EDAC software implementation to protect small satellites memory

Rita Roca Taxonera

School of Electrical Engineering

Final Project

Espoo 21.05.2019

Supervisor

Alexandre Bosser, Ph.D.

Advisor

Prof. Jaan Praks

Aalto University
School of Electrical Engineering

Copyright © 2019 Rita Roca Taxonera

Aalto University
School of Electrical Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's final project

Author Rita Roca Taxonera

Title EDAC software implementation to protect small satellites memory

Degree programme Electronics and electrical engineering

Major Industrial Engineering **Code of major** -

Supervisor Alexandre Bosser, Ph.D.

Advisor Prof. Jaan Praks

Date 21.05.2019 **Number of pages** 60+15 **Language** English

Abstract
Radiation is a well-known problem for satellites in space. It can produce different negative effects on electronic components which can provoke errors and failures. Therefore, mitigating these effects is especially important for the success of space missions. One of the techniques to increase the reliability of memory chips and reduce transient errors and permanent faults is Error Detection and Correction (EDAC). EDAC codes are characterised by the use of redundancy to detect and correct errors. This final project consists in the implementation of a software EDAC algorithm to protect the main memory of a microcontroller. The implementation requirements and the issues of software EDAC are described and the test results are commented.

Keywords software EDAC, error detection and correction (EDAC), single-event upset (SEU), LEO, small satellites

Preface

I would like to thank Alexandre Bosser for supervising this final project, for being always available and for his invaluable advice and support. I also would like to thank Jaan Praks for introducing me to the satellites world that was new to me, and also for all his great help and advice. I would like to thank Petri Niemelä all his support during the implementation.

Thanks to all my friends from Rumbau for always being there for me, especially to Mercè, who encouraged me to start this great adventure in Finland.

Thanks to all my friends at the Silta student residence, for all their support, for the great moments we spent together and for making this experience unforgettable.

Finally, thanks a lot to all my family, but especially to my parents, for providing me with unfailing support and continuous encouragement throughout my years of study. Without them, none of this would have been possible.

Barcelona, 21.5.2018

Rita Roca Taxonera

Contents

Abstract **3**

Preface **4**

Contents **5**

Symbols and abbreviations **8**

1 Introduction **10**

2 Radiation Environment in LEO **12**
- 2.1 Solar energetic particles . . . 12
- 2.2 Trapped radiation . . . 13
- 2.3 Galactic cosmic rays . . . 15

3 Memory devices **17**
- 3.1 Volatile Memories . . . 18
  - 3.1.1 SRAM . . . 18
  - 3.1.2 DRAM . . . 19
- 3.2 Non-Volatile Memories . . . 20
  - 3.2.1 Flash . . . 20
  - 3.2.2 FRAM . . . 21

4 Radiation effects on electronic compontents **22**
- 4.1 Cumulative effects . . . 22
- 4.2 Single event effects . . . 23
  - 4.2.1 SEU . . . 24
  - 4.2.2 SEFI . . . 26

5 Error Detection And Correction codes **27**
- 5.1 Linear Block Codes . . . 30
  - 5.1.1 Minimum distance . . . 32
  - 5.1.2 Detection and correction capabilities of a code . . . 32
  - 5.1.3 Systematic codes . . . 33

5.1.4 Modifications to Linear codes . . . 34
5.1.5 Binary Hamming codes . . . 36

6 Memory protection with EDAC **37**
6.1 Hardware EDAC . . . 37
6.2 Software EDAC . . . 38
6.2.1 Scrubbing . . . 38
6.2.2 Check bits location . . . 39
6.2.3 Interleaving . . . 40
6.2.4 Protection of the scrubbing program . . . 41

7 FreeRTOS **42**

8 Experimental setup **43**

9 Software implementation **45**
9.1 Bootloader and application processes . . . 45
9.2 Memory map . . . 46
9.3 Threads . . . 47
9.4 Scrubbing Algorithm . . . 49
9.4.1 Code selected: Hamming(39,32) . . . 49
9.4.2 Implementation . . . 50

10 Results **51**
10.1 Experimental results . . . 51
10.2 Application of the results to a case study . . . 54

11 Summary **56**

References **57**

A Hamming(39,32) matrices **61**

B Bootloader code **62**
B.1 Configuration files . . . 62
B.1.1 (config.h) . . . 62

B.1.2 Scatter file (EDACBootloader.sct) 62
B.2 Main function (main.c) 63
B.3 Jump to Application function (bootloader.c) 64

C Application code **64**
C.1 Configuration files 64
C.1.1 (config.h) 64
C.1.2 Scatter file (APP.sct) 65
C.2 Main function (main.c) 65
C.3 Thread functions (main.c) 66
C.4 Hamming code (hamming.c) 68
C.5 Code for the introduction of errors (error.c) 73

Symbols and abbreviations

Symbols

d_{min}	Minimum Hamming distance
R_c	Code Rate
$\mathbf{C_b}$	Block Code
$\mathbf{G}$	Generator matrix
$\mathbf{H}$	Parity check matrix
$\mathbf{c}$	Codeword
$\mathbf{m}$	Message
$\mathbf{s}$	Syndrome
$\mathbf{e}$	Error
k	Number of message bits
n	Number of codeword bits
t	Error-correction capability of a code
I_x	Identity matrix. x corresponds to the number of rows of the matrix

Abbreviations

ACE	Advanced Composition Explorer
BCH	Bose-Chaudhuri-Hocquenguem
CMOS	Complementary Metal-Oxide-Semiconductor
CME	Coronal Mass Ejection
DEC-TED	Double Error Correction - Triple Error Detection
DRAM	Dynamic Random-Access Memory
DWC	Duplication With Comparison
ECC	Error Control Code / Error Correcting Code
EDAC	Error Detection And Correction
EEPROM	Electrically-Erasable Programmable Read-Only Memory
FRAM	Ferroelectric Random-Access Memory
GCR	Galactic Cosmic Ray
HW	Hardware
IC	Integrated Circuit
IDE	Integrated Development Environment
I/O	Input/Output
LEO	Low Earth Orbit
LET	Linear Energy Transfer
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
MBU	Multiple-Bit Upset
MCU	Multiple-Cell Upset
NMOS	Negative Metal-Oxide-Semiconductor
NVM	Non-Volatile Memory
OS	Operating System

PMOS	Positive Metal-Oxide-Semiconductor
RO	Read Only
RS	Reed-Solomon
RTOS	Real-Time Operating System
RW	Read-Write
SAA	South Atlantic Anomaly
SBU	Single-Bit Upset
SEC-DED	Single Error Correction - Double Error Detection
SEE	Single Event Effect
SEFI	Single-Event Functional Interrupt
SEP	Solar Energetic Particle
SEU	Single-Event Upset
SPE	Solar Particle Event
SRAM	Static Random-Access Memory
SW	Software
TID	Total Ionizing Dose
TMR	Triple Modular Redundancy
USART	Universal Synchronous/Asynchronous Receiver/Transmiter
USB	Universal Serial Bus
UV	Ultraviolet

1 Introduction

The Sun is the main source of interplanetary radiation. As a result, the radiation environment is more intense as the distance to the Sun decreases. Satellites in the Low Earth Orbit (LEO) are subject to radiation due to different sources: solar energetic particles, trapped radiation in the Earth's magnetic field and galactic cosmic rays.

Radiation can produce different negative effects on electronic components which can provoke errors and failures in the satellites. Therefore, mitigating these effects is especially important for the success of space missions. Over the years, different ways to protect the devices from radiation have been studied, for example, the improvement of the hardware design of the components to make them less sensitive to radiation.

One category of radiation effects are the single-event effects (SEEs). SEEs in microelectronics are caused when highly energetic particles strike sensitive regions of an electronic component. The particle strike can cause different effects such as transient disruption of a circuit operation, a change of logic state, or even permanent damage to the device or integrated circuit (IC) [33].

According to Moore's law, due to the advancement of technology, the physical size of IC features has decreased and more complex circuitry has been developed. Due to this significant reduction in the size of the transistors, the amount of energy required to induce a change in their logical state has also been reduced. This fact has led to an increase in the sensitivity of components towards SEEs, and the study of solutions to mitigate them has become more important.

One type of SEE is the Single-Event Upset (SEU). An SEU is produced when an energetic particle changes the state of a circuit, causing one or more bit flips in memory cells or registers. These errors are non-destructive effects since the device can continue to perform normally by reprogramming the circuit into its correct logical state.

One of the techniques to increase the reliability of memory chips and reduce transient errors and permanent faults is Error Detection and Correction (EDAC). EDAC codes were first studied in the late 1940s by Claude Shannon and Richard W. Hamming, who were interested in how to correct errors that appeared in the messages sent over long telephone lines. Since then, new codes have been developed and those already existing have been improved. EDAC codes are characterised by the use of redundancy to detect and correct errors.

The main goal of error-control coding is to provide reliable messages in an efficient manner. It is necessary to find a balance between reliability and efficiency, since higher reliability can imply more calculation time and/or higher memory usage and, therefore, less efficiency. Hence, for a given application case, the best possible trade-off between reliability and added overhead must be decided to determine the most appropriate EDAC code.

Error-correction codes can be implemented using hardware or software. Hardware implementation is done by extending the memory bus architecture to accommodate the check bits, and adding extra circuitry to detect and correct memory errors [35]. The main drawback of hardware EDAC is that it is an expensive solution, as it adds complexity to the design and increases the silicon footprint.

Software EDAC are a low-cost solution and provide the flexibility of implementic more complex coding schemes [35]. However, their reliability is lower compared to hardware EDAC, especially in high-radiation environments.

This project consists in the implementation of a software EDAC algorithm to protect the main memory of a microcontroller. The main objectives are to implement a scrubbing algorithm capable of protecting the memory against SEUs, and to analyse whether software EDAC can be a reliable solution when hardware-implemented EDAC is not possible. To develop the scrubbing algorithm, it is necessary to decide different factors such as the EDAC code that will be used and the structure of the program, among others.

This document is structured as follows:

The first sections (2 to 7) are a theoretical introduction to the radiation environment in LEO and its effects on memory devices, EDAC codes and their application in memory protection, and real-time operating systems.

The last sections (8 to 10) describe the EDAC software implementation performed in this project, the test procedure used to analyse its reliability and performance, and the results obtained.

2 Radiation Environment in LEO

Radiation is the propagation of energy in two possible ways: in the form of high-speed particles (electrons, protons, neutrons and heavy ions), or in the form of electromagnetic waves (radio waves, microwaves, infrared, visible and UV light, X-rays and gamma rays).

The main source of interplanetary radiation is the Sun. The Sun produces a continuous stream of particles known as solar wind. This stream is composed mainly of protons and electrons in almost equal numbers, and a low percentage of heavy nuclei. In the vicinity of the Earth, the solar wind speed can vary between 300 and 1000 km/s, depending on solar activity [6].

The Sun also routinely emits massive bursts of plasma in events known as coronal mass ejections (CMEs). These events are known as solar particle events (SPEs). SPEs are unpredictable and can occur at any time during the solar cycle. The frequency of SPEs directed to the Earth can vary from one every two months to one every two years [2]. Large SPEs -which can last for several days- occur more frequently in periods of increased solar activity [1].

Energetic particles in the heliosphere are variable in intensity and composition. Figure 1 shows the helium, oxygen and iron fluences obtained by the spacecraft Advanced Composition Explorer (ACE) during a 3-year period (from 1997 to 2000). The measurements were made at 1.5 million km from Earth and 148.5 million km from the Sun [10].

The energy range between 5 and 10 keV/nucleon that can be observed in Figure 1 is dominated by the solar wind (including both slow and high-speed wind). In the intermediate region, from $\sim$ 30 keV/nucleon to $\sim$ 30 MeV/nucleon different sources or events contribute to the fluence spectra. [8]

The particle radiation in LEO originates from three main sources: solar energetic particles, trapped radiation and galactic cosmic rays.

2.1 Solar energetic particles

The particles emitted during CMEs are known as solar energetic particles (SEPs). They are mainly protons, electrons and heavier charged particles. These particles travel in a spiral path along the Sun's magnetic field lines and a small percentage of them reaches the Low Earth Orbit [3].

Figure 2 gives the fluence energy spectra for the larger SPEs between 1956 and 1990.

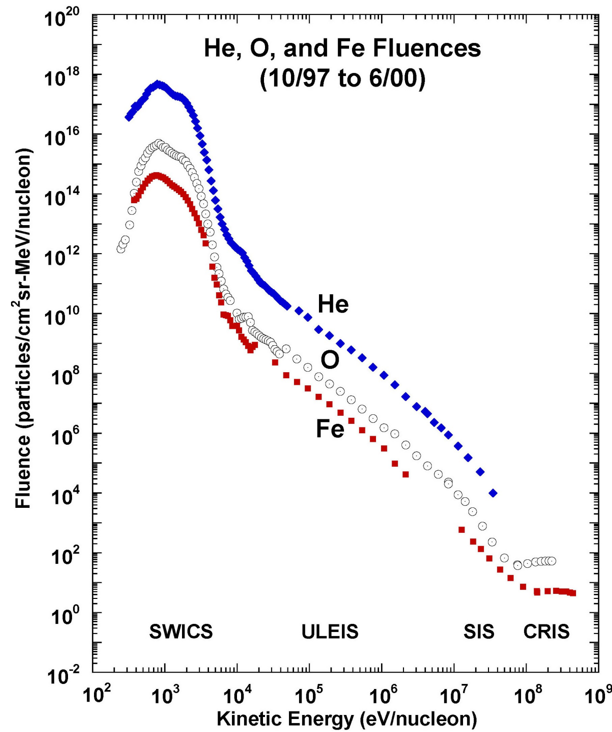


Figure 1: Fluences of He, O, and Fe nuclei measured by several instruments on board ACE during a 3-year period. Figure taken from [8].

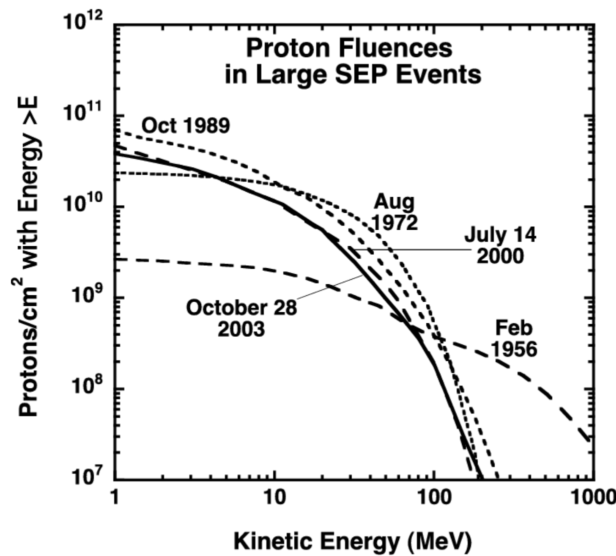


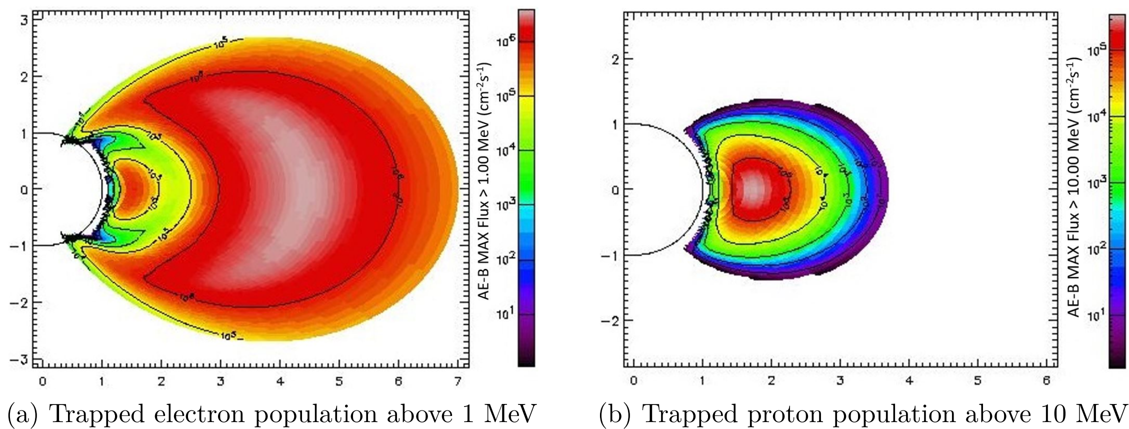
Figure 2: Solar particle event fluence: spectra of larger solar proton events from 1956 to 2003. Figure taken from [9].

2.2 Trapped radiation

When the energetic particles emitted by the Sun encounter the Earth's magnetic field, they are deflected by Lorentz forces. Due to this phenomenon, some of the

particles, mainly protons and electrons, can be trapped in a region called the Van Allen radiation belts. This region contains two main belts: a relatively stable inner belt dominated by the presence of protons, and an outer belt consisting predominantly of high-energy electrons. The energies for protons range from 0.01 to 400 MeV, with fluxes that range from 600 to $10^8/\text{cm}^2$; electron energies range from 0.4 to 4.5 MeV, with fluxes that range from 100 to $4 \cdot 10^8/\text{cm}^2$ [2]. Variations in the particle fluxes are caused by fluctuations in solar activity, such as the frequency and magnitude of solar particle events. In general, the Sun's activity increases and decreases over an 11-year period, called a solar cycle.

The particles trapped in the Van Allen Radiation Belts are distributed within the region in a non-homogeneous way, both in altitude and in latitude. Figure 3 shows the distribution of trapped protons with energies above 10 MeV and the trapped electron population above 1 MeV.



(a) Trapped electron population above 1 MeV

(b) Trapped proton population above 10 MeV

Figure 3: Particle fluxes of trapped radiation. Figures taken from [11].

One of the reasons for this varied distribution is that the outer electron radiation belt is closer to the Earth at high latitudes than at low altitudes. The other main reason is known as the *South Atlantic Anomaly* and is located at low altitudes and low inclinations [5].

The *South Atlantic Anomaly* (SAA) is caused by the displacement and inclination of the geomagnetic axis in relation to the Earth's rotation axis [2], as shown in Figure 4. This phenomenon increases the flux of trapped particles (especially protons) in that region. The protons of the SAA are a known radiation problem for spacecraft in LEO.

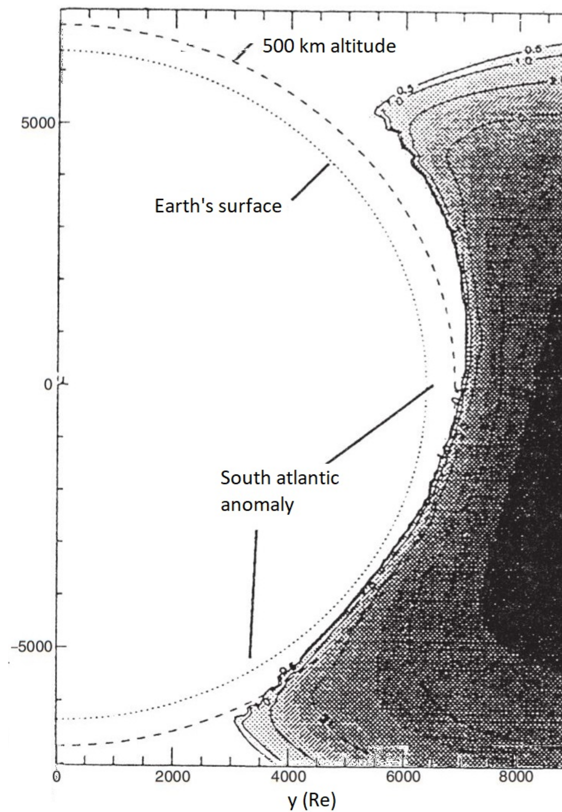


Figure 4: South Atlantic anomaly: proton radiation belt is brought below 500 km altitude due to tilt and offset of the geomagnetic axis. Figure taken from [2].

2.3 Galactic cosmic rays

Galactic cosmic rays (GCRs) are highly energetic nuclei (mainly between 100 MeV per nucleon and 10 GeV per nucleon) which are accelerated in the shocks produced by supernova explosions [1]. Although the flux of particles is expected to be almost constant outside of the Solar system, GCRs are affected by the modulation of solar activity and, in periods of low solar activity, the flux of low-energy GCRs in the inner Solar System is four to five times greater than during high solar activity [3]. GCRs are mainly composed of protons and alpha particles, and a few heavy ions. Figure 5 shows a GCR spectrum. It can be seen that the GCR fluxes are anti-correlated with solar activity: when the solar activity is the highest, GCR fluxes are low and vice versa.

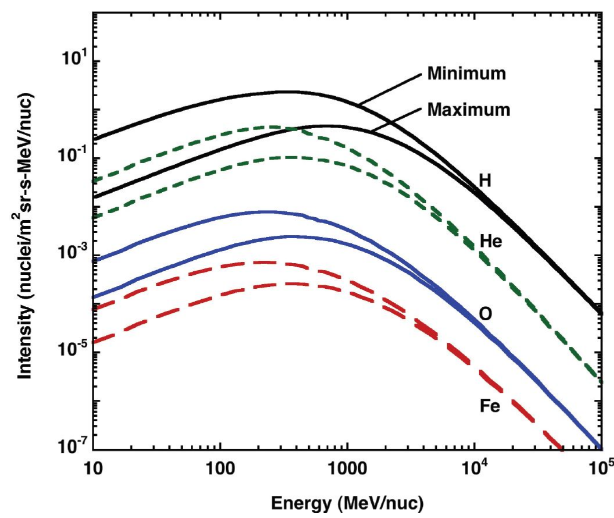


Figure 5: GCR spectra depicting the intensity variations between solar maximum and solar minimum conditions. The upper curve for each species is for solar minimum, when cosmic rays can penetrate into the inner heliosphere more easily. Figure taken from [1], source: Courtesy of R.A. Mewaldt, California Institute of Technology.

3 Memory devices

Over the years, different memory technologies have been used to store digital data. One of the first devices designed to store data is magnetic core storage, which was developed in the early 1950s. These first devices, in which data could be accessed in microseconds, started being replaced in the 1960s by solid-state memories, which can access the data in a matter of nanoseconds. Solid-state memories are implemented on a single integrated circuit (IC) made up of transistors printed on small chips of silicon.

In 1965, Gordon Moore, the co-founder of Fairchild Semiconductor and CEO of Intel, wrote a paper describing the growth of the number of components per IC. He predicted that each year the number of components per IC would double. Ten years later he revised his prediction and changed it to a duplication every two years. [13]

This growth trend may have changed in recent years, but it is certain that the number of components per IC has grown exponentially over the years due to the successful miniaturization of transistors.

In solid-state memories information is stored in binary form in unit memory cells (one bit per memory cell). Each memory cell is a circuit that contains from one to several MOS transistors, and sometimes other elements depending on the technology [14]. Figure 6 shows the schematic of a Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET).

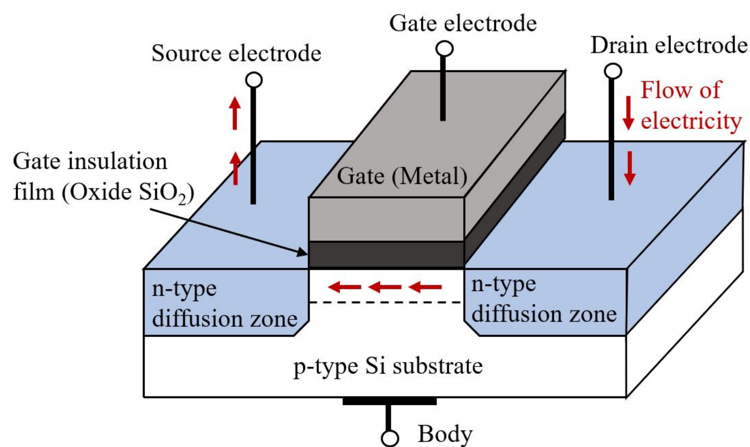


Figure 6: Schematic of an N-channel MOSFET

The current between drain and source can be regularly switched on and off by applying low voltage at the gate electrode.

Memory cells are organized in memory arrays. A peripheral circuitry is used to access the memory array for writing or reading. Figure 7 shows a component layout for a Static Random-Access Memory (SRAM) array.

To perform a write operation, the voltage on the corresponding *word line* is changed in order to activate that row of cells. Finally, to store the information in

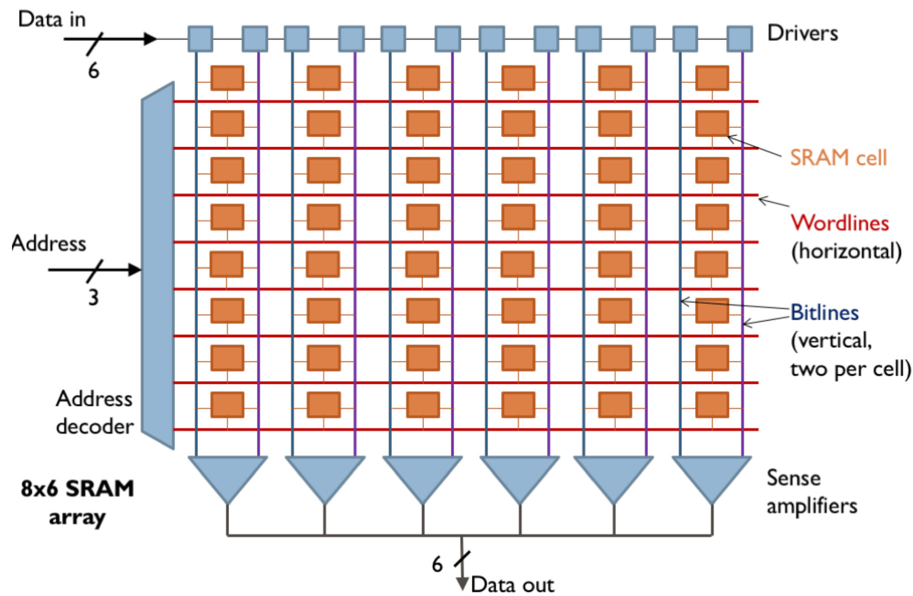


Figure 7: Component layout for a 8-location SRAM array where each location holds 6 bits of data. Figure taken from [17].

the cell, voltages are applied to the *bit lines*. When a read operation is performed, the information is retrieved by sensing the voltage on the bit lines with the sense amplifiers. [17]

Solid-state memories can be divided into two main categories: *volatile memories* and *non-volatile memories* (NVMs).

3.1 Volatile Memories

Volatile memories are defined as memories in which the information stored is lost when power supply to the device is discontinued. These memories work at high speed but, in general, have a high power consumption and lower storage capacity than NVMs. Two examples of this type of memory are Static Random-Access Memory (SRAM) and Dynamic Random-Access Memory (DRAM).

3.1.1 SRAM

SRAM uses bistable latching circuitry (flip-flop) to store each bit. An SRAM cell consists of six transistors: two of them serve to select the memory cell and the other four form two pairs of cross-coupled inverters which are used to define the memory state (see Figure 8).

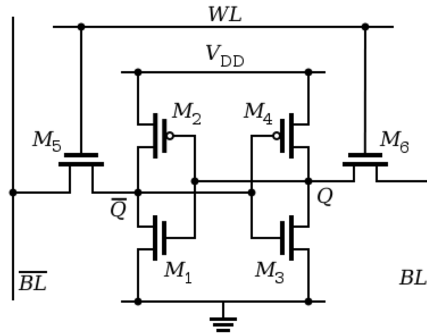


Figure 8: Electrical schematic for a six-transistor CMOS SRAM cell. Figure from the public domain.

The *word line* (WL) that is attached to the gates of M5 and M6 transistors allows access to the memory cell. These transistors also control the *bit line* nodes BL and $\overline{BL}$. M1, M2, M3 and M4 form the two cross-coupled inverters that are responsible for storing the bit value.

SRAMs operate very fast and have low power consumption and good endurance. However, since SRAM cells require a large number of transistors (6 per cell), they cannot reach the high densities of other technologies such as DRAM. This fact makes them comparatively more expensive to manufacture. [14]

SRAMs are used in microcontrollers, microprocessors and high-performance processors among others. Due to their high speed, they are usually used as caches.

The decrease in both the size of the memories and the operating voltage in recent years has increased the susceptibility of SRAM to radiation, especially to SEEs [15]. This is explained in more detail in section 4.

3.1.2 DRAM

A DRAM cell consists of a transistor and a capacitor (Figure 9). The capacitors can either be charged or discharged. These two states are used to represent the two values of a bit. The transistor isolates the capacitor when no write or read operations are performed, which is approximately 99% of the time [16]. The *word line* is connected to the gate node of the transistor and the *bit line* is connected to the drain node.

Due to leakage currents, the capacitor slowly discharges. For this reason, to maintain the memory state, the data in the capacitors are periodically *refreshed* (rewritten) by an external circuit [12]. DRAMs are called “dynamic” because of the required refreshing operation. The capacitors are also discharged when reading the cell so the cell must be rewritten after every read operation [14].

DRAMs operate slower than SRAMs and have higher power consumption. However, their simplicity allows them to reach high densities and small sizes, which makes them much cheaper per bit than SRAMs. These memories are used when a low-cost

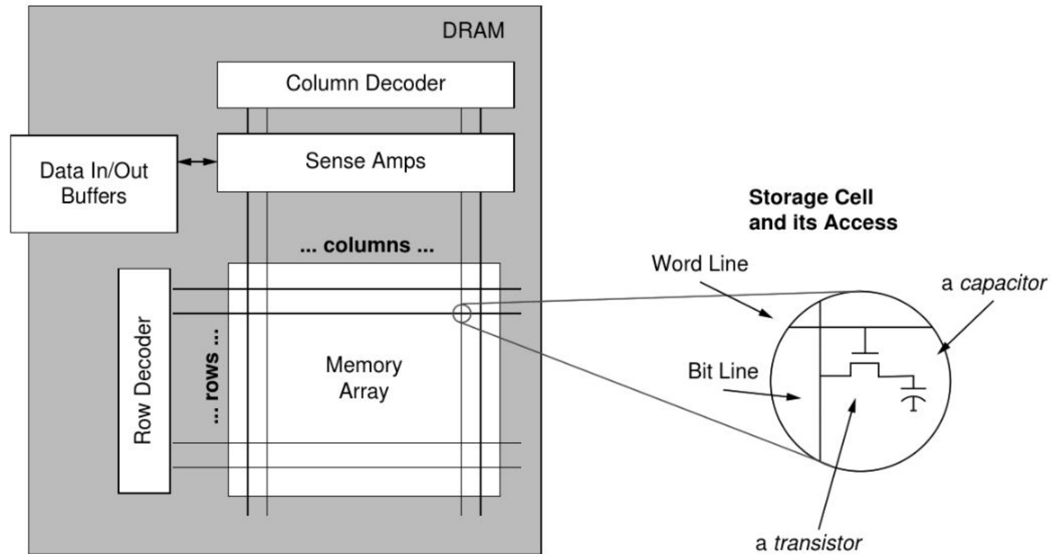


Figure 9: DRAM chip's memory array. The DRAM memory array is a grid of storage cells, where one bit of data is stored at each intersection of a row and a column. Figure taken from [12].

and high-capacity solution is required [14]. They can be found as main working memories in modern computers or portable devices, among others.

DRAMs, like SRAMs, have become more sensitive to radiation in recent years due to the decrease of the power voltage and memory cells size [16].

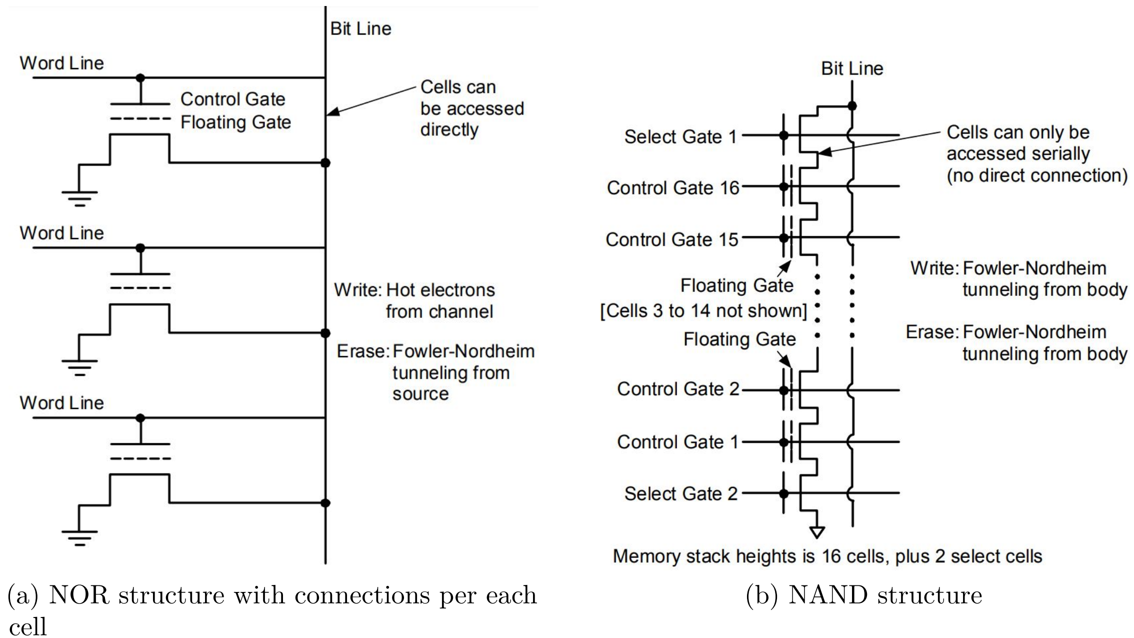
3.2 Non-Volatile Memories

NVMs can retain their contents even when the power supply is removed. These memories are often used in computers for the long-term data storage, have a low power consumption and a large capacity. Some examples of this type of memory are Electrically-Erasable Programmable Read-Only Memory (EEPROM), Flash Memory and Ferroelectric Random-Access Memory (FRAM). The last two types are described with more detail below.

3.2.1 Flash

Flash memory stores information in memory cells made up of *floating gate* MOSFETs. The current charge that flows between the source and the drain is controlled by two gates: a floating gate and a control gate. The floating gate is interposed between the control gate and the MOSFET channel and is insulated by an oxide layer, which makes that electrons placed on it are trapped. The quantity of charge trapped in the floating gate is used to set the logic state of the cell (in case of multi-level charge storage more than one bit per cell can be stored).

There are two possible structures of flash memory: NOR and NAND architecture. The NOR structure provides direct access to each cell, which allows rapid random access (about 100ns), but also contributes to an augment of the cell areas and, consequently, a higher cost (Figure 10 (a)) [18]. The NAND structure is more compact due to there is no direct access to each cell (Figure 10 (b)). In NAND structure, cells are written or read in blocks, which makes it slower than NOR structure.



(a) NOR structure with connections per each cell

(b) NAND structure

Figure 10: Flash memory structures. Figures taken from [19].

Their main drawbacks are that they have low endurance, require high operating voltages and are slow in the execution of reading and writing operations. This type of memory is sensitive to TID and SEE [18].

3.2.2 FRAM

FRAM construction is similar to DRAM but instead of a dielectric layer uses a ferroelectric layer, which makes it a non-volatile memory. After a reading operation, like in DRAM, FRAM memory cells need to be rewritten [14].

FRAM has a lower power usage, greater endurance and faster programming speed than the Flash memory. However, Flash memories have better storage density and cost than FRAMs. [21]

Data are stored in FRAM cells using polarization. This fact makes the memory cells immune to soft errors from injected energetic particles. However, these memories can be affected by radiation when energetic particles impact the control logic circuitry and peripheral circuit components, for example, the address decoders, I/O buffers, power switch, or sense amplifiers. [20]

4 Radiation effects on electronic compontents

Radiation produces three main types of effects on electronic components: displacement damage, Total Ionizing Dose (TID), and Single-Event Effects (SEE). The first two are known as *cumulative effects* since they arise progressively from cumulative radiation damage. On the other hand, SEEs are prompt phenomena which can produce transient or permanent effects.

These three effects can also be classified into two different groups: *ionizing damage* (TID and SEE) and *non-ionizing damage* (displacement damage). Ionizing damage occurs when semiconductor materials are bombarded with charged particles or photons with high enough energy to create an electron-hole pair. Non-ionizing damage is due to atomic displacements in the materials.

Most of the effects of radiation on electronic components in space come from ionizing damage. The table 3 shows the relationship between the two types of ionizing damage and the different sources of radiation in LEO.

Table 3: Sources of ionizing damage and their corresponding effects in near-Earth environment. Table from [4].

Radiation Source	**Particle Type**	**Effects on Electronics**
Solar energetic particles	Electrons	TID
	Protons	TID, SEE
	Heavy charged particles	SEE
Trapped particles	Electrons	TID
	Protons	TID, SEE
Galactic cosmic rays	Protons	TID, SEE
	Heavy ions	SEE

4.1 Cumulative effects

Displacement damage occurs when an energetic particle (neutron, electron, proton or heavy ion) transfers enough energy to move an atom of the material from its normal lattice position to a different one [3]. This movement creates lattice defects, as can be seen in Figure 11.

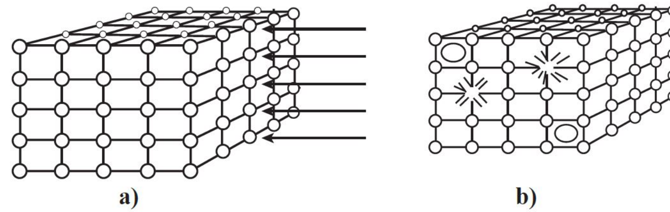


Figure 11: Displacement damage: **a)** Ordered atoms in a crystal before bombardment by electrons. **b)** Crystal with vacancies and interstitial atoms. Figure taken from [6].

The performance of semiconductors worsens with the increase in the number of defects in the crystalline lattice. The accumulation of defects can gradually degrade the material and, eventually, end in functional failure.

TID effects on electric components are due to exposure to ionizing radiation over a period of time. When ionizing radiation interacts with semiconductors and insulators such as silicon dioxide, it generates electron-hole pairs. These electrons and holes can recombine, but where holes have lower mobility (in insulators, for example), they may be trapped within the material, while electrons can more easily be transported away by electric fields.

The magnitude of TID effects depends on several factors such as the total radiation dose and its energy, the temperature during and after irradiation and the bias applied. The threshold voltage of a transistor, which is the voltage required to turn on the device, may be affected differently depending on the bias conditions during irradiation. [7]

Continuous exposure to ionizing radiation leads to an accumulation of trapped holes (positives charges) which modify the electric fields within the device and can ultimately lead to an increase in leakage currents. Over time, these leakage currents produce a continuous decrease in functionality until the device finally fails. The total ionization dose rate increases considerably for higher inclination orbits, for a given altitude. [6]

4.2 Single event effects

Single-event effects (SEEs) are transient or permanent effects that occur when a single energetic particle strikes an electronic component. The energetic particle, when passing through the material, loses its energy and ionizes the medium, generating electron-hole pairs, as shown in Figure 12. The ionization track left by the charged particle can interrupt the normal function of a circuit causing a SEE.

A wide range of single-event effects can be produced by the charge deposition of an energetic particle. These can be classified into two main categories: soft errors and hard errors. Soft errors can be corrected by reprogramming or restarting the device, while hard errors are non-recoverable since they produce physical damage to

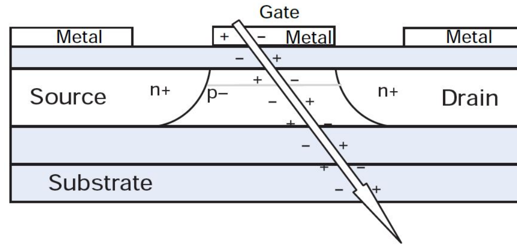


Figure 12: Deposition of energy in a semiconductor device. Figure taken from [6].

a circuit element.

SEE can have serious consequences for spacecraft such as loss of information, functional failure or loss of control. Due to a continuous advance in technology (smaller sizes of integrated circuits, higher speeds and more complex circuitry), the sensitivity to SEEs has increased, and the study of solutions to combat them has become of great importance [15].

Linear energy transfer (LET) is a parameter used to measure the amount of energy per unit path length that a charged particle deposits in the material. LET is given by:

$$LET = \frac{1}{\rho}\frac{dE}{dx}, \tag{1}$$

where ρ is the density of the material and $\frac{dE}{dx}$ is the rate of energy loss in the material. The units of LET are MeV/cm^2/mg. The total energy lost by a particle is given by the integral of LET over path length [34].

The LET of a particle varies as it travels through the material. Figure 13 shows the curve obtained using the average LET for a simulated particle in an ion beam versus the depth of the material. This curve can be of particular interest for understanding the interaction of a given energetic particle with the matter. In Figure 13, the point of maximum stopping power is called *Bragg peak* and in general occurs when the particle reaches energy near 1 MeV/nucleon [33].

The LET parameter allows interpreting the SEEs that can occur in a specific spatial environment.

The two types of SEE on which the present work is focused are described below: Single-Event Upsets (SEUs) and Single-Event Functional Interrupts (SEFIs).

4.2.1 SEU

A single-event upset is produced when an individual charged particle changes the state of a circuit. It can cause transient voltage and current pulses in an analog device or circuit, or one or more bit flips in memory cells or registers [6]. SEUs are soft errors and non-destructive effects since the device can continue to perform normally after a restart or by reprogramming the circuit into its correct logical state.

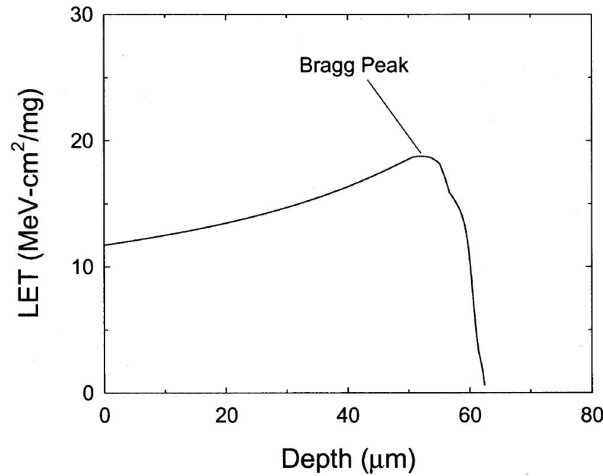


Figure 13: Linear energy transfer (LET) versus depth curve for 210-MeV chlorine ions in silicon. Figure taken from [33].

There are two types of ionizing radiation that can end up producing SEUs: *direct ionization* by the incident particle itself, and *indirect ionization* caused by the secondary particles that are produced due to nuclear reactions between the incident particle and the struck device [33].

Direct ionization is mostly caused by heavy ions and is the main source of SEUs. The heavy ion passes through a semiconductor material and loses its energy as it generates electron-hole pairs. Figure 14 illustrates SEU mechanism in an SRAM cell.

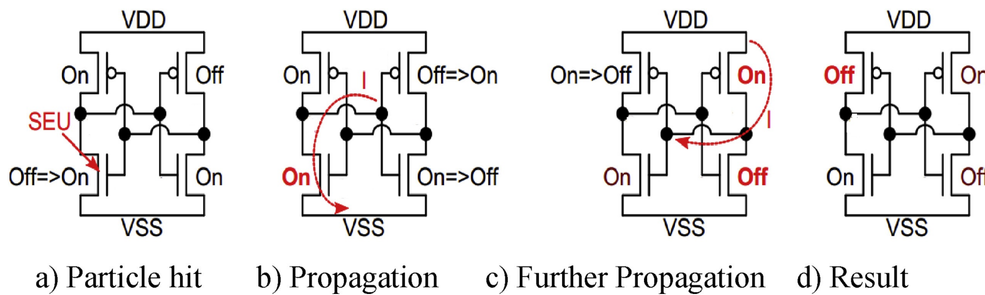


Figure 14: SRAM cell fault model. (a) A particle strikes a transistor in ”off” state changing it to ”on” state. (b) The collector of the left NMOS collects the charge generated and creates a current I that discharges the gate of the right transistor. (c) Right transistor toggles and enables current to charge gates of left transistors. (d) Left PMOS switches off and the circuit reaches a stable condition. Figure taken from [24].

Protons and neutrons can also produce significant upset rates through *indirect ionization*. When these particles enter the semiconductor lattice and collide with a target nucleus, nuclear reactions may occur. The products of those nuclear reactions, which can be much heavier than the original proton or neutron, can deposit energy along their paths causing an SEU [33].

The strike of a single energetic particle can result in one or more bit-flips. If it causes only one bit-flip, it is called Single-Bit Upset (SBU), whereas if it produces several bit-flips, it is known as Multiple-Cell Upset (MCU). If in an MCU two or more bits are involved in the same logical word, the event is called Multiple-Bit Upset (MBU). Figure 15 shows examples of one of the most common types of MCU, consisting of a cluster of a few bit-flips (up to 10-20 in some cases) that are topologically close and appear simultaneously [25].

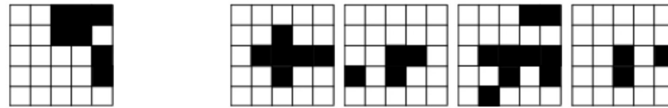


Figure 15: Example of a common type of MCU cluster. The white pixels represent non-corrupted cells and the black pixels represent upset cells. Figure taken from [25].

As the size of memory technologies is reduced, MCU rate increases since more memory cells fall under the footprint of a single energetic particle strike [26].

4.2.2 SEFI

Single-event functional interrupts (SEFIs) are produced when an ion strike triggers an integrated circuit (IC) test mode, a reset mode, or some other mode that causes the IC to temporarily lose functionality [22]. This type of single event effects can differ from one device to another since they are dependent on the design of the device's peripheral circuitry. SEFIs are more likely in memory chips with complex control circuits due to the increase in the circuit area exposed to the particles flux [23].

A SEFI event may lead, for example, to the cluster shown in Figure 16. A cluster like that may be the result of a temporary failure of the memory's I/O data buffers or synchronisation circuitry [27].

Figure 16: Example of a SEFI cluster, as it appears on physical (top) and logical (bottom) bitmaps. Figure taken from [27].

5 Error Detection And Correction codes

Error Detection and Correction (EDAC) codes, also known as Error Control Codes (ECCs), are codes designed to detect and correct errors in read or transmitted data. This is achieved by adding a controlled redundancy into the information to be protected.

This discipline started being studied in the late 1940s by Claude Shannon and Richard W. Hamming, who worked in the problem of error control on noisy channels. Both of them worked for Bell Telephone Laboratories and were interested in how to correct errors that appeared in the messages sent over long telephone lines and that were due, among other reasons, to lightning and crosstalk [28]. Since then, the study of this field has grown exponentially and has become more necessary due to advances in, for example, digital communication, space exploration and computer storage among other applications.

Although Shannon and Hamming were contemporaries, they used different techniques to address the problem of error control on noisy channels. Shannon studied it from a statistical/existential point of view, whereas Hamming used a combinatorial/-constructive approach. Because of these different approaches, they came to different conclusions: Shannon found the limits for ideal error control, while Hamming showed how to construct and analyze the first practical error control systems. [28]

The main goal of error-control coding is to provide reliable messages/data in an efficient manner. To be reliable, the received or read data must resemble the original data within narrow tolerances if the application requires it. To be efficient, error correction must be done in a small amount of time. It is necessary to find a balance between reliability and efficiency and, for each specific application case, to study what is more important to guarantee.

Since the early days of error-control coding, new codes and decoding methods have been developed, looking for ways to increase both reliability and efficiency. In addition, rapid advances in electronic and optical devices have allowed the implementation of powerful codes with high performance. [30]

The messages to protect consist of a set of symbols from a finite alphabet. The protection of the data is done by coding, which is the process of converting messages into codewords. This process is done in a way that every codeword is uniquely decodable, which means that each codeword is only related to one message. Codewords must be longer than messages, to add the redundancy symbols that make it possible to detect and correct errors. These errors can appear due to, for example, noisy channels in communication or bit-flips caused by radiation in satellite memories, among others.

Figure 17 shows the application of error-control coding in a digital communication system. In digital communication, the messages are encoded before being sent, in order to be able to correct errors that may appear due to a noisy channel. After the transmission through the channel, the codewords are corrected and decoded before

reaching the *sink*.

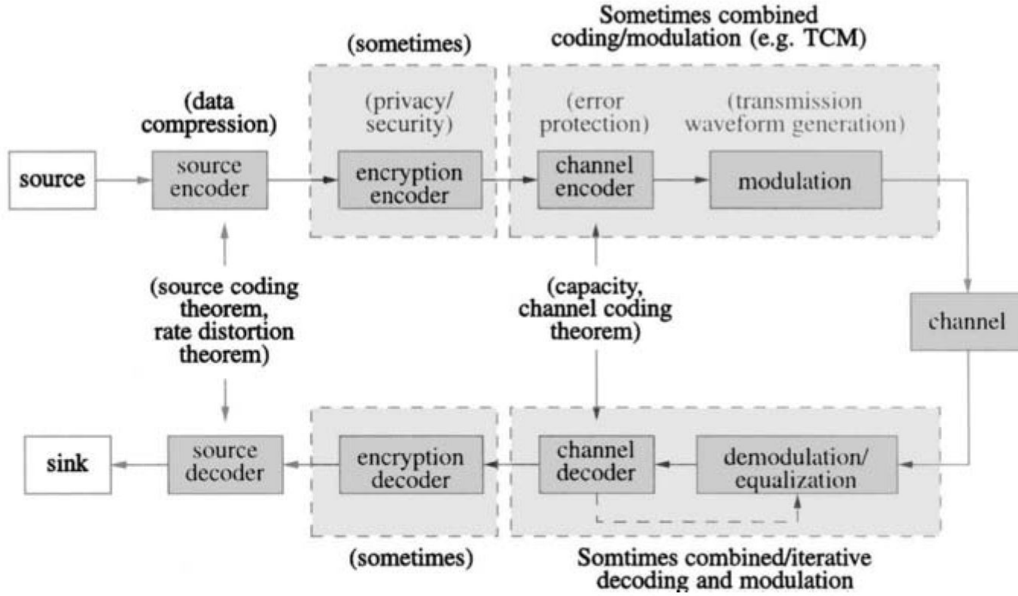


Figure 17: General framekwork for digital communications. Figure taken from [29].

As commented at the beginning of the section, there are several error-correction codes. These can be classified into two main categories: block coding and convolution coding. The present work will focus on block coding and especially in one type of codes: Hamming codes. For more information on convolutional coding, refer to [28], [29], [30].

Block coding is organized so that the message, generally in binary format, is grouped into blocks of k bits (message bits), constituting a set of 2^k possible messages. The encoder takes each block of k bits to convert it to a codeword of n bits, $n > k$. The decoder recovers the original message containing the information from the codeword. Block codes are denoted by $C_b(n, k)$: n is said to be the *length* of the code and k is the *dimension* of the code.

The *code rate* R_c measures the relationship between the codeword symbols and the message symbols. In the case of binary messages, it can be calculated as follows:

$$R_c = k/n, \tag{2}$$

where,

$$k : \text{number of message bits},$$
$$n : \text{number of codeword bits}.$$

The code rate is a measure of the level of redundancy. R_c serves as an indicator of the additional resources needed when codes are used: time to process more bits per word, more physical space needed to store the encoded information (*overhead*)... Therefore, R_c should be kept at a reasonable level and always be less than one, to add the necessary redundancy to correct and detect errors. [30]

Other important parameters for codes are the number of erroneous bits that can be detected and the number of those that can be corrected. In general, more errors can be detected than corrected, since to correct them, the position of the error is needed.

Finally, when selecting a code, the encoding and decoding speed of the code should be taken into account, since it is important that protection does not significantly alter the execution time of the processes.

There is a wide range of types of block codes: linear block codes, cyclic codes, Bose-Chaudhuri-Hocquenghem (BCH), Reed-Solomon (RS), Hadamard, Reed-Muller, etc. In Table 4 can be found a comparison in general terms of three EDAC codes commonly used in memories. However, it is important to note that within each type of code there may be many codes with different variations that can cause the code parameters to change (e.g. decoding speed, code rate, correction and detection capability).

Table 4: Comparison of some EDAC codes used in memories. Information from [31] and [35].

Characteristic	Hamming (SEC-DED)	RS (DEC-TED)	BCH (DEC-TED)
Check-bit Overhead Varies depending on the number of check bits.	7-32%	13-75%	13-75%
Error correction capability	Single Error Correction - Double Error Detection (SEC-DED)	Double Error Correction - Triple Error Detection (DEC-TED); Efficient for correlated errors (e.g. burst)	Double Error Correction - Triple Error Detection (DEC-TED); Efficient for uncorrelated errors (e.g. random errors)
Implementation	Simple to implement Binary based	Complex to decode and implement Symbol based (consists of a set of finite field elements)	Complex to decode and implement but simpler than RS Binary based
Decoding speed	High	Medium	Medium

As it can be seen in Table 4, a code that guarantees greater error correction capability, such as RS or BCH, can have as a disadvantage, a worse decoding speed (efficiency) and also higher check-bit overhead. [31]

5.1 Linear Block Codes

Linear block codes are the most easily implemented and therefore most widely used of the block codes. In this section, the basic concepts to generate this type of codes will be explained. However, the definition or where do they come from will be omitted. For a more deeper and theoretical definition, refer to [28], [29], [30].

A linear block code $\mathbf{C_b}[n, k]$ is composed of a set of M codewords $\{\mathbf{c_0}, \mathbf{c_1}, \mathbf{c_2}, ..., \mathbf{c_{M-1}}\}$, where each codeword is of the form $\mathbf{c} = (c_0, c_1, ..., c_{n-1})$. The encoding process consists of dividing the data to be protected into blocks or messages ($\mathbf{m} = (m_0, m_1, ..., m_{k-1})$) of length k, and mapping those blocks into codewords in $\mathbf{C_b}[n, k]$ (see Figure 18).

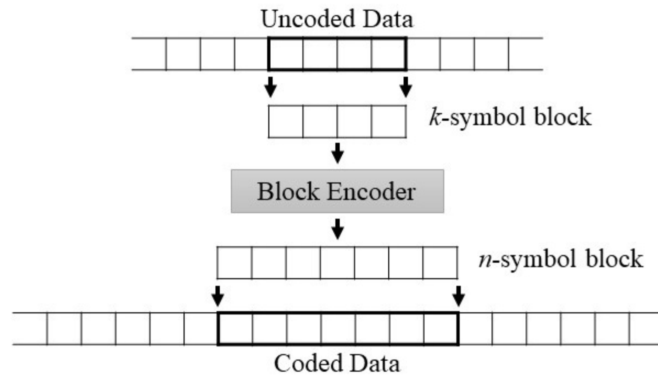


Figure 18: Encoding process for linear block codes. Figure taken from [28].

Table 5 shows an example of linear block code with $[n, k] = [5, 2]$ parameters.

Table 5: Linear block code with $(n, k) = (5, 2)$ parameters.

messages (m)	codewords (c)
00	00000
01	01011
10	10101
11	11110

Due to their configuration, a characteristic of linear block codes is that the sum of any of two codewords is also a codeword. This can be verified in Table 5: the modulo-2 addition of the second and third codewords results in the fourth one.

In a linear block code, there exists a set of k linearly independent codewords $\{\mathbf{g_0}, \mathbf{g_1}, ..., \mathbf{g_{k-1}}\}$, such that all possible codewords can be generated as a linear

combination of them. This set of codewords are known as **generator matrix G**:

$$\mathbf{G} = \begin{bmatrix} \mathbf{g_0} \\ \mathbf{g_1} \\ \vdots \\ \mathbf{g_{k-1}} \end{bmatrix} = \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{bmatrix}. \quad (3)$$

Using the generator matrix, the encoding process is very simple. For a given message $\mathbf{m} = (m_0, m_1, ..., m_{k-1})$, the corresponding codeword can be calculated as follows:

$$\begin{aligned} \mathbf{c} = (c_0, c_1, ..., c_{n-1}) = \mathbf{m} \cdot \mathbf{G} = (m_0, m_1, ..., m_{k-1}) \cdot \begin{bmatrix} \mathbf{g_0} \\ \mathbf{g_1} \\ \vdots \\ \mathbf{g_{k-1}} \end{bmatrix} \\ = m_0\mathbf{g_0} + m_1\mathbf{g_1} + \ldots + m_{k-1}\mathbf{g_{k-1}}. \end{aligned} \quad (4)$$

Another characteristic of these codes is that for each linear block code $[n, k]$ of dimension k, there exists a dual code $[n, n-k]$ of dimension $(n-k)$ [28]. The codewords in this dual code can also be formed using a set of $n-k$ linearly independent codewords $\mathbf{h_0}, \mathbf{h_1}, ..., \mathbf{h_{n-k-1}}$. The basis formed by this codewords is used to build the following matrix H:

$$\mathbf{H} = \begin{bmatrix} \mathbf{h_0} \\ \mathbf{h_1} \\ \vdots \\ \mathbf{h_{n-k-1}} \end{bmatrix} = \begin{bmatrix} h_{0,0} & h_{0,1} & \ldots & h_{0,n-1} \\ h_{1,0} & h_{1,1} & \ldots & h_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ h_{n-k-1,0} & h_{n-k-1,1} & \ldots & h_{n-k-1,n-1} \end{bmatrix}. \quad (5)$$

The H matrix of the dual code of C is known as the **parity check matrix**.

For every linear block code is satisfied that

$$\mathbf{G} \cdot \mathbf{H^T} = 0. \quad (6)$$

From the equation 6, it can be concluded that a vector $\mathbf{v}$ is a codeword of the code C, if and only if:

$$\mathbf{s} = \mathbf{v} \cdot \mathbf{H^T} = 0, \quad (7)$$

where $\mathbf{s}$ is called the **syndrome** [28].

The equation 7 can be used in order to detect errors. If $\mathbf{v}$ is a codeword, as stated by the equation, $\mathbf{s} = \mathbf{0}$. In case there is an error in the vector such that $\mathbf{v} = \mathbf{c} + \mathbf{e}$, then $\mathbf{s} \neq \mathbf{0}$ and an error has been detected:

$$\mathbf{s} = \mathbf{v} \cdot \mathbf{H^T} = (\mathbf{c} + \mathbf{e}) \cdot \mathbf{H^T} = \mathbf{cH^T} + \mathbf{eH^T} = 0 + \mathbf{eH^T} = \mathbf{eH^T}. \quad (8)$$

However, an error can be undetected if the error pattern is equal to a codeword. That happens when the number and positions of the errors are such that the transmitted codeword is converted into another codeword, so $\mathbf{s} = \mathbf{e} \cdot \mathbf{H^T} = 0$. Therefore, in a code $C_b[n, k]$, there are $2^k - 1$ undetectable error patterns [30].

5.1.1 Minimum distance

Before defining the minimum distance of a linear block code it is necessary to define the Hamming distance. The Hamming distance between two vectors is the number of positions in which the vectors differ. For example, if $\mathbf{c_1} = (0110010)$ and $\mathbf{c_2} = (1010011)$, then $d(\mathbf{c_1}, \mathbf{c_2}) = 3$.

The **minimum distance** d_{min} of a linear block code $C_b[n, k]$, is the minimum value of the distance between all the possible pairs of codewords of that code [29]:

$$d_{min} = min\{d(\mathbf{c_i}, \mathbf{c_j}); \mathbf{c_i}, \mathbf{c_j} \in \mathbf{C_b}; \mathbf{c_i} \neq \mathbf{c_j}\}. \tag{9}$$

The minimum distance d_{min} can also be calculated as the minimum number of columns of the parity check matrix of the code which when added together result in the all-zero vector $\mathbf{0}$ [30].

For a linear block code $C_b[7, 4]$ with the following parity check matrix

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix},$$

the addition of the 1st, 3rd and 7th columns results in the all-zero vector. Therefore, the minimum distance of the code is $d_{min} = 3$.

An (n, k) code with minimum distance d_{min} can be also denoted as an (n, k, d_{min}) code.

5.1.2 Detection and correction capabilities of a code

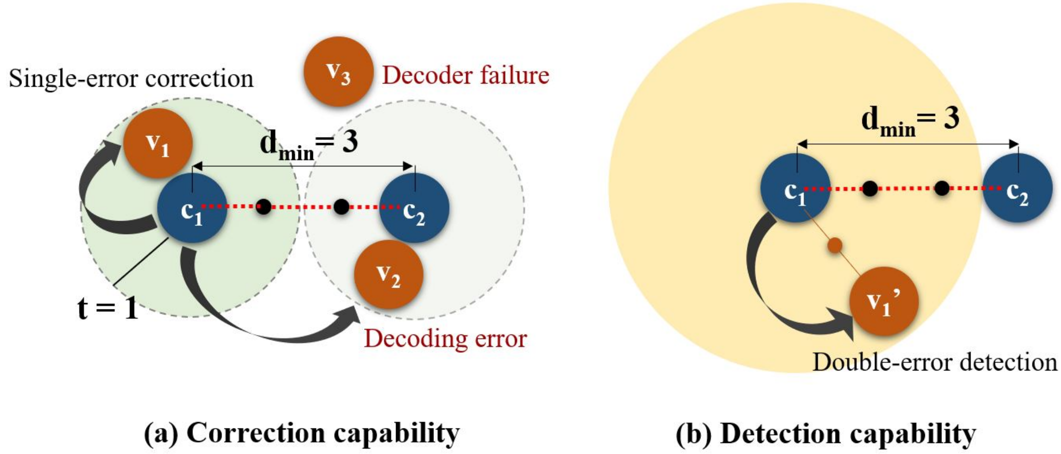
If fewer than d_{min} errors are introduced to a codeword $\mathbf{c}$, these can be detected, since $\mathbf{v} = \mathbf{c} + \mathbf{e}$ cannot be another codeword in this case. If d_{min} or more errors are introduced, then $\mathbf{v}$ can become a different codeword, making it not possible to detect that there are errors.

A code C can correct an error vector $\mathbf{e}$ introduced to a codeword $\mathbf{c_1}$ if for all the codewords, $\mathbf{c_1}$ is the unique closest codeword to $\mathbf{v_1} = \mathbf{c_1} + \mathbf{e}$. Therefore, the number of errors that an (n, k, d_{min}) code can correct is

$$t = \left\lfloor \frac{d_{min} - 1}{2} \right\rfloor, \tag{10}$$

where the notation $\lfloor x \rfloor$ means to take the greatest integer $\leq x$ [29].

Figure 19 shows a graphic representation of the concepts explained above for a code with $d_{min} = 3$ and $t = 1$. If the number of errors introduced in a codeword is greater than the correction capability $\mathbf{t}$, either a *decoding error* or a *decoder failure* may occur. A *decoding error* is produced when the resulting vector happens to be in the correctable area of a different codeword (see $\mathbf{v_2}$ in Figure 19(a)). A *decoder failure* is produced when the resulting vector does not fall into any of the decoding spheres (see $\mathbf{v_3}$ in Figure 19(a)). [29]



(a) Correction capability **(b) Detection capability**

Figure 19: Representation of the detection and correction capabilities of a code $C_b[n, k, d_{min}]$ with $d_{min} = 3$. $\mathbf{c_1}$ and $\mathbf{c_2}$ are codewords of C_b. The code is capable of single-error correction **or** double-error detection, but it can not do both at the same time.

5.1.3 Systematic codes

A code $C_b[n, k]$ is **systematic** if the message symbols can be found unchanged in the codeword. A code can be rewritten in order to transform it into a systematic one. The generator matrix of a systematic code is written in the form

$$G = \begin{bmatrix} P & I_k \end{bmatrix} = \begin{bmatrix} p_{0,0} & p_{0,1} & \dots & p_{0,n-k-1} & 1 & 0 & 0 & \dots & 0 \\ p_{1,0} & p_{1,1} & \dots & p_{1,n-k-1} & 0 & 1 & 0 & \dots & 0 \\ p_{2,0} & p_{2,1} & \dots & p_{2,n-k-1} & 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \\ p_{k-1,0} & p_{k-1,1} & \dots & p_{k-1,n-k-1} & 0 & 0 & 0 & \dots & 1 \end{bmatrix}, \tag{11}$$

where P generates the check symbols and I_k leaves the message/data symbols unchanged: $\mathbf{c} = \mathbf{m} \cdot \begin{bmatrix} P & I_k \end{bmatrix} = \begin{bmatrix} \mathbf{p} & \mathbf{m} \end{bmatrix}$.

If C is in the systematic form, the parity check matrix can be written as

$$H = \begin{bmatrix} I_{n-k} & -P^T \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & -p_{0,0} & -p_{1,0} & \dots & -p_{k-1,0} \\ 0 & 1 & 0 & \dots & 0 & -p_{0,1} & -p_{1,1} & \dots & -p_{k-1,1} \\ 0 & 0 & 1 & \dots & 0 & -p_{0,2} & -p_{1,2} & \dots & -p_{k-1,2} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 & -p_{0,n-k-1} & -p_{1,n-k-1} & \dots & -p_{k-1,n-k-1} \end{bmatrix}. \tag{12}$$

Depending on the application for which the protection is needed, a systematic coding scheme may be necessary to keep the information symbols visible. For example, if the code has to protect the data residing in a memory, it may be necessary to use a systematic encoding, so that data can be continuously fetched and used without unnecessary decoding operations.

However, in some cases, a non-systematic code may be a better solution. For

example, in applications involving sequential decoding, a non-systematic code may offer a lower undetected error probability. [32]

The systematic coding scheme is not necessary when the codewords are not accessed directly but are decoded before being used. For example, in a communication system, the messages are delivered to the EDAC encoder, which calculates the check bits. Then, the codewords are transmitted through the channel and given to the EDAC decoder. After checking the codewords and correcting possible errors, the data is decoded and ready to use. Another application in which it is not necessarily needed a systematic code is a secondary storage system, such as a hard disk. In this case, the data are decoded when it is retrieved into a memory buffer for use. [35]

5.1.4 Modifications to Linear codes

Table 6 describes the different modifications that can be done to linear codes to obtain new codes with changes in some of the parameters.

Table 6: Modifications to a linear code $C_b[n, k, d]$. Information taken from [28] and [29].

Technique	Description	Scheme
Augmenting $[n, k+1, \leq d]$	Consists of adding codewords to C_b. Can be done by adding rows to the generator matrix.	$\mathbf{G}' = \begin{bmatrix} g_{0,0} & \cdots & g_{0,n-1} \\ \vdots & \ddots & \vdots \\ g_{k-1,0} & \cdots & g_{k-1,n-1} \\ 1 & \dots & 1 \end{bmatrix}$
Expurgating $[n, k-1, \geq d]$	Consists of deleting codewords of C_b. Can be done by taking away rows of the generator matrix.	$\mathbf{G}' = \begin{bmatrix} g_{0,0} & \cdots & g_{0,n-1} \\ \vdots & \ddots & \vdots \\ g_{k-2,0} & \cdots & g_{k-2,n-1} \end{bmatrix}$

Extending $[n+1, k, d+1]$	Consists of adding extra check bits to C_b. Can be done by adding columns to the generator matrix. If d is odd, when extending the code, the capability of detecting errors is increased by one unity: $$d = 3, t = \lfloor 1.5 \rfloor = 1$$ $$d' = 4, t' = \lfloor 2 \rfloor = 2.$$	$\mathbf{G}' = \begin{bmatrix} g_{0,0} & \cdots & g_{0,n-1} & g_{0,n} \\ \vdots & \ddots & \vdots & \vdots \\ g_{k-1,0} & \cdots & g_{k-1,n-1} & g_{k-1,n} \end{bmatrix}$ $\mathbf{H}' = \begin{bmatrix} h_{0,0} & \cdots & h_{0,n-1} & 0 \\ \vdots & \ddots & \vdots & \vdots \\ h_{n-k-1,0} & \cdots & h_{n-k-1,n-1} & 0 \\ 1 & \cdots & 1 & 1 \end{bmatrix}$
Puncturing $[n-1, k, \leq d]$	Consists of deleting check bits of C_b. Can be done by taking away columns of the generator matrix.	$\mathbf{G}' = \begin{bmatrix} g_{0,0} & \cdots & g_{0,n-2} \\ \vdots & \ddots & \vdots \\ g_{k-1,0} & \cdots & g_{k-1,n-2} \end{bmatrix}$
Lengthening $[n+1, k+1, d']$	Lengthening is a combination of extending followed by augmenting.	$\mathbf{G}' = \begin{bmatrix} g_{0,0} & \cdots & g_{0,n-1} & g_{0,n} \\ \vdots & \ddots & \vdots & \vdots \\ g_{k-1,0} & \cdots & g_{k-1,n-1} & g_{k-1,n} \\ g_{k,0} & \cdots & g_{k,n-1} & g_{k,n} \end{bmatrix}$
Shortening $[n-1, k-1, d']$	Shortening is a combination or expurgating followed by puncturing. When a code is shortened by deleting all codewords containing a zero in a position i and puncturing in the i_{th} coordinate, $d' \geq d$, since only zeros have been removed. Depending on how this operation is carried out, the minimum distance of the code may be increased ($d' > d$), which increases the code detection capability [32].	$\mathbf{G}' = \begin{bmatrix} g_{0,0} & \cdots & g_{0,n-2} \\ \vdots & \ddots & \vdots \\ g_{k-2,0} & \cdots & g_{k-2,n-2} \end{bmatrix}$

5.1.5 Binary Hamming codes

Binary Hamming codes $H[n, k, d_{min}]$ are linear block codes characterized by the following parameters:

$$\begin{aligned}
&\text{Length :} && n = 2^m - 1, \\
&\text{Number of message bits :} && k = 2^m - m - 1, \\
&\text{Number of check bits :} && n - k = m, m \geq 2, \\
&\text{Error-correction capability :} && t = 1, d_{min} = 3,
\end{aligned} \tag{13}$$

where m is the number of check bits. [28]

The parity check matrices for a Hamming code of length $(2^m - 1)$ can be constructed easily by using as columns all nonzero binary m-tuples. For example, a $H[7, 4]$ code is defined by the parity check matrix

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}.$$

Due to their simplicity, Hamming codes have a high coding and decoding speed. These codes are capable of correcting 1-bit error in the block or detecting up to 2-bit errors.

As described in subsection 5.1.4, when extending a code which has odd minimum distance the capability of the code to detect errors is increased by one unity. Therefore, since all Hamming codes have $d = 3$, extending them makes possible to have single-error correction and, at the same time, double-error detection (SEC-DED) (see Figure 20).

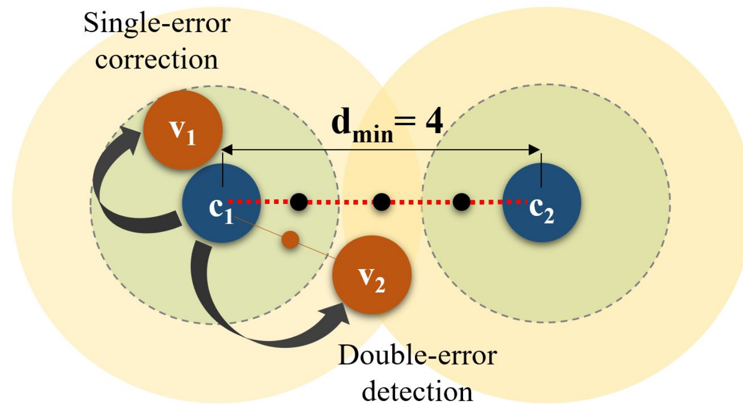


Figure 20: Scheme of a SEC-DED code with $d_{min} = 4$.

6 Memory protection with EDAC

Over the years, different techniques have been developed to increase the reliability of memory chips and reduce transient errors and permanent faults. Depending on the phase of the development cycle, two different strategies can be used: *fault avoidance* and *fault tolerance*.

Fault avoidance strategies are used during the design phase and their objective is to prevent the occurrence of faults. Some examples of fault avoidance are the improvement of the materials and the devices used.

Fault tolerance strategies are used during the execution time. These techniques use redundancy to detect and recover from faults and can be implemented through hardware, software or a combination of both. Some examples of redundancy are Duplication With Comparison (DWC), Triple Modular Redundancy (TMR) [31] or Error Detection and Correction (EDAC), which can be implemented with hardware or software.

6.1 Hardware EDAC

When EDAC codes are implemented in hardware, the memory bus architecture is extended to accommodate extra check bits. Another element necessary to implement hardware EDAC is the encoding/decoding circuitry that allows detecting and correcting memory errors [35].

Figure 21 shows a basic scheme of hardware implemented EDAC. As it can be seen, the check bits are added at the end of each data word, which is referred to as *horizontal code*.

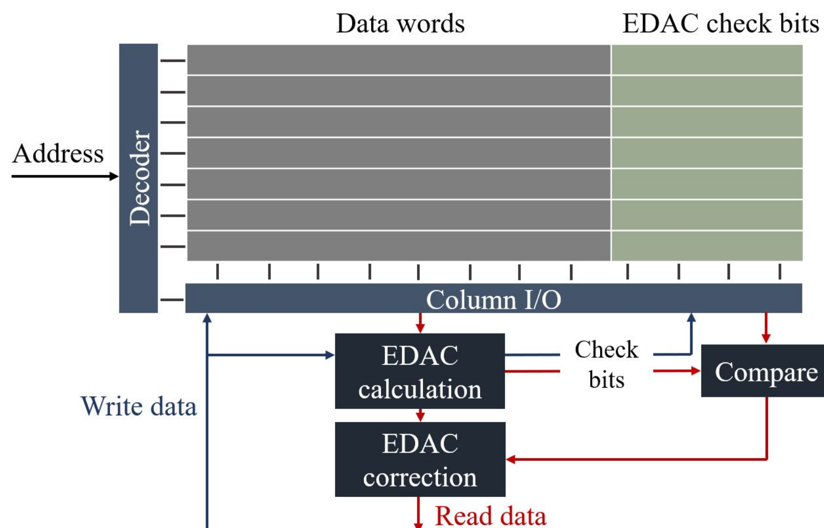


Figure 21: Basic scheme of hardware EDAC. Figure adapted from [31].

Hardware EDAC checks all the data that are read from memory in every read

operation, which makes it a highly reliable solution. Depending on the application, hardware EDAC may also perform what is known as *periodic scrubbing*. *Periodic scrubbing* consists in periodically reading all the data in the memory and correcting the errors. This is done to avoid the accumulation of errors, which could lead to the presence of several non-correctable errors.

The main drawback of implementing EDAC via hardware is that it is a very expensive solution. The high cost of hardware EDAC is due to the fact that it requires additional hardware components, extra power and extra area and shielding [31].

The fact of checking the data in every read operation makes hardware EDAC a highly reliable solution. In addition to this, hardware EDAC maintains good reliability both in low and high-radiation environments. [35]

6.2 Software EDAC

Software EDAC, unlike hardware EDAC, is a low-cost solution and has the flexibility to implement different and more complex codes. However, it has lower reliability than hardware EDAC, especially in high-radiation environments. Another drawback is that with software EDAC, extra memory accesses are needed to fetch the check bits while with hardware EDAC, these are fetched from the memory at the same time the corresponding data bits are accessed [35].

Some of the requirements needed to implement memory protection with software EDAC are described below.

6.2.1 Scrubbing

The operation of reading the bits from a memory block, correcting any error found and writing the bits back is referred to as *scrubbing*. The time between two successive scrub operations is known as *scrubbing interval*.

When software EDAC is implemented to protect main memories, it is not possible to check each word in every read operation as in hardware EDAC, since this would have a great overhead in the program execution time [35]. Therefore, software EDAC performs only *periodic scrubbing*.

As can be seen in Table 7, the reliability of software EDAC is closely related to the scrubbing interval selected and the upset rate of the environment [35].

From Table 7 different conclusions can be drawn. First, software EDAC reliability decreases drastically when the upset rate increases. For this reason, software EDAC should not be used to protect memories in high-radiation environments. Another fact that can be observed is that variations in the scrubbing interval have a greater effect on the reliability in high-radiation environments. In contrast, in low-radiation environments, this effect is minimal.

Table 7: Reliability sensitivity to scrubbing interval for a program Software EDAC and Hardware EDAC. The units of the upset rate are **upset/bit-cycle** (number of single-bit upsets in a cycle). This is derived from 10 upsets/Mbyte-day using a clock rate of 25 MHz. Table taken from [35].

Reliability for a Period of 1 Day				
Scrubbing interval	**SW EDAC, upset rate=** $5.52 \cdot 10^{-19}$	$5.52 \cdot 10^{-18}$	**HW EDAC, upset rate=** $5.52 \cdot 10^{-19}$	$5.52 \cdot 10^{-18}$
10 min	0.935506	0.513345	0.999999	0.999904
20 min	0.935504	0.513274	0.999998	0.999808
30 min	0.935503	0.513202	0.999997	0.999712
40 min	0.935502	0.513130	0.999996	0.999617
1 day	0.935319	0.503198	0.999862	0.996297

Memory can be divided into two types of information: code and data. The code segment contains instructions and when it is loaded, it rarely changes. On the other hand, data segments are constantly changing during execution. Therefore, protecting data segments with software EDAC may not be worthwhile since updating the check bits after each change can generate a great overhead [35].

6.2.2 Check bits location

The location of the check bits can follow two types of coding: *horizontal code* and *vertical code*.

Hardware EDAC, as stated before, is implemented using a horizontal code, which means that the check bits are located at the end of the data words (Figure 22(a)). In horizontal codes with software EDAC, the check bits are concatenated and located in a memory region separated from the data words (Figure 22(b)). If multiple bit-flips occur in different words but at the same bit position within each word (*bit-slice*), horizontal codes are capable of correcting them since they belong to different data words. On the contrary, if several bit-flips occur in the same word, they may be non-correctable.

Software EDAC can also be implemented using a vertical code. In vertical codes, the check bits are calculated over the data bits corresponding to one bit-slice of consecutive words in a block (Figure 22(c)). Vertical codes are easier to implement via software than horizontal codes since they can encode all the bit-slices in parallel while, in horizontal codes, several shifts and logical operations are required for encoding each word [35]. If multiple bit-flips occur in the same bit-slice, vertical codes are not able to correct them since, in this case, they belong to the same data word. On the contrary, if several bit-flips occur in the same word, vertical codes can correct them.

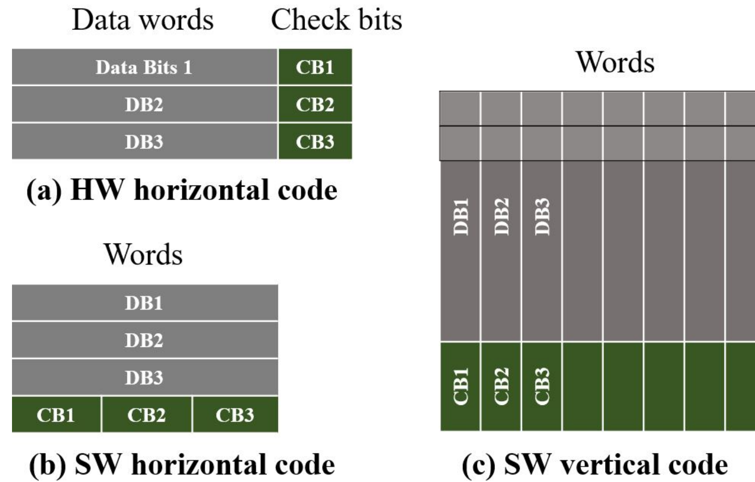


Figure 22: Horizontal and vertical code schemes. Figure adapted from [35].

6.2.3 Interleaving

Logical mapping of bits differs from physical mapping. Usually, the physical bits of a word are stored distant from each other. This is known as *bit interleaving*. Bit interleaving is used to increase performance and to optimize the layout of the cell I/O circuits [26]. This technique also increases the correction capability of EDAC towards MCU in horizontal codes. If a single energetic particle affects more than one adjacent bits, they may be correctable if bits are interleaved since they will correspond to different logical data words (see Figure 23).

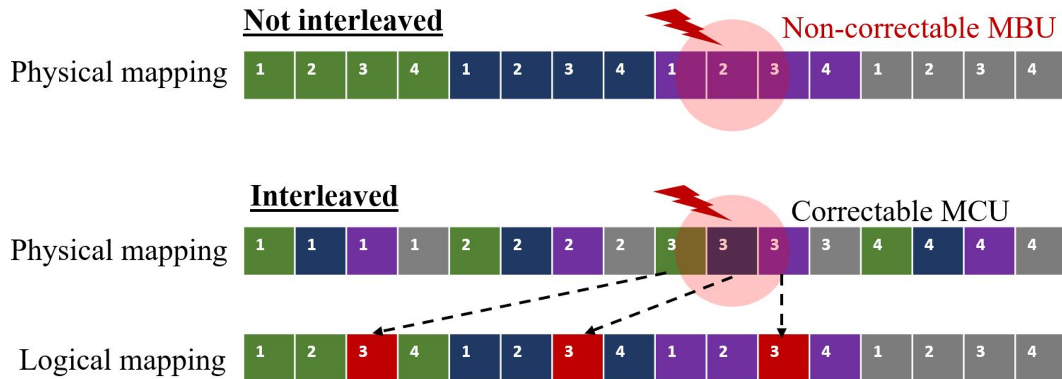


Figure 23: Simplified scheme of protection against MCU provided by bit interleaving.

However, bit interleaving is not useful to correct MCU in a vertical code. When using a vertical code, to handle MCU that happen in adjacent bits, these can be separated by interleaving the words that belong to a protected block, as shown in Figure 24 [35].

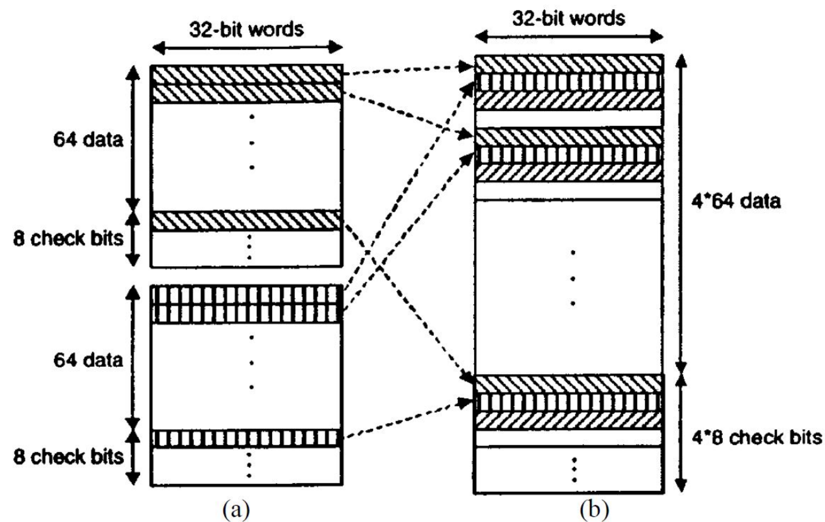


Figure 24: Simplified scheme of word interleaving. **(a)** Blocks of EDAC protected data and the corresponding check bits. **(b)** Location of the words using word interleaving. Figure taken from [35].

6.2.4 Protection of the scrubbing program

The scrubbing program is also vulnerable to errors. Therefore, it should be specially protected. Different solutions can be adopted to ensure its protection. One possible solution is to have a second copy of the EDAC program and do cross-checking using both copies of the program in order to ensure its correctness. Another solution is to have a second copy stored in a radiation-tolerant memory, e.g. FRAM, and in case an error in the program is found, copy the image of the scrubbing program again. [35]

7 FreeRTOS

FreeRTOS is a class of Real-Time Operating System (RTOS). The OS is used in real-time applications and is designed to be small enough to run on a microcontroller, although its use is not limited to this one. For this reason, FreeRTOS does not have the full functionality of an RTOS: it only provides the core real-time scheduling functionality, inter-task communication, timing and synchronization primitives [38]. FreeRTOS does not support networking, external hardware access and filesystem [40].

FreeRTOS can be structured as a set of independent tasks or threads. These threads are executed periodically and only one thread can be running at any point in time. The RTOS scheduler is responsible for deciding which thread should be running, taking into account each thread's priority and other factors such as semaphores, mutexes and queues. Since only threads have been used in the implementation, the other elements will not be explained. More information can be found in [38] and [39].

8 Experimental setup

The hardware and software tools used for the implementation are detailed in the following section.

The microcontroller used for the implementation is the STM32L432-KC. This microcontroller is based on the high-performance ARM® Cortex®-M4 32-bit RISC core operating at a frequency of up to 80 MHz [37].

The main reason for selecting this device was the large amount of available SRAM memory (64kB). Given that the objective of the project is the software implementation of memory scrubbing algorithms, having more memory facilitates the implementation and performance test of the algorithm.

In addition, the STM32L432-KC is provided with other features that are also useful for the implementation. The key features of the device are detailed in Figure 25.

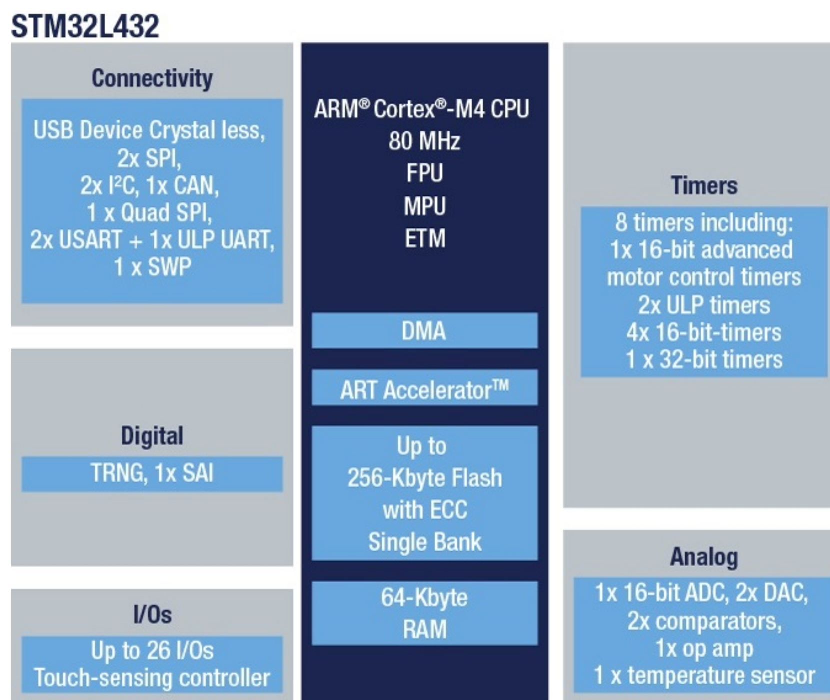


Figure 25: STM32L432Kx family device key features [37].

The board used is the STM32 Nucleo-32 board, which is shown in Figure 26. This board does not require any separate probe, as it integrates on-board ST-LINK/V2-1 debugger/programmer. Therefore, only a Micro-USB to USB cable is necessary to program and debug it.

The Integrated Development Environment (IDE) used in this project has been Keil® MDK.

For STM32 users there is a graphical tool called STM32CubeMX that allows

Figure 26: STM32 Nucleo-32 board.

a very easy configuration of the microcontroller, as well as the generation of the corresponding initialization C code for the ARM® Cortex®-M core. This tool has been used to set up the configuration required for the project.

The software used to get the results of the test has been STMStudio. This program reads and displays the variables of the application in real time and is a non-intrusive tool, so it preserves the real-time behaviour of the application.

Finally, a program called RealTerm has been used as a serial terminal to receive and send information through USART.

9 Software implementation

In order to protect the data and code regions of other tasks, the EDAC program needs privileges to read and write in those regions. Normally, the code region is located in flash. The first disadvantage of having the code to protect located in flash is that flash memory is programmed by blocks so, in order to correct a bit when an error occurs, the whole block where this bit is located has to be erased and rewritten. The second drawback is that flash memory can only be written a limited number of times: the endurance of STM32L432-KC is 10,000 cycles [37]. In case the error rate is low, this second disadvantage would not be a big problem since it would not be necessary to correct/write too often. Despite this, it is important to keep this fact in mind.

For these two reasons, it was decided to locate the programs to be protected in SRAM. Using SRAM both problems detailed before are solved. The disadvantage of using SRAM is that, as explained in section 3, SRAM is more sensitive to radiation.

In order to run the application from SRAM, a bootloader was programmed.

9.1 Bootloader and application processes

The structure of both bootloader and application are shown in the block diagram of Figure 27.

The function of the bootloader is to receive via USART the binary file containing the application, copy it into the SRAM and start the execution of the application. The application, in this implementation, is sent from the computer to the microcontroller.

When the application starts, hardware initialization is performed first. Then, the different threads are created (the behaviour of the threads will be explained in detail in the following subsection 9.3). Before starting the scheduler, the parity bits for the regions to be protected are calculated and stored in a specific memory region.

In case a non-correctable error is detected, a system reset will be performed. Once the bootloader is restarted, the application can be fetched again without errors from a secure source. As stated at the beginning of this subsection, in this implementation, the binary file of the application is stored in the computer and sent from there to the microcontroller. However, in an embedded application, the optimal place to store the binary file of the application would be an external radiation-tolerant memory, such as an FRAM, as discussed in section 3.

The most relevant code for the implementation of the bootloader and the application can be found in Appendix B and Appendix C respectively.

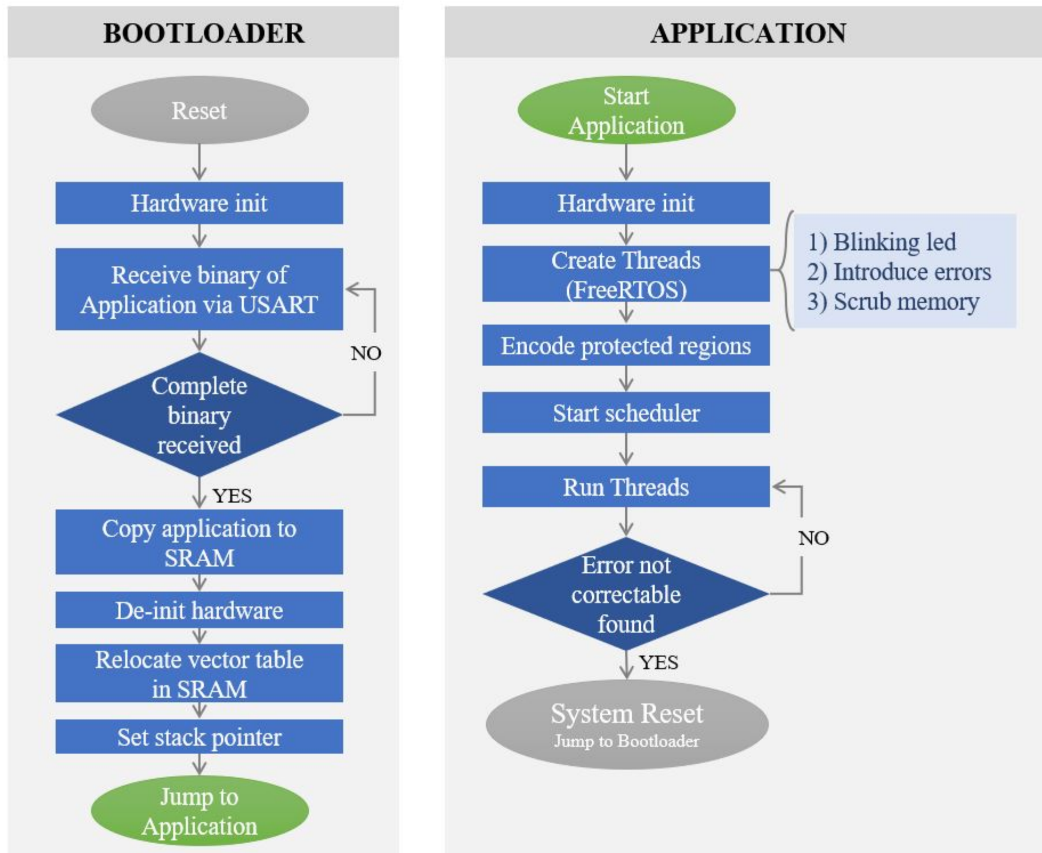


Figure 27: Block diagram of bootloader and application processes.

9.2 Memory map

Figure 28 shows the memory map of the implementation, and its details are described below.

The SRAM1 of the microcontroller has been divided into different regions: application code and RO data, Parity bits, Error function, Application RW data, and Bootloader RW data. The first two regions are the ones to be protected (25 kB).

The application does not use all the protected memory space (25 kB). 11 kB are used for RO code and 3.5 kB are used for RO data, including the parity bits. Part of the remaining space contains dummy RO data and their parity bits (4.4 kB), and the rest of the space is not used. The scrubbing algorithm protects all the used space, including dummy data. Therefore, only 18.9 kB are protected by the scrubbing algorithm and consist of 58.2% of RO code, 18.5% of RO data and 23.3% of dummy RO data.

In order to test the performance of the scrubbing algorithm, it was necessary to implement a function that introduces random errors in the protected memory space (18.9 kB) with a determined periodicity. This function has been located in a separate region after the parity bits region.

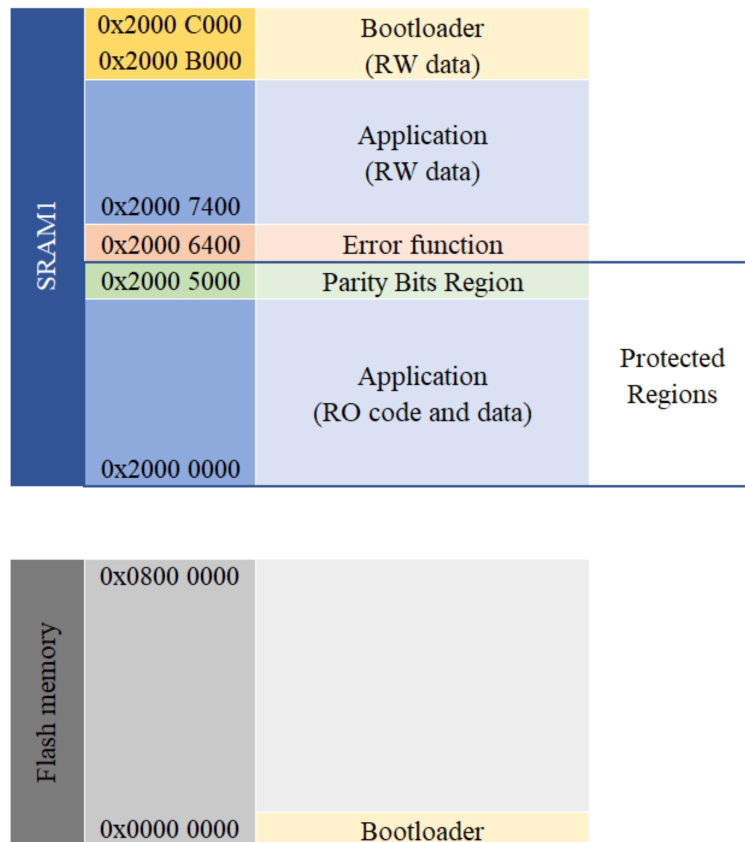


Figure 28: Memory map of the implementation

The last regions of SRAM1 are the RW data of both application and bootloader. It was decided not to protect those regions since as explained in section 6, RW regions are variable and therefore, if they are protected, the parity bits need to be recalculated after every write operation, which will negatively affect the performance.

In this implementation, the bootloader is located at the beginning of the flash memory and is the only thing located there. However, if desired, there could be other things running from flash, for example, the error function.

When Keil® MDK is used, the memory regions are configured in a file called *scatter file*. The scatter files corresponding to the implementation can be found in Appendices B.1 and C.1.

9.3 Threads

As described in section 7, a Real-Time Operating System (FreeRTOS) has been used in order to create a set of independent threads that inter-communicate. The following threads have been programmed: **`ThreadApplication()`**, **`ThreadError()`** and **`ThreadScrubbing()`**. The threads are described below and table 8 shows a summary of their characteristics: the priority, the time interval between two thread

executions (periodicity) and their main tasks.

Table 8: Characteristics of the threads implemented.

Thread	Priority	Periodicity	Function
Application	0	500ms	Blink a led
Error	0	ERROR_PERIODICITY	Call errorInProtectedRegion() Send message via USART
Scrubbing	3	SCRUB_PERIODICITY	Call HammingScrubMemory() Send message via USART If uncorrectable error, call HAL_NVIC_SystemReset()

The only function of **ThreadApplication()** is to make a led blink. This thread has been created to test the inter-communication between threads and to facilitate the testing of the programmed scrubbing algorithm.

ThreadError() calls the function errorInProtectedRegion(). This function introduces an error in one of the two protected regions (RO code and data of the application, or parity bits). This error consists of changing the value of a single bit in a memory address. The address and location of the bit to change are calculated using the function rand() of the C library math.h, which returns a pseudo-random number between 0 and 1. Since this function returns pseudo-random numbers, different seeds have been used to obtain different numbers in each iteration.

ThreadError() also sends a message through USART indicating in which of the protected regions the error was introduced. This thread is executed with a periodicity indicated in the constant variable ERROR_INTERVAL.

Finally, **ThreadScrubbing()** calls the function HammingScrubMemory(), which performs the detection and correction of the errors. This thread has the highest priority, so if other threads need to be executed at the same time, this one will run first. The time between two thread executions is indicated in the constant variable SCRUB_INTERVAL.

If the HammingScrubMemory() function detects an uncorrectable error, the function HAL_NVIC_SystemReset() is called and the following message is sent through USART: "Uncorrectable error detected, restarting bootloader...".

To test the correct behaviour of the scrubbing algorithm against MBUs, the function errorInProtectedRegion() was modified to introduce double errors. As expected, the algorithm is capable of detecting every MBU and, after detecting it, performs a system reset.

The threads implementation can be found in Appendix C.3.

9.4 Scrubbing Algorithm

9.4.1 Code selected: Hamming(39,32)

The EDAC code selected to implement the scrubbing algorithm was Hamming(39,32): each 32-bit protected word has 7 additional check bits, producing a 39-bit codeword. Since the parity bits are stored in 8-bit variables, the use of the check bits results in a 25% overhead in memory usage. One characteristic of Hamming codes is that they have a fast encoding/decoding speed, which serves to reduce the performance overhead in real-time systems. This characteristic was considered of special importance at the time of choosing the code to be used, above the ability to correct multiple errors.

The Hamming(39,32) code comes from Hamming(63,57), with parameters $[n, k, d] = [63, 57, 3]$. The final code has been obtained by applying two modification techniques: shortening and then extending.

As explained in section 5.1.5, Hamming codes have a minimum distance d of three between codewords, which makes them capable of correcting single errors or detecting double errors.

The original code has been first shortened from Hamming(63,57) to Hamming(38,32), with parameters $[38, 32, \geq 3]$. This first modification has been done because full words in SRAM are 32 bits long. By shortening the code, we are also reducing the error rate per word, so there is less probability of finding more than one error in the same word, which would not be correctable. Depending on how the shortening is done, the minimum distance may be increased, which is also beneficial (see subsection 5.1.4).

In order to have a SEC-DED code able to detect every double error in a word, Hamming(38,32) has been extended to Hamming(39,32). By extending the code, as explained in subsection 5.1.4, the minimum distance increases one unity. Therefore, the parameters of this code are $[39, 32, 4]$ and the code is able to detect two errors in a word and correct one.

As described in subsection 5.1.3, it is necessary to keep the protected data bits in their original form so that they are transparent to the rest of the system. For this reason, the code matrices have been modified to the systematic form.

The final schemes of the matrices that have been used are shown below:

$$G = \begin{bmatrix} P & I_k \end{bmatrix} = \begin{bmatrix} P_{7x32} & I_{32} \end{bmatrix} \tag{14}$$

$$H = \begin{bmatrix} I_{n-k} & P^T \end{bmatrix} = \begin{bmatrix} I_7 & P^T_{32x7} \end{bmatrix} \tag{15}$$

The complete matrices corresponding to schemes 14 and 15 can be found in Appendix A.

9.4.2 Implementation

The functions that have been programmed for the scrubbing algorithm are described below:

The function `HammingEncode()` calculates the check bits (*syndromes*) for all protected words in memory and stores them in the region *parity bits*. This function is called only once before starting the scheduler.

The function `HammingScrubMemory()` iterates through the addresses of the protected words and for each address calls the function `HammingDecode()`. This function, given a pointer to the protected word and a pointer to the corresponding syndrome, calls the function `HammingCheckSyndrome()` to check if there is an error and corrects it.

`HammingCheckSyndrome()` checks for an error in a codeword by multiplying the codeword by the parity check matrix **H** (7). If there is no error, the result is zero. If there is a single-error, the function returns a syndrome which is equivalent to one of the columns in **H** (8). This column is used to identify in which position of the codeword the error is located. If there is a double-error (uncorrectable), the function returns a value that does not correspond to any column of **H**.

If `HammingCheckSyndrome()` has returned a non-zero value, `HammingDecode()` looks up whether the value corresponds to any of the columns in **H**. If a match is found, the codeword is corrected by flipping the bit in the corresponding position. If no column matches the syndrome, it means an uncorrectable error has been found.

`HammingDecode()` returns `1` if an error has been corrected, `0` if the word has no errors and `0xFF` if an uncorrectable (double) error has been found.

`HammingScrubMemory()` returns the total number of errors corrected during scrubbing or, if a non-correctable error was detected, returns `0xFF`.

The functions developed for the implementation of the scrubbing algorithm can be found in Appendix C.4.

10 Results

10.1 Experimental results

Different configurations were used in order to test the performance of the scrubbing algorithm implemented. For each configuration, the application was executed 10 times. In each one of the iterations, a different seed was used for the calculation of pseudorandom numbers, so the errors introduced were different in every execution. The four configurations used are summarized in the following table 9:

Table 9: Characteristics of the testing configurations.

	Scrubbing algorithm	Scrubbing periodicity	Error periodicity
1	NO	-	1s
2	YES	10s	1s
3	YES	2s	1s
4	YES	1s	1s

As shown in Table 9, in all cases, single-errors were introduced with a constant periodicity of one second. This was done to evaluate the performance of the software EDAC when changing the scrubbing periodicity (time between two thread executions).

However, a typical value for the SRAM SEU rate in LEO is $\sim 6 \cdot 10^{-7}$ SEU/bit-day [36], which in the protected region in this project, corresponds to an SEU rate of ~ 0.093 SEU/day.

The functions developed for the introduction of errors can be found in Appendix C.4.

One of the main drawbacks of using a software EDAC to protect main memory is that single-bit errors can cause failures. Data in main memories are read and used constantly. Therefore, if a single-bit error is entered in an instruction and that data is read before the next scrub operation, the read instruction will be erroneous. If this happens, depending on where the error is located, it can cause failures. [35]

This problem was found when testing the project implementation. When this type of error occurs, the ARM Cortex-M core stops executing the current instruction, and usually branches to the hard fault handler function, where it enters an infinite loop. A hard fault is the default exception and can be triggered for two different reasons: an error during exception processing or an exception that cannot be managed by any other exception mechanism [42].

When a hard fault is triggered, a possible solution would be to catch the interruption, program a hard fault handler to try to fix the situation and continue with the execution. However, in some cases, it may not be possible to continue the execution from the instruction that caused the fault. For example, there is a

possibility that a corrupted instruction does not immediately cause a hard fault, but that the following instruction does. If this happens, tracing the corrupted instruction is almost impossible.

Another possibility after correcting the error, would be to restart the thread that was running in the moment of the fault. For systems running a real-time operating system, the task that created the fault may be terminated and restarted from the fault handler if needed [42].

In case the reading of the erroneous data does not cause a hard fault, there may be another problem. The bit-flip may remain imperceptible and, consequently, the processor may make a calculation error. Recovering from this condition could be possible only with software redundancy and restartable software modules.

The results obtained when errors are introduced with a periodicity of one second are plotted in the following graphs. As can be seen in Figure 29, the average duration of the executions is limited due to the problem with hard faults explained above.

As can be seen in Figure 29, the application with no EDAC lasts an average of $\sim$ 18 seconds, which means 18 errors have been introduced before failing. This may be for different reasons. As explained in subsection 9.2, not all the protected memory space contains relevant data: if the error is introduced in the dummy RO data, nothing will happen. If an error is introduced in the RO data, it may cause erroneous data, but not necessarily the application fails in this case. The errors introduced in the RO code (58.2% of the protected space) are the ones that, if read, can cause failures. However, there may be code that is not periodically read, but only at the beginning of the execution; an error introduced in this type of code will not affect the application.

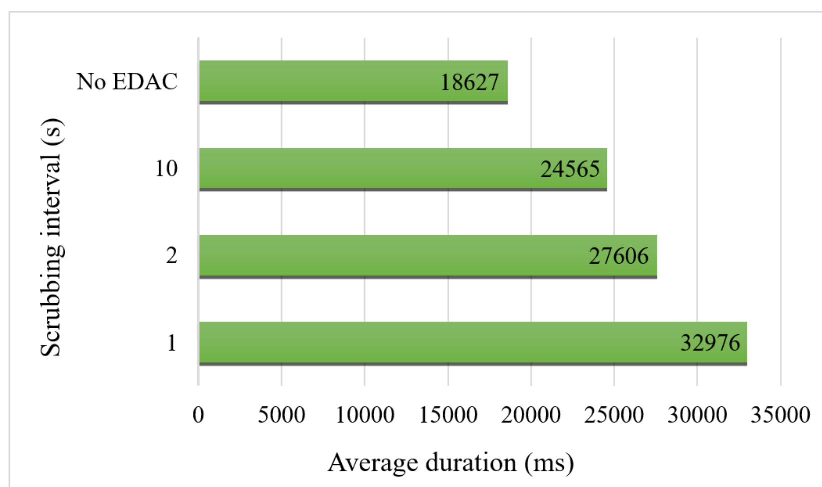


Figure 29: Average duration of the application for the established configurations.

The same test was executed 20 times for the no-EDAC configuration but removing the dummy RO data. Therefore, the errors were introduced only in relevant RO code and data. With this new configuration the mean time between failures caused

by accumulated errors was of 16987 ms.

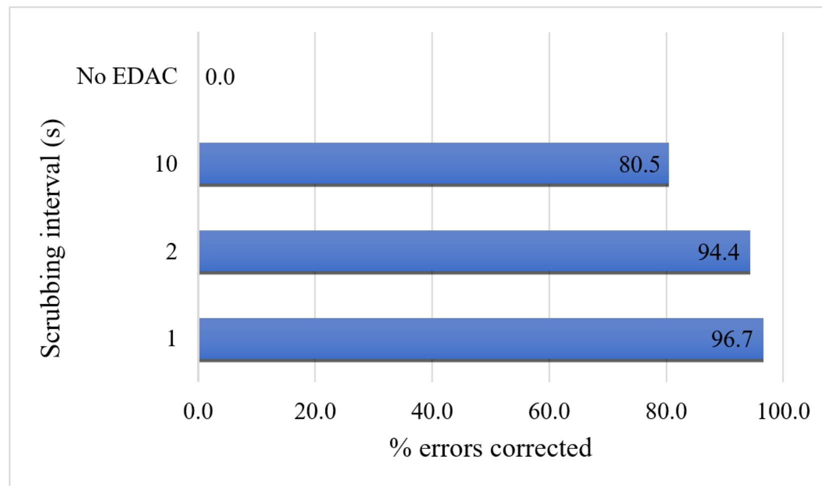


Figure 30: Percentage of errors corrected for the established configurations.

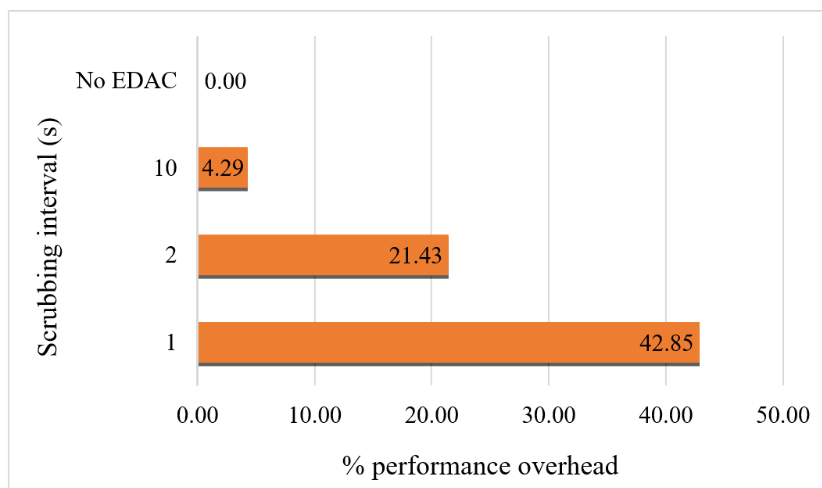


Figure 31: Performance overhead due to the scrubbing algorithm for the established configurations. The performance overhead was obtained by dividing the scrub duration by the time elapsed from the start of the scrub operation until the beginning of the next one.

From the results obtained, it can be concluded that the reliability of the software scrubbing algorithm is closely related to the scrubbing periodicity. As can be seen in Figures 30 and 29, the percentage of errors corrected and the average duration of the application increase when the scrubbing period is shorter, since there are fewer possibilities of making calculation errors due to corrupted data.

However, as illustrated in Figure 31, the use of a short scrubbing period results in a high overhead on system performance. Consequently, it is important to choose the scrubbing periodicity correctly in order to find a reliable solution without excessive performance overhead.

The scrub operation is done in 1.3713 seconds. Since the protected region has a size of 18.9 kB, the scrubbing speed of the algorithm is 13.78 kB/s.

As discussed previously, the use of software scrubbing algorithm for protecting the main memory is risky, since single-bit errors can cause failures if they occur after the last scrub operation and before the time of reading. Therefore, hardware EDAC provides greater reliability to protect the main memory and should be used whenever possible, since it checks and corrects all the data read from the memory.

On the other hand, software EDAC can be a good solution for secondary storage memories, where data are not constantly read and written from the memory [35]. In this case, the scrubbing algorithm can always be executed before a read operation and therefore, single-bit errors cannot corrupt the data.

10.2 Application of the results to a case study

The impact of using a software scrubbing algorithm to protect the memory of the FORESAIL-1 small satellite was analyzed.

FORESAIL-1 is a CubeSat mission of the Finnish Center of Excellence in Research of Sustainable Space. It hosts two payloads: the particle telescope (PATE) and a plasma brake experiment for plasma measurement and satellite deorbiting. PATE is a particle detector capable of measuring electron and proton fluxes, their energies and pitch angles. The avionics of this small satellite are designed and built in Aalto University; it will be launched in late 2019. [43]

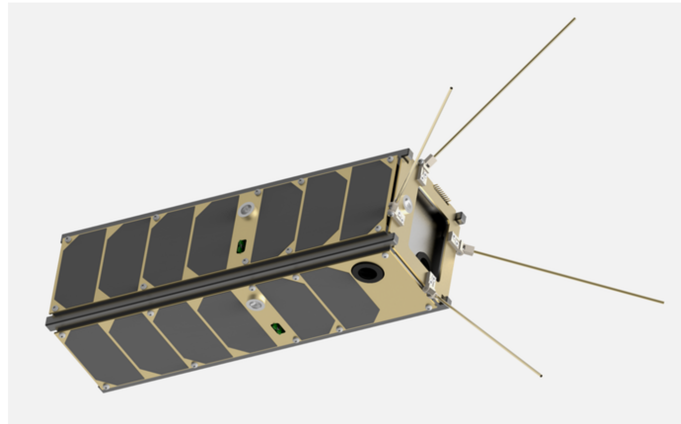


Figure 32: FORESAIL-1 satellite. Figure from [43].

The amount of code and RO data to protect on this satellite is approximately 391 kB ($\sim$ 383 kB of code and $\sim$ 8 kB of RO data). Therefore, the amount of memory required to store the corresponding check-bits is $\sim$ 97.75 kB.

Consequently, the total amount of protected memory (code, RO data and check-bits) is 488.75 kB. Taking into account an SEU rate in LEO of $\sim 6 \cdot 10^{-7}$ SEU/bit-day [36], the estimated SEU rate in the FORESAIL-1 protected memory is $\sim$ 2.40

SEU/day.

Since the scrubbing speed of the algorithm is 13.78 kB/s, the estimated time required to scrub the FORESAIL-1 protected memory would be ∼ 35.47 seconds.

Although the SEU rate is not high, the scrubbing interval must be low to increase the reliability and to avoid possible failures due to errors occurring between two scrub operations. Table 10 shows the performance overhead for different scrubbing intervals and the average of errors that can be present in the protected code given the scrubbing interval and an SEU rate of ∼ 2.40 SEU/day.

Table 10: Performance overhead for different scrubbing intervals.

Scrubbing interval	**Performance overhead**	**Average of errors**
10 min	5.91%	0.02 errors
20 min	2.95%	0.03 errors
30 min	1.97%	0.05 errors
1 hour	0.98%	0.1 errors

11 Summary

This final project manuscript presented the implementation of a software scrubbing algorithm to protect the main memory of an MCU against SEU. The scrubbing algorithm was implemented using Hamming(39,32) code. Hamming(39,32) is a SEC-DED code, which means that is able to detect double errors in a word and to correct single errors.

Hamming codes do not have high capability of error correction and detection. However, their fast decoding speed is especially important in real-time applications to reduce the performance overhead. Codes with a higher decoding speed should be selected above those capable of correcting multiple errors but with a slow decoding speed. There are several ways to reduce the possible occurrence of multiple errors and increase the code reliability, such as reducing the scrubbing interval and using bit/word interleaving.

When implementing an EDAC code with software, an important parameter to take into account is the error rate. Software-implemented EDAC should not be used in high-radiation environments since their robustness in those environments is especially low [35]. However, for LEO, where the expected SEU rate is low, software EDAC can be an appropiate solution.

Another important parameter in software EDAC is the scrubbing interval. Since software EDAC only performs periodic scrubbing, its reliability is closely related to it. A lower scrubbing interval increases the reliability of the algorithm since there are fewer possibilities of making calculation errors due to corrupted data.

One of the main drawbacks of software EDAC for protecting the main memory is that single-bit errors can cause failures. This is because the data of the main memory are read and used constantly. Therefore, if the error is entered in an instruction and that data is read before the next scrub operation, the read instruction will be erroneous and can lead to the triggering of a hard fault. Different solutions can be developed to reduce the impact of this type of errors. For example, a hard-fault handler could be implemented, which may permit the correction of the error and after, continue with the execution or, if it is not possible, restart the thread that was running when the error occurred.

The conclusion that can be drawn from the point above, is that protection of main memories should be done with hardware EDAC if possible, to avoid the risk of single-bit errors causing failures. However, software EDAC can be an acceptable/appropriate solution for secondary storage memories, where data are not constantly read and written and where the scrubbing algorithm can be executed always before a read operation [35].

The advantages of software EDAC with respect to hardware EDAC are that it is a low-cost solution and that there is more flexibility to implement different codes, adapting the implementation to the needs of each case.

References

[1] National Research Council (U.S) *Space Radiation Hazards and the Vision for Space Exploration.* Washington, DC, National Academies Press, 2006.

[2] Fortescue, P., Swinerd, G. and Stark, J. *Spacecraft Systems Engineering.* Wiley, 2011.

[3] Johnston, A., *Reliability and radiation effects in compound semiconductors.* Singapore, World Scientific, 2010.

[4] Nikicio, A. N., Loke, W. T., Kamdar, H. and Goh, C. H. "Radiation Analysis and Mitigation Framework for LEO Small Satellites," *Communication, Networks and Satellite, IEEE International Conference on,* pp. 59-66, 2017.

[5] Velazco, R., Fouillat, P. and Reis, R. *Radiation Effects on Embedded Systems.* The Netherlands, Springer, 2007.

[6] Pisacane, V. L. *Space Environment and Its Effects on Space Systems.* AIAA education series, American Institute of Aeronautics and Astronautics, 2008.

[7] Ma, T. P. and Dressendorfer, P. V. *Ionizing radiation effects in MOS devices and circuits* John Wiley & Sons, New York, 1989.

[8] Mewaldt, R. A., Mason, G. M., Gloeckler, G., Christian, E. R., Cohen, C. M. S., Cummings, A. C., Davis, A. J., Dwyer, J. R., Gold, R. E., Krimigis, S. M., Leske, R. A., Mazur, J. E., Stone, E. C., von Rosenvinge, T. T., Wiedenbeck, M. E., Zurbuchen, T. H. "Long-Term Fluences of Energetic Particles in the Heliosphere." Wimmer-Schweingruber RF, Joint SOHO/ACE *workshop on solar and galactic composition*, vol. 598. AIP conference proceedings, pp. 165-170, 2001.

[9] Mewaldt, R. A., Cohen, C. M. S., Labrador, L., Leske, R. A., Mason, G. M., Mallikarjun, D., Looper, M. D., Mazur, J. E., Selesnick, R. S., Haggerty, D. K. "Proton, helium, and electron spectra during the large solar particle events of October-November 2003." *Journal of Geophysical Research*, vol. 110, A09S18. doi:10.1029/2005JA011038, 2005.

[10] Advanced Composition Explorer (ACE) Mission Overview. *Mission Summary.* http://www.srl.caltech.edu/ACE/ace_mission.html. Accessed May 15, 2019.

[11] The Earth's trapped particle radiation environment. https://www.spenvis.oma.be/help/background/traprad/traprad.html. Accessed May 15, 2019.

[12] Jacob, B., W. Ng, S. and Wang, D. T. *Memory Systems: Cache, DRAM, Disk.* EEUU, Elsevier Inc., 2008.

[13] Mollick, E. “Establishing Moore’s law,” in *Annals of the History of Computing, IEEE,* vol. 28, no. 3, Jul. 2006, pp. 62-75.

[14] Bosser, A. L. *Single-Event effects of space and atmospheric radiation on memory components.* Doctoral dissertation, University of Jyväskylä, 2017.

[15] Srinivasan, J., Adve, S., Bose, P. and Rivers, J. “The impact of technology scaling on lifetime reliability,” in *Dependable Syst. and Networks, 2004 Int Conf. on.* pp.177-186, June 2004.

[16] Scheick, L. Z., Guertin, S. M. and Swift, G. M. “Analysis of radiation effects on individual DRAM cells,” in *Nuclear Science, IEEE Trans. on.* vol. 47, no. 6, Dec. 2000.

[17] Computation Structures. *L14. The Memory Hierarchy.* https://computationstructures.org/lectures/caches/caches.html. Accessed April 30, 2019.

[18] Gerardin, S. and Paccagnella, A. “Present and Future Non-Volatile Memories for Space,” in *Nuclear Science, IEEE Trans. on.* vol. 57, no. 6, pp. 3016-3039, Dec. 2010.

[19] Nguyen, D. N. and Scheick, L. Z. “TID, SEE and radiation induced failures in advanced Flash memories,” in *Proc. IEEE Radiation Effects Data Workshop,* pp. 18-23, Jul. 2003.

[20] Gupta, V., Bosser, A., Tsiligiannis, G., Zadeh, A., Javanainen, A., Virtanen, A., Puhner, H., Saigné, F., Wrobel, F. and Dilillo, L. “Heavy-ion radiation impact on a 4Mb FRAM under different test conditions,” in *IEEE, 15th European Conference on Radiation and Its Effects on Components and Systems,* Moscow, 2015.

[21] Nishi, Y. *Advances in Non-volatile Memory and Storage Technology.* Cambridge, Elsevier Ltd., 2014.

[22] Koga, R., Penzin, S. H., Crawford, K. B., and Crain, W. R. “Single event functional interrupt (SEFI) sensitivity in microcircuits,” in *Proc. 4th RADECS,* pp. 311-318, 1997.

[23] Pontarelli, S., Cardarilli, G. C. and Salsano, A. “Error correction codes for SEU and SEFI tolerant memory systems,” in *24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems,* 2009.

[24] Nihdin, T. S., Bhattacharyya, A., Behera, R. P., Jayanthi, T. and Velusamy, K. “Understanding radiation effects in SRAM-based field programmable gate arrays for implementing instrumentation and control systems of nuclear power plants,” *Nuclear Engineering and Technology,* Vol. 49, Issue 8, pp. 1589-1599, India, 2017.

[25] Dilillo, L., Bosser, A., Gupta, V., Wrobel, F. and Saigné, F. "Real-Time SRAM Based Particle Detector," in *Proc. Int. Workshop Adv. Sensors Interfaces (IWASI),* Gallipoli, Italy, Aug. 2005, pp. 58-62.

[26] Kim, J., Hardavellas, N., Mai, K., Falsafi, B. and Hoe, J. "Multi-bit error tolerant caches using two-dimensional error coding," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Mictroarchitecture.* IEEE Computer Society, pp.197-209, 2007.

[27] Bosser, A., Gupta, V., Tsiligiannis, G., Javanainen, A., Kettunen, H., Puchner, H., Saigné, F., Virtanen, A., Wrobel, F., Dilillo, L. "Investigation on MCU Clustering Methodologies for Cross-Section Estimation of RAMs," in *Nuclear Science, IEEE Transactions on,* vol. 62, no. 6, pp. 2620-2626, Dec. 2015.

[28] Wicker, S. B., *Error Correction Coding. Mathematical Methods and Algorithms.* New Jersey, Prentice Hall, 1995.

[29] Moon, T. K., *Error Control Systems for Digital Comunication and Storage.* New Jersey, John Wiley and Sons, 2005.

[30] Castiñeira, J. and Guy, P. *Essentials of Error-Control Coding.* England, John Wiley and Sons, 2006.

[31] Castano, V. and Schagaev, I. *Resilient Computer System Design.* Switzerland, Springer, 2015.

[32] Lin, S. and Costello, D. J. *Error Control Coding: fundamentals and applications.* New Jersey, Prentice Hall, 1983.

[33] Dodd, P. E. and Massengil, L. W. "Basic Mechanisms and Modeling of Single-Event Upset in Digital Microelectronics," *Nuclear Science, IEEE Transactions on,* vol. 50, no. 3, pp. 583-602, 2003.

[34] Schwank, J. R. *Basic mechanisms of radiation effects in the natural space radiation environment.* Sandia National Labs., Albuquerque, United States, 1994.

[35] Shirvani, P., Saxena, N. and McCluskey, E. "Software/Implemented EDAC Protection Against SEUs," *Proceedings, IEEE Aerospace Conference,* vol. 49, no. 3, pp. 273-284, 2000.

[36] Lovellette, M. N., Wood, K. S., Wood, D. L., Beall, J. H., Shirvani, P. P., and McCluskey, E. J. "Strategies for Fault-Tolerant, Space-Based Computing: Lessons Learned from the ARGOS Testbed," *Reliability, IEEE Transactions on,* vol. 5, no. 3, pp. 2109-2119, 2002.

[37] STMicroelectronics "STM32L432KB STM32L432KC DataSheet", 2019.

[38] STMicroelectronics "Developing Applications on STM32Cube with RTOS", https://www.st.com/content/ccc/resource/technical/document/user_manual/2d/60/ff/15/8c/c9/43/77/DM00105262.pdf/files/DM00105262.pdf/jcr:content/translations/en.DM00105262.pdf. Accessed December, 2019.

[39] FreeRTOS, https://www.freertos.org/RTOS.html. Accessed December, 2019.

[40] Rajesh, R. and Mathivanan, B. *Communication and power engineering.* Walter de Gruyter GmbH, Berlin, 2016.

[41] Radiation Belts with Satellites. https://www.nasa.gov/mission_pages/sunearth/news/gallery/20130228-radiationbelts.html. Accessed January 1, 2019.

[42] Application Note 209: Using Cortex-M3/M4/M7 Fault Exceptions. http://www.keil.com/appnotes/files/apnt209.pdf. Accessed April 4, 2019.

[43] Foresail-1 https://www.aalto.fi/en/spacecraft/foresail-1. Accessed May 21, 2019.

A Hamming(39,32) matrices

$$G = \begin{bmatrix}
0&0&0&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&0&1&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&1&0&1&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&0&1&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&1&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&1&0&1&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&1&1&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&1&1&1&1&1&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&1&1&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&0&0&1&0&1&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&0&1&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&0&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&1&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&1&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&1&0&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0\\
0&1&1&1&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&1&1&1&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&1&1&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&1&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
1&0&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0\\
1&0&0&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
1&0&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0\\
1&0&0&1&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0\\
1&0&0&1&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0\\
1&0&0&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&1&1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1
\end{bmatrix}$$

$$H = \begin{bmatrix}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&0\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&1&1&1&1&1&1&1&0&0&0&0&0&0&0&1&1&1&1&1&1&1&1&0&0&0&0&0&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&1&1&1&1&1&0&0&0&1&1&1&1&0&0&0&0&1&1&1&1&1&0&0&0&1&1&1&1\\
0&0&0&0&1&0&0&1&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&0&0&1&1&1&0&1&1&0&0&1&1\\
0&0&0&0&0&1&0&1&1&0&1&0&1&0&1&1&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1&0&1\\
0&0&0&0&0&0&1&1&1&1&0&1&0&0&1&1&1&1&0&1&0&0&1&1&0&0&1&0&1&1&0&1&1&1&0&1&0&0&0
\end{bmatrix}$$

B Bootloader code

B.1 Configuration files

B.1.1 (config.h)

The file *config.h* contains defines that are used in the scatter files *EDACbootloader.sct* and *APP.sct*. These defines correspond to the addresses and sizes of the different memory regions.

```
#ifndef __CONFIG_H
#define __CONFIG_H

#define BOOT_ROM_ADDRESS      0x08000000
#define BOOT_ROM_SIZE         0x00040000
#define BOOT_RAM_ADDRESS      0x2000B000
#define BOOT_RAM_SIZE         0x00001000

#define APP_ROM_ADDRESS         0x20000000
//Program
#define APP_ROM_SIZE            0x00005000
//Parity bits sections starts at the end of ROM
#define APP_PARITYBITS_ADDRESS (APP_ROM_ADDRESS + APP_ROM_SIZE)
//7 parity bits every 32 bits of data. Aproximated to 8.
#define APP_PARITYBITS_SIZE     (APP_ROM_SIZE / 4)

#define APP_ERROR_ADDRESS       (APP_PARITYBITS_ADDRESS +
    APP_PARITYBITS_SIZE)
#define APP_ERROR_SIZE          0x1000

#define APP_RAM_ADDRESS         (APP_ERROR_ADDRESS + APP_ERROR_SIZE)
#define APP_RAM_SIZE            (BOOT_RAM_ADDRESS - (APP_ERROR_ADDRESS
    + APP_ERROR_SIZE))

#define APP_TOTAL_SIZE          (BOOT_RAM_ADDRESS - APP_ROM_ADDRESS)

#endif
```

B.1.2 Scatter file (EDACBootloader.sct)

The scatter file *EDACBootloader.sct* determines how the memory layout is organized. The objects are allocated in the defined memory regions. The region `ER_APP` is not initialized since is the region in which the application will be copied. The regions `ER_IROM1` and `RW_IRAM1` are for the bootloader RO and RW data.

```
#! armcc -E -I C:\Users\ritar\Documents\FINALPROJECT\cubemx\FreeRTOS\
    APP_SRAM\MDK-ARM\cfg

#include "config.h"

; Memory Regions
```

```
LR_IROM1 BOOT_ROM_ADDRESS BOOT_ROM_SIZE  {
  ER_IROM1 BOOT_ROM_ADDRESS BOOT_ROM_SIZE  {
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
   .ANY (+XO)
  }
  ER_APP APP_ROM_ADDRESS UNINIT APP_TOTAL_SIZE  {
   *(NoInit)
  }
  RW_IRAM1 BOOT_RAM_ADDRESS BOOT_RAM_SIZE  {
   .ANY (+RW +ZI)
  }
}
```

B.2 Main function (main.c)

The main function contains the behaviour of the Bootloader as defined in Section 9.1.

```
int main(void)
{
  /* MCU Configuration------------------------------------------*/
  /* Reset of all peripherals, Initializes the Flash interface and the
    Systick. */
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_USART2_UART_Init();
  HAL_UART_Receive_DMA(&huart2, aRxBuffer, RXBUFFERSIZE);

  /* Infinite loop */
  while (1)
  {
    HAL_Delay(200);
    if(number_rxBytes >= TXBUFFERSIZE){
      number_rxBytes = 0;

      /* Jump to User define Application Address */
      print("Launching Application.");
      Bootloader_JumpToApplication();
      while(1)
      {
        ;
      }
    }
  }
```

```
}
```

B.3 Jump to Application function (bootloader.c)

```
void Bootloader_JumpToApplication(void)
{
  /* Get the application entry point (Second entry in the application
   Vector Table) */
  uint32_t  JumpAddress = *(__IO uint32_t*)(PROGRAM_START_ADDR + 4);
  pFunction Jump = (pFunction)JumpAddress;

  HAL_RCC_DeInit();
  HAL_DeInit();

  SysTick->CTRL = 0;
  SysTick->LOAD = 0;
  SysTick->VAL  = 0;

  /* Vector Table, MSP and Relocation in Internal SRAM (Application)
   */
  SCB->VTOR = PROGRAM_START_ADDR;
  __set_MSP(*(__IO uint32_t*)PROGRAM_START_ADDR);

  Jump();
}
```

In the file *bootloader.h* must be indicated the size of the application's binary file that has to be loaded:

```
/*** Bootloader Configuration
    ********************************************/
#define SET_VECTOR_TABLE        1       /* Automatically set vector
    table location before launching application */

#define PROGRAM_SIZE        18232                   /* Size of program
    to load*/
#define PROGRAM_START_ADDR        (uint32_t)0x20000000  /* Start
    address to write the program */
```

C Application code

C.1 Configuration files

C.1.1 (config.h)

Same as in Appendix B.

C.1.2 Scatter file (APP.sct)

The scatter file *APP.sct* determines the different memory regions required for the Application: The regions `ER_IROM1_APP` and `RW_IRAM1_APP` are for the Application RO and RW data. The region `RW_PARITYBITS_APP` is the region in which the check bits will be stored. The region `ER_ERROR_APP` is the region in which the functions to introduce errors *error.o*,the scrubbing functions *hamming.o* and the file *tasks.o* are allocated. These files are allocated here in order to not introducing errors in them.

```
#! armcc -E -I .\cfg

#include "config.h"

LR_IROM1_APP APP_ROM_ADDRESS APP_ROM_SIZE  {
  ER_IROM1_APP APP_ROM_ADDRESS APP_ROM_SIZE  {
   *.o (RESET, +First)
   *(InRoot$$Sections)
   .ANY (+RO)
   .ANY (+XO)
  }
  RW_PARITYBITS_APP APP_PARITYBITS_ADDRESS APP_PARITYBITS_SIZE  {
   *(syn)
  }
  ER_ERROR_APP APP_ERROR_ADDRESS APP_ERROR_SIZE {
   error.o
   tasks.o
   hamming.o
  }
  RW_IRAM1_APP APP_RAM_ADDRESS APP_RAM_SIZE  {
   .ANY (+RW +ZI)
  }
}
```

C.2 Main function (main.c)

```
int main(void)
{
  /* Memory section to be protected */
  sectionProgram.baseAddress = (uint32_t *)&Image$$ER_IROM1_APP$$Base;
  sectionProgram.lengthSection = (uint32_t)&
   Image$$ER_IROM1_APP$$Length;

  /* Start address where are located parity bits */
  baseSyndrome = (uint8_t *)&Image$$RW_PARITYBITS_APP$$Base;

  DummyArrays();

  /* MCU Configuration-----------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the
     Systick. */
```

```
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART2_UART_Init();

  /* Create the thread(s) */
  /* definition and creation of Application */
  osThreadDef(Application, ThreadApplication, osPriorityNormal, 0,
   128);
  ApplicationHandle = osThreadCreate(osThread(Application), NULL);

  /* definition and creation of ScrubbingMemory */
  osThreadDef(ScrubbingMemory, ThreadScrubbing, osPriorityRealtime, 0,
    128);
  ScrubbingMemoryHandle = osThreadCreate(osThread(ScrubbingMemory),
   NULL);

  /* definition and creation of ErrorFunction */
  osThreadDef(ErrorFunction, ThreadError, osPriorityNormal, 0, 128);
  ErrorFunctionHandle = osThreadCreate(osThread(ErrorFunction), NULL);

  /* Calculate parity bits of the memory section we want to protect */
  HammingEncode(sectionProgram.baseAddress, sectionProgram.
   lengthSection, baseSyndrome);
  srand(SEED);

  /* Start scheduler */
  osKernelStart();

  /* We should never get here as control is now taken by the scheduler
    */
  /* Infinite loop */
  while (1)
  {
  }
}
```

C.3 Thread functions (main.c)

All the threads created are written below.

```
/**
  * @brief Dummy Threads. Toggles a LED.
  * @param None
  * @retval None
  */
void ThreadApplication(void const * argument)
{
  /* Infinite loop */
  for(;;)
```

```
  {
      HAL_GPIO_TogglePin(GPIOB, LD3_Pin);
      osDelay(500);
  }
}

/**
  * @brief Scrubbing Thread. Calls the function to scrub the memory.
  *        performs a System Reset if an uncorrectable error has been
   detected.
  * @param None
  * @retval None
  */
void ThreadScrubbing(void const * argument)
{
  uint8_t tx_scrub[14] = "Scrubbing...\r\n";
  uint8_t tx_bootloader[56] = "Uncorrectable error detected,
   restarting bootloader...\r\n";

  //scrubbing interval
  TickType_t xLastWakeTime;
  const TickType_t xDelay = pdMS_TO_TICKS( SCRUB_PERIODICITY );
  xLastWakeTime = xTaskGetTickCount();

  /* Infinite loop */
  for(;;)
  {
    HammingScrubMemory(sectionProgram.baseAddress, sectionProgram.
   lengthSection, baseSyndrome);

    /* if uncorrectable error detected, bootloader should be restarted
    */
    if( uncorrectableError == 0xFF ){
        HAL_UART_Transmit(&huart2, tx_bootloader, sizeof(tx_bootloader
   ), 100);
        HAL_NVIC_SystemReset();
    }
    HAL_UART_Transmit(&huart2, tx_scrub, sizeof(tx_scrub), 100);

    /* The task should execute every SCRUB_PERIODICITY ms */
    osDelayUntil(&xLastWakeTime, xDelay);
  }
}

/**
  * @brief Error Thread. Calls the function to introduce errors in the
    memory.
  * @param None
  * @retval None
  */
void ThreadError(void const * argument)
{
  uint8_t tx_error[26] = "Error in program memory \r\n";
  uint8_t tx_error1[27] = "Error in syndromes block \r\n";
```

```
  uint8_t locationError = 0;

  /* Infinite loop */
  for(;;)
  {
    locationError = errorInProtectedRegion(sectionProgram.baseAddress,
    sectionProgram.lengthSection, baseSyndrome);

    if (!locationError) {
      HAL_UART_Transmit(&huart2, tx_error, sizeof(tx_error), 1000);
    } else {
      HAL_UART_Transmit(&huart2, tx_error1, sizeof(tx_error1), 1000);
    }
    osDelay(ERROR_PERIODICITY);
  }
}
```

C.4 Hamming code (hamming.c)

This file contains the functions required to perform the memory scrubbing using Hamming code (39,32). The whole file *hamming.c* is written below.

```
/* Includes ------------------------------------------------*/
#include "hamming.h"

/* Constants and variables ---------------------------------*/
#ifndef null
#define null ((void*) 0)
#endif

#define CODEWORD_LEN    0x27 // Total bits per codeword
#define MESSAGE_LEN     0x20 // Message bits per codeword
#define PARITY_LEN      0x07 // Check bits per codeword

#define UNCORRECTABLE   0xFF

// Periodicity with which scrub of memory is performed in milliseconds
const uint16_t SCRUB_PERIODICITY = 1000;
// This variable is set to 1 if an uncorrectable error has been found
uint8_t uncorrectableError = 0;
// Counter for the total number of errors corrected
uint16_t errorsCorrected = 0;

// First seven columns of matrix G. Columns needed to calculate the
    syndrome
static const uint32_t _colG[7] = {
  0x0000007E,
  0x007FFF00,
  0x7F00FF00,
  0x0F8F0F8F,
  0xB33333B3,
  0xD5D55555,
```

```
  0xE9E996E8
};

// Columns of matrix H
static const uint8_t _colH[39] = {
  0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01, 0x07,
  0x13, 0x15, 0x16, 0x19, 0x1A, 0x1C, 0x1F, 0x0B,
  0x23, 0x25, 0x26, 0x29, 0x2A, 0x2C, 0x2F, 0x31,
  0x32, 0x34, 0x37, 0x38, 0x3B, 0x3D, 0x3E, 0x0D,
  0x43, 0x45, 0x46, 0x49, 0x4A, 0x4C, 0x0E
};

/* Functions -------------------------------------------------*/
/**
  * @brief Performs the inner product of two uint8_t vectors
  * @param vector1: first vector
  * @param vector2: second vector
  * @param vLength: length of vectors
  * @retval uint8_t with the result of the inner product
  */
static uint8_t InnerProduct8(uint8_t vector1, uint8_t vector2, uint8_t
    vLength)
{
  uint8_t result = 0x00;
  uint8_t calc;

  calc = vector1 & vector2;
  for ( uint8_t i=0; i < vLength; i++ )
  {
    result ^= (calc & 1);
    calc>>=1;
  }

  return result;
}

/**
  * @brief Performs the inner product of two uint32_t vectors
  * @param vector1: first vector
  * @param vector2: second vector
  * @param vLength: length of vectors
  * @retval uint8_t with the result of the inner product
  */
static uint8_t InnerProduct32(uint32_t vector1, uint32_t vector2,
   uint8_t vLength)
{
  uint8_t result = 0x00;
  uint32_t calc;

  calc = vector1 & vector2;
  for ( uint8_t i=0; i < vLength; i++ )
  {
    result ^= (calc & 1);
    calc>>=1;
```

```
  }

  return result;
}

/**
  * @brief Calculates the syndrome bits corresponding to a message
  * @param message
  * @retval uint8_t with the syndrome calculated
  */
static uint8_t HammingCalculateSyndrome(uint32_t message)
{
  uint8_t syndrome = 0x00;

  for ( uint8_t i=1; i <= PARITY_LEN; i++ )
  {
    uint8_t innProd;

    innProd = InnerProduct32(message, _colG[i-1], MESSAGE_LEN);
    innProd <<= (PARITY_LEN - i);
    syndrome |= innProd;
  }
  return syndrome;
}

/**
  * @brief Check if there is an error in a codeword. Multiplies a
   given codeword (message + syndrome) with H matrix, if there is no
   error, the result should be zero.
  * @param message: information bits of the codeword
  * @param syndrome: check bits of the codeword
  * @retval uint8_t 0 if there is no error, a syndrome if there is an
   error
  */
static uint8_t HammingCheckSyndrome(uint32_t message, uint8_t syndrome
   )
{
  uint8_t calcSyndrome;

  // (codeword * H)
  for ( uint8_t i=1; i <= PARITY_LEN; i++ )
  {
    uint8_t innProdSyn = InnerProduct8(syndrome, _colH[i-1],
   PARITY_LEN);
    uint8_t innProdMess = InnerProduct32(message, _colG[i-1],
   MESSAGE_LEN);
    uint8_t innProd = innProdSyn ^ innProdMess;
    innProd <<= (PARITY_LEN - i);
    calcSyndrome |= innProd;
  }

  return calcSyndrome;
}
```

```
/**
  * @brief Performs the encoding of the message block allocated in the
     address received
  * @param *message: pointer to data block to be encoded
  * @param blocklength: length of the data block to encode
  * @param *syndrome: pointer to where the calculated syndrome block
    has to be allocated
  * @retval None
  */
void HammingEncode(uint32_t* message, uint32_t blockLength, uint8_t*
    syndrome)
{
  uint32_t numMessages = blockLength/4;
  for (uint32_t i = 0 ; i<numMessages; i++)
  {
    // Calculate the syndrome and store it in its address
    *(syndrome + i) = HammingCalculateSyndrome(*(message + i));
  }
}

/**
  * @brief Performs the decoding of the codeword allocated in the
    address received. If detected an error in the codeword it is
    corrected. If more than one error detected or uncorrectable
    returns UNCORRECTABLE (0xFF)
  * @param *message: pointer to data to be decoded
  * @param *syndromeAddress: pointer to syndrome corresponding to the
    data to be decoded
  * @retval uint16_t containing only the message decoded or
    UNCORRECTABLE
  */
uint16_t HammingDecode(uint32_t* message, uint8_t* syndromeAddress)
{
  uint8_t calcSyndrome = 0x00;
  uint16_t correctedErrors = 0;

  calcSyndrome = HammingCheckSyndrome(*message, *syndromeAddress);

  // If there is an error (calcSyndrome not equal to zero), correct it
  if (calcSyndrome)
  {
  // Look if the syndrome corresponds to a column of the parity-check
    matrix
  for( uint8_t i=0; i < CODEWORD_LEN; i++ )
  {
    if ( _colH[i] == calcSyndrome )
    {
      // If the error is in the syndrome, correct the bit affected
      if (i < PARITY_LEN) {
        uint8_t maskCorrector = 0x00;
        maskCorrector = (uint8_t)1<<(PARITY_LEN - (i + 1));
        *syndromeAddress ^= maskCorrector;
        correctedErrors = 1;
      // If it is in the message, correct the bit affected
```

```
        } else {
          uint32_t maskCorrector = 0x0000;
          maskCorrector = (uint32_t)1<<(CODEWORD_LEN - (i + 1));
          *message ^= maskCorrector;
          correctedErrors = 1;
        }
        return correctedErrors;
      }
    }
    // If no match found, it means there is an uncorrectable error
    if (!correctedErrors)
    {
      correctedErrors = UNCORRECTABLE;
      return correctedErrors;
    }
    }
    return correctedErrors;
  }

  /**
    * @brief Scrubs the memory block indicated and corrects errors if
      detected
    * @param *firstMessage: pointer to the beginning of the data block
      to be scrubbed
    * @param blocklength: length of the data block to scrub
    * @param *syndromes: pointer to the beginning of the corresponding
      syndromes block
    * @retval uint8_t containing the number of errors corrected in that
      scrub operation or
    *     returns 0xFF if an uncorrectable error has been found
    */
  uint8_t HammingScrubMemory(uint32_t* firstMessage, uint32_t
      blockLength, uint8_t* syndromes)
  {
    uint32_t* addressToScrub = firstMessage;
    uint8_t* syndromeToScrub = syndromes;
    uint8_t correctionsCounter = 0x00;
    uint32_t numMessages = blockLength/4;

    for ( uint32_t i=0; i<numMessages; i++ )
    {
      uint16_t numCorrectedErrors;
      numCorrectedErrors = HammingDecode(addressToScrub, syndromeToScrub
     );

      if ( numCorrectedErrors == UNCORRECTABLE ) {
        uncorrectableError = numCorrectedErrors;
        return uncorrectableError;
      } else {
        // first bit of syndrome is always zero since its length is 7
        // if a bit-flip is found in that position, correct it
        uint8_t syn2bits = (*syndromeToScrub) & 0x7F;
        if (syn2bits != *syndromeToScrub) numCorrectedErrors++;
```

```
      correctionsCounter += numCorrectedErrors;
    }
    addressToScrub += 1;
    syndromeToScrub += 1;
  }

  errorsCorrected += correctionsCounter;
  return correctionsCounter;
}
```

C.5 Code for the introduction of errors (error.c)

The whole file *error.c* is written below.

```
/* Includes ---------------------------------------------------*/
#include "error.h"

/* Constants and variables ------------------------------------*/
#define MESSAGE_LEN     0x20 // Number of bits of the messages
#define PARITY_LEN      0x07 // Number of check bits per codeword

// Periodicity with which errors are introduced in memory in
   milliseconds
const uint16_t ERROR_PERIODICITY = 1000;
// Counter for the errors introduced in memory
uint16_t errorsIntroduced = 0;

/* Functions --------------------------------------------------*/
/**
  * @brief Gets a random number within a certain range.
  * @param range
  * @retval Integer random number
  */
uint32_t randomNumberInRange(uint16_t range)
{
  double randNum;

  //We get a random number between 0 and 1 and then we scale it
  randNum = ((double)rand() / RAND_MAX) * (range-1);

  return (uint32_t)randNum;
}

/**
  * @brief Introduces single bit errors in a block of memory of 32-bit
    words.
  * @param *memoryBlock: pointer to first address of memory block
  * @param sizeMemoryBlock: size in bytes of memory block
  * @retval None
  */
void singleBitErrors32(uint32_t* memoryBlock, uint32_t sizeMemoryBlock
   )
{
```

```
  uint32_t randAddress;
  uint32_t randBitPosition;
  uint32_t maskError = 0x00000000;

  randAddress = randomNumberInRange(sizeMemoryBlock);
  randBitPosition = randomNumberInRange(MESSAGE_LEN);

  maskError = 1<<randBitPosition;

  //Change bit in the error position
  *(memoryBlock + randAddress) ^= maskError;
}

/**
  * @brief Introduces single bit errors in a block of memory or 8-bit
    words.
  * @param *memoryBlock: pointer to first address of memory block
  * @param sizeMemoryBlock: size in bytes of memory block
  * @retval None
  */
void singleBitErrors8(uint8_t* memoryBlock, uint32_t sizeMemoryBlock)
{
  uint32_t randAddress;
  uint8_t randBitPosition;
  uint8_t maskError = 0x00;

  randAddress = randomNumberInRange(sizeMemoryBlock);
  randBitPosition = randomNumberInRange(PARITY_LEN);

  maskError = 1<<randBitPosition;

  //Change bit in the error position
  *(memoryBlock + randAddress) ^= maskError;
}

/**
  * @brief Decides randomly where to introduce a single error: in
    memoryBlock or syndrome Block
  *    calls the function that introduces errors.
  * @param *memoryBlock: pointer to first address of memory block
  * @param sizeMemoryBlock: size in bytes of memory block
  * @param *syndromeBlock: pointer to first address of syndromes block
  * @retval 0 if error in program memory; 1 if error in syndromes
    block
  */
uint8_t errorInProtectedRegion(uint32_t* memoryBlock, uint32_t
   sizeMemoryBlock, uint8_t* syndromeBlock)
{
  uint32_t memoryRegion;
  uint32_t numMessages = sizeMemoryBlock/4;
  uint32_t totalSizeProtectedMem = sizeMemoryBlock + (sizeMemoryBlock
    /4);
```

```
  //Select randomly if error is introduced in protected memory block
   or in the syndromes region (also protected)
  memoryRegion = randomNumberInRange(totalSizeProtectedMem);
  if (memoryRegion <= sizeMemoryBlock) {
    singleBitErrors32(memoryBlock, numMessages);
    errorsIntroduced++;
    return 0;
  } else {
    singleBitErrors8(syndromeBlock, numMessages);
    errorsIntroduced++;
    return 1;
  }
}
```