NTNU – Trondheim
Norwegian University of
Science and Technology

# Improvement in the Reliability of a Bi-Processing Unit Satellite Subject to Radiation-Induced Bit-Flips

Mayeul Marcadella

Embedded Computing Systems
Submission date: June 2014
Supervisor: Bjørn B. Larsen, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

# Problem Description

## Improvement in the Reliability of a Bi-Processing Unit Satellite Subject to Radiation-Induced Bit-Flips

This work is part of the NUTS project which aims at designing, building, testing and launching a double CubeSat by 2015. Currently most of the hardware has already been designed, while a lot of effort remains to be done on the software side.

Designing a spacecraft is a very challenging task because of the constraints imposed by the outer space environment such as space radiations which affect the reliability of electronic components. Some research on how to leverage the hardware redundancy of NUTS needs to be done in order to improve the lifetime of the system.

**The task in this assignment is to research and design, given the existing hardware architecture of NUTS, software mechanisms that would increase the tolerance of the satellite to space radiation effects such as bit-flips in the memory devices. One possible axis of research could be the cooperation between the two processing units embedded in the satellite. The student should test his ideas on an experimental platform and propose an implementation for NUTS if time permits.**

Supervisor: Assoc. Prof. Bjørn B. Larsen, IET

# Abstract

The design of reliable spacecrafts is a challenging task because of the harsh constraints imposed by the outer space environment. One major cause of failure of part or totality of the system lies in the space radiations which affect the embedded electronic components, such as the introduction of bit-flips in the memory devices.

The work accomplished in this thesis attempts to improve the reliability of NUTS (NTNU - Test Satellite). The focus has been set on the research of software techniques leveraging the hardware architecture available in order to achieve tolerance against radiation-induced bit-flips.

A study of the effect of bit-flips in both data and program memory has led to the establishment of a stack of techniques aiming at increasing the reliability of the system in a radiated environment. These techniques consist of the use of watchdogs, the corruption detection and correction of the program memory, the recourse of a JTAG channel to reprogram a deficient processing unit, and the takeover of the whole system by one processing unit in the event of a permanent failure of the second one.
Each technique has been thoroughly tested individually in the presence of bit-flip injection. Additionally, a test of the whole protective stack showed very positive results since the system has been able to run successfully for more than 8 hours sustaining a bit-flip density 250 times higher than the expected on-orbit rate.

# Preface

This thesis is submitted to the Norwegian University of Science and Technology as partial fulfillment of the requirements for the European Master in Embedded Computer Systems (EMECS) degree.

This work has been performed at the Department of Electronics and Telecommunications, NTNU, under the direction of Assoc. Prof. Bjørn B. Larsen.

## Acknowledgments

I would like to extend my gratitude to my supervisor Bjørn B. Larsen and co-supervisor Roger Birkeland for their guidance and support, and I particularly thank Roger for having suggested this very interesting and exciting research topic to me.
I would like to thank my friend Anders T. Akre for his invaluable help, as well as all the students working on the NUTS project who have always been available to answer my questions.
Finally, I dedicate this thesis to all my friends from EMECS with whom I spent two fantastic years – et surtout à mes parents qui m'ont toujours accompagné dans mes choix et fait tout leur possible pour m'offrir l'éducation dont je rêvais.

# Contents

**Appendices**

# List of Figures

# List of Tables

# List of Acronyms

**ADCS** Attitude Determination and Control System.

**COTS** Commercial Off-The-Shelf.

**CP** Concurrent Periodic.

**CRC** Cyclic Redundancy Check.

**DD** Displacement Damage.

**ECC** Error Correcting Code.

**EDAC** Error Detection and Correction.

**EDC** Error Detecting Code.

**EDC/ECC** Error Detecting Code / Error Correcting Code.

**EEPROM** Electrically Erasable Programmable Read-Only Memory.

**EPROM** Erasable Programmable Read-Only Memory.

**EPS** Energy Power System.

**FPGA** Field Programmable Gate Array.

**GPIO** General Purpose Input/Output.

**IR** infrared.

**ISR** Interrupt Service Routines.

**JTAG** Joint Test Action Group.

**LED** Light-Emitting Diode.

**LEO** Low Earth Orbit.

**LFSR** Linear Feedback Shift Register.

**MCU** microcontroller.

**NAROM** Norwegian Center for Space-related Education.

**NCL** Non-Concurrent Linear.

**NTNU** Norwegian University of Science and Technology.

**NUTS** NTNU - Test Satellite.

**OBC** On-Bord Computer.

**PC** Program Counter.

**PMCDC** Program Memory Corruption Detection and Correction.

**rad-hard** radiation hardened.

**RAM** Random Access Memory.

**RSP** Resilient System Prototype.

**RTOS** Real-Time Operating System.

**SEB** Single Event Burnout.

**SEE** Single Event Effects.

**SEFI** Single Event Functional Interrupt.

**SEGR** Single Event Gate Rupture.

**SEL** Single Event Latch-up.

**SET** Single Event Transient.

**SEU** Single Event Upset.

**SOC** System-On-Chips.

**TAP** Test Access Port.

**TCK** Test Clock.

**TDI** Test Data In.

**TDO** Test Data Out.

**TID** Total Ionizing Dose.

**TMR** Triple-Modular Redundancy.

**TMS** Test Mode Select.

**USART** Universal Asynchronous Receiver/Transmitter.

**XIP** eXecute In Place.

# Chapter 1 Introduction

In 1999, California Polytechnic State University and Stanford University released a first draft of what would, with time, become the CubeSat standard [web14a]. The purpose was to provide students with the opportunity to design, build and operate low-cost satellites with obvious didactic purpose in a wide range of engineering disciplines. Today's widespread popularity, with dozens of ongoing projects [web14a], testifies to the success of CubeSat and its ability to meet its academic purposes.

In September 2010, the Norwegian University of Science and Technology (NTNU) initiated its second CubeSat project called NTNU - Test Satellite (NUTS) within the Norwegian student test satellite program run by the Norwegian Center for Space-related Education (NAROM) [web14b]. The scientific purpose of the project is to launch an infrared (IR) camera in order to record for the first time gravity waves in the atmosphere [web14b].

The first mission called NUTS-1 consists in the launch of a double cubesat "2U" (20x10x10cm) embedding a visible frequency-domain camera as scientific payload [Bir11] and that will have for purpose to evaluate and validate the multitude of technical choices that have been made the past four years.

## 1.1 NUTS' Architecture Overview

NUTS is designed with a modular architecture. Figure 1.1 provides an overview of the main sub-systems listed below:

- **On-Bord Computer (OBC)**: module responsible for the command scheduling and dispatching, the processing and storage of the payload data, and the event logging. The OBC comprises an Atmel AVR32UC3 microcontroller, 16

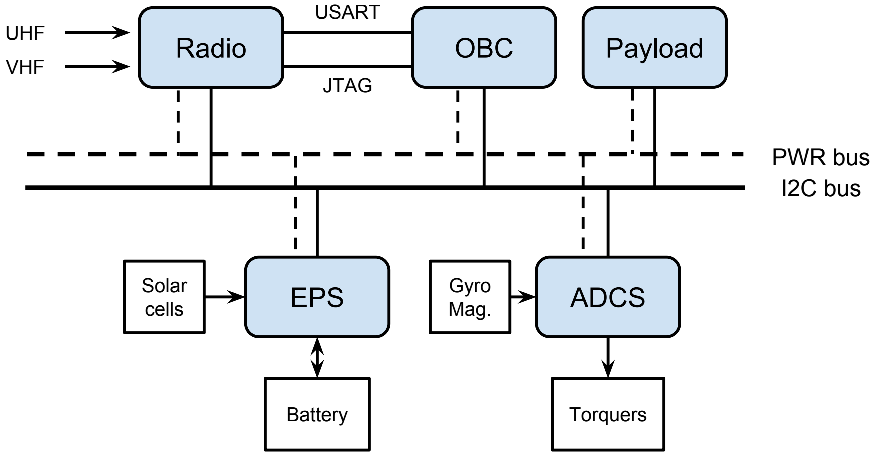

**Figure 1.1:** Overview of the architecture of NUTS. Adapted from [web14b].

MB of additional SRAM, 16 GB of NAND flash and fflash memory (a radiation hardened flash memory device). The OBC is one master of the main bus (I2C).

- **Radio**: module responsible for the bi-directional communication with the ground station. It comprises the same microcontroller and memories as the OBC and it can also been configured as an I2C bus master.
- **Energy Power System (EPS)**: module responsible for the storage and distribution of energy harvested from solar panels.
- **Attitude Determination and Control System (ADCS)**: module responsible for the determination and correction of the attitude of the satellite. It uses a combination of gyroscopes and a magnetometer for the attitude determination and some magneto-torquers for the attitude correction.
- **Payload**: NUTS' payload is a camera. NUTS-1 will embark a standard camera, while it should be an IR camera in the next missions.

All the modules are connected to the **backplane** which acts as the backbone of the whole system. The backplane provides each module with a double power-supply bus (one is redundant) and an access to the main bus (I2C). Additionally, both master microcontrollers (from the OBC and Radio) have a parallel bidirectional USART channel and the JTAG pins of each microcontroller is connected to some dedicated GPIO pins of the other microcontroller.

Software-wise, FreeRTOS is the Real-Time Operating System (RTOS) used on NUTS.

## 1.2 Specific Requirements Imposed to Space Technologies

The work presented in this report is one of the numerous contributions aiming at improving the reliability of NUTS [AS12] [deg13] [Vol11] [Gis12] [m12].
Space technologies are very challenging to design since they present a large number of engineering challenges inherent to remoteness and space environment. Some of those difficulties are:

- void,
- intense accelerations and vibrations during the launch phase,
- intense temperatures and temperature variations,
- energy harvesting and power management,
- radiations and their effect on electronic components,
- remoteness.

The last item of this non exhaustive list is actually the one that makes it so hard to design and operate such systems: no human intervention is possible once the device is on orbit and any failure of one part of the system is likely to lead to an end of the mission if no recovery is possible or has been planned to circumvent this failure.

## 1.3 Scope

NUTS can already rely on an Error Detection and Correction (EDAC) system implemented by Ødegaard [deg13], a robust data bus [Vol11] [Gis12], and a secure power management system [Bru11]. But none of those designs protects against hard failures of the processing units.

The microcontroller (MCU) in any application is the brain of the system. Since it is, among others, in charge of the control of the sub-systems, its failure will have major repercussions on the behavior of the system. Most likely any component controlled by it will become unavailable. In the case of NUTS, without any protection, a permanent failure of the Radio will result in an interruption of the communications with the satellite (only an autonomous beacon would still emit). The end of the mission would also occur if the OBC fails (cf. Section 1.1).

Given the dramatic consequences of the failure of a master MCU and the high probability expected of such an event to occur when using Commercial Off-The-Shelf (COTS) components [CR98] – as it is the case for any CubeSat, it seems clear that some effort must be done in order to contend with these kind of event.

With the present work, pursuing the task undertaken by Ødegaard [deg13], we attempt to offer to NUTS an additional layer of protection against processing units failures, based on cooperation-based recovery methods.
More specifically, given the lack of reliability of flash memories in space environment (cf. Paragraph 2.4.3), attention has been drawn to understand the effect of bit-flips in memory devices, and a solution has been researched to overcome them.
Most of the hardware of NUTS being already designed and in its final state, all the research has been based on the existing hardware and the solution proposed are exclusively software-based.

## 1.4 Contributions

A complete list of contributions to the NUTS project brought by this work is provided in Table 1.1.

## 1.5 Report Organization

The organization of the present report is described below:

- **Chapter 1: Introduction** describes the context of this work.
- **Chapter 2: Background** provides the reader with necessary informations about reliability and faults, fault-tolerance and redundancy, space radiation effects, as well as flash memories.
- **Chapter 3: Preliminary Work** presents the model employed to develop this work. A study of the effect of bit-flips in memory devices precedes a proposition of solution.

**Table 1.1:** List of contributions made by this work.

| **Task** | **Section** |
|---|---|
| Studied the effect of radiation-induced bit-flips in data and program memories. | 3.2, Appendix A |
| Presented a simplified bi-processing unit model of the system {OBC, Radio}. | 3.1 |
| Proposed a selection of techniques aiming at improving the reliability of the system. | 3.3 |
| Developed an experimental platform provided with a debug communication facility. | 4.4 |
| Designed a communication protocol stack over USART for exchange of information between the processing units. | 4.3 |
| Implemented a JTAG controller. | 4.5 |
| Provided the experimental platform with a bit-flip injection tool. | 4.6.1 |
| Implemented a facility for detecting and correcting program memory faults (PMCDC). | 5.1 |
| Implemented a facility for reprogramming the processing units (Reprogramming). | 5.3 |
| Implemented a mechanism for one MCU to take the control of the whole system in the event of the failure of the second one (Takeover). | 5.4 |
| Defined an algorithm to articulate the fault-tolerant techniques when integrated together. | 6 |
| Tested and evaluated the gain in reliability provided by each technique, individually and altogether. | 7 |

- ♣ **Chapter 4: Implementation (1): RSP Platform** is the first chapter in a set of three dealing with implementation considerations. This chapter presents the RSP platform.

- ♣ **Chapter 5: Implementation (2): Fault-Tolerant Techniques** goes into detail about the implementation of each fault-tolerant technique.

- ♣ **Chapter 6: Implementation (3): Articulation of the Fault-Tolerant Techniques** concludes with the integration and articulation of the techniques.

- ♣ **Chapter 7: Results** reports the results of the experiments carried out in order to assess the performance of the proposed solution.

- ♣ **Chapter 8: Related Work** presents a literature revue about related projects.

♣ **Chapter 9: Conclusions and Future Work** delivers the conclusions about the work performed during this thesis. Additionally, some propositions of improvement and future tasks are provided.

# Chapter 2 Background

This chapter provides the reader with an overview of some important concepts and definitions used throughout this report. The two first sections define some important terminology related to reliability and fault tolerance. The third section deals with space radiations and their consequences on electronics components, while the last part provides some informations about flash technology.

## 2.1 Reliability and Faults

This project is mainly about increasing the lifetime of a satellite, or, in other words, making the system more reliable and more tolerant to errors. Before proceeding we need to define precisely these terms.

### 2.1.1 Reliability, Failure and Faults

The main criteria of a dependable system is its **reliability**. Burns and Wellings [BW09] define the reliability of a system as "a measure of the success with which the system conforms to some authoritative specification of its behavior".
Any deviation from the specification of the behavior of the system is called a **failure**. Hence a highly reliable system is a system with low failure rate [BW09].
A failure originates from an **error**. An error is an unexpected problem internal to the system, while a failure is the external appearance. The mechanical or algorithmic cause of a error is called a **fault** [BW09].

A fault is said **active** when it leads to an error. Before this point it is said **dormant**. For instance a bit-flip in a register is a dormant fault as long as the register is not

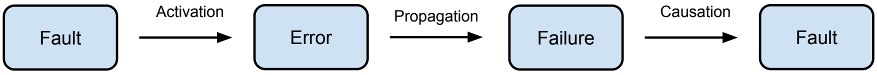

**Figure 2.1:** Fault, error, failure, and fault causation. Adapted from [BW09].

used, but activates as soon as it is (resulting in an error).

An error **propagates** throughout the system leading to a failure of the system. With our example, it happens when the wrong value stored in the register is used in some calculations, which leads to a wrong result that is used further in some other calculations. The chain of wrong results reaches the border of the subsystem resulting in a fault for the next subsystem called a **fault causation**. Ultimately, the whole system experiences a failure. This mechanism of fault propagation is summarized in Figure 2.1.

### 2.1.2 Classification of Faults

It is common to distinguish three types of faults [BW09]:

- **Transient faults**: faults introduced in the system at a particular time which remain in the system for some period of time before disappearing. They can be dormant during all their lifetime (which means that they do not generate an error), or can activate at some point (inducing an error). An example of such faults is a bit-flip in a RAM memory.

- **Permanent faults**: faults introduced in the system at a particular time which remain in the system until it is fixed. An example is a bit-flip in the program memory, a broken wire or a software design error.

- **Intermittent faults**: transient faults that occur from time to time. An example is a component sensitive to heat that stops working if the temperature is too high and starts working properly again when the devices cools down.

### 2.1.3 Fault Prevention and Fault Tolerance

There are two approaches available for increasing the reliability of a system [BW09]. **Fault prevention** is a method that attempts to eliminate as much fault as possible before the system enters its operational phase. It includes:

- using more reliable components such as radiation hardened (rad-hard) components,
- shielding the hardware to protect it against interferences,
- code verification,
- testing.

The second approach is **fault tolerance** which focuses on enabling a system to remain operational even in the presence of failures. We can distinguish between three levels of fault tolerance [BW09]:

- **Full fault tolerance**: the system continues to operate without any loss of functionality or performance in the presence of faults.
- **Graceful degradation**: the system continues to operate in the presence of faults, accepting some loss of functionality or performance during the phase of recovery or reparation.
- **Fail safe**: the system accepts a temporary halt in its operation. Before interrupting its service the system sets up a safe configuration.

More than one level of fault tolerance can be employed within one system.

## 2.2 Fault Tolerance and Redundancy

We have seen in the previous section that both fault prevention and fault tolerance can be utilized to improve the reliability of a system. Since this work is focused on the latter, we now introduce the keystone of fault tolerance: **redundancy** [BW09].

### 2.2.1 Temporal, Spatial and Value Redundancy

Fault tolerance is achieved by introducing redundant elements in the system in order to perform detection and recovery of failures. One can distinguish between [Dor03]:

- **Temporal redundancy**: consists in executing consecutively several times the same action. Example: retransmission of a message.
- **Spatial redundancy**: consists in the addition to the system of one or more copy of the same component (hardware or software).
- **Value redundancy**: consists in storing duplicated information. Parity bit or Error Correcting Code (ECC) are examples of value redundancy techniques.

### 2.2.2 Static and Dynamic Redundancy

The literature lists two types of redundancy [BW09]:

- **Static redundancy**: the redundant components within a system masks the effects of faults. Triple-Modular Redundancy (TMR) and majority voting are examples of static redundant systems.
- **Dynamic redundancy**: the redundancy within the system indicates whether an output is valid or erroneous. It is only an error detection facility and an other system must perform recovery. Check-sums and parity-bits are examples of dynamic redundant techniques.

Redundancy can be supplied by hardware as well as software. Applied in software, static and dynamic redundancy are respectively called **N-version programming** and **error detection and recovery**.

## 2.3 Space Radiation Effects On Electronic Components

One of the aspect that makes the design of space technologies so challenging is the extreme constraints imposed by the harsh outer space environment. Any satellite or other device supposed to operate in space have to be carefully designed to contend with void, extreme temperature variations, intense accelerations and space radiation.

Radiations in space are produced by particles emitted from different sources within or outside the solar system. Those radiations can degrade and cause failures to the embedded electronic components [Har14]. They are a primary concern for NUTS and other CubeSats because they embed COTS components that are not radiation tolerant.

### 2.3.1 Why Should We Care About Space Radiations?

The present section deals with the origin of space radiation and lists some of their effect on electronic components. But we can illustrate here and now the importance of including space radiations in the list of top-priority design constraints from the earliest stage of the design process.

Bekkeng in [Bek] provides an example of disastrous consequence of a single bit-flip on the OBC of a satellite called Clementine. Hit by an energetic particle resulting in a bit-flip, the OBC sent an unintentional command to the attitude control system. By the time the OBC had rebooted, the satellite was flipping very fast and the fuel tank was empty. This lead to the end of the mission.

The conclusion of this paragraph is that one single bit-flip can happen at any time, and can eventually result in disastrous consequences – namely the end of a mission.

### 2.3.2 Low Earth Orbit Environment

Due to the fusion reactions that take place in its heart, the Sun expels permanently in all directions particles (electrons, protons and heavier nuclei) that form the **solar wind**. Solar wind intensity varies with the activity of the Sun. Combined with particles produced by other stars and supernovas, they form the **primary cosmic rays**. Deviated and trapped by Earth's magnetic field, they form radiation belts called **Van Allen Radiation belts** [Har14]. Figure 2.2 illustrates the two main belts surrounding the Earth.
Because the belts descend lower around the poles, the satellites navigating in a **Low Earth Orbit (LEO)** with a high latitude (> 45°), such as NUTS, encounter a much lager amount of trapped electrons than satellites orbiting in a lower latitude (< 30°).

The primary cosmic rays interacts with high altitude atmosphere which yields secondary rays. The combination of the two flavors of radiations forms the LEO space radiation environment [Har14] in which NUTS will spend its life.

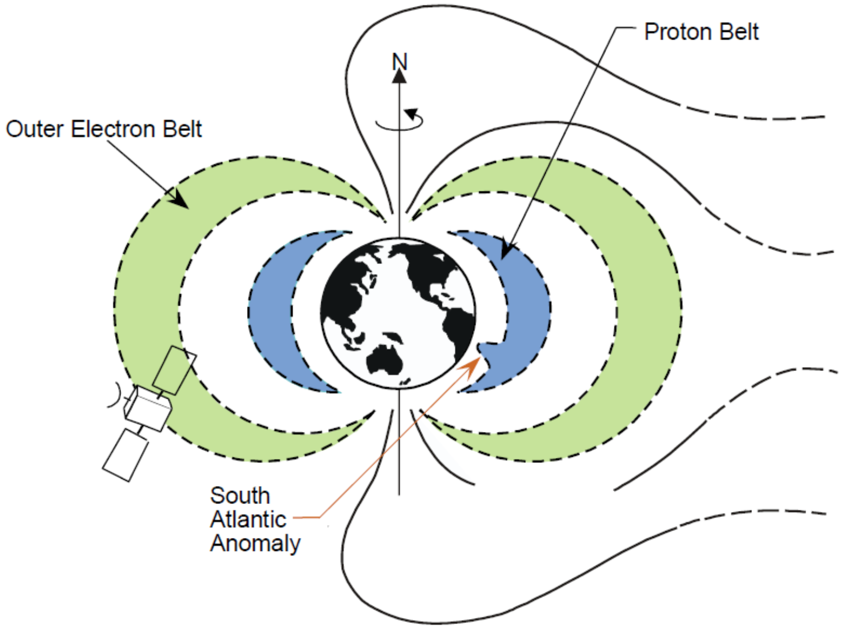

**Figure 2.2:** Illustration of the electromagnetic belts. Reproduced from [Joh].

### 2.3.3 Effects on Electronic Components

Space radiations can alter the behavior of embedded electronic components by degrading the properties of the silicon and altering the state of a gate. Those perturbations are the result of three distinct effects: the **total ionization dose effect**, the **displacement damage effect** and **single event effects**.

#### 2.3.3.1 Total Ionizing Dose Effect

When particles such as protons or electrons present in electromagnetic belts penetrate the electronic components, they can be trapped in the silicon and build-up charges in the gate oxide, resulting in a shift in the threshold voltage. If the shift is large enough, a transistor can get stuck in one or the other state, modifying permanently the behavior of the circuit. This phenomena is called Total Ionizing Dose (TID) effect [OM03]. The TID effect is therefore a cumulative action which degrades the performances of the device over time.

TID is measured in dose, which is defined as the amount of energy deposited per unit mass of material [Lau11]. According to [Har14], the typical dose rate that a satellite orbiting on a high latitude orbit faces is about 1 to 10 krad(Si)$\cdot$year$^{-1}$. From the same source, the admissible total dose that can support a commercial electronic component before failure is 2 to 10 krad. Hence the lifetime of the electronic components is limited to about 1 year due to total dose effect. The figures are rather old (1999) and new components have probably different properties and tolerance to radiation dose, but it provides a useful order of magnitude for the maximal lifetime of the satellite.

#### 2.3.3.2 Displacement Damage Effect

Energetic particles such as protons and heavy ions can also dissipate their energy by displacing silicon atoms from their proper lattice location creating crystal defects. The electrical properties of the device being changed, this can permanently alter the behavior of the circuit resulting in hard errors. This phenomena is called Displacement Damage (DD) effect [Bek].
The DD effect is important for solar panels since it is the origin of the gradual reduction of the power output delivered.

#### 2.3.3.3 Single Event Effects

If TID and DD effects act over time, space radiations can also have immediate effects. High energy heavy ions produced by stars (especially the Sun) can penetrate the electronic components and can cause transient pulses in the logic. Each of these events is initiated by the strike of one single particle leading to the name of **Single Event Effects (SEE)** [Har14]. SEEs can have various effects on the system ranging from inoffensive transient soft errors to destructive hard errors [Lau11]. Some soft errors includes [Liu01]:

- **Single Event Transient (SET)**: A brief voltage spike occurs at the node.
- **Single Event Upset (SEU)**: An erroneous signal such as a flipped bit or logic state is produced.
- **Single Event Functional Interrupt (SEFI)**: A component misbehaves due to the corruption of the control logic.

Hard errors count:

- **Single Event Latch-up (SEL)**: An excessive amount of charges may incur damage to the chip due to the apparition of excessive currents. A SEL can be cleared by power cycling the device.
- **Single Event Burnout (SEB)**: Localized current results in a catastrophic device failure.
- **Single Event Gate Rupture (SEGR)**: The impact of an energetic particle results in a gate oxide breakdown.

According to Harkins [Har14], the SEE rate expected for a satellite located on a high latitude orbit is $10^{-5}$ error·bit$^{-1}$·day$^{-1}$. Since we did not manage to find more up-to-date data, we will use this value in the remaining of this report as the on orbit bit-flip density expected for the NUTS missions.

## 2.4 Flash Memories Description and Reliability

The majority of this work deals with the detection and correction of bit-flips in the program memory of the two MCUs embedded in NUTS. Both of them are Atmel chips which use a NOR flash memory as program memory. This is the reason why a section is dedicated to flash technologies.

### 2.4.1 Overview of Flash Memory Technologies

Flash memories are non-volatile memory devices present in most modern System-On-Chips (SOC), that can be erased and reprogrammed by blocs of memory [Che09]. Replacing the old Erasable Programmable Read-Only Memory (EPROM) and Electrically Erasable Programmable Read-Only Memory (EEPROM), this technology offers the advantages of high bit-density and performance.

Two technologies are available on the market. **NOR** memories offer low-capacity high-read performance and the possibility of executing source code directly from the flash memory called eXecute In Place (XIP). **NAND** memories are by contrast high-capacity data storage with high erase-and-write rates [Che09]. While NOR flash memories are straightforward to use, NAND memories requires sophisticated I/O interface as well as a virtual memory mapping in order to avoid writing into bad blocks [Che09]. Both NAND and NOR Flash memories are present in NUTS' hardware.

Although reading can be achieved relatively easily and quickly on both NAND and NOR technologies, writing is much more challenging and very slow (few milliseconds)

[DNNJ99]. That is the reason why flash memories are subdivided in blocks and a write operation can only be performed on a whole block. Moreover, since the sequence of operations needed to carry out a write, a complex command state machine is used. The complexity of the overall system makes it difficult to assess the sensitivity of flash memory devices to space radiation.

### 2.4.2 Flash Memory Limitations

All current architectures of flash memory suffer from several reliability concerns. Among other, we can mention [Che09]:

- Bit-flipping: a bit stored on the memory is flipped. More frequent in NAND than NOR technologies, it is recommended to use an Error Detecting Code / Error Correcting Code (EDC/ECC) if it is utilized to store critical data such as configuration files.
- Bad-block handling: NAND devices are shipped with invalid blocs that must be tracked and avoided.
- Endurance: maximal number of erase/write cycles for a memory device. It is typically of one million cycles for a NAND device, and only few hundreds of thousands for a NOR device.
- Retention: ability to retain data over time. It depends on the number of erase/write cycles.

### 2.4.3 Reliability of Flash Devices in Radiated Environments

In this paragraph we are interested in the reliability of flash memory devices in space environment.

Chen in [Che09] states that flash memories are quite sensitive to radiations. Flash devices fail for radiation level as low as 20 krad(Si). He precises that they are relatively safe to use in read-only mode, but are much less reliable in erase/write mode. This would be due to failures of the charge-pump employed to program the memory blocs.
According to the same paper, flash memories are sensitive to SEUs which in some cases lead to catastrophic failures, particularly with modern multilevel flash memories. The same conclusions are shared by Cellere *et al.* [GCP01] who study the effect of high energy ions on flash devices. The authors qualify ion effect as a serious threat

to flash memories in space application, especially modern multilevel devices where the use of very thin variations of the threshold voltage can easily lead to bit-flips.

# Chapter 3
# Preliminary Work

As stated in Section 1.3, this work focuses on how to make NUTS resistant to bit-flips in its memory devices by leveraging the existing hardware. In this chapter we first enunciate the hypothesis on which the work is built on, and presents the model that will serve on base for the development of a solution. We then present a qualitative study of the effects of a bit-flip in a memory device and we finally propose a four-layer stack of protections destined to prolong the lifetime the satellite in a radiated environment.

## 3.1 Presentation of the Model

In this section we present the model of the system which will direct our technical choices in the design of a system resistant to faults in its memory devices.

### 3.1.1 Boundaries of the System

According to the problem description, this work is limited to the two main processing units – OBC and Radio module – and their interaction as illustrated in Figure 3.1.

### 3.1.2 Addition of a Parallel Communication Chanel

Initially only one communication channel was planned to link the Radio to the OBC module: the main bus (I2C) ensuring the dialog between all the modules of the satellite. But since it was difficult to forecast its response time and loading, it did not seemed to be the best option to send urgent messages between the two control

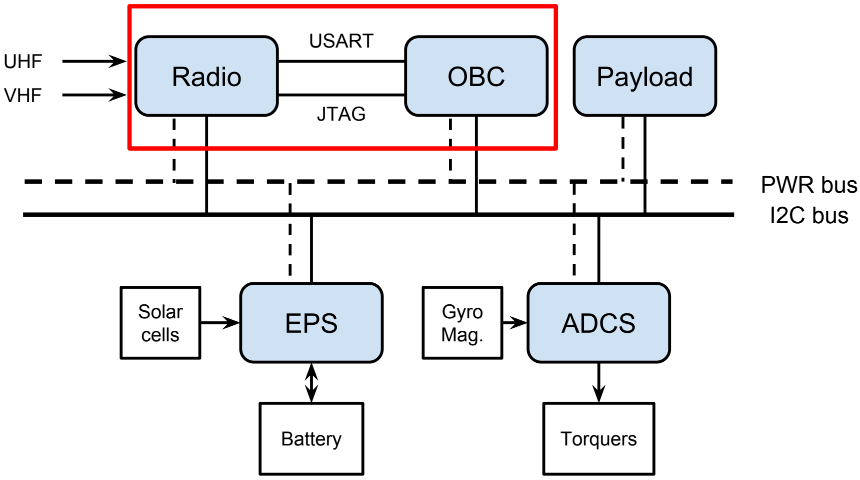

**Figure 3.1:** Boundary of the system considered in this work.

modules (OBC and Radio). Moreover due to the increased complexity of the protocol and the hardware, it seemed advantageous to deploy a parallel bidirectional USART channel using exclusively two GPIO pins in both MCUs. I supported the addition of this secondary channel and it is now officially part of the design. This secondary communication channel is only employed to transmit messages related to the activity of both modules.

A double JTAG connection (the JTAG pins of on MCU being connected to four GPIO pins on the second MCU and *vice versa*) was already present on the schematics but not used.

### 3.1.3 Simplification of the System

This task being a research work, the hardware of the satellite has not been used directly, which would otherwise have required the implementation of some drivers for the memories and would have made the experimentations more difficult. Instead, a simplified system, also called **model**, has been employed to develop and test the ideas using exclusively two Atmel Xplained boards (embedding an AVR32UC3A MCU). Great care has been taken so that the implementation of the simplified system could be reused easily on the real hardware of the satellite. This section describes this model.

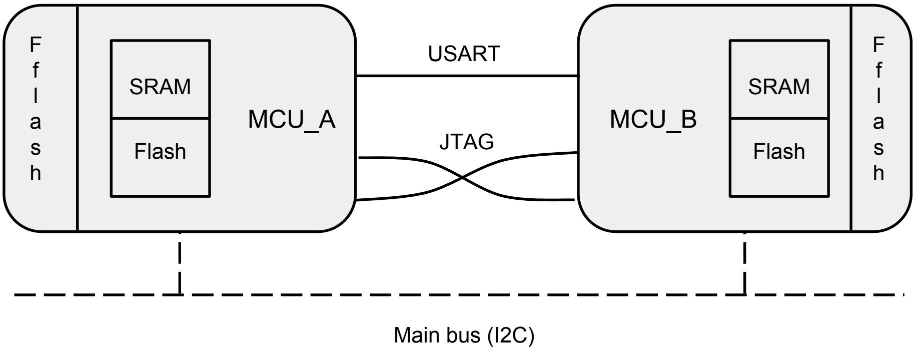

**Figure 3.2:** Schematic representation of the model.

As explained in Paragraph 1.1, the OBC and Radio modules contain both an AVR32UC3A MCU and a similar set of memories (SRAM, NAND flash and fflash). Since we extracted both modules from the global system, our model does not contain any assumption about the role played by each of them. Moreover, because they are so similar in term of hardware, the model does not make any distinction between the Radio and the OBC. Each of them is mapped on two distinct MCUs called **MCU_A** and **MCU_B**.
Contrary to the additional SRAMs and NAND flash memory devices that are ignored in the model since they do not bring any additional functionality, the fflash is preserved because it has the particularity of being rad-hard. In our model, the fflash memories are mapped to the second half of the internal flash memory of the MCU.

A schematic of the simplified system used to develop, implement and experiment the fault tolerant techniques is finally presented in Figure 3.2, while a photography of the experimental platform can be seen in Figure 3.3.

## 3.2 Study of the Effects of a Bit-flip in a Memory Device

This section deals with the effects of bit-flips in the data and program memories. For the latter a model is presented and the case where the program memory is periodically cleared from any bit-flip is also considered.

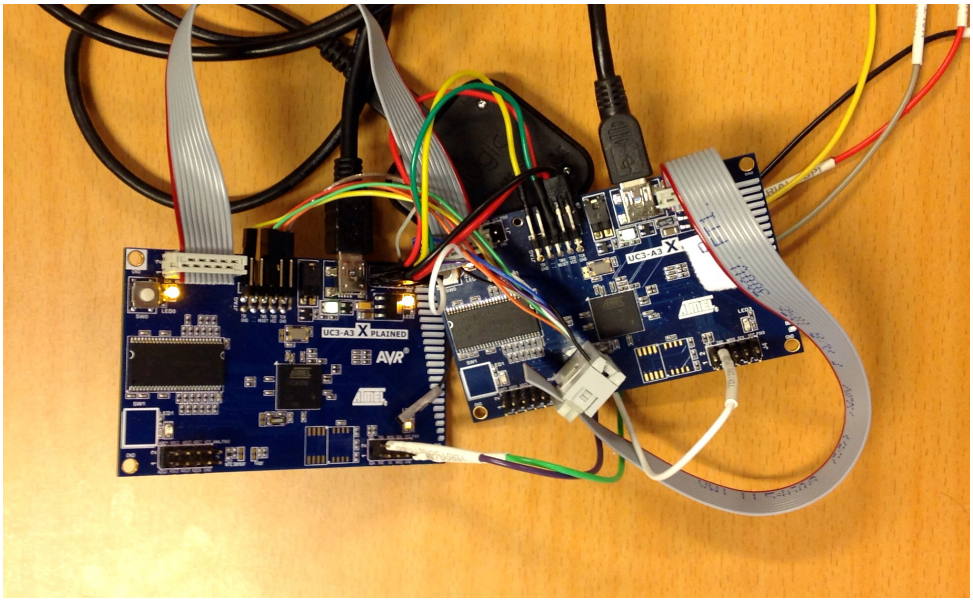

**Figure 3.3:** Photography of the experimental platform.

### 3.2.1 Effects of Bit-flips In the Data Memory

The data memory, contains the heap, stack and some other data used by the program. The information stored in data memory is volatile meaning that it is peculiar to the instance of the program being currently executed and therefore does not need to be remembered to run a new instance. In our case, the data memory is implemented with an SRAM which is a fast access volatile memory (it looses its content as soon as the power supply is removed).

#### 3.2.1.1 Model of the Data Memory

The compilers typically use the RAM to store the stack and the heap. Additionnaly, with FreeRTOS each task having its own stack [Bar09], the RAM contains the heap, the main stack and the stacks of the running tasks. Figure 3.4 illustrates a simple model of the data memory.

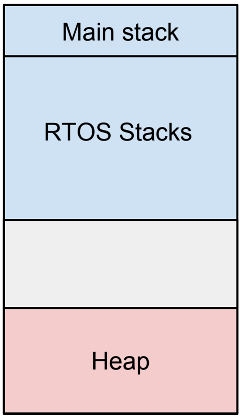

**Figure 3.4:** Diagram illustrating the model of the data memory.

#### 3.2.1.2 Consequences of a Bit-flip

First of all it is clear that a bit-flip in the data memory is a transient fault since the data stored in RAM are only related to the current instance of the program being executed. For instance restarting the program would clear any error altering the data memory.

There are several cases to consider depending where the bit-flip occurs and what kind of data it corrupted:

- Data stored in the heap or local data stored in the stack:
  - If the data is not an address, it will lead to a wrong result and eventually lead to an error due to the ulterior use of that faulty result. This chain reaction can be slow depending at which frequency the corrupted data is used in subsequent computations.
  - If the data is an address, it will certainly result in an error because of alignment or bound limitation, or simply fetching a wrong value.
- Stack is hit on a return address or stack pointer: will eventually (likely immediately) result in an error.

That being said, a bit-flip might also be harmless if it occurs outside any heap or stack, or if it hits a local variable that is not used anymore (just before a return for instance).

The conclusion is that the more tasks run and the more function calls (*i.e.* large stack), the more the probability for a bit-flip in the data memory to result in an error. A bit-flip may remain dormant for quite a long time or even remain silent.

#### 3.2.1.3 Possible Solutions

**Rebooting** The first obvious solution to recover for such a fault is to reboot the MCU. By doing so, all the bit-flips are naturally removed from the stacks and heap. This is a good idea provided that one can tolerate an interruption of the service delivered by the system.

**EDC/ECC in the Data Memory** Another way to recover from a bit-flip in the RAM is to use an EDC/ECC such as the one proposed in [deg13] [Whi82]. Each data stored in the RAM is encoded using an EDC/ECC which enables to get the initial value if the data happens to be corrupted (within the theoretical limits offered by the code).

**Using Checkpointing** Lastly, all the checkpointing algorithms leveraging the backward redundancy are effective ways to deal with that kind of faults (among others) [deg13] [BW09] [TP10].

### 3.2.2 Effects of Bit-flips In the Program Memory

The program memory, implemented on a flash memory in our case, contains the sequence of instructions defining the software as well as constant data.

#### 3.2.2.1 Model of the Program Memory

Figure 3.5 illustrates a model of the program memory developed in this thesis. The execution flow of a concurrent program can be split up in two sections:

- **Non-Concurrent Linear (NCL) section**: non-concurrent linear part of the program corresponding to the boot sequence and the initializations. It ends at the launch of the RTOS kernel. In this section we also include the constant data that are loaded to RAM during the booting sequence.

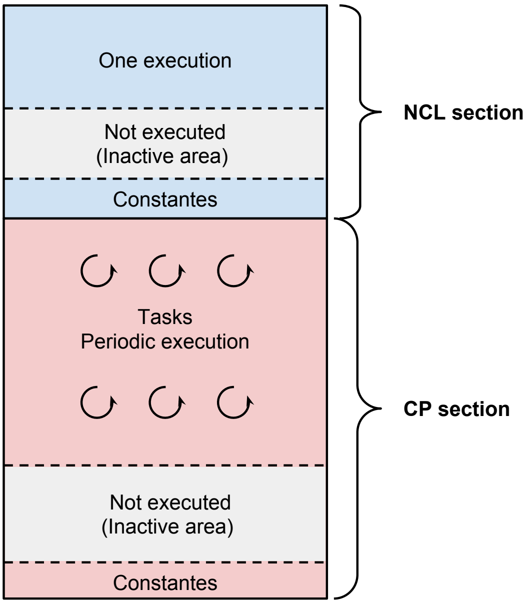

**Figure 3.5:** Diagram illustrating the model of the program memory.

- **Concurrent Periodic (CP) section**: concurrent part starting after the launch of the RTOS kernel that consists in a set of concurrent tasks and daughter functions executing periodically (each task having its own frequency) as well as some Interrupt Service Routines (ISR) called sporadically. We do not make any distinction between a task and the functions called by that task: they belong to the same entity executing at a same frequency (or multiple of that frequency). The RTOS kernel is no exception and can be seen as a set of tasks executed at very high frequency. The CP section also includes constant data that are accessed directly by the program (they are not copied to RAM during the booting sequence).

In both sections, all the instructions are not executed all the time due to the presence of conditions. Those instructions form the inactive area of the section (grey areas on the figure) by opposition of the active area (in color). For the CP section, the inactive area is in perpetual evolution since each execution of a task will potentially select a new path in the condition tree.

**Note:** Contrary to the model of the data memory, this one does not respect the real layout of the memory. Each area represent a functional property providing no information on the memory layout.

#### 3.2.2.2 Consequences of a Bit-flip

First of all, it is clear that, contrary to a the case of the data memory, a bit-flip in a program memory is a permanent fault since the behavior of the device will remain affected permanently.

Our model of the program memory showed two distinct sections (NCL and CP). The latency of the expression of an error depends on which one is altered.

- If a bit-flip occurs in the NCL section, the fault will remain dormant until this section is executed again (provided that the bit-flip is located in the active area).
- If a bit-flip occurs in the CP section, then the fault will remain dormant until the next execution of the task it altered (if located in the active area). Therefore, the latency of the error is at most equal to the period of the altered task.

Once again, the effect of a bit-flip depends on the type of instruction/data that is altered:

- Instruction opcode: if the new opcode is not valid then an error occurs immediately. Should this not be the case, then if the size of the new instruction is the same as the initial one, the result of the expected computation is wrong with the chain reaction it implies. The last case is if the size of the new instruction does not match with the initial instruction. Then there is a shift in the input for the decoder which results in an immediate catastrophic collapse of the program.
- Operand or any data: if the data is not an address, it will lead to a wrong result and eventually yield to an error due to an implied chain reaction. If the data is an address, it will quickly result in an error because of alignment or bound limitation, or simply fetching a wrong value.

The conclusion is that a bit-flip in the program memory is a hard error that may remain silent until the next reboot, or result in an error after at most the period of the task it altered. There is very little chance an altered instruction does not result in an error quickly.
A consequence of the model is that the more tasks and the larger their frequency, the quicker a fault will result in an error.
Another corollary is that, paradoxically, contrary to the case of the data memory, an increase in the frequency of the bit-flips does not increase the risk of a failure. More detailed explanations are provided in Paragraph 7.2.1.2.

#### 3.2.2.3 Possible Solutions

Although bit-flips in the program memory are hard errors, they do not require human intervention as it would be the case to fix a broken wire for instance. There are several possible solutions available.

**EDC/ECC on Program Memory** A first idea could be to use an EDC/ECC for the instructions itself. Properly designed and using pipelining, it would probably be possible to keep an acceptable frequency.
However, a major difficulty would be to generate the encoded software from the code provided by the compiler because all the addresses would be shifted with different amounts (since an instruction or group of instructions would be encoded with a length larger than the initial length considered by the compiler). No mention or implementation of this method has been found in the literature.

**Code replication** Another way to increase the robustness of the system against corruption of the program memory could be to have several copies of a function at one's disposal and to use a new copy when the current one seems to be altered. This idea is proposed by Ødegaard [deg13] but without implementation or justification albeit whether copies achieve their purpose or not is not trivial.

When one use $n$ copies, one multiply by $n$ the memory area which in turns multiply by $n$ the probability of a bit-flip. Hence, the copies does not seem to help at all.
Actually it does because, if on average the $n$ copies would be corrupted at the same time, in reality one copy will remain unaffected longer than the other and especially longer than the average. Figure 3.6 presents the result of a simulation of 9 systems using 1 to 9 copies (1 meaning one instance of the program in memory). The ordinates represents the reliability improvement defined by the relation $i = \frac{r-1}{100}$, where $r$ is the ratio of the average time before failure with $n$ copies over the average time before failure with one copy.

We notice that using copies helps and the larger the number of copies the greater the reliability. With only two copies one can expect an improvement of 50% which is quite significant. Of course the additional memory used comes as a cost.

**Fixing the Program Memory** Another idea would be to correct the program memory whenever a failure is detected. The correct image of the program memory

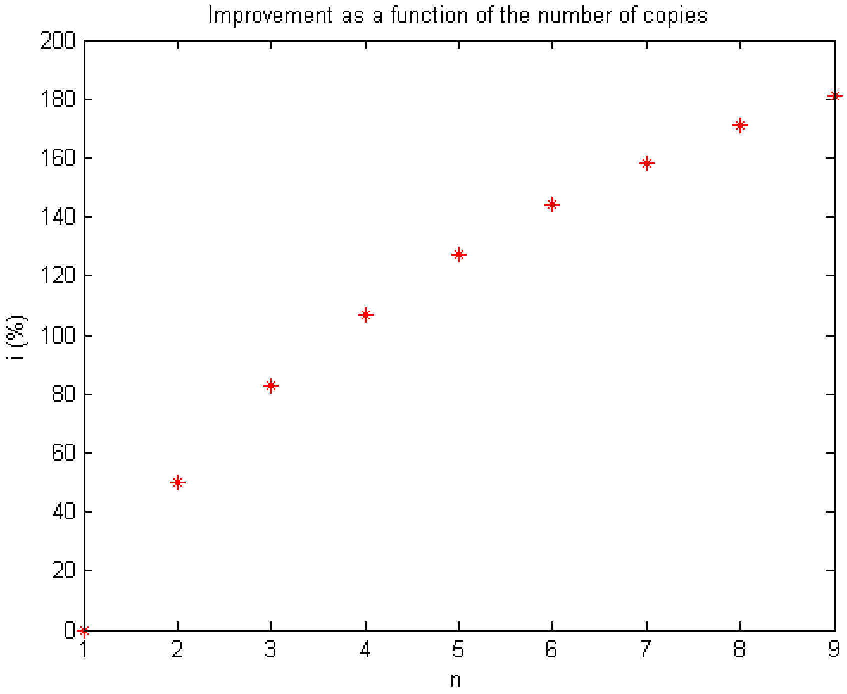

**Figure 3.6:** Reliability improvement as a function of the number of copies.

would be restored from a safe backup image of the program stored in a rad-hard memory.

A variant of this technique would be to check and detect periodically a corruption of the program memory by the use of an Error Detecting Code (EDC). The correction would happen by using the safe image stored in a rad-hard memory. The idea is proposed by Obispo [Fit12] and mentioned by Ødegaard [deg13].

If it is obvious that this solution cannot harm the reliability it is not granted that it would help. Figure 3.7 illustrates the probability of getting an error when a bit-flip is introduced in a periodic function of frequency $f_t$ depending on the frequency $f_c$ with which the program memory is checked and fixed. The blue curve shows the probability of failure as a function of $a = \frac{f_c}{f_t}$, and the red dots are the results of an experimentation. The mathematical expression and the demonstration are provided in Appendix A as part of this thesis.

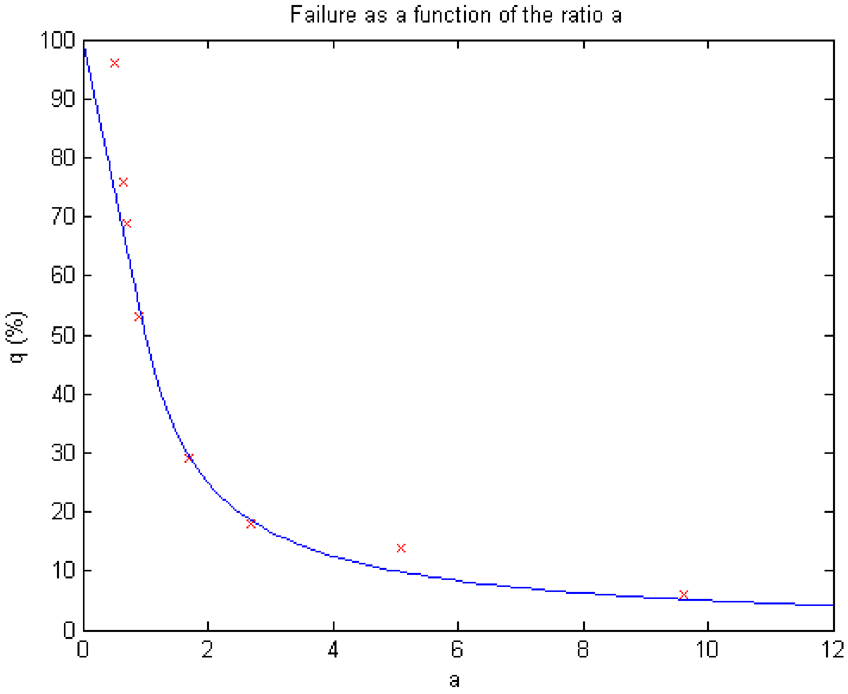

**Figure 3.7:** Error probability as a function of $a$.

We can notice that this method would only be efficient if the frequency of the memory fixing is at least lower than the frequency of the tasks.

**Process Pair** On a system having two processors, it is also possible to use a process pair such as the one proposed by Torres-Pomales in [TP10]. Both processors carry out the same computations and their outputs are analyzed. When they match, a switch selects the output of the first processor. However, the switch selects the backup processor when it considers that the primary one does not deliver correct results (if any). While the secondary processor is running, the primary one can be fixed (using the method described in the previous paragraph).
This technique offers the advantage of being able to recover from a hard-error seamlessly without suffering from any service interruption. However, it requires some additional hardware and the decision about which processor is delivering erroneous results is difficult with only two processors since no voting algorithm can be used.

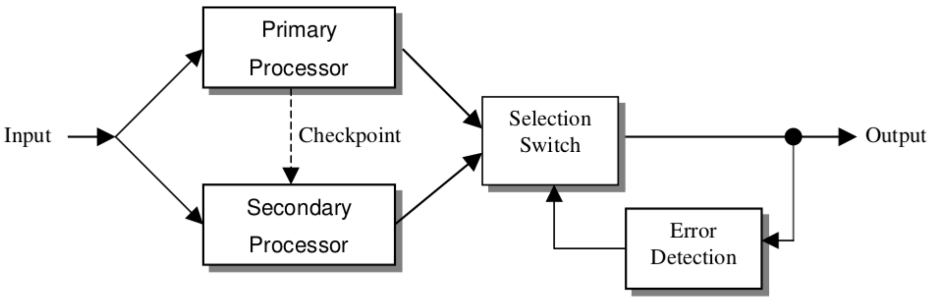

**Figure 3.8:** Logical representation of a process pair. Reproduced from [TP10].

**Master-Slave Configuration** The Master-Slave method is a generalization with $n$ processors of the process pair discussed previously. An example of implementation is proposed by Caldwell *et al.* in [RH01] and [CR98]. It removes the main drawback of the process pair by allowing the use of voting algorithm, but it requires even more additional hardware and the algorithm becomes quite complex. NUTS embedding only two processors, this method cannot be employed.

**Takeover** In the case of a system containing two processors, if one fails, the second can run the critical tasks of the first one in addition of its own. If its resources are not large enough to run all those tasks, it can run only a subset of its tasks.
This offers the possibility of providing a degraded service at very low cost.

## 3.3 Proposed Stack of Protection Layers Against Bit-flips

After having got acquainted with the architecture of the system, the effect of bit-flips in both data and program memories and presented some solutions to contend with this kind of faults, in this section we gather all this information and discuss each solution to measure how well they can answer our problem. Finally, we propose a stack of four solutions which is implemented and fully tested in Chapters 5 and 7 respectively.

### 3.3.1 Design Requirements

The purpose of this work is to make NUTS more reliable and make it deliver its service over the longest possible period of time in a radiated environment, by leveraging the existing hardware. Unless having recourse to fault prevention such as the use of electromagnetic shield and rad-hard components, which is outside the scope of this work, the solution is to achieve fault tolerance.
Paragraph 2.1.3 defines three levels of fault tolerance. The purpose of this paragraph is to choose the appropriate one depending on the requirements of the mission.

The strongest level of fault tolerance – called full fault tolerance – guaranties that the system will continue to deliver an uninterrupted and accurate service even in the presence of faults. This is suitable for systems where material or physical damages could be caused due to a failure of the system.
It is very difficult and costly to achieve, and would require a lot of redundant components which is contrary to the philosophy of CubeSat satellites that are systems extremely constrained in term of volume, weight, power supply and cost. With such limitations, to implement a full fault tolerant system does not seems reasonable.

Actually, an interruption of the service provided by NUTS is more than acceptable as well as the delivery of a partial service. Therefore the appropriate level of fault tolerance lies between fail safe and graceful degradation. An interruption of the system is acceptable because no life or good is in danger and a failure of the transmission of some data can be attempted again on the next passing over the ground station.

Finally, what we want to achieve is a system that can recover from any corruption of any memory device present in the system while providing a degraded or interrupted service during the phase of recovery, and providing full service once the recovery is complete. In case the system cannot fully recover, it should then deliver a degraded service.
The solutions should exclusively make use of the hardware already designed and should have a minimal impact on the resources of the satellite such as processing time, performance and power consumption, and memory usage.

### 3.3.2 Fault and Failure Detection

The first step to get a fault tolerant system is to be able to detect a fault – in our case a bit-flip – and a failure. This is the topic of this section.

#### 3.3.2.1 Fault Detection

**Error Detection Codes and Checksums** The main fault – bit-flip in our context – detection method discussed in the literature is the use of EDC [Whi82] [deg13], where the data is stored along with a code redundantly encoding part of the data (making use of space redundancy).

A variant is to compute checksums and store them in a reserved space in memory. It facilitates mass checking and consumes less memory since one checksum value can be used for a block of data instead of a unique piece of data.
When implementing such a scheme, it is easy to fall in the pitfall of using a look-up table in order to get a better throughput. Firstly, although it complies with the will of minimizing the processing time utilization, it violates the will of keeping a small print in memory. Ødegaard [deg13] for instance uses a look-up table of several KB which is not negligible.
Secondly, since the bit-flips rate is proportional to the area occupied in memory, this look-up table would become quickly corrupted and the EDC values would report errors although the data is perfectly correct. Hence, in this radiated context it seems more reasonable to avoid look-up table-based implementations.

One interesting choice to implement such a scheme is the use of a maximal length Galois Linear Feedback Shift Register (LFSR) which requires very little software (only 8 lines of C code), is very fast (cf. Chapter 7), and provides a very good discrimination ($2^{32} - 1$ values for a 32-bit LFSR [RW07]).

#### 3.3.2.2 Failure Detection

We dispose of several methods in order to detect the failure of a module or part of a module.

**Majority Voting** A lot of papers ([BW09] [CR98] [TP10]) propose to use a majority vote algorithm in order to detect a failure, but this implies that the system has at least three processors dedicated to one unique task. Our system counts only two of them and each carries out different tasks, which excludes any vote algorithm as well as the process pair configuration mentioned by Torres-Pomales in [TP10].

**Test** Ødegaard in [deg13] proposes the use of a test that would detect a failure in the most critical parts of the system if the result of this test is not conform to an expected value. The main critic of this method is that it does not provide either any information about the nature of the fault or its location. Moreover, in the event of an error, instead of giving back a wrong result, it would most likely result in a hard error forcing the MCU to reboot.

**Watchdogs** The following solution is with no doubt the simplest one and the most efficient. It consists in using software or hardware watchdogs. Most of the CubeSat satellites uses this efficient protection ([deg13], [Man11] [Fit12]). If the system fails to reset a counter, a reboot is instantiated. It is a necessary protection layer since it catches all errors (anticipated or not) that the previous protection layers have missed. One of its advantage is that it is the simplest and efficient solution to implement. The drawback is that it goes with an interruption of the system.

**Heartbeats** In some cases it happens that the reboot is not enough (as it is the case if a bit-flip affects the booting sequence for instance). In this case, a second processor should be aware of the out-of-order state of the first processor so that it can take some corrective actions (cf. next section).
This failure detection of an other module can be accomplished by the periodic transmission of short messages, called heartbeats, with the only purpose of informing a watcher module that it is properly running. As soon as one heartbeat is missing, the listened module is considered as out-of-order. This idea is proposed and implemented in [deg13].

### 3.3.3 Recovery

Once a fault or failure has been detected it is necessary to choose an appropriate corrective method in order to restore the system into a working state. The discussion of several recovery methods is the object of the present section.

#### 3.3.3.1 Recovery From Corruption of the Data Memory

**EDC/ECC on Data Memory** Ødegaard in [deg13] proposes to use Error Correcting Code (ECC) in data memory. The advantage of this method is that it enables to recover from data memory corruption without interrupting the service.
As explained earlier (cf. Paragraph 3.3.2.1) the implementation must be chosen carefully in order to minimize the impact on memory usage. Furthermore, the implementation of Ødegaard present the particularity of storing the data in flash memory which present the disadvantage of being time consuming. Moreover he applies EDC/ECC to only a subset of the stack and heap which he refers to as *critical*

*data*. But there is no such thing as a critical data when it comes to fault tolerance. Any corrupted pointer, even if considered as not critical, can lead to dramatic crash of the module. The EDC/ECC should be applied to any data stored in data memory.

**Checkpointing** Checkpointing is also proposed and implemented in [deg13]. This is also an interesting feature since it enables to recover from a lot of soft errors without interrupting the service. Two major concerns about this method are that it is complex to implement and is greedy of memory for the storage of backup images [BW09]. In our space radiated context this is clumsy since if a backup image is corrupted, then the whole checkpointing process will fail.

**Reboot** The last solution, also the simplest to implement, is to use a reboot. We know from Paragraph 3.2.1.3 that rebooting the system clears any corruption of the data memory as well as clear the dangerous SELs if the reboot is operated by a power cycle. The drawback is the interruption of the service, but its obvious simplicity and effectiveness makes it an unavoidable solution.

#### 3.3.3.2 Recovery From Corruption of the Program Memory

**Process Pair and Master-Slave** Let us exclude straight away process pair and master-slave techniques that require at least respectively two and three processors per module although there is only one available per module in our case.

**EDC/ECC on Program Memory** One interesting solution would be the use of EDC/ECC for the program memory but the difficulty of generating the code presented in Paragraph 3.2.2.3 remains a major concern. Moreover the processing overhead and increase in power consumption are two arguments which lead us to believe that a software implementation would not meet the requirement for NUTS.

**Code Replication** Code replication is an interesting idea, but, aside from being extremely greedy of memory (it improves the reliability by only 83% with a reasonable number of three copies), it raise one serious concern. Even if we keep track of where the error appeared, we cannot be sure that the fault is located in the function where the error was detected. Indeed, the fault may have been dormant for some amount of time before triggering an error in another part of the code (and it is often the case). Therefore, it seems that this idea is viable only if it is coupled with a system of checksums to know precisely where the code is corrupted. If the idea seems straightforward, its implementation is much more complex.

**Memory Fixing** The next method consists in fixing the memory. If we know what part of the program memory is corrupted, it is easy to fix it provided that a safe backup image of the initial program is stored in a rad-hard location. Similarly to the code replication method, this technique should be coupled with a system of check-sums.

The advantage over the previous method is the reduced memory print. However, an added difficulty is that the source code of the drivers employed to write to the flash memory must be located away from the standard source code (from at least one flash page to be precise). Otherwise, if a writing operation happens on the same page as the driver, it would be deleted while being in use which would lead to a major failure.

Another drawback is that it implies to use the flash memory in write mode, which is not recommended in a radiated environment since it increases the risk of faults (cf. Paragraph 2.4.3).

Despite all those complications, fixing the program memory lets hope for a greater lifetime than the one expected with code replication.

**Reprogramming** One limit to the previous method resides in the fact that if the drivers employed to write into the internal flash memory and to read from the external fflash memory are corrupted, the attempt to fix the content of the program memory will fail. It cannot solve the problem of corruption of the booting sequence either.

One additional layer of protection would be to be able to reprogram the deficient device by using a JTAG connection. It still suffers from the problem of flash reliability in write mode (cf. Paragraph 2.4.3), but nothing prevents the device from being reprogrammed twice in a row if the first trial fails. This method works as long as both devices (the one being reprogrammed and the one that performs the reprogramming action) are not out-of-order at the same time, but this event being unlikely, the technique offers a great reliability improvement.

One more advantage of this technique is that with very little additional work it is possible to implement a code update facility which can be used for software update as well as multiple mission support [Fit12].

**Takeover** The takeover method described Paragraph 3.2.2.3 enables the system to deliver a functional degraded mode with one module becoming the only master of the satellite after the failure of the second module. The master module runs a

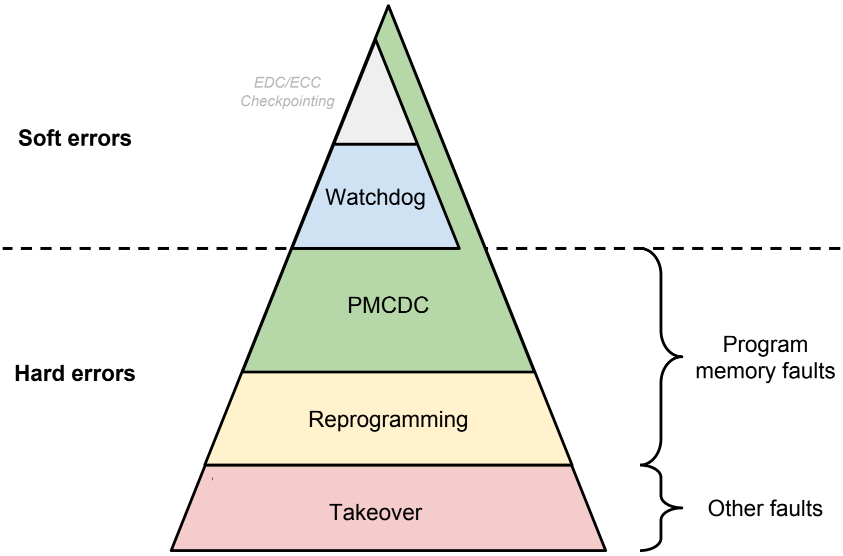

**Figure 3.9:** Proposed protective stack against bit-flips in memory devices.

critical subset of its tasks as well as the critical tasks of the out-of-order module. It is a last recourse measure but may offer an increased lifetime to the satellite in case the crash of one of the MCUs is the point of failure.

### 3.3.4 Proposed Protective Stack

After having discussed the methods in the context of NUTS, we present a selection of fault tolerant techniques that we implement and test in Chapters 5 and 7 respectively. Each technique is efficient at a different level in the maturity of the the error spawned by the fault, and all of them together form a protective stack presented in Figure 3.9.

The higher in the stack, the earlier the technique treats the symptoms set off by the fault. The methods on the top intervene when the fault is still dormant, while the ones on the bottom act when the system has already failed.
The stack is split up in two sets of fault-tolerant techniques. The upper part deals with soft errors (bit-flips in the data memory and other transient faults), while the second part is the set of techniques that treat hard errors (bit-flips in the program memory and other permanent faults in the case of the very bottom technique).

**Treatment of Soft Errors** Starting by the top of the stack, we find EDC/ECC and checkpointing that are present for completeness only. They have been implemented by Ødegaard [deg13] so they are not part of this work and whether they should be included in this stack or not is a design decision to be taken when the real protective stack will be implemented for NUTS.
The remaining technique is the use of the internal hardware watchdog which reboots the system if a counter has not been reset on time.

**PMCDC** Moving down to the part of the stack dealing with hard errors, we find the Program Memory Corruption Detection and Correction (PMCDC). It consists in using checksums of blocks of program memory, computed thanks to a Galois LFSR, in order to detect a program memory corruption. The altered block is replaced by an unaltered one stored in fflash memory. The place of this method in the stack is plural because PMCDC can operate at different level of action:

- the intervention of PMCDC can be periodic (preventive checks),
- or can take place in the booting sequence (curative).

This explains its spreading in the stack.

**Note:** The implementation of PMCDC contains a complex algorithm capable of detection a corruption of the PMCDC facility itself and able to fix it in the majority of the cases (only a corruption of the flash drivers is unrecoverable). All the details are provided in Section 5.1.3.

**Reprogramming** If PMCDC fails, the module will eventually undergo a failure because the faults in the program memory can accumulate. This is at this stage that the cooperation between both MCUs comes into action.
We use heartbeats so that each module know the state of the other. If a module is not responsive (heartbeat miss), the other module first operates an external reset and if it is not sufficient, it initiates a reprogramming action of the faulty module. The details of this procedure and the algorithm to detect a failure of one of the module are explained in Section 5.3.

**Takeover** The last fault tolerant technique of the stack acts when the reprogramming of the module could not be performed or did not restore it in a working state. In this case, the working module takes over and run a degraded mode where it performs its critical tasks as well a the critical tasks of the other module.

The algorithm which detects when a reprogramming failed and when a takeover action is needed, is described in Chapter 6.

**Remark** This work makes no usage of the exception vector which has been left empty. As a result, each time an exception is thrown, the processor executes an infinite empty loop. The handling of exception is obviously a very useful feature to reduce the number of service interruption but was avoided here for ease of analyze of the behavior of the system in presence of bit-flips.

# Chapter 4

# Implementation (1): RSP Platform

This chapter is the first of a series of three that describe implementation considerations. It provides an overview of the whole design and details some constructs on which lies the project such as the USART and debug communication channels as well as the JTAG controller. The second part deals with the implementation of the fault tolerant mechanisms listed in Paragraph 3.3.4, while the third chapter explains how all those mechanisms are articulated together.

This report cannot contain all the details of the source code that represent several thousands of line of code. Are described here only the main software mechanisms. For more details please consult the *read_me.txt* files that document thoroughly the project and the well commented source code if more details are needed.

## 4.1 Overview of the Project

The software project is called Resilient System Prototype (RSP). It gathers all the work produced during this thesis of which an overview is provided in Figure 4.1. RSP is a robust platform on which can be added prototype solutions to test multi-MCU fault tolerant techniques.

In blue are represented the constructs building the platform. We find:

- the USART communication channel enabling the transmission and reception of flags, heartbeats and messages between two MCUs,
- the Debug communication channel (over USART) enabling the transmission of data to a computer,
- the JTAG controller

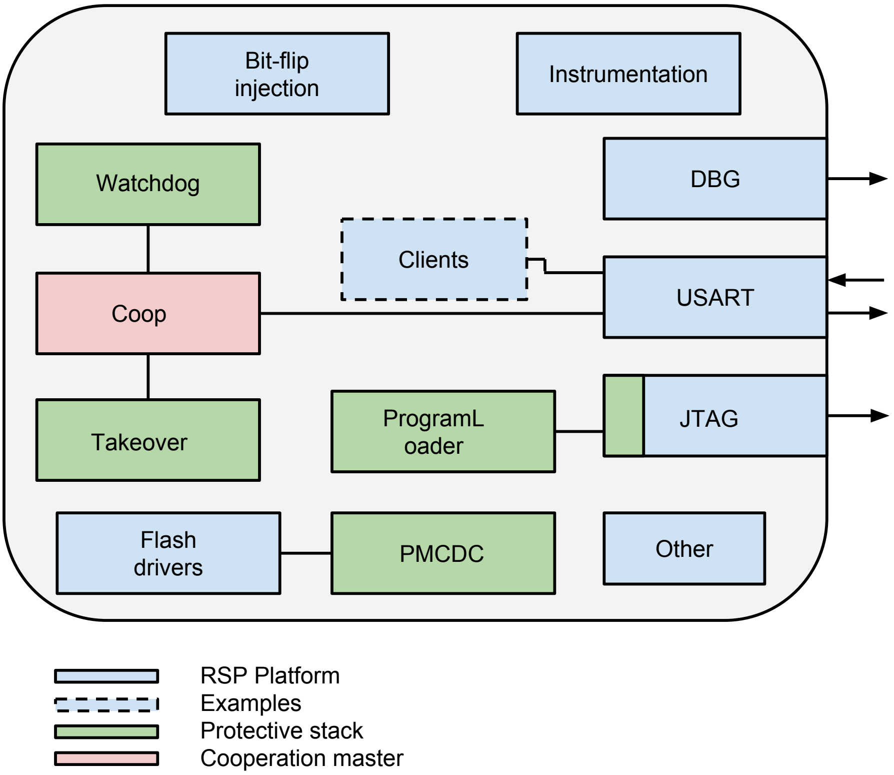

**Figure 4.1:** Overview of RSP.

- a module generating random bit-flips in the internal RAM and flash memory,
- the driver for the internal flash memory and its mock counterpart to mimic the driver of the external fflash memory,
- some instrumentation counters,
- some other tools such as an activity measurement system, or a blinker task used to trace the activity of the MCUs as well as a set of client tasks (reader and writer) of the USART channel provided as examples.

In green are displayed the constructs implementing the fault-tolerant layers of the stack defined in Paragraph 3.3.4 and described in Chapter 5. It includes:

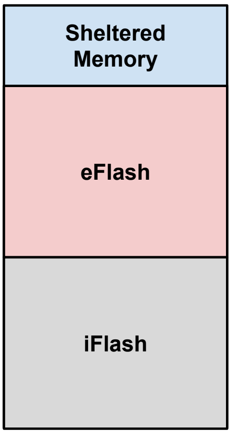

**Figure 4.2:** Overview of the memory map.

- the internal hardware watchdog,
- the PMCDC module,
- the program loader (ProgramLoader) and the front-end of the JTAG module (utilized for reprogramming purpose),
- the takeover mechanism.

In red is represented the algorithm articulating the different fault-tolerant techniques of the stack. It is described in Chapter 6.

## 4.2 Overview of the memory map

As explained in the Paragraph 3.1.3, the external fflash is mapped in the internal flash memory, yielding to a subdivision of the flash memory. The bottom part, called **iFlash**, corresponds to the internal flash memory while the top part, called **eFlash**, contains the external fflash memory content. Figure 4.2, which is a simplified diagram of the memory map, shows a third section on the very top called **sheltered memory**. The purpose of this part will be explained in Paragraph 4.6.1, while the detailed memory map will be unveiled in Section 5.5.

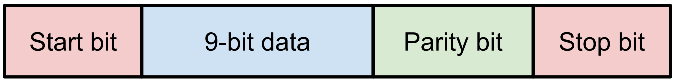

**Figure 4.3:** Diagram of a USART frame.

## 4.3 USART Channel

The purpose of the USART communication channel is to have a bidirectional communication channel between the two MCUs of the system. They can use this facility to exchange data.

### 4.3.1 Frames and Messages

A minimalist communication protocol has been designed in order to organize and send data through the USART channel. The protocol enables some tasks running on any MCU, called **client_writer**, to address flags, such as heartbeats or reprogram requests for instance, or more complex data to other tasks running on the other MCU, called **client_reader**.

#### 4.3.1.1 USART Frames

Figure 4.3 describes the frame corresponding to the data link layer. One frame begins with a start bit followed by a 9-bit data character, one parity bit and ends with a stop bit. The parity bit is able to detect an odd number of errors during the transmission. Currently no action is taken in case a corrupted message is received.

#### 4.3.1.2 Messages

Data can be addressed to a specific task or a specific subset of tasks running on the other MCU. For this to happen, data are encapsulated in the form of **messages**. As shown in Figure 4.4a, a message is made up of 5 elements:

- a *START_FRAME_FLAG* (9 bits)

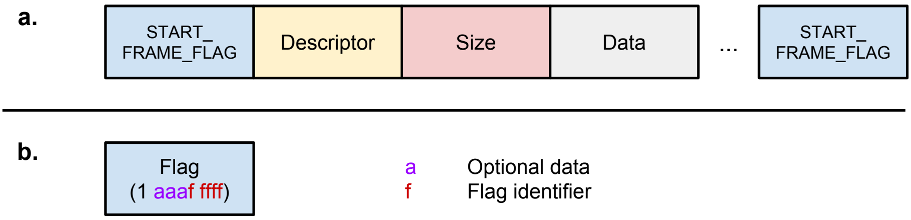

**Figure 4.4:** Diagram of a USART message.

- a descriptor (9 bits)
- a size (9 bits)
- a data field (variable)
- an *END_FRAME_FLAG* (9 bits)

The **descriptor** field is an 8-bit value (the most significant bit being always 0) which acts as a client_reader address. Each client_reader has its own 8-bit value and it receives exclusively the messages whose descriptor field matches its 8-bit address.
The **size** field is an 8-bit value (the most significant bit being always 0) corresponding to the size in bytes of the data field.
The **data field** is the sequence of 8-bit characters to transmit (the most significant bit being always 0).
*START_FRAME_FLAG* and *END_FRAME_FLAG* are 5-bit values (the most significant bit being always 1 and the bits 6, 7 and 8 are don't care values) called **flags**. They respectively mark the beginning and the end of a message.

#### 4.3.1.3 Flags

It is also possible to send flags (other than *START_FRAME_FLAG* and *END_FRAME_FLAG*) that can be seen as messages without data, such as *HEARTBEAT_FLAG* (heartbeat) or *REQ_REPROGRAM_FLAG* (reprogramming request) for instance. Each flag is defined by a 5-bit value encoded using a 2 out of 5 code in order to maximize the

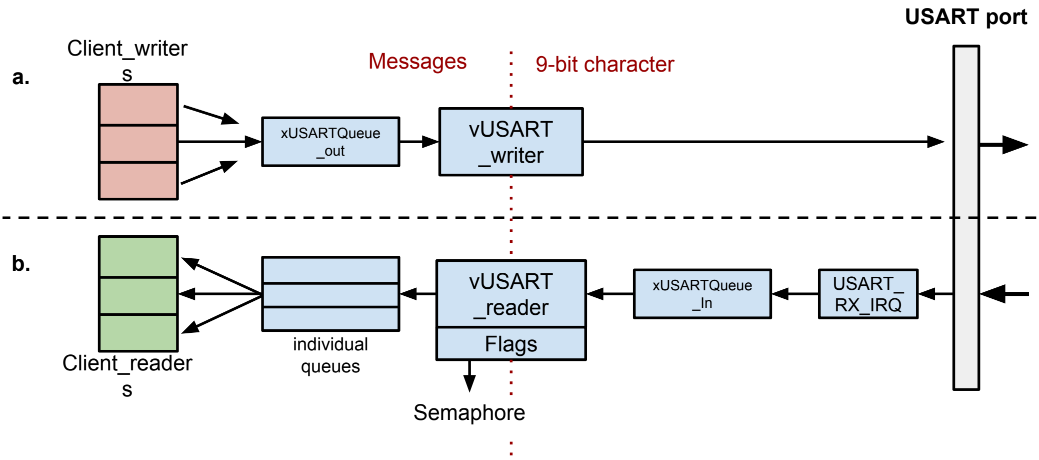

**Figure 4.5:** Functional schematic of the USART channel.

hamming distance. It is however possible to use bits 6, 7 and 8 to transmit some additional information (as it is done with *HEARTBEAT_FLAG* which sends the activity of the MCU). Bit 9 is always 1.
Figure 4.4b shows the structure of a flag.

### 4.3.2 Transmission

The transmission of data over the USART channel suffers from one major constraint: when the RTOS kernel is running, more than one task may want to send some data simultaneously. The solution to this problem is illustrated in Figure 4.5a. Each client_writer task that want to send data transmits a message (cf. below) to a queue called *xUSARTQueue_out*. On the other side of the queue, the task *vUSART_writer* fetches the messages encodes them into a sequence of 9-bit characters that are transmitted over the USART channel.

### 4.3.3 Reception

Figure 4.5b shows the reception mechanism. The reception of a 9-bit character triggers an interrupt serviced by the *ISR USART_RX_IRQ*. This ISR sends the character to a first queue called *xUSARTQueue_in*. On the other side of the queue, a task called *vUSART_reader* gathers those characters and build messages with them. When a message is reconstituted it is sent to a secondary queue to the destination of the addressee of the message called a *client_reader* thanks to the descriptor field (cf. Paragraph 4.3.1.2).

**Note:** The flags (other than *START_FRAME_FLAG* and *END_FRAME_FLAG*) are not transmitted to the secondary queues (since a flag does not encompass a descriptor field) but rather they trigger semaphore activity within *vUSART_reader*.

## 4.4 Debug Communications

RSP contains a utility called **DBG** that enables to transmit messages or data to a computer over an additional USART link. The DBG utility includes functions to print strings, integers, hexadecimal values, binary values, and so on.

There are different cases to consider depending on whether the debug port (USART port used as debug channel) has been configured, or whether the RTOS kernel is running, but a unique and seamless interface makes it simple for the user who just need to use one function to send a message or print data. Each case is illustrated in Figure 4.6.

**Before the Setting-Up of the Port** If we are in the boot section before the setting up of the debug port, we send the messages to be sent to a buffer called *bufferStr*. The whole buffer is then printed by the function *printBootMessages* called once the setting up of the port is complete.

**When the RTOS Kernel Is Not Running** If the RTOS kernel is not running it means that the Program Counter (PC) is in a mono-threaded part of the program and the messages can therefore be sent directly to the debug port without any risk of an interference.

**When the RTOS kernel is running** However, if the RTOS kernel is running, there may be more than one task that want to print a message at the same time. Because the tasks are likely to run in an interleaved manner, both messages would be blended together and produce an incorrect output. Hence, a direct access to the debug port cannot be provided. To solve this problem, any task that wants to print on the debug port adds its message on a queue (*xDBGQueue*). On the other side of this queue, a task (*WriterDBG*) prints the messages in the order they arrive. Since

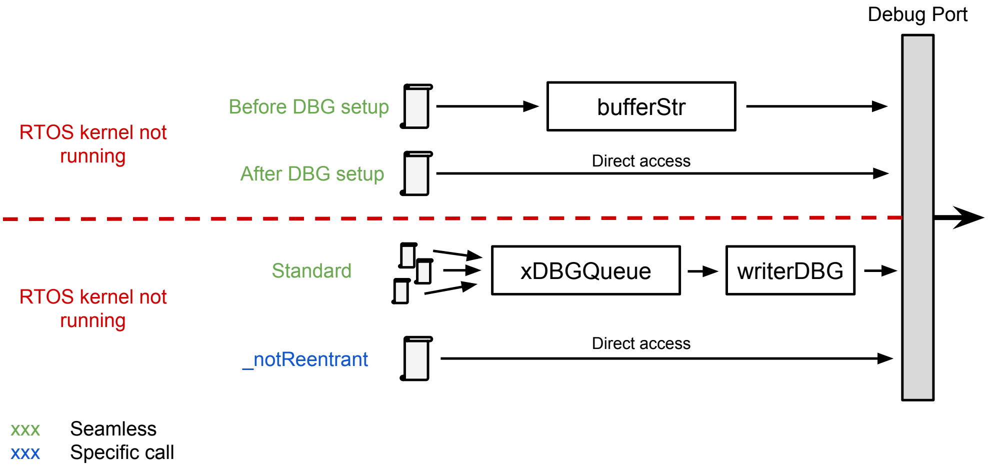

**Figure 4.6:** Functional schematic of the Debug channel.

WriterDBG is the only task have direct access to debug port (cf. exceptions below), no interleaving can perturb the output.

**_notReentrant calls** In some cases, the debug printing function itself produces an error and wants to print a message. We then observe a recursive call to the function yielding to a deadlock of the debug facility. For that reason, each debug printing function has a counterpart (wearing the same name with suffix *_notReentrant*) that writes directly to the debug port. The implementation of a debug printing function should only use *_notReentrant* functions to report an error. We also used *_notReentrant* functions within ISRs even though it was not compulsory.

## 4.5 JTAG Controller

The platform RSP also comprises a JTAG controller which can be used, among others, to reprogram an other device. A very complete description of the JTAG protocol is provided in [Atm12].

### 4.5.1 JTAG Signals

The controller controls four GPIO pins corresponding to the signals:

- Test Clock (TCK): clock shifting the JTAG registers,
- Test Mode Select (TMS): enabling the navigation in the Test Access Port (TAP) controller (a state machine),
- Test Data In (TDI): data sent inside the JTAG registers,
- Test Data Out (TDO): data received out of the JTAG registers.

The three signals TCK, TMS an TDI have to be generated by the programming device and sent to the JTAG pins of the other device. TDO is generated by the programmed device and send back to the programming device. TMS, TDI and TDO are sampled on the rising edge of the clock TCK. There is no precise specification regarding the frequency or the periodicity of the clock in the standard.

### 4.5.2 Design of the Controller

A schematic of the JTAG controller, also called **back-end** is provided in Figure 4.7. Each JTAG instruction is parsed leading to a sequence of states (low or high) for TMS and TDI. Those states are encoded as characters and recorded into a circular buffer called *jtagBuffer*. Each item of this buffer represents the state of TMS and TDI over one clock cycle. *vJtag_task* goes to sleep until the buffer is empty (i.e. all the signal states encoded in the buffer have been transmitted to the target device). Once the instruction completes, the task is awakened and moves on the the next instruction.

In parallel, the buffer is progressively read and its content converted from signal states to signal toggles by the task *vRead_jtagBuffer_task*. More precisely, when awake, the task computes which combinations of signals should be toggled depending on their current state and the state we want them to get on the next clock cycle (which is given by the value encoded by the next item in the buffer).

A timer is used to trigger an interrupt periodically. When the interrupt handler (*jtag_tc_irq*) is called, one or part of the signals TCK, TMS and TDI are toggled according to the result of the previous computation executed by the task *vRead_jtagBuffer_task*, and this very task is awakened to compute the next toggles. TCK is always in the toggle list.

The signal TDO is sampled on the rising edge of TCK and transmitted to *vRead_jtagBuffer_task*

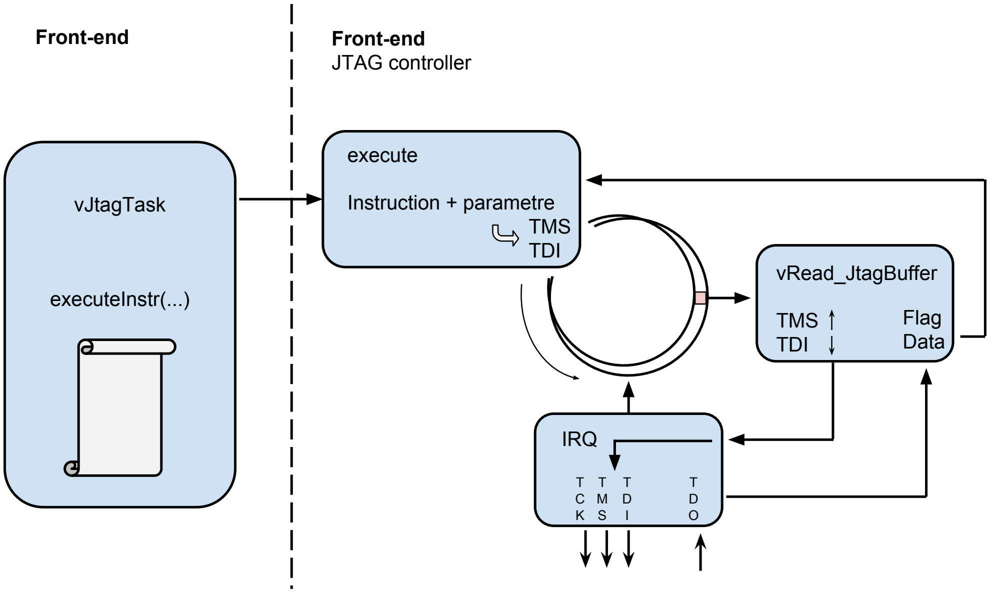

**Figure 4.7:** Schematics of the reprogramming mechanism: front-end and back-end (JTAG Controller).

in order to convert it into data and raise some flags (protected, busy, error) if appropriate.

## 4.6 Tools for Experimentation

RSP platform includes some useful tools for experimentation purpose.

### 4.6.1 Bit-flip Injection

The bit-flip injection facility enables an MCU to randomly flip bits in its own flash and SRAM memories. The flips can be periodic or randomly distributed in time, and the location of the bit-flips can be either randomly spread in the whole memory device, randomly spread in a section of the memory, or precisely defined.

Introducing bit-flip via software is straightforward and very convenient to use contrary to a the use of radioactive sources. It is possible to chose precisely the rate the bit-flips and their location.
However one flaw is that the bit-flip should not affect the bit-flip injection facility itself. This is the purpose of the sheltered memory section in the memory map (cf. Paragraph 4.2) which cannot be affected by the simulated bit-flips and in which are gathered all the experimental facilities such as the bit-flip injection.

By definition the eFlash memory is not altered by bit-flips either since it is a model of fflash memory (rad-hard). So finally only the RAM memory and iFlash memory can suffer from bit-flips inserted by the bit-flip facility.

### 4.6.2 Instrumentation Counters

The sheltered memory has also room for counters (in a section called **sheltered data**) that can be used to instrument the system. Their presence in the flash memory enables them to preserve their content even in case of the failure of the MCU.
The counters can be used to count the number of reboots, of power cycling, of bit-flip introduced and fixed in the program memory, and so on.

### 4.6.3 Activity Measurement

This tool provides a measure of the activity of the MCU. A counter is incremented in the idle task (that is when no task is running) and the value it reaches after a fixed period can be converted into an activity level utilized to infer the activity of the chip and also to tell whether some tasks are starving.
In the current implementation, the activity level is transmitted within the heartbeat flag so that the other MCU is aware of the activity of the chip.

### 4.6.4 Starvation Detection

The platform is also provided with a tool raising a warning flag when the tasks running at the specified level of priority or lower are starving.

Chapter 5

# Implementation (2): Fault-Tolerant Techniques

This chapter focuses on the implementation of the fault tolerant techniques listed in Paragraph 3.3.4. The way they are articulated together to form a robust fault-tolerant system is delayed to Chapter 6.

## 5.1 Program Memory Corruption Detection and Correction (PMCDC)

The second line of defense against failures induced by electromagnetic radiations after the use of internal watchdog is the use of PMCDC. It is one of the largest contribution of the work.

### 5.1.1 Overview of the Method and Issues

PMCDC is a facility that periodically checks the validity of the program memory and fixes it if a bit-flip is detected. PMCDC runs after a reboot (in the boot section) as well.
When PMCDC runs, the MCU computes a series of check-sums of sections of the program memory, called **signatures**, and compares them with pre-computed values stored in memory. If the values match, then the section is valid. However if they do not match, the content of the section has been altered and needs to be reprogrammed from a safe backup image of the initial program.

This scheme seems very simple and efficient, but a real implementation unveils quickly a lot of difficulties:

- We need to have a safe backup image of the program memory which we can utilize to replace corrupted sections of the program memory.
- We need the pre-computed signatures.
- We need to check that the code responsible for the checking has not been corrupted and fix it if it did.
- We need to protect the signatures against data corruption: if one signature is corrupted then, each time PMCDC is triggered, the corresponding section would be considered as altered and replaced even if it is not.
- Because of the way a flash memory works, it is not possible to change only one bit or byte. A whole page of 512 bytes have to be cleared then rewritten. If we mix the drivers to access the flash memory (*iFlashDriver*) and the fflash memory (*eFlashDriver*) with the standard code, and a bit-flip occurs in a function located in the same page as one of the drivers, then during the operation the whole page will be cleared, including the driver being currently used. This would fatally lead to a failure of the MCU.
- We need to find a fast algorithm to compute signatures in order to minimize the overhead of running PMCDC. The implementation should also occupy as little area as possible in memory in order to minimize the risk of a corruption due to radiation-induced bit-flips.

### 5.1.2 Implementation

#### 5.1.2.1 Flash Drivers

In order to operate, PMCDC requires two drivers able to read from the external fflash memory (*eFlashDriver*) and write to the internal flash memory (*iFlashDriver*). Since our system does not contain any external fflash memory, but uses a mapping to the internal flash memory, a mock driver has been implemented to read from it. The only operation needed to reuse the code implemented in this work is to replace this mock driver by the real driver of the fflash memory device, as illustrated in Figure 5.1.

#### 5.1.2.2 Drivers Location

Let us consider the $5^{th}$ issue first. The drivers utilized to read and write from the memory can not be mixed with the standard program source code, for the reason

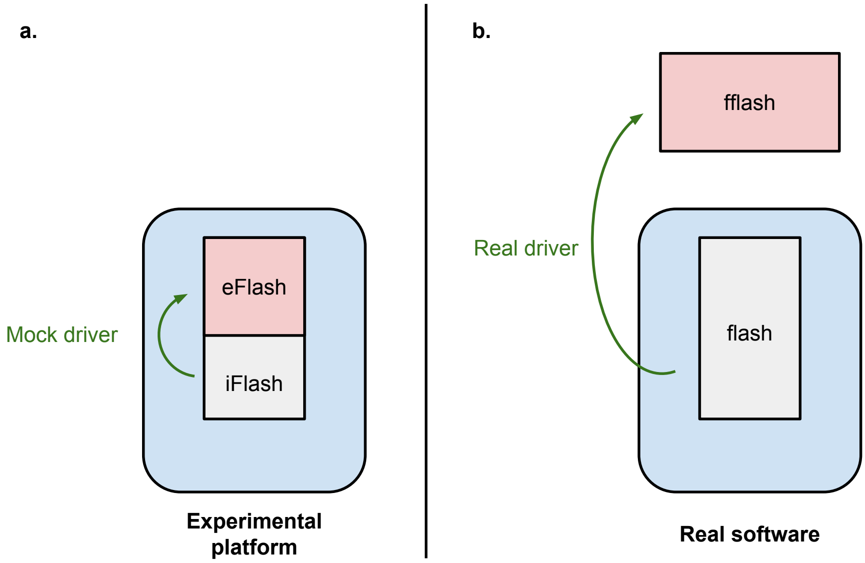

**Figure 5.1:** A real driver replaces the mock one found on the experimental platform RSP.

mentioned above. The solution is to separate them an place them in two distinct locations called **eFlashDriver** and **iFlashDriver**, located far away from any flash page that could potentially be erased, just below the **signature section** which is the part of the memory where all the signatures are stored. The code relocation is performed thanks to the linker script. The updated diagram of the memory map is shown in Figure 5.2.
The consequence of this is that it is possible to detect a corruption of the drivers, but it is not possible to correct it directly.

#### 5.1.2.3 Generation of the Signatures and Image Copy

In order to solve the two first issues, we make a distinction between two cases:

- the MCU has just been programmed by a computer (which means we load a new version of the code into the system),

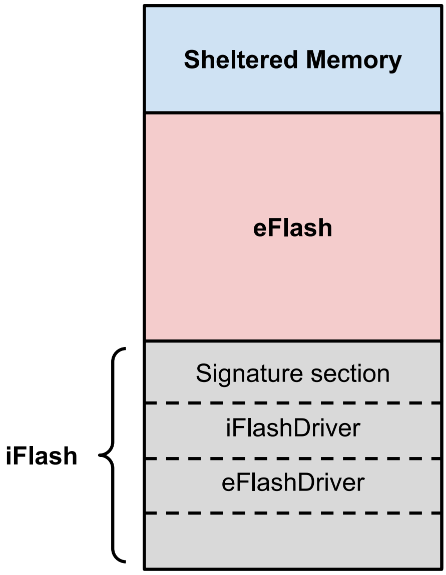

**Figure 5.2:** Updated diagram of the memory map containing the flash drivers.

- the MCU has been restarted due to a power cycle, an external reset or a watchdog timeout.

In the first case, the function *SMCDC_init_procedure* is called. This function computes the signatures of the **standard code** (all the source code except the flash drivers and the code stored in sheltered memory) and the flash drivers, stores them into the signature section, computes and stores the signature of the signatures (called **superSignature**) and copies the standard code and signatures section into eFlash. Hence when we program the MCU for the last time before the launch of the satellite, the signatures are pre-computed and the whole program memory is copied into the resistant fflash memory. Figure 5.3 shows the updated diagram of the memory map. In the second case, we only perform a PMCDC to correct some potential bit-flips.

Figure 5.4 illustrates the whole PMCDC mechanism.

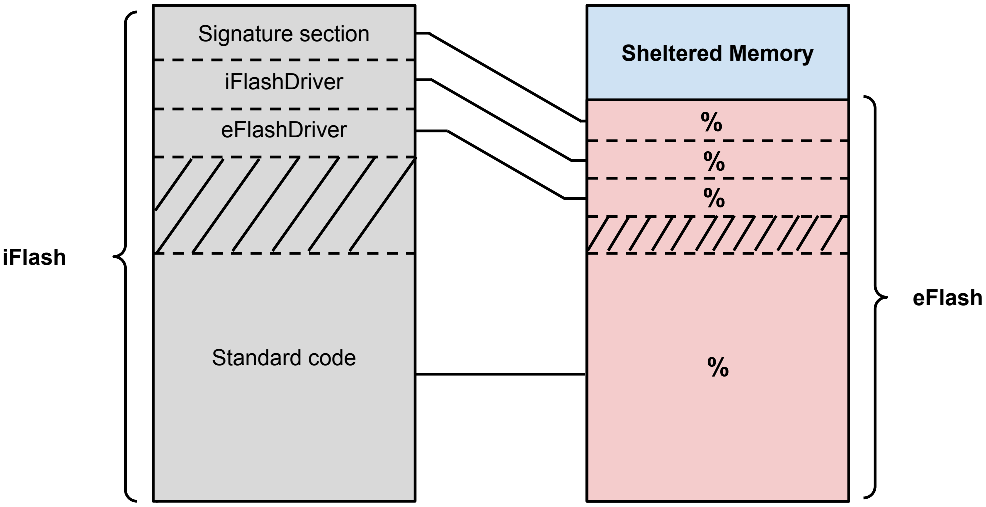

**Figure 5.3:** Updated diagram of the memory map containing the signature section and the copy in eFlash.

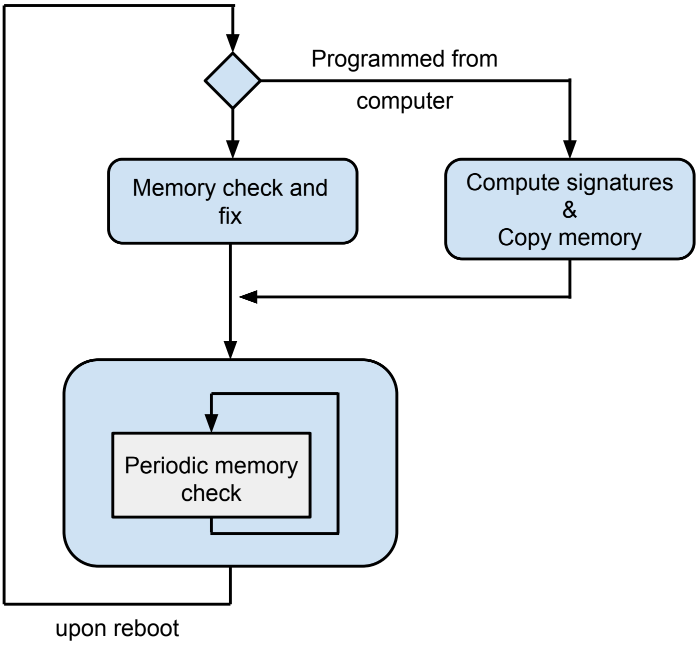

**Figure 5.4:** Flowchart of the PMCDC facility.

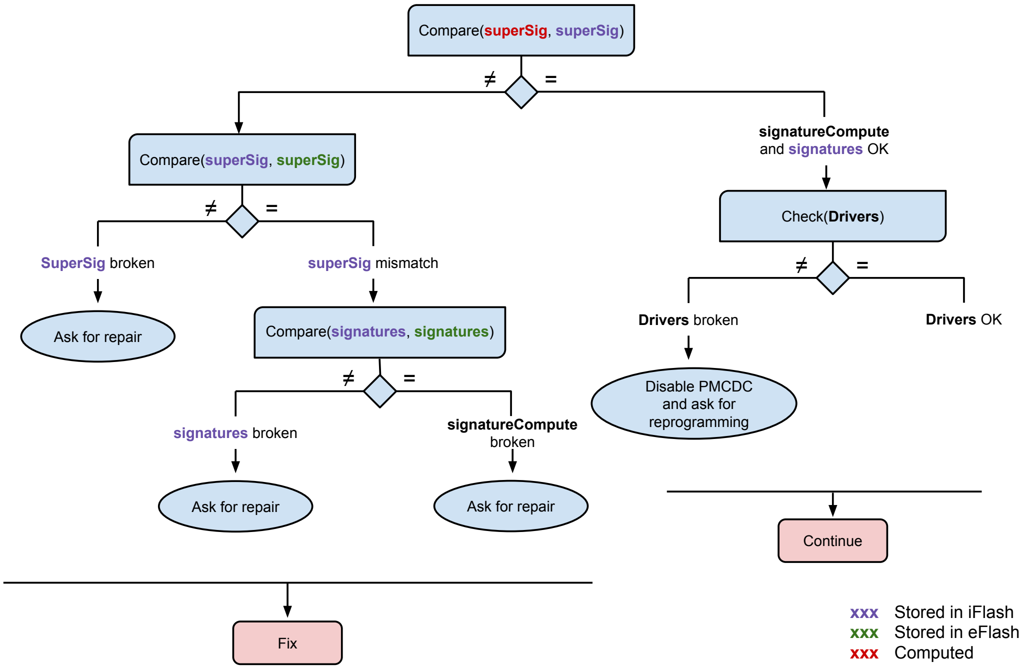

**Figure 5.5:** Self corruption detection and resilience of PMCDC.

#### 5.1.2.4 Choice of the Check-sums Algorithm

The check-sum algorithm employed to compute the signatures efficiently is a Galois LFSR. LFSR are extremely fast and their implementation is extremely small compared to any other check-sum algorithm (such as Cyclic Redundancy Check (CRC)). The Galois flavor has been retained because of its increased speed compared to the XOR LFSRs (since only the last bit needs to be examined). The feedback polynomial has been chosen as a maximum-length LFSR length repeat sequence, which maximizes the discrimination of the 32-bit check-sums.

### 5.1.3 Self Check

The points 3 and 4 of the list above raise the issue of a corruption of the PMCDC facility. This is solved by the algorithm described in this section and illustrated in Figure 5.5.

The algorithm relies on the signature of the signatures named superSignature. First, we compare the computed superSignature with the iFlash superSignature. If they match, it means that all the signatures and the implementation of the algorithm are unaltered and we can use it to check and fix the standard code and the drivers.
Otherwise we compare the iFlash superSignature with the eFlash. If they are different then it means that the iFlash superSignature is corrupted and we ask for its repair.
Otherwise, we compare the whole iFlash signature section with the eFlash one. If they differ, it means that the iFlash signature section is broken and we ask for its repair. Otherwise it means that it is the function *signatureCompute* (in charge of computing the signatures) itself that is corrupted and we ask for its repair.
Provided that both drivers and the function in charge of fixing the memory (*fix_memory*) are not corrupted, the deficient parts of the checking mechanism are fixed and we can proceed to safely to the checking of the whole program memory.
Additionally, we know that the flash drivers can be checked but not fixed by PMCDC. Hence, if we notice a corruption of one or both drivers, the corrupted MCU deactivates PMCDC, and sends a *REQ_REPROGRAM_FLAG* flag, which initiates a reprogramming procedure and eventually the corrupted drivers will get corrected.

## 5.2 Heartbeats and Failure Detection

The detection of the failure of the other MCU is accomplished by using heartbeats. Heartbeats are flags that are periodically sent through the USART communication channel by both MCUs. If the last heartbeat received from the other MCU is beyond a certain time limit, then the MCU is considered to be faulty.

The failure detection is implemented as follows. A task (*vHeartBeat_tap_task*) is in charge of resetting a counter each time a heartbeat issued by the other MCU is received. If the counter reaches its limit value, an interrupt is triggered which indicates that at least one heartbeat have not been issued in time (**heartbeat miss**).

## 5.3 Reprogramming

Reprogramming the other MCU is one of the key contribution of this work. Upon request of the faulty MCU or absence of heartbeats, the other MCU can initiate a reprogramming of the device which will then recover a fault-free image.

### 5.3.1 Indirect Reprogramming

It would not be feasible to use the JTAG controller for restoring the full image of the faulty device for several reasons:

- the whole image of faulty device should be stored in the fflash of the working device which as not been planned: the fflash of one device contains the image of this device, not the other one,
- the transmission rate achieved by the JTAG controller is slow (count about 20 seconds to reprogram 1 KB), albeit it consumes a lot of processing resources. So, sending the whole image would require a lot of energy, power, and the operation would have a lot of impact on the schedule,
- A long transmission time would increase the risk of corruption of the transmitted source code.

For those reasons, instead of reprogramming the faulty device with the whole image, the device is reprogrammed with a small utility program called *ProgramLoader* that fetches by itself the backup image on its own external fflash memory and load it in the internal program memory.
The advantage of this procedure is two-fold:

- *ProgramLoader* is a generic program that can be used without any modification on both devices,
- it is much smaller than the full image.

The source code of *ProgramLoader* is loaded in an empty area between the trampoline and the beginning of the standard code which ensures that it does not overwrite itself when it loads the backup image into the flash memory.

### 5.3.2 ProgramLoader

*ProgramLoader* is optimized to fit in the smallest possible area. Currently it is only 1 Kbit. It contains only two custom flash drivers (*iFlashMicroDriver* and *eFlashMicroDriver*) and a small function that copies the fault-free remote image.

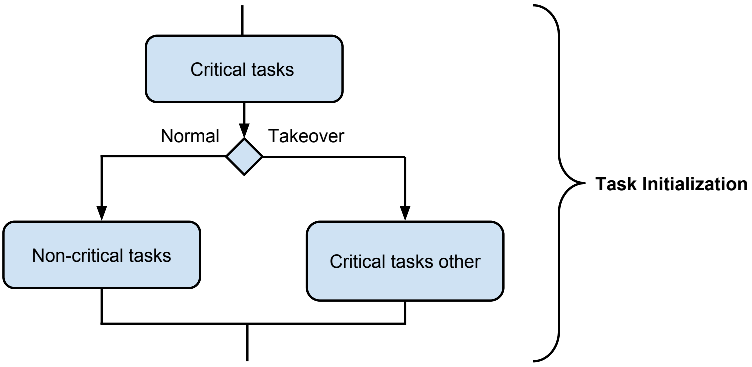

**Figure 5.6:** Schematic of the takeover mechanism.

## 5.4 Takeover

The takeover of one MCU, when the second one seems to be irrecoverable, is the last protection implemented. It consists in running on the working MCU the critical tasks of the out-of-order MCU. In case the RAM memory or processing resources is not large enough to run those new tasks along the normal one, the MCU can run only a subset of his tasks.
As shown in Figure 5.6, in the current implementation, when the MCU decides that the other MCU is irrecoverable, it raises the flag *OTHER_MCU_DOWN* and reboots. When the processor reaches the initialization of the tasks, it initializes its critical tasks. Then it initializes the critical tasks of the other MCU if the flag is raised, or its non-critical tasks otherwise.

If ever the MCU suddenly receives heartbeats from the alleged damaged MCU, it clears the flag and reboots, which has for effect to run the normal set of tasks and no critical tasks of the other MCU.

## 5.5 Detailed Memory Map

The previous discussions lead to a complete description of the memory map that we summarize here.

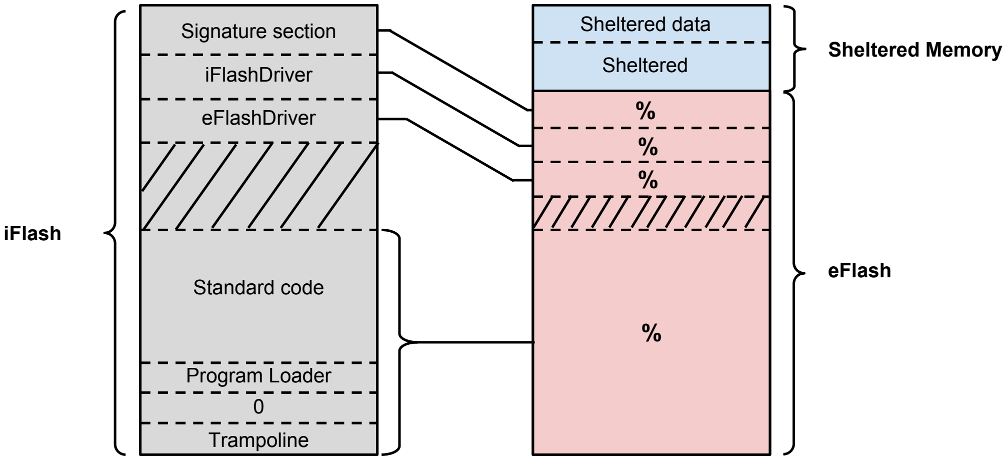

**Figure 5.7:** Full memory map.

As presented in Figure 5.7, the memory is divided in three sections.
The **iFlash memory** contains the source code of the embedded software including:

- the trampoline (couple of instructions making the PC jump to the startup functions),
- a zeroed section (not used),
- a zeroed section that can be overridden by the source code of *ProgramLoader*, which occurs during a reprogramming procedure,
- the standard source code,
- the eFlash and iFlash drivers,
- a section containing some available locations to write useful data and the signatures at the very top.

The **eFlash memory** contains an exact copy of the iFlash memory with for only difference the copy of the drivers and signature section that are shifted down in order to leave some room for the sheltered memory.
The **sheltered memory** contains the source code performing bit-flips and debug facility. Only the iFlash memory is can be altered by bit-flips.

# Chapter 6

# Implementation (3): Articulation of the Fault-Tolerant Techniques

The implementation of the fault-tolerant techniques – PMCDC, reprogramming and takeover – have been presented in the previous chapter. It is now time to discuss the way they are put and articulated together to get a robust bit-flip tolerant system. The first section presents how the mechanisms are integrated in the system, while the second section deals with the cooperative algorithms that articulates them.

## 6.1 Integration of the Fault-tolerant Mechanisms

Now that we have described the implementation of each fault-tolerant mechanism, we explain how they are integrated together in one system.

First let us remind that there are four mechanisms implemented as shown in Figure 6.1. Two of them are local to each MCU:

- internal watchdog (in yellow),
- PMCDC (in blue),

and the remaining two stem from a cooperation between both MCUs:

- reprogramming procedure (in red),
- takeover (in green).

Upon timeout of the internal watchdog (or external reset, power cycling, reprogramming or takeover), the MCU reboots. In the booting sequence, it performs a check

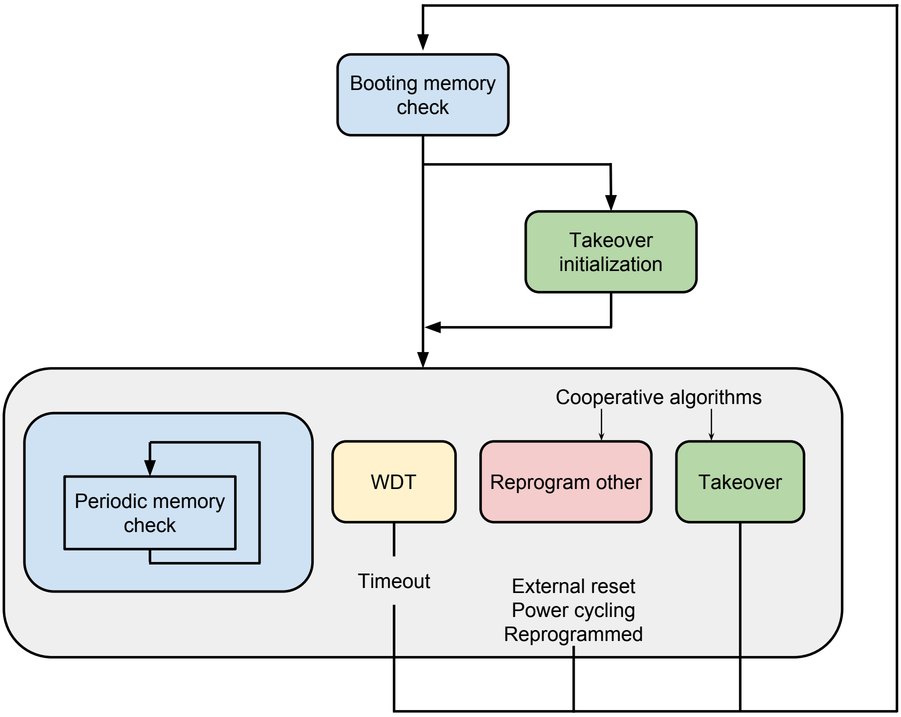

**Figure 6.1:** Schematic of the fault-tolerant mechanisms put together.

of the program memory and fix it if necessary, and if it is in a takeover procedure, it initializes the tasks of the other MCU.

Once the RTOS kernel is running, the MCU performs periodically memory checks and fixes. The cooperative algorithms, topic of the next paragraph, decide when a reprogramming procedure or takeover should be initiated.

## 6.2 Cooperative Algorithms

### 6.2.1 Overview

As detailed in Figure 6.2, the cooperative algorithms analyze the heartbeats and request flags (such as *REQ_REPROGRAM_FLAG*) received (and if needed other

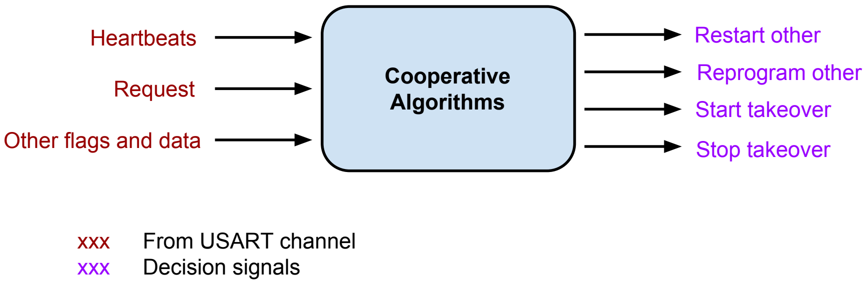

**Figure 6.2:** Role of the cooperative algorithms.

data and flags transmitted over the USART communication channel) in order to determine the state of the other MCU. Actions are taken accordingly among:

- restart the other device,
- perform a reprogramming of the other device,
- initiate a takeover,
- end a takeover.

### 6.2.2 Detail of the Algorithms

As shown in Figure 6.3, there are three concurrent sequences of events that can lead to a restart of the other MCU, reprogramming procedure or takeover:

- MCU_A requests a reprogramming by issuing a *REQ_REPROGRAM_FLAG* flag to MCU_B, then MCU_B reprograms MCU_A.
- MCU_A has rebooted *MAX_NUM_OF_RESTARTS* in less than TIME_BASE _MAX_RESTARTS. We suspect a bit-flip in the boot section or the previous reprogramming failed. Then MCU_B reprograms MCU_A. This is implemented in the function *vResflash_Rst*.
- If MCU_A has not issued a heartbeat since *TIMEOUT_BEFORE_RESTART*, then MCU_B restarts MCU_A via external restart. If after that MCU_B still does not receive any heartbeat for *TIMEOUT_BEFORE_REPROGR*,

then MCU_A is considered as stuck in the boot section and MCU_B reprograms MCU_A. Starting from the moment when the reprogramming is complete, MCU_A has then *TIMEOUT_WAIT_RESTART* to send a heartbeat. If it fails doing so a new reprogramming is performed. At most *NB_MAX_CONSEC_REPROGR* can be formed consecutively. Beyond this limit, MCU_A is considered as definitely out-of-order. The flag *OTHER_MCU_DOWN* is set and MCU_B reboots. With this flag on, the critical tasks of MCU_A are run on MCU_B if it has enough resource. Some non-critical tasks can be dropped on MCU_B in order to free some resource. This is implemented in *vRestart_resflash_HB*. At anytime if a heartbeat is finally received from MCU_A, MCU_B clears the flag and reboots, which brings him back to the normal state.

**Note:** Restarting the other device could be considered as an independent layer in the protective stack. But due to its modest effectiveness (cf. Paragraph 7.3.1) and the fact that it had to be deactivated in the final test (cf. Paragraph 7.3.2) it is considered as optional preliminary part of the reprogramming process.

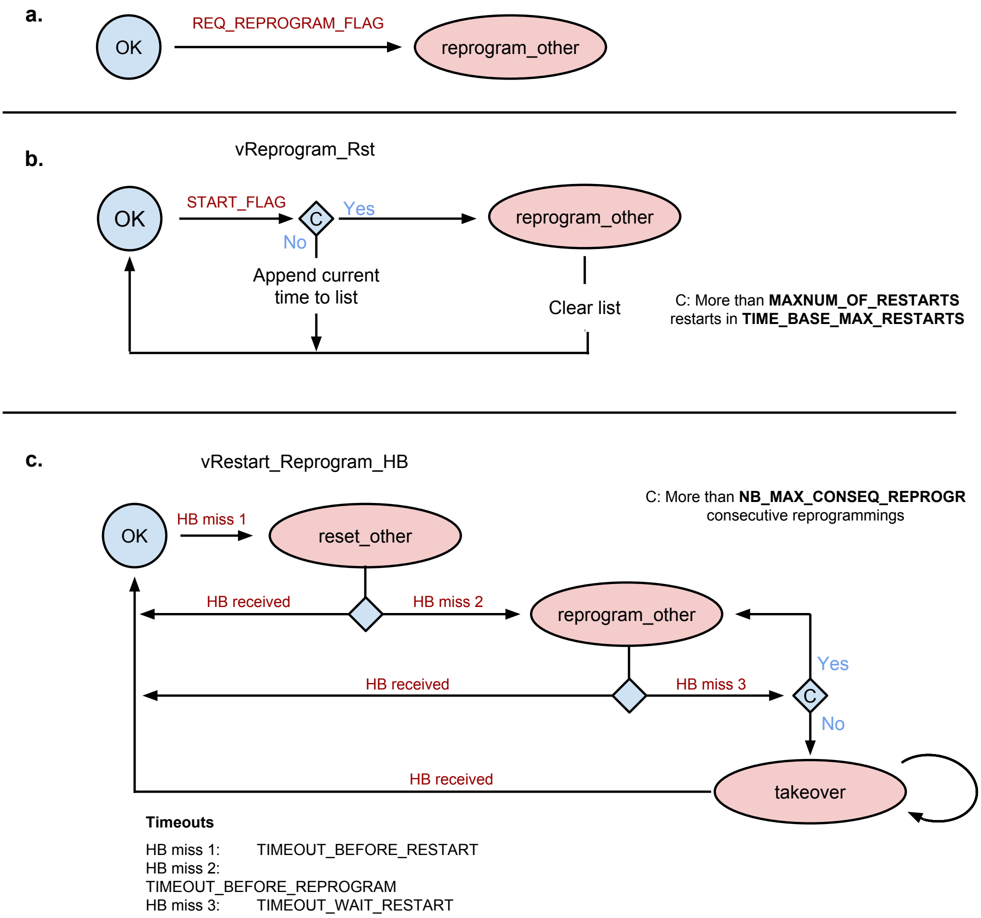
a.
OK
REQ_REPROGRAM_FLAG
reprogram_other
b.
vReprogram_Rst
OK
START_FLAG
C
Yes
No
reprogram_other
Append current time to list
Clear list
C: More than MAXNUM_OF_RESTARTS restarts in TIME_BASE_MAX_RESTARTS
c.
vRestart_Reprogram_HB
C: More than NB_MAX_CONSEQ_REPROGR consecutive reprogrammings
OK
HB miss 1
reset_other
HB received
HB miss 2
reprogram_other
Yes
HB received
HB miss 3
C
No
HB received
takeover
Timeouts
HB miss 1: TIMEOUT_BEFORE_RESTART
HB miss 2: TIMEOUT_BEFORE_REPROGRAM
HB miss 3: TIMEOUT_WAIT_RESTART

**Figure 6.3:** Schematic summarizing the cooperative algorithms.

Chapter 7

# Results

After having detailed the implementation of the whole system in the previous chapters, we provide here the results of experiments aiming at assessing the increasing in reliability brought by the four implemented protection layers against bit-flips in the memory devices.

## 7.1 Preliminaries

In each of the experiments described in the next sections, all or part of the system is tested in the presence of bit-flips in the RAM memory, the program memory, or both. All, or only the protection layers that are to be assessed, are activated. The data are collected from the instrumentation counters (cf. Paragraph 4.6.1).

### 7.1.1 Experimental Setup

#### 7.1.1.1 Compilation

The code has been compiled with no optimization.

#### 7.1.1.2 Useful Area and Independence

We define **useful areas** as areas in the memory device that are effectively used. The bit-flips in the program memory are exclusively injected in the useful areas. However, in the RAM, the bit-flips are sent everywhere in the device and a corrective factor of

0.4 (representative of the ratio of the useful area over total area) is applied to the results.
The use of the useful area produces results that are independent of the size of the code or the amount of RAM used.

#### 7.1.1.3 Bit-flip Periods and Time Scaling

We define two periods. $P_{RAM}$ is the period of the bit-flip insertion in the RAM memory and $P_{PRGM}$ is the period of the bit-flip insertion in the flash memory. Both are defined as

$$P = \frac{c}{SrA},$$

where $r$ is the SEE rate expected for a satellite located on a high latitude orbit (in error·bit$^{-1}$·s$^{-1}$) and $A$ is the area of the memory section to receive bit flips (in bit). $c$ is a corrective factor equal to 1 for $P_{PRGM}$ because the bit-flips are sent to the useful areas exclusively, and $c = 0.4$ for $P_{RAM}$ because the bit-flips are injected in the whole RAM (cf. previous paragraph).
$S$ is a parameter that scales the amount of radiation simulated. If $S = 1$, the systems receives the same amount of bit-flips as if it was on orbit. Otherwise, the system receives $S$ times more bit-flips, as if the time was scaled by a factor $S$.

The effect of this scaling has been tested in the first two experiments in order to check that a time scaling does not affect the result of the experience and it can be employed to accelerate the experiments safely.

**Note:** Since the size of the RAM memory and the size of the used flash memory are different, $P_{RAM}$ is in general different from $P_{PRGM}$.

#### 7.1.1.4 Bit-flip Sequences

The bit-flips are injected randomly in the memories by using a pseudo-random sequence generator. Repeating twice the same experiment with the same pseudo-random sequence will lead to the same result. When the device reboots or is power cycled, it does not restart the same pseudo-random sequence but continues where it stopped. Only reprogramming the device by using a computer can restart the sequence form its beginning.
Unless a sequence number is specified for each measure, the data collection of an experiment is done using a unique sequence restarted from its beginning for each new measure.

#### 7.1.1.5 Loading

During the experiments the MCUs execute the tasks of the RSP platform that are needed to run the protection layers activated in the test. Additionally, the USART clients – reader and writer – provided as examples are run as application, and the blinker task is used to keep track of the state of the MCUs.

**Note:** Both MCUs do not necessarily execute the same code.

### 7.1.2 Definitions and Notations

#### 7.1.2.1 Definition of Failure

Some of the following experiments measure a number of bit-flips before failure. It is necessary to define what we mean by failure in this context.

Let us remind that since the exception vector has been left empty (cf. Paragraph 3.3.4), when an exception occurs, the processor jumps to an infinite empty loop. This is what we call a failure.
When a failure occurs, the internal watchdog (used in all the experiments) timeouts and reboot the system. Hence the number of watchdog timeouts $N_{wdt}$ indicates the number of failures the MCU suffered during the experiment.

#### 7.1.2.2 Notations

We define in Table 7.1 a list of useful notations used in the remaining of the chapter.

### 7.1.3 Caution About the Results

None of the results of the experiments described in this chapter should be used to infer an expected increasing in the lifetime of the satellite (which you will not find in this report) for the reasons listed below.

Firstly, the causes of failure of NUTS are plural and a failure of an MCU may not be the first catastrophic failure putting an end to the mission.
Then, are studied in this report only the effect of a bit-flips in memory devices. But it does not cover the SEEs happening in the logic of the electronic circuitry.

**Table 7.1:** List of some useful notations.

| | |
|---|---|
| $S$ | Scale factor for the bit-flip injection frequency. |
| $P_{RAM}$ | Period of the bit-flip injection in the RAM. |
| $P_{PRGM}$ | Period of the bit-flip injection in the program memory. |
| $P_{pmcdc}$ | Period of the memory periodic checks performed by the PMCDC facility. |
| $N_{wdt}$ | Number of watchdog timeout occurred. |
| $N_{reb}$ | Number of times the device rebooted voluntarily. |
| $N_{erst}$ | Number of external resets occurred. |
| $N_{fix}$ | Number of memory sections were fixed. |
| $N_{reprogr}$ | Number of times the device was reprogrammed. |
| $N_{erst}^{other}$ | Number of times the device reseted the other MCU. |
| $N_{reprogr}^{other}$ | Number of times the device initiated a reprogramming procedure of the other MCU. |
| $N_{bf}^{RAM}$ | Number of bit-flips injected in the RAM. |
| $N_{bf}^{PRGM}$ | Number of bit-flips injected in the program memory. |
| IF | Irrecoverable Failure. |

Finally, the consequences of bit-flips are very dependent on the software that is being executed. Indeed, as seen in Paragraph 3.2.2.1, the celerity at which a fault results in an error is dependent on some parameters of the program such as the number of tasks and their frequency.

For those reasons, the results are employed to assess qualitatively the performance of the solution proposed and they do not replace a proper and indispensable test session with the final software.

## 7.2 Individual Tests

The first set of experiments aims at testing and evaluating the four protection layers selected in the Paragraph 3.3.4.

### 7.2.1 Watchdog Tests

The purpose of the experiment is to study the behavior of one MCU which sustains bit-flips in its RAM first and then in its program memory.

#### 7.2.1.1 Bit-flips in the RAM

In this experiment, one MCU, with only the internal watchdog for protection, runs with bit-flips injected in its RAM with different periods until at least 500 bit-flips have been injected. The number of watchdog timeouts $N_{wdt}$ is measured for each period. The result of the experiment is provided in Table 7.2.

**Table 7.2:** Result of the experiment with bit-flips in the RAM and varying period.

| **Scale** $S$ | $P_{RAM}$ **(s)** | $N_{bf}^{RAM}$ | $N_{wdt}$ | $\frac{N_{bf}^{RAM}}{N_{wdt}}$ | **Final state** |
|---|---|---|---|---|---|
| **2000** | 8.2 | 200 | 4 | 50 | Working |
| **4000** | 4.1 | 202 | 5 | 40 | Working |
| **8000** | 2.1 | 203 | 5 | 40 | Working |
| **15000** | 1.1 | 203 | 6 | 34 | Working |
| **30000** | 0.54 | 208 | 6 | 35 | Working |
| **60000** | 0.33 | 201 | 5 | 40 | Working |

We can see that albeit the variance in the ratio $\frac{N_{bf}^{RAM}}{N_{wdt}}$ is important, there is no obvious correlation between the number of watchdog timeouts $N_{wdt}$ and the period of the bit-flip injection in the RAM $P_{RAM}$. In average, a failure occurs every 40 bit-flips regardless of the bit-flip frequency.

In the previous experiment, each measurement was using the same pseudo-random sequence of bit-flip so that the only varying parameter was the bit-flip injection period. In a new experiment, the scale is fixed to $S = 30000$ (which is tantamount to $P_{RAM} = 0.54$s) and each measurement uses a different pseudo-random sequence. The result is shown in Table 7.3.

The experiment shows an independence of the average number of bit-flip per watchdog timeout $\frac{N_{bf}^{RAM}}{N_{wdt}}$ with the sequence of bit-flips injected, which is still about 40 on average.

**Table 7.3:** Result of the experiment with bit-flips in the RAM and varying sequence.

| **Sequence** | $N_{bf}^{RAM}$ | $N_{wdt}$ | $\frac{N_{bf}^{RAM}}{N_{wdt}}$ | **Final state** |
|---|---|---|---|---|
| **1** | 208 | 6 | 35 | Working |
| **2** | 201 | 4 | 50 | Working |
| **3** | 212 | 7 | 30 | Working |
| **4** | 206 | 4 | 52 | Working |

We conclude that the number of watchdog timeout per bit-flip injected in the RAM is on average independent of the pseudo-random sequence and the frequency of the bit-flip injection. In addition it confirms the assumption that the watchdog is able to restore the MCU in a working state by clearing all the the faults injected in RAM.

#### 7.2.1.2 Bit-flips in the Program Memory

In this experiment, one MCU, with only the internal watchdog for protection, runs with bit-flips injected in its program memory with three different periods and for fifteen different pseudo-random sequences. The measure of the number of bit-flips injected $N_{bf}^{PRGM}$ is taken after the first failure. The result of the experiment is provided in Table 7.4.

**Table 7.4:** Result of the experiment with bit-flips in the program memory.

| **Scale** | | **Sequence** | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $S$ | $P_{PRGM}$ **(s)** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
| **1000** | 21 | 4 | 4 | 6 | 7 | 1 | 7 | 1 | 1 | 7 | 15 | 4 | 10 | 2 | 4 | 7 |
| **2000** | 11 | 4 | 4 | 6 | 7 | 1 | 7 | 1 | 1 | 7 | 15 | 4 | 10 | 2 | 4 | 7 |
| **8000** | 2.6 | 4 | 4 | 6 | 7 | 1 | 7 | 1 | 1 | 7 | 15 | 4 | 10 | 2 | 4 | 7 |

As predicted in Paragraph 3.2.2.1, the number of bit-flips injected before failure is independent of the frequency of the injection, but varies quite a lot depending on the sequence. On average 5.3 bit-flips are introduced before failure.

### 7.2.2 PMCDC Tests

This set of experiments aims at evaluating the performance of the PMCDC facility, which checks and corrects the program memory at run-time.

#### 7.2.2.1 Experiment With Fixed Bit-flip Injection Period

One MCU, protected by the internal watchdog and the PMCDC facility, runs with bit-flips injected in its program memory. For each pseudo-random sequence are measured the number of bit-flips injected in the program memory $N_{bf}^{PRGM}$, the number of memory section fixed $N_{fix}$, the number of watchdog timeouts $N_{wdt}$ and the information whether the PMCDC has been deactivated because of a corruption of the flash drivers occurred, and if so, for which bit-flip.
One measure stops when the MCU seems not being able to recover a stable state (typically when stuck more than ten seconds in the booting sequence). We then say that the MCU suffers from an Irrecoverable Failure (IF).

Two sets of measures are provided:

- on the left (columns $A$) is a measure with bit-flips injected everywhere in the source code including the flash drivers,
- on the right (columns $B$) is a measure with bit-flips injected everywhere in the source code excluding the flash drivers.

The second set of measures enables to evaluate the pure performance of PMCDC when it is not deactivated prematurely because of the corruption of one of the flash drivers (which cannot been fixed if altered). Note that even without corruption of the drivers, PMCDC can still be deactivated in some other cases.

The period of PMCDC $P_{pmcdc}$ is s, the scale is $S = 2000$, which corresponds to a bit-flip injection period $P_{PRGM}$ of 10s.

The results are provided in Table 7.5.

If we analyze the results, we find that on average 9 bit-flips can be injected before irrecoverable failure for experiment $A$ (non-selective bit-flip injection), while the value rises up to 24 bit-flips for experiment $B$ (no bit-flip in the flash drivers). It represents an improvement of 64% and 350% respectively compared to the previous experiment (without PMCDC).

**Table 7.5:** Results of the experiment with PMCDC with fixed injection period.

| | $N_{bf}^{PRGM}$ **before IF** | | $N_{fix}$ | | $N_{wdt}$ | | **PMCDC disabled** | |
|---|---|---|---|---|---|---|---|---|
| **Sequence** | $A$ | $B$ | $A$ | $B$ | $A$ | $B$ | $A$ | $B$ |
| **1** | 18 | 72 | 0 | 59 | 0 | 4 | 1 | N/A |
| **2** | 20 | 7 | 11 | 4 | 3 | 2 | 14 | N/A |
| **3** | 6 | 6 | 3 | 3 | 0 | 0 | 2 | 2 |
| **4** | 7 | 7 | 6 | 6 | 2 | 3 | 7 | 7 |
| **5** | 11 | 68 | 0 | 56 | 0 | 8 | 1 | N/A |
| **6** | 5 | 5 | 4 | 4 | 1 | 1 | 5 | 5 |
| **7** | 4 | 4 | 3 | 3 | 0 | 0 | 4 | 4 |
| **8** | 2 | 23 | 0 | 16 | 0 | 5 | 1 | N/A |
| **9** | 13 | 8 | 6 | 6 | 1 | 1 | 7 | N/A |
| **10** | 4 | 63 | 0 | 51 | 0 | 8 | 1 | N/A |
| **11** | 11 | 6 | 4 | 4 | 2 | 2 | 5 | N/A |
| **12** | 3 | 45 | 1 | 38 | 3 | 8 | 2 | N/A |
| **13** | 9 | 4 | 2 | 2 | 1 | 1 | 3 | N/A |
| **14** | 10 | 41 | 7 | 35 | 2 | 9 | 8 | N/A |
| **15** | 7 | 2 | 0 | 0 | 0 | 0 | 1 | N/A |

The difference can be explained by the fact that in most cases PMCDC is not deactivated during the measurement, which means that it continues to fix potentially harmful faults at runtime before they become active. On the contrary, in experiment $B$, PMCDC is deactivated after on average 4 bit-flips. Once deactivated, the system is equivalent to the one tested in the previous experiment: only the internal watchdog as protection.

In addition, we note that 37% of the bit-flips are corrected in experiment $A$ against 63% for experiment $B$. This is not quite exact since two bit-flips in the same memory section will count for only one section fixed. So $N_{fix}$ is actually an underestimation of the number of faults actually corrected.

There are two cases where the system with PMCDC had a smaller lifetime than the one without PMCDC. In all other cases, the reliability of the system with PMCDC is improved up to 15 times.

Finally, let us highlight that it is the couple {watchdog, PMCDC} that enables to get a reliability improvement. Indeed, even with PMCDC fixing 63% of the faults, one bit-flip out of six results in a failure before PMCDC have time to correct it. The use of the watchdog is then necessary to recover from these failures and fix it during the reboot sequence.

**Note:** The irrecoverable failures occur when the booting sequence is corrupted and the MCU reboots.

The conclusion of this section is that the tandem watchdog and PMCDC improves with no doubts the tolerance against corruption of the program memory. The important difference in performance observed between the system without bit-flips in the flash drivers and the system with corruption of the flash drivers indicates that a lot of improvement could be achieved by reducing drastically the size of both flash drivers.

#### 7.2.2.2 Experiment With Variable Bit-flip Injection Periods

A similar experiment is now conducted but with a fixed sequence and variable bit-flip injection periods. The results are given in Table 7.6.

**Table 7.6:** Results of the experiment with PMCDC with variable injection periods.

| **Scale** $S$ | $P_{PRGM}$ **(s)** | $N_{bf}^{PRGM}$ | $N_{fix}$ | $N_{wdt}$ |
|---|---|---|---|---|
| 500 | 40 | 70 | 62 | 4 |
| 1000 | 20 | 68 | 60 | 4 |
| 2000 | 10 | 72 | 59 | 4 |
| 4000 | 5.1 | 55 | 45 | 4 |
| 8000 | 2.5 | 55 | 44 | 4 |
| 12000 | 1.7 | 27 | 18 | 0 |
| 16000 | 1.6 | 12 | 7 | 2 |

We can see two trends in these data:

- when the frequency of the bit-flip injection is smaller or similar than the frequency of the periodic checks performed by PMCDC: the behavior of the system is independent of the scaling,
- when the frequency of the bit-flips is higher than the frequency of the periodic checks performed by PMCDC: the reliability decreases drastically with the increase of the scale.

This can be easily explained by the fact that in the second case, PMCDC is overwhelmed by the number of bit-flips, and a failure occurs before the system had time to correct any bit-flip.

The conclusion is that the period of the periodic checks performed by PMCDC $P_{pmcdc}$ should be at least of the same order of magnitude as the expected period of the bit-flips.

### 7.2.3 Reprogramming Test

This experiment attempts to validate the reprogramming protection layer which consists in reprogramming an unresponsive MCU with a working one.

One MCU, called MCU_A, runs with bit-flips injected to its program memory and all protection layers deactivated. In parallel, the second MCU, called MCU_B, runs without any bit-flip injection and with the reprogramming facility activated.
The settings of the reprogramming device are:

- wait 10s before resetting MCU_A,
- wait 10s before reprogramming MCU_A,
- wait 12s before reprogramming anew MCU_A,
- maximum 5 restarts per period of 10 minutes.

We observe that MCU_B is able to detect an irrecoverable failure of MCU_A and instantiates a reprogramming of the latter. Once the operation completes, MCU_A reboots and is able to run its code correctly until the next irrecoverable failure. MCU_B detects the failure and instantiates anew a reprogramming operation, and so on.
Additionally, when the power cycle MCU_A 5 times in a raw, which means that MCU_B sees 5 restarts, the latter initiates a reprogramming procedure on MCU_A.

The reprogramming facility works as defined in Section 6.2.2.

### 7.2.4 Takeover Test

In this experiment we check the proper working order of the takeover protection layer which consists in running the critical tasks of an out-of-order MCU (MCU_A) on a working MCU (MCU_B).

In this test, MCU_A and MCU_B run without any bit-flip injection. In addition, MCU_B has its reprogramming and takeover facility activated. The USART channel is disconnected. MCU_A runs a blinker task $T_A^c$ that makes one of the LEDs found on the XPlained board blink at a frequency $f_A$ while MCU_B runs a blinker task $T_B^{nc}$ at a frequency $f_B$. The blinker task $T_A^c$ is considered as critical for the system, contrary to the blinker task $T_B^{nc}$ that is non-critical.

We observe that initially MCU_A and MCU_B have a LED blinking at a respective frequency $f_A$ and $f_B$. Then, because USART channel is disconnected, MCU_B cannot receive the heartbeats sent by MCU_A and is convinced that the latter failed. It attempts to performs 5 consecutive reprogramming of MCU_A. After the fifth unsuccessful reprogramming it reboots and the LED blinks at the frequency $f_A$ instead of $f_B$. It shows that the takeover has taken place:

- the critical task of MCU_A (here $T_A^c$) is executed on MCU_B,
- the non-critical task of MCU_A (here $T_B^{nc}$) is not executed.

While in master mode, no further attempt the reprogram MCU_B is carried out.

If we connect the USART channel, then MCU_B, receiving heartbeats from MCU_A, considers the latter as anew part of the system and switches back to normal mode. After reboot, its LED blinks at the frequency $f_B$ indicating that it executes only its tasks and no task from MCU_A.

The takeover facility works as defined in Section 6.2.2.

## 7.3 Integral Tests

The last two experiments test the integration of the whole protective stack altogether. The first test is unilateral meaning that only one MCU sustains bit-flip injection, while the second one does not. The last experiment is bilateral: both MCUs run with bit-flip injection.

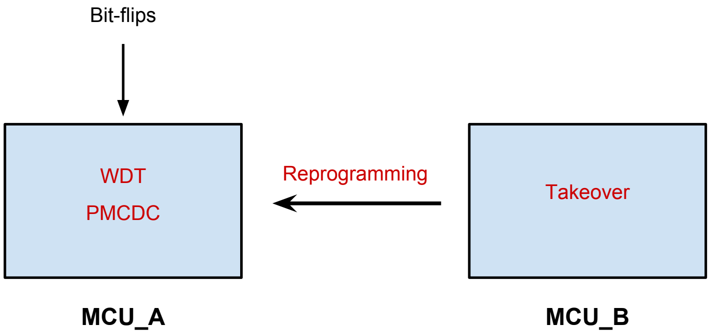

**Figure 7.1:** Schematic of the unilateral test.

### 7.3.1 Unilateral Test

In the unilateral test, only MCU_A sustains bit-flip injection in the RAM and program memory, while MCU_B runs free of any fault injection. The purpose of this experiment is to study how well the protective stack performs as long as one MCU is working properly.

As shown in Figure 7.1, MCU_A is protected by its internal watchdog, PMCDC facility and can be reprogrammed by MCU_B. MCU_B can take the full control of the system in case MCU_A seems unrecoverable.

A photography of the experimental setup is shown Figure 7.2. The MCU on the left is MCU_A, while the on the right is MCU_B. The grey set of wires linking the top left GPIOs of the MCUs is the USART communication channel. The JTAG link starts on the top right GPIOs of MCU_B and ends in the JTAG pins of MCU_A located on the left of the power supply.

The period of the periodic checks performed by PMCDC $P_{pmcdc}$ is 4s. The scale is $S = 4000$, which implies a bit-flip injection period of 10.3s in the RAM, and 5.1s in the program memory. The parameters of the cooperative algorithms are the same as the one used in the PMCDC test (cf. Paragraph 7.2.2). The measurements were stopped after 240 minutes.
Tables 7.7 and 7.8 provide the data gathered from MCU_A and MCU_B respectively.

The figures from MCU_A show that at least 73% of the bit-flips in the program memory have been fixed at runtime meaning that PMCDC enabled to divide by 4 the apparent number of bit-flips. 150 reboots (due to watchdog timeouts or external

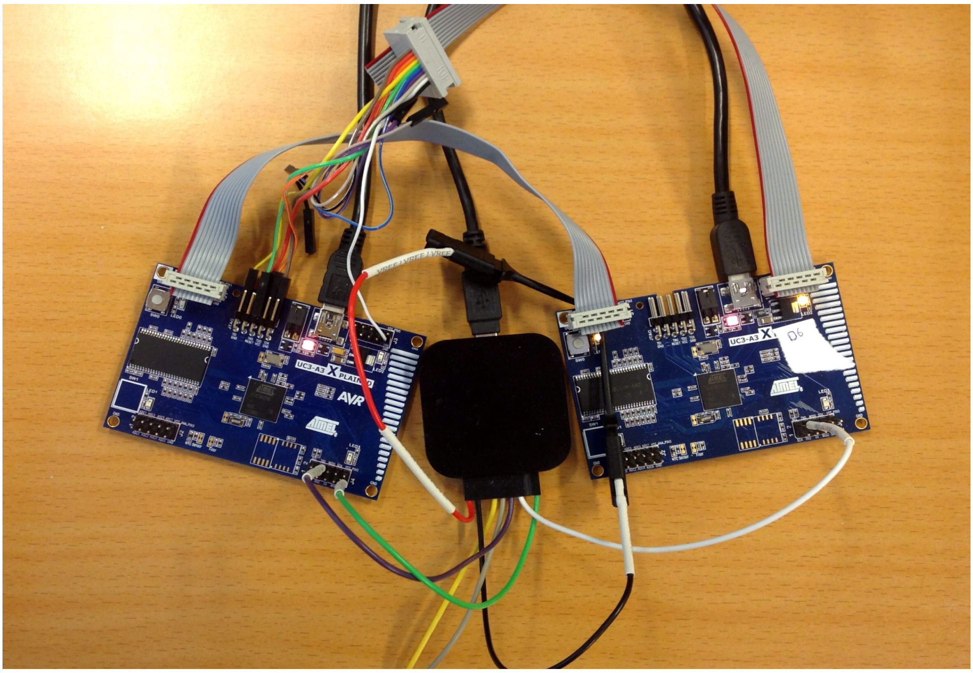

**Figure 7.2:** Photography of experimental setup for the unilateral test.

**Table 7.7:** Data gathered from MCU_A after the unilateral experiment.

| $N_{bf}^{PRGM}$ | $N_{bf}^{RAM}$ | $N_{fix}$ | $N_{wdt}$ | $N_{erst}$ | $N_{reprogr}$ | **Final state** |
|---|---|---|---|---|---|---|
| 1159 | 289 | 843 | 140 | 10 | 165 | Working |

**Table 7.8:** Data gathered from MCU_B after the unilateral experiment.

| $N_{erst}^{other}$ | $N_{reprogr}^{other}$ | $N_{reb}$ | **Takeover** |
|---|---|---|---|
| 90 | 181 | 13 | 0 |

resets) has successfully restored the MCU in a stable state, in addition to 165 reprogrammings, which means 1 reprogramming every 7 bit-flips in the program memory. The comparison between the number of time MCU_A claims having been reprogrammed $N_{erst}$ and the number of time MCU_B has effectively attempted to reprogram MCU_A $N_{reprogr}^{other}$, shows a difference of 16, meaning that 9% of the reprogramming attempts failed. Among those 16 failed attempts, 13 can be attributed to a failure of the JTAG controller (as testify the 13 self restarts).
Finally, let us note that 11% of the external restarts let MCU_A execute some code

until the boot section, but it is not even sure whether managed to complete the booting sequence. So we can conclude that the use of the external reset seems useless in this test.

The conclusion of this experiment is that the system was able to contend with bit-flip injection in the RAM and the program memory with a frequency 4000 higher than the one it will face on orbit. The system ran successfully for 4 hours in this intense simulated radiated environment and could have run longer. As long as one MCU is working properly the system has no reason to stop working (if we consider only faults in the memory devices).
However, some work could be done on the JTAG controller to reduce the amount of reprogramming failures.

### 7.3.2 Bilateral Test

The final experiment is the bilateral test where both MCU sustain bit-flip injection in the RAM and program memory. On both devices all the protective layers are activated with the exception of the reset of the other MCU (part of the reprogramming facility). It had to be deactivated because the reset of one MCU was triggering the reset of the the other one. Given the modest advantages provided by the use of the reset seen in the previous experiment (cf. Paragraph 7.3.1), it has simply been deactivated in this test.

A schematic representation of the experimental setup is provided Figure 7.3, and a photography is shown Figure 7.4. The grey wires are the USART communication channel, while the two sets of colored wires are the two JTAG cross connections.

The period of the bit-flip injection in the program memory has been chosen so that it would take longer than one standard reprogramming process. The scale was $S = 250$, which implies a bit-flip injection period of 165s in the RAM, and 59s in the program memory. The period between two bit-flip injections in the program memory was actually randomly distributed between in the interval $[0.5, 1.5] \cdot P_{bf}^{PRGM}$ with an average of $P_{bf}^{PRGM}$. The period of the periodic checks performed by PMCDC was $P_{pmcdc} = 15$s. Two different pseudo-random sequences has been used on both devices. The parameters of the cooperative algorithms are the same as the one used in the PMCDC test (cf. Paragraph 7.2.2). The measurements were stopped after 480 minutes (8 hours).

Table 7.9 presents the data collected from both MCUs at the end of the experiment.

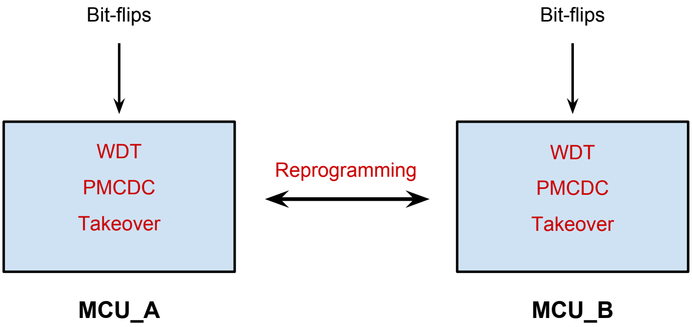

**Figure 7.3:** Schematic of the bilateral test.

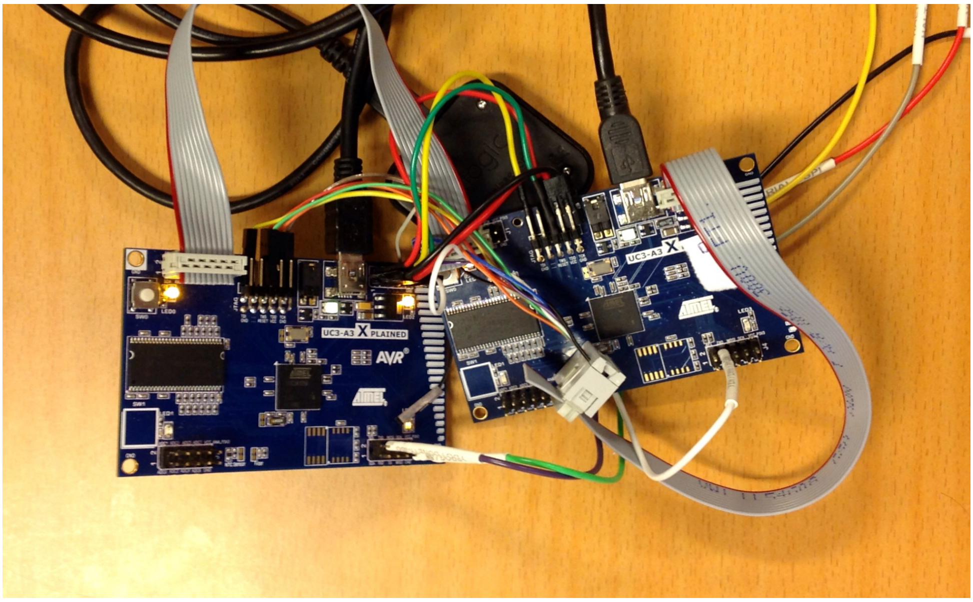

**Figure 7.4:** Photography of experimental setup for the bilateral test.

**Table 7.9:** Data gathered from both MCUs after the bilateral experiment.

| **Device** | $N_{bf}^{PRGM}$ | $N_{bf}^{RAM}$ | $N_{fix}$ | $N_{wdt}$ | $N_{reprogr}$ | $N_{reprogr}^{other}$ | **Final state** |
|---|---|---|---|---|---|---|---|
| **1** | 393 | 127 | 329 | 35 | 40 | 46 | Working |
| **2** | 390 | 127 | 332 | 22 | 44 | 44 | Working |

The figures collected on both devices are similar which is logical given the symmetry of the system.
We can notice that the ratio of the number of sections fixed over the number of bit-flips injected in the program memory $\frac{N_{fix}}{N_{bf}^{PRGM}}$ is a little bit larger than in the previous experiment (84% against 73%). This is probably due to the variation in the ratio of $P_{bf}^{PRGM}$ over the period of the PMCDC checks $P_{pmcdc}$ (4s against 1.3s). This could also explain the improvement in the number of bit-flips injected in the program memory per reprogramming sustained (9 against 7).
The rate of reprogramming failure remains similar.
On average, the service duration without interruption was about 6 minutes for both devices, which is the value for an amount of radiation 250 times higher than the expected value.
No takeover was performed by any device during the experiment.

The results of this test are very positive. The system, equipped with the full protective stack of bit-flip tolerant techniques, was able to run autonomously for 8 hours with an amount of bit-flips 250 times higher than the expected orbital amount. After 8 hours in this intensive simulated radiated environment, the system was still running and delivering a full service which leaves us to hope for a great improvement of the lifetime of the satellite if the protective stack design in this thesis is used on NUTS.

**Note:** A second experiment with different bit-flip sequences was conducted. After 138 minutes, MCU_A switched to single master mode following 5 unsuccessful reprogramming attempts of MCU_B. It was able to run 7 minutes before an irrecoverable failure occurred. Once again, the additional time of service offered by the takeover technique need to be scaled by 250. The reason why the reprogramming of MCU_B failed 5 times in a raw is not fully understood.

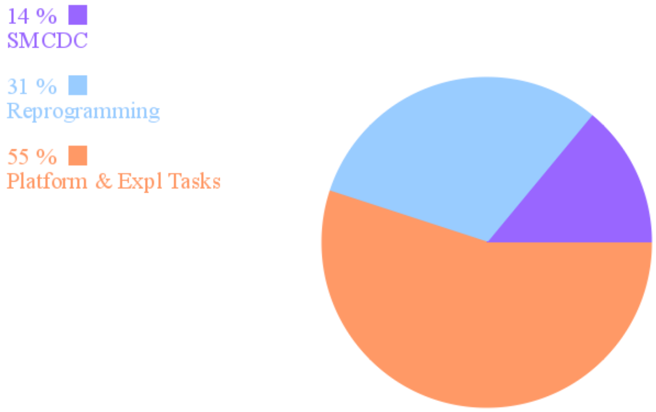

**Figure 7.5:** Repartition of the program memory usage.

## 7.4 Resources Utilization

Let us conclude this chapter by an analysis of the resources utilization of the implementation.

Figure 7.5 shows the repartition of the program memory usage between the three main components of the system. The platform and task examples (in orange) take up more than half of the design. The reprogramming facility (in blue) take up one third while the PMCDC facility (in purple) represents only one fifth.
Some work on the reprogramming facility seems necessary in order to achieve a lower memory print. In the current design the protective techniques take up about 45% of the design which is quite a lot.

Figure 7.6 shows the space the design takes up in the overall flash memory available. More than 80% remain free on both devices to implement the embedded software of the satellite which is with no doubt enough.

The RAM, processing time, power and energy consumption has not been assessed.

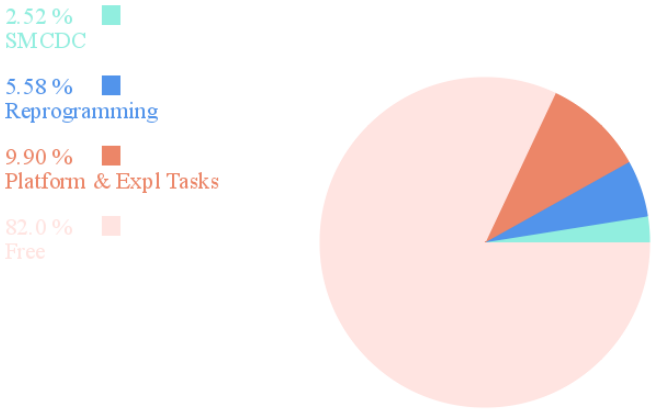

**Figure 7.6:** Estimation of the program memory utilization.

# Chapter 8
# Related Work

## 8.1 Literature Review

The literature counts plentiful of papers dealing with reliability of space-based systems. This section presents a selection of some of them that propose interesting ideas that inspired this work.

Caldwell *et al.* [RH01] and [CR98] present an architecture constituted of eight MCUs using the approach of voting and comparison in order to achieve a fault-tolerant system. The design counts permanently one master coordinating the computation of the slaves, and elected by them. The system is dynamic in the way that when the master fails to deliver a result, one of the slaves is elected as master. The deficient parts are excluded from the system and reinserted once they have recovered a stable state.

White [Whi82] unveils the concept of Error Detection and Correction (EDAC). He presents a memory architecture able to detect and correct errors induced by radiations where an EDAC facility, intermediate between the CPU and the memory, eliminates any error.

Charles *et al.* [CHY04] show the advantages of embedding in space-based systems a Triple-Modular-Redundant processor implemented on an FPGA. In addition to the fact that the design is fault-tolerant thanks to the triple redundancy and an embedded EDAC facility, their solution present the advantages of being flexible, reconfigurable and enables on-orbit upgrades to the software as well as the soft-core architecture.

Li and MacMillin [LM93] propose a fault-tolerant distributed deadlock detection algorithm using a priority-based probe algorithm.

Burns and Wellings [BW09] summarize a large number of fault-tolerant software techniques such as checkpointing and error handling, and illustrating how they can be implemented.

In addition to listing a lot of very useful software methods and concepts which can be employed to achieve reliable system, Torres-Pomales [TP10] introduces the concept of process pair which can be implemented with two MCUs.

## 8.2 Other CubeSat Satellites Review

All CubeSat satellites having similar constraints (limited budget, space and power supply), it is interesting to review the fault-tolerant solution employed on other CubeSat satellites.

Fitzsimmons [Fit12] presents its work on the software update facility implemented on the future satellites developed by PolySat (Cal Poly's CubeSat development group). Software updates are to be used for software upgrade purpose as well as recovery. He uses a system of partitions that can be upgraded independently and a check-sum verification to validate the upgrade.
Interestingly enough, he states that, the results from the six past missions of PolySat oriented the group to the conclusion that the increase in complexity due to the use of redundant MCUs may have been the cause of mission failures instead of providing more reliability. The new architecture of their satellites counts from now on only one single high performance and low power MCU. This is the opposite of most fault-tolerant strategies [RH01], [CHY04], [TP10], [CR98].

Manyak [Man11], from the same university, details the fault-tolerant software architecture used on the latest generations of PolySat satellites. He explains that the maximization of the payload volume justifies the abandonment of redundant components, the reliability of the satellite relying exclusively on fault-aware software. His system makes use of two layers of watchdogs and a radiation hardened memory. The latter is used until the content of the RAND memory has been validated.
Furthermore, the author provides a very interesting and detailed review of fault-tolerant solutions employed on nine CubeSats.

Chapter 9

# Conclusions and Future Work

In this thesis, the improvement of the reliability of NUTS in a radiated environment has been covered. A study of the effects of bit-flips in memory devices has been conducted, a set of software solutions has been listed, and, selecting the best options, a protective stack aiming at increasing the tolerance of the system in the presence radiation-induced bit-flips has been proposed. Implemented in a simplified prototype, each of them has been thoroughly tested, first individually to assess their performance, then altogether to measure the increase in reliability brought by the solution.
Due to the lack of maturity of NUTS' embedded software, this work remained in the experimental domain and serves as proof-of-concept.

The following sections summarizes and discusses the results achieved, and finally a list of further improvements and future work is given.

## 9.1 Summary of the Results

As reminded in Figure 9.1, the proposed stack contains four protection techniques acting at different level in the failure process:

- Internal watchdog which reboots the MCU in case of timeout,
- Program Memory Corruption Detection and Correction (PMCDC) facility which detects and corrects bit-flips in the program memory,
- Reprogramming facility which enables one MCU to reprogram the other one in case of failure of the latter,

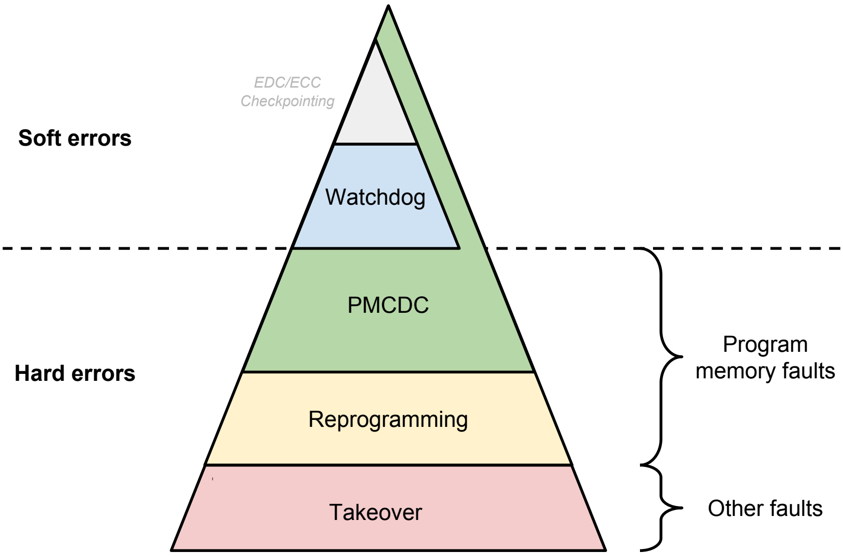

**Figure 9.1:** Implemented stack of protection techniques.

- Takeover mechanism with which one MCU can run the critical tasks of the other MCU in order to provide a degraded service in case of the irremediable failure of the other MCU.

A prototype implementing all those techniques has been able to run for more than 8 hours in a simulated radiated environment introducing 250 times more bit-flips in both RAM and program memory than the expected on-orbit rate. The PMCDC facility was able to clear more than 80% of the faults before they introduce any error in the system, dividing by 5 the apparent bit-flip rate. The experiment was stopped after 8 hours, but both MCUs were still delivering correctly their service. Those results are every positive and leave the hope of an interesting gain in reliability for NUTS.

Regarding resource utilization, the implementation of the protective stack requires 18% of the internal flash memory, which leaves 82% to implement the embedded software of NUTS on both MCUs.

This work has been accomplished with the concept of re-usability in mind. Particularly, care has been taken in order to facilitate the integration of this work with the software of the satellite. As a result, using the protective stack on the real hardware, along with the embedded software should require a minimal amount of effort, while providing a promising increase in reliability of the system.

## 9.2 Improvement and Future Work

The present work proposed a solution to cope with radiation-induced bit-flips for NUTS. But a lot of work remains to do before it can be used in space. This section provides a list of ideas of possible improvement as well as the future work required in order to integrate it with the embedded software of the satellite.

### 9.2.1 Improvement

The tests reported in Chapter 7 raised some axis of improvement that are enunciated in this paragraph.

#### 9.2.1.1 PMCDC

The experiment described in Paragraph 7.2.2 showed that a reduction of the size of the flash drivers would improve dramatically the performance of the PMCDC facility. Hence some work should be carried out in this direction.

Additionally, moving earlier the initial memory check currently present at the beginning of the main function would reduce the probability of irrecoverable failure. Its ideal location is as early as possible, before the call of the main function.

#### 9.2.1.2 Reprogramming

In the same way, the testing of the reprogramming procedure unveiled a failure rate of about 10% (cf. Paragraph 7.3.1). Some improvement of the JTAG controller could probably decrease the number of reprogramming procedures needed, and hence minimize the risk of a catastrophic simultaneous failure of both MCUs.

Additionally, the reprogramming process uses a backup image stored in a rad-hard memory device. In this work it has been assumed that no data corruption could ever happen in this image. Unfortunately this is possible. Therefore, checking the backup image could be considered.

#### 9.2.1.3 Takeover

Some improvement could be brought to the takeover mechanism which requires a reboot of the working MCU each time it switches between normal and damaged mode. The interruption of service implied could be avoided if the launch and shutting down of critical and non-critical tasks was performed dynamically.

### 9.2.2 Future Work

Finally this paragraph lists some tasks that need to be carried out in order to integrate this work with NUTS's source code.

An implementation of an EDAC and a recovery method based on checkpointing has been implemented by Ødegaard in [deg13]. Both methods has their place in the protective stack (cf. Figure 9.1). Some work is needed to integrate them whith the four already gathered.
In addition, all the work presented in this report has been developed and tested on a prototype using exclusively two Atmel XPlained boards. Before trying to merge the code developed here and the software of the satellite, the implementation should be first ported to the hardware of NUTS. Fortunately, this task should not be difficult thanks to the care taken while developing this project.
Then, the protective task should be integrated in the software developed for NUTS. The numerous parameters defining its behavior should be adjusted to match the real constraints of the mission and the characteristics of the embedded software.
Finally, the whole system should be carefully tested in order to assess the tolerance of the system in radiated environment.

# References

[AS12] K. Ø degaard A. Skavhaugn. Survey of correction methods for faults and errors induced by cosmic radiation on operating system level in cubesats. In *IAA Conference On University Satellite Missions And Cubesat Workshop*, volume 2, 2012.

[Atm12] Atmel. *AT32UC3A3*, October 2012.

[Bar09] R. Barry. *Using the FreeRTOS Real Time Kernel.* 2009.

[Bek] J. K. Bekkeng. Radiation effects on space electronics. Presentation at the University of Oslo.

[Bir11] R. Birkeland. Nuts-1 mission statement. June 2011.

[Bru11] D. Bruyn. Power distribution and conditioning for a small student satellite. Master's thesis, NTNU, June 2011.

[BW09] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages.* Addison Wesley, 4th edition edition, 2009.

[Che09] Y. Chen. Flash memory reliability nepp 2008 task final report. Technical report, Jet Propulsion Laboratory, 2009.

[CHY04] H. Loomis A. Ross C. Hulme, USMC and R. Yuan. Configurable fault-tolerant processor (cftp) for spacecraft onboard processing. In *IEEE Aerospace Conference Proceedings*, pages 2269–2276, 2004.

[CR98] D. W. Caldwell and D. A. Rennels. Minimalist recovery techniques for single event effects in spaceborne microcontrollers. Technical report, UCLA, 1998.

[deg13] K. Ødegaard. Error detection and correction for low-cost nano satellites. Master's thesis, NTNU, July 2013.

[DNNJ99] G. M. Swift D. N. Nguyen, S. M. Guertin and A. H. Johnston. Radiation effects on advanced flash memories. *IEEE Transactions on Nuclear Science*, 46(6):1744–1751, 1999.

[Dor03] Kevin Dorow. Flexible fault tolerance in configurable middleware for embedded systems. *Proceedings of the 27th Annual International Computer Software and Applications Conference*, 2003.

[Fit12] S. Fitzsimmons. Reliable software updates for on-orbit cubesat satellites. Master's thesis, Cal Poly, June 2012.

[GCP01] A. Chimenton J. Wyss A. Modelli L. Larcher G. Cellere, P. Pellati and A. Paccagnella. Radiation effects on floating-gate memory cells. *IEEE Transactions on Nuclear Science*, 48(6):2222–2228, December 2001.

[Gis12] A. Giskeoedegaard. Implementing csp over i2c on ntnu test satellite. Master's thesis, NTNU, December 2012.

[Har14] Wil Harkins. Space radiation effects on electronic components in low-earth orbit. http://www.nasa.gov/offices/oce/llis/0824.html, 2014.

[Joh] A. H. Johnston. Space radiation effects on microelectronics. Presentation from the National Aeronautics and Space Administration.

[Lau11] J-M. Lauenstein. *Single-Event Gate Rupture In Power MOFSETS: A New Radiation Hardness Assurance Approach.* PhD thesis, University of Maryland, 2011.

[Liu01] Chung-Yu Liu. A study of flight-critical computer system recovery from space radiation-induced error. Technical report, Honeywell Defense Avionics Systems, 2001.

[LM93] P. Li and B. McMillin. Fault-tolerant distributed deadlock detection/resolution. *IEEE*, 1993.

[m12] D. Holmstrøm. Software and software architecture for a student satellite. Master's thesis, NTNU, 2012.

[Man11] Greg Manyak. Fault tolerant and flexible cubesat software architecture. Master's thesis, Cal Poly, June 2011.

[OM03] T. R. Oldham and F.B. McLean. Total ionizing dose effects in mos oxides and devices. *IEEE transactions on nuclear science*, 50(3):483 – 499, June 2003.

[RH01] D. A. Rennels and R. Hwang. Recovery in fault-tolerant distributed microcontrollers. Technical report, University of California at Los Angeles, 2001.

[RW07] T. Molteno R. Ward. Table of linear feedback shift registers. Technical report, University of Otago, October 2007.

[TP10] Wilfredo Torres-Pomales. Software fault tolerance: A tutorial. Technical report, NASA, October 2010.

[Vol11] M. Volstad. Internal data bus of a small student satellite. Master's thesis, NTNU, June 2011.

[web14a] Cubesat website. http://cubesat.org/, 2014.

[web14b] Nuts cubesat project homepage. http://nuts.cubesat.no/, 2014.

[Whi82] J.B. White. Fault tolerant memory system architecture for radiation induced errors. *IEEE transactions on aerospace and electronic systems*, AES-18(01):39–47, January 1982.

Appendix A

# Theoretical Study of the Effect of a Periodic Correction of the Program Memory

One solution to make a system tolerant to bit-flips discussed in Paragraph 3.2.2.3 consisted in checking periodically the content of the program memory to detect bit-flips and to fix it. This Appendix is dedicated to a mathematical study of this case.

## A.1 Hypothesis

We consider a task $T$ which runs periodically with a period $P_t$. The task that checks the content of the memory and corrects it if needed is called $C$ and runs periodically with a period $P_c$. Let $a$ be the ratio $\dfrac{P_t}{P_c}$.

Both tasks are supposed to complete infinitely quickly. A task that is executed is represented by an arrow on the graphs displayed in Figure A.1. The color red or blue depend on the value of $a$.

Let $\mathcal{P}$ be the density of probability of the event "failure" which is characterized by a bit-flip affecting the task $T$ and $T$ being called in this corrupted state (*i.e.* before the checking task $C$ had had time to clear the fault). The probability of a bit-flip corrupting $T$ being dependent of the size of $T$ and the temporal density of bit-flips, we call $\mathcal{Q}$ the average probability that a bit-flip results in a failure, simply related to $\mathcal{P}$ by a proportional relation

$$\mathcal{P} \propto S \cdot \mathcal{Q},$$

where $S$ has been defined in Chapter 7 as the scale factor of the density of bit flips.

## A.2 Demonstration

The demonstration is split up by the discrimination of the $a$.

### A.2.1 Case Where $a \geq 1$

Let us consider first that $a \geq 1$. The periodic execution of $T$ is represented by the red arrows, while the execution of the checking task $C$ is represented by blue arrows.

Let us suppose that $a \in \mathbb{Q}$, i.e. $\exists\,(m, n) \in \mathbb{N} \times \mathbb{N}^* / a = \dfrac{m}{n}$. Let us choose $m$ and $n$ so that the fraction $\dfrac{m}{n}$ is irreducible. With this convention, for $t = mn$, both tasks have the same delay as the one they had at $t = 0$. Figure A.1 illustrates the case where $a = \dfrac{5}{4}$ over one full period $(= 20)$.

By definition of $\mathcal{Q}$, we suppose that one bit-flip affecting $T$ will occur sometimes during the period of time represented on the diagram with an equal probability (represented by the horizontal black line).

Let us suppose that a bit-flip corrupts $T$ somewhere in one of the area in white. The next event is a memory check (blue arrow) which will clear the error before $T$ is effectively used. There is no failure in this case. Hence the white areas do not contribute for $\mathcal{Q}$.
However if the bit-flip occurs in a grey area, the next event is an execution of $T$ (red arrow) which implies a failure of the system since $C$ did not have time to correct the fault before the corrupted task is used. Hence the grey areas contribute for $\mathcal{Q}$.

Let $\delta$ be the delay between the execution of $T$ and $C$ at $t = 0$. Figure A.1 shows three diagrams for the three values $\delta = \left\{0^-, \dfrac{1}{10}, \dfrac{1}{5}\right\}$. Note that $\delta = \dfrac{1}{5} \Leftrightarrow \delta = 0^-$.

By counting of the grey area, we find the expression

$$\mathcal{Q}_\delta = \frac{n-1}{2m} + \frac{\delta n}{m},$$

with $\delta \in \left[0, \dfrac{1}{n}\right[$. Since we are not interested in the probability in each delay $\delta$ but rather in the average value, by linearity of $\mathcal{Q}_\delta$ in relation to $\delta$, we get:

$$\mathcal{Q} = \frac{n}{2m} = \frac{1}{2a}, \forall a \in \mathbb{Q}, a \geq 1$$

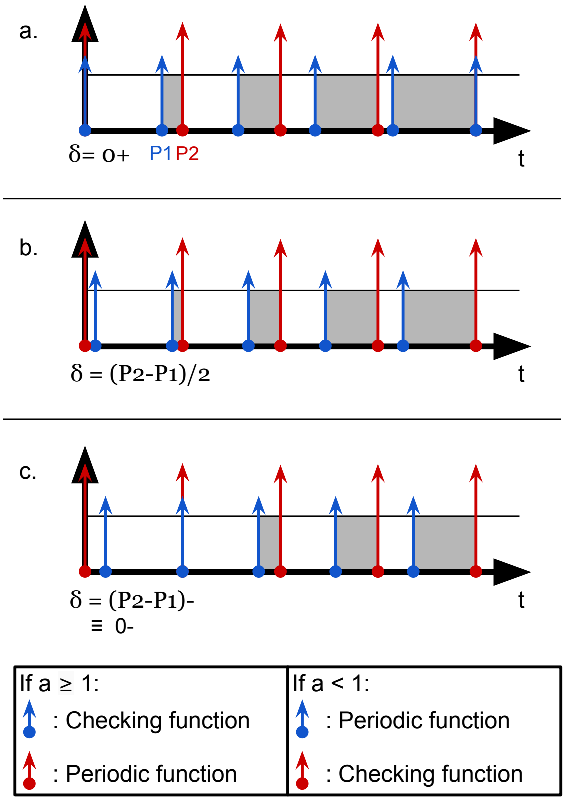

**Figure A.1:** Diagram representing the cyclic function calls and memory checks.

Finally, by density of $\mathbb{Q}$ in $\mathbb{R}$ we get:

$$\mathcal{Q} = \frac{1}{2a}, \forall a \in [1, \infty[$$

### A.2.2 Case Where $a < 1$

All we need to do is to reverse the color code: the red arrows represent the execution of the checking task $C$ and the blue ones are the representation of the execution of the task $T$. Once we notice that the case where $a < 1$ is equivalent to the previous case with $\overline{a} = \frac{1}{a} \geq 1$ where we estimate the white area instead of the grey area, it is immediately possible to conclude that:

$$\mathcal{Q} = 1 - \frac{a}{2}, \forall a \in [0, 1[$$

## A.3 Summary

If $\mathcal{P}$ be the density of probability of the event "failure", and $\mathcal{Q}$ is the average probability that a bit-flip results in a failure. $\mathcal{P} \propto S \cdot \mathcal{Q}$ and

$$\mathcal{Q} = \begin{cases} 1 - \frac{a}{2}, & \forall a \in [0, 1[ \\ \frac{1}{2a}, & \forall a \in [1, \infty[ \end{cases} \qquad (\Lambda)\,,$$

where $a = \frac{P_t}{P_c}$.

## A.4 Experimental Verification

An experimentation has been carried out in order to check the mathematical expression demonstrated above.

A periodic function *target* was periodically called on an MCU protected by PMCDC. The ratio of the period of the function over the period of the periodic checks is defined as $a$ to respect the conventions of the demonstration. Given the importance of $a$ in the equation we want to verify, the synchronization of the two periodic elements has been particularly carefully controlled.
Only the first byte of *target* was flipped at random periods of times. Instead of executing the function when called, the first byte was simply analyzed in order to determine whether a bit-flipped had occurred or not. If flipped, then the experimental platform would record a failure.

For each value of $a$ tested, 250 bit-flips has been sent to the system, and the number of failure has been recorded. Fig. A.2 presents the experimental data (red crosses) along with a plot of the mathematical formula $(\Lambda)$.

We can observe that the experimental data are overall close to the theoretic curve.

We have been doing a lot of approximations in the establishment of $(\Lambda)$ which can explain the differences. The most obvious is the fact that, contrary to what has been assumed, a check and correction performed by the PMCDC facility is not instantaneous (it lasts about 1s, depending on the size of the program memory and the number of sections to be fixed).

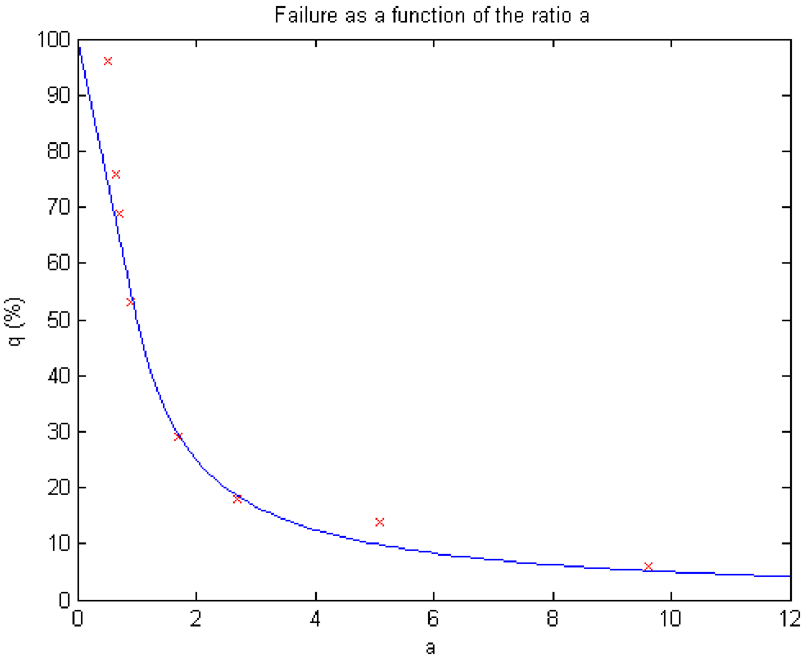

**Figure A.2:** Error probability $\mathcal{Q}$ as a function of $a$.