VWO 6, 19/20

Version 15.0

11-3-20

Natanael Djajadi
Kees de Hoogh
Amy van der Meijden

How to prevent our computers from crashing in outer space?

Redundant Software Technology

Docent: Jeanna de Haan

Meesterproef

Client: Roland Weigand

Company: European Space Agency

1. INFORMATION PAGE

1.1. DATA

School:	Christelijk Lyceum Delft
Location:	Molenhuispad 1
Class:	V6o&o1
Group:	3
Team name:	Bits of Ank
Contractors:	Natanael Djajadi, Kees de Hoogh, Amy van der Meijden
Docent:	J. De Haan
Client:	Roland Weigand
Company:	European Space Agency (ESA), European Space Research and Technology Centre
Location:	Noordwijk, The Netherlands
Date start:	04-09-2019
Date end:	18-03-2020

2. PREFACE

We present to you the final report of our ‘Meesterproef’-project: “Redundant Software Technology”. This report has been generated as the result of a study to fulfil the graduation requirements of ‘Technasium’ at Christelijk Lyceum Delft. The project started on the 4th of September 2019 and the finalisation date is the 18th of March 2020.

This project has been made possible by Roland Weigand, the client who works at ESA. Together with Mr. Weigand, we formulated the research question of this project. In each meeting, we received many tips and tricks to fulfil the client's assignment. The project research was difficult, but the project was successfully finished with the help of the client, experts, and teachers. They were always available and open to questions and we are grateful for their support.

We would like to thank the client for giving us this opportunity to do a project for the ESA. We also would like to thank the experts Eduardo Leegwater Simoes, who works on blockchain-technology at the company ‘Atos’, and Paul Bokel, who is a Computer Science-teacher at our school. In addition, we would like to thank our teacher, Jeanna de Haan, who supported us during the project, and our English teacher, Anne-Sophie Kooij, who helped us with formulating the sentences in our report in formal English.

We hope you will enjoy the report.

Natanael Djajadi
Kees de Hoogh
Amy van der Meijden

Delft, 11-03-20

3. RESUME

This project is about software redundancy. So-called ‘bit-flips’ can occur - due to space radiation - when electronics are in the outer space. This can affect data that is being sent to earth. The goal of the project was to find and compare different ways to protect (make redundant) the data against these bit-flips via software. Consequently, the main question of this project is: *“How can software be made more redundant against bit-flips, induced by space radiation?”*

Before dividing this question into multiple sub-questions, literature research was done. Using the literature research, two ways to protect software were chosen: the ‘Triple Modular Redundancy’ (TMR) and the ‘Hamming code’. Furthermore, multiple ways to compare these protection-methods were found: the runtime, the used memory (code size and data size), the ‘Bit-Error-Rate’ (BER) and the ‘Failures-in-Time’ (FIT). By means of this information, the following sub-questions were defined:

1. *“How much does the number of failures per unit of time (‘FIT’) of a protected program (using the Triple Modular Redundancy or Hamming code) differ from a non-protected program?”*
2. *“Which of the means of protection for software, Triple Modular Redundancy or the Hamming code - measured using the number of injected errors per time unit (‘BER’), number of failures per time unit (‘FIT’) and runtime - is more efficient?”*
3. *“What is the price in runtime and code size by protecting the unprotected program with Triple Modular Redundancy and Hamming code?”*

Subsequently, a program of requirements regarding the research, end result and programming code was made. Based on this, two different codes were written in C: software that simulates bit-flips and an application software on which the TMR and Hamming code can be applied.

At last, the application software, the software with TMR and the software with Hamming code were tested. After running this software, the software gave the number of errors, the number of non-errors, and the runtime. With this data, the BER and the FIT were calculated. The code size and data size were calculated as well.

After doing the measurements, the conclusion can be drawn that there is always a price to pay for protection. When using the TMR, it will cost more memory than the Hamming code. On the other hand, when using the Hamming code, it will use more runtime than the TMR. The client can decide for himself whether he prefers a low runtime or less used memory.

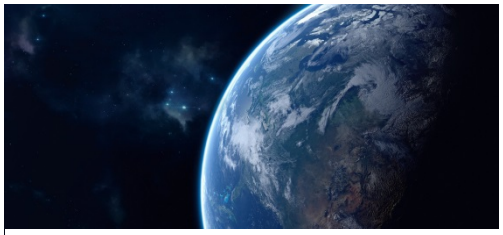


Figure 1 Our atmosphere

4. INDEX

1. Information page 1
   1.1. Data 1
2. Preface 2
3. Resume 3
4. Index 4
5. Introduction 6
   5.1. Motivation and relevancy 6
   5.2. The client 7
   5.3. Preparations 7
   5.4. Project steps 8
   5.5. Content of the report 8
6. Research questions 9
   6.1. Main question 9
   6.2. Sub-question 9
7. Orientation 11
   7.1. Counting in binary 11
   7.2. What are the effects of space radiation on software? 12
   7.3. Programming in C 12
   7.4. How to protect software against bit-flips 13
   7.5. Definitions 16
8. Program of requirements 17
   8.1. Research 17
   8.2. Programming code 18
9. Method of working 19
   9.1. Application software 19
   9.2. Principle of flipping bits 21
   9.3. How to inject 22
   9.4. Protection 22
   9.5. Measurement methods and implementation 23
10. Results 24
   10.1. Baseline measurement 24
   10.2. TMR 26
   10.3. Hamming code 28
11. Conclusion 28
   11.1. Sub-question 1 28
   11.2. Sub-question 2 29

11.3. Sub-question 3 29
11.4. Main question 29
12. Discussion 29
13. Recommendation 30
14. Sources 31
14.1. Final Report sources 31
14.2. Project plan sources 32
15. Attachments 34
15.1. Attachment 1: Written permission of ESA logo 34
15.2. Minutes of meeting 34
15.3. Attachment 2: results 38
15.4. Attachment 4: Code protections 41
15.5. Attachment 5: Projectplan 46

5. INTRODUCTION

5.1. MOTIVATION AND RELEVANCY

Radiation can be found everywhere around us. On earth, people are protected by the atmosphere and Earth's magnetic field. But when one launches electronics - like satellites - into space, they are not protected. Then software can be affected by space radiation.

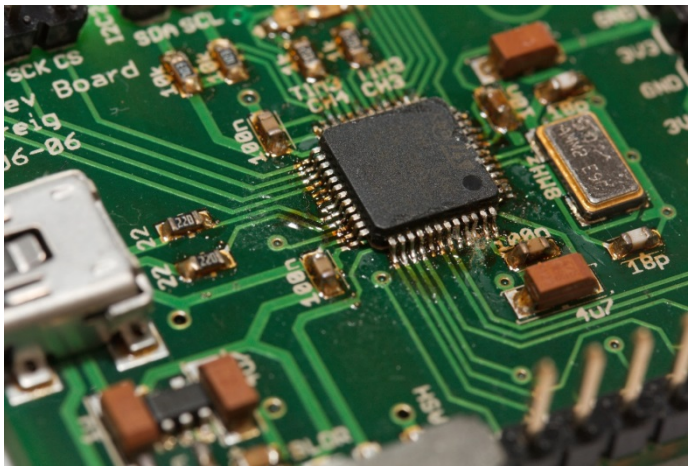


Figure 1 A computer chip

Space radiation can cause 'bit-flips' (so-called SEU's = Single Events Upsets), which are common for digital electronics, like computer chips. If data bits are flipped – which means that the value of a variable may be changed - the computer may produce wrong results, start executing incorrect, undesired instructions, or even branch to a wrong address. Most of the time, the bit-flips are harmless, because a lot of data in computers are stored but never used. But in some cases, it can lead to crashes or to wrong decisions.

In everyday life, people generally do not think about how disadvantageous these crashes or wrong decisions could be. Yet, electronics in space are often used in daily life, for instance for navigation, predicting the weather, the internet, and so on. This is also why this project belongs to the 'Bètaworld' (a category in Technasium) *Lifestyle & Design.* More information about this Bètaworld can be found in the attachments > Projectplan > Bètaworld.

Nowadays, space agencies try to prevent bit-flips by designing chips to be more tolerant against space radiation. Our client is part of the section at ESA that is responsible for this. It is relatively difficult to solve this problem via hardware because manufacturing a chip just for space is very expensive. That is why we make software tolerant with the use of protection strategies, such that cheaper commercial computer chips could be used as well. The main question of this project is:

"How can software be made more redundant against bit-flips, induced by space radiation?"

5.2. THE CLIENT

Our client, Mr. Weigand, is a chip designer, who focuses on redundancy. He ensures that the chips are robust against space radiation. He works in the department “Microelectronics Section” in ESTEC (European Space Research And Technology Centre), a branch of ESA.

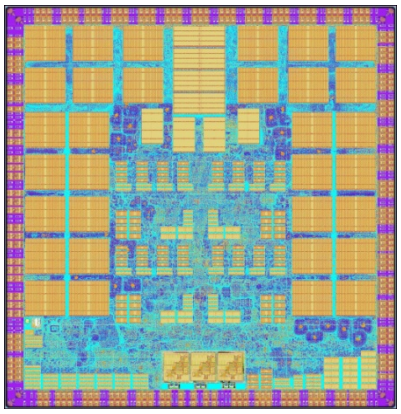


Figure 2 A redundant chip created by our client, Mr. Weigand

ESA is an international organisation. The mission of ESA is to shape the development of Europe’s space capability and to ensure that investment in space will deliver benefits to Europe and the world. It is ESA’s task to prepare and implement the European space programme. Some programmes are designed to find out more about Earth, our Solar System and Universe, to develop satellite-based technologies and services and much more. More information about the client can be found in the attachments > Projectplan > The client.

5.3. PREPARATIONS

Before the programming can be started, an environment had to be found that can code in C and to make and execute codes at high performance. At first, a website was used that was able to do these things but it was not able to execute big codes. Therefore, we chose to use a laptop with Linux ‘Ubuntu’. Installing Linux on a computer and getting ready to code was quite tricky. A tutorial (Canonical Ltd., n.d.) was followed to install it properly. Thereafter, a few essential commands were entered from another tutorial (ProgrammingKnowledge2, 2018) to build and run C programs. With the use of Visual Studio Code, coding was made easier, because it recognizes mistakes while working on codes.

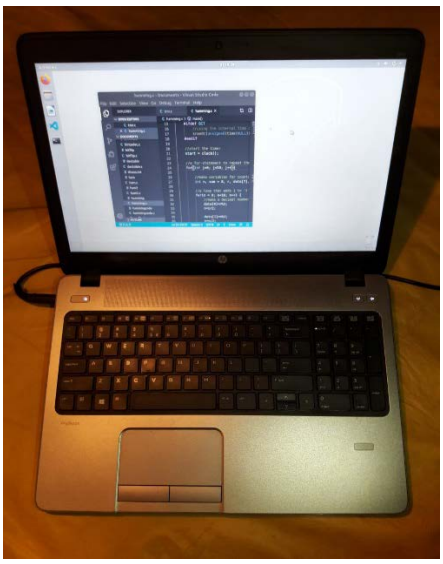


Figure 3 The computer with Linux

5.4. PROJECT STEPS

The following steps were taken in the project. A more elaborate version can be found in the attachments > Projectplan > Project steps.

Step 1: Get familiar with space radiation effects on chips and computers

1. *What are the effects of space radiation on the software?*
2. *How to protect the software against bit-flips?*

Step 2: Specification of requirements

Step 3: The research I

1. *How to make a program that randomly changes bits?*
2. *Baseline measurement*

Step 4: The interim presentation

Step 5: The research II

1. *Add chosen methods to the software*
2. *Testing the protected programs*

Step 6: The second interim presentation

Step 7: End result - the report and presentation

1. *Make a report of the research*
2. *Make a presentation of the research*

5.5. CONTENT OF THE REPORT

This report firstly describes the research questions and the literature research (orientation) of the project. The literature research was done in order to get the knowledge which is needed to find an answer to the research question. After the orientation, one can find the program of requirements, the application software and the software which simulates the bit-flips. Then the results are shown, containing the baseline measurement and the measurement of two different ways of protection: The Triple Modular Redundancy and the Hamming code. Using these results, a conclusion was drawn. The things that could have made the research less reliable can be found in the discussion. And finally, recommendations for follow-up research are discussed.

6. RESEARCH QUESTIONS

In the research, the following questions will be answered.

6.1. MAIN QUESTION

As mentioned before, the main question of this research is:

“How can software be made more redundant against bit-flips, induced by space radiation?”

6.2. SUB-QUESTION

The main question is divided into the following sub-questions. The used terms (like ‘BER’, ‘FIT’, ‘Triple Modular Redundancy’ and ‘Hamming code') will be explained in the ‘Orientation’.

6.2.1. SUB-QUESTION 1

“How much does the number of failures per unit of time (‘FIT’) of a protected program (using the Triple Modular Redundancy or Hamming code) differ from a non-protected program?”

This sub-question is important to study, because if there is not a big difference with or without protection, it would be meaningless to implement any protection to the software. This is because redundancy has impact on runtime and the runtime would preferably be as low as possible. Most likely, the difference between the Failures-In-Time (FIT) for the protected and non-protected programs is considerably high. This is based on the fact that ideally the FIT of the protected program is 0.

6.2.2. SUB-QUESTION 2

“Which of the means of protection for software, Triple Modular Redundancy or the Hamming code - measured using the number of injected errors per time unit (‘BER’), number of failures per time unit (‘FIT’) and runtime - is most efficient?”

It would be very useful to answer this question, because in general, we prefer a lower runtime. If redundancy is implemented in a certain kind of software, the runtime will increase. By using statistics of the FIT, the Triple Modular Redundancy (TMR) and the Hamming code can be compared.

In the operation of these two types of protection, the hypothesis is that these two will be equivalent when it comes to protection, but we expect the runtime of the TMR to be higher, since the data is tripled.

6.2.3. SUB-QUESTION 3

“What is the consequence for runtime and code size by protecting the unprotected program with Triple Modular Redundancy and Hamming code?”

Runtime and code size are important metrics in relation to the practical implementation of the protection measures. Different applications need different kind of protections. Depending on the application, clients might want a small code size or a fast runtime.

Because of the protection, the code needs extra logic to correct the bit-flip, and we expect that this will increase the code size. Also, the runtime will be longer, since the program should take more time to correct the bit-flip.

7. ORIENTATION

7.1. COUNTING IN BINARY

Before diving into the subject of our project, this section explains how the software works in the language of a computer. At school, pupils learn that there are ten different numbers. This is called the decimal system. The decimal system has historically developed because humans have 10 fingers.

For example, in the decimal system, the number 6359 is the sum of: $6\times10^3 + 3\times10^2 + 5\times10^1 + 9\times10^0$

It can also be shown in a table like this:

10^3	10^2	10^1	10^0
*	*	*	*
6	3	5	9
=	=	=	=
6000	300	50	9

If the numbers in the blue row are placed next to each other, the number 6359 will be the outcome and if the red row is summed up (6000+300+50+9), the outcome is 6359.

On the contrary, most computers work with the binary system, which means that it uses only two different numbers, namely 0 and 1. This is because two states can easily be stored in a (bistable) electronic circuit. If 6359 is written in binary, it is necessary to split 6359 in powers of two. So 6359 would be: $2^{12} + 2^{11} + 2^7 + 2^6 + 2^4 + 2^2 + 2^1 + 2^0$.

2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
*	*	*	*	*	*	*	*	*	*	*	*	*
1	1	0	0	0	1	1	0	1	0	1	1	1
=	=	=	=	=	=	=	=	=	=	=	=	=
4096	2048	0	0	0	128	64	0	16	0	4	2	1

If the numbers in the blue row are placed next to each other, 1100011010111 will be the outcome and if the red row is summed, the outcome is 6359. This means that 1100011010111 in the binary system is the same number as 6359 in the decimal system.

One 0 or 1 in a computer is called a 'bit' (a binary digit) and 8 bits in a row is called a 'byte'. This means that a byte can have 2^8 = 256 different forms. Our computer translates this binary language to a language that is understandable for non-experts by processing one byte at a time.

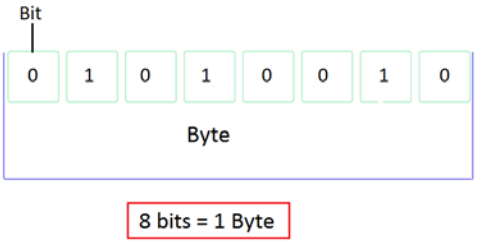


Figure 5 The difference between a bit and a byte

7.2. WHAT ARE THE EFFECTS OF SPACE RADIATION ON SOFTWARE?

Everywhere around us are ionizing particles like electrons, neutrons, protons, ions and, photons (also known as space radiation). When an ionizing particle hits a sensitive node on a device, this may cause a SEU (a Single Event Upset) meaning that a 0 can change into a 1 and vice versa. The consequence of this is that the data has inadvertently been changed. This can lead to crashes and wrong decisions.

7.3. PROGRAMMING IN C

In this project, a software - which is programmed in C - will be tested and improved. C is a general-purpose programming language that is widely used around the world. This means it can be used to develop operating systems, embedded systems, databases and so on. This software is also chosen because it is a procedural language, which means that the instructions in a C program are executed step by step. Finally, C programming is faster in executing than most programming languages like Java, Python, and other languages. In order to understand the software, it is necessary to understand the basics of C.

7.3.1. BITWISE OPERATORS

When programming in C, operations can be performed by operators on a bit level or on a byte level. This project concerns bit-flips, so bitwise operators will be used. By using these operators, the bits will be manipulated. Examples of bitwise operators are 'NOT', 'OR' and 'AND'. Bitwise operators are explained below:

For example, the bits *p*, *q*, and *r* can each have two values, namely true (1) and false (0). Variable *r* depends on *p* and *q* and on which bitwise operator is used. When using the 'OR'-operator, you say that *r* is true when *p* or *q* is true. For example, when *p* is true and *q* is false, *r* is true.

When using the 'OR'-operator, the table would look like this

p	q	r
true	true	true
true	false	true
false	true	true
false	false	false

When using the ‘AND’-operator, one says that r is true when p and q are true. In a table:

p	q	r
true	true	true
true	false	false
false	true	false
false	false	false

In a computer, these operators are often used, but instead of ‘true’ and ‘false’, the bits take the value 1 (true) or 0 (false).

7.4. HOW TO PROTECT SOFTWARE AGAINST BIT-FLIPS

To protect a software, the software should correct a bit-flip when space radiation is making a bit flipping. When one wants to protect software against bit-flips it is important to be able to detect the error in the software, but it is also useful to know exactly where the error has occurred in order to correct this mistake. There are various ways - with different degrees of protection - to accomplish this, but all solutions require additional storage and processing power.

The following methods show different ways to detect and/or correct bit-flips.

7.4.1. TRIPLE MODULAR REDUNDANCY

Triple Modular Redundancy is one of the simplest ways to make software (almost) redundant. A process is being performed three times and the result will be processed by a majority-voting to have a single output. For example, if a 1 is sent three times, the receiver should receive ‘111’ and he can be very sure that the original value is a 1. But if a bit-flip takes place, and the receiver receives ‘110’, he can still be fairly sure that the original value is a 1 because of the majority-voting.

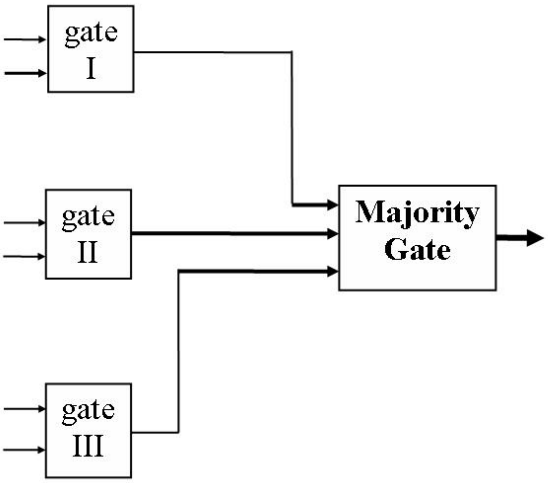


Figure 6 Triple Modular Redundancy

7.4.2. PARITY BIT

A parity bit - or a check bit - is a bit which is added to control a set of bits, using parity. In other words, the parity bit makes sure that the set of bits that it controls is in total an even number. The set of bits ‘0010110’, for example, has three 1s, which is an odd number. The parity bit will therefore be a 1 to make the sum an even number. The set of bits including the parity bit will be 00101101. This parity bit is continuously checking if the sum is an even number. When the sum is not even anymore, an error will occur and then the user knows that a bit has been flipped in that set of bits. The cost of this method is low, because only one extra bit is needed to check multiple bits (for example a byte). A problem of this method is that you don’t know which of the bits in that set has been flipped, so you can only detect, not correct. In order to perform ‘Error Detection And Correction’, a more complex scheme is needed, using more memory.

7.4.3. HAMMING CODE

Using the Hamming code, a fault in the data can be detected as well as corrected, without doubling the number of bits. This code works by using the following principle, which is visualized below:

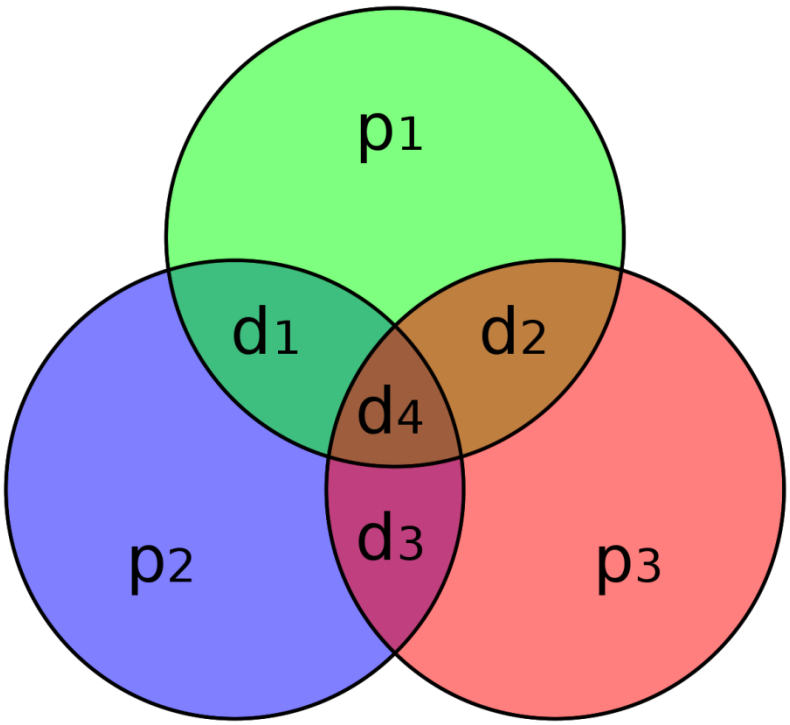


Figure 7 The Hamming code

The set of bits exists of d1, d2, d3, and d4, plus the parity bits p1, p2, and p3. Every parity bit checks the bits which are in its circle.

- p1 is the parity bit of d1, d4 and d2
- p2 is the parity bit of d1, d4 and d3
- p3 is the parity bit of d3, d4 and d2

When one of the bits is flipped, the parity bits will present an error if the bit-flip has occurred in its part of the circle. So when the bit d3 flips, the parity bits p2 and p3 will give an error. With that information, it is clear that the flipped bit is d3 or d4, and because p1 does not present an error, d1, d4 and d2 are not flipped. Through this, it is clear that d3 is the flipped bit.

This method is able to detect where the bit has flipped, with a relatively small number of extra bits. In this method 3 parity bits were used, to control a set of 4 bits.

7.4.4. CHECKSUM

The checksum code is another way to detect an error, which has the following principle:

Suppose that there is a set of numbers:

25 11 12 7 13 4

The sum of these numbers is 72. Now the remainder of this sum - after dividing with a randomly chosen number - is taken: 72 modulo* 16 = 8. So 8 is the checksum, which is added to the set of numbers:

25 11 12 7 13 4 **8**

The code continuously checks if the sum of the values including the checksum value (72+8=80) modulo 16 is 0. If a bit has been flipped, the sum modulo 16 will not be 0 anymore.

* The result of a modulo division is the remainder of an integer division of the given numbers. For example: 27 / 16 = 1, remainder 11. Consequently, 27 modulo 16 = 11

7.5. DEFINITIONS

In order to test the software on its redundancy, the Bit Error Rate (BER), the Failures In Time (FIT), the code size, and the data size will be measured.

7.5.1. BER

BER stands for the ‘Bit-Error-Rate’. It is the rate of how many bit-flips occur per unit of time. It depends on the radiation spectrum present in a satellite orbit and the type of chip being used.

7.5.2. FIT

FIT stands for the ‘Failures-In-Time’. The FIT is - compared to the BER - a statistical measure of how many program failures to expect per time unit. The FIT depends on the BER, but also on how the program is implemented.

7.5.3. CODE SIZE

The code size is the size of the executed code. This is dependent on the length of a code.

7.5.4. DATA SIZE

The data size is the size of the data in bytes. This depends on the number of variables in a program. Each variable is 8 bytes on the computer that is used, an Intel 64-bit Linux.

8. PROGRAM OF REQUIREMENTS

Below, the program of requirements of the research, and of the programming code are shown.

8.1. RESEARCH

Requirements	Explanation
The application program should run easily.	The application program will run quickly, so no time is lost while waiting for execution.
The program should be executed from a command-line.	It will be easier for the researchers to check if a bit-flip occurs in the program.
The results of bit-flips should be detected: The program should be self-checking, returning an error code if there is a failure.	It will be easier for the researchers to check if a bit-flip occurs in the program.
It is possible to perform fault injection in the program.	Otherwise, the bit-flip will not (or rarely) occur. A bit-flip will occur when a fault injection is performed.
It should be possible to measure the execution time of the program with / without protection scheme, but not including the fault injection code or any other debug code such as “printf” and “scanf”.	The consequences of the protection scheme (execution time, memory usage) is an important result for the user / customer.
The results are processed in results with statistics.	Where and when the bit-flips occurred will be clear and well-arranged with the help of statistics, like graphs.
The program should be redundant against bit-flips.	The updated program should have relatively fewer bit-flips than the original program.

8.2. PROGRAMMING CODE

Requirements	Explanation
The application program should be able to compile with and without debug code.	The program will not use the “printf” statement when not executed in debug mode, so runtime is not affected.
The application program should be executed in a command-line text file.	The program will be made in Visual Studio Code and the program can be successfully executed in the terminal.
The effects of bit-flips should be detected.	If the result is not as expected, it will return the program with an error. So the effects of bit-flips can be detected.
It should be possible to perform fault injection in the program.	It is possible to perform fault injection with a code made in a different program.
The effects of bit-flips should be translated into quantitative metrics BER and FIT.	It is possible to check whether a bit-flip has occurred. The numbers of errors can be detected and counted. The runtime will be measured. By using BER and FIT the results can be monitored.

9. METHOD OF WORKING

9.1. APPLICATION SOFTWARE

The following program was written to be tested on its redundancy. This software is chosen, because it is a relatively simple program for adding code. The program will calculate the sum from zero to 15, which is 120, and will check if this is correct. In case of a bit-flip, the result could differ from 120. It also measures the runtime. When debugging, the individual numbers, the sum, and the runtime are provided as output.

The code of the program is shown below, with an explanation per line.

```
//get a random number for r

#include <stdio.h>
#include <time.h>

clock_t start, end;
double cpu_time_used;

int main()
{
	//start the timer
	start = clock();

	//a for-statement to repeat the program a certain times
	for(int j=0; j<10; j++){

		int sum = 0;
		//a loop that adds 1 to 'i' until 15
		for(int i = 0; i < 16; i++)
		{
			#ifdef DEBUG
			printf("%d ", i);
			#endif
			//add up i to sum
			sum += i;
	}

	//check if the sum is equal to 120
	if (sum == 120) {
		#ifdef DEBUG
		printf("\nSum: %d\n", sum);
		#endif
	}
	else {
		printf("error!");

	}
}
///stop the timer and print the runtime
end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
printf("Runtime:%f \n", cpu_time_used);

return 0;
}
```

#include <stdio.h> Means that the content of ‘stdio’ is inserted in this place. Stdio.h is the header file for standard input and output.

int main() {}
Announces that there is a code that finally returns a certain variable number. The code includes everything between the curly brackets.

int sum = 0;
Creates the variable ‘sum’ and gives this variable a value of 0.

for(int j=0; j<10; j++){}
Consists of three steps:

1. It announces a loop that creates a variable ‘j’ and gives it an initial value of 0.
2. If the value of j is less than 16, the loop part is executed.
3. At the end of the execution, j is incremented, and will repeat again from step 2.

The part between the curly brackets is executed 16 times.

#ifdef DEBUG #endif
This part has two statements. It begins with ‘#ifdef’ and ends with ‘#endif’. The part between these statements is not included in the compiled program – if “DEBUG” is not defined - and therefore does not affect runtime.

sum += i;
The value of ‘j’ is added to ‘sum’ in the loop. When the loop is done, the value of ‘sum’ should be 120.

if (sum == 120) {} else {}
This code checks if the ‘sum’ is equal to 120 and executes the part between the first two brackets. If it is not equal to 120, then the part after ‘else’ between the brackets is executed.

return 0;
This statement is used to exit the program and to signal that no error occurred.

9.2. PRINCIPLE OF FLIPPING BITS

In order to find the FIT rate, the program and protection scheme have to be tested, the bits have to be randomly flipped via software, since testing in space or in a particle accelerator is very expensive. The program injects errors into its variables using the following code.

```
int main() {
        int a;
        //ask for a number for the variable ‘a’
        scanf("%d", &a);

        //get a random number for r
        srand((unsigned)time(NULL));
        int r = rand() % 8;

        //simulate the bit-flip
        a ^= (1 << r);
        printf("You entered: %d \n", a);

        return 0;
}
```

`int main() {}`
Announces that there is a code written, which finally returns a certain variable number. The code includes everything between the curly brackets;

`int a;`
Means that there is an integer* called ‘a’;

`scanf("%d", &a);`
Takes input from the user and stores this in ‘a’. ‘%d’ means that a decimal number is expected;

`srand((unsigned)time(NULL));`
Is in order to get a random number, depending on the computer’s internal time clock;

`int r = rand() % 8;`
Means that integer ‘r’ is a random number between 0 and 7. Consequently, ‘r’ can assume 8 different numbers;

`a ^= (1 << r);`
Makes sure that there is a bit flip on integer ‘a’.

* a whole number that can be positive, negative or zero

9.3. HOW TO INJECT

To simulate the effect of space radiation which can arrive any moment during the execution of the program, a separate program should run fault injections independently in a multi-tasking environment. Since it is difficult to implement a program into another program, it was agreed that the application program will have ‘random’ fault injection. So the code of the fault injection is entered within the plain code to perform a bit-flip. The principle of flipping the bits can be found in section 9.2 Principle of flipping bits.

9.4. PROTECTION

The Triple Modular Redundancy program and the Hamming code program can be found in the attachments. For the Hamming code, a program (Mishra, 2017) is used. This code is edited so it functions our ‘counting machine’.

9.5. MEASUREMENT METHODS AND IMPLEMENTATION

9.5.1. CALCULATE FIT AND BER

For space applications, the client should be provided with an expected failure rate. This can be done with the Failures-In-Time (FIT), a statistical measure of how many program failures to expect per time unit. Ideally, the FIT should be 0, but with bit-flips, this is sometimes difficult to achieve. At the same time, the radiation effects are characterized by the rate of how many bit-flips per time unit occurs, the BER. In this case, the BER depends on how many errors are injected per run and on the runtime per run.

This is the procedure to determine the FIT and the BER:

For each test result, the program is run N times. The starting value of N is one million and after three tests N is increased with one million until ten million is reached. Each test is carried out three times, because the average of these three values will get closer to the actual value. The results will contain the following data:

- The number of runs N
- How many errors are injected (Ni) (With one injection per run, Ni = N)
- How many functional failures occur (Nf)
- How many good runs without failures occur (Ng)
- The runtime (T0) for all runs

With this information, the BER can be calculated, using the formula BER = Ni / T0. This means the BER is the errors that are injected divided by the runtime. The FIT can be calculated using the formula FIT = Nf / T0, so the functional failures divided by the runtime. The functional failures are the errors in the execution of the program.
It can also be verified that N = Nf + Ng.

9.5.2. CALCULATE THE DATA SIZE

To calculate the data size, the relevant variables in the program are counted. Every variable is 8 bytes. So the total amount of bytes is equal to variables × bytes.

9.5.3. CALCULATE THE CODE SIZE

In order to see the size of executed code, the program needs to be disassembled with the command 'objdump -d [code]'. This will show many hexadecimal numbers. The most interesting part of the output is the <main> segment. To get the code size, one should subtract the first from the last hexadecimal number. Hexadecimal counting is a system of numerical notation that has 16 characters in its base (instead of 10 in the decimal system).

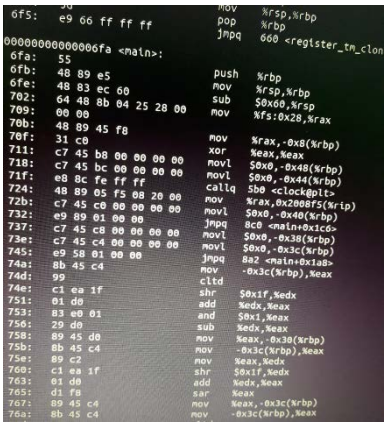


Figure 8 Executing the command 'objdump - [code]'

10. RESULTS

The results of the measurements described in section 9.5 will be shown below. The application software (baseline measurement) and the application software with TMR protection and Hamming code protection were tested and compared. The test was performed on the:

- Number of errors
- Number of non-errors
- Runtime

Using the number of errors and runtime, the BER and the FIT were calculated.

The TMR and the Hamming code were also tested on its memory size, so that these two can be characterized regarding the efficiency of the used memory.

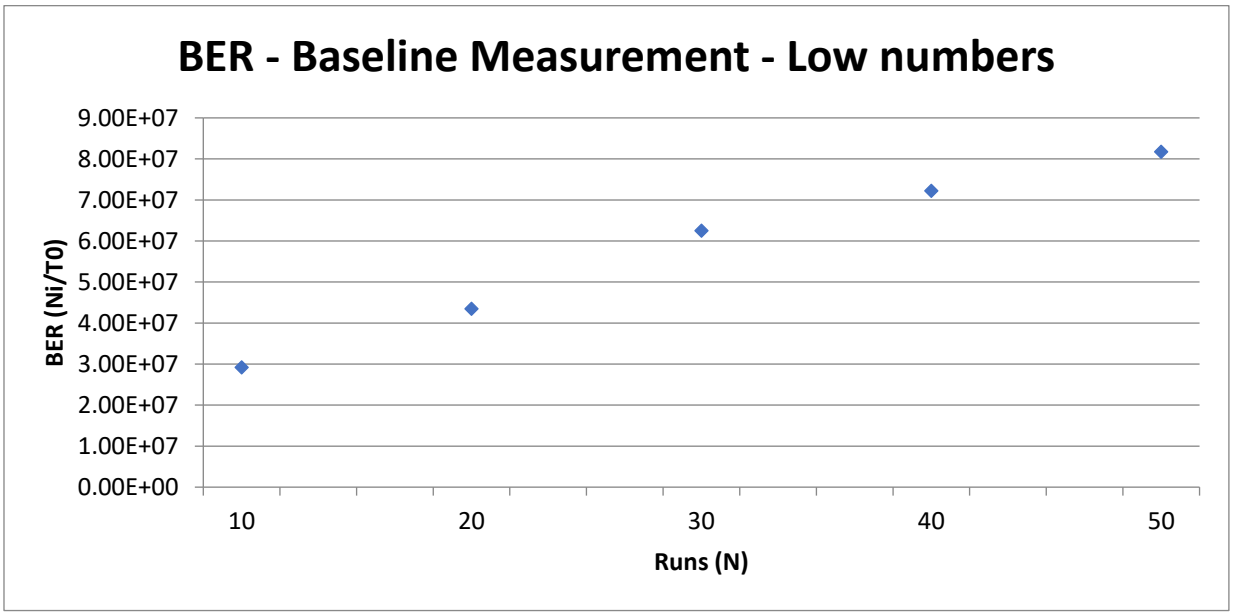
The three variations of the software were tested with a low number of runs (10, 20, …, 50) as well as a high number of runs (10^6, 20^6, 30^6, …, 10^7). The low number of runs is in order to visualize the modification of the BER, because with a high number of runs the BER is almost constant.

10.1. BASELINE MEASUREMENT

The results of the baseline measurement are shown below.

10.1.1. BER

Graph 1

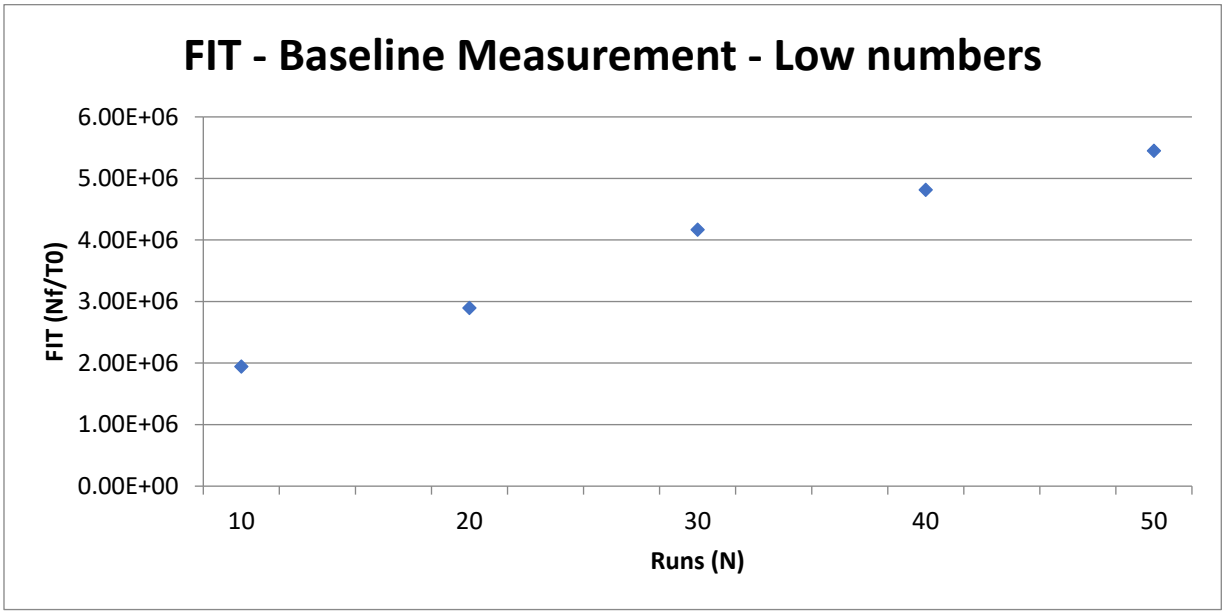


This is the graph of the BER of the baseline measurement, using low numbers. One can see that the BER increases when the number of runs increases.

With high numbers, the value is approximately $3{,}7 \times 10^8$.

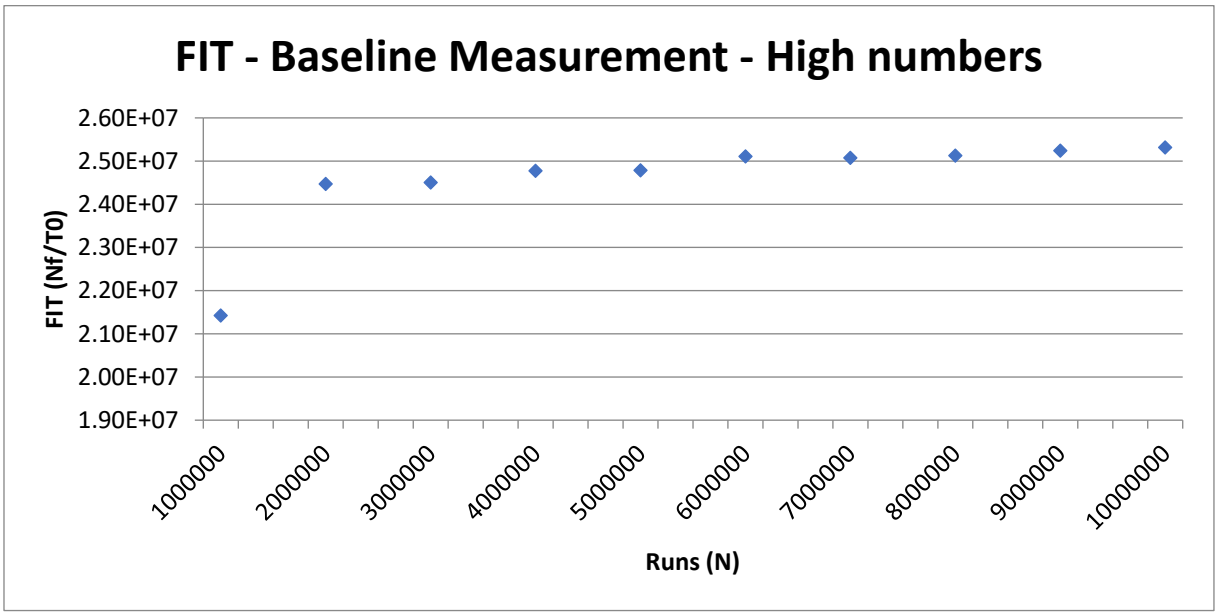
10.1.2. FIT

Graph 2



This is the graph of the FIT of the baseline measurement, using low numbers. As well as the BER, the FIT increases with the number of runs.

Graph 3



When looking at the FIT using high numbers, one can see that the increasing trend does not continue. It looks like the FIT becomes a constant value.

10.1.3. CODE SIZE

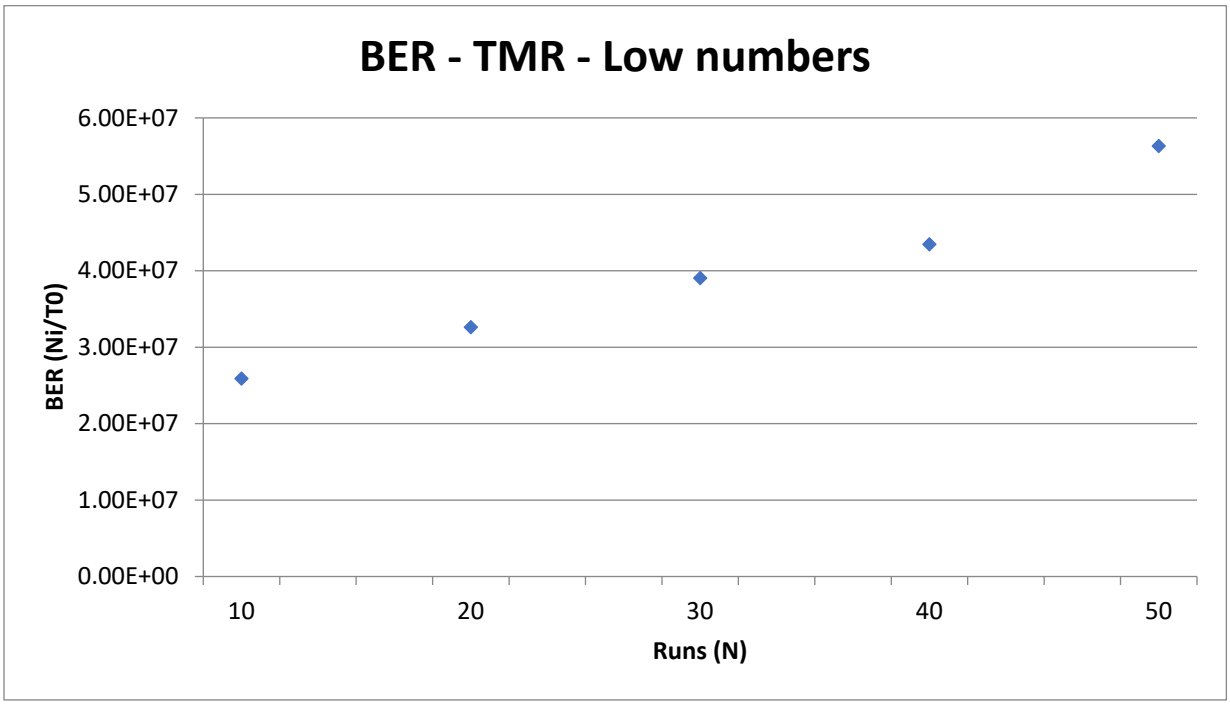
The code size is 7fb – 6da = 121. In decimal value is this 289 bytes.

10.2. TMR

The results of the software with the TMR are shown below. Using this protection, the number of errors is continuously 0, which means that the FIT is also continuously 0 (since FIT = Nf / T0).

10.2.1. BER

Graph 4



With high numbers, the value is approximately $1{,}6 \times 10^8$.

10.2.2. DATA SIZE

The TMR stores the variables three times to compare them later with each other. These are stored in data[0], data[1], and data[2]. These should be multiplied by four to have a better comparison with the Hamming code, as they both should use binary numbers.

The number of variables in the code should be multiplied by eight. This is because on a intel 64 bits Linux, every memory word is eight bytes. Therefore, the memory usage of the TMR is 96 bytes.

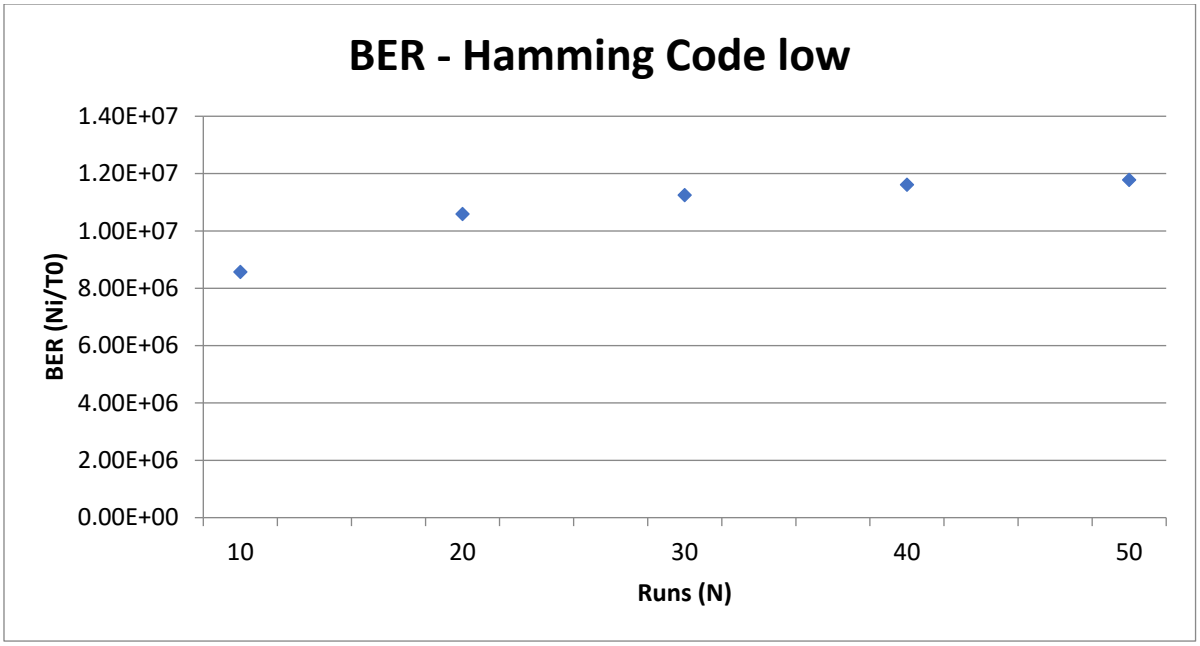
10.2.3. CODE SIZE

The hexadecimal code size of TMR is 86d – 6fa = 173. In decimal value is this 371 bytes.

10.3. HAMMING CODE

10.3.1. BER

Graph 5



With high numbers, the value is approximately $3{,}8 \times 10^7$.

10.3.2. DATA SIZE

The Hamming code stores 7 bytes for one number. These variables are data[0] until data[7]. It stores the variable ‘c’ as well. Therefore, the memory usage of the Hamming code is 8 variables. This means that the data size is 64 bytes.

10.3.3. CODE SIZE

The code size is 95f – 6fa = 256. In decimal value is this 738 bytes.

11. CONCLUSION

11.1. SUB-QUESTION 1

“How much does the number of failures per unit of time (‘FIT’) of a protected program (using the Triple Modular Redundancy or Hamming code) differ from a non-protected program?”

The difference is immense. Because the TMR and the Hamming code do not have functional failures, the FIT is 0. But the baseline measurement has almost every time a functional failure, so the FIT average is above the $2{,}45 \times 10^7$ in our research.

11.2. SUB-QUESTION 2

"Which of the means of protection for software, Triple Modular Redundancy or the Hamming code - measured using the number of injected errors per time unit ('BER'), number of failures per time unit ('FIT') and runtime - is more efficient?"

By looking at the results, the TMR is more efficient than the Hamming code in runtime. The FIT is the same for both. TMR costs more storage, so Hamming code is regarding storage more efficient.

Table 1 The end results

	Runtime	BER	FIT	Code size	Data size
Plain code	$4{,}05 \times 10^{-8}$	$3{,}71 \times 10^{8}$	$2{,}45 \times 10^{7}$	289 bytes	NA
TMR	$9{,}19 \times 10^{-8}$	$1{,}63 \times 10^{8}$	0	371 bytes	88 bytes
Hamming code	$8{,}09 \times 10^{-7}$	$1{,}85 \times 10^{7}$	0	738 bytes	64 bytes

11.3. SUB-QUESTION 3

"What is the price in runtime and code size by protecting the unprotected program with Triple Modular Redundancy and Hamming code?"

By looking at table 1, the runtime of the TMR is 55,93% more slowly than the plain code and the runtime of the Hamming code is 94,99% more slowly than the plain code. The code size of the TMR is 22,10% and of the Hamming code 60,84% more bytes than the plain code. If the client wants a better runtime, TMR is more efficient. If the client wants a smaller code size, the TMR is also better. For a smaller data size the Hamming Code is most efficient.

11.4. MAIN QUESTION

"How can software be made more redundant against bit-flips, induced by space radiation?"

By using different protections like the TMR and Hamming code, the software can be made more redundant against bit-flips. But there is always a price to pay for protection. This can influence the runtime, the code size, and data size.

12. DISCUSSION

In line with the hypothesis, the difference between the protected and unprotected programs FIT is immense. The difference is $2{,}45 \times 10^{7}$. But - as the results indicate - there is always a price to pay for protection and this price varies between the different protections. For example, the TMR's runtime is faster than the Hamming code, but it

uses more data. By protecting with the Hamming code the bit-flip is checked by different parity bits, so it takes longer to determine where the bit-flip occurred and to correct it, explaining the larger runtime and code size. This is not in line with our hypothesis, since we expected the runtime of TMR to be higher.

The larger data size of the TMR can be explained because in the TMR the data is tripled, so it cost more storage. This is in line with our hypothesis.

The general application of the results is limited by the design of the protection in this study. If it happens that two bitflips take place at the same time, the Hamming code and the TMR are not able to correct the fault. It is beyond the scope of this study to simulate the effect of space radiation which can arrive at any (random) moment during the execution of the program, by running run fault injection independently in a separate program,.

Moreover, the following things could affect the outcome of the research:

- The runtime can be affected by the computer. The computer is a non-realtime operating system and this will affect the time measured by the laptop.
- A computer is not capable of generating a completely random number, because there is always an algorithm included when one wants a random number from a computer. This could have affected the results.
- We are not a professional software engineer, so we cannot guarantee that the written code is the best one or the most optimal one.
- We do not know if the bit-flip simulator is a good simulation of the bit-flips which occurs in space.

13. RECOMMENDATION

Further studies should take into account which computer one will use, because there is a possibility it will impact the results. Also, for a more reliable research, more protection methods, other than TMR and Hamming code, should be tested to check whether other redundancy methods can be used, that can handle more bit-flips than just one in parallel.

14. SOURCES

14.1. FINAL REPORT SOURCES

14.1.1. SITES

Canonical Ltd. (n.d.). Install Ubuntu desktop. Retrieved November 27, 2019, from https://ubuntu.com/tutorials/tutorial-install-ubuntu-desktop#1-overview

Mishra, N. (2017, March 6). Hamming Code in C and C++. Retrieved January 10, 2020, from https://www.thecrazyprogrammer.com/2017/03/hamming-code-c.html

Parewa Labs Pvt. Ltd. . (n.d.). Learn C Programming | Programiz. Retrieved 8 March 2020, from https://www.programiz.com/c-programming

ProgrammingKnowledge2. (2018, May 22). *How to Compile and Run C program Using GCC on Ubuntu 18.04 LTS (Linux)* [Video file]. Retrieved from https://www.youtube.com/watch?v=oLjN6jAg-sY

Wikipedia. (n.d.). Bitwise operations in C. Retrieved December 11, 2019, from https://en.wikipedia.org/wiki/Bitwise_operations_in_C

14.1.2. FIGURES

ESA logo:
European Space Agency. (n.d.). ESA Logotype. Retrieved 8 March 2020, from https://www.esa.int/esalogo/signature.html

Laptop in space:
Quach, K. (2017, August 12). Place your bets: How long will 1TFLOPS HPE box last in space without proper rad hardening. Retrieved 8 March 2020, from https://www.theregister.co.uk/2017/08/12/spacex_hpe_iss_launch/

CLD logo:
Christelijk Lyceum Delft. (n.d.). CLD - Home. Retrieved 8 March 2020, from https://www.chrlyceumdelft.nl/

Technasium logo:
Bouwend Nederland. (2019, October 3). Technasium Innovatieprijs. Retrieved 8 March 2020, from https://jegaathetmaken.nl/technasium-innovatieprijs/

Figure 1: Our atmosphere
Fonds d'écran Belle terre, planète bleue 2560x1440 QHD image. (n.d.). Retrieved 8 March 2020, from https://fr.best-wallpaper.net/Beautiful-Earth-blue-planet_wallpapers.html

Figure 2: A computer chip
Stuivenberg, J. (2015, May 27). Deze houten computer chip is volledig biologisch afbreekbaar. Retrieved 8 March 2020, from https://numrush.nl/deze-houten-computer-chip-is-volledig-biologisch-afbreekbaar/

Figure 3: A redundant chip created by our client, Mr. Weigand
European Space Agency. (2016, December 8). GR740 next-generation microprocessor. Retrieved 8 March 2020, from http://www.esa.int/ESA_Multimedia/Images/2016/12/GR740_next-generation_microprocessor

Figure 5: The difference between a bit and a byte
Keshav's Blog: Bit vs Byte. (n.d.). Retrieved 8 March 2020, from http://keshavsps.blogspot.com/2017/11/bit-vs-byte.html

Figure 6: Triple Modular Redundancy
Wikipedia. (n.d.). Triple Modular Redundancy. Retrieved December 11, 2019, from https://en.wikipedia.org/wiki/Triple_modular_redundancy#/media/File:Triple_Modular_Redundancy.JPG

Figure 7: The Hamming code
Wikipedia. (n.d.). Hamming code. Retrieved December 11, 2019, from https://en.wikipedia.org/wiki/Hamming_code#/media/File:Hamming(7,4).svg

14.2. PROJECT PLAN SOURCES

14.2.1. SITES

Esa. (z.d.-a). ESA Microelectronics Section. Retrieved 28 September 2019, from https://www.esa.int/Our_Activities/Space_Engineering_Technology/Microelectronics/ESA_Microelectronics_Section

Esa. (z.d.-b). What is ESA? Retrieved 28 September 2019, from https://www.esa.int/About_Us/Welcome_to_ESA/What_is_ESA

Esa. (z.d.-c). ESTEC: European Space Research and Technology Centre. Retrieved 28 September 2019, from http://www.esa.int/About_Us/ESTEC/ESTEC_European_Space_Research_and_Technology_Centre

Esa. (z.d.-d). Systems and software engineering. Retrieved 28 September 2019, from https://www.esa.int/Our_Activities/Space_Engineering_Technology/Systems_and_software_engineering

14.2.2. FIGURES

ESA logo: Satellite Telemetry and Command Solutions. (n.d.). Retrieved 28 September 2019, from https://www.renesas.com/kr/en/solutions/key-technology/rad-hard/satellite-telemetry.html

CLD logo: Christelijk Lyceum Delft, Delft | Technasium.nl. (n.d.). Retrieved 28 September 2019, from https://www.technasium.nl/netwerken/technasium/zuid-holland/christelijk-lyceum-delft-delft

Figure 1: Satellite Telemetry and Command Solutions. (n.d.). Retrieved 28 September 2019, from https://www.renesas.com/kr/en/solutions/key-technology/rad-hard/satellite-telemetry.html

Figure 2: De 7 Bèta werelden. (n.d.). Retrieved 28 September 2019, from https://www.kajmunk.nl/Technasium/De-7-B%C3%A8ta-werelden

15. ATTACHMENTS

15.1. ATTACHMENT 1: WRITTEN PERMISSION OF ESA LOGO

This project has been executed in cooperation with ESA, and the use of the ESA logo for the report and presentation is permitted.

15.2. MINUTES OF MEETING

15.2.1. MINUTES OF MEETING 1

An interim presentation took place on 20-11-2019. It started with the orientation about bitwise operators, mathematical models, Hamming code, ecc ram, hash, checksum, cryptography, peek and poke memory. The specification of requirements and the chosen software were shown to the client.

It was noted that a presentation should not only show the internal details of the project to be discussed in a progress meeting. As soon as ‘external audience’ is present, the presentation should also contain a general introduction about the project and its goals, for those who are not directly involved in the project.

After the presentation, there was a consultation about how the project should proceed and what to do about the software.

All codes need different protections. It always need additional storage. The degree of protection differ. Space radiation does not impact runtime, but redundancy does. There is always a price to pay for protection.

A suggestion from the client was to test two types of protections (or if there is more time, more types can be tested). One of the protection was by using triple modular redundancy (TMR). Every bit is triplicated (= stored three times). After the computer has read the triple bits from memory, it will compare and check which the good code was. So when it had sent 111 111 111 and the computer received 111 101 111, it will know it was 111. It takes more storage and has a longer runtime. The other protection is by using the Hamming code. Protection is always by adding redundant data.

Another protection that the researchers do not have to use, but it may be useful, is by using matrices. It takes all the bits and stores them in a matrix. For example, 16 variables of 32-bit width could be stored in a matrix of 32 columns and 16 rows. You can add 1 row of for the parity bits over the columns, and one column for the parity bits over the rows. When you read data, you can check the parity bit of rows and columns. If a bit is flipped, you know exactly which one.

The program is running, and to simulate the effect of space radiation which can arrive at any (random) moment during the execution of the program, we should run fault injection in a separate program, running independently in a multi-tasking environment. Since this is difficult to implement, it was agreed that the application program will be “instrumented” with a ‘pseudo-random’ fault injection.

To make the injection, identify all variables, put in an array and start with one variable (random and take all program and take into a block of memory). In the loop must be the injection. At first, do one injection into one single bit of only one variable per run. So in fact, you need three random numbers: time, which variable, which bit inside the variable.

For many space applications, we need to provide an expected failure rate to the customer, in other words a statistical measure of how many program failures to expect per time unit, or Failures-In-Time (FIT). Ideally the FIT should be 0, but this is sometimes difficult to reach. At the same time, the radiation effects are characterized by the rate of how many bit-flips per time unit occur, the so-called Bit-Error-Rate (BER). The procedure to determine the FIT rate as a function of BER is the following:

- a) We measure the run-time T0 of the program without fault injection.
- b) If we inject 1 error per run, the BER = 1/T0, if 2 errors per run, BER = 2/T0 etc.
- c) We perform N runs of random fault injection and count how many runs experienced an error (Nf). The FIT can then be calculated as FIT = Nf / (N * T0).

Focus on the data program

The hints the client has given:
- Try and continue coding, but if you're stuck send it to me and send the code.
- It is important to know what happens in computer and in the data. What is the black box doing? - It is important to understand each line.

For example, a "for" loop does multiple accesses to the loop variable:
for (i = 0; i < N; i++) {
…..
}
The loop variable I is accessed many times:

- At start of loop, it is initialised (means: **write** 0)
- At the end of every iteration, it is **read**, then compared with the end value, incremented (add +1), and then we **write** back to memory.

If you want to add data protection to the loop variable, then you have to make sure that the redundancy (triplication or Hamming code) is **generated at every write** and **checked at every read.**

So most likely you will have to re-write your program, and decompose every complex statement into individual memory read- and write- operations.

Besides the failure rate (FIT), your customer also wants to know the price he has to pay for the protection. The "price" can be expressed in terms of execution speed or memory usage. Usually, a compromise between protection and cost is necessary. There is no perfect world, there is no "0 faults at 0 cost"

- Measure the duration of execution of program -> gives matrix and rate of bitflips.

- Which of code is most efficient?
- What is the storage while the program is running and measure the duration
- It is also important to have a virtual box (linux). It is a computer in a computer
- Then you have a command time to measure the duration.

→ get linux control and use time function. Watch out: The time function gives several output values. We are not interested in the wall-clock time, we are interested only in the CPU time (user and system), and there are usually different time functions available on a linux system, read the manual! To get better statistics it is advised to run the program many times in a loop (for example: run 100 times and divide the result by 100).

- To measure the run-time, you have to remove the fault injection and other diagnostic, in particular delete printf, because not necessarily because it has an impact on the runtime. Try and use conditional codes. All this code (fault injection + diagnostic) can be made conditional by wrapping it between:

#ifdef <condition>
…..
#endif

Use two condition parameters: INJ for the code doing the injection, and DIAG for any other diagnostic output.

#ifdef INJ
… *code for fault injection …*
#endif

#ifdef DIAG
…. *printf …*
#endif

Note that the #ifdef can be used many times in the program, wherever you have conditional code.

You can then compile the code with or without the parameter:

gcc -DINJ → to compile with the injection code
gcc -DDIAG → to compile with the diagnostic code
gcc -DINJ -DDIAG → to compile with both

15.2.2. MINUTES OF MEETING 2

A second interim presentation took place on 29-01-2020. We covered the actions we have done, the added sub-questions, the code, and the results. The presentation ended with what we have planned to do in the next weeks and the expectations of the client were asked.

Afterward, there was a conversation about how we could make the presentation clear and understandable, yet in-depth. The first solution is to do the presentation in Dutch, so there will not be a big language barrier. We should also try to visualize subjects that are hard to understand, for example with illustrations and animations that explain how bits are manipulated.

Furthermore, we should be clear why we do this project and why it is so relevant in the end presentation. Firstly, the introduction: ‘Where did we start? What is the problem? (Radiation, protons, and ions for example) What if it fails? What can we do to prevent this from happening?’ We should also explain the principle of the bit flipping.

In the report, we should talk about the needs of a customer. Some customers want a lower runtime, other customers want a shorter code, and others would prefer less storage occupation. We could check how many kilobytes are used in a program with a matrix.

What is said too, is that a computer is a non-real time operating system and this will affect the time measured by the laptop when we do the research. We also should measure the runtime (T0) with the program that is not affected by the injection. In order to do that, wrap the injection code by #ifdef INJECTION and re-compile without INJECTION parameter. We should also do a test, for example, 10 times to be more accurate. Furthermore, the formula of BER is possibly wrong. it was agreed that the client would look into it (*) and that we would also search the internet ourselves. At last, we should use comments in the code to make clear what the code does.

(*) The formula of the BER is not wrong as such, a “rate” is always a number of events divided by the time during which these events have occurred. However, I think it was wrongly understood. For the BER, you divided the number of faulty runs by the time. The misunderstanding is the interpretation of the word “error”. Let’s distinguish two types of error:

1. The errors you inject (= bit-flips)
2. The errors in the execution of the program (= functional failures)

The BER, in this case, should be understood as the first type of errors, hence the basic rate of injection: how many errors do we inject into our design (per time unit)? Consequently, the BER should be the total number of injections (faulty or not!) divided by the total CPU time. The BER is a dependent variable, but not a result. It depends on other variables you can control, namely the execution time and the number of bits you inject per run. If there was time, the program could be modified for example to inject two or more bits per run, for example into different variables (not only the loop variable i). The BER would then be higher, because you are injecting more errors in the same execution time. Note again that the BER is not a result, it just describes what you have done. It will allow the user to judge if your test conditions correspond to his real application conditions.

For the FIT rate, I would then take the definition “Failures In Time”, so here you should take the number of functional failures divided by the execution time. This is actually the RESULT of your protection: the FIT with protection should be much higher than without protection.

The interesting things for the client are:
1. How far can we go in the limited amount of time?
2. To what extent can we model the injection, so it will simulate the reality?
3. The most important question is: how much can you improve the fault tolerance by your design? For report and presentation it will be very important to show the following essential information:

a) How much can we reduce the FIT with our protection methods (TMR, Hamming)
b) What is the price in execution time (protected / unprotected, without printf, without injection)
c) What is the price in code size. If you compile with gcc, compare the size of the executable programs (without printf, without injection)

If we have time left, we could try to make more complex programs redundant against bitflips. First, we want to test the protected programs and finish the end-report and end presentation.

15.3. ATTACHMENT 2: RESULTS

15.3.1. BASELINE MEASUREMENT

Low numbers:

Runs (N)	ERROR (Nf)	NO ERROR	T0	T0 per run	BER (Ni/T0)	FIT (Nf/T0)
10	10	0	4,00000E-06	4,00000E-07	3,75000E+07	2,50000E+06
10	10	0	6,00000E-06	6,00000E-07	2,50000E+07	1,66667E+06
10	10	0	6,00000E-06	6,00000E-07	2,50000E+07	1,66667E+06
20	20	0	8,00000E-06	4,00000E-07	3,75000E+07	2,50000E+06
20	20	0	6,00000E-06	3,00000E-07	5,00000E+07	3,33333E+06
20	20	0	7,00000E-06	3,50000E-07	4,28571E+07	2,85714E+06
30	30	0	6,00000E-06	2,00000E-07	7,50000E+07	5,00000E+06
30	30	0	8,00000E-06	2,66667E-07	5,62500E+07	3,75000E+06
30	30	0	8,00000E-06	2,66667E-07	5,62500E+07	3,75000E+06
40	40	0	8,00000E-06	2,00000E-07	7,50000E+07	5,00000E+06
40	40	0	8,00000E-06	2,00000E-07	7,50000E+07	5,00000E+06
40	40	0	9,00000E-06	2,25000E-07	6,66667E+07	4,44444E+06
50	50	0	8,00000E-06	1,60000E-07	9,37500E+07	6,25000E+06
50	50	0	8,00000E-06	1,60000E-07	9,37500E+07	6,25000E+06
50	50	0	1,30000E-05	2,60000E-07	5,76923E+07	3,84615E+06

High numbers:

Runs (N)	ERROR (Nf)	NO ERROR	T0	T0 per run	BER (Ni/T0)	FIT (Nf/T0)
1000000	994830	5170	0,044193	4,41930E-08	3,39420E+08	2,25110E+07
1000000	994837	5163	0,046678	4,66780E-08	3,21351E+08	2,13128E+07
1000000	994837	5163	0,048653	4,86530E-08	3,08306E+08	2,04476E+07
2000000	1989774	10226	0,083456	4,17280E-08	3,59471E+08	2,38422E+07
2000000	1989661	10339	0,082691	4,13455E-08	3,62796E+08	2,40614E+07
2000000	1989615	10385	0,078022	3,90110E-08	3,84507E+08	2,55007E+07
3000000	2984566	15434	0,121249	4,04163E-08	3,71137E+08	2,46152E+07
3000000	2984621	15379	0,121900	4,06333E-08	3,69155E+08	2,44842E+07
3000000	2984808	15192	0,122281	4,07603E-08	3,68005E+08	2,44094E+07
4000000	3979425	20575	0,160175	4,00438E-08	3,74590E+08	2,48442E+07
4000000	3979326	20674	0,163493	4,08733E-08	3,66988E+08	2,43394E+07
4000000	3979539	20461	0,158301	3,95753E-08	3,79025E+08	2,51391E+07
5000000	4974173	25827	0,203446	4,06892E-08	3,68648E+08	2,44496E+07
5000000	4974310	25690	0,198412	3,96824E-08	3,78001E+08	2,50706E+07
5000000	4974185	25815	0,200268	4,00536E-08	3,74498E+08	2,48376E+07
6000000	5968816	31184	0,235682	3,92803E-08	3,81870E+08	2,53257E+07

6000000	5968904	31096	0,238000	3,96667E-08	3,78151E+08	2,50794E+07
6000000	5968814	31186	0,239565	3,99275E-08	3,75681E+08	2,49152E+07
7000000	6963924	36076	0,277999	3,97141E-08	3,77699E+08	2,50502E+07
7000000	6963946	36054	0,279295	3,98993E-08	3,75947E+08	2,49340E+07
7000000	6963627	36373	0,275919	3,94170E-08	3,80546E+08	2,52379E+07
8000000	7958754	41246	0,315877	3,94846E-08	3,79895E+08	2,51957E+07
8000000	7958657	41343	0,311460	3,89325E-08	3,85282E+08	2,55527E+07
8000000	7958770	41230	0,323244	4,04055E-08	3,71237E+08	2,46216E+07
9000000	8953801	46199	0,355965	3,95517E-08	3,79251E+08	2,51536E+07
9000000	8954033	45967	0,353779	3,93088E-08	3,81594E+08	2,53097E+07
9000000	8953663	46337	0,354534	3,93927E-08	3,80782E+08	2,52547E+07
10000000	9948178	51822	0,392401	3,92401E-08	3,82262E+08	2,53521E+07
10000000	9948656	51344	0,392401	3,92401E-08	3,82262E+08	2,53533E+07
10000000	9948374	51626	0,394226	3,94226E-08	3,80492E+08	2,52352E+07

15.3.2. BASELINE MEASUREMENT TMR

Low number of runs:

Runs (N)	ERROR (Nf)	NO ERROR	T0	T0 per run	BER (Ni/T0)	FIT (Nf/T0))
10	0	10	7,00000E-06	7,00000E-07	2,14286E+07	0
10	0	10	8,00000E-06	8,00000E-07	1,87500E+07	0
10	0	10	4,00000E-06	4,00000E-07	3,75000E+07	0
20	0	20	1,00000E-05	5,00000E-07	3,00000E+07	0
20	0	20	1,20000E-05	6,00000E-07	2,50000E+07	0
20	0	20	7,00000E-06	3,50000E-07	4,28571E+07	0
30	0	30	1,00000E-05	3,33333E-07	4,50000E+07	0
30	0	30	1,30000E-05	4,33333E-07	3,46154E+07	0
30	0	30	1,20000E-05	4,00000E-07	3,75000E+07	0
40	0	40	1,60000E-05	4,00000E-07	3,75000E+07	0
40	0	40	1,20000E-05	3,00000E-07	5,00000E+07	0
40	0	40	1,40000E-05	3,50000E-07	4,28571E+07	0
50	0	50	1,30000E-05	2,60000E-07	5,76923E+07	0
50	0	50	1,40000E-05	2,80000E-07	5,35714E+07	0
50	0	50	1,30000E-05	2,60000E-07	5,76923E+07	0

High number of runs:

Runs (N)	ERROR (Nf)	NO ERROR	T0	T0 per run	BER (Ni/T0)	FIT (Nf/T0))
1000000	0	1000000	0,096663	9,66630E-08	1,55178E+08	0,00000E+00
1000000	0	1000000	0,096023	9,60230E-08	1,56213E+08	0,00000E+00
1000000	0	1000000	0,097291	9,72910E-08	1,54177E+08	0,00000E+00
2000000	0	2000000	0,187829	9,39145E-08	1,59720E+08	0,00000E+00

2000000	0	2000000	0,186924	9,34620E-08	1,60493E+08	0,00000E+00
2000000	0	2000000	0,187799	9,38995E-08	1,59745E+08	0,00000E+00
3000000	0	3000000	0,275278	9,17593E-08	1,63471E+08	0,00000E+00
3000000	0	3000000	0,276233	9,20777E-08	1,62906E+08	0,00000E+00
3000000	0	3000000	0,278371	9,27903E-08	1,61655E+08	0,00000E+00
4000000	0	4000000	0,361218	9,03045E-08	1,66105E+08	0,00000E+00
4000000	0	4000000	0,367737	9,19343E-08	1,63160E+08	0,00000E+00
4000000	0	4000000	0,368398	9,20995E-08	1,62867E+08	0,00000E+00
5000000	0	5000000	0,453992	9,07984E-08	1,65201E+08	0,00000E+00
5000000	0	5000000	0,456245	9,12490E-08	1,64385E+08	0,00000E+00
5000000	0	5000000	0,454238	9,08476E-08	1,65112E+08	0,00000E+00
6000000	0	6000000	0,545526	9,09210E-08	1,64978E+08	0,00000E+00
6000000	0	6000000	0,547766	9,12943E-08	1,64304E+08	0,00000E+00
6000000	0	6000000	0,545857	9,09762E-08	1,64878E+08	0,00000E+00
7000000	0	7000000	0,632084	9,02977E-08	1,66117E+08	0,00000E+00
7000000	0	7000000	0,637521	9,10744E-08	1,64700E+08	0,00000E+00
7000000	0	7000000	0,635102	9,07289E-08	1,65328E+08	0,00000E+00
8000000	0	8000000	0,731809	9,14761E-08	1,63977E+08	0,00000E+00
8000000	0	8000000	0,726064	9,07580E-08	1,65275E+08	0,00000E+00
8000000	0	8000000	0,725129	9,06411E-08	1,65488E+08	0,00000E+00
9000000	0	9000000	0,813194	9,03549E-08	1,66012E+08	0,00000E+00
9000000	0	9000000	0,818672	9,09636E-08	1,64901E+08	0,00000E+00
9000000	0	9000000	0,815573	9,06192E-08	1,65528E+08	0,00000E+00
10000000	0	10000000	0,905103	9,05103E-08	1,65727E+08	0,00000E+00
10000000	0	10000000	0,904813	9,04813E-08	1,65780E+08	0,00000E+00
10000000	0	10000000	0,909567	9,09567E-08	1,64914E+08	0,00000E+00

15.3.3. BASELINE MEASUREMENT HAMMING CODE

Low number of runs:

Runs (N)	ERROR (Nf)	NO ERROR	T0	T0 per run	BER (Ni/T0)	FIT (Nf/T0))
10	0	10	1,7E-05	1,7E-06	8,8E+06	0
10	0	10	2,0E-05	2,0E-06	7,5E+06	0
10	0	10	1,6E-05	1,6E-06	9,4E+06	0
20	0	20	2,8E-05	1,4E-06	1,1E+07	0
20	0	20	2,9E-05	1,5E-06	1,0E+07	0
20	0	20	2,8E-05	1,4E-06	1,1E+07	0
30	0	30	4,0E-05	1,3E-06	1,1E+07	0
30	0	30	4,0E-05	1,3E-06	1,1E+07	0
30	0	30	4,0E-05	1,3E-06	1,1E+07	0
40	0	40	5,2E-05	1,3E-06	1,2E+07	0
40	0	40	5,2E-05	1,3E-06	1,2E+07	0
40	0	40	5,1E-05	1,3E-06	1,2E+07	0
50	0	50	6,3E-05	1,3E-06	1,2E+07	0

50	0	50	6,4E-05	1,3E-06	1,2E+07	0
50	0	50	6,4E-05	1,3E-06	1,2E+07	0

High number of runs:

Runs (N)	ERROR (Nf)	NO ERROR	T0	T0 per run	BER (N/T0)	FIT (Nf/T0))
1000000	0	1,0E+6	0,422597	4,22597E-07	3,549480E+07	0,00
1000000	0	1,0E+6	0,398128	3,98128E-07	3,767632E+07	0,00
1000000	0	1,0E+6	0,40173	4,01730E-07	3,733851E+07	0,00
2000000	0	2,0E+6	0,790852	3,95426E-07	3,793377E+07	0,00
2000000	0	2,0E+6	0,786612	3,93306E-07	3,813824E+07	0,00
2000000	0	2,0E+6	0,793053	3,96527E-07	3,782849E+07	0,00
3000000	0	3,0E+6	1,172045	3,90682E-07	3,839443E+07	0,00
3000000	0	3,0E+6	1,153052	3,84351E-07	3,902686E+07	0,00
3000000	0	3,0E+6	1,181307	3,93769E-07	3,809339E+07	0,00
4000000	0	4,0E+6	1,572541	3,93135E-07	3,815480E+07	0,00
4000000	0	4,0E+6	1,577375	3,94344E-07	3,803787E+07	0,00
4000000	0	4,0E+6	1,578787	3,94697E-07	3,800385E+07	0,00
5000000	0	5,0E+6	1,974059	3,94812E-07	3,799278E+07	0,00
5000000	0	5,0E+6	1,970278	3,94056E-07	3,806569E+07	0,00
5000000	0	5,0E+6	1,972974	3,94595E-07	3,801367E+07	0,00
6000000	0	6,0E+6	2,370371	3,95062E-07	3,796873E+07	0,00
6000000	0	6,0E+6	2,352587	3,92098E-07	3,825575E+07	0,00
6000000	0	6,0E+6	2,359207	3,93201E-07	3,814841E+07	0,00
7000000	0	7,0E+6	2,771851	3,95979E-07	3,788082E+07	0,00
7000000	0	7,0E+6	2,753554	3,93365E-07	3,813253E+07	0,00
7000000	0	7,0E+6	2,760613	3,94373E-07	3,803503E+07	0,00
8000000	0	8,0E+6	3,142613	3,92827E-07	3,818478E+07	0,00
8000000	0	8,0E+6	3,146563	3,93320E-07	3,813684E+07	0,00
8000000	0	8,0E+6	3,143318	3,92915E-07	3,817622E+07	0,00
9000000	0	9,0E+6	3,532263	3,92474E-07	3,821912E+07	0,00
9000000	0	9,0E+6	3,53961	3,93290E-07	3,813979E+07	0,00
9000000	0	9,0E+6	3,531802	3,92422E-07	3,822411E+07	0,00
10000000	0	1E+07	3,931704	3,93170E-07	3,815139E+07	0,00
10000000	0	1E+07	3,923453	3,92345E-07	3,823162E+07	0,00
10000000	0	1E+07	3,940124	3,94012E-07	3,806986E+07	0,00

15.4. ATTACHMENT 4: CODE PROTECTIONS

The Triple Modular Redundancy program:

```
///a code that counts from 0 to 15 and adds these up
///simulated bitflips will change these numbers
///the program corrects these bits with the use of triple mdular redundancy
#include <stdio.h>
#include <time.h>
```

```
clock_t start, end;
double cpu_time_used;

int main() {
	//make variables that count the errors and non-errors
	int nonerror = 0, error = 0;

	#ifdef BIT
	//using the internal time clock for a random number
	srand((unsigned)time(NULL));
	#endif

	//start the timer
	start = clock();

	//a for-statement to repeat the program a certain amount of times
	for(int j=0; j<1000000; j++){

		//make variables for counting from 0 to 15, the sum, and the tmr
		int n, sum = 0, data[3];

		for(n = 0; n < 16; n++) {
			//make the three variables from data[] equal to the number
			data[0] = n;
			data[1] = n;
			data[2] = n;

			#ifdef BIT
				//make variables that will be random numbers to simulate a bit-flip
				int r, r2;

				//get a random number for 'r' from 0 to 4
				r = rand() % 5;

				//get a random number for 'r' from 0 to 2
				r2 = rand() % 3;

				//flip the bit defined by 'r2' in 'data'
				data[r2] ^= (1 << r);
			#endif

			//compare the three vraiables of 'data[]' and make 'n' the correct one
			if(data[1] != data[2]) {
				if(data[2] == data[0]) {
					n = data[0];
				}
			}
			if(data[1] != data[0]) {
				if(data[1] == data[2]) {
					n = data[2];
				}
			}
			if(data[2] != data[0]) {
				if(data[0] == data[1]) {
					n = data[1];
				}
```

```
                }

                #ifdef NUM
                        //print the corrected number (0-15)
                        printf("%d ", n);
                #endif

                //add up n to sum
                sum += n;
        }

        //check if the sum is equal to 120
        if (sum == 120) {
                #ifdef SUM
                        //print the sum
                        printf("\nSum: %d\n", sum);
                #endif

                //add 1 to 'nonerror'
                nonerror++;
        } else {
                #ifdef ERROR
                        //print if a error has happened
                        printf("error! \n");
                #endif

                //add 1 to 'error'
                error++;
        }

    }
    //stop the timer
    end = clock();

    //calculate the time used by the timer
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    #ifdef DEBUG
        //print the runtime measured by the timer
        printf("Runtime:%f \n", cpu_time_used);

        //print the amount of errors and non-errors measured by 'count2' and 'count1'
        printf("error:%d, no error:%d ", error, nonerror);
    #endif

    return 0;
}
```

The Hamming code program:

```
///a code that counts from 0 to 15 and adds these up
///simulated bitflips will change these numbers
///the program corrects these bits with the use of Hamming code
#include<stdio.h>
#include<time.h>
```

```
clock_t start, end;
double cpu_time_used;

int main() {
	//make variables that count the errors and non-errors
	int nonerror = 0, error = 0;

	#ifdef BIT
		//using the internal time clock for a random number
		srand((unsigned)time(NULL));
	#endif

	//start the timer
	start = clock();

	//a for-statement to repeat the program a certain amount of times
	for(int j=0; j<1000000; j++){

		//make variables for counting from 0 to 15, the sum, the Hamming code, and correction
		int n, sum = 0, r, data[7], c;

		//a loop that adds 1 to 'i' until 15
		for(n = 0; n<16; n++) {
			//make a decimal number 'n' (0-15) to a binairy number 'data[]' (0-1)
			data[0]=n%2;
			n=n/2;

			data[1]=n%2;
			n=n/2;

			data[2]=n%2;
			n=n/2;

			data[4]=n%2;
			n=n/2;

			//Calculation of even parity
			data[6]=data[0]^data[2]^data[4];
			data[5]=data[0]^data[1]^data[4];
			data[3]=data[0]^data[1]^data[2];

			#ifdef BIT
				//make a variable that will be a random number to simulate a bit-flip
				int r;

				//get a random number for 'r' from 0 to 6
				r = rand() % 7;

				//flip the bit defined by 'r' in 'data'
				data[r] ^= (1 << 0);
			#endif

			//Calculation of the spot where the bit-flip happened
			c=(data[3]^data[2]^data[1]^data[0])*4+
				(data[5]^data[4]^data[1]^data[0])*2+
```

```
                (data[6]^data[4]^data[2]^data[0]);

            //Correction of the bit-flip
            if(c!=0) {

                if(data[7-c]==0) {
                    data[7-c]=1;
                } else {
                    data[7-c]=0;
                }
            }

            //Converting the binairy numbers 'data[]' to decimal numbers
            n = data[0] + data[1]*2 + data[2]*4 + data[4]*8;

            #ifdef NUM
                //print the corrected number (0-15)
                printf("%d ", n);
            #endif

            //add up n to sum
            sum += n;
        }
        //check if the sum is equal to 120
        if (sum == 120) {
            #ifdef SUM
                //print the sum
                printf("\nSum: %d\n", sum);
            #endif

            //add 1 to 'nonerror'
            nonerror++;
        } else {
            #ifdef ERROR
                //print if a error has happened
                printf("error! \n");
            #endif

            //add 1 to 'error'
            error++;
        }
    }
    //stop the timer
    end = clock();

    //calculate the time used by the timer
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    #ifdef DEBUG
        //print the runtime measured by the timer
        printf("Runtime:%f \n", cpu_time_used);

        //print the amount of errors and non-errors measured by 'count2' and 'count1'
        printf("error:%d, non-error:%d ", error, nonerror);
    #endif

    return 0;
```

```
}
```

15.5. ATTACHMENT 5: PROJECTPLAN

VWO 6, 19/20

Version 1.5.3
11-10-19

Natanael Djajadi
Kees de Hoogh
Amy van der Meijden

Projectplan

Redundant Software Technology

Docent: Jeanna de Haan

Meesterproef

Client: Roland Weigand

Company: European Space Agency

CONTENT

Introduction 49

Data 49

The students 50

Natanael Djajadi 50

Kees de Hoogh 50

Amy van der Meijden 50

The project 51

The client 51

His job 51

The company 51

Bètaworld 51

Problem definition 52

Research/design question 52

Project result 52

Delimination 52

Project steps 53

Planning 54

Sources 56

Sites 56

Figures 56

INTRODUCTION

Everywhere around us is radiation. In space, mostly space radiation. On earth, people are protected by the atmosphere. But when one launches electronics - like satellites - into space, they are not protected. Then software can be affected by the space radiation. Bit-flips (so-called SEU = Single Events Upsets) are common for digital electronics, like computer chips.

Figure 4 Satellite

If data bits are flipped, the value of a variable may be changed. If program bits are flipped, the computer may start executing wrong, undesired instructions, or even branch to a wrong address. Most of the time, the bit-flips are harmless, because a lot of data in computers are stored but never used. But in some cases, it can lead to crashes or to wrong decisions.

To cope with the SEUs, sometimes special computer chips are developed, but these are very expensive.

The project is focused on the protection of the data used in a program.

DATA

School: Christelijk Lyceum Delft
Location: Molenhuispad 1
Class: V6o&o1
Group: 3
Team name: Bits of Ank
Contractors: Natanael Djajadi, Kees de Hoogh, Amy van der Meijden
Docent: J. De Haan

Client: Roland Weigand
Company: European Space Agency,
European Space Research and Technology Centre
Location: Noordwijk

Date start: 04-09-2019
Date end: 18-03-2020

THE STUDENTS

NATANAEL DJAJADI

My name is Natanael Djajadi and I am 17 years old. I chose this project because I like projects related to software and computers. I really want to study Computer Science and Engineering (TU Delft) which is the reason I chose a project in this area. I am also interested in electronics related to space, like rocket ships or satellites. So I am very pleased with this assignment and I am really excited.

This project will be really beneficial to my English and I can learn a lot about programming from this project, which both can be very useful for my next study, Computer Science and Engineering.

Portfolio: https://natanaeldj.jimdo.com/

KEES DE HOOGH

My name is Kees de Hoogh, I am sixteen years old and I am interested in architecture, informatics and sport. I am planning to start a study in civil engineering or architecture. Although these studies have little in common with this current project, I am very excited about what we will to achieve in the upcoming months. Particularly, the programming to simulate disturbances in space and see whether we can find a solution for this problem. The conceptual thinking and translation into a practical result forms a good basis for an academic study.

Portfolio: https://keesdh.wixsite.com/kees

AMY VAN DER MEIJDEN

My name is Amy van der Meijden and I am sixteen years old. I am mainly interested in the mathematical subjects and especially Math and Computer Science, which I also want to study. I am very excited about this project because I think this project relates to the next study I will follow. Computer Science and Engineering is completely in English so I think this project will be a good exercise to improve my English as well.

Portfolio: https://amy.vandermeijden.net

THE PROJECT

THE CLIENT

HIS JOB

Our client, Mr. Weigand, is a chip designer, who focusses on redundancy. He ensures that the chips are robust against space radiation. He works in the department “Microelectronics Section” in ESTEC (European Space Research And Technology Centre), a branch of ESA.

THE COMPANY

Our project is for the ESA (European Space Agency), which is an international organisation. The mission of ESA is to shape the development of Europe’s space capability and to ensure that investment in space will deliver benefits to Europe and the world. It is ESA’s task to prepare and implement the European space programme. Some programmes are designed to find out more about Earth, our Solar System and Universe, to develop satellite-based technologies and services and much more.

ESA also has branches in some European countries, each with different responsibilities. The ESTEC, the European Space Research and Technology Centre, is located in Noordwijk, the Netherlands. The ESTEC is ESA’s technical heart, where most of ESA’s projects are born and where they are led through the different development phases. Mr. Weigand works here, in the section ‘Microelectronics’.

THE JOB

There are several separate elements which are needed to complete a space mission. A spacecraft itself consists of many elements with different individual subsystems developed by specialized teams. The client is a chip designer, but we will work in a different section, namely the Systems and Software Technology Section.

BÈTAWORLD

This project belongs to the Bètaworld Lifestyle & Design.

Today, all sorts of things from space, like GPS or weather prediction, are widely used by people and are involved in our lifestyle. In this project we will use these techniques and that is why this project belongs to Lifestyle & Design.

This project is about making the software redundant against space radiation. This can be beneficial for the future.

Figure 5 Lifestyle and Design

PROBLEM DEFINITION

As mentioned before in the introduction, systems on earth are protected from space radiation by the atmosphere. Satellites, and specifically chips in space are not protected and they have a chance of being affected by space radiation. This can cause bit-flips in chips, which will for instance lead to faults in the data or even a computer crash.

Imagine this happening in a rocket ship. Bit-flips could cause problems, like engine or steering failures. Such failures could be catastrophic and lead to a loss of the rocket or satellite, so the risk of the occurrence of bit-flips should be as small as possible.

Space agencies try to prevent this by designing chips to be more tolerant against space radiation. Our client is part of the section that is responsible for this. But it is difficult to solve this problem via hardware because manufacturing a chip just for space is very expensive. That is why we test tolerance by software backups, such that cheap commercial computer chips could be used as well.

RESEARCH QUESTION

Our research/design question is: ‘How to make a chosen software running on a commercial computer tolerant against faults induced by space radiation?’

PROJECT RESULT

The client expects, as a project result, to see statistics of the research that we have done. We will compare the results of the old and the new software (size and performance) and a demonstration of the new software that we developed.

DELIMINATION

We will focus on the principles how to make software tolerant against bit-flips. However, we will not use complex programs to develop our software. Hardware, such as chips, will not be discussed.

PROJECT STEPS

Step 1: Orientate in familiarisation with space radiation effects on chips and computers

3. *What are the effects of space radiation on the software?*

We will analyse literature and process the information. Here we filter out the most important pieces. We will search for additional information on the internet as well. By answering this question, we will understand how bit-flips originate.

4. *How to protect the software against bit-flips?*

First we will search for a software, which is easily compiled and executed in (non-graphic) command-line mode. Therefore, we will ask an expert or search for software and filter the software to use in our project. Then we become familiar with the software and test how it works. Finally, we find and analyse methods to protect software from bit-flips by literature and internet.

Step 2: Specification of requirements

We will make a specification of requirements, to get more clarity about the criteria of the research and end result.

Step 3: The research (I)

3. *How to make a program that randomly changes bits?*

We try to find an expert who has knowledge about bit-flips, informatics etc. With the help from the expert, we will make a program that randomly changes bits, to ensure the research will be reliable. By doing so, the program makes the changes stable and equivalent.

4. *Baseline measurement*

We use the program to change bits in the original, unprotected software. Then we will adjust the bits of the chosen software, so we perform fault injection. Then we look into and processing the results with statistics.

Step 4: The interim presentation

We will make an appointment with the client, so we can check together if we are on the right path. We also use this time to discuss how we can tackle present problems.

Step 5: The research (II)

3. *Add chosen method(s) to the software*

We will choose methods of our orientation to add to our software, so our software will be protected against bit-flips.

4. *Testing the protected program*

We use the program to adjusted, protected change bits in the software. We will then perform fault injection. Then we look into and processing the results with statistics.

Step 6: The second interim presentation

We will make an second appointment with the client, so we can check if we still are on the right path. We also use this time to discuss if our research is what he had in mind.

Step 7: End result - the report and presentation

3. *Make a report of our research*

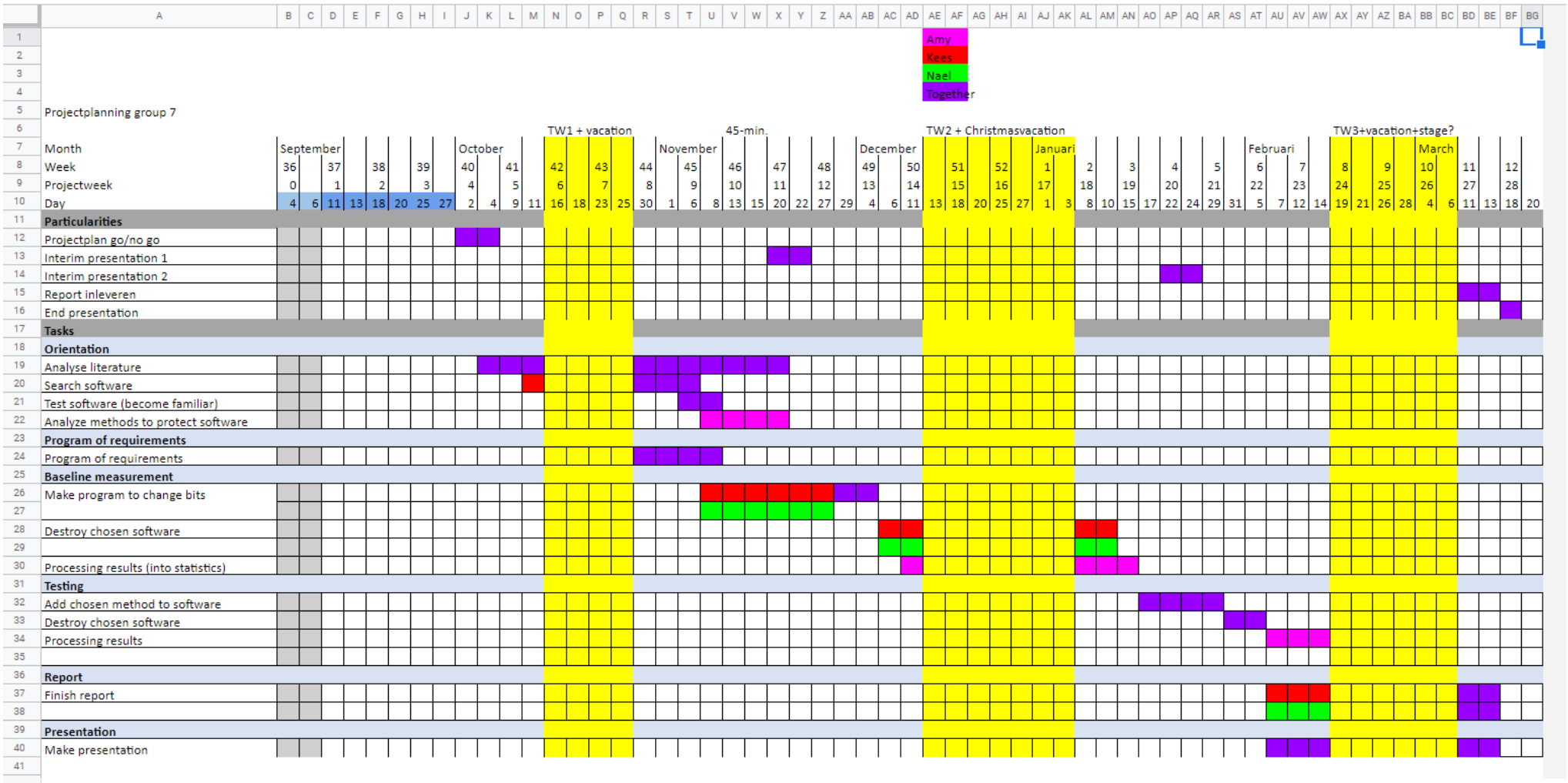
We will make a report of our research. It includes orientation, program of requirements, the research and the results of our research. At last, we make a conclusion and a discussion.

4. *Make a presentation of our research*

We will make a presentation with a demonstration and graphics. In this presentation we will show our research process and the outcome of it (supported by statistics).

PLANNING

On the next page


Amy
Kees
Nael
Together
Projectplanning group 7
TW1 + vacation
45-min.
TW2 + Christmasvacation
TW3+vacation+stage?
Month
September
October
November
December
Januari
Februari
March
Week
Projectweek
Day
Particularities
Projectplan go/no go
Interim presentation 1
Interim presentation 2
Report inleveren
End presentation
Tasks
Orientation
Analyse literature
Search software
Test software (become familiar)
Analyze methods to protect software
Program of requirements
Program of requirements
Baseline measurement
Make program to change bits
Destroy chosen software
Processing results (into statistics)
Testing
Add chosen method to software
Destroy chosen software
Processing results
Report
Finish report
Presentation
Make presentation

SOURCES

SITES

Esa. (z.d.-a). ESA Microelectronics Section. Geraadpleegd op 28 september 2019, van https://www.esa.int/Our_Activities/Space_Engineering_Technology/Microelectronics/ESA_Microelectronics_Section

Esa. (z.d.-b). What is ESA? Geraadpleegd op 28 september 2019, van https://www.esa.int/About_Us/Welcome_to_ESA/What_is_ESA

Esa. (z.d.-c). ESTEC: European Space Research and Technology Centre. Geraadpleegd op 28 september 2019, van http://www.esa.int/About_Us/ESTEC/ESTEC_European_Space_Research_and_Technology_Centre

Esa. (z.d.-d). Systems and software engineering. Geraadpleegd op 28 september 2019, van https://www.esa.int/Our_Activities/Space_Engineering_Technology/Systems_and_software_engineering

FIGURES

ESA logo: Satellite Telemetry and Command Solutions. (z.d.). Geraadpleegd op 28 september 2019, van https://www.renesas.com/kr/en/solutions/key-technology/rad-hard/satellite-telemetry.html

CLD logo: Christelijk Lyceum Delft, Delft | Technasium.nl. (z.d.). Geraadpleegd op 28 september 2019, van https://www.technasium.nl/netwerken/technasium/zuid-holland/christelijk-lyceum-delft-delft

Figure 1: Satellite Telemetry and Command Solutions. (z.d.). Geraadpleegd op 28 september 2019, van https://www.renesas.com/kr/en/solutions/key-technology/rad-hard/satellite-telemetry.html

Figure 2: De 7 Bèta werelden. (z.d.). Geraadpleegd op 28 september 2019, van https://www.kajmunk.nl/Technasium/De-7-B%C3%A8ta-werelden