A reliable, fault-tolerant storage system for DelFFi’s nano-satellites

A THESIS PRESENTED
by
MAHER A. SALLAM

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR A BACHELOR’S DEGREE
IN THE SUBJECT OF
COMPUTER ENGINEERING

THE HAGUE UNIVERSITY OF APPLIED SCIENCES
Delft
June 2014
# Contents

1 **Introduction**  
1.1 Document outline ................................................. 2

2 **Organization and approach** ........................................ 5  
2.1 Project description .................................................. 5  
2.2 Company .................................................................. 7  
2.3 Stakeholders ............................................................. 7  
2.4 Approach ................................................................. 9  
2.5 Risks & mitigation ...................................................... 12  
2.6 Planning ................................................................. 13  
2.7 Summary ................................................................. 16

3 **Literature study** ....................................................... 17  
3.1 Analysis of SEUs in space ............................................. 18  
3.2 Causes of SEUs ......................................................... 19  
3.3 Calculation of SER .................................................... 22  
3.4 Flash memory categorization ....................................... 28  
3.5 Secure Digital (SD) cards ........................................... 29

4 **System requirements** .................................................. 33  
4.1 Use cases ................................................................. 33  
4.2 Functional requirements .............................................. 35  
4.3 Non-functional requirements ....................................... 36  
4.4 Use case scenarios .................................................... 37

5 **Architecture & design** ............................................... 41  
5.1 Fault-tolerance through hardware TMR .......................... 41  
5.2 Detecting SD card controller malfunctions ...................... 44  
5.3 Development hardware .............................................. 47  
5.4 SD card communication protocol .................................. 50  
5.5 Structure of the storage system ................................... 54  
5.6 Software design ........................................................ 58
Appendix D Appendix: Design UML diagrams

D.1 Use cases diagram ........................................ 156
D.2 Packages diagram ...................................... 157
D.3 Class diagrams .......................................... 158
D.4 Activity diagrams ...................................... 162

Appendix E Appendix: Test plan

E.1 Introduction .............................................. 165
E.2 Testing strategy ......................................... 165
E.3 Unit Testing .............................................. 166
E.4 Performance Testing .................................... 166
E.5 System and Integration Testing ...................... 167

Appendix F Appendix: Test procedure & results

F.1 Introduction .............................................. 169
F.2 Unit Testing ............................................. 169
F.3 Performance Testing .................................... 172
F.4 System and Integration Testing ...................... 173
A reliable, fault-tolerant storage system for DelFFi’s nano-satellites

ABSTRACT

TU Delft has started working on its third satellite mission called DelFFi, which is planned for a launch in 2015. In this graduation project spanning 17 weeks, a fault-tolerant storage system consisting of three SD cards was designed, implemented and tested for DelFFi that utilizes triple-modular redundancy on a block-level to restore data upon detecting a mismatch. The research and development was performed in the faculty of aerospace engineering of the Delft University of Technology. The storage system will be used to store measurements recorded by the scientific apparatuses on board the satellites.

Redundancy is achieved by having three replicas of all the data on three different SD cards. The choice for this architecture was clear after an extensive literature study was concluded. As part of the analysis, an estimation revealed that 2-3 errors could occur every second on the storage system because of space radiation. Without hardware redundancy, the integrity of the stored measurements is jeopardized.

FAT32 was implemented as the file system for storing data on the storage system. To verify the quality of the system, three types of testing were performed: unit testing, performance testing and integration testing. In addition, multiple aspects of the system were measured to ensure compliance with specifications such as power consumption.
# Listing of figures

1.1 An artist impression of DelFFi ........................................ 2

2.1 Organogram .......................................................... 8

3.1 Mass stopping power of protons with varying energies in silicon 20
3.2 Total exposure to trapped protons for circular orbits ............ 22
3.3 Proton upset rates for low polar orbits .......................... 25
3.4 Atmospheric upset rates for avionics as a function of the FOM 26
3.5 Cumulative avionics upsets ...................................... 27
3.6 SanDisk microSD card block diagram .............................. 30

4.1 Use case diagram .................................................... 34

5.1 TI MSP430F2418 micro-controller ............................... 48
5.2 Overview of the development board ............................... 49
5.3 USB debug interface for the micro-controller ................... 50
5.4 Breakout board of the SD card sockets .......................... 50
5.5 Structure of the storage system ................................ 54
5.6 FAT32 layout ........................................................ 55
5.7 Packages diagram ................................................ 59
5.8 SPI bus configuration ............................................. 60
5.9 Activities diagram of detecting and correcting errors in a block read 63

6.1 SPI analyser .......................................................... 73

7.1 Power measurement set-up ....................................... 89
7.2 Shunt resistor used in power consumption measurement ....... 90
7.3 Power consumption for 1 block step ............................ 91
7.4 Power consumption for 5 blocks step ............................ 91

C.1 Use case diagram .................................................. 124

D.1 Complete use cases digram ..................................... 156
D.2 Packages diagram ................................................ 157
D.3 Utility class diagram ............................................. 158
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Listing of stakeholders</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Project risks and mitigation measures</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>Global outline of the project plan</td>
<td>14</td>
</tr>
<tr>
<td>5.1</td>
<td>Comparison between two methods for implementing ECC on the storage system</td>
<td>47</td>
</tr>
<tr>
<td>5.2</td>
<td>Comparison between SD card communication protocols</td>
<td>52</td>
</tr>
<tr>
<td>7.1</td>
<td>Count of unit tests per package with their results</td>
<td>82</td>
</tr>
<tr>
<td>7.2</td>
<td>Execution time of CRC computation</td>
<td>84</td>
</tr>
<tr>
<td>7.3</td>
<td>Execution time of reading and writing a block to one SD card</td>
<td>84</td>
</tr>
<tr>
<td>7.4</td>
<td>Execution time of a selected function from the file system</td>
<td>84</td>
</tr>
<tr>
<td>B.1</td>
<td>Global outline of the project plan</td>
<td>120</td>
</tr>
<tr>
<td>F.1</td>
<td>Execution time of CRC computation</td>
<td>173</td>
</tr>
<tr>
<td>F.2</td>
<td>Execution time of reading and writing a block to one SD card</td>
<td>173</td>
</tr>
<tr>
<td>F.3</td>
<td>Execution time of a selected function from the file system</td>
<td>174</td>
</tr>
</tbody>
</table>
List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off-The-Shelf</td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
</tr>
<tr>
<td>ECC</td>
<td>Error Correcting Code</td>
</tr>
<tr>
<td>FAT</td>
<td>File Allocation Table</td>
</tr>
<tr>
<td>FOM</td>
<td>Figure Of Merit</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>LET</td>
<td>Linear Energy Transfer</td>
</tr>
<tr>
<td>LSP</td>
<td>Liskov’s Substitution Principle</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-Up Table</td>
</tr>
<tr>
<td>MLC</td>
<td>Multi-Level Cell</td>
</tr>
<tr>
<td>MMC</td>
<td>Multimedia Card</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>OBC</td>
<td>On-Board Computer</td>
</tr>
<tr>
<td>OCP</td>
<td>Open-Close Principle</td>
</tr>
<tr>
<td>OOD</td>
<td>Object Oriented Design</td>
</tr>
<tr>
<td>RUP</td>
<td>Rational Unified Process</td>
</tr>
<tr>
<td>SD</td>
<td>Secure Digital</td>
</tr>
<tr>
<td>SDHC</td>
<td>SD High Capacity</td>
</tr>
<tr>
<td>SDSC</td>
<td>SD Standard Capacity</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>SDXC</td>
<td>SD eXtended Capacity</td>
</tr>
<tr>
<td>SER</td>
<td>Soft Error Rate</td>
</tr>
<tr>
<td>SEU</td>
<td>Single Event Upset</td>
</tr>
<tr>
<td>SLC</td>
<td>Single-Level Cell</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td>SRP</td>
<td>Single Responsibility Principle</td>
</tr>
<tr>
<td>TDD</td>
<td>Test Driven Development</td>
</tr>
<tr>
<td>TLC</td>
<td>Triple-Level Cell</td>
</tr>
<tr>
<td>TMR</td>
<td>Triple Modular Redundancy</td>
</tr>
<tr>
<td>UHS</td>
<td>Ultra High Speed</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>XP</td>
<td>eXtreme Programming</td>
</tr>
</tbody>
</table>
To my father, mother and sister,
who will always be the beacon of light guiding me through life.
Foreword

I wrote this report as a part of the graduation project for my bachelor study in computer engineering. The space systems engineering chair within TU Delft’s faculty of the aerospace engineering entrusted me with the development of the storage system for their upcoming mission called DelFFi. While working there, I saw a potential astronaut in most people I came across and spoke to. Seeing the spark in their eyes as they discuss complex subjects related to space was amazing every single time. This graduation project is a fantastic chapter in my education story and life in general that I was very lucky to get a chance to complete.

What got me interested in pursing this project was the fact that the software developed is for a space satellite. I knew before starting that such projects can be difficult to complete successfully because of how strict their requirements are, but the challenge became my main incentive. I learned a lot about space, satellites and their electronic systems. Also, being included in the DelFFi’s development team for nearly half a year gave me a first hand look at how satellites are designed and developed, especially their electrical circuits and software, which is an experience that I’m sure will come handy in the future.

A special thanks are due to my mentor at the faculty, ir. Jasper Bouwmeester. In the first place for offering the graduation project to me and in the second place for all the guidance, feedback, help, advice and support he has given me while working there. I’m also thankful to Dr. Chris Verhoeven for pointing me in the direction of ir. Jasper Bouwmeester to find the graduation project. In addition, I would like to thank ir. Nuno Santos, DelFFi’s OBC designer, who helped me many times when I had problems with electrical circuits and components. Many thanks go to both examiners from the Hauge university of applied science, Mr. Kurt Köhler and Mr. John Visser, for their feedback on the first revision of the report and for the assessments.

I would also like to thank all the master students I shared an office with as the sole bachelor student in that room. You made me feel right at home, and I especially enjoyed all the ”Where is our coffee, intern?” type jokes. Thanks for all the advice you have given me on numerates occasions and for the fun time we had there.

Finally, I wish only the best for DelFFi as a mission and for everyone involved in its development. My hope is that the developed storage system helps the DelFFi team in the way I intended for it and to also be used on future space missions that the TU Delft starts. I can’t wait for the launch of DelFFi!

Maher Sallam, 01/06/2014
Introduction

Human curiosity knows no boundaries. This is clearly evident in the story of developing space exploration. The road to visit space is filled with a sheer amount of obstacles for humans, from being creatures without biological wings to the fact that space is inhabitable to any living organism. Still, human intelligence allowed our race to develop solutions to start a journey in that field that reached interplanetary spaceflight.

Delft University of Technology, also known as TU Delft, is paving the way for future space engineers to continue that journey by providing an aerospace education program. The facility of aerospace engineering of the TU Delft is the only institute carrying out research and education directly related to aerospace engineering in the Netherlands. As part of that research, the Delfi Space program was born and it has already launched two nano-satellites into orbit: Delfi-C3 in 2008 and Delfi-n3Xt in 2013.

TU Delft has started working on its third satellite mission called DelFFi, which is planned for a launch in 2015. It comprises two nano-satellites, called Delta and Phi shown in fig. 1.1, which will demonstrate formation flying as part of the QB-50 mission. QB-50 is a unique mission establishing an international network of 50 nano-satellites for multi-point, in-situ measurements in the lower thermosphere and re-entry research.
There are many subsystems on-board the two satellites: On-Board Computer, Electrical Power Subsystem, Micro propulsion, etcetera. On all of these subsystems, there is one or more micro-controllers doing dedicated tasks. Typically there are peripherals like sensors and actuators or memory which needs to be controlled by pieces of service layer code. This code handles the hardware interface and provides a standardized set of functions to the application layer. The development of DelFFi started in 2014, hence the service layer doesn’t exist yet.

The main quest of this graduation project was to develop the storage system for DelFFi which is a part of its server layer. A fault-tolerant storage system consisting of three SD cards was designed, implemented and tested that utilizes triple-modular redundancy on a block-level to restore data upon detecting a mismatch. The storage system will be used to store measurements recorded by the scientific apparatuses on board DelFFi. FAT32 was implemented as the file system for storing data on the storage system. To ensure the quality of the system, three types of testing were performed: unit testing, performance testing and integration testing. In addition, multiple aspects of the system were measured to ensure compliance with specifications such as power consumption.

1.1 DOCUMENT OUTLINE

The document in your hand begins by presenting the organization and approach used throughout the project in chapter 2. After that, chapter 3 discusses the important lessons learned in the literature study. Hereafter, system requirements are listed in chapter 4. Subsequently, the fault-tolerant architecture of the storage system is described in detail in chapter 5. In addition, the software design consisting of packages and classes is outlined in that chapter. Chapter 6 contains the information regarding how the implementation of the storage system was done. Next, the testing methodology is described in chapter 7 with the results of the tests. We end discussing the graduation project in chapter 8 with an evaluation of the process, documents and products. Finally, chapter 9 includes a sum-
mary of the work in the form of a conclusion.
Introduction
2 Organization and approach

In this graduation project spanning 17 weeks, the research and development was performed in the faculty of aerospace engineering of the Delft University of Technology solely by the graduate and under guidance from the company mentor.

This chapter begins by briefly presenting the problem, goal and result of the project. More information regarding these items can be found in the graduation plan. The plan is an appendix that is included in chapter A. Subsequently, the organization within which the project takes place is described. Hereafter, a discussion of the approach used in this project is given, which is also included in the work plan found in chapter B.

2.1 Project description

2.1.1 Problem to be solved

There are many subsystems onboard these two satellites: OnBoard Computer, Electrical Power Subsystem, Radio Transceivers, Attitude Determination and Control, Micro propulsion, QB-50 Sensor Suite Payload, etcetera. On all of these subsystems, there is one or more microcontrollers doing dedicated tasks. Typically there are peripherals like
sensors, actuators, ADCs, DACs, internal communication or memory which needs to be controlled by pieces of service layer code. This code handles the hardware interface and provides a standardized set of functions to the application layer. The service layer is yet to be developed.

2.1.2 Project goals

The service layer needs to be designed, implemented and tested for a few microcontrollers that will be used by DelFFi. Two tasks carried out by this layer have already been determined. More could follow once the hardware has been identified in early 2014.

First, the on-board data storage. A Texas Instruments MSP430F2418 micro-controller needs to store data on an SD card. The service layer will allow reading and writing packets of data from and to the SD card, respectively. Error detection and correction code will be implemented which can deal with a 1:2 bitflip ratio. The latter is an extreme case caused by long term radiation in space where electrons are hitting the memory causing malfunctions. Here is reliability of utmost importance.

Second, Bluetooth communication. DelFFi will have an on-board experiment which includes a Bluetooth link between a few temperature sensors on the body of the satellite and the on-board computer. The service layer will allow a reliable link in a real satellite situation.

2.1.3 Project result

After completion, the service layer will provide a standardized set of functions to the application layer for communication with specific hardware devices. More importantly, the service layer will ensure a reliable operation through well documented design choices and thorough testing. Documentation will be provided in English.

Delivered (intermediate) products

- Work plan
- Functional and non-functional requirements document
- Test plan
- Test reports
- Design documents (UML diagrams)
- Implementation code
2.2 Company

Delft University of Technology, also known as TU Delft, is one of the biggest public technical universities in the Netherlands and Europe. More than 17,000 students and 2,400 scientists study and research, respectively, a plethora of fields of science in its eight faculties and many research institutes. The university was founded in 8 January 1842 by king Willem II and has since acquired multiple names before being called TU Delft.

The facility of aerospace engineering is one of the major faculties inside the TU Delft with four dedicated departments, around 2 300 students and 15 full-time professors. Moreover, this faculty is one of the largest faculties devoted entirely to aerospace engineering in Europe. It is the only institute carrying out research and education directly related to aerospace engineering in the Netherlands.

Figure 2.1 shows the organization of aerospace engineering faculty with the position of the graduate within the hierarchy. The DelFFi mission is developed under the Space Systems Engineering chair of the space engineering department. Therefore, it follows that the graduation project is also done in the same department.

2.3 Stakeholders

Table 2.1 shows the stakeholders involved in this project. Officially, the dean of the aerospace engineering faculty Prof. dr. ir. H. Bijl is the project provider with which the work placement agreement was signed. However, the primary mentor for the graduate within the faculty is ir. J. Bouwmeester who was the project manager for Delfi-n3Xt. Dr. ir. Jian Guo is the project manager of DelFFi.

Table 2.1: Listing of stakeholders

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hester Bijl</td>
<td>Aerospace engineering faculty dean</td>
<td>Project provider</td>
</tr>
<tr>
<td>Jasper Bouwmeester</td>
<td>Researcher Small Satellite Technology</td>
<td>Primary company mentor</td>
</tr>
<tr>
<td>Jian Guo</td>
<td>DelFFi Project Manager</td>
<td>DelFFi Project Manager</td>
</tr>
<tr>
<td>Maher Sallam</td>
<td>CE student (Graduate)</td>
<td>Software developer</td>
</tr>
</tbody>
</table>
Figure 2.1: Organogram of the aerospace engineering faculty
2.4 Approach

After evaluating multiple software development methodologies, it was decided to use OpenUP for the project. The next subsections discuss the evaluation of the methods considered and the reason behind choosing OpenUP.

2.4.1 Selection of a development methodology

A broad spectra of development methods were considered ranging from heavyweight methodologies like the Rational Unified Process (RUP) to more lightweight ones such as Scrum and eXtreme Programming (XP). Moreover, Test Driven Development, which is a relatively new method, was also considered. Only iterative and incremental methods are considered as it has been proved constantly that they are superior to static methods like waterfall.

Reliability in this project is of utmost importance. The most common and effective way to ensure this non-functional requirement is by taking it into consideration while designing the architecture and by extensive testing afterwards. Therefore, reliability will be an important criterion in the evaluation.

Lightweight methods

All of the lightweight methods adopt the philosophy that the project should be designed and implemented at the same time because of time constraints and because producing a shippable product is their priority. Afterwards, the design is improved by means of refactoring. Although this might produce a robust design eventually, it can’t guarantee reliability of operation in each iteration. Said another way, reliability, and non-functional requirements in general, are an afterthought in lightweight methods.

Heavyweight methods

In contrast, heavyweight methods focus on generating a lot of design and architecture documents and tests to preserve quality. They are most suitable for large projects with many people involved. The danger with using such a method for a one person project is that the documentation required by these methods can be overwhelming and could result in not completing the project within its time limit.

Test Driven Development
Test driven development (TDD) distinguishes itself with a unique development cycle. While most methods will first design an architecture and then carry out tests to verify the implementation, TDD begins by writing tests for the use cases. Since no implementation has been developed, the tests should fail in the first run. Then code is written to make the tests pass. There is no formal design phase in TDD; the tests implicitly define the architecture of the software. TDD’s approach has the possibility to ensure any non-functional requirement if tests can be written for them. However, for an embedded project like this one, it is quite hard to write tests for some non-functional requirements, and especially for the low-level parts of the implementation. Furthermore, not having formal design documents will make it difficult to evaluate the quality of the architecture.

2.4.2 RATIONALE BEHIND CHOOSING OPENUP

For this project, a middle ground should be chosen between lightweight and heavyweight methods. A good combination would provide sufficient documentation of the design choices while at the same time not sacrificing quality. It would also allow producing working functionality with each increment. Furthermore, testing should be a key characteristic of the method.

OpenUP can provide this middle ground. It’s a simplified version of RUP that only keeps all of the core principles of RUP, which is as aforementioned a heavyweight method. The architecture-centric approach allows taking reliability in consideration the design process. Furthermore, tests are performed multiple times within each iteration.

The four core principles of OpenUP are listed below.

- Collaborate to align interests and share understanding.
- Balance competing priorities to maximize stakeholder value.
- Focus on the architecture early to minimize risks and organize development.
- Evolve to continuously obtain feedback and improve.

2.4.3 PRACTICES AND MEASUREMENTS

In each iteration a burndown report will be used to track and measure the work being done and to collect metrics for the next iterations.
The architecture will be mostly described by Unified Modeling Language (UML) models and where needed elaborated textually. UML is the canonical language to describe models within the software development industry.
2.5 **Risks & mitigation**

The risks and mitigation measures are described in table 2.2 which is based on a template provided by OpenUP. They were gathered by taking a critical look at what can jeopardize the project. In addition, discussions with the stakeholders helped in compiling the risks list.

Table 2.2: Project risks and mitigation measures

<table>
<thead>
<tr>
<th>ID</th>
<th>Headline</th>
<th>Description</th>
<th>Type</th>
<th>Impact</th>
<th>Probability</th>
<th>Magnitude</th>
<th>Mitigation Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reliability is not sufficient</td>
<td>The architecture and implementation should be highly reliable to allow the product to withstand the harsh conditions in space.</td>
<td>Direct - Technical</td>
<td>5</td>
<td>40%</td>
<td>2.0</td>
<td>Research will be conducted to have a better understanding of the reliability issues. Architecture and design choices will be made to ensure reliability. They will also be discussed and approved with the company mentor. Furthermore, extensive tests will be performed.</td>
</tr>
<tr>
<td>2</td>
<td>Project is not finished before deadline</td>
<td>If a lot of problems occur, they can delay the expected time for finishing tasks and hence the deadline could be missed</td>
<td>Direct - Schedule</td>
<td>5</td>
<td>30%</td>
<td>1.5</td>
<td>Graduate will try to find solutions as soon as a problem occurs. Unsolvable problems will be communicated to company mentor and to university mentors.</td>
</tr>
</tbody>
</table>
2.6 Planning

A global overview of the planning is outlined in the table below. It shows the phases defined for carrying out the work and the iterations within each one of them. It also sets the primary objectives to achieve. In additions, the length of the iterations is shown.

<table>
<thead>
<tr>
<th>Phase Description</th>
<th>Hardware/Task</th>
<th>Iteration</th>
<th>Direct/Indirect</th>
<th>Resource</th>
<th>Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development</td>
<td>Hardware is needed to develop and verify the implementation and to debug problems.</td>
<td>3</td>
<td>40%</td>
<td>1.2</td>
<td>Required hardware will be reported to the company mentor as soon as they are required.</td>
</tr>
<tr>
<td>Developing for a new microcontroller architecture with no prior experience</td>
<td>The MSP430 is a new microcontroller for the graduate. Developing for it can prove to be difficult.</td>
<td>4</td>
<td>20%</td>
<td>0.8</td>
<td>Specifications of the MSP430 will be studied carefully. Moreover, examples and tutorials will be used in case they are needed.</td>
</tr>
<tr>
<td>Power consumption of developed product is high</td>
<td>The satellite has a limited amount of power in comparison to a PC. Therefore, care must be taken to not exceed the power limitations.</td>
<td>4</td>
<td>10%</td>
<td>0.4</td>
<td>Power consumption will be calculated for any suggested hardware solution (e.g., using multiple SD cards for redundancy) that doesn’t directly use the micro-controller.</td>
</tr>
</tbody>
</table>
Table 2.3: Global outline of the project plan

<table>
<thead>
<tr>
<th>Phase</th>
<th>Iteration</th>
<th>Primary objectives</th>
<th>Start date</th>
<th>End date</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception</td>
<td>1</td>
<td>1. Create project plan</td>
<td>10/02/2014</td>
<td>28/02/2014</td>
<td>3 weeks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Create project requirements document</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Research MSP430 development, communication with SD card, reliability of SD card in space (redundancy, ECC... etc.)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Elaboration</td>
<td>1</td>
<td>1. Prioritize project requirements</td>
<td>03/03/2014</td>
<td>28/03/2014</td>
<td>4 weeks</td>
</tr>
<tr>
<td>(Storage)</td>
<td></td>
<td>2. Define use cases and scenarios for storage solution</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Design architecture of storage solution</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Construction (Storage) | 1 | 1. Stabilize architecture design  
2. Start implementing the design  
3. Create test plan and test implementation so far | 31/03/2014 | 17/04/2014 | 3 weeks |
|-----------------------|---|---------------------------------------------------------------------------------------------------------------------------------|-------------|-------------|---------|
|                       | 2 | 1. Stabilize implementation of storage solution  
2. Perform extensive testing of implementation | 22/04/2014 | 23/05/2014 | 5 weeks |
2.7 **Summary**

In this graduation project spanning 17 weeks, the storage system and Bluetooth communication is developed for DelFFi as a part of the service layer. After completion, the service layer will provide a standardized set of functions to the application layer for communication with specific hardware devices. More importantly, the service layer will ensure a reliable operation through well documented design choices and thorough testing.

The organization of the aerospace engineering faculty was presented along with a list of the stakeholders. The project is developed under the Space System Engineering chair.

OpenUP will be employed as the software development method since its suitable for use by a single person. Being a simplified version of RUP, OpenUP provides a suitable middle ground between lightweight and heavyweight methods that focuses on architecture and testing to minimize risks. Architecture will be described using UML models.

A list of risks has been compiled and prioritized to tackle them in order of importance based on magnitude. Moreover, mitigation strategies have been described. Finally, the planning has been outlined including the primary objectives per phase.
Developing a storage system falls predominantly within the expertise of electrical and computer engineers. However, when the system is on a spacecraft, some degree of knowledge in aerospace engineering is required to achieve the goals of the project. At the inception of the project a literature study was conducted to primarily answer two questions related to the storage system. The first question is: how susceptible to errors will the storage system be when it’s exposed to radiation in space? The second question reads: how reliable are SD cards in general?

Both question arose as a consequence of a meeting with Jasper Bouwmeester in the first few days of the project. In the meetings he shared that DelFi’s predecessor nanosatellite called Delfi-n3xt had a storage system that failed a few weeks after the satellite went into orbit. The storage system was experimental and consisted of a single micro SD card that was used to store measurements. Since it was experimental, the failure didn’t jeopardize the operation of Delfi-n3xt.

In contrast, the storage system on DelFFi is a requirement that must be included on all QB50 satellites and is defined in the QB50 system requirements document [29]. Therefore, DelFFi’s storage system is not considered an experiment but is actually a main
system that should be robust.

To ensure that DelFFi's storage system survives this time, one must first know what radiation does to electronics in space. Then, flash memory must be studied to gain the insight required in designing a storage system that will not only survive, but also ensure fault-tolerance to preserve the integrity of the measurements stored.

Some readers might wonder why SD cards will be investigated instead of other storage media such as HDDs or SSDs. This is because the DelFFi development team has decided to go with SD cards before the inception of the project. As we shall see in this chapter, their choice was not wrong.

This chapter delves into the significant information gained in the literature study phase and shows the analysis with its results that led to answering both of the aforementioned questions.

3.1 Analysis of SEUs in space

Exposing electronic devices to radiation can cause different malfunctions in electronics [14]. These malfunctions can be grouped in two categories: permanent and transient errors.

Permanent errors are the result of sufficiently high energy particles hitting the internal hardware thereby causing the values of bits to be physically stuck at either logic one or zero [33]. This kind of errors damages the hardware and therefore it is irreparable from software. In other words, once a permanent error occurs, it's not possible to restore the affected bits to a valid state by performing a software routine.

Transient errors, commonly referred to as single event upsets (SEUs) in aerospace literature and as soft errors generally, are also caused by high energy particles. When such a particle strikes, it causes the value of a bit, for example in a flip-flop, to temporarily change. As the name suggests, this kind of errors does not damage the hardware. However, it can corrupt saved data as the duration of the state change is unknown.

Since permanent errors are uncorrectable once they occur and must be dealt with on hardware-level, one must prevent them from inducing error by blocking radiation from hitting the electronics. However, nano-satellites developed by universities are used to explore the usage of Commercial Off-The-Shelf (COTS) products. Moreover, radiation shielding can be very expensive and use a fair amount of valuable space within a satellite. Consequently, they avoid using radiation shielding and look for other solutions to deal with these problems.
Furthermore, this graduation project will not be developing a custom flash-chip that can tolerate permanent errors. Therefore, we mainly focus on analyzing SEUs.

### 3.2 Causes of SEUs

In space, collections of high energy particles that travel at immense speeds are called cosmic rays. Cosmic rays increase greatly with altitude. This phenomenon was discovered by Victor Hess and it lead to him winning the Nobel Prize in physics for 1936 [1]. These rays are divided into four categories based on origin and altitude as follows: primary, solar, secondary and terrestrial cosmic rays [34]. The distinction between the first two categories is not very sharp and sometimes solar cosmic rays are included in primary cosmic rays [34]. For most space missions, only primary and secondary cosmic rays are of interest when evaluating SEUs because terrestrial cosmic rays are found at a very low altitude.

The QB50 mission targets the lower thermosphere (90-400 km above sea level), with an inclination of 98.8° thereby specifying a polar orbit for the satellites [29]. The flux of the secondary particles peaks at an altitude of around 15 km (called the Pfotzer point) [23, Section 1.7]. Beyond that point, the flux re-declines until it reaches the amount found at sea level. Therefore, secondary cosmic rays will not pose a threat for the storage system. Analyzing errors caused by these rays can be of interest for avionic systems as airplanes typically cruise at an altitude of around 10km. For DelFFi however, the errors induced by the secondary cosmic rays should be comparable to the amount induced at sea level.

Primary cosmic rays in contrast will probably be the major contributor of SEUs in the storage system. These rays consist mostly of protons (92 %) and have energies that range in orders of magnitudes from 1 GeV to $10^{20}$ GeV. In comparison, particles in the secondary cosmic rays are measured in MeV. Theoretically, a particle with the lowest energy from the primary cosmic rays (e.g., 1 GeV) will have 3 orders of magnitude more energy than a typical particle from the secondary cosmic rays (e.g., 1 MeV). Because of the suspicion that primary cosmic rays will induce the most SEUs in the storage system, this hypothesis is investigated further in the next sections.

When a high energy particle penetrates silicon crystals, it creates electron-hole pairs in the bulk or the substrate of a transistor [23, Section 1.7.3]. This process is referred to as direct ionization. To create one such pair, around 3.6 eV is required. The most important aspect to consider here is if the generated charge by these electron-hole pairs is
enough to generate a bit-flip. The threshold charge at which a bit-flip is guaranteed to be induced is called the critical charge $Q_{\text{cri}}$ in the literature. Multiple equations used for calculating the soft error rate (SER) caused by exposure to radiations include $Q_{\text{cri}}$ as one of the parameters. Unfortunately, $Q_{\text{cri}}$ is very difficult to calculate because it’s dependent on a plethora of factors, even when a simulator is employed \[1^8\].

### 3.2.1 Generated charge from collisions with primary cosmic rays

Due to their massive energies, one can expect that the charge generated by collisions with primary cosmic rays will be sufficient to cause a bit-flip in most transistors, even if $Q_{\text{cri}}$ is unknown for that device. If this is true, the SER will then depend on the probability of $Q_{\text{cri}}$ being generated.

To calculate the generated charge, the following equation can be used:

$$Q = S(E) \cdot e$$

where $S(E)$ is the stopping power for a particle with a certain energy in a given material and $e$ is the elementary charge ($\approx 1.6 \times 10^{-19}$ C). The stopping power is defined as the energy lost by a particle per unit length. The concept of linear energy transfer (LET) typically found in literature is equivalent to the stopping power. Strictly speaking, LET is
equal to \( S(E) \) when all the energy absorbed by the medium is utilized for the production of electron-hole pairs [23, Section 1.7.3].

The U.S. National Institute of Standards and Technology (NIST) provides an online tool to calculate the stopping power of protons in different materials. Figure 3.1 shows the mass \( S(E) \) for protons in silicon for a range of energies. Mass \( S(E) \) can be converted to stopping power by multiplying by the density of the material, which is 2.3290 g cm\(^{-3}\) for silicon. As can be seen in the graph, higher energy particles have less stopping power.

For primary cosmic rays, this implies that the generated charge will actually be low because of the linear relation between the generated charge and the stopping power, as can be inferred from the equation above. This observation is further confirmed by the following quote from [28, Section 1.1]:

> It is not the proton passage that produces the effect. The proton itself produces only a very small amount of ionization. Very few devices are sensitive enough to respond to the proton ionization.

### 3.2.2 Heavy recoil nuclei induced by proton strikes

The discussion in the previous section considered the charge generated by direct ionization from a proton strike. However, the same strike can induce nuclear reactions in silicon resulting in heavy recoil nuclei capable of generating SEUs [28, Section 1.1]. Atomic recoil is a quantum phenomenon which is inherently difficult to explain. Fortunately, we are not trying to explain the process of how SEUs are induced, but we are trying to measure the SER of these errors. The details of the reactions occurring within silicon are of little interest to the calculations of SER.

About one proton in \( \text{10}^5 \) will undergo a nuclear reaction capable of generating an SEU [28, Section 1.1]. Because of QB50’s low altitude orbit, trapped protons by Earth’s magnetic field are a concern. In this area surrounding Earth, protons in the South Atlantic anomaly are the source of the majority of SEUs in satellites [28, Section 2.3.4].

Using fig. 3.2 taken from [28, Section 2.3.4], the total exposure to proton particles with high energy (\( > 30 \text{ MeV} \)) can be estimated for DelFFi. At the start altitude (320 km), the fluence is around \( \text{10}^6 \) protons/cm\(^2\) each day. Based on the previous result, between 1 to 2 nuclear reactions/cm\(^2\) will be induced each day. However, an occurring nuclear reaction doesn’t constitute an SEU because the generated charge must first surpass \( Q_{cri} \).
3.2.3 Summary

To summarize, of the four categories of cosmic rays, primary cosmic rays will induce the majority of SEUs for the storage system. The bit-flips will occur not because of direct ionization, but because of nuclear reactions in silicon after a strike. The amount of nuclear reactions induced everyday is estimated to be between 1 and 2 nuclear reactions/cm².

This partially answers the first question in the intro to the chapter. It confirms that radiation is dangerous to electronics and therefore that the storage system on DelFFi is susceptible to errors induced by radiation. However, we still don’t know how much errors will be produced. Knowing the SER allows us to make the right decisions while designing the system for fault-tolerance.

3.3 Calculation of SER

A semi-empirical method for calculating SER was proposed by Taber and Normand that doesn’t require of knowledge of $Q_{cri}$. It is called the neutron cross-section (NCS) method \[23, \text{Section 2.2.2}\]. The mathematical equation reads:

\[
SER = \int_{E_{neutron}} \sigma \frac{dN}{dE} \, dE
\]

where $\frac{dN}{dE}$ is the differential neutron flux spectrum (expressed in neutrons/cm²-hour-MeV) and $\sigma$ is the neutron cross section. The same equation can be used to calculate SER induced by protons as well after substituting neutrons data with protons data in the
3.3 Calculation of SER parameters [28, Section 8.2].

However, even though this method is relatively easier for calculating SER, obtaining the parameters is still quite difficult. The cross-section is a function of $S(E)$. Hence, one must first experimentally gather multiple cross-sections for proton upsets. Then, the cross-section curve must be interpolated; Peterson suggests using the log-normal distribution here. Only then can the SER be calculated.

This means that we can’t directly calculate the SER ourselves without having the cross-sections, which we can’t obtain since a large particles accelerator is required to carry the experiments that can produce high energy particles.

3.3.1 Previous research into SER in flash memory

Since the SER can’t be directly calculated, previous research into the same subject was studies. Thereafter, the gained knowledge can be used to make an estimation of the SER.

Multiple researchers have investigated the effects of radiation on flash memory. One of the first papers written about this subject was published by Schwartz et al. [30] in 1997. Fogle et al. [16] found the limiting proton cross-section for a 64 MiB flash memory to be $3 \times 10^{-18}$ cm$^2$/bit. This cross-section is described as limiting because it’s obtained at saturation from proton energy. In other words, increasing the proton energy after this point doesn’t affect the cross-section anymore. In their paper, Fogle et al. reported that the observed cross-section in their experiments is 300x smaller than the one seen by Schwartz et al. in the 7 years older device.

In 2007, Oldham et al. [27] measured the effects of total ionizing dose (TID) and SEUs on a 4 GiB Samsung flash memory fabricated with 63nm technology. This flash uses single level cell (SLC) technology to store one bit per cell. Although referred to as the saturation cross-section in their paper, the measured limiting cross-section was $5 \times 10^{-11}$ cm$^2$/bit.

This result is surprising because one might expect a decrease of the cross-section with technology scaling as was reported by Fogle et al. Namely, as the area per transistor decreases, protons hitting the silicon will have a smaller chance of colliding with a particle thereby decreasing chances of causing an SEU. However, differences in fabrication processes between manufactures together with new technologies used to increase yield and capacity of flash memories could explain the increase of the cross-section.
3.3.2 Estimating SER using previous research data

We can use the results of research presented in the previous section to make an estimation of the expected SER for a particular flash memory in DelFFi.

Most probably, the flash memory on board the nano-satellites will have a capacity in the order of gigabytes. Hence, cross-section measured by Oldham et al. is the most appropriate one for calculating our estimations.

The cross-section is calculated using the equation [28, Section 2.10.11]:

\[
\sigma = \frac{N}{F}
\]

where \( \sigma \) is the proton cross-section, \( N \) is the number of upsets and \( F \) is the fluence measured in protons/cm\(^2\). Since we already estimated the fluence at DelFFi’s orbit per day, we can calculate the expected upsets per bit each day as follows:

\[
N = 5 \times 10^{-11} \times 10^6
\]
\[
= 5 \times 10^{-5} \text{ upsets/bit - day}
\]

For the whole flash memory, the SER per day becomes:

\[
SER = 5 \times 10^{-5} \times 4 \times 10^9
\]
\[
= 2 \times 10^5 \text{ upsets/day}
\]

Another way for estimating SER will be by calculating the figure of merit (FOM) for the cross-section [28, Section 7.8]. Using equation 8.4 from [28] leads to a FOM of about \( 2 \times 10^{-6} \) for the flash memory as follows:

\[
FOM = 4.5 \times 10^4 \times \sigma_{PL}
\]
\[
= 4.5 \times 10^4 \times 5 \times 10^{-11}
\]
\[
\approx 2 \times 10^{-6}
\]

In section 12.2 of the same book, fig. 3.3 can be found which graphs the upset rate per
3.3 Calculation of SER

Figure 3.3: Proton upset rates for low polar orbits. The numbers (18, 93, 268, 1324, and 4370) are the rate coefficients for these orbits.

day as a function of FOM for low polar orbits. For DelFFi’s altitude, the upset rate per bit each day is around an order of magnitude higher than the FOM value. This equals an upset rate of $6 \times 10^{-5}$ upsets per bit each day. Again, the SER is then:

\[
SER = 6 \cdot 10^{-5} \times 4 \cdot 10^9 \\
\approx 2 \cdot 10^5 \text{ upsets/day}
\]

Given a lot of estimation and rounding has been done, having obtained the same SER from two different methods confirms the correctness the calculations. The calculated SER suggests that 2-3 SEUs will occur every second on a 4 GiB flash memory device in DelFFi’s orbit, which could amount to huge data losses as errors accumulate.
3.3.3 Comparison to SER in avionic systems

How does the calculated SER in DelFFi’s orbit compare to the expected SER for avionic systems? Airplanes already suffer from the effects of bit-flips although they cruise at much lower altitudes \[28\]. The reason for comparing the SER in DelFFi’s orbit with that of airplanes instead of comparing it to data from other satellites is to illustrate the increase in SER with altitude. If we see an increase in the SER, then we know that the measures used in airplanes to combat SEUs might not be sufficient for DelFFi. However, if the change in SER is minuscule, then SEUs will be as much of a problem for DelFFi as they are to airplanes and hence one can see how airplane manufacturers deal with SEUs.

Combining the FOM value for the 4 GiB flash memory together with fig. 3.4 from \[28\, Section 9.1.3\] answers the question. At an altitude of around 13km, at which airplanes typically cruise, the amount of upsets due to neutron particles roughly equals \(3 \times \text{FOM}\). The cross-section for neutrons in the memory at sea level is in the same order of magnitude as the one for protons \[16\]. Therefore, FOM need not be calculated again for neutrons; we can use the previously calculated FOM here. Hence, the SER is:

\[
\begin{align*}
\text{SER} &\approx 3 \times \text{FOM} \times 4 \cdot 10^9 \\
&= 3 \times 2 \cdot 10^{-6} \times 4 \cdot 10^9 \\
&\approx 2 \cdot 10^4 \text{ upsets/day}
\end{align*}
\]

To confirm the calculations, one can use the results published by Taber et al. \[32\].

Figure 3.4: Atmospheric upset rates for avionics as a function of the FOM.
in 1993 showing cumulative upsets in SRAMs on board a military aircraft flying at an altitude of around 9km. Taking the average of the data in fig. 3.5 enables us to estimate the upset rate per hour. The rounded average is $2 \times 10^{-3}$ upsets per hour. In one day, the SER is then:

$$\text{SER} = 2 \cdot 10^{-3} \times 24$$

$$\approx 5 \cdot 10^{-1} \text{ upsets/day}$$

Our calculations fall right within the margin given by Fogle et al. [16] estimating that SERs in flash memory are 3 to 5 order of magnitudes less compared to SRAMs.

The result suggests an order of magnitude increase in the SER for DelFFi's orbit compared to an airplane's flight. Avionic systems already face different problems caused by SEUs and implement multiple measures to counter them. Add to that the difference in exposure time between an airplane's flight and a satellite's orbit and one can comprehend the seriousness of the threat SEUs pose to DelFFi. Therefore, a factor 10 increase of SER mandates serious considerations in the design of the storage system to avoid loss of valuable data.
3.3.4 Conclusion

In conclusion, calculating the SER for DelFFi’s orbit directly is not possible because parts of the information required for the equations are unattainable. Using previous research however, an estimation of the SER was made that predicted 2-3 SEUs will occur every second on a 4 GiB flash memory device in DelFFi’s orbit.

Compared to airplanes, DelFFi will face an order of magnitude more SEUs in its orbit. As a result, SEUs are a real danger to DelFFi’s storage system and can’t be ignored; serious measures must be implemented in the storage system to avoid loss of valuable data. With this we found the answer to the first question (how susceptible to errors will the storage system be when it’s exposed to radiation in space?) and were able to quantify the danger in terms of the amount of SEUs per second.

3.4 Flash Memory Categorization

After establishing that SEUs are a real threat to DelFFi’s storage system, one can start looking into possible countermeasures to mitigate this problem. Two common solutions are error correcting code (ECC) and redundancy [5]. However, to implement a robust solution, a great deal of insight into the employed hardware is required. To give the reader a better grasp of the different terminology used to describe flash memory, an overview is presented without delving too much into details.

Flash memory is a type of non-volatile storage medium that comes in two main types: NAND and NOR. Most mass-storage devices use NAND flash memory because it allows packing more transistors per area compared to NOR-type flash at the cost of sacrificing random-access to individual bytes [13]. Thus, NAND memory can only read and modify multiple bytes at a time. On the other hand, NOR flash is used for devices that require fast random-access such as program memory in micro-controllers. Since DelFFi’s storage system is also a mass-storage device, NAND flash will be the main focus in the rest of the document.

Usually, NAND flash is logically structured hierarchically into pages, blocks and planes. A page is a fixed-sized set of bits grouped together into a unit. Blocks are sets of pages. Reading and programming operations can be performed on individual pages. However, when performing erasure, an entire block has to be erased. Finally, blocks are grouped into planes that can simultaneously perform different operations. In other words, a flash memory with N planes can perform N operations in parallel.
Each cell within a NAND flash can be used to store one, two or three bits in single-level cell (SLC), multi-level cell (MLC) and triple-level cell (TLC) devices, respectively. A cell in a flash device is typically a floating-gate transistor [13].

### 3.4.1 Considerations when using NAND flash memory

NAND flash has multiple issues that designers of file systems need to be aware of. After a certain number of erasure cycles, cells cannot be considered reliable anymore for storing data. Two terms are used in the literature to describe this phenomenon: endurance and wearing out. The amount of erasure cycles that a flash memory cell can tolerate before becoming unreliable is called *endurance*. When a cell exceeds its endurance value, it’s described as a *worn out* cell. A typical value for the endurance of an SLC flash memory is about $10^5$ erasure cycles [13, Section 2.2]. To distribute data evenly across blocks in the entire flash memory, *Wear leveling* techniques are used to level and to minimize the number of erasure cycles of each block.

The limitation of block erasure in NAND flash manifests itself as another issue when a page has to be overwritten. It is only possible if the whole block is erased and then re-programmed. To overcome this problem, the updated content of the page is written to a new page [13, Section 2.4]. The new page is then marked as valid while the old one becomes invalidated. To facilitate the mapping between the old page number and the new one, the address translation table is adjusted accordingly. This means that the number of invalid pages gradually increases over time until all free pages are used. A cleaning mechanism is required to erase invalid pages to free up space. This operation is referred to as garbage collection. Garbage collection as one can imagine decreases the performance of NAND flash and impacts the endurance negatively as well.

### 3.5 Secure Digital (SD) cards

Flash memory chips come in different form factors and sometimes they are even integrated into the same die for special applications. Two common storage devices that include flash memory are memory cards and USB flash drives. Since their invention, memory cards have been developed by multiple companies each giving their product a commercial name resulting in a long list of memory card types. Examples include PC Card, Memory Stick, Multimedia card (MMC) and Secure Digital (SD) card. The rest of this document only focuses on SD cards since it’s the type that has been chosen to be on board DelFFi.
A lot of secrecy surrounds the development techniques of SD Cards because of its commercial nature. For developers interested in writing software to communicate with the cards, they are always directed towards the SD association which define the specification for such cards, all the way from the physical requirements to the communication protocols. However, when analyzing a specific card for mission critical objectives, knowledge of the actual hardware implementation becomes vital.

Beside the normal advantages inherent to flash memory, examining the internal structure of SD cards uncovers fascinating facts. All SD cards contain a small embedded controller. These controllers need to run at relatively high frequencies ranging from 25Mhz up to 100Mhz to support the transfer speeds mandated by the SD specification. One of the last publicly available OEM manuals published by Sandisk for their line of micro SD cards includes a block diagram of the internal structure of their products. The block diagram in fig. 3.6 shows the embedded controller chip. These controllers perform a wide range of functions, ranging from data exchange and error correction to defect handling and wear leveling.

Due to fabrication errors, some transistors on a silicon die become unreliable for storing data. However, to increase the yield, manufactures use a technique called bad block remapping in which the controller is loaded with the addresses of bad blocks to ensure that they are inaccessible by users. These cards are then sold as smaller capacity cards based on the bad blocks amount. In some cases, over 80% of blocks are bad thereby resulting for example in an originally manufactured 16 GiB card being sold as a 2 GiB card.

Another interesting feature of SD cards, and NAND flash in general, is that error correcting codes (ECCs) are systematically used to improve the level of reliability. Some manufactures implement ECC in hardware.
3.5 Secure Digital (SD) cards

and some opt for cheaper software solutions. Surprisingly, sophisticated multi-bit ECCs are regularly implemented. Bose-Chaudhuri-Hocquenghem (BCH) codes are linear codes that are widely adopted in flash memories [13, Section 2.7]. This is due to the fact that flash memory can be quite unreliable without these protective measures [20]. Typically, ECC algorithms have high computational needs that can form a challenge for implementers. However, because of the powerful micro-processors incorporated in SD cards, which run at high frequencies to support the transfer rates demanded by the specification, it’s feasible to implement them.

Aside from the statement in their product-manual, one can conclude that SanDisk SD cards have reliable ECC algorithms by considering a patent they filled for a method to implement ECC in non-volatile memory [24]. Not only is the ECC calculated for a block, but SanDisk engineers optimize the case for when a block is erased and they make sure power failures are accounted for.

Furthermore, even when ignoring the unreliability of SD cards without decent ECC algorithms [20], one can assume that some form of ECC is implemented because the SD specification requires storing ECC failures in the status register. The stored bit indicates that the ECC algorithm has been applied on the data but failed to correct the data [3, Section 7.3.4].

3.5.1 Conclusion

Flash memory is a non-volatile storage medium that comes in two main types: NAND and NOR. Most mass-storage devices, including SD cards, make use of NAND flash. Devices with embedded flash memory are quite complex because of the inherent issues associated with flash, such as erase cycles and accumulation of invalid pages. SD cards solve these issues by embedding a micro-controller that regulates access to the actual flash memory. Moreover, sophisticated multi-bit ECCs are implemented due to the fact that flash memory can be quite unreliable without these protective measures.

The answer to the second question (how reliable are SD cards in general?) becomes obvious with these findings. SD cards manufactures take multiple measures to increase the reliability of their products, from implementing sophisticated ECC algorithms to using multiple techniques to prolong the life of the flash memory such as wear leveling. However, given that the storage system failed in Delfi-n3xt is an indication that another approach is required to ensure that the same does not happen again.
Literature study
The storage system will be used to store instruments recorded by the scientific apparatuses on board DelFFi. Without useful information to store, there is naturally no need for a storage system. Just like in normal operating systems that provide different functions to applications running on top of them, the storage system must provide its services to other applications. Hence, an application programming interface (API) must be developed to allow access to the storage services.

This chapter describes the user cases, functional and non-function requirements of the storage system. It also discusses the use case scenarios which are fully included in the requirements document found in chapter C of the appendices.

4.1 USE CASES

Figure 4.1 shows a simplified version of the use cases for the API. The full use cases diagram is included in the requirements document found in attachment C. The uses cases are grouped into three categories:

a) file operations; b) folder management; and c) system statistics.
Sharp readers might have seen that the use cases imply that the measurements will be stored in files that can be grouped in folders as is typical in file systems. That was chosen because this is the canonical way file systems store their data in and there is no need to reinvent the wheel by developing our own structure for storing the measurements. Moreover, using familiar concepts such as files and folders to describes the uses cases make them accessible to more stakeholders.

The first two categories are fairly obvious and don’t requires any elaboration, especially because any user with basic computer knowledge should be familiar with them. System statistics most notably allow acquiring the amount of single event upsets (SEUs) per SD card and the total SEUs as well.

Although normal users of a storage system don’t pay attention to reliability measures, researchers and engineers working on DelFFI will most definitely require such functionality from the system. These metrics are valuable for evaluating the effectiveness of the implemented fault-tolerance. Moreover, future research can be conducted while making use of the data to improve next iterations of the system. Hence, system statistics are included as a part of the use cases.
4.2 Functional requirements

After establishing the first version of the use cases diagram, they were used to define the functional requirements. There is an important relationship between use cases and functional requirements: the former is a subset of the latter. Use cases therefore already define a part of the functional requirements, but only those which a user will interact with. These are called behavioural requirements. The rest of the functional requirements describe what the system does internally.

MoSCoW analysis was used to prioritize the functional requirements \[8\]. There are four priorities defined in the MoSCoW method:

- **M** (must have): What must be delivered?
- **S** (should have): What should be delivered as a high priority but not essential?
- **C** (could have): What could be delivered if there was available time / budget / resource?
- **W** (would have): What would be delivered all other requirements have been finished?

The prioritization was done based on feedback from Japser Bouwmeester on the first revisions of the requirements document. For example, it is essential that the storage system allows storing and reading of data in files, but organizing the data into folders is of less importance. Therefore, operations relating to storing and reading data in files will be a Must have.

The full list of functional requirements can be found in appendix in C. Below a selected set is presented that is related to file operations which shows that illustrates the use of all priorities available in the MoSCoW method:

1. The storage system will allow writing data to a file. (M)
2. The storage system will allow reading data from a file. (M)
3. The storage system will allow reading data from a file on an individual SD card. (S)
4. The storage system will allow checking if a file exists. (S)
5. The storage system will allow checking if a file exists on an individual SD card. (C)

6. The storage system will allow reading file attributes. The returned attributes must include: a) file size; b) file visibility (hidden or not); c) read mode (read-only or not); and d) last modification date and time. (W)

4.3 **NON-FUNCTIONAL REQUIREMENTS**

Non-functional requirements define the quality assurances that a system provide. They describe how a system works as opposed to the functional requirements which establish what a system does. For grouping the non-functional requirements, the classification provided by the quality model in the ISO/IEC 9126 standard is followed [31].

Some non-functional requirements are listed below for illustration purposes. The full list of non-functional requirements can be found in appendix in C.

**Functionality**

**Accuracy**

1. The storage system shall record last modification time of files to second-precision. The precision is restricted since the measurements data will include the actual time of performing the measurement. The time is taken from the RTC module integrated into the OBC. Therefore, files don’t have to record the time with high accuracy.

**Functionality Compliance**

2. The storage system shall adhere to the SD specification version 4.10 for communicating with the SD cards.

3. The storage system shall adhere to the FAT32 specification for designing and implementing the file system.

**Reliability**

**Fault tolerance**

4. The storage system shall detect a maximum of 2 bit-flips in a block.

5. The storage system shall correct a maximum of 1 bit-flip in a block.
4.4 Use case scenarios

**Usability**

**Understandability**

6. The storage system shall have an API that imitates the file functions provided in the `stdio.h`. Since most programmers are accustomed to the C-API of working with files, the API in DelFFi will follow the same conventions to make it accessible and easy to use. For example, to store data into a file, the file must be opened, data can then be written to the file and finally the file has to be closed.

**Efficiency**

**Time behaviour**

7. The storage system shall handle writing 2 kbit of data in under 200 ms.

**Resource behaviour**

8. The storage system shall consume a maximum of 22 kB from the flash memory on board the MSP430F2418 micro-controller for its code. This ensures that the file system doesn’t use all the space on the OBC and that enough space is left for the application code.

4.4 Use case scenarios

Use case scenarios are in essence a detailed elaboration of the use cases. The scenarios describe the steps performed by an actor to complete a use case, either successfully or via an alternative flow in case of errors. Use case scenarios tend to be quite lengthy since they describe the steps in details. Hence, only one use case scenario is presented below that describes writing data to a file. The rest of the scenarios can be found in chapter C.

4.4.1 Scenario: writing or appending data to file

**Brief description**  This use case describes how a user can write or append data to a file.

**Actors**

1. User
System requirements

PRECONDITIONS

1. The storage system is on and operational.
2. The data to be written exists in a buffer.

MAIN FLOW OF STEPS

1. The user acquires a file handle with the appropriate parameters.
2. The write function is called. The file handle is supplied together with the buffer to the write function.
3. All SD cards write the data from the buffer to the file.
4. The amount of bytes written is returned. In this case it'll equal the buffer size.
5. The user releases the file handle.
6. The use case ends successfully.

ALTERNATIVE FLOW

1. **File handle can’t be acquired**
   If in step 1 an error occurs while acquiring the file handle, for example because the file is read-only, then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.

2. **There isn’t enough free space on the SD cards**
   If in step 2 the check for free space fails (refer to special requirement 1), then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.

3. **The file attributes can’t be updated**
   If in step 2 an error occurs while updating the file attributes, for example the new size, then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.

4. **An acquired file handle can’t be released**
   If in step 3 an error occurs while releasing the file handle, for example because of an SD card timeout, then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.
4.4 Use case scenarios

**Special requirements**

1. The free space on all SD cards is always the same. Therefore, the buffer size must not exceed the free space on any SD card.

**Post-conditions**

1. **Successful completion**
   
   (a) The data is written the file.
   
   (b) The file size and modification time are updated accordingly.
   
   (c) The ECC for the blocks occupied by the file are calculated and stored.
   
   (d) The data buffer is unchanged.

2. **Failure condition**

   (a) The data buffer is unchanged.
System requirements
We begin this chapter by describing the architectural choice for the storage system that was based on the analysis done in chapter 3. Then, the chapter discusses why there was no need to implement software ECC. The following section compares communication protocols supported by SD cards with the choice made for the storage system. Thereafter, the structure of the storage system is descripted. In addition, the file system and partitioning of the SD cards are touched upon. Finally, we delve into the details of the software design including the packages and classes used to describe it.

5.1 Fault-tolerance through hardware TMR

After exploring the effects of radiation on electronics in general and on flash memory in particular in chapter 3 and determining that some measures must be taken to avoid data loss due to SEUs, we focus on discussing the methods for achieving fault-tolerance for DelFFi’s storage system.

Fault-tolerant systems are typically realized using redundancy where an odd number $N$ of resources, e.g. processors or storage mediums, bigger than 1 are voting on the
validity of the data. With binary data, choosing $N = 3$ to recover bit-errors allows all
conflictions to be resolved using majority voting. For sequences of bits with length $n$, for
example bytes, a majority vote can only occur if $N$ is an odd number such that $N > n$.
For example, a simplistic system with 2-bit output requires generating the output from 5
independent units to get a majority vote before outputting the overall result. If this sys-
tem had only used four or less units, there is a chance that each unit will have a different
result thereby constituting a failure.

SEUs almost always occur as single-bit flips in sequences of bits. The double-bit error
rate in SRAMS is only 1-5% of the single-bit error rate [23, Section 1.8]. Given that flash
memory is 3 to 5 times less susceptible to SEUs, the chance of double-bit errors in the
storage system is minuscule. In addition, SD cards already implement multi-bit ECCs
in their controllers rendering chances of data corruption astronomically low. Therefore,
double-bit flips, and multi-bit flips in general, are not considered a real danger.

Nimmagadda [25] developed a file system for an SD card that will be on board a
nano-satellite using a more recent version of the same development board used in this
project. Given the resemblance of his project to ours, evaluating the implementation
described in his thesis is wise. One SD card was used for storage with triple modular re-
dundancy (TMR) of data as the fault-tolerance mechanism; three copies of each file are
stored to facilitate error recovery with no software ECC. The thesis doesn’t include the
reasons behind the choice for that particular implementation.

There is one issue with the approach taken by Nimmagadda to provide fault-tolerance.
Schwartz et al. [30] showed that the bulk of SEUs on a flash memory was due to control
circuitry. In addition, the complexity of tasks performed by embedded controllers on
SD cards far surpasses that of the control logic inside a 1997 flash memory thereby in-
creasing the impact of errors. Clearly, redundancy of data on the same SD card doesn’t
protect against SEUs in embedded controllers inside SD cards. Therefore, the level of
fault-tolerance afforded by data redundancy on a single SD card is deemed insufficient
for DelFFi’s storage system.

Satellites make use of hardware redundancy to counter errors induced by SEUs [26].
Based on these approaches, the best choice for the storage system will also be to use
hardware redundancy to provide robust fault-tolerance. As a consequence, it was de-
cided to employ a form of triple modular redundancy (TMR) with hardware whereby 3
SD cards will be used to store the data.

Although this deviates from the original plan of the project to use a single SD card,
the results of the analysis done in the previous chapter with the information given above
justified the choice. This change was discussed with both stakeholders, Jasper Bouwmeester and DelFFI’s project manager and they agreed to continue with this architecture. However before agreeing, it was asked to provide an estimation of the power consumption that this new architecture has.

### 5.1.1 Power consumption estimation

Devices with a limited amount of power such as cell phones and laptops measure the power consumption of their software to be able to provide users with statistics regarding how much longer they can run. In a satellite, it is even more important to know the power consumption since its power system can’t be extended with a bigger battery once the satellite is in orbit. To account for the power consumption of the new architecture of the file system, an estimation was made of how much power the system will consume for writing a block of data.

A SanDisk SD card uses a maximum of 100 mA for both reading and writing [10, Section 2.1]. However, typically only 350 µA is consumed while the card is idle. Since SD cards generally require a voltage of 3.3 V, the power consumption becomes:

\[
P_{\text{work}} = 100 \cdot 10^{-3} \times 3.3 = 330 \text{ mW} \\
P_{\text{idle}} = 350 \cdot 10^{-6} \times 3.3 \approx 1 \text{ mW}
\]

where \(P_{\text{work}}\) and \(P_{\text{idle}}\) is the power consumed when doing work (i.e., either reading or writing) and when idle, respectively. Combining the consumption for both modes of operation for an SD card gives the overall expression for power consumption each second:

\[
P(t_{\text{work}}) = t_{\text{work}} \times 330 \times 10^{-3} + (1 - t_{\text{work}}) \times 1 \times 10^{-3}
\]

where \(t_{\text{work}}\) is the fraction of a second in which the SD card is performing. Hence, \(0 \leq t_{\text{work}} \leq 1\).

The on board computer (OBC) has an 8 MHz oscillator thereby limiting the theoretical maximum communication rate to around 1 MB/s. To reach this rate, a bit has to be sent with each cycle, which is practically impossible because of the cost of other operations executed before transmission. Nevertheless, we assume the maximum rate in the calculations.
Only a few hundred bytes will be written to the storage system in DelFFi each second. As a typical data size, 512 B has been chosen to be the baseline for calculating the power consumption. The choice is motivated by the fact that SDHC cards have their block size fixed to 512 B.

Given the transmission rate of 1 MB/s, the period time per bit is 125 ns. Therefore, sending 512 B will take 512 µs and hence it amounts to a power consumption of:

\[
P(512 \cdot 10^{-6}) = 512 \cdot 10^{-6} \times 330 \cdot 10^{-3} + (1 - 512 \cdot 10^{-6}) \times 1 \cdot 10^{-3} \\
= 1.2 \text{ mW}
\]

For 3 SD cards, the power consumption becomes \(3 \times 1.2 = 3.6 \text{ mW}\). The power requirements for the storage system are relatively small due to the fact that only a few kBs are written each second. Therefore, supplying the power shouldn’t be difficult for the satellite’s power system.

5.2 Detecting SD card controller malfunctions

With the architectural change to use 3 SD cards instead of a single one as was planned at the inception of the project, the usefulness of an extra layer of software ECC came into question. Recall that it was defined in the original plan for the project to implement ECC as part of the storage system. The hardware redundancy will correct not only single-bit bit-flips, but is capable to recover from any multi-bit bit-flips. Hence, one might assume that ECC should not be implemented anymore. However, an investigation was carried out to see if there was any extra protection ECC can provide.

The synergistic effect resulting from the combination of redundancy and ECC is used in RAM to combat bit faults and is explained in detail in [19, Section 4.2.1]. In this case, the effect manifests itself in allowing the detection of potential controller malfunctions in addition to the corrections of bit-flips.

5.2.1 SD card internal ECC vs Software Hamming code

If ECC is implemented on the storage system, it will not primarily be to correct single-bit bit-flips; hardware redundancy can correct these errors more effectively especially when a malfunction in the control circuitry of an SD card occurs. The primary objective of ECC on the storage system will be to allow the detection of malfunctions in an
SD card controller for gathering statistics. In the future, these statistics can prove to be very valuable in improving the design and architecture of the storage system for the next generations of nano-satellites.

There are two methods to use ECC for the detection of potential controller malfunctions. The first one relies on the passive, internal implementation of an SD card. When a read operation fails, an SD card doesn’t return any data block. Instead, an error token is returned with the cause of the failure [3, Section 7.2.3]. In this token, one bit specifies whether the error was caused because ECC couldn’t correct a detected error. There is also another bit that indicates an actual internal controller error.

The other method treats the SD card as raw storage blocks and deploys a layer of software calculated ECC bits appended to the actual data. The internal SD card ECC is ignored although it’s passively running since the SD specification doesn’t provide a command to disable it. Hamming codes are commonly chosen because they are optimal in the sense that they require the least amount of redundant bits to correct any single-bit error [22, Section 2]. This means that only the least amount of storage is wasted on correction bits.

To understand how controller malfunctions are detected, an example is given. For simplicity of explanation, it’s assumed that one block fails in the read operation. If after reading all 3 blocks from the SD cards a mismatch is detected, Hamming code is used to correct the mismatched block. Then the blocks are compared again. If the comparison fails here, it means that even after correcting the bit-flip, the block is not valid for another reason. The cause of the error can be one of two things; either a multi-bit bit-flip has happened that tricked Hamming code into reporting a corrected error or a controller malfunction has occurred. The chances of the former has already been stated above to be minuscule. However, we can’t exclude this possibility. Therefore, the only thing that can be concluded is that a potential controller malfunction has happened.

**Fitting Hamming code into SD card blocks**

Hamming code requires $2N$ redundant bits to protect $2^N$ bits. A *code block* is defined as the concatenation of the data bits and the redundant bits. Hence, the size of a code block is $2^N + 2N$. Since each data block is 512 byte on an SD card, there are $2^{12}$ bits in one data block.

The optimal design of software ECC allows storing as much data bits as possible while minimizing the redundant bits. It’s clear that $2^{12}$ data bits with their redundant bits can’t fit in one block. The obvious solution is to decrease the data bits to $2^{11}$, but this wastes a
A lot of storage space; almost half the data block will be wasted - 2026 bits from 4069 bits to be exact.

A better variation stores multiple code blocks in one data block to waste the minimum space possible. To find the optimal number of code blocks, a function is defined that calculates the wasted number of bits:

\[ f: \mathbb{N}^2 \rightarrow \mathbb{N} \]

\[ f(K, N) = 2^{12} - K(2^N + 2N) \]

where \( K \) is the number of code blocks within one data block. The optimal solution is the one that minimizes both \( f(K, N) \) and \( K \) while maximizing \( N \). Minimizing \( K \) means that we are looking for the most CPU-efficient number of code blocks that runs the calculation of code blocks the least amount of times per data block.

The best choices were determined empirically to be \( N = 11 \) and \( K = 2 \). In other words, each data block contains 2 code blocks and each code block protects \( 2^{11} \) bits of data. With these numbers, only 44 bits per data block is left unused or almost 1.1%. To make sure all data is written, the excess \( 2^{12} - 2^{11} = 2 \) bits that don’t fit in a data block will need to be stored on a new data block to facilitate reconstructing the required block upon a read operation.

**Comparison and verdict**

Both methods have their advantages and disadvantages. With the internal SD card ECC, only a failed attempt at correcting a detected error is reported. This will limit the amount of statistics that can be gathered from such a failure. On the other hand, the software Hamming code will allow recording statistics for error detection, error correction and failures of the algorithm. These are two extra metrics that can’t be recorded when using the internal ECC.

When it comes to extra resources usage, the first method uses none. First, storage space is considered. Data blocks on an SD card actually have extra user-unaccessible bits at the end for storing the ECC redundant bits. As explained in the previous section, 1.1% of each block is wasted. For example, in a 4 GB SD card, around 44 MB is left unoccupied. Second, there is no computational overhead on the OBC associated with the first method since the calculations are performed on the SD card. Calculating the generator matrix for a Hamming block and performing the detection and correction obviously
consume some clock cycles on the OBC.

Another aspect to consider is the fact that before the Hamming code tries to detect and correct errors, the passive internal ECC of an SD card would have been applied to the data. If the internal ECC corrected the errors, running our Hamming code will be futile. However, when the internal ECC fails, the SD specification requires an error token to be returned without the actual data! Therefore, Hamming code will not get a chance to try and repair the error. In this light, the software Hamming code only acts as a verification mechanism to whether the internal ECC actually corrected the errors. But since hardware redundancy is already incorporated into the storage system, the other 2 SD cards can be used to fulfill the same role.

Table 5.1: Comparison between two methods for implementing ECC on the storage system.

<table>
<thead>
<tr>
<th>Criterium</th>
<th>Internal ECC</th>
<th>Software ECC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available metrics for statistics</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Wasted storage space</td>
<td>0%</td>
<td>1.1%</td>
</tr>
<tr>
<td>Uses OBC processing power</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Can control ECC algorithm</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 5.1 displays an overview of the important differences between the two methods. Because of the wasted storage space, computational power associated with the second method in addition to the fact that it can’t handle the cases where the internal ECC fails, the first method is chosen to be implemented on DelFFi storage system for detecting potential controller errors.

5.3 DEVELOPMENT HARDWARE

It’s helpful to describe the hardware that is used to develop the storage system before discussing the software structure. This knowledge helps the reader acquire the mindset involved in making the design.

Some of the decisions that were made by DelFFi’s development team before the inception of this project dictated the choice for the hardware that the storage system must use. The software of the storage system has to be developed to run on the TI MSP430F2418 micro-controller shown in figure 5.1. This due to the fact that DelFFi’s OBC includes the micro-controller as its brain. Another decision that was made was to use an SD card
as a storage medium, which entails that the micro-processor must have a way to communicate with the card.

The TI MSP430F2418 belongs to the MSP430 family of low-power micro-controllers. The device has a 16-bit RISC CPU that can operate on 16-bit registers. Included are the standard set of features that developers expect from a modern micro-controller such as hardware timers (including watchdog), A/D and D/A converters, DMA and support for multiple communication buses: UART, I²C and SPI. Its support for SPI is crucial since it’s one of the two buses available for communicating with an SD card, with the second being their proprietary bus.

For developing the software, an existing CubeSat development board (revision B) was used. This board is manufactured by Pumpkin and it’s originally intended for quickly developing nano-satellites in conjunction with their hardware for building satellites. It includes a lot of features that are important for satellites such as a dedicated transceiver socket. Figure 5.2 shows the board.

However, it’s used in this project solely to develop the storage system. Hence, we only need the board to provide us with two functionalities. First, software must be uploaded and run on the micro-controller. This can be done using the JTAG connector integrated on the board. Furthermore, JTAG allows the code to be debugged in real-time, which is very helpful in development. An MSP-FET430UIF USB debug interface, shown in fig. 5.3, can be connected to this connector to upload code to the micro-controller and to perform real-time debugging of the software.

Second, the controller must be able to communicate with the SD card. In the first week of the project it was discovered that all boards of this type contained a design error whereby the SD card socket is incorrectly wired. Before discovering this problem, it was observed that the SD card was responding with strange responses. Multiple SD cards were used to eliminate the possibility that the SD card was at fault. Still, all SD cards showed the same behaviour. We used an oscilloscope to monitor the signals that were sent by a test program that was written specifically to investigate this issue. After monitoring all the pins on the SD card socket it was determined that the manufacturer reversed the connections on the socket.

We contacted the manufacturer through email to verify that the discovery of the design fault. They confirmed it and said that all revision B boards contained this error.
Figure 5.2: An overview of the development board used to develop the storage system. Sections of the board are highlighted to indicate their functionality. Notice that the SD card is unusable on all boards of this type due to design error by the manufacturer.

However, since this board was very old, it was not covered anymore by the warranty and therefore they can’t send us a board of revision C where the issue was resolved.

As a consequence of the design fault discovery, the on-board socket can’t be used for development purposes. To deal with this problem, a bread board combined with an external SD card socket was connected to the micro-controller via the IO pins shown in the top right corner of fig. 5.2. The external SD card socket was soldered to its own breakout board to simplify inserting it into the bread board. The breakout board includes three SD card sockets and is shown in fig. 5.4.

Finally, three SanDisk SD cards with a capacity of 16GB are used as the storage media. On DelFFi a smaller variant of these SD cards will be used, namely micro SD cards to save on space inside the body of the satellites. Both types support the same commands and therefore it is not an issue to develop the storage system using normal SD cards instead of micro SD cards.
5.4 SD card communication protocol

This section briefly describes the supported buses for communicating with an SD Card, compares them and concludes with the chosen protocol for DelFFi.
5.4 SD card communication protocol

5.4.1 Card capacities

SD Cards come in three capacity classifications: SD Standard Capacity (SDSC), SD High Capacity (SDHC) and SD eXtended Capacity (SDXC). The classifications support capacities of up to 2GB, 32GB and 2TB, respectively. An important thing to note is that the supported communication protocols differ with capacity classifications. The support is indicated in the next section.

5.4.2 Communication protocols

The Physical Layer Simplified Specification [3] defines three protocols, called Bus Protocol in the document, for interfacing with an SD Card. The protocols are: Serial Peripheral Interface (SPI) protocol, SD bus protocol and Ultra High Speed II (UHS-II) protocol. Each protocol only works with its respective physical bus. For example, the SD protocol can only be used to exchange data on top of the SD bus.

Serial Peripheral Interface (SPI) protocol

The SPI bus is a 4-wire bus with a master/slave architecture used for full-duplex synchronous serial communication. The specification defines a protocol for transmitting data back and forth between a host and the card using this bus with a transfer rate of up to 3.1MB/s. Support for this bus protocol is mandatory by the specification across all capacity classifications. Furthermore, no license is required for implementing the SPI protocol.

SD bus protocol

The SD bus protocol is a full-duplex proprietary protocol developed by the SD Card Association that uses a set of either 3, 4, 5 or 6 wires for its bus of which 1, 2, 3 or 4 are used for data exchange, respectively. For example, in a 5-wire configuration for the bus, 3 wires will be used for exchanging data along 2 wires used for the clock and command/response. Data can be exchanged with a maximum throughput of 25MB/s, which is 8 times higher than the throughput of the SPI protocol. However, this speed is only achievable with the high speed mode of operation that uses a 50MHz clock together with 4 data-wires. The default speed mode can use a maximal clock frequency of 25Mhz. When the default speed mode is used to transfer data on 1 wire, the SD bus has the same transfer rate limit as the SPI, which is 3.1MB/s.
Support for this protocol in its default speed mode is mandated by the specification from all capacity classifications; however, it's optional for the high speed mode. A license costing $1000 is required for the usage of this protocol. Therefore, implementation information is classified and not present in the simplified version of the specification that's freely accessible.

**Ultra High Speed (UHS) II protocol**

SDHC and SDXC cards of high capacity need to use the UHS-II protocol that can accommodate transfer speeds going as high as 312MB/s when operated in its half-duplex mode. This protocol is also proprietary and therefore the same restrictions apply for obtaining the full version of its specification. Only a simplified version is freely available.

To achieve such high throughput, this protocol uses the 4 data-wires configuration of the SD bus for data exchange. The throughput increases with the clock frequency used until it reaches the aforementioned theoretical maximum. Unlike the previous two protocols, this protocol can only be used with cards that implement the UHS standard.

### 5.4.3 Protocols comparison

Table 5.2 shows a comparison between the major differences in the communication protocols. Clearly UHS-II is the winner when it comes to throughput. However, according to the development team the data generated by DelFFi is measured in KB/s. Hence, the high throughput afforded by this protocol is excessive for DelFFi. Furthermore, using UHS-II will greatly limit the choice for an SD Card. For the given reasons UHS-II and throughput will not be considered anymore in the rest of this section.

<table>
<thead>
<tr>
<th>Criterium</th>
<th>SPI</th>
<th>SD bus</th>
<th>UHS-II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus wires</td>
<td>4</td>
<td>4, 5 or 6</td>
<td>6</td>
</tr>
<tr>
<td>Maximum throughput</td>
<td>3.1MB/s</td>
<td>25MB/s</td>
<td>312MB/s</td>
</tr>
<tr>
<td>Supported in all cards</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>License required</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
The SPI bus protocol has multiple advantages compared to the SD bus protocol. Most, if not all, modern microcontrollers provide dedicated hardware support for implementing serial data exchange using the SPI bus. The TI MSP430 family of microcontrollers, of which the MSP430F2418 will be the one orchestrating operations on the On-Board Computer (OBC) in DelFFi, is no exception. This allows developing an efficient, reliable implementation which can be quite hard to realize with bitbanging. The same is not true for the SD bus.

Furthermore, very little can be said about the SD bus in general as the critical implementation information is classified. The last is another advantage of the SPI protocol; it’s use is free of charge and doesn’t require obtaining a license, which makes it an enticing option for a non-commercial, educational project like DelFFi.

The only advantage for using the SD bus protocol, excluding throughput as already mentioned, will be that its bus require one less wire. SPI uses a wire called Chip Select (CS) to specify the master’s intent to transmit data, which the SD bus doesn’t have to do. This will translate into saving a little more space on the nano-satellites.

### 5.4.4 SD CARD CONTROLLERS

A viable option to make use of the SD bus in DelFFi without obtaining a license would be to employ an SD card controller. The manufacturer of the controller would have had to pay for the license and hence relieving us from this responsibility. These controllers conform to the specifications and behave as a proxy between the SD card socket and the CPU allowing communication via the SD bus. This option is however not explored further as it doesn’t in it self provide any extra advantages.

In fact, the space that would have been saved by using the SD bus will get consumed by the controller chip, and most probably even more space will be required. Furthermore, most of the SD card controllers are optimized for transfer speed. Iterating the fact that throughput is not a criteria, the major benefit of such a controller ceases to exist.

### 5.4.5 VERDICT

Considering the amount of advantages SPI has, the smidgen of space saved by the SD bus doesn’t justify acquiring a $1000 license. Employing an SD card controller is also not beneficial as mitigating the cost of the license in order to use the SD bus will actually result in losing space. Therefore, the SPI protocol is deemed to be the most suitable communication method for DelFFi.
5.5 Structure of the storage system

Figure 5.5 shows the structure of the storage system that allows it to be fault-tolerant. At the bottom, there are three identical SD cards. Each SD card will contain an exact replica of the data on the other two. That way, if one block on an SD card has different information, the other two can be used to restore the 3 SD card to have the same content.

![Diagram showing the structure of the storage system](image)

**Figure 5.5**: Hierarchical view of the logical layers in the the storage system. These layers work together to provide fault-tolerance.

A driver will be written that will communicate individually with each SD card and to fetch data from all of them. When the data has been read, all the three blocks, one from each card, will be delivered to a voter. The voter chooses one of the three blocks and passes that block to the file system which interprets the content to file system structures such as files and folders. It’s the task of the file system to keep track of which blocks belong to which files and things of this nature.

5.5.1 FAT32 file system

FAT32 is one of, if not the must, used file systems in the world. All major operating systems support FAT32 and as a consequence its used on removable storage media such as USB sticks to increase operability. Having this long history behind FAT32 and its widespread use gives confidence in its robustness and allows it to be suitable for systems where reliability is of high priority. Furthermore,FAT32 is relatively compact in
size due to the fact that it has its origins in the late 1970s and early 1980s when computers weren’t as powerful and resourceful as they are today \[9\]. This is an advantage on embedded systems that are also restricted on resources.

Moreover, it is logically simple as can be seen in fig. 5.6. The file system begins with a boot sector and some reserved sectors, followed the file allocation table (FAT) and its backup and then comes the actual user data. This means that the file system consumes only a small amount of a storage medium leaving most of the space for the actual data. For a satellite where it’s impossible to increase the size of the SD cards after the satellite has been launched, that fact is very advantageous.

![Diagram showing the structure of the FAT32 file system](image)

**Figure 5.6:** Layout of the FAT32 file system showing the major parts.

The file allocation table (FAT) contains the mapping between files and their sectors on a storage medium. A folder in FAT32 is simply a unique file with different attributes from a normal file. This simplifies the implementation quite a bit since the same code that will handle file operations can also be used on folders with minor changes.

Due to its simplicity and robustness, FAT32 has been chosen for DelFFi’s storage system.
5.5.2 SD CARDS PARTITIONING

Before being able to utilize FAT32 to store files on a drive, the drive has to be formatted and partitioned. Otherwise, the drive will not contain the layout shown fig. 5.6. As already stated, removable storage media come pre-formatted with FAT32 from their manufacturers. SD cards are no exceptions. However, this presents a problem to the TMR mechanism used to detect and correct errors.

The boot sector contains information that uniquely identifies a drive, such as the boot signature field and the volume ID field. When the boot sector is read from the 3 cards, each buffer content will be different and hence TMR will report a failure. If we fail to read the boot sector of FAT32, the whole file system is rendered unable since the boot sector contains the address of the first block of the FAT. Without a solution, the storage system will fail to read any piece of data unless TMR is disabled, but that defeats the purpose of why we added TMR in the first place, which is to have fault-tolerance.

One solution to this problem is to make the code ignore the changes in boot sector and keep TMR for the rest of the blocks. However, a consequence of this solution is that the boot sector will not have two copies on the other 2 SD cards. If the boot sector is corrupted by an SEU then we will not be able to repair it! The boot sector is one of the most important sectors within the file system and has to be protected. Hence, this solution is not good enough.

A better solution is to format all SD cards to contain the exact same information, including the boot sector. This way TMR will always get the same information. After trying multiple formatting utilities on Windows and Linux, it was determined that such utilities don’t provide an option to specify the content of the boot sector. This is logical since this information is too low-level and most users won’t use it. Furthermore, the FAT32 specification requires the boot signature to be random and hence no formatting utility will have an option to have a fixed signature. However, since the 3 SD cards in the storage system are exact replica of each other, they must all have the same the boot signature.

Since no available tool can be used, it was decided to implement our own utility to format the SD cards to FAT32. The utility will create a single FAT32 partition that makes uses of the whole SD card space. Naturally, users will be able to specify the the boot signature as a parameter.

This utility can easily be developed using function from the Windows API that handle most of the low level work. For example, to format a drive, one function call is needed.
The function is `DeviceIoControl` which needs to be called with the `IOCTL_DISK_CREATE_DISK` parameter. Since the utility is just a few lines of code with function calls to the Windows API, it is not described in detail in the report.

### 5.5.3 Partial Data Scrubbing

Data scrubbing is a technique used to decrease the accumulation of single-bit bit-flips by periodically scanning the data for errors and correcting them. Accumulation of errors happens because the internal ECC of SD cards is applied to the data only on a read operation. That means TMR will not get a chance to detect errors in a block until it's read. If a block is left unread for a long period of time, the amount of bit-flips within that block increases accordingly.

As one can imagine, applying data scrubbing to the whole storage system will eventually become a performance bottleneck as the amount of stored data increases. At one point, even the whole processing period on the OBC will not be sufficient to scan all the blocks; not to mention that we are dealing with 3 SD cards. Therefore, data scrubbing of the storage system is impossible.

However, blocks in the storage system don't have a uniform importance. In other words, errors in some blocks could have catastrophic results to the entire storage system if they are uncorrectable in comparison to other blocks. Partial data scrubbing can be employed in such cases to only scrub blocks with high priority. Figure 5.6 displayed the layout of FAT32 file system. The two most important parts of the FAT32 file system are the boot sector and the File Allocation Table (FAT).

Without these two parts, the storage system becomes nothing more than a large array of zeros and ones. Efficient implementations of FAT32 only read both of these sections from disk in the initialization and then cache the information in memory. FAT is only updated on disk when the size of a file changes or when a new file is created. In all other cases, the memory-cached FAT is used. Hence, depending on the application, FAT might not be accessed on the storage system for a long period of time. Therefore, the boot Sector and FAT will be scrubbed on all SD cards with each OBC cycle to ensure no accumulation of errors occurs.

The layout of FAT32 in fig. 5.6 shows a backup of FAT with no corresponding one for the boot sector. The backup FAT is not required by the FAT32 specification, but formatting applications generally create it regardless because of its importance. In DeFFi storage system there will be 6 replicas of FAT as there are 2 per SD card. To make sure
partial scrubbing doesn’t become a bottleneck when FAT grows with files, the backup FAT will not be scrubbed.

5.6 Software design

One of the challenges while designing the packages was to find a balance between applying good design principles and making it feasible to implement the design on an embedded system. Principles such as the Single Responsibility Principle (SRP), the Open-Close Principle (OCP) and Liskov’s Substitution Principle (LSP) all increase the re-usability and abstraction of the software [21]. However, at the same time they increase the complexity of implementation and impact the performance.

The software has to be implemented on a micro-controller that doesn’t have the same amount of computing power or memory as a normal PC does. Hence, a compromise was chosen to use the minimum amount of classes possible. Another thing of note is that micro-controllers generally don’t support programming languages with Object Oriented Design (OOD) such as C++ or Java. The MSP430F2418 is no exception as the only high-level language supported on it is C. This restrict us from using programming concepts such as interfaces in the design. However, it’s still possible to emulate classes using structs with functions that take a struct as a parameter.

5.6.1 Packages and classes

Figure 5.5 showed the logical structure of the storage system. The upper part of the figure represents the software layers and it contains three entities: the file system, TMR voter and the card driver.

Since each software layer provides a different service, a package was created for each one of them. This increases the modularity of the system and allows future reuse. A hybrid packages diagram is shown in fig. 5.7 that contains both the packages and classes. The normal packages diagram and all the classes diagrams can be found in the design documents appendix in chapter D. There are five packages in total. We describe each of them in a counter clockwise order starting from lower right corner.

Utility package

The utility package provides classes that implement general functionalities. A simple interface to the SPI bus on the micro-controller is implemented by the SPI class. It pro-
Figure 5.7: Packages diagram for the storage system. The diagram shows the classes contained within each package and how they are connected.

provides function to send and receive bytes without having to work directly with registers and pins.

For protecting the transmission of data to the SD card, the SD specification allows enabling CRC verification. There are two types of CRC defined by the specification: CRC-7 for the commands and CRC-16 for the data; the former is a byte long and the latter is 2 bytes long. The CRC class implements the algorithms for computing both types of CRC.

Since some non-functional requirements define constraints on the speed of writing data to the storage system, one must find a method to perform the measurement of code execution speed on the micro-controller. Fortunately, one can use the hardware timer for this purpose. The Timer class provides a simple interface to get how much time in milliseconds has passed since starting the main program. In that sense it’s technically an accumulating timer (i.e., it accumulates the count of milliseconds passed).

As can be seen in fig. 5.5, the only other package that uses the utility package is the SD card driver package. Normally, if a package is only used one other package, they are merged since keeping them separate is an unnecessary complexity. However, we chose to maintain the current design in the name of increasing re-usability because both CRC and SPI are likely to be used by other modules. Furthermore, the functionality of the two packages are logically different and separating them is an application of the Single Responsibility Principle (SRP).
SD card driver package

SD cards provide their storage services through commands defined in the SD specification [3]. The SD protocol for communicating with an SD card is relatively complex because it has to support different types of SD cards (e.g., SDSC and SDHC). Furthermore, the specification has added extensions and amendments to correct bugs which increased the complexity even more. Hence, a driver must be developed that hides all this complexity behind a simple interface.

The **SD Card** class provides that simple interface to an SD card. It contains function to read and write a single block and multiple blocks, respectively. More importantly, it automatically detects the type of the card, its size and enables CRC for secure communication in the initialization function.

![SPI bus configuration for connecting the micro-controller to the SD cards. Notice that the cards share three signals: CLK, MOSI and MISO. The CS signal is used for selection. The arrows at the end of a connection line indicates the direction of data flow.](image)

An instance of this class can be used to communicate with one SD card. Since the storage system uses 3 SD card, we simply create three objects; one for each card. It makes use of the SPI class to transmit the commands and receive the responses. To support connecting the micro-controller to multiple SD cards while allowing it to control them individually, the SPI bus is configured to have one master and multiple slaves as shows in fig. 5.8.

The clock, input and output signals of the SPI bus are shared among the SD cards. The card select signal is used to address each card by asserting that signal for the appropriate card before transmitting a command. Consequently, when the user initializes an
instance of the **SD Card** class, the card select pin must be given as an argument to identify the card.

**TMR package**

This package encapsulates the decision making algorithm for detecting and correcting errors in triple modular redundancy (TMR). It includes three classes: TmrVoter, Comparator and Statistics.

The **TmrVoter** class takes its name from the fact that its functionality is similar to counting votes. In a democracy, when everyone votes for the same thing, there is a consensus. When the majority agree on the same thing, although some might disagree, this constitutes a majority vote. When none agrees, another way has to be found to make a choice.

The same applies for the TmrVoter class in the storage system. It reads three blocks, one from each SD card, and decides which block is the correct one (i.e., the block without errors). If the bits in of all blocks match, any block can be chosen since they are identical and a consensus is found. If two blocks have the same bits while the third differs, one of the two blocks is chosen as the majority vote. In this case, since a block differed from the rest it has to be corrected; the chosen block is written to the SD card from which the different block came to restore all block to the same correct state. If all blocks differ, the TMR voter fails to make a decision.

The **Comparator** class is used internally by the TmrVoter to actually find the majority block. This was done to separate the implementation of the search algorithm from the detection and correction functionality. Another advantage is that the search algorithm can be replaced in the future without having to modify the TmrVoter class. Technically, this is an application of the Strategy design pattern [17].

As was defined in the functional requirements, the storage system must gather statistics of the amount of errors detected in the system. The **Statistics** class keeps track of five counters per SD card; a counter for each type of error that can be detected: read, write, controller, ECC and corrected errors. The first two types are self-explanatory and are reported by the SD card driver when it fails. Controller errors are the internal controller errors of an SD card while ECC errors are reported by an SD card when it fails to correct an error in a block. Both errors have been discussed in length in the previous chapter (refer to section 5.2 for more information). Lastly, a corrected error is an error that was detected by majority voting and was corrected.
File system package

The file system package provides the API to the storage system. The File system class allows users to mount and unmount FAT32 for usage. In addition, it has methods to retrieve statistics about space usage (i.e., the total amount of block consumes by the file system including data). Partial data scrubbing, which is described in section 5.5.3, is also a part of this class’ methods.

All the functionality related to storing data in files and reading it from them is contained within the File class. Further, this class has static methods for renaming, removing and checking the existence of a particular file. The Folder class only contains static methods for managing folder in a similar fashion to what the File class provides.

This package depends internally on the FatFS library which is described next.

FatFs library

Microsoft offers a specification document where FAT32 is explained in detail [9]. A developer can make use of that document to fully implement a set of functions that interact with the file system. However, there are multiple libraries freely available on the web that already implement FAT32. These libraries have the advantage that they have been used by other developers before which helps in improving them and eliminating any bugs they might contain. We chose to use one of these libraries to help with implementing the API of the storage system; the chosen library is called FatFS [15].

FatFs distinguishes itself from the other FAT32 libraries by being tailored and optimized for used on small embedded systems. This is evident by the sheer amount of implementations of FAT32 file systems using this library on different micro-controllers, including AVR and PIC. Moreover, the library avoids the use of expensive calculations internally to improve performance. These reasons made it a good choice to help in implementing the API of the storage system that is encapsulated in the file system package.

5.6.2 Activities diagrams

Although we already alluded to how reading and writing operation are performed, the cooperation between the different packages wasn’t described before. This section discusses how a block is read and written from the 3 SD cards. For reading a block, it also shows where the correction of errors takes place. The descriptions are visually aided by activity diagrams.
Reading a block

The reading of a block will be performed as shown in fig. D.7 which is included in the appendices in chapter D. Activities are color coded to indicate the phase the system is in while reading. Blue activities indicate a read retry after the system has failed to read from an SD card. The addition of a read retry phase increases the reliability since the internal controllers of an SD card could at a given moment be running a long background process which comes in the way of a read operation. By retrying the read we give it a second chance to deliver the data.

After reading a block from the 3 SD cards succeed, the blocks are compared to detect errors and correct them as shown in fig. 5.9. If more than 2 blocks match, then any kind of bit-flip, whether it's single-bit or multi-bit, can be tolerated and the block will be restored to a valid state by majority voting. In the rare chances that the 3 blocks differ, the reading fails. However, for this to happen, two or more bit-flips in more than one SD card has to have happened. As already stated, the chances of such situations are very small. In this case, although discouraged, the user might try to read the individual block from each SD card if that's desirable.
Writing a block

Writing data to the storage system is done in an identical fashion to the reading operation. The activity diagram of this operation, shown in fig. D.8 that can be found in chapter D, is very similar to the reading one. When a data block has to be stored, the block is transferred to the 3 SD cards. Only when all cards have written the block without errors does the writing succeed. In all other cases, the operation is considered to have failed. This guarantees that an SD card remains an exact replica of the other 2. For the afore-mentioned reason, users don’t have the option to write blocks to an individual SD card.
In this chapter we discuss how the architecture and design done in the previous chapter is implemented. In addition, we describe the documentation of the code and some general conventions used throughout the software.

The software was developed in an integrated development environment (IDE) provided by TI for their line of products called CodeComposer. This IDE contains a compiler specifically tailored for TI micro-controllers. Additionally, because the MSP-430 UIF USB debug interface was used for deploying the code, CodeComposer offers debugging capabilities such as breakpoints and real-time code stepping, which are very helpful in developing the software. The debug interface was shown in fig. 5.3 of chapter 5.

6.1 General Coding Conventions

A few conventions were used throughout the code base to overcome some of the issues associated with developing in the C language. The first one unifies the return values of function to the its execution status (i.e., did the function succeed or did an error happen?). The second one is there to avoid name collisions. Both conventions are described
in this section.

### 6.1.1 Execution Status as a Return Value

First, the return values of all functions is going to indicate the status of executing the operation, regardless of the type of the function. For example, let’s take the function for checking file existence in File class. Normally in C, a boolean function such as this one will be declared as follows:

```c
uint8_t file_exists(char* path);
```

The return value in this case is the existence of the file; zero if it does not exist and non-zero otherwise. However, what if an error happens while trying to check the existence? C as a language doesn’t support exceptions for signalling unexpected errors. Therefore, instead of returning the existence, a code represented by a signed integer is returned that describes errors; the result is given back to the user in a pointer parameter. Then, the function will be declared in the following manner:

```c
int8_t file_exists(const char* path, uint8_t *exists);
```

In the implementation of the storage system it was decided to define an `enum` for the result for each package as can be seen in the classes diagram in chapter D. This is more descriptive than using a signed integer; the SD card driver package has the `SDResult` type, the file system package has the `FSResult` type, etcetera.

### 6.1.2 Sudo-namespaces through Prefixing

The second convention was to prefix all functions with `delffi` followed by an abbreviation of the package name. Since C doesn’t support name-spaces, this sudo-namespacing method had to be done to avoid name collision in code. Furthermore, having sudo-namespaces increases the portability of the code base and allows a greater re-usability, especially for general functionalities such as those in the Utility package.

To elaborate on the previous example, the function for checking file existence will actually be declared as shown in listing 6.2.

```c
delffi_fs_result_t delffi_fs_file_exists(const char* path,
                                         uint8_t *exists);
```
The name of the function begins with `delffi`, then comes `fs` which is an abbreviation for file system and finally we append the actual function name. Notice that both the return type and the function now have the prefixes. Although this elongates the functions names, the advantages more than outweigh this issue.

### 6.2 Converting UML to C

When the storage system was design, it was designed with the implementation in mind as already described in section 5.6. As a consequence, it was trivial to convert the interface into functions and structures. Each class was declared in its own header file that was supplemented by an implementation file. Since packages are not supported in C, they were implemented simply as folders in which we group the files of all the classes within that package.

For example, the functions of the SPI class were declared in the `spi.h` file and were implemented in `spi.c`. Both files were added to the `utility` folder of the code base. The same applies for all the other packages and classes.

To implement a class, it is defined as a `struct` that contains all its attributes. Non-static methods of that class were implemented as functions that took a pointer to the `struct` before the actual parameters of the method. In contrast, static methods are class methods and not object methods and hence don’t require passing the pointer to class `struct`. Therefore, they were defined are normal function as was shown in listing 6.2.

Since the checking file existence method is static, lets look at another function as an example to see how non-static methods are implemented, namely the function for setting the block length of the SD card class. Listing 6.3 shows that function together with the `struct` used for the class definition.

**Listing 6.3: Block length setter of an SD card**

```c
typedef struct {
    uint8_t id;
    volatile uint8_t *card_select_port;
    uint16_t card_select_pin;
    uint8_t is_mmc;
    uint32_t block_length;
    delffi_sd_type_t type;
    delffi_sd_result_t last_result;
} delffi_sd_card_t;
```
6.3 Documentation

All structures, enumerations and functions within the code base were documented using the Doxygen comment blocks. Doxygen is a tool that allows generating a documentation suite [11]. The generated suite makes exploring the code base much easier for new developers that are going to make use of the storage system. Furthermore, having a separate documentation of the code, generated from the one inside the code, allows the development team of DelFFi to convert the code into a static library and thereby protecting the code if they wish without sacrificing documentation.

Doxygen defines a structure for comment blocks that allows this tool to parse the code and generate the documentation. To continue with our example, listing 6.4 contains the comment block of the function for setting the block length of the SD card class.

Listing 6.4: Comment block of block length setter

```c
/**
 * Sets the block length of a card.
 *
 * On non high capacity cards it’s possible to change the count of
 * bytes in a block (i.e., length of a block). This function provides
 * that functionality.
 *
 * @note This function has no effect on high capacity SD cards as their
 *       block length is fixed to 512 bytes.
 *
 * @param card an uninitialized card data structure.
 * @param block_length the new length of the block in bytes.
 *
 * @return operation result. See @ref delffi_sd_result_t.
 */
delffi_sd_result_t delffi_sd_set_block_length(delffi_sd_card_t *card,
                                             uint32_t block_length);
```

Notice that there are some keywords within the comment block that helps Doxygen identify some aspects of the code, in this case this were: @note, @param, @return and
6.4 Implementing the storage system

Having a design for a software system makes implementing it a straightforward process. A developer only has to implement the functionality described by the design. This section outlines some important aspects of the implementation of each package of the storage system. The implementation of packages is discussed in the same order employed in the previous chapter to describe their design. Refer to section 5.6.1 for more information.

One thing to note is that only the important and interesting aspects of the implementation are discussed. As a consequence, a description for the implementation of some parts is not included here. For example, the implementation of the Timer class is not discussed since it was trivial. Readers interested in the details of the implementation can consult the code.

6.4.1 Utility package

SPI class

The SPI class was the first one implemented in the storage system. The implementation encapsulated how the micro-controller uses registers to send and receive data behind a C interface. Constants were defined for the register names and flags to allows porting the implementation to another micro-controller in the future, again this is done for the sake of increasing code re-usability.

SPI is a two-way synchronous serial data bus. With each clock pulse, both parties involved in the communication simultaneously send and receive a byte. Technically, the sending of a byte happens on one edge of the clock while the receiving on the other edge. Hence, if we want to send data, we’ll still receive a byte at the same time, but this byte can be ignored. Likewise, to receive data, we must send a byte first that the receiver will probably ignore as it will be sending.

Listing 6.5: Sending and receiving data through SPI
Listing 6.5 displays a code sample from the implementation. First, the definition of the constant for the buffer register together with the data sending function is shown. This function is the heart of the SPI class. Notice how the function returns only after it ensures that the byte has been sent. Second, the receiving function is included as well in the listing. It shows how we first send a dummy byte to allow us to receive the data thereafter.

**CRC class**

Cyclic redundancy check (CRC) is mathematical in nature [11]. As a consequence, it can be challenging to make an efficient implementation. Furthermore, deep understanding behind how it works is required to make a secure implementation. Ross John Anderson, who is Cambridge University’s Head of Cryptography, reported in a paper that the vast majority of failures in cryptographic systems occur because of weak implementations, not because the algorithms are poorly designed [12]. CRC is the backbone of our communication channel with the SD card that protects data exchange; if it fails we can’t guarantee that the right commands and data are being exchanged.

To avoid repeating the same mistake in our implementation of CRC, we looked into
using open source implementations that have a focus on security. A good choice was to look into the code base of the Linux kernel. A lot of developers contribute bug fixes and review the code of Linux as it is one of the most used operating systems globally. Fortunately, both CRC-7 and CRC-16 were found in the source code. Only one minor adjustment had to be done to the CRC-7 implementation since the SD specification requires ending the 7 bits with a constant 1 thereby making the result one byte long (i.e., 8 bits).

Listing 6.6: CRC-7 look up table

```c
/* Table for CRC-7 (polynomial x^7 + x^3 + 1) */
const uint8_t CRC-7 syndrome_table[256] = {
  0x00, 0x09, 0x12, 0x1b, 0x24, 0x2d, 0x36, 0x3f,
  0x48, 0x41, 0x5a, 0x53, 0x6c, 0x65, 0x7e, 0x77,
  0x19, 0x10, 0x0b, 0x02, 0x3d, 0x34, 0x2f, 0x26,
  // The rest of the entries...
};
```

The Linux implementation of CRC relies on pre-calculated tokens stores in a look-up table (LUT). Listing 6.6 shows a snippet of the LUT used in CRC-7 calculation. To calculate the CRC value for a piece of data, this LUT is indexed using bitwise manipulations for each byte of the data. The primary reason for using LUT is to increase the speed of the computation. This optimization is done because real-time implementations can be very inefficient. However, there is one problem with using the LUT based implementation for DelFFi. What happens if an SEU happened in the program memory of the micro-controller that corrupts an entry in the LUT? The answer is that all subsequent CRC computations will be performed incorrectly. Consequently, the whole storage system will fail since the SD card will reject any command we send because its CRC field is wrong.

The program memory of the MSP430F2418 doesn’t contain any redundancy to counter SEUs like we used for the SD cards simply because it’s not space-qualified. One thing to note here is that this corruption possibility of the program memory applies to the whole software of DelFFi, not just to the storage system software or the LUT of the CRC implementation. Hence, a reader might wonder why is not a issue for the main code of the OBC. There are two parts to the answer.

First, recall that code is stored on NOR flash as opposed to data which is stored on NAND flash as was described in section 3.4. The program memory on MSP430F2418 is actually a combination of code memory and data memory. The LUT will be stored in the
data memory which is a NAND flash. The Delfi team didn’t notice any code corruption in all previous missions which could suggest that NOR memory is less prone to errors from SEUs than NAND memory. Second, the main OBC code is executed regularly thereby decreasing the chance of errors accumulation. We already described partial data scrubbing used in the storage system to achieve the same effect (refer to section 5.5.3). The LUT on the other hand will be used only when data exchange happens with the SD cards, which will occur much less frequently.

Consequently, an alternative implementation was needed that computes CRC in real-time in a reasonable time. Naturally, the realtime implementation will be slower than the one using an LUT. This trade off of speed for reliability is justified when the software is to be used in space and hence reliability is much more important. However, the implementation should also not be very slow that it becomes a bottleneck. Looking through the Kernel source code we found the code that was used to generate the LUT used in the primary LUT implementation of CRC. The functions defined in this code were extracted and added to make our real-time CRC-7 and CRC-16 implementations.

Now we had two implementations of CRC, one carries all the computations in real-time and the other relies on an LUT. Both implementations were benchmarked to see if the real-time implementation was very inefficient or not. Fortunately, the difference in speed was small. Computing the CRC-7 of an SD card command (6 bytes) was faster by 0.5 ms with LUT-implementation. The LUT-implementation was also faster by 3 ms when computing the CRC-16 of a data block (512 bytes).

In light of the benchmark results, the real-time implementation of CRC was used in the CRC class.

6.4.2 SD CARD DRIVER PACKAGE

Implementing the driver for the SD card was done by carefully studying the SD specification. It describes how to initialize the card, how to read and write data to it and a lot more.

Although there are open source implementations of SD card driver, the vast majority are tailored for PCs that contain SD card reads. These reads use the proprietary SD bus to maximize throughput, but the storage system will use the SPI bus as was discusses in section 5.4. The code in these implementations can’t be used because the protocols on the two buses are very different.

The small minority of open source implementations using SPI for communication
have reliability issues. It was observed that most of these implementations ignore some steps and errors defined in the SD specification, presumably to improve performance. Hence, it was decided to create a new robust SD card driver for the storage system.

Debugging errors was very difficult in the beginning. SD cards don’t contain any debugging facilities such as a log of errors or a port to probe their internal micro-controller. SPI was our only way to communicate with them. Hence, the responses of the SD cards had to be analysed in the case of an error to debug the problem.

To help with capturing the commands sent to an SD card and its response, a Beagle SPI analyser was used. The analyser is shown in fig. 6.1. It comes with a program that allows capturing all the communication on the SPI bus. With the aid of this analyser, one can see how the SD card react to any command by recording its response.

Using both the SD specification as a guide and the SPI analyser as a debugging tool, the implementation of the driver was completed successfully. The driver was able to perform all the functionality defined in the design.

All errors that an SD card can return as defined in the specification are handled in code. There are a plethora of errors that could happen and handling them could impact the performance negatively. Again, reliability is of utmost importance for the storage system and hence all errors are handled. shows selected parts of error handling from the write block function. The code in the listing is incomplete with comments indicating what the omitted code does to save space.
delffi_sd_result_t
delffi_sd_write_block(delffi_sd_card_t *card, uint32_t address,
   const uint8_t *data)
{
   // Variables and definitions...

   /* Send the write single block command and receive the R1 response */
   command_without_deselect(card, DELFFI_SD_CMD24,
       address, DELFFI_SD_CMD24_RL, response);

   if (response[0] != 0x00) {
      if (did_command_crc_fail(response[0]))
         return DELFFI_SD_RESULT_ERROR_COMMAND_CRC;
      if (response[0] & 0x40)
         return DELFFI_SD_RESULT_ERROR_WRITE_ADDR_MISALIGNED;
      if (response[0] & 0x80)
         return DELFFI_SD_RESULT_ERROR_WRITE_ADDR_OUTBOUNDS;
      return DELFFI_SD_RESULT_ERROR_WRITE_UNKNOWN;
   }

   // Send the data follows by its CRC and get response...

   /* Check the response token */
   switch (((response[0] & 0x0E) >> 1))
   {
      case DELFFI_SD_SPI_WRITE_ACCEPTED:
         ret = DELFFI_SD_RESULT_OK;
         break;
      case DELFFI_SD_SPI_WRITE_ERROR_CRC:
         ret = DELFFI_SD_RESULT_ERROR_WRITE_BLOCK_CRC;
         break;
      case DELFFI_SD_SPI_WRITE_ERROR_WRITE:
         ret = DELFFI_SD_RESULT_ERROR_WRITE_BLOCK;
         break;
      default:
         ret = DELFFI_SD_RESULT_ERROR_WRITE_UNKNOWN;
         break;
   }
In addition, the SD specification also contains some tips on how to optimize a few operations. For example, it recommends pre-erasing some blocks on a card before writing multiple blocks. Hence, the driver also implements a function to allow the user to perform this optimization. The documentation of the pre-erase and write multiple blocks functions helps the user know that fact as can be seen in the listing below.

Listing 6.8: Pre-erase function is provided to speed writing of multiple blocks

```c
/*
* Writes multiple blocks of data to the card.
* @note The specification recommends pre-erasing some blocks before writing multiple block to speed the operation.
* To do that, call @ref delffi_sd_pre_erase before this function.
* @param card an _initialized_ card data structure
* @param address the block address (must be block aligned)
* @param data the data buffer to write to the card (must contain at least 'blocks_count'+'card->block_length' worth of bytes).
* @param block_count the amount of blocks to write.
* @return operation result. See @ref delffi_sd_result_t.
*/
delffi_sd_result_t delffi_sd_write_blocks(delffi_sd_card_t *card,
                                          uint32_t address,
                                          const uint8_t *data,
                                          uint16_t blocks_count);

/**
* Pre-erases an amount of blocks.
* This function is often used in juncution with
* @ref delffi_sd_write_blocks to speed the write.
* */
```
Implementation

```c
*delpfi_sd_result_t* delffi_sd_pre_erase(*delffi_sd_card_t* card, uint32_t num_blocks);
```

TMR package

The most interesting aspect of the implementation of the TMR package is how it finds the majority block. Specifically, the algorithm used by the Comparator class for this purpose. The Statistics class simply increases counters while the TmrVoter class calls functions from other classes. Hence the last two classes are not discussed here.

For finding a majority block within an array, the Boyer-Moore Majority Vote Algorithm was used [6]. It was chosen because its complexity is $O(n)$, which means that its execution time scales linearly with the amount of blocks in the array. Furthermore, it can be implemented very efficiently as shown by this algorithm’s authors [7]. The implementation of the find majority function follows the one provided by the authors carefully to make sure no overhead is being added that will impact the performance.

The algorithm requires us to provide a method to compare two blocks. Normally in C, one can use a standard function such as `memcmp` for this purpose. However, this function is not available on the micro-controller. This forced us to add our own comparison function to the Comparator class as is shown in the class diagram in fig. D.5. The function simply compares the byte content of the buffers as can be seen in listing 6.9.

Listing 6.9: Block comparison function

```c
uint8_t delffi_tmr_compare_buffers(uint8_t* first, uint8_t* second, uint16_t length)
{
    while(length--)
        if (first[length] != second[length]) return 0;
    return 1;
}
```

File system package
The FatFS library was chosen in the design phase to aid in implementing the file system API. Section 5.6.1 has more information about this subject. The functions in the file system proxy their parameters to the library that does the actual work. As an example, the implementation of the function for reading data from a file is discussed. The same method is used for implementing all the other functions.

Listing 6.10 shows the file read function. Notice how the error codes of library are converted to the error codes of the file system package. This encapsulation hides the usage of the library from external users and allows changing the implementation in the future if needed without affecting the interface, which improves re-usability.

```c
#include <fatfs.h>

delffi_fs_result_t delffi_fs_file_read(delffi_fs_file_t *file,
                                         uint8_t data[],
                                         uint16_t bytes_count,
                                         uint16_t* bytes_read)
{

    FRESULT result = f_read(file, data, bytes_count, bytes_read);
    if (result != FR_OK) {
        if (result == FR_DISK_ERR)
            return DELFFI_FS_RESULT_ERROR_SD_CARDS;
        if (result == FR_INVALID_OBJECT)
            return DELFFI_FS_RESULT_ERROR_INVALID_STRUCTURE;

        return DELFFI_FS_RESULT_ERROR;
    }

    return DELFFI_FS_RESULT_OK;
}
```

Although FatFS was sufficient implement all the functionality required from the file system API, there was still one function that wasn’t directly implementable. Specifically, the partial data scrubbing function. To implement the data scrubbing operation described in section 5.5.3, the library had to be extended with a function that accesses the internal data structures. After accessing these structures, it invalidates the cached boot sector data and FAT table to force the library to re-load them from the SD cards, thereby forcing a read operation on all SD cards.
Testing & measurements

Three types of tests are performed on DelFFi’s storage system: unit testing, performance testing and integration testing. This chapter goes through each one of them and discuss the approach used to perform the tests. Also, the results of each test type are reported. The full test plan can be found in chapter E while the procedure of performing the tests and their results are included in chapter F.

7.1 Unit Testing

Unit testing is used to test the code on a high level and to make sure that an interface does what a user expects from it. Google’s googletest library is used to write the unit tests since it provides a robust framework in addition to a tests runner that supports multiple operating system.

The tests will be run on a PC and not on the micro-controller since they test the interface. As a consequence, we are free to write the tests in a language such as C++ that support compiling C code while providing extra features thereby simplifying the process of writing tests. Integration tests, which will be described in the following sections, are
the ones that ensure the code works on the hardware.

All the packages of the storage system are unit tested. For each condition that changes the result of a function a test case is written. This helps in maximizing the test coverage.

7.1.1 Approach

Google’s testing framework, called googletest, comes with template project files for different IDE’s to help set up an environment to write and run tests. The template project for Visual Studio was used to write the unit tests since the development of the storage system took place on a Windows machine.

All of the tests for the Utility package were straightforward assertions of expectations from functions. However, the same can’t be said when it came to testing the SD card driver and the file system. Both of these packages depend on other packages and expects them to provide services. The SD card driver requires the SPI package to send the issued command and expects to gets responses back. Moreover, the SD card driver needs the responses to comply with the SD specification to function properly and hence this will require having the actual hardware (i.e., the SD card) be part of the testing process.

The file system package bases its services on the TMR package which in turn makes use of the SD card driver. This means that neither the card driver nor the file system could be tested in isolation which is the goal of a unit test as the name implies.

As customary in unit testing, stubs were employed to resolve that issue and to allow isolating the testing of the modules. For the card driver, a stubbed version of the SPI interface was implemented using C++ containers to simulate sending and receiving data. Listing F.1 shows how the implementation was accomplished. Notice the two containers sent and received that are defined in the code.

Listing 7.1: Stubbed implementation of the SPI module for testing the card driver

```cpp
// Stub out the SPI protocol implementation
static delffi_sd_card_t card;
static std::vector<uint8_t> sent;
static std::queue<uint8_t> received;

void delffi_spi_send(uint8_t byte) {
    // Only add byte to container if the card is selected
    if ( !(card.card_select_port & card.card_select_pin) )
        sent.push_back(byte);
}
```
Listing F.2 displays how both containers are used in the tests. In each test case, the `received` container is first filled with the data that the driver expects to receive from the card for it to function correctly. Then, expectation statements from the testing framework are used to test whether the driver sends the correct commands by inspecting the `sent` container. The `sent_bytes(n)` method returns the last n bytes from the `sent` container. Variables in the code beginning with `_cmd` are arrays of bytes that contain the right sequence of bytes for a given command. For example, here is one of those commands:

```c
_vec0 = vector<uint8_t>({ 0x40, 0x00, 0x00, 0x00, 0x00, 0x95, 0xFF });
```

Listing 7.2: One test case of the SD card driver

```c
delffi_sd_result_t result;

received << 0x01; // response of CMD0
received << 0x01 << 0x00 << 0x00 << 0x01 << 0xAA; // response of CMD8
received << 0x01 << 0x00 << 0xFF << 0x80 << 0x00; // response of CMD58
received << 0x00; // *incorrect* response of CMD59

result = delffi_sd_init(&card);

EXPECT_EQ(sent_bytes(7), _cmd0);
EXPECT_EQ(sent_bytes(7), _cmd8);
EXPECT_EQ(sent_bytes(7), _cmd58);
EXPECT_EQ(sent_bytes(7), _cmd59);

EXPECT_EQ(result, DELFFI_SD_RESULT_ERROR_ENABLE_CRC);
```

For testing the file system, a similar approach was used by providing a virtual FAT32 drive using a temporary file formatted with our `VirtualDrive` class developed as part of the formatter program. Before each test, the temporary file is created, the test is then run and afterwards the file gets deleted. This ensures that tests don’t interfere with each
Testing & measurements

other. Again, all for the sake of tests isolation.

After all the tests were written, Visual Studio compiles them and generates one executable that can be run to see the results.

7.1.2 RESULTS

Table 7.1 shows the amount of unit tests that were developed and their results. To see results of individual tests, the generated executable inside the unit-tests directory of the project code can be run.

<table>
<thead>
<tr>
<th>Package</th>
<th>Tests count</th>
<th>Tests result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD card driver</td>
<td>39</td>
<td>All pass</td>
</tr>
<tr>
<td>File system</td>
<td>65</td>
<td>All pass</td>
</tr>
<tr>
<td>TMR</td>
<td>10</td>
<td>All pass</td>
</tr>
<tr>
<td>Statistics</td>
<td>6</td>
<td>All pass</td>
</tr>
<tr>
<td>CRC (LUT)</td>
<td>6</td>
<td>All pass</td>
</tr>
<tr>
<td>CRC (real-time)</td>
<td>6</td>
<td>All pass</td>
</tr>
<tr>
<td></td>
<td>132</td>
<td>All pass</td>
</tr>
</tbody>
</table>

7.2 PERFORMANCE TESTING

To make sure that the system meets the speed requirements, performance testing is performed on the SD card driver and the file system functions. The hardware timer of the MSP430F2418 controller is used to measure the execution time in the manner described in section 5.6.1.

Three parts of the software have been empirically identified to be a potential speed bottleneck:

1. Calculation of CRC

2. Reading and writing blocks to the SD card

3. Reading and writing files to the file system with TMR
The last item includes the execution time of the second one. For example, to read a file from the storage system, blocks will be read from the 3 SD cards. However, reading a file and reading a block from the SD card are two tasks of two different packages: the file system and the SD card driver, respectively. Therefore, it was chosen to test both of them to have performance tests for both modules.

7.2.1 Approach

Before running a tested item, a function from the Timer class is used to get the current milliseconds count and this value is stored in a variable. Then, the test is run. After completion, the difference in time between the current time counter and the stored one is computed.

There are two methods to get the computed execution time. The first one is to use the IDE debugger by putting a breakpoint after computing the time. This approach however required compiling the code which will disable optimization. As a result, the execution time will not be accurate.

Another way is to store the time in a file after the test finishes and then reading the file on a PC from the SD card. This method allows compiling the code in Release mode with optimizations which will result in actual execution time. Therefore, this method was chosen.

A file called performance-test.c was created in the storage system project. This file contains a function per test case that runs an operation while calculating the difference in time between the time before and after the operation call. Afterwards, the execution time is written to a file on the storage system. The file name identifies the test case it corresponds to; for example, the file crc7.txt corresponds to the test case of calculating CRC-7 for a packet.

The code was compiled with maximum optimization enabled. Each test was run 5 times. The average execution time is reported with the deviation next to it.

7.2.2 Results

The following tables show the execution time of the tested items defined in the Test Plan. Results indicating an execution time of 0ms means that the operation was done in less than a millisecond. A packet contains 6 bytes of data while a block 512 bytes.

CRC computation
Testing & measurements

Table 7.2: Execution time of CRC computation

<table>
<thead>
<tr>
<th>Test case</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC7 (computed in real time) of a packet</td>
<td>1ms ±1ms</td>
</tr>
<tr>
<td>CRC16 (computed in real time) of a block</td>
<td>9ms ±1ms</td>
</tr>
</tbody>
</table>

SD card driver

Table 7.3: Execution time of reading and writing a block to one SD card

<table>
<thead>
<tr>
<th>Test case</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading a block from card</td>
<td>13ms ±2ms</td>
</tr>
<tr>
<td>Writing a block to card</td>
<td>13ms ±2ms</td>
</tr>
</tbody>
</table>

File system

Table 7.4: Execution time of a selected function from the file system

<table>
<thead>
<tr>
<th>Test case</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating an empty file</td>
<td>82ms ±2ms</td>
</tr>
<tr>
<td>Creating a file with a block of data</td>
<td>362ms ±5ms</td>
</tr>
<tr>
<td>Reading a block from a file</td>
<td>43ms ±2ms</td>
</tr>
<tr>
<td>Writing a block to a file (without closing)</td>
<td>39ms ±2ms</td>
</tr>
<tr>
<td>Writing a block to a file (with closing)</td>
<td>78ms ±2ms</td>
</tr>
<tr>
<td>Appending a block to a file (without closing)</td>
<td>39ms ±2ms</td>
</tr>
<tr>
<td>Appending a block to a file (with closing)</td>
<td>78ms ±2ms</td>
</tr>
</tbody>
</table>

As can be seen, the most expensive operation is creating a file with some data. However, once the file has been created, both writing and reading is relatively fast.

Since the file system caches the file attributes and only writes them back to the drive once the file is closed, the results of writing and appending are included with and without the overhead of closing. The cost of closing a file is that of writing an extra block, which can be confirmed from the results and is to be expected since the file attributes are updated using one block of data.
Another thing of note is that the overhead of TMR code that compares the blocks is so low that it does not affect the execution time. Writing a block to a file takes exactly three times as much as writing a block to one SD card. Therefore, virtually all of the time is spent writing that one block to the three SD as if there was no code that compares the blocks and looks for mismatches (i.e., TMR voter).

Recall that one of the non-functional requirements from section 4.3 specifies that writing 2 kbit of data has to happen in less than 200 ms. The results in table 7.3 prove that this requirement is met as 2 kbit is equal to 250 bytes which is less than one block.

7.3 System and Integration Testing

Integration testing ensures that there are no compatibility issues between the different packages. In contrast to unit testing that uses stubs to allow testing of individual modules, these tests are run on the micro-controller with the SD cards.

The structure of the storage system defines a hierarchy of layers than begin with the file system API at the top, going through TMR, and ending at the bottom with the 3 SD cards accessed by the SD card driver as previously shown in fig. 5.5.

To make sure that the right parameters are being passed from one layer to the other and that they are correctly integrated, one must test functions that go through the whole hierarchy. Since unit testing already cover all aspects of the interface, such as different error conditions and valid parameters, only the operations that involve multiple layers should be tested.

For the previous reasons, the following set of tests have been devised that test the storage system:

1. When reading a file that is identical on all SD cards, the file is read from all the SD cards no and error is reported.

2. If a block corresponding to a file is corrupted on only one SD card, the system must recover from that error, the correct block must be written to the mismatched SD card and the statistics must be updated correctly.

3. When writing a file, the data must be written to all SD cards.
7.3.1 APPROACH

For the lack of a better mechanism to perform these tests, a slightly complicated method is used. However, since there are only 3 tests, this wasn’t an issue.

First, all the SD cards are inserted sequentially to the PC. Each one gets formatted and the content of the cards is set up appropriately for a test. For example, for the second test item one must ensure that a block on one of the SD cards differs from the other two cards. Listing F.3 shows how the integration test does that based on the index of the inserted card. The _setup_test function is a private method that formats the SD card before running the lambda expression passed as a parameter, which is a feature in the new C++11 standard.

Listing 7.3: First step of correcting blocks integration test on PC

```cpp
void IntegrationTests::correcting_block_test()
{
    _setup_test("correcting blocks", [this](unsigned int card_i) {
        FIL file;
        FRESULT result = f_open(&file, _T("file.txt"),
                                FA_WRITE | FA_CREATE_ALWAYS);

        if (result != FR_OK) {
            stringstream ss;
            ss << "Couldn't create test file. Result: " << result << ".";
            throw runtime_error(ss.str());
        }

        // Change the content of first card
        if (card_i == 0)
            f_puts(_T("Holla"), &file);
        else
            f_puts(_T("Hello"), &file);

        f_close(&file);
    });
}
```

Then, the cards are inserted into the storage system and a piece of test code is run for a particular test. Continuing with our example, that code reads the file from the 3 SD cards.
While running the code on the storage system, an SPI sniffer will be used to monitor the activity of the SPI bus and to ensure that data is being exchanged with the SD cards. For the first tested item, that means monitoring the exchange of data with the 3 SD cards. Finally, the cards are inserted back into the PC to check the results of the test.

### 7.3.2 Results

All 3 tests passed confirming the correctness of the integration process.

In the first test the file was read from all SD cards. This was observed by running the test 3 times and each time we used the SPI analyser to capture the communication with a different SD card.

In the second test it was observed that only the first card (i.e., the card with index zero) received a write command with the corrected text, which was "Hello" in the tests as can be seen in listing F.3.

The third test was confirmed by reading all SD card on a PC and it was observed that all of them contained the correct data.

### 7.4 Measurements

Two important aspects of the storage system were measured. First, the code size of the software had to be determined to see if it meets requirements. In addition, the power consumption was measured and compared to previous calculations that estimated the consumption for the purpose of finding out whether the calculations were correct. More importantly, the results of the power measurement allows DelFFi's development team to make accurate predictions of their applications' power consumption when they store data on the storage system. Both measurements are discussed in this section.

#### 7.4.1 Code size

One of the non-functional requirements in section 4.3 dictates that the size of the storage system software must not exceed 22 kB in size. Hence, it was crucial to keep an eye on the code size as functionality was being developed.

There was some uncertainty in the beginning of the implementation of whether this requirement would be met since the defined size is relatively small considering the amount of functionality that needs to be provided. Furthermore, when the requirement
was defined, no libraries were chosen yet. Hence, the size of FatFS was not taken into account.

We measured the code size by looking through the generated log files of the compiler. One of the log files for the last compilation of the code reported that the entire code size is 15,628 B. In light of this result, we conclude that the requirement was met with a margin that allows extra functionality or future bug fixes to be added.

### 7.4.2 Power Consumption

One of the risks of the project was that the power consumption will be too high for the power system of DelFFi to handle. An estimation was made in section 5.1.1 of the power consumption for writing or reading a block to all 3 SD cards. To verify these results, one must measure the actual power consumption. This was feasible since the faculty of aerospace engineering at the TU Delft had the equipment for such measurements.

Power consumption is calculated from current consumption. Since power equals current multiplied by voltage, knowing current allows the calculation of power. The voltage in this case is the operation voltage of an SD card, which is 3.3 V.

Measuring current is not straightforward. In contrast to the voltage that can be measured using a multi-meter, current must be measured indirectly. There are two main ways to measure current. The first is based on electromagnetics and is associated with the early moving coil (d’Arsonval) meter, and the second is based on the main theory of electricity, Ohm’s law.

The second method is far easier and the equipment in the faculty were designed for performing this kind of measurement. Hence, this is the method we used. The method relies on Ohm’s law stating that current equals voltage divided by resistance. If we use a resistor with a known value of resistance, and measure the voltage drop across this resistor, the current value can be deduced.

For this method to work, the chosen resistor must have a small resistance to allow current to flow through it. Such resistors in a circuit are called shunt resistors to differentiate them from normal resistors that do block current.

A set-up, which is shown in fig. 7.1, was built for measuring the power consumption. For continually measuring the voltage drop across the shunt resistor, a National Instruments data acquisition (DAQ) device was used which can also be seen in the fig. 7.1.

The faculty developed a LabView program that makes used of the DAQ device to automatically convert the voltage drop to current measurement and to record the results.
The user of the program has only to enter the value of the resistance of the shunt resistor. A 5 \( \Omega \) shunt resistor was used as can be seen in fig. 7.2. This value for the shunt resistor was chosen empirically.

Data had to be written to the SD card during the measurement to force it to consume power. For that purpose, a simple application was written that made use of the SD card driver. A suggestion from Jasper Bouwmeester was to measure the power consumption for different block count to see if the consumption scaled linearly or not. Hence, the application had a parameter to specify how many blocks to write in each test.

The recorded current measurements were then converted to power measurements as was described using a Python script. The results are shown in fig. 7.3. The estimation for writing one block was 1.2 mW which is very close to the actual consumption measured at 1.6 mW; the difference is a mere 0.4 mW.

Another thing to note is that the results show indeed that the power scales linearly with block count. This is even more evident when increasing the step between the block
count from 1 to 5 as shown in fig. 7.4. This fact allows the DelFFi team to make more accurate predictions of the power consumption based on the size of the data that their applications are going to save on the storage system.
7.4 Measurements

Figure 7.3: Power consumption for 1 block step

Figure 7.4: Power consumption for 5 blocks step
Testing & measurements
After describing all the phases we went through to develop the storage system, this chapter takes a critical look at the process, document and products that resulted from the work.

8.1 OpenUP evaluation

OpenUP proved to be a good choice for this project. It allowed organizing the development effort into phases and iterations as was shown in section 2.6. Moreover, the same phasing was also used to order the chapters of this thesis. Furthermore, the templates provided by this development methodology on its website helped give structure to the test plan and test results. We believe the reason for the success of OpenUP in this project was because not all its practices were followed such as micro-increments. The disadvantage was however for seeing OpenUP as a guideline and not as a regime that has to be strictly followed is that it added more organizational work which could have been avoided.

In section 2.4.3 we stated that burndown charts will be used to provide metrics for
the next iteration. However, burndown charts were not very effective. Although they were regularly updated, little information from previous burndown charts to the next ones. Furthermore, since the project was done by a single person, the charts just became redundant information that were never used anymore after an iteration. Hence, one could argue that they shouldn’t have been used in the project. Nevertheless, they didn’t have any serious negative impact beside the time consumed in filling and updating them.

Two iterations in the construction phase were enough to develop and test the storage system. However, the elaboration phase should have also been split into two iterations. Specifically, designing the architecture should have been an iteration in itself to avoid tasks overlap.

In hindsight, although the issues described above could have been mitigated with a better planning, OpenUP helped immensely in shaping the path to achieve the goal of the project.

8.2 Documents evaluation

First, we review the requirements document found in chapter C. This document was often updated based on stakeholders feedback, especially the non-functional requirements. That helped in constraining the functionality being developed. However, one lesson that was learned is to get feedback on the requirements as fast as possible after every update. The requirements were discussed with the stakeholders three times. In the second time they had some reservations about some requirements, which came because of updates that were not reviewed by them. This issue was resolved by updating the document once more and describing the reasoning behind some requirements, but this situation could have been avoided by asking for feedback earlier.

The most important thing to remember is that the requirements document served its purpose. It defined the constraints the storage system has to abide. As a consequence, the system was developed in accordance with stakeholders’ expectations.

Next, the test plan and reports are discussed. The test plan attached in chapter E was created with a template provided by OpenUP. It was a great help in structuring the testing process. With the definition of the items to test, the approach and the expected results, the tests were developed with ease. Because the experience of multiple developers went into making the template we used, it worked well for this project without issues.

The results of the unit tests served as a confirmation to the correct implementation of the storage system. The performance tests verified that the system meets some crucial
8.3 Architecture & implementation evaluation

There is some uncertainty with regards to how well the architecture of the storage system will be against SEUs in space. As already discussed in section 5.1, a new architecture was devised for the storage system that differed from the original planed one based on the results of the literature study in chapter 3. In section 3.3 we stated that calculating the SER is not feasible because a powerful acceleator is required which we don’t have access to. As a consequence, section 3.3.2 contained an estimation of the SER. In the end however, it’s an estimation that could be wrong. If the SER is much less than the estimation, then the whole change of architecture to 3 SD cards with hardware TMR might not be necessary. Never the less, the literature study and the calculations all vote in favor of the need for the architecture change.

Another thing that gives us confidence in that we made the right choice by changing the architecture is the fact that the stakeholders approved the changes based on all the analysis done in this project. All calculations justify the Jasper Bouwmeester and Jian Guo are experienced space engineers that have a better insight into electronics’ issues in space. Their approval of the architectural changes means that they believe it has added benefits to DelFFi. In the end, only after DelFFi is in orbit will we know for sure.

The design of the packages of the storage system was done with the implementation in mind as was described in section 5.6. Hence, it was easily implementable with the methods described in section 6.2. The tests report showed that the implementation was done correctly and efficient as it met the speed requirements.

In hindsight the functionality of the storage system could have been split in less packages to simplify the design and to make implementing it easier. However, since the code size requirement was met with some margin this was not an issue.

The code of the storage system was sent to one of the software engineer working on DelFFi for review. The engineer reported that "the work is well done." One of the comments in the review is to provide a simple guide to explain how the storage system can be used without going through this thesis. This will be resolved by updating the documen-
8.4  PROJECT PROBLEM & GOAL EVALUATION

This section discusses the state of the project problem after the completion of the project. In addition, it goes through which goals were achieved and which weren't.

8.4.1  THE STORAGE SYSTEM

The storage system was fully designed, developed and tested. It became the sole focus of the project. An extensive literature study was performed that resulted in an architectural change that influenced all the aspects of the project.

At the inception of the project DelFFi had no service layer. As already described in the introduction of chapter 3, the storage system on DelFFi is a requirement that must be included on all QB50 satellites and is defined in the QB50 system requirements document [29]. Hence, the storage system is one of the main systems on DelFFi. With the development of the storage system a big part of the service layer was completed. As a consequence, one of important components of the service layer was finished.

8.4.2  BLUETOOTH COMMUNICATION

Bluetooth communication was not developed for DelFFi. Multiple factors contributed to the suspension of work on Bluetooth communication.

First, because the storage system had a much bigger priority than Bluetooth communication, its development was suspended to allow the implementation of the new architecture of the storage system. Furthermore, a lot of time was spent in the literature study phase to discover the need for the new architecture.

One can argue that better planning and time management could have resulted in achieving this project goal, however, we believe that all the project time was required to develop a robust and reliable storage system suitable for use on a satellite. Moreover, carrying the extensive tests and measurements shown in chapter 7 added a lot of workload that has to be taken into consideration.

Second, the Bluetooth experiment on DelFFi was approved by the Jian Guo, who is the DelFFi's project manager, 3 weeks before the end of this graduation project. Until that point it wasn't even decided whether any work put into the development of Blue-
tooth communication would have been used on DelFFi. Hence, we focused on the storage system that is a main system on DelFFi.

8.4.3 Summary

In summary, the goal of developing a storage system for DelFFi was achieved. The storage system is a big part in solving the problem of not having a service layer. It was chosen to suspend the development of Bluetooth communication in favor of implementing the new architecture of the storage system.

8.5 Demonstrated competences evaluation

The graduation plan found in chapter A included a set of competences that had to be demonstrated in this project. We describe how this work shows the their demonstration.

First, A5 was fulfilled by gathering the system requirements and creating the requirements document in chapter C. A lot of time was spent of the literature study to allow designing a new architecture for the storage system which was described in chapter 5. This fulfills C8.

Next, C13 was fulfilled by the design choices described in section 5.6. It was chosen to use the minimum amount of classes possible in the design because the system will be used on an embedded system. Furthermore, a non-functional requirement was defined that define a maximum time constraint on the execution speed of writing a block to the SD card in section 4.3. This real-time requirement was met as was shown in section 7.2 along with the execution time of other functions.

The implementation of the storage system described in chapter 6 fulfills D16. Finally, D17 is fulfilled by the 3 types of tests and the measurements described in chapter 7.
Conclusion

TU Delft has started working on its third satellite mission called DeFFi, which is planned for a launch in 2015. It comprises two nano-satellites, called Delta and Phi. There are many subsystems on-board the two satellites: On-Board Computer, Electrical Power Subsystem, Micro propulsion, etcetera. On all of these subsystems, there is one or more micro-controllers doing dedicated tasks. Typically there are peripherals like sensors and actuators or memory which needs to be controlled by pieces of service layer code. This code handles the hardware interface and provides a standardized set of functions to the application layer. The development of DeFFi started in 2014, hence the service layer doesn’t exist yet.

One main part of the service layer was developed in the graduation project: the storage system. OpenUP was employed as the software development method since its suitable for use by a single person. Being a simplified version of RUP, OpenUP provides a suitable middle ground between lightweight and heavyweight methods that focuses on architecture and testing to minimize risks.

At the inception of the project a literature study was conducted to primary answer two questions related to operating electronics in space. The first question is: how suscep-
Consequence

*tile to errors will the storage system be when it's exposed to radiation in space?* We were able to quantify the susceptibility by making an estimation revealing that 2-3 SEUs will occur every second on a 4 GiB flash memory device in DelFFi's orbit.

The second question reads: *how reliable are SD cards in general?* The answer to the question is that SD cards manufacturers take multiple measures to create reliable products, from implementing sophisticated ECC algorithms to using multiple techniques to prolong the life of the flash memory such as wear leveling. However, the storage system on DelFFi's predecessor failed while also making use of an SD card as the storage medium. That was an indication that another approach is required to ensure that the same does not happen again on DelFFi.

Other satellites make use of hardware redundancy to counter errors induced by SEUs. Based on these approaches, it was decided to employ a form of triple modular redundancy (TMR) with hardware whereby 3 SD cards will be used to store replicas of data. Based on an analysis of the need for a software ECC, it was decided to not include it because of the wasted storage space, computational power associated with it while at the same time it can't handle the cases where the internal ECC of SD cards fails.

FAT32 was the file system of choice for the storage system. This is due to simplicity and robustness. A technique called partial data scrubbing was implemented to decrease the accumulation of single-bit bit-flips with time in the two most important parts of the FAT32 file system: the boot sector and the File Allocation Table (FAT). The implementation of the file system was aided by the use of the FatFs library.

A driver for the SD card driver was implemented from scratch because it was observed that most of the freely available open source implementations ignore some steps and errors defined in the SD specification there by rendering them less reliable.

A few conventions were used throughout the code base to overcome some of the issues associated with developing in the C language. The first one unifies the return values of function to the its execution status. The second one employs sudo-namespaces through prefixing to avoid name collisions. All structures, enumerations and functions within the code base were documented using the Doxygen comment blocks. This allows the generation of a documentation suite to help new developers understand the software.

DelFFi's storage system was tested with three types of testing: unit testing, performance testing and integration testing. Unit tests were written for all the packages in the storage system with the help of Google's testing framework, called googletest. The results of performance testing verified that the storage system met a particular non-functional requirement with regards to execution time. In addition, the execution time of other
function was measured. Integration testing ensured that there are no compatibility issues between the different packages. All the test cases of the three types of tests passed.

Two important aspects of the storage system were measured. First, the code size of the software had to be determined to see if it meets requirements. In addition, the power consumption was measured and compared to previous calculations that estimated the consumption for the purpose of finding out whether the calculation were correct. More importantly, the results of the power measurement allows DelFFi’s development team to make accurate predictions of their applications’ power consumption when they store data on the storage system.

The evaluation of the whole project was positive. However, some issues were identified that can be avoided in future projects. All the competences specified in the graduation plan were demonstrated in the work without exception.
References


Federico Faccio. Radiation effects in the electronics for CMS. Tech. rep. CERN.

FatFs - Generic FAT File System Module. URL: http://elm-chan.org/fsw/ff/00index_e.htm.


Appendix: Graduation plan

A.1 Information of graduate and host company

Graduation block: 2014-1.1 (starts before or on 10 February 2014)
Start date for graduation assignment: 10 February 2014
Date of handing in the graduation file according to year schedule: 6 June 2014

Student number: 10102035
Last name: Mr. Sallam
Initials: M.
First name: Maher
Address: Burgemeester van Haarenlaan 1037
Post code: 3118 GA
City: Schiedam
Telephone number: 0639799366
Mobile number: 0639799366
Private email address: maher@sallam.me
Appendix: Graduation plan

**Education:** Computer Engineering (Dutch: Technische Informatica)

**Location:** Delft

**Variant:** Full time

**Name of academic advisor:** Sjaak van Peski

**Name of advising examiner:** Kurt Kohler

**Name of second examiner:** John Visser

**Name of company:** Technische Universiteit Delft

**Department in company:** Faculteit Luchtvaart- en Ruimtevaartechniek

**Visit address of company:** Kluyverweg 1

**Post code of visit address:** 2629 HS

**Mailbox number:** 5058

**Post code of mailbox number:** 2600 GB

**City:** Delft

**Telephone of company:** +31 15 27 82058

**Telefax of company:** +31 5 27 81822

**Internet site of company:** http://www.lr.tudelft.nl

**Last name of employer:** Bouwmeester

**Initials of employer:** J.

**Title of employer:** ir.

**Function of employer:** Delfi Nanosatellite Program Manager & Researcher Small Satellite Technology

**Dial number of employer:**

**Email of employer:** Jasper.Bouwmeester@tudelft.nl

**Last name of company mentor:** Bouwmeester

**Initials of company mentor:** J.

**Title of company mentor:** ir.

**Function of company mentor:** Delfi Nanosatellite Program Manager & Researcher Small Satellite Technology

**Dial number of company mentor:**

**Email of company mentor:** Jasper.Bouwmeester@tudelft.nl

**Dial number of graduate:** n.v.t.

**Function graduate (part time/dual):** Software developer
A.1.1 Title of graduation assignment:

Developing a service layer for the DelFFi space mission at the Delft University of Technology.

A.2 Assignment description

A.2.1 Company

Delft University of Technology, also known as TU Delft, is one of the biggest public technical universities in the Netherlands and Europe. More than 17,000 students and 2,400 scientists study and research, respectively, a plethora of fields of science in its eight faculties and many research institutes. The university was founded in 8 January 1842 by king Willem II and has since acquired multiple names before being called TU Delft.

The facility of aerospace engineering is one of the major faculties inside the TU Delft with four dedicated departments, around 2300 students, 15 full time professors and more than 50 UHD's en UD's. Moreover, this faculty is one of the largest faculties devoted entirely to aerospace engineering in Europe. It is the only institute carrying out research and education directly related to aerospace engineering in the Netherlands.

A.2.2 Problem

The DelFFi mission is the third satellite mission of TU Delft, which is planned for a launch in 2015. It comprises two nanosatellites, called Delta and Phi, which will demonstrate formation flying as part of the QB-50 mission. QB-50 mission is a unique mission establishing an international network of 50 nanosatellites for multi-point, in-situ measurements in the lower thermosphere and re-entry research.

There are many subsystems onboard these two satellites: OnBoard Computer, Electrical Power Subsystem, Radio Transceivers, Attitude Determination and Control, Micro propulsion, QB-50 Sensor Suite Payload, etcetera. On all of these subsystems, there is one or more microcontrollers doing dedicated tasks. Typically there are peripherals like sensors, actuators, ADCs, DACs, internal communication or memory which needs to be controlled by pieces of service layer code. This code handles the hardware interface and provides a standardized set of functions to the application layer. The service layer is yet to be developed.
A.2.3 Goal of the Graduation Assignment

The service layer needs to be designed, implemented and tested for a few microcontrollers that will be used by DelFFi. Two tasks carried out by this layer have already been determined. More could follow once the hardware has been identified in early 2014.

First, the onboard data storage. A Texas Instruments MSP430 microcontroller needs to store data on an SD card. The service layer will allow reading and writing packets of data from and to the SD card, respectively. Error detection and correction code will be implemented which can deal with a 1:2 bitflip ratio. The latter is an extreme case caused by long term radiation in space where electrons are hitting the memory causing malfunctions.

Second, Bluetooth communication. DelFFi will have an onboard experiment which includes a Bluetooth link between a few temperature sensors on the body of the satellite and the onboard computer. The service layer will allow a reliable link in a real satellite situation.

A.2.4 Result

After completion, the service layer will provide a standardized set of functions to the application layer for communication with specific hardware devices. More importantly, the service layer will ensure a reliable operation through well-documented design choices and thorough testing. Documentation will be provided in English.

A.2.5 Undertaken Duties, Including a Global Phasing, Milestones and Associated Activities

- Creating a work plan (Dutch: plan van aanpak), including risks analysis (5 days)
- Researching literature about error detection and correction algorithms, efficient MSP430 programming and datasheets of modules and sensors (12 days)
- Compiling a list of functional and non-functional requirements then afterwards prioritizing them via MoSCoW analysis (4 days)
- Creating a test plan (5 days)
- Design the architecture using different UML diagrams (15 days)
- Implementing the design in low-level programming language C (20 days)
- Testing and verifying the implementation's correctness (6 days)
• Transfer and delivery of finished products (3 days)
• Writing the graduation thesis (15 days)

A.2.6 Delivered (intermediate) products

• Work plan
• Functional and non-functional requirements document
• Test plan
• Test reports
• Design documents (UML diagrams)
• Implementation code

A.2.7 Demonstrated competences and the form of demonstration

1. A5 - Opstellen van systeemeisen (requirements): to be demonstrated by deriving the functional and non-functional requirements. Afterwards, the complied list will be prioritized via MoSCoW analysis.

2. C8 - Ontwerpen van een technisch informatie systeem: to be demonstrated by designing the architecture of the service layer and illustrating that using UML diagrams.

3. C13 – Het betrekken van real-time aspecten bij een ontwerp: to be demonstrated by taking into account the cpu-usage and reliability of communication in the architecture design.

4. D16 - Het realiseren van software: to be demonstrated by implementing the design in a low-level programming language such as C.

5. D17 - Testen van software systemen: to be demonstrated by creating a test plan, executing the tests and finally writing the results in test reports.
Appendix: Graduation plan
Appendix: Work plan

B.1 Introduction

B.1.1 Background

The DelFFi mission is the third satellite mission of TU Delft, which is planned for launch in 2015. It comprises two nanosatellites, called Delta and Phi, which will demonstrate formation flying as part of the QB-50 mission. QB-50 is a unique mission establishing an international network of 50 nano-satellites for multi-point, in-situ measurements in the lower thermosphere and re-entry research.

B.1.2 Project Description

There are many subsystems onboard the two satellites: On-Board Computer, Electrical Power Subsystem, Radio Transceivers, Attitude Determination and Control, Micro propulsion, QB-50 Sensor Suite Payload, etcetera. On all of these subsystems, there are one or more microcontrollers doing dedicated tasks. Typically there are peripherals like sensors, actuators, ADCs, DACs, internal communication or memory which needs to be
controlled by pieces of service layer code. This code handles the hardware interface and provides a standardized set of functions to the application layer. The service layer is yet to be developed.

B.1.3 PROJECT GOALS

The service layer needs to be designed, implemented and tested for a few microcontrollers that will be used by DelFFi. Two tasks carried out by this layer have already been determined. More could follow once the hardware has been identified in early 2014.

First, the on-board data storage. A Texas Instruments MSP430F2418 microcontroller needs to store data on an SD card. The service layer will allow reading and writing packets of data from and to the SD card, respectively. Error detection and correction code will be implemented which can deal with a 1:2 bitflip ratio. The latter is an extreme case caused by long term radiation in space where electrons are hitting the memory causing malfunctions. Here is reliability of utmost importance.

Second, Bluetooth communication. DelFFi will have an on-board experiment which includes a Bluetooth link between a few temperature sensors on the body of the satellite and the onboard computer. The service layer will allow a reliable link in a real satellite situation.

B.1.4 PROJECT RESULT

After completion, the service layer will provide a standardized set of functions to the application layer for communication with specific hardware devices. More importantly, the service layer will ensure a reliable operation through well documented design choices and thorough testing. Documentation will be provided in English.
B.2 Project Organization

In this graduation project spanning 17 weeks, the research and development will be performed in the faculty of aerospace engineering of the Delft University of Technology solely by the graduate under guidance from the company mentor.

The graduate will be responsible for designing the architecture, developing the software and testing the implementation. Important decisions will be made by the graduate after discussing them with the company mentor.

B.2.1 Stakeholders

The following table shows the stakeholders involved in this project.

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>Japser Bouwmeester</td>
<td>Researcher Small Satellite Technology</td>
<td>Company mentor</td>
</tr>
<tr>
<td>Maher Sallam</td>
<td>CE student (Graduate)</td>
<td>Software developer</td>
</tr>
</tbody>
</table>
B.3 APPROACH

After evaluating multiple software development methodologies, it was decided to use OpenUP for the project with a few adaptations to be suitable for use by one person. The next subsections discuss the evaluation of the methods considered and the reason behind choosing OpenUP.

B.3.1 EVALUATION OF DEVELOPMENT METHODOLOGIES

A broad spectra of development methods were considered ranging from heavyweight methodologies like the Rational Unified Process (RUP) to more lightweight ones such as Scrum and eXtreme Programming (XP). Moreover, Test Driven Development, which is a relatively new method, was also considered. Only iterative and incremental methods are considered as it has been proved constantly that they are superior to static methods like waterfall.

Reliability in this project is of utmost importance. The most common and effective way to ensure this non-functional requirement is by taking it into consideration while designing the architecture and by extensive testing afterwards. Therefore, reliability will be an important criterion in the evaluation.

Lightweight methods

All of the lightweight methods adopt the philosophy that the project should be designed and implemented at the same time because of time constraints and because producing a shippable product is their priority. Afterwards, the design is improved by means of refactoring. Although this might produce a robust design eventually, it can’t guarantee reliability of operation in each iteration. Said another way, reliability, and non-functional requirements in general, are an afterthought in lightweight methods.

Heavyweight methods

In contrast, heavyweight methods focus on generating a lot of design and architecture documents and tests to preserve quality. They are most suitable for large projects with many people involved. The danger with using such a method for a one person project is that the documentation required by these methods can be overwhelming and could result in not completing the project within its time limit.
B.3 Approach

Test Driven Development

TDD distinguishes itself with a unique development cycle. While most methods will first design an architecture and then carry out tests to verify the implementation, TDD begins by writing tests for the use cases. Since no implementation has been developed, the tests should fail in the first run. Then code is written to make the tests pass. There is no formal design phase in TDD; the tests implicitly define the architecture of the software. TDD’s approach has the possibility to ensure any non-functional requirement if tests can be written for them. However, for an embedded project like this one, it is quite hard to write tests for some non-functional requirements, and especially for the low-level parts of the implementation. Furthermore, not having formal design documents will make it difficult to evaluate the quality of the architecture.

B.3.2 Rationale behind choosing OpenUP

For this project, a middle ground should be chosen between lightweight and heavyweight methods. A good combination would provide sufficient documentation of the design choices while at the same time not sacrificing quality. It would also allow producing working functionality with each increment. Furthermore, testing should be a key characteristic of the method.

OpenUP can provide this middle ground. It’s a simplified version of RUP that only keeps all of the core principles of RUP, which is as aforementioned a heavyweight method. The architecture-centric approach allows taking reliability in consideration the design process. Furthermore, tests are performed multiple times within each iteration.

The four core principles of OpenUP are listed below.

- Collaborate to align interests and share understanding.
- Balance competing priorities to maximize stakeholder value.
- Focus on the architecture early to minimize risks and organize development.
- Evolve to continuously obtain feedback and improve.

B.3.3 Practices and Measurements

In each iteration a burndown report will be used to track and measure the work being done and to collect metrics for the next iterations.
The architecture will be mostly described by Unified Modeling Language (UML) models and where needed elaborated textually. UML is the canonical language to describe models within the software development industry.

Testing of the low-level code for the MSP430 will be done via TI’s Debug Server Scripting (DSS) which allows runtime inspection of the chip’s status. For the high-level implementation, Google’s googletest library is a suitable choice to perform unit-testing.
B.4 Risks & mitigation

The risks and mitigation measures are described in the risk list document using the OpenUP provided template.
B.5 Planning

A global overview of the planning is outlined in Table B.1. The detailed project plan with the performed tasks can be found in its dedicated document produced with Microsoft Project. The planning only accounts for the first 2 months of the project to be as concrete as possible. As the project progresses, the planning will be extended and adjusted accordingly.

Table B.1: Global outline of the project plan

<table>
<thead>
<tr>
<th>Phase</th>
<th>Iteration</th>
<th>Primary objectives</th>
<th>Start date</th>
<th>End date</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception</td>
<td>1</td>
<td>1. Create project plan</td>
<td>10/02/2014</td>
<td>21/02/2014</td>
<td>2 weeks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2. Create project requirements document</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3. Research MSP430 development, communication with SD card, reliability of SD card in space (redundancy, ECC... etc.)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Elaboration (Storage) | 1 | 1. Prioritize project requirements  
2. Define use cases and scenarios for storage solution  
3. Design architecture of storage solution | 24/02/2014 | 07/03/2014 | 2 weeks |
| --- | --- | --- | --- | --- | --- |
| Construction (Storage) | 1 | 1. Stabilize architecture design  
2. Start implementing the design  
3. Create test plan and test implementation so far | 10/03/2014 | 22/03/2014 | 2 weeks |
| | 2 | 1. Stabilize implementation of storage solution  
2. Perform extensive testing of implementation | 24/03/2014 | 04/04/2014 | 2 weeks |
B.6 Conclusion

In this graduation project spanning 17 weeks, the service layer needs will be designed, implemented and tested for a few microcontrollers that will be used by DelFFi. After completion, the service layer will provide a standardized set of functions to the application layer for communication with specific hardware devices. More importantly, the service layer will ensure a reliable operation through well documented design choices and thorough testing.

OpenUP will be employed as the software development method a few adaptations to be suitable for use by a single person. Being a simplified version of RUP, OpenUP provides a suitable middle ground between lightweight and heavyweight methods that focuses on architecture and testing to minimize risks.

Architecture will be described using UML models. For testing, various methods are chosen to maximize coverage.

A list of risks has been compiled and prioritized to tackle them in order of importance based on magnitude. Moreover, mitigation strategies have been described.

The planning has been outlined including the primary objectives per phase. The detailed version contains the details of the performed tasks.
C.1 Use cases

A user will expect certain functionality to be provided by a storage system. For DelFFi, fig. C.1 shows the use cases for the application programming interface (API) of that system. The uses cases are grouped into three categories: a) file operations; b) folder management; and c) system statistics.

Sharp readers might have seen that the use cases imply that the measurements will be stored in files that can be grouped in folders as is typical in file systems. That was chosen because this is the canonical way file systems store their data in and there is no need to reinvent the wheel by developing our own structure for storing the measurements. Moreover, using familiar concepts such as files and folders to describes the uses cases make them accessible to more stakeholders.

The first two categories are fairly obvious and don’t requires any elaboration, especially because any user with basic computer knowledge should be familiar with them. System statistics most notably allow acquiring the amount of single event upsets (SEUs) per SD card and the total SEUs as well. Although normal users of a storage system don’t
pay attention to reliability measures, researchers and engineers working on DelFFi will most definitely require such functionality from the system. These metrics are valuable for evaluating the effectiveness of the implemented fault-tolerance. Moreover, future research can be conducted while making use of the data to improve next iterations of the system. Hence, system statistics are included as a part of the use cases.
C.2 Functional requirements

There is an important relationship between use cases and functional requirements: the former is a subset of the latter. Use cases therefore already define a part of the functional requirements, but only those which a user will interact with. These are called behavioral requirements. The rest of the functional requirements describe what the system does internally.

MoSCoW was used to prioritize the functional requirements \[8\]. There are four priorities defined in the MoSCoW method:

- **M** (ust) have: What must be delivered?
- **S** (ould) have: What should be delivered as a high priority but not essential?
- **C** (ould) have: What could be delivered if there was available time / budget / resource?
- **W** (ould) have: What would be delivered all other requirements have been finished?

The prioritization was done based on feedback from Japser Bouwmeester on the first revisions of the requirements document. For example, it is essential that the storage system allows storing and reading of data in files, but organizing the data into folders is of less importance. Therefore, operations relating to storing and reading data in files will be a **M**ust have.

The functional requirements for the API of the storage system are listed below. Next to each requirement a letter is used to indicate its priority.

**File operations**

1. The storage system will allow reading data from a file. (**M**)
2. The storage system will allow reading data from a file on an individual SD card. (**S**)
3. The storage system will allow writing data to a file. (**M**)
4. The storage system will allow appending data to a file. (**S**)
5. The storage system will allow creating files. (**M**)
6. The storage system will allow removing files. (S)

7. The storage system will allow checking if a file exists. (S)

8. The storage system will allow checking if a file exists on an individual SD card. (C)

9. The storage system will allow reading file attributes. The returned attributes must include: a) file size; b) file visibility (hidden or not); c) read mode (read-only or not); and d) last modification date and time. (W)

10. The storage system will allow reading attributes of a file on an individual SD card. The returned attributes will be the same as the ones returned for the normal operation. (W)

11. The storage system will allow modifying file attributes. The modifiable set of attributes must include: a) file visibility (hidden or not); and b) read mode (read-only or not). (W)

12. The storage system will allow renaming files. (C)

13. The storage system will allow moving a file into a folder. (W)

**Folder management**

1. The storage system will allow creating folders. (C)

2. The storage system will allow removing folders. (C)

3. The storage system will allow renaming folders. (C)

4. The storage system will allow checking if a folder exists. (C)

5. The storage system will allow checking if a folder exists on an individual SD card. (W)

6. The storage system will allow moving a folder into another folder. (C)
C.2 Functional requirements

**System statistics**

1. The storage system will allow reading the total size of the storage medium. \( M \)
2. The storage system will allow reading the total size of an individual SD card. \( M \)
3. The storage system will allow reading the free space left on the storage medium. \( S \)
4. The storage system will allow reading the SEUs per SD card. \( M \)
5. The storage system will allow reading the total SEUs. \( C \)

**Internal operations**

1. The storage system will allow formatting the SD cards into a usable state. \( M \)
2. The storage system will allow reading the ECC of a block. \( C \)
Appendix: Requirements document

C.3 Non-functional requirements

For grouping the non-functional requirements, the classification provided by the quality model in the ISO/IEC 9126 standard is followed [31]. The non-functional requirements are:

**Functionality**

**Accuracy**

1. The storage system shall record last modification time of files to second-precision. The precision is restricted since the measurements data will include the actual time of performing the measurement. The time is taken from the RTC module integrated into the OBC. Therefore, files don’t have to record the time with high accuracy.

2. The storage system shall record a maximum of $2^{32} - 1$ encountered SEUs (i.e., a 32-bit number). This is a consequence of the fact that the micro-controller of the OBC doesn’t have 64-bit numbers.

**Functionality Compliance**

3. The storage system shall adhere to the SD specification version 4.10 for communicating with the SD cards.

4. The storage system shall adhere to the FAT32 specification for designing and implementing the file system.

5. The storage system shall support only the 8.3 file naming convention in which the filename has a maximum of 8 letters. Although this will restrict the file and folder names to be 8 characters long, it simplifies the implementation immensely since the FAT32 specification doesn’t support long file names by default.

**Reliability**

**Fault tolerance**

6. The storage system shall detect a maximum of 2 bit-flips in a block.

7. The storage system shall correct a maximum of 1 bit-flip in a block.
Usability

Understandability

8. The storage system shall have an API that imitates the file functions provided in the `stdio.h`. Since most programmers are accustomed to the C-API of working with files, the API in DelFFi will follow the same conventions to make it accessible and easy to use. For example, to store data into a file, the file must be opened, data can then be written to the file and finally the file has to be closed.

Efficiency

Time behaviour

9. The storage system shall handle writing 2 kbit of data in under 200 ms.

Resource behaviour

10. The storage system shall consume a maximum of 22 kB from the flash memory on board the MSP430F2418 micro-controller for its code. This ensures that the file system doesn’t use all the space on the OBC and that enough space is left for the application code.

Maintainability

Testability

11. The storage system shall provide unit tests for all modular components of the system.
C.4 USE CASE SCENARIOS

An elaboration of the use cases is given in this section. The scenarios describe the steps performed by an actor to complete a use case, either successfully or via an alternative flow in case of errors.

C.4.1 WRITING OR APPENDING DATA TO FILE

Brief description
This use case describes how a user can write or append data to a file.

Actors
1. User

Preconditions
1. The storage system is on and operational.
2. The data to be written exists in a buffer.

Main flow of steps
1. The user acquires a file handle with the appropriate parameters.
2. The write function is called. The file handle is supplied together with the buffer to the write function.
3. All SD cards write the data from the buffer to the file.
4. The amount of bytes written is returned. In this case it’ll equal the buffer size.
5. The user releases the file handle.
6. The use case ends successfully.

Alternative flow
1. File handle can’t be acquired
   If in step 1 an error occurs while acquiring the file handle, for example because the file is read-only, then
   (a) A distinctive error code is returned.
(b) The use case ends with a failure condition.

2. **There isn’t enough free space on the SD cards**
   If in step 2 the check for free space fails (refer to special requirement 1), then
   
   (a) A distinctive error code is returned.
   
   (b) The use case ends with a failure condition.

3. **The file attributes can’t be updated**
   If in step 2 an error occurs while updating the file attributes, for example the new size, then
   
   (a) A distinctive error code is returned.
   
   (b) The use case ends with a failure condition.

4. **An acquired file handle can’t be released**
   If in step 3 an error occurs while releasing the file handle, for example because of an SD card timeout, then
   
   (a) A distinctive error code is returned.
   
   (b) The use case ends with a failure condition.

**Special requirements**

1. The free space on all SD cards is always the same. Therefore, the buffer size must not exceed the free space on any SD card.

**Post-conditions**

1. **Successful completion**
   
   (a) The data is written the file.
   
   (b) The file size and modification time are updated accordingly.
   
   (c) The ECC for the blocks occupied by the file are calculated and stored.
   
   (d) The data buffer is unchanged.

2. **Failure condition**
   
   (a) The data buffer is unchanged.
C.4.2 READING DATA FROM FILE

Brief description
This use case describes how a user can read data from a file.

Actors
1. User

Preconditions
1. The storage system is on and operational.
2. The file to read data from exists.
3. A large enough buffer is allocated for the data.
4. The amount of bytes to be read is equal or less than the file size.

Main flow of steps
1. The user acquires a file handle with the appropriate parameters.
2. The read function is called. The file handle, the amount of bytes to read and the buffer are supplied to the read function.
3. Data is read from all SD cards and stored in the buffer.
4. The amount of bytes written is returned. In this case it’ll equal the amount of bytes given.
5. The user releases the file handle.
6. The use case ends successfully.

Alternative flow
1. File handle can’t be acquired
   If in step 1 an error occurs while acquiring the file handle, for example because of an SD card timeout, then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.
2. **ECC fail**
   If in step 3 an uncorrectable error is detected by ECC, for example a multi-bit error, then
   
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.
   
   In this situation, a user can try to obtain the data from an individual SD card if necessary.

3. **An acquired file handle can’t be released**
   If in step 3 an error occurs while releasing the file handle, for example because of an SD card timeout, then
   
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.

**Post-conditions**

1. **Successful completion**
   
   (a) The data is stored in the buffer.
   (b) Detected single-bit errors while reading are corrected. In addition, the erroneous blocks will be updated with the corrected bits as well.

2. **Failure condition**
   
   (a) The data buffer is partially modified.
C.4.3 READING FILE ATTRIBUTES

Brief description
This use case describes how a user can read the attributes of a file.

Actors
1. User

Preconditions
1. The storage system is on and operational.
2. The file to read the attributes of exists.
3. A large enough buffer is allocated for the attributes.

Main flow of steps
1. The read file attributes function is called with the file path and the buffer as parameters.
2. The file attributes are read and stored in the buffer.
3. Success return value is returned.
4. The use case ends successfully.

Alternative flow
1. ECC fail
   If in step 2 an uncorrectable error is detected by ECC, for example a multi-bit error, then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.

In this situation, a user can try to obtain the data from an individual SD card if necessary.
Post-conditions

1. **Successful completion**
   
   (a) The file attributes is stored in the buffer.

   (b) Detected single-bit errors while reading are corrected. In addition, the erroneous blocks will be updated with the corrected bits as well.

2. **Failure condition**

   (a) The file attributes buffer is partially modified.
C.4.4  MODIFYING FILE ATTRIBUTES

Brief description
This use case describes how a user can modify the attributes of a file.

Actors
1. User

Preconditions
1. The storage system is on and operational.
2. The file to modify the attributes of exists.

Main flow of steps
1. The modify file attributes function is called with the file path and the new attributes as parameters.
2. The file attributes are modified on all SD cards.
3. Success return value is returned.
4. The use case ends successfully.

Alternative flow
1. Modification fail
   If in step 2 an uncorrectable error is detected by ECC, for example a multi-bit error, then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.

In this situation, a user can try to obtain the data from an individual SD card if necessary.
Post-conditions

1. **Successful completion**
   
   (a) The file attributes are modified on all SD cards.
   
   (b) Detected single-bit errors while reading are corrected. In addition, the erroneous blocks will be updated with the corrected bits as well.

2. **Failure condition**

   (a) The file attributes are partially modified.
C.4.5 CHECKING IF A FILE EXISTS

Brief description
This use case describes how a user can check if a file exists.

Actors
1. User

Preconditions
1. The storage system is on and operational.

Main flow of steps
1. The check existence function is called with the file path as a parameter.
2. A value is returned indicating whether the file exists.
3. The use case ends successfully.

Alternative flow
1. ECC fail
   If in step 2 an uncorrectable error is detected by ECC, for example a multi-bit error, then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.

   In this situation, a user can try to check the existence of a file on an individual SD card if necessary.

Post-conditions
1. Successful completion
   (a) Detected single-bit errors while existence are corrected. In addition, the erroneous blocks will be updated with the corrected bits as well.

2. Failure condition
   (a) The existence of a file is unknown.
C.4.6 Removing a file

Brief description
This use case describes how a user can remove a file.

Actors
1. User

Preconditions
1. The storage system is on and operational.
2. The file to remove exists.

Main flow of steps
1. The remove file function is called with the file path as a parameter.
2. A value is returned indicating the successful removal of file.
3. The use case ends successfully.

Alternative flow
1. The file can’t be removed
   If in step 1 an error occurs while removing the file, for example because the file is read-only, then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.

Post-conditions
1. Successful completion
   (a) The file is removed from all SD cards.
2. Failure condition
   (a) The file is in an unknown state.
C.4.7 Moving a file into a folder

Brief description
This use case describes how a user can move a file into a folder.

Actors
1. User

Preconditions
1. The storage system is on and operational.
2. The to be moved file exists.
3. The container folder exists.

Main flow of steps
1. The move function is called with the file path and the container folder path as parameters.
2. A value is returned indicating the successful move of the file.
3. The use case ends successfully.

Alternative flow
1. The file already exists
   If a file already exists with the same name as the to be moved one, then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.

2. ECC fail
   If in step 1 an uncorrectable error is detected by ECC, for example a multi-bit error, then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.
Post-conditions

1. Successful completion
   
   (a) Detected single-bit errors while checking existence are corrected. In addition, the erroneous blocks will be updated with the corrected bits as well.
   
   (b) The file has been moved.

2. Failure condition
   
   (a) The file has not been moved.
C.4.8  RENAMING A FILE

**Brief description**

This use case describes how a user can change the name of a file.

**Actors**

1. User

**Preconditions**

1. The storage system is on and operational.
2. The to be renamed file exists.

**Main flow of steps**

1. The rename function is called with the old file path and the new file name as parameters.
2. A value is returned indicating the successful rename of the file.
3. The use case ends successfully.

**Alternative flow**

1. **The file already exists**
   If a file with the same chosen new name already exists, then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.

   2. **ECC fail**
      If in step 1 an uncorrectable error is detected by ECC, for example a multi-bit error, then
      (a) A distinctive error code is returned.
      (b) The use case ends with a failure condition.
Post-conditions

1. Successful completion
   
   (a) Detected single-bit errors while checking existence are corrected. In addition, the erroneous blocks will be updated with the corrected bits as well.
   
   (b) The file has been renamed.

2. Failure condition
   
   (a) The file has not been renamed.
C.4.9 CREATING A FOLDER

Brief description
This use case describes how a user can create a folder.

Actors
1. User

Preconditions
1. The storage system is on and operational.

Main flow of steps
1. The create folder function is called with the new folder path as a parameter.
2. A value is returned indicating the successful creating of the folder.
3. The use case ends successfully.

Alternative flow
1. The folder already exists
   If the folder already exists, then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.
2. Parent folder is write protected
   If the parent folder is write protected, then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.

Post-conditions
1. Successful completion
   (a) The folder is created on all SD cards.
2. Failure condition
   (a) The folder is not created.
C.4.10 REMOVING A FOLDER

Brief description
This use case describes how a user can remove a folder.

Actors
1. User

Preconditions
1. The storage system is on and operational.
2. The folder to remove exists.
3. The folder is empty. The user must remove all files and sub-folders inside the folder before calling this function.

Main flow of steps
1. The remove folder function is called with the folder path as a parameter.
2. A value is returned indicating the successful removal of the folder.
3. The use case ends successfully.

Alternative flow
1. The folder can’t be removed
   If in step 1 an error occurs while removing the folder, for example because the folder is locked, then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.

Post-conditions
1. Successful completion
   (a) The folder is removed from all SD cards.

2. Failure condition
   (a) The folder is not removed.
C.4.11 CHECKING IF A FOLDER EXISTS

Brief description
This use case describes how a user can check if a folder exists.

Actors
1. User

Preconditions
1. The storage system is on and operational.

Main flow of steps
1. The check existence function is called with the folder path as a parameter.
2. A value is returned indicating whether the file exists.
3. The use case ends successfully.

Alternative flow
1. ECC fail
   If in step 2 an uncorrectable error is detected by ECC, for example a multi-bit error, then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.

   In this situation, a user can try to check the existence of a folder on an individual SD card if necessary.

Post-conditions
1. Successful completion
   (a) Detected single-bit errors while checking existence are corrected. In addition, the erroneous blocks will be updated with the corrected bits as well.
   (b) The existence of a folder is determined.

2. Failure condition
   (a) The existence of a folder is unknown.
C.4.12 Moving a folder into another folder

**Brief description**

This use case describes how a user can move a folder into another folder.

**Actors**

1. User

**Preconditions**

1. The storage system is on and operational.
2. The to be moved folder exists.
3. The container folder exists.

**Main flow of steps**

1. The move function is called with the old folder path and container folder path as parameters.
2. A value is returned indicating the successful move of the folder.
3. The use case ends successfully.

**Alternative flow**

1. The folder already exists
   
   If a folder already exists with the same name as the to be moved one, then
   
   (a) A distinctive error code is returned.
   
   (b) The use case ends with a failure condition.

2. ECC fail
   
   If in step 1 an uncorrectable error is detected by ECC, for example a multi-bit error, then
   
   (a) A distinctive error code is returned.
   
   (b) The use case ends with a failure condition.
Post-conditions

1. **Successful completion**

   (a) Detected single-bit errors while checking existence are corrected. In addition, the erroneous blocks will be updated with the corrected bits as well.

   (b) The folder has been moved.

2. **Failure condition**

   (a) The folder has not been moved.
C.4.13 RENAMING A FOLDER

Brief description
This use case describes how a user can change the name of a folder.

Actors
1. User

Preconditions
1. The storage system is on and operational.
2. The to be renamed folder exists.

Main flow of steps
1. The rename function is called with the old folder path and the new folder name as parameters.
2. A value is returned indicating the successful rename of the folder.
3. The use case ends successfully.

Alternative flow
1. The folder already exists
   If a folder with the same chosen new name already exists, then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.

2. ECC fail
   If in step 1 an uncorrectable error is detected by ECC, for example a multi-bit error, then
   (a) A distinctive error code is returned.
   (b) The use case ends with a failure condition.
Appendix: Requirements document

Post-conditions

1. **Successful completion**
   
   (a) Detected single-bit errors while checking existence are corrected. In addition, the erroneous blocks will be updated with the corrected bits as well.
   
   (b) The folder has been renamed.

2. **Failure condition**

   (a) The folder has not been renamed.
C.4.14 READING STORAGE SYSTEM STATISTICS

Brief description
This use case describes how a user can read the statistics of the storage system.

Actors
1. User

Preconditions
1. The storage system is on and operational.
2. A large enough buffer is allocated for the statistics.

Main flow of steps
1. The read storage statistics function is called with the buffer as a parameter.
2. The storage statistics are read and stored in the buffer.
3. Success return value is returned.
4. The use case ends successfully.

Alternative flow
1. ECC fail
   If in step 2 an uncorrectable error is detected by ECC, for example a multi-bit error, then
      (a) A distinctive error code is returned.
      (b) The use case ends with a failure condition.

In this situation, a user can try to obtain the data from an individual SD card if necessary.

Post-conditions
1. Successful completion
   (a) The storage statistics are stored in the buffer.
(b) Detected single-bit errors while reading are corrected. In addition, the erroneous blocks will be updated with the corrected bits as well.

2. **Failure condition**

   (a) The storage statistics buffer is partially modified.
C.4.15 Reading error statistics

Brief description
This use case describes how a user can read the collected error statistics of the storage system.

Actors
1. User

Preconditions
1. The storage system is on and operational.
2. A large enough buffer is allocated for the statistics.

Main flow of steps
1. The read error statistics function is called with the buffer as a parameter.
2. The error statistics are read and stored in the buffer.
3. Success return value is returned.
4. The use case ends successfully.

Post-conditions
1. Successful completion
   (a) The error statistics are stored in the buffer.
Appendix: Design UML diagrams

All UML diagrams created for the storage system are included in this chapter. For more information about the system requirements, refer to ?? . The architecture and design are described in detail in chapter 5.
D.1 Use cases diagram

Figure D.1: Complete use cases diagram
D.2 Packages diagram

Figure D.2: Packages diagram
D.3 CLASS DIAGRAMS

D.3.1 UTILITY PACKAGE

Figure D.3: Utility class diagram
D.3.2 SD CARD DRIVER PACKAGE

Figure D.4: SD card driver class diagram
D.3.3 TMR PACKAGE

Figure D.5: TMR class diagram
D.3.4 File system package

Figure D.6: File system class diagram
D.4 Activity diagrams

D.4.1 Reading a block

Figure D.7: Reading a block activity diagram
D.4.2 Writing a block

Figure D.8: Writing a block activity diagram
Appendix: Design UML diagrams
Appendix: Test plan

E.1 Introduction

This document describes the types of tests that will be performed on DelFFi’s storage system: unit testing, performance testing and integration testing.

E.2 Testing strategy

Three types of tests are chosen to be performed on DelFFi’s storage system: unit testing, performance testing and integration testing.

Unit testing is used to test the code on a high level and to make sure that an interface does what a user expects from it. Google’s googletest library will be used to write the unit tests since it provides a robust framework that’s supported by multiple platforms.

To make sure that the system meets the speed requirements, performance testing is performed on the SD card driver and the file system functions. The hardware timer of the MSP430 controller will be used to measure the execution time.

Integration testing ensures that there are no compatibility issues between the different
internal modules. In contrast to unit testing that uses stubs to allow testing of individual modules, these tests are run on the hardware with the SD cards.

E.3 Unit Testing

E.3.1 Tested Items

All the modules of the storage system will be unit tested, including the utility modules such as the CRC and timer module.

E.3.2 Approach

For each condition that changes the result of a function a test case will be written. This helps in maximizing the test coverage. The different macros provided by googletest provides for asserting expectations will be used inside the test cases to write the tests.

E.3.3 Passing Criteria

The system is considered to have a correctly implemented API if all the unit tests pass without exception.

E.4 Performance Testing

E.4.1 Tested Items

Three parts of the software have been empirically identified to be a potential speed bottleneck:

1. Calculation of CRC
2. Reading and writing blocks to the SD card
3. Reading and writing files to the file system with TMR

The last item includes the execution time of the second one. For example, to read a file from the storage system, blocks will be read from the 3 SD cards. However, reading a file and reading a block from the SD card are two tasks of two different modules: the file system and the SD card driver, respectively. Therefore, it was chosen to test both of them to have performance tests for both modules.
E.4.2 **APPROACH**

Before running a tested item, the timer module on the MSP430 controller is used to store the current time counter in a variable. Then, the tested item is run. After completion, the difference in time between the current time counter and the stored one is computed.

There are two methods to get the computed execution time. The first one is to use the debugger and to put a breakpoint after computing the time. This approach however required compiling the code which will disable optimization. As a result, the execution time will not be accurate.

Another way is to store the time in a file after the test finishes and then reading the file on a PC from the SD card. This method allows compiling the code in Release mode with optimizations which will result in actual execution time. Therefore, this method was chosen.

E.4.3 **PASSING CRITERIA**

All execution times of the tested items must be less than 200ms as was defined in the non-functional requirements.

E.5 **SYSTEM AND INTEGRATION TESTING**

E.5.1 **Tested items**

The architecture of the storage system defines a hierarchy of layers that begin with the file system API at the top, going through TMR, and ending at the bottom with the 3 SD cards access by the SD card driver.

To make sure that the right parameters are being passed from one layer to the other and that they are correctly integrated, one must test functions that go through the whole hierarchy. Since unit testing already cover all aspects of the interface, such as different error conditions and valid parameters, only the operations that involve multiple layers should be tested.

For the previous reasons, the following set of tests have been devised that test the storage system:

1. When reading a file that is identical on all SD cards, the file is read from all the SD cards no and error is reported.
2. If a block corresponding to a file is corrupted on only one SD card, the system must recover from that error, the correct block must be written to the mismatched SD card and the statistics must be updated correctly.

3. When writing a file, the data must be written to all SD cards.

E.5.2 APPROACH

For the lack of a better mechanism to perform these tests, a slightly complicated method is used. However, since there are only 3 tests, this shouldn’t be an issue.

First, all the SD cards are formatted on a PC and the content of the cards is set up appropriately for the tests. For example, for the second test one must insure that a block on one of the SD cards differs from the other two cards.

Then, the cards are inserted into the storage system and a piece of test code is run for a particular test. For the second test item that code will read the file from the SD cards.

While running the code on the storage system, an SPI sniffer will be used to monitor the activity of the SPI bus and to ensure that data is being exchanged with the SD cards. For the first tested item, that means monitoring the exchange of data with the 3 SD cards.

Finally, the cards are inserted back into the PC to check the results of the test.

E.5.3 PASSING CRITERIA

All the tested items must pass for the integration to be considered successful.
Appendix: Test procedure & results

F.1 Introduction

This document describes the procedure of executing the Test Plan and the results of the tests.

F.2 Unit Testing

F.2.1 Procedure

Google's testing framework, called googletest, comes with template project files for different IDE's to help set up an environment to write and run tests. The template project for Visual Studio was used to write the unit tests since the development of the storage system took place on a Windows machine.

All of the tests for the utility modules were straightforward assertions of expectations from functions. However, the same can’t be said when it came to testing the SD card driver and the file system. Both of these modules interact with other modules and expects them to provide services for them. The SD card driver requires the SPI module to
send the issued command and expects to get responses back. Moreover, the SD card driver needs the responses to comply with the SD specification to function properly and hence this will require having the actual hardware (i.e., the SD card) be part of the testing process. The file system bases its services based on the TMR modules which in turn makes use of the SD card driver. This means that neither the card driver nor the file system could be tested in isolation which is the goal of a unit test as the name implies.

As customary in unit testing, stubs were employed to resolve that issue and to allow isolating the testing of the modules. For the card driver, a stubbed version of the SPI interface was implemented using C++ containers to simulate sending and receiving data. Listing F.1 shows how the implementation was accomplished. Notice the two containers sent and received that are defined in the code.

Listing F.1: Stubbed implementation of the SPI module for testing the card driver

```cpp
// Stub out the SPI protocol implementation
static delffi_sd_card_t card;
static std::vector<uint8_t> sent;
static std::queue<uint8_t> received;

void delffi_spi_send(uint8_t byte) {
    // Only add byte to container if the card is selected
    if (!((card.card_select_port) & card.card_select_pin))
        sent.push_back(byte);
}

uint8_t delffi_spi_receive() {
    uint8_t result = received.front();
    received.pop();
    return result;
}
```

Listing F.2 displays how both containers are used in the tests. In each test case, the received container is first filled with the data that the driver expects to receive from the card for it to function correctly. Then, expectation statements from the testing framework are used to test whether the driver sends the correct commands by inspecting the sent container. The sent_bytes(n) method returns the last n bytes from the sent container. Variables in the code beginning with _cmd are arrays of bytes that contain the right sequence of bytes for a given command. For example, here is one of those commands: 

```
_cmd0 = vector<uint8_t>({ 0x40, 0x00, 0x00, 0x00, 0x00, 0x00, 0x95, 0xFF });
```
Listing F.2: One test case of the SD card driver

```c
TEST_F(SdCardTest, HandlesInitializeWithEnablingCrc) {
  delffi_sd_result_t result;

  received << 0x01; // response of CMD0
  received << 0x01 << 0x00 << 0x00 << 0x01 << 0xAA; // response of CMD8
  received << 0x01 << 0x80 << 0xFF << 0x80 << 0x00; // response of CMD58
  received << 0x00; // *incorrect* response of CMD59

  result = delffi_sd_init(&card);

  EXPECT_EQ(sent_bytes(7), _cmd0);
  EXPECT_EQ(sent_bytes(7), _cmd8);
  EXPECT_EQ(sent_bytes(7), _cmd58);
  EXPECT_EQ(sent_bytes(7), _cmd59);

  EXPECT_EQ(result, DELFFI_SD_RESULT_ERROR_ENABLE_CRC);
}
```

For testing the file system, a similar approach was used by providing a virtual FAT32 drive using a temporary file formatted with our VirtualDrive class developed as part of the formatter program. Before each test, the temporary file will be created, the test would be run and afterwards the file gets deleted. This ensures that tests don’t interfere with each other. Again, all for the sake of tests isolation.

After all the tests were written, Visual Studio compiles them and generates one executable that can be run to see the results.

### F.2.2 Results

The following table shows the amount of unit tests that were developed and their results. To see results of individual tests, run the generated executable inside the `unit-tests` directory of the project.
F.3 PERFORMANCE TESTING

F.3.1 PROCEDURE

A file called performance-test.c was created in the storage system project. This file contains a function per test case that runs an operation while calculating the difference in time between the time before and after the operation call. Afterwards, the execution time is written to a file on the storage system. The file name identifies the test case it corresponds to; for example, the file crc7.txt corresponds to the test case of calculating CRC7 for a packet.

The code was compiled with maximum optimization enabled. Each test was run 5 times. The average execution time is reported with the deviation next to it.

F.3.2 RESULTS

The following tables show the execution time of the tested items defines in the Test Plan. Results indicating an execution time of 0 ms means that the operation was done in less than a millisecond. A packet contains 6 bytes of data while a block 512 bytes.

**CRC computation**

**SD card driver**

**File system**

As can be seen, the most expensive operation is creating a file with some data. However, once the file has been created, both writing and reading is relatively fast.

### Appendix: Test procedure & results

<table>
<thead>
<tr>
<th>Module</th>
<th>Tests count</th>
<th>Tests result</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD card driver</td>
<td>39</td>
<td>All pass</td>
</tr>
<tr>
<td>File system</td>
<td>65</td>
<td>All pass</td>
</tr>
<tr>
<td>Statistics</td>
<td>6</td>
<td>All pass</td>
</tr>
<tr>
<td>CRC (LUT)</td>
<td>6</td>
<td>All pass</td>
</tr>
<tr>
<td>CRC (realtime)</td>
<td>6</td>
<td>All pass</td>
</tr>
<tr>
<td>TMR</td>
<td>10</td>
<td>All pass</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>132</strong></td>
<td><strong>All pass</strong></td>
</tr>
</tbody>
</table>

---

F.3 PERFORMANCE TESTING

F.3.1 PROCEDURE

A file called performance-test.c was created in the storage system project. This file contains a function per test case that runs an operation while calculating the difference in time between the time before and after the operation call. Afterwards, the execution time is written to a file on the storage system. The file name identifies the test case it corresponds to; for example, the file crc7.txt corresponds to the test case of calculating CRC7 for a packet.

The code was compiled with maximum optimization enabled. Each test was run 5 times. The average execution time is reported with the deviation next to it.

F.3.2 RESULTS

The following tables show the execution time of the tested items defined in the Test Plan.

Results indicating an execution time of 0ms means that the operation was done in less than a millisecond. A packet contains 6 bytes of data while a block 512 bytes.

**CRC computation**

**SD card driver**

**File system**

As can be seen, the most expensive operation is creating a file with some data. However, once the file has been created, both writing and reading is relatively fast.
### Table F.1: Execution time of CRC computation

<table>
<thead>
<tr>
<th>Test case</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC7 (computed in real time) of a packet</td>
<td>1ms ±1ms</td>
</tr>
<tr>
<td>CRC16 (computed in real time) of a block</td>
<td>9ms ±1ms</td>
</tr>
</tbody>
</table>

### Table F.2: Execution time of reading and writing a block to one SD card

<table>
<thead>
<tr>
<th>Test case</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reading a block from card</td>
<td>13ms ±2ms</td>
</tr>
<tr>
<td>Writing a block to card</td>
<td>13ms ±2ms</td>
</tr>
</tbody>
</table>

Since the file system caches the file attributes and only writes them back to the drive once the file is closed, the results of writing and appending are included with and without the overhead of closing. The cost of closing a file is that of writing an extra block, which can be conformed from the results and is to be expected since the file attributes are updated using one block of data.

Another thing of note is that the overhead of TMR code that compares the blocks is so low that it does not affect the execution time. Writing a block to a file takes exactly three times as much as writing a block to one SD card. Therefore, virtually all of the time is spent writing that one block to the three SD as if there was no code that compares the blocks and looks for mismatches (i.e., TMR voter).

## F.4 System and Integration Testing

### F.4.1 Procedure

For the lack of a better mechanism to perform these tests, a slightly complicated method is used. However, since there are only 3 tests, this wasn’t an issue.

First, all the SD cards are inserted sequentially to the PC. Each one gets formatted and the content of the cards is set up appropriately for a test. For example, for the second test item one must ensure that a block on one of the SD cards differs from the other two cards. Listing F.3 shows how the integration test does that based on the index of the inserted card. The _setup_test function is a private method that formats the SD card before running the lambda expression passed as a parameter, which is a feature in the
Table F.3: Execution time of a selected function from the file system

<table>
<thead>
<tr>
<th>Test case</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating an empty file</td>
<td>82ms ±2ms</td>
</tr>
<tr>
<td>Creating a file with a block of data</td>
<td>362ms ±5ms</td>
</tr>
<tr>
<td>Reading a block from a file</td>
<td>43ms ±2ms</td>
</tr>
<tr>
<td>Writing a block to a file (without closing)</td>
<td>39ms ±2ms</td>
</tr>
<tr>
<td>Writing a block to a file (with closing)</td>
<td>78ms ±2ms</td>
</tr>
<tr>
<td>Appending a block to a file (without closing)</td>
<td>39ms ±2ms</td>
</tr>
<tr>
<td>Appending a block to a file (with closing)</td>
<td>78ms ±2ms</td>
</tr>
</tbody>
</table>

new C++11 standard.

Listing F.3: First step of correcting blocks integration test on PC

```cpp
void IntegrationTests::correcting_block_test()
{
    _setup_test("correcting blocks", [this](unsigned int card_i) {
        FIL file;
        FRESULT result = f_open(&file, _T("file.txt"),
            FA_WRITE | FA_CREATE_ALWAYS);

        if (result != FR_OK) {
            stringstream ss;
            ss << "Couldn't create test file. Result: " << result << ".";
            throw runtime_error(ss.str());
        }

        // Change the content of the first card
        if (card_i == 0)
            f_puts(_T("Holla"), &file);
        else
            f_puts(_T("Hello"), &file);

        f_close(&file);
    });
}
```

Then, the cards are inserted into the storage system and a piece of test code is run for
a particular test. Continuing with our example, that code reads the file from the 3 SD cards.

While running the code on the storage system, an SPI sniffer will be used to monitor the activity of the SPI bus and to ensure that data is being exchanged with the SD cards. For the first tested item, that means monitoring the exchange of data with the 3 SD cards.

Finally, the cards are inserted back into the PC to check the results of the test.

F.4.2 RESULTS

All 3 tests passed confirming the correctness of the integration process.

In the first test the file was read from all SD cards. This was observed by running the test 3 times and each time we used the SPI analyser to capture the communication with a different SD card.

In the second test it was observed that only the first card (i.e., the card with index zero) received a write command with the corrected text, which was "Hello" in the tests as can be seen in listing F.3.

The third test was confirmed by reading all SD card on a PC and it was observed that all of them contained the correct data.