reset before it reaches a predefined value, it is assumed that the watched component has failed, and the watchdog attempts to restore the system by performing some recovery actions.

The desired effect of the recovery action of all Aalto-1 watchdog systems is a full system restart, although there are some minor differences in how the watchdogs achieve the system restart and what kind of effects it has on the overall FDIR logic.

The main design goal is to ensure that at least one watchdog is watching the system at all times, and if a working communication loop is possible with some system configuration, the system will eventually converge to that working configuration.

Table 4.1 lists the watchdogs included in the Aalto-1 architecture.

Watchdog	**Watched component**	**Recovery action**
The Arbiter	OBC heartbeat	Arbiter logic and OBC reboot
Kernel watchdog	Userland watchdog	OBC reboot
Userland watchdog	Communication loop	Watchdog termination
EPS watchdog	I^2C bus	Power cycle

Table 4.1: Watchdogs in the Aalto-1 architecture

4.1 Watchdogs in previous nanosatellite projects

Watchdog timers have been used extensively in previous nanosatellite projects. Internal watchdogs in the OBC processors are often used, but sometimes additional external chips are added.

In the WinCube satellite the internal watchdog of the MSP430-based OBC was combined with an external MAX6814 watchdog integrated chip. The internal watchdog is straightforward to use, but having another watchdog adds an extra layer of protection. MAX6814 was chosen, because it does not use an oscillating crystal, which could be affected by temperature variations in space. [42]

The hardware architecture of the PilsenCube picosatellite somewhat resembles the architecture of Aalto-1. The satellite includes two OBC microcontrollers, and processor switching is performed by a dedicated hardware watchdog that includes a timer. [15]

J. Beningo reviewed watchdog architectures in CubeSat applications, and suggests that fault-tolerant CubeSat designs should in general be preferred to fault-avoiding designs due to resource constraints involved in CubeSat projects. [5]

4.2 The Arbiter and OBC heartbeat

The most important watchdog component in Aalto-1 is the Arbiter, which listens to the heartbeat signal of the currently selected OBC. The Arbiter controls the reset lines of both OBCs, so it can perform OBC switching, and reboot an OBC at any time. When the heartbeat stops or is never started after boot, an internal interrupt-based timer eventually triggers the Arbiter logic, which decides the next boot configuration and resets the OBCs. Triggering the Arbiter logic is therefore an important recovery action in the watchdog infrastructure, and the watchdogs and the boot procedure are designed to work together to trigger this logic. The system design could be simplified if triggering the Arbiter logic could be directly requested from the OBC, but unfortunately this feature was never implemented. [27]

The OBC heartbeat is implemented by using a standard Linux GPIO LED trigger driver (`ledtrig-heartbeat`). The kernel sends one initial heartbeat when the GPIO is initialized, but the continuously active LED trigger has to be activated from userland. This guarantees that the Arbiter will react to any failure during boot that prevents the userland from booting. Once activated, the heartbeat will be sent at regular intervals as long as the kernel is running normally. Once the heartbeat is active, only a kernel panic can

stop the heartbeat.

4.3 EPS watchdog and the I²C bus

The EPS also works as a watchdog in the system by monitoring the I²C bus, and power cycling the entire system if there is no bus activity for a couple of minutes. In normal scenarios frequent I²C activity is caused by processes such as housekeeping data collection, so the EPS should not restart the system unless the housekeeping daemon dies or the I²C bus becomes blocked for some reason. [16]

Since the EPS watchdog does not watch the communication software in any way, it can be considered a separate part of the FDIR architecture.

4.4 Improvements to the original design

The original FDIR design relied almost exclusively on the Arbiter. However, the only input to the Arbiter from the OBC is the heartbeat, which only signals that the Linux kernel is alive. In case of kernel panic, the heartbeat is stopped. However, the existence of the heartbeat is not sufficient to determine whether the system has a working communication loop. For example, the system might boot and start the kernel heartbeat, but the communication daemon fails immediately at startup. Therefore more watchdog mechanisms are needed.

4.4.1 Kernel watchdog and userland watchdog

The AT91RM9200 hardware used in the OBC includes a hardware watchdog, which is supported by the Linux kernel used in Aalto-1. The kernel watchdog can reboot the OBC, but rebooting will not be visible to the Arbiter, and therefore is not included in the statistics collected by the Arbiter and used in the Arbiter logic.

The kernel watchdog is not active by default, but it is activated by the userland watchdog program, which also resets the kernel watchdog periodically. If the userland watchdog terminates, the kernel watchdog is no longer reset and will eventually reboot the system. This also leads to a temporary stop in the OBC heartbeat, but if the system boots quickly back up, the Arbiter might not trigger its logic. Since the Arbiter is the only component in the system that can switch boot configurations, the system design must in some other way guarantee that reboots by the kernel watchdog will

eventually lead to the Arbiter logic. This is achieved by a forced heartbeat termination in the boot procedure after the system has been rebooted several times.

4.4.2 Userland watchdog and the communication loop

The userland watchdog is a simple daemon program that watches the update timestamp of a single file in the filesystem, and periodically resets the kernel watchdog. If the watched file has not been updated within a predetermined time interval, the watchdog terminates. Termination eventually leads to full system reset by the kernel watchdog. Source code of the fundamental functionality of the userland watchdog is included in Appendix A.

The watched file is touched in the communication software main loop by using the POSIX `utimensat` function. If the communication software terminates, crashes, or hangs, the file timestamp will not be updated and the watchdog will terminate.

The userland watchdog has an important role during the boot procedure. By default the OBC heartbeat is not enabled to guarantee that any kind of boot failure leads to a reset by the Arbiter. However, once the userland watchdog has started successfully, it also enables the heartbeat and takes over the main responsibility of guaranteeing a working communication loop. Since the heartbeat is enabled only after the userland watchdog is active, there is no time period during the boot procedure when a system hang would go undetected. This is a very important design detail, because the Arbiter cannot alone guarantee working communication, and is oblivious to problems if the heartbeat is still active.

4.4.3 Boot procedure

The boot procedure illustrated in Figure 4.1 has three possible outcomes:

1. **No heartbeat, which triggers a reset by the Arbiter**. Eventually the Arbiter will try different boot configurations.

2. **Normal boot**. The boot script starts the userland watchdog, the communication software, and the scheduler.

3. **Emergency boot**. The boot script starts the userland watchdog and the communication software in emergency mode.

Booting starts from the system bootloader, which loads the Linux kernel from the chosen boot memory and mounts the chosen root filesystem. Eventually the kernel starts the init program, which runs the boot script that

implements the userland side of the boot procedure. If the system does not successfully reach the boot procedure, the OBC heartbeat is never started and the Arbiter will eventually reboot the system and potentially switch the boot configuration.

The design distinquishes between normal and emergency boots, but this distinction is not used in the final flight model software. The original idea was to have a separate emergency mode where only the minimal software required to achieve the communication loop would be started.

The boot script that implements the boot procedure uses a three-byte preallocated file to store three separate unsigned counter values in non-volatile memory. Preallocation and the use of random access functions to update the file guarantee that the boot script works even if the root filesystem is full. It is important to remember that the boot counter values are specific to a single root filesystem, so if the Arbiter switches to a different OBC or a different root filesystem, different values are used.

4.4.3.1 Boot counters

The normal boot counter is a decreasing counter, which is never reset automatically. Therefore, after a certain number of reboots, normal mode is never attempted again without manual intervention.

When normal boots are no longer available, emergency boots are attempted if the counter is not zero. All software involved in an emergency boot should do a minimal amount of work to achieve a working communication loop, and must work even if the filesystem is full.

When emergency boots are no longer available, forced resets are attempted if the counter is not zero. The system halts execution and never starts the heartbeat, so the Arbiter will eventually reset the system. Forced resets are needed to force the Arbiter to eventually attempt other boot configurations, even if the reboots are performed by the kernel watchdog.

When forced resets are no longer available, the emergency counter and reset counter are reset to 20, and the system halts which eventually forces a reboot by the Arbiter. This action restarts the cycle in the logic, although normal boots will not be attempted again.

4.4.3.2 An example chain of events

Let us assume the communication daemon in a certain root filesystem is permanently broken and fails to start properly. Also, let us assume the system boot is very quick, so a reboot by the hardware watchdog is seen by the Arbiter as just a small gap in the heartbeat. Therefore, repeated booting by the

hardware watchdog is not enough to reliably trigger the Arbiter switching logic. If the system does not use any special boot procedure, the following chain of events may be observed.

1. A single heartbeat is sent as described in Section 4.2.
2. Userland watchdog is started, heartbeat is started.
3. Communication daemon is started and fails immediately.
4. Userland watchdog times out and kernel watchdog is no longer reset.
5. Kernel watchdog times out, reboots the system.
6. Go back to step 1.

It is possible the loop is terminated by some other system, such as the EPS, but the boot procedure has no direct way of breaking out of this endless loop that never enables working communication with the ground station. If we assume the boot procedure explained in Figure 4.1 is used in the system, the observed events are different. For simplicity, let us assume that all boot counters are initialized with the value 1.

1. A single heartbeat is sent as described in Section 4.2.
2. Normal boot counter is decreased and boot is attempted.
3. Kernel watchdog times out, reboots the system (steps 2-5 in previous list).
4. A single heartbeat is sent.
5. Since normal boot counter is 0, emergency boot counter is decreased and boot is attempted.
6. Kernel watchdog times out, reboots the system.
7. A single heartbeat is sent.
8. Since emergency boot counter is 0, forced resets counter is decreased and the heartbeat stops.
9. Arbiter switching logic is executed.

The only point where the execution of the Arbiter switching logic is guaranteed is step 9. The presence of the boot counters guarantees that repeated reboots will eventually lead to the execution of the Arbiter switching logic.

4.5 Potential failure scenarios and design issues

4.5.1 Boot failures and reboot loops

The causes and effects of boot failures have been carefully evaluated in the system design. If the system fails to boot for any reason, the Arbiter will reboot the system, and if the error persists, it is guaranteed to eventually try different boot configurations.

A more complicated scenario is an infinite reboot loop, where the system fails consistently, but the Arbiter fails to switch the boot configuration. The addition of forced reboots decreases the probability of this scenario, but it cannot be completely ruled out. For example, the Arbiter considers a boot successful, if it gets enough heartbeats after booting the system.

4.5.2 System crash or hang

Crashes of different components in the system have been carefully evaluated in the system design. All components except the Arbiter itself are watched by a watchdog mechanism. If the crash was caused by a programming error or a SEU, a reboot might be enough to fix the cause. If the issue is permanent and causes consistent crashes, the system will eventually try to switch to different boot configurations, which might make it possible to recover.

4.5.3 Infinite loop in communication software

A software error could lead to two kinds of infinite loops in the communication software. If the infinite loop is an inner loop in the main loop, the communication software stops touching the watch file, and the userland watchdog will exit eventually triggering the kernel watchdog. However, if the main loop gets stuck in an infinite loop that doesn't enable communication, it will still keep touching the watch file, and the watchdog mechanisms will not be able to detect the error condition. Therefore the communication software has to prefer fail-fast semantics whenever possible, and exit instead of ignoring critical errors. If the communication system fails to signal failure to the other FDIR components, recovery might not be possible.

4.5.4 Design complexity

One of the main issues with the boot procedure design is the use of repeated reboots in order to trigger the Arbiter logic. The same effect could be achieved in a simpler way if the Arbiter supported running the switching

logic by request. However, the repeated reboots are required, because the Arbiter code was frozen before the boot procedure was analyzed in detail, and new commands could not be added anymore once the need for forcing the Arbiter logic was understood. By forcing repeated reboots and making sure that the heartbeat is never started, the boot script can ensure that the Arbiter will run its logic eventually, but this design is not optimal.

In a better design all watchdogs would either be completely isolated, or would work towards a common goal. The EPS watchdog in Aalto-1 does not add much value, because it only watches secondary components in the system, and can disrupt the main watchdogs by rebooting at unpredictable times. If watching the I^2C bus is considered important, the userland watchdog could be very easily extended to watch more system resources. This would make I^2C errors to use the same recovery mechanism as the other watchdogs in the system.

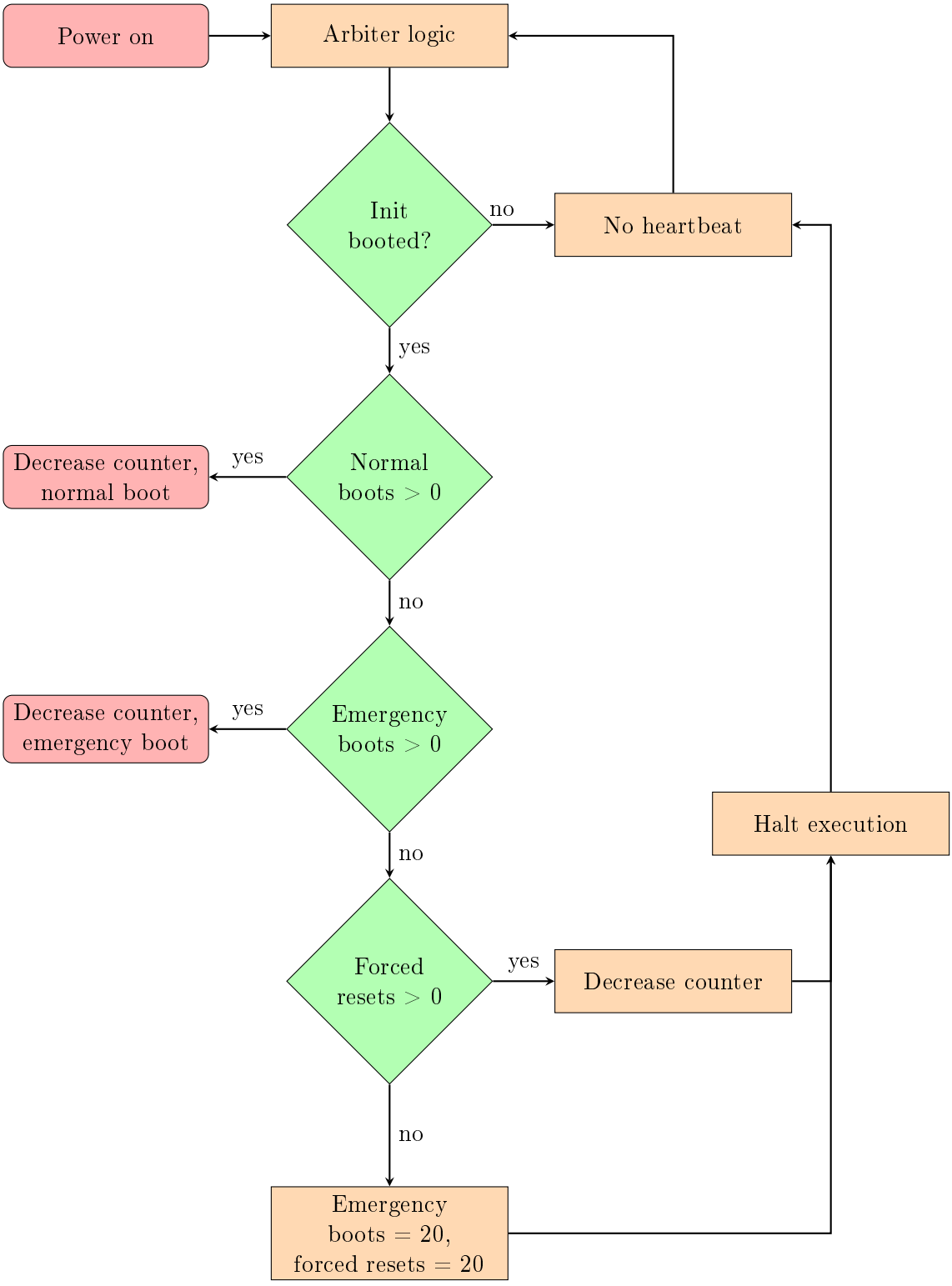

Figure 4.1: Improved boot procedure

Chapter 5

Aalto-1 I^{2}C protocol and library

I^{2}C [37] is a popular 2-wire bus used for connecting integrated circuits to each other. The byte-oriented protocol is simple, and does not have support for high-level reliability-related features like checksums. In a nanosatellite reliability of internal communication is desirable, so it might be necessary to add a higher level protocol on top of I^{2}C if possible. However, supporting an extended protocol requires modifying the devices connected to the bus, so it is only feasible if custom devices are used.

In Aalto-1, the OBC uses I^{2}C to communicate with AaSI, the Arbiter, ADCS, Electrostatic Plasma Brake (EPB), EPS, the Real-Time Clock (RTC), and a temperature sensor. AaSI, EPB, and the Arbiter support an enhanced custom protocol, while the other devices use the standard low-level I^{2}C protocol. [46]

An I^{2}C bus may have multiple masters, but in Aalto-1 the OBC is the only master in order to eliminate the need for clock synchronization and bus arbitration. Therefore all communication is always started by the OBC, and the other devices simply react to commands, and may also return a response to the OBC. [38]

5.1 I^{2}C in previous nanosatellite projects

Some of the previous nanosatellite projects that have used I^{2}C include Delfi-n3Xt and several Cal Poly CubeSats. A survey [7] conducted by
J. Bouwmeester and J. Guo suggested that most nanosatellites with a shared data bus use I^{2}C.

In the Dutch Delfi-n3Xt satellite I^{2}C was chosen over CAN for power, availability, and familiarity reasons. The protocol was not extended, but the hardware architecture of the system includes additional bus protector circuits

to reduce the risk of rogue devices hanging the bus indefinitely [10, 12]

CP2, also known by its in-orbit name CP4, was the first Cal Poly satellite to use the CPX bus, which is an internal data bus based on I²C. A high-level protocol was added and the I²C library software for PIC microcontrollers was replaced with a more robust software implementation. Even with these improvements, CP2 failed in orbit three months after launch. The reason for the failure is not fully known, but one plausible explanation is a blocked I²C bus caused by a rogue device. [14, 24]

Cal Poly CP6 included several changes to the I²C-based CPX bus. CRC checksums were added to the protocol, which reduces the chance of bit flips causing serious errors [32]. Error counts were included in telemetry collection, and data collected over the lifetime of CP6 suggests an estimated bus transaction error rate of 8% [28].

5.2 Aalto-1 I²C protocol

The Aalto-1 I²C protocol is a packet-based protocol layered on top of the low-level I²C protocol. It provides support for variable-length packets, packet checksums, and includes a standard way for specifying commands and status bytes. A single packet can be up to 257 bytes long, and consists of 4 bytes of protocol fields, and up to 253 bytes of data. Table 5.1 describes the packet structure. [41]

Field	Size (bytes)	Description
Packet length	1	Total length of command, status, and data
Command	1	Device-specific command code
Status	1	Command execution status
Data	0-253	Additional data
Checksum	1	8-bit XOR of preceding bytes
Total	4-257	

Table 5.1: Aalto-1 I²C protocol packet structure

In Aalto-1, the OBC is the only master, while all other devices are slaves. A single I²C transaction can be a one-way command or a two-way request and response pair. Checksums are calculated on both sides for all sent packets, and the receiving end verifies the checksum before accepting a packet. In addition to checksum verification, command and status bytes are checked.

A one-way transaction is a simple command to a slave device, and no response or any kind of acknowledgement is sent back. Therefore one-way

commands are not guaranteed to always be reliable, since the protocol can only guarantee the integrity of the command packet, but invalid packets will simply be dropped and not even retried, since by definition the device will not be able to send a response to the OBC to request a retry.

When performing a two-way transaction, the OBC sends a request and expects a response back. A response packet in a two-way transaction is expected to have the same command byte as in the request. Reading the response is retried three times, but writes are not automatically retried. Therefore it is the application's responsibility to retry the high-level operation in failure scenarios.

The software implementation is mostly in the user-space `libicp` library described in the next section. However, since the low-level I²C protocol start and stop sequences are sent by the kernel driver, a kernel patch was needed to add support for variable-length packets. Variable-length support is enabled by using a custom `ioctl` call, which is done automatically by the user-space library when extended protocol packets are sent. When raw I²C is used to communicate with ADCS or EPS, variable-length support is automatically disabled, so programs can freely mix standard and extended protocol calls without manually switching protocols.

Table 5.2 describes an example packet from the OBC to AaSI. The command is an AaSI-specific extended text command that requests the current value of sensor 1 register 1. Table 5.3 describes an example response packet from AaSI that contains the successful register value 7.

Field	**Data (in hex)**	**Description**
Packet length	`06`	
Command	`43`	AASI_CMD_TEXT (extended text command)
Status	`00`	Requests have status `0x00`
Data	`53 31 2C 31`	"S1,1"
Checksum	`3B`	

Table 5.2: Example Aalto-1 I²C request packet

5.3 libicp shared library

The `libicp` shared library provides high-level functions for internal communication, including functions for accessing I²C devices from user-space. The library uses directly the `/dev/i2c-0` device exposed by the kernel. The device and the physical I²C bus are shared and `libicp` calls may be done by

Field	Data (in hex)	Description
Packet length	08	
Command	43	AASI_CMD_TEXT (extended text command)
Status	00	
Data	53 31 2C 31 2C 37	"S1,1,7"
Checksum	2F	

Table 5.3: Example Aalto-1 I²C response packet

multiple programs, so a named POSIX semaphore is used to prevent simultaneous I²C transactions from interfering with each other. The semaphore is acquired and released within `libicp` calls, so it is not possible for a library client to misuse the semaphore mechanism in normal conditions. While the semaphore is locked, a special signal handler is temporarily installed to detect POSIX SIGINT and SIGTERM signals. The signals are caught, delayed, and propagated only after the semaphore has been unlocked, so the semaphore is not accidentally left in a locked state. Other signals and error scenarios can still lead to issues with the semaphore. The library uses POSIX `read` and `write` functions and does not use the POSIX O_NONBLOCK flag, so the calls are blocking calls.[1]

The most commonly used function in the library is i2c_send_d, which is used to send an Aalto-1 I²C request packet and read a corresponding response. An extra parameter *msdelay* can be used to insert a millisecond-precision sleep between the request and response to give the target device time to react to the request. Reading the response is retried up to three times if a general read I/O error is detected, but a protocol verification error, such as an invalid checksum, will immediately terminate the call and it is up to the caller to retry the entire operation if necessary.

5.4 Improvements to the original design

Minor Aalto-1 I²C protocol violations in AaSI were found in a late project phase. For example, the protocol requires that the command field in the response packet is equal to the command field in the request packet, but this is untrue for a certain AaSI command. Therefore a new function with a quirks parameter was added to `libicp`. The quirks parameter is a bit field that specifies which protocol validity checks should be skipped when processing

[1]Reference: Aalto-1 libicp library source code, 2.1.2016

Function	Purpose
i2c_send	Aalto-1 I²C protocol request and response
i2c_send_d	Like i2c_send, but with an extra parameter *long msdelay*
i2c_send_dq	Like i2c_send_d, but with an extra parameter *int quirks*
i2c_send_only	Aalto-1 I²C protocol request
i2c_send_raw	Raw I²C protocol request and response
i2c_send_raw_d	Like i2c_send_raw, but with an extra parameter *long msdelay*

Table 5.4: Fundamental libicp I²C functions

the response packet. This mechanism documents known violations at call sites in the source code, and allows other error checks to remain in place.

5.5 Potential failure scenarios and design issues

5.5.1 Radiation-induced errors in I²C devices

Bit flips in the I²C bus can corrupt data packets sent between devices. The extended Aalto-1 protocol uses a packet checksum, which reduces the chances of undetected bit flip errors. However, the protocol and the `libicp` library do not provide a consistent and heavily tested recovery mechanism, so the effect of a bit flip depends on the application and the device it is communicating with. An application with fail-fast behaviour terminates immediately when an unexpected error occurs, which is acceptable if other recovery mechanisms such as watchdogs are working as expected. In the worst case an application will ignore an error, and continue even though the application or the device may be in a wrong state.

The raw I²C protocol has no error detection or recovery mechanisms, so many kinds of errors are possible. Further analysis would be needed to determine what kind of errors corrupted packets could cause in practice. For example, sending the wrong commands to the EPS could in theory cause severe problems, but this is difficult to determine without studying the EPS software in detail.

The I²C bus is a shared bus and in Aalto-1 includes a wide variety of components with different expected levels of radiation resistance, so hard errors in a single device can in the worst case prevent all other devices from using the bus correctly. The effects of such scenario could be slightly reduced

by dividing critical and non-critical components to separate shared buses if possible.

5.5.2 Weak XOR checksum effectiveness

The checksum field in packets makes the Aalto-1 I²C protocol more reliable than the raw protocol in a nanosatellite environment. However, a XOR checksum is not very effective in detecting errors that affect more than one bit. For example, if a certain bit position is permanently stuck with a certain value, a XOR checksum can fail to detect the error in all even-sized checksum blocks.

T. Maxino investigated the effectiveness of different checksum algorithms in the context of embedded networks [31]. XOR had the lowest efficiency, but also the smallest compute cost. However, a simple two's complement checksum was found to be nearly twice as effective as XOR with very little or no change in compute cost. More advanced algorithms, such as CRC are significantly more effective, but also require a greater compute cost.

For nanosatellite I²C communication, a two's complement checksum would be a better checksum than XOR, and even CRC-based checksums could be considered if the increased compute cost is acceptable. K. McCabe [32] modified the CPX nanosatellite I²C bus to use 8-bit CRC, so it has already been used in an existing nanosatellite project. A checksum longer than 8 bits could be used to provide even better resilience to errors. T. Maxino [31] suggests that doubling the checksum size can halve the probability of undetected errors.

5.5.3 Semaphore abandonment

A named POSIX binary semaphore can become abandoned if it is locked by a process but never unlocked. Even if the process crashes, no automatic unlocking is performed. Therefore a binary semaphore may become abandoned and no other process can unlock it, unless some recovery mechanism is implemented. It is possible to call `sem_post` from a different process, but it is a challenging task to implement the required signaling between processes to support recovery without compromising the correct use of the semaphore.

The temporary POSIX signal handler used by `libicp` avoids this issue in case of `SIGINT` and `SIGKILL` signals, but other signals and many kind of error conditions involving a crash or a hang are still possible and can completely block the use of I²C in the system. The EPS watchdog will eventually reset the system since no I²C communication will be possible, but the timeout is

fairly large, and while the semaphore in question is cleared, the issue may appear again after rebooting the system.

One way to avoid the issue would have been to implement `libicp` as a daemon instead of a library. A daemon could easily serialize access to the I²C bus without requiring complicated signaling between multiple processes. For example, a normal semaphore could be used, and crashes in client programs or the daemon itself would not result in any permanent locking of the bus.

Another option would have been to consider the use of shared robust pthread mutexes instead of POSIX semaphores. A pthread mutex can be placed in shared memory and used in a robust manner in multiple processes as long as `PTHREAD_PROCESS_SHARED` and `PTHREAD_MUTEX_ROBUST` are set. If a process locks the mutex but terminates for any reason, a robust mutex guarantees that some other thread will get the return value `EOWNERDEAD`, and can mark the mutex as consistent and recover it. Since no actual state would be guarded in `libicp`, the mutex could be marked as consistent without any extra work. [21]

5.5.4 Blocking I/O

All reads and writes done by `libicp` use blocking calls, so there is no support for any kind of timeouts. Therefore an I²C protocol read can in theory block the calling process indefinitely. Since writes are never retried, it is possible that the calling process will never wake up if the slave device does not send anything back. A better option would be to use something that supports timeouts. For example, the POSIX `select` function could be used to read the response.

If a broken slave device blocks all I²C communication permanently, the system can still boot, but the EPS watchdog will not be reset. Therefore periodical reboots can be expected in this scenario.

Chapter 6

Aalto-1 communication software

Aalto-1 uses two radios for communicating with the ground station. The primary channel uses UHF, but S-band can also be used as a secondary channel for downlinking purposes. S-band has a much higher bandwidth, which is useful, because the satellite can generate a significant amount of data that needs to be downlinked. [22]

The UHF radio uses omnidirectional antennas, so it is not dependent on a specific orientation of the satellite, and can function even when the ADCS system is malfunctioning. [44]

This chapter focuses on the UHF radio and related protocols, because the satellite can still function if the S-band radio stops working.

6.1 Communication protocols in previous nanosatellite projects

A survey [33] conducted by P. Muri and J. McNair found the AX.25 and CW protocols to be the most widely used communication protocols in CubeSat projects. Many satellites use amateur radio frequencies, and AX.25 is a popular protocol among amateur radio operators.

The CubeSat Space Protocol (CSP) [26] is a protocol that can be used transparently for both internal and external communication in a CubeSat. The protocol has been designed to be usable with even 8-bit microcontrollers, and could be used to implement a service-oriented architecture in a nanosatellite. Unfortunately the CSP 1.0 protocol does not include protection for the packet headers, which is a fundamental design flaw from a reliability point of view.

6.2 Communication protocols

The main goal of the communication system is to transport packets between the ground station and the satellite. Packets sent from either the satellite or ground station software use a high-level protocol layered on top of a low-level protocol. From a reliability point of view we are above all interested in error detection and correction, so we focus here on protocol reliability, not on low-level radio communication details.

Figure 6.1 illustrates how a TC packet from the ground station software is transported to the OBC software, and how a TM packet from the OBC software is transported to the ground station software. Table 6.1 summarizes the used protocols and packet verification mechanisms, such as checksums and hashes.

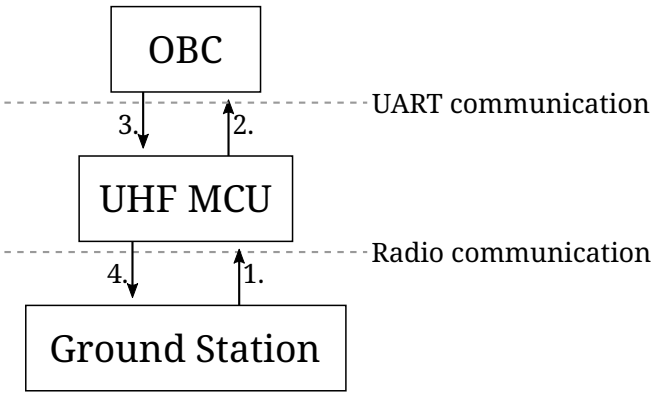

Figure 6.1: Packet transportation for a TC request and TM response

	Protocols	**Packet verification**
1.	Aalto-1 high-level, Aalto-1 low-level	HMAC, CRC-16
2.	Aalto-1 high-level, libicp	HMAC, XOR
3.	Aalto-1 high-level, libicp	XOR
4.	Aalto-1 high-level, Aalto-1 low-level	CRC-16 (unchecked)

Table 6.1: Protocols and packet verification in Figure 6.1

6.2.1 Low-level packet transportation

Aalto-1 includes support for AX.25 as the link-layer protocol in both the satellite-side software and the ground station software. However, by default

a custom protocol is used, and at the time of writing this thesis, AX.25 support was not actively used.

The UHF radio on the satellite side uses a transceiver chip from Texas Instruments, which provides hardware support for low-level concerns such as synchronization words. It also supports CRC-16 checksums, which can be used to verify the validity of received packets. The transceiver chip is connected to a radio microcontroller, which controls the transceiver and communicates with the OBC using an UART connection. If an invalid packet is detected, an error message is sent to the OBC. Successful packets are sent using the UART connection, and are received by the communication software running on the OBC. An internal communication protocol similar to the Aalto-1 I^2C protocol is used in UART communication, so errors such as bitflips can be detected using the XOR checksum.

On the ground station the low-level communication protocol is implemented with custom software. The ground station software adds a CRC-16 checksum to sent TC packets, and on the satellite side the UHF radio microcontroller verifies the checksum.

When a TM packet is sent from the satellite, the satellite communication software sends it using UART to the UHF radio microcontroller, which adds a CRC-16 checksum to the packet. The ground station radio software receives the packet and calculates the checksum, but at the time of writing this thesis, the checksum was not validated. Therefore, bit errors in TM packets will not be detected at the radio software level.[1]

6.2.2 Application-level protocol

The low-level protocol enables point-to-point packet transmission, but does not provide multiplexing or other mechanisms for differentiating between packet types, sources, or destinations. The high-level protocol is used at the communication software level once a packet has been successfully transported using the low-level protocol. Table 6.2 describes the packet structure in the high-level protocol.

The high-level communication protocol also includes a keyed-hash message authentication code (HMAC) based on a SHA-256 hash from the Secure Hash Algorithm 2 (SHA-2) family. The ground-station software calculates a HMAC code for each TC packet with a shared secret, and appends the HMAC bytes to the packet data field. Therefore the authenticity of the TC packets sent to the satellite can be verified. However, TM packets originating from the satellite do not include the HMAC code, so the ground-station

[1] Reference: Aalto-1 source code, 2.1.2016

cannot verify the authenticity of incoming packets.

The service and subservice fields enable multiplexing of packets, so they are in many ways similar to ports in the widely used UDP protocol. However, the service and subservice are never dynamic, and the supported values are fixed in advance. Each service exists only once in the system, so a stateful service such as command execution uses global state and only supports one execution at a time.

Field	Size (bytes)	Description
Packet type	1	0x10 = Telemetry, 0x11 = Telecommand
Sequence	2	Unsigned 16-bit sequence number
Service	1	Service number
Subservice	1	Subservice number
Data	0-247	Packet data
Total	5-252	

Table 6.2: Aalto-1 application-level communication protocol packet structure

6.3 Communication daemon in the satellite

A communication software daemon is running at all times in the satellite. The main loop of the program listens for radio packets from the UHF MCU using the POSIX select function. The HMAC of incoming packets is checked, and valid packets are passed to service-specific handler functions.

The system tracks and stores in non-volatile memory a timestamp of last valid communication with the ground station. When the system detects that the timestamp has not been updated in a predefined interval, the communication software attempts to reset the currently used UHF radio by requesting a radio swtich from the Arbiter, and then quickly switching back to the original radio. If the timestamp is still not updated after a longer interval, the system requests a UHF radio switch from the Arbiter. The timestamp is updated when a valid packet from the UHF MCU is received, but this is regardless of the HMAC check. Therefore packets which fail authentication will still update the timestamp and can delay potential UHF radio switching.

6.4 Potential failure scenarios and design issues

6.4.1 Infinite failure loop

The ultimate goal of most reliability-related features in the satellite is to keep the communication software main loop running. However, even if the main loop is running, it does not necessarily guarantee that the satellite is capable of meaningful communication with the ground station. Therefore it is highly important that the communication software handles errors instead of ignoring them and prefers fail-fast behaviour.

An infinite main loop that does not enable successful communication is a very bad scenario, because none of the watchdogs will reset the system as long as the communication loop keeps resetting the userland watchdog.

6.4.2 UHF radio breakage

A UHF radio can break during flight, but the Aalto-1 nanosatellite includes two radios, so the other radio can still be operational. Switching is primarily done by the communication daemon, but the Arbiter switching logic can also switch the UHF radio. The main weakness in the radio switching is the reliance on timestamps stored in the root memory. If the satellite RTC fails even temporarily, the timestamps might not be reliable between reboots. Also, if the Arbiter switches to a different root memory, the previous timestamp information is no longer available.

6.4.3 Weaknesses in checksums and authentication

The architecture includes several ways to detect packet validity, but it also has weaknesses. The XOR checksum used in the UART communication has the weakness mentioned in Chapter 5, and TM packets do not have sufficient packet validity checks. It is tempting to think that TM packets do not need to be checked if they are simply data that is collected, such as telemetry data. However, the data in TM packets can also used for decision making, and incorrect data could lead to incorrect action, either by a human operator or an automated process.

6.4.4 Weaknesses in recovery from errors

The architecture includes ways to detect errors, but error recovery is often missing or incomplete. For example, a TC packet with an invalid CRC-16 checksum is silently dropped, and the ground station must detect the

failure by the use of a timeout. However, since TM packets can also be unreliable, the ground station cannot always assume that the TC packet was not received. TC packets can cause non-idempotent side-effects in the satellite, so resending TC packets in all timeout scenarios could result in undesired effects in the satellite.

6.4.5 Risky software development process

The communication software in Aalto-1 was one of the last software components to be finished. Significant changes were made very late in the project, and there is no thorough automated test suite that could be used to verify the correctness of refactoring or bigger changes. These factors together add a lot of uncertainty to the communication protocols and software. Manual testing has been used to confirm that everything works as expected in normal conditions, but it is not sufficient to achieve high confidence in the components.

Chapter 7

Discussion

Nanosatellite projects attempt to make good compromises in hardware and software choices in order to drive down the costs and minimize barriers of entry for research. Choosing the right hardware and software for a mission is challenging, and data gathered from existing research is useful in guiding the selection process. An MSP430 MCU is a popular device that has been included in many successful CubeSat projects, but it would be interesting to see even more variety in nanosatellite hardware and to have empirical results of how well different devices work in space.

One highly interesting research area is the study of radiation effects on COTS hardware. There are already papers that have studied simple hardware such as 16-bit MCUs, but it would be useful to have access to more data about powerful hardware, such as 32-bit ARM processors. For example, it would be interesting to see comprehensive radiation tolerance data about Freescale iMX6 or Qualcomm Snapdragon processors. Unfortunately many hardware vendors don't have publicly available information about radiation resistance of their devices.

On the software side important criteria include approachability and familiarity to students, software complexity, and hardware requirements. A Linux operating system is most likely more familiar to students instead of alternatives such as FreeRTOS, unless the students are specifically studying embedded software. It would be interesting to have more empirical results of standard Linux operating systems in CubeSat projects. Due to communication constraints a nanosatellite cannot be as approachable as a simple embedded computer such as Raspberry Pi running Linux, but the benefits of having a familiar system should not be underestimated.

H. Sanmark describes in his thesis many fundamental challenges faced in the development of Aalto-1 nanosatellite software. Communication between groups developing independent components, rapid changes in requirements,

and lack of experience are significant complicating factors. These factors also make it difficult to define a clear and appropriate software development process for nanosatellite projects. [40]

Several CubeSats have been launched, and many more are currently under development. However, it seems that most projects write their own software and there is very little sharing involved, except for satellites sharing a common lineage. The CubeSat Space Protocol project is a good example of a public open source project that can benefit multiple CubeSat projects. It would be useful to have more projects like it to faciliate nanosatellite software development through code reuse. This could be especially useful for software components that benefit from rigorous testing, such as communication protocols.

More powerful hardware means more flexibility in software choices. If the software does not have tight performance requirements and does not need access to low-level features of C, many software components could be written using a different language more suitable for general-purpose programming. For example, dynamic programming languages such as Python or Javascript are much more approachable than C, and their performance characteristics are most likely acceptable if the hardware is powerful enough. Javascript is sometimes thought to be a programming language reserved for web browsers, but platforms such as Node.js make it a truly general-purpose programming language.

P. Niemelä recognizes the importance of software testing in nanosatellite projects. A core challenge in testing embedded software is the effect of the runtime environment and hardware to the software. For example, software might be developed in parallel with the hardware, so testing the software in a real hardware environment can be difficult or impossible. In Aalto-1 the use of Linux made it possible to perform some tests on development machines without access to real hardware. [36]

The Rust programming language could be a good candidate for nanosatellite software, because it is a systems programming language like C, but offers several language features that attempt to eliminate many common errors typically found in C code. For example, safe Rust code includes memory safety guarantees that help eliminate errors such as dangling pointers and buffer overflows. Rust is a very new programming language, so at the moment students are most likely not familiar with it, but this might change in a couple of years if Rust becomes more widely adopted. The performance of Rust is similar to C and C++, and it has been used to implement highly efficient software, such as zero copy parsers [11].

Using high-level programming languages can reduce the risk of low-level programmer errors, but many ideas could also be adopted from safety-critical

software to further improve the reliability of nanosatellite software. Safety-critical design processes such as STPA [47] could be used when designing a software and hardware architecture for a nanosatellite. An important detail that might be forgotten in nanosatellite reliability is the possibility of human error. It is very easy to focus on space as an environment when assessing the reliability, but a human operator error can also lead to a mission failure.

In addition to using modern programming languages and design processes, formal tools could be used to verify correctness of nanosatellite software components. The CubeSat Space Protocol has been used in several CubeSats, but it seems that no proof of correctness has been performed. G. Holzmann [20] recommends that newly designed protocols should be treated with suspicion until their correctness has been proved.

Protocols are perhaps the most obvious candidates for verification, but any kind of program behaviour, especially of critical components, could be verified. In a simple case program behaviour could be modeled as a state machine, and a model checker could be used to verify assumptions about the model. X. Gan, J. Dubrovin, K. Heljanko [18] used symbolic model checking to verify a a satellite attitude and orbit control system. Verifying software components with such tools and methodologies could improve the reliability of a nanosatellite.

A core challenge in building reliable software components is integration with other components and the runtime environment. For example, the original design of a reliability-related component such as a watchdog may seem good if the component is inspected in isolation. However, when looking at the big picture and how components interact, it might be possible to find gaps in the design. In some cases several components may even compete with each other even though the components may work well in isolation. This necessitates evaluation and inspection of components from both low-level and high-level points of view.

Chapter 8

Conclusions

The goal of this thesis was to evaluate important parts of Aalto-1 nanosatellite software from a reliability point of view. Building a reliable satellite is a difficult task, and requires careful design considerations in both hardware and software. Since communication with failed satellites might be limited or impossible, evaluation and analysis of the hardware and software components is important. Evaluation results together with data collected during the mission can be used to improve the design, implementation, and working processes of future nanosatellite projects. The failure rate of 50% in university nanosatellite projects suggests that there are still many potential research topics and improvement areas in the field. Finding a good compromise between approachability to students and the use of robust but difficult techniques and tools is a fundamental challenge, which is one major area of difference between industrial and educational projects.

We have evaluated several critical components of the Aalto-1 software architecture, and added improvements to the original design. Reliability has clearly been considered in the major areas of the architecture, but there is still room for improvement. Some parts of the architecture have some unnecessary complexity, and there are also some known flaws. The communication software is an extremely important software component, and while it may not be a brittle component in practice, the evaluation pointed out several uncertainties in the design and implementation. The probability of having such uncertainties could be reduced in future projects by building the communication software and having end-to-end tests for it as early as possible in the project.

Potential future research topics include robust communication protocols and managing hardware redundancy and fault detection in a Linux nanosatellite. The field could also benefit from more active sharing of built software. However, simply sharing source code is not enough, and shared software

should be built and documented with the assumption that other projects might be interested in reusing the software. Critical software components such as communication protocols could also benefit from formal verification.

Even with some flaws in the design, the future of Aalto-1 looks bright. The time after the soon approaching launch will be the final validation of the design and implementation of the satellite. Nanosatellite reliability is an active research topic, and regardless of the mission result of Aalto-1, future projects can benefit from the contributions of Aalto-1 and this thesis.

Bibliography

[1] *MISRA C: Guidelines for the Use of the C Language in Critical Systems*. Motor Industry Research Association, 2013. ISBN 978-1906400101.

[2] Atmel. ARM920T-based microcontroller AT91RM9200, revision 1768i-atarm–09-jul-09, July 2009. URL `http://www.atmel.com/images/doc1768.pdf`.

[3] R. Barry et al. FreeRTOS. URL `http://www.freertos.org`. Accessed on January 24th, 2016.

[4] R. C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on*, 5(3):305–316, 2005. ISSN 1530-4388. doi:10.1109/TDMR.2005.853449.

[5] J. Beningo. A review of watchdog architectures and their application to Cubesats, April 2010. URL `http://www.beningo.com/wp-content/uploads/images/Papers/WatchdogArchitectureReview.pdf`.

[6] N. W. Bergmann and A. S. Dawood. Reconfigurable computers in space: problems, solutions and future directions. In *The 2nd Annual Military and Aerospace Applications of Programmable Logic Devices (MAPLD'99) Conference*, 1999.

[7] J. Bouwmeester and J. Guo. Survey of worldwide pico- and nanosatellite missions, distributions and subsystem technology. *Acta Astronautica*, 67(7–8):854–862, 2010. ISSN 0094-5765. doi:10.1016/j.actaastro.2010.06.004.

[8] C. Brandon. Use of Ada in a student CubeSat project. *Ada User Journal*, 29(3):213–216, 2008.

[9] California Polytechnic State University. CubeSat design specification, rev 13. 2014.

[10] N. E. Cornejo, J. Bouwmeester, and G. N. Gaydadjiev. Implementation of a reliable data bus for the Delfi nanosatellite programme. In *Small Satellites for Earth Observation: 7th International Symposium of the International Academy of Astronautics (IAA), 4-8 May 2009, Berlin, Germany*, 2009.

[11] G. Couprie. Nom, A byte oriented, streaming, zero copy, parser combinators library in Rust. In *2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, May 21-22, 2015*, pages 142–148. IEEE Computer Society, 2015. ISBN 978-1-4799-9933-0. doi:10.1109/SPW.2015.31.

[12] S. De Jong, G. T. Aalbers, and J. Bouwmeester. Improved command and data handling system for the Delfi-n3Xt nanosatellite. In *59th International Astronautical Congress: IAC 2008, 29 September-3 October 2008, Glasgow, Scotland*, 2008. ISBN 978-1-6156-7160-1.

[13] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, 2009. ISSN 0018-9162. doi:10.1109/MC.2009.118.

[14] J. Farkas. CPX: Design of a standard CubeSat software bus. *California Polytechnic State University, California, USA*, 2005.

[15] P. Fiala and A. Vobornik. Embedded microcontroller system for PilsenCUBE picosatellite. In *16th IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems, DDECS 2013, Karlovy Vary, Czech Republic, April 8-10, 2013*, pages 131–134. IEEE Computer Society, 2013. ISBN 978-1-4673-6135-4. doi:10.1109/DDECS.2013.6549804.

[16] J. Finnholm, J. Hemmo, and T. Nikkanen. Aalto-1 design of the electrical power system, 2013. A1-OBC-DD-02-v2.

[17] P. Fortescue, J. Stark, and G. Swinerd. *Spacecraft Systems Engineering*. Wiley, 4th edition, 2011. ISBN 978-0-470-75012-4.

[18] X. Gan, J. Dubrovin, and K. Heljanko. A symbolic model checking approach to verifying satellite onboard software. *Science of Computer Programming*, 82:44–55, 2014. doi:10.1016/j.scico.2013.03.005.

[19] L. Hatton. Safer language subsets: an overview and a case history, MISRA C. *Information & Software Technology*, 46(7):465–472, 2004. ISSN 0950-5849. doi:10.1016/j.infsof.2003.09.016.

[20] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Inc., 1991. ISBN 0-13-539925-4.

[21] IEEE. *1003.1 Standard for Information Technology - Portable Operating System Interface (POSIX) System Interfaces, Issue 6*. 2001.

[22] A. Kestilä, A. Näsilä, M. Komu, J. Praks, and A. Hakkarainen. Aalto-1 experiment interface document, 2013. A1-SYS-EID-01-v7.

[23] A. Kestilä, T. Tikka, P. Peitso, J. Rantanen, A. Näsilä, K. Nordling, H. Saari, R. Vainio, P. Janhunen, J. Praks, and M. Hallikainen. Aalto-1 nanosatellite – technical description and mission objectives. *Geoscientific Instrumentation, Methods and Data Systems*, 2(1):121–130, 2013. doi:10.5194/gi-2-121-2013.

[24] B. Klofas. Improving receive sensitivity of the CPX bus. *California Polytechnic State University, California, USA*, 2008.

[25] T. Lappalainen. Nanosatelliittien ohjelmistot. Bachelor's thesis, Aalto University, 2011.

[26] J. Ledet-Pedersen, J. D. C. Christiansen, D. E. Holmstrom, et al. The CubeSat Space Protocol. URL `https://github.com/GomSpace/libcsp`. Accessed on January 24th, 2016.

[27] H. Leppinen, N. Silva, P. Niemelä, H. Forstén, H. Sanmark, and J. Javanainen. Aalto-1 on-board computer user manual, 2015. A1-OBC-UM-01-v1.

[28] G. Manyak. Fault tolerant and flexible CubeSat software architecture. Master's thesis, California Polytechnic State University, 2011.

[29] G. Manyak and J. M. Bellardo. PolySat's next generation avionics design. In *Space Mission Challenges for Information Technology (SMC-IT), 2011 IEEE Fourth International Conference on*, pages 69–76. IEEE, 2011. doi:10.1109/SMC-IT.2011.13.

[30] R. H. Maurer, M. E. Fraeman, M. N. Martin, and D. R. Roth. Harsh environments: Space radiation environment, effects and mitigation. *Johns Hopkins APL technical digest*, 28(1):17–29, 2008.

[31] T. C. Maxino and P. J. Koopman. The effectiveness of checksums for embedded control networks. *IEEE Trans. Dependable Sec. Comput.*, 6 (1):59–72, 2009. ISSN 1545-5971. doi:10.1109/TDSC.2007.70216.

[32] K. McCabe. Enhancements to the CPX I^{2}C bus. *California Polytechnic State University*, 2007.

[33] P. Muri and J. McNair. A survey of communication sub-systems for intersatellite linked systems and CubeSat missions. *Journal of Communications*, 7(4):290–308, 2012. doi:10.4304/jcm.7.4.290-308.

[34] K. Nakaya, K. Konoue, H. Sawada, K. Ui, H. Okada, N. Miyashita, M. Iai, T. Urabe, N. Yamaguchi, M. Kashiwa, K. Omagari, I. Morita, and S. Matunaga. Tokyo tech CubeSat: CUTE-I – design & development of flight model and future plan. In *21st International Communications Satellite Systems Conference and Exhibit, International Communications Satellite Systems Conferences (ICSSC)*. American Institute of Aeronautics and Astronautics, 2003. doi:10.2514/6.2003-2388.

[35] A. Näsilä, A. Hakkarainen, J. Praks, A. Kestilä, K. Nordling, R. Modrzewski, H. Saari, J. Antila, R. Mannila, P. Janhunen, R. Vainio, and M. Hallikainen. Aalto-1: A hyperspectral earth observing nanosatellite. In *Proc. SPIE*, volume 8176. International Society for Optics and Photonics, 2011. doi:10.1117/12.898125.

[36] P. Niemelä. Nanosatelliittien lento-ohjelmistojen laadunvarmistus. Bachelor's thesis, Aalto University, 2014.

[37] NXP Semiconductors. I^{2}C-bus specification and user manual, revision 6, April 2014. URL `http://www.nxp.com/documents/user_manual/UM10204.pdf`.

[38] E. Razzaghi. Design and qualification of on-board computer for Aalto-1 CubeSat. Master's thesis, Aalto University, 2012.

[39] E. Razzaghi and S. Lan. Aalto-1 design of the on board computer hardware, 2012. A1-OBH-DD-01-v4.

[40] H. Sanmark. Lento-ohjelmistojen kehitysprosessit nanosatelliiteille. Bachelor's thesis, Aalto University, 2013.

[41] H. Sanmark, N. Silva, P. Niemelä, O. Khurshid, L. Shengchang, A. Yanes, A. Ilmanen, and H. Leppinen. Aalto-1 OBC – payload communication protocol definitions and details, 2013. A1-OBH-DS-03-v4.

[42] D. Schor, J. Scowcroft, C. Nichols, and W. Kinsner. A command and data handling unit for pico-satellite missions. In *Electrical and Computer Engineering, 2009. CCECE '09. Canadian Conference on*, pages 874–879, 2009. doi:10.1109/CCECE.2009.5090254.

[43] R. C. Seacord. *The CERT® C Coding Standard: 98 Rules for Developing Safe, Reliable, and Secure Systems*. Addison-Wesley Professional, 2nd edition, 2014. ISBN 978-0-13-380538-3.

[44] M. Siddique. Aalto-1 CubeSat mission design optimization. Master's thesis, Aalto University, 2015.

[45] N. Silva. Aalto-1 high-level description of on-board computer software, 2013. A1-OBS-DD-02-v1.

[46] N. Silva. Aalto-1 OBS – interaction with subsystems, 2013. A1-OBS-IF-01-v1.

[47] M. V. Stringfellow, Leveson N. G., and B. Owens. Safety-driven design for software-intensive aerospace and automotive systems. *Proceedings of the IEEE*, 98(4):515–525, 2010. ISSN 0018-9219. doi:10.1109/JPROC.2009.2039551.

[48] I. Sünter. *Software for the ESTCube-1 command and data handling system*. PhD thesis, University of Tartu, 2014.

[49] M. Swartwout. Attack of the CubeSats: A statistical look. In *AIAA/USU Conference on Small Satellites*, 2011.

[50] M. Swartwout. A statistical survey of rideshares (and attack of the CubeSats, part deux). In *Aerospace Conference, 2012 IEEE*, pages 1–7, 2012. doi:10.1109/AERO.2012.6187008.

[51] M. Swartwout. The long-threatened flood of university-class spacecraft (and CubeSats) has come: Analyzing the numbers. In *AIAA/USU Conference on Small Satellites*, 2013.

[52] M. Swartwout. The first one hundred CubeSats: A statistical look. *Journal of Small Satellites*, 2(2):213–233, 2013.

[53] Texas Instruments. MSP430FR57xx family: User guide, revision c, November 2013.

[54] W. J. Ubbels, A. R. Bonnema, E. D. van Breukelen, J. H. Doorn, R. van den Eikhoff, E. van der Linden, G. T. Aalbers, J. Rotteveel, R. J. Hamann, and C. J. M. Verhoeven. Delfi-C3: a student nanosatellite as a test-bed for thin film solar cells and wireless onboard communication. In *Recent Advances in Space Technologies, 2005. RAST 2005. Proceedings of 2nd International Conference on*, pages 167–172. IEEE, 2005. doi:10.1109/RAST.2005.1512556.

[55] T. Vladimirova, C. P. Bridges, G. Prassinos, X. Wu, K. Sidibeh, D. Barnhart, A.-H. Jallad, J. R. Paul, V. Lappas, A. Baker, K. Maynard, and R. Magness. Characterising wireless sensor motes for space applications. In *Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007), August 5-8, 2007, University of Edinburgh, Scotland, United Kingdom*, pages 43–50. IEEE Computer Society, 2007. ISBN 978-0-7695-2866-3. doi:10.1109/AHS.2007.39.

[56] M. Zanata, N. Wrachien, and A. Cester. Ionizing radiation effect on ferroelectric nonvolatile memories and its dependence on the irradiation temperature. *Nuclear Science, IEEE Transactions on*, 55(6):3237–3245, 2008. doi:10.1109/TNS.2008.2006052.

Appendix A

Userland watchdog source code

The following listing shows the C source code of the main loop function of the userland watchdog. The main loop accepts several parameters that are read from command-line arguments.

```
static void main_loop(bool foreground, int timeout_s,
                      int sleep_s, time_t watch_timeout_s)
{
  int log_options = LOG_PID | LOG_PERROR;
  openlog("userland-watchdog", log_options, LOG_LOCAL0);

  if (!foreground) {
    /* daemon() is not portable, but uclibc implements
     * it in a sane way */
    if (daemon(0, 0) < 0) {
      syslog(LOG_ERR, "Failed to daemonize: %s", strerror(errno));
      return;
    }
  }

  if (!watch_file_create(WATCHDOG_WATCH_FILE)) {
    return;
  }

  int fd = watchdog_open(timeout_s);
  if (fd < 0) {
    return;
  }
```

```
if (!activate_heartbeat()) {
  return;
}

/* monotonic timestamp of current time */
struct timespec current_time;
/* monotonic timestamp of last time we saw a modification */
struct timespec last_modification;

if (clock_gettime(CLOCK_MONOTONIC, &current_time) < 0) {
  syslog(LOG_ERR, "Failed to get current time: %s", strerror(errno));
  return;
}
last_modification = current_time;

time_t last_st_mtime = 0;
struct stat file_stat;

syslog(LOG_INFO, "Watchdog started: timeout=%ds, sleep=%ds",
    timeout_s, sleep_s);

while(true) {
  if (!watchdog_keepalive(fd)) {
    return;
  }

  if (clock_gettime(CLOCK_MONOTONIC, &current_time) < 0) {
    syslog(LOG_ERR, "Failed to get current time: %s", strerror(errno));
    return;
  }

  if (stat(WATCHDOG_WATCH_FILE, &file_stat) < 0) {
    syslog(LOG_ERR, "Failed to stat target file: %s", strerror(errno));
    /* We don't need to exit immediately after a single stat error.
     * If the error is permanent, the watch timeout will exceed later
     * since the last_modification is never updated */
  } else {
    if (file_stat.st_mtime != last_st_mtime) {
      /* Somebody has touched the file! */
      last_st_mtime = file_stat.st_mtime;
      last_modification = current_time;
    }
  }
```

```
    if (current_time.tv_sec - last_modification.tv_sec > watch_timeout_s) {
      syslog(LOG_ERR, "Watch timeout exceeded");
      return;
    }

    /* We can ignore the return value, because it's always ok
     * to sleep less than sleep_s seconds */
    sleep(sleep_s);
  }
}
```