Imperial College London

Department of Computing

Fault-Tolerant Control Systems for Nanosatellites in Low Earth Orbit

by
Ivan Avanessov

Submitted in partial fulfilment of the requirements for the MSc Degree in Computing Science of Imperial College London

Supervisor:
Prof. Julie McCann

9 September 2016

Abstract

Space has always been an interesting and appealing area of research. However, space exploration is considered very expensive and often unreliable. Nanosatellites, such as CubeSats and PocketQubes, are a very new concept in terms of space research and currently small spacecraft are gaining more and more interest not only from researchers and educational institutions. The main idea behind nanosatellites is to make space exploration affordable to the broader public and very often they are made from commercial-off-the-shelf (COTS) components to keep the costs low.

COTS electronics are not designed for functioning in outer space and require both hardware and software protection against possible errors, which await in space. Since, it is quite a new area of research, it still lacks standardization and there are as yet very few commonly accepted solutions for an on-board maintenance program.

This project aims to develop a coherent and functional fault-tolerant and self-healing software framework, which is designed for COTS microcontrollers that are planned to be used as main computing and maintenance units in a nanosatellite, and implement it on test devices. The framework also performs a variety of other tasks as a regular space mission may require. Microcontrollers are very limited in memory size and processing speed, thus, the framework needs to accommodate these limitations.

To develop such a software framework, it was first necessary to carefully evaluate the space environment and discover the sources of risks for a regular microcontroller. It was established that Total Ionizing Dose is unlikely to pose any threats to a satellite in LEO as the satellite will not stay in space long enough to get sufficient amounts of radiation. The temperature for a satellite, which orbits the Earth in LEO, can fluctuate from -30 to 50°C, which is in the survival range of COTS electronics. However, the major sources of risks are Single Event Effects in the form of bit flips anywhere in a chip, and angular rotation resulting in an unstable power supply and program interruptions

Therefore, the framework is designed to perform on-board maintenance (via the I^2C communication protocol) and various deployment procedures (hardware pin electric current manipulations). It consists of methods, which do not take longer than 1-2 sec to run individually and can identify faults in CPU, ALU, RAM, permanent storage and program code storage. In addition, the framework can fix these faults, store variables reliably, track its progress and perform self-healing as necessary using checksums and Triple Modular Redundancy algorithms.

This work should serve as a base for a larger project – PAMSAT, and as a point of reference for other research institutions and universities, which are keen on exploring nanosatellite standards and building or designing a spacecraft of their own. The project outlines the most significant threats to a COTS microcontroller and how those threats should be dealt with, which saves a lot of time for those who have a specific plan for their own nanosatellite and need a framework to incorporate.

Acknowledgments

I would like to thank my supervisor, Professor Julie McCann, for giving me this unique opportunity to work on this amazingly interesting and challenging project and for her valuable support and guidance throughout.

I am also extremely grateful to Michael Johnson, PAMSAT project manager, to whom I am highly indebted for initiating my interest in this area, for his never-ending enthusiasm and unyielding support in this work. His advice and guidance were fantastic and have helped me enormously.

I would like to thank Luke Bussell and Benhur Johnson, for both a very informative and enjoyable experience working as part of the PAMSAT team.

I wish to express my sincere gratitude to my parents, my family and my friends for endless patience and support during the term of this project.

Contents

1 Acronyms and Glossary

1P	One unit PocketQube (5 x 5 x 5cm)
1U	One unit CubeSat (10 x 10 x 10cm)
AESE	Adaptive Emerging Systems Engineering
COTS	Commercial off-the-shelf
EEE	Imperial College London Electronic and Electrical Engineering building
ESD	Electrostatic Discharge
FTCS	Fault Tolerant Control System
GS	Ground station
I2C	Inter Integrated Circuit Communications
IDE	Integrated Development Environment
LEO	Lower Earth Orbit (400-650km)
MIC	Mission Interface Computer
MISO	Master In Slave Out
MOSI	Master Out Slave In
PAMSAT	Parametric Advanced Membrane Satellite and Testbed
SCL	Serial Clock Line
SDA	Serial Data Line
SEE	Single Event Effect
SEU	Single Event Upset
SPI	Serial-Peripheral Interface
SRAM	Static Random-Access Memory
SS	Slave Select
TID	Total Ionizing Dose
TMR	Triple Modular Redundancy
TWIST	High performance thin film small satellite subsystems with a twist
UHF	Ultra-High Frequency
USB	Universal Serial Bus

2 Introduction

2.1 Motivation

The age of our universe according to astronomers is around 13.8 billion years [1], the observable universe has a diameter of about 21 billion parsecs, which is 91 billion light years [2], and today it is still impossible to confirm whether there is life on other planets, what dark matter consists of, or what is dark energy [3]. With the number of stars approximately equal to the number of grains of sand on all the beaches on our planet combined [2], the universe itself becomes an area of research as huge as ones imagination can allow it to be with an enormous potential for ground-breaking discoveries.

FIGURE 1: VIRGIN GALACTIC CONCEPT SPACECRAFT [8]

The exploration of our solar system began in 1959 [4] and to this day the total distance driven by rovers on Mars is approximately 37.2 miles [5]. One could argue that our exploration of Mars so far can be compared to exploring Earth "through a window of a train from London to Oxford" [6]. Indeed, there are only a handful of active space missions, which are engineered to study the Earth and other planets in our solar system, including but not limited to Saturn, Neptune, their moons and other solar systems [7]. Moreover, so far only Voyager 1, a spacecraft launched in 1977, on 25th of August 2012 has become the first and only manmade object to leave our solar system and is currently more than 20 billion kilometres away from Earth [9]. Voyager 2 is currently "right behind" it at more than 16.5 billion kilometres away from the Earth (as of 9 September 2016).

FIGURE 2: VOYAGER 1 (LEFT) AND VOYAGER 2 (RIGHT) [9]

The study of outer space was initially developed as a very costly area of research and a good example of it is the cost of Voyager 1 and Voyager 2 space missions, which were 3.6 billion USD each (Figure 3). The average cost of a Space Shuttle launch was approximately 450 million USD per mission [10]. There had to emerge another way of exploring the universe, which would be noticeably cheaper and affordable not only to giant wealthy corporations, but also to universities and smaller research institutions.

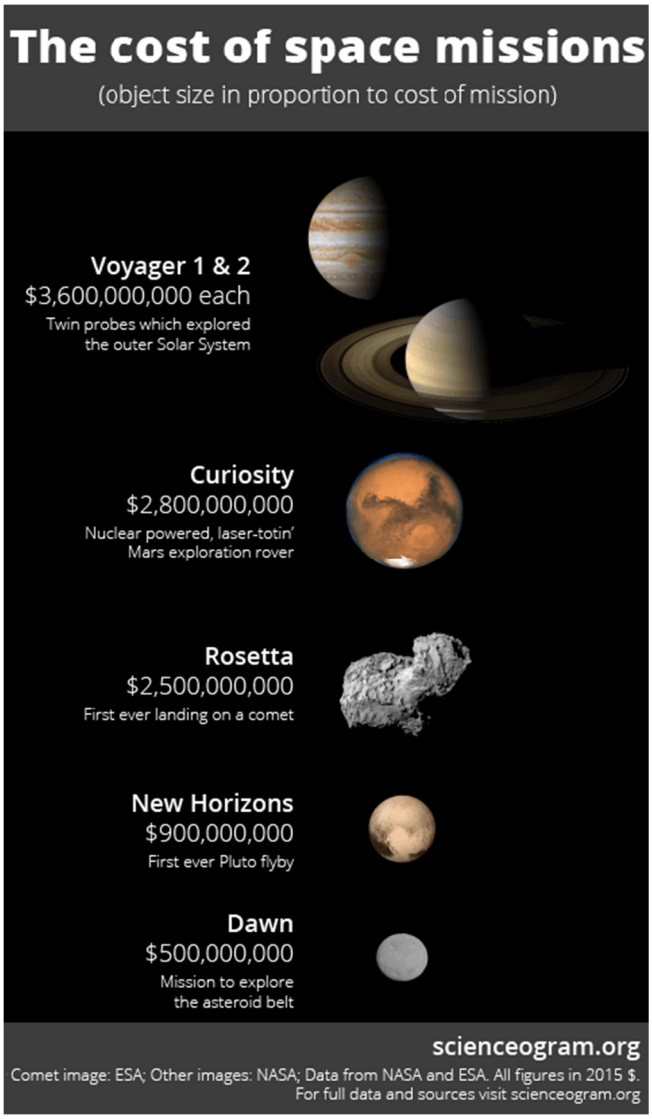

FIGURE 3: COST OF SPACE MISSIONS [13]

The breakthrough in the reduction of space mission costs happened in 1999, when a new satellite standard emerged – CubeSats. A CubeSat is a spacecraft, belonging to the category of microsatellites, defined around a standard 1U unit of 10 x 10 x 10cm (Figure 4). Currently CubeSats are launched into space attached as a secondary payload to multi-million USD space missions. The typical cost to launch a 1U CubeSat into Low Earth Orbit (LEO), which is from about 300 kilometres to about 1000km above the Earth's surface, is around 100,000 USD [11] There are also private companies working on building rockets specifically designed to deliver CubeSats to LEO. Therefore, if you use commercial off-the-shelf (COTS) components to build a spacecraft, it becomes noticeably cheaper than a usual space mission, which is exactly the case for most CubeSats that are launched into LEO or further.

Another standard, which was first defined in 2009 and derived from the CubeSat format, is the PocketQube [12]. The size of a standard PockeQube is 1P, which is a fraction of a 1U and is 5 x 5 x 5 cm (Figure 4). The purpose of nanosatellites such as PocketQubes and CubeSats is to perform space science and help universities around the world explore and research space through designing, producing and launching low-cost satellites made of low cost off-the-shelf components. Overall, nanosatellites are a very new yet very rapidly developing area of research.

Since the space environment is very extreme and nanosatellites are mostly built from COTS components and have small physical size, it is crucial for the space mission success to ensure that the on-board electronics can withstand being in space and can perform the required tasks with limited computing power. Therefore, it is essential to develop a piece of software, which provides the housekeeping for the Mission Interface Computer (MIC) and is resistant to the most likely threats of space. A MIC is a reconfigurable computer which manages all the on-board devices and sensors as required by a space mission. Hence, the purpose of this project can be summarised as follows:

- **Evaluate the most common and likely types of software failure in space**
- **Develop fault tolerant self-healing software, which is resistant to the most common types of failure, for a nanosatellite**

- **Make the software as lightweight as possible to satisfy the memory restrictions of the on-board computer chips**

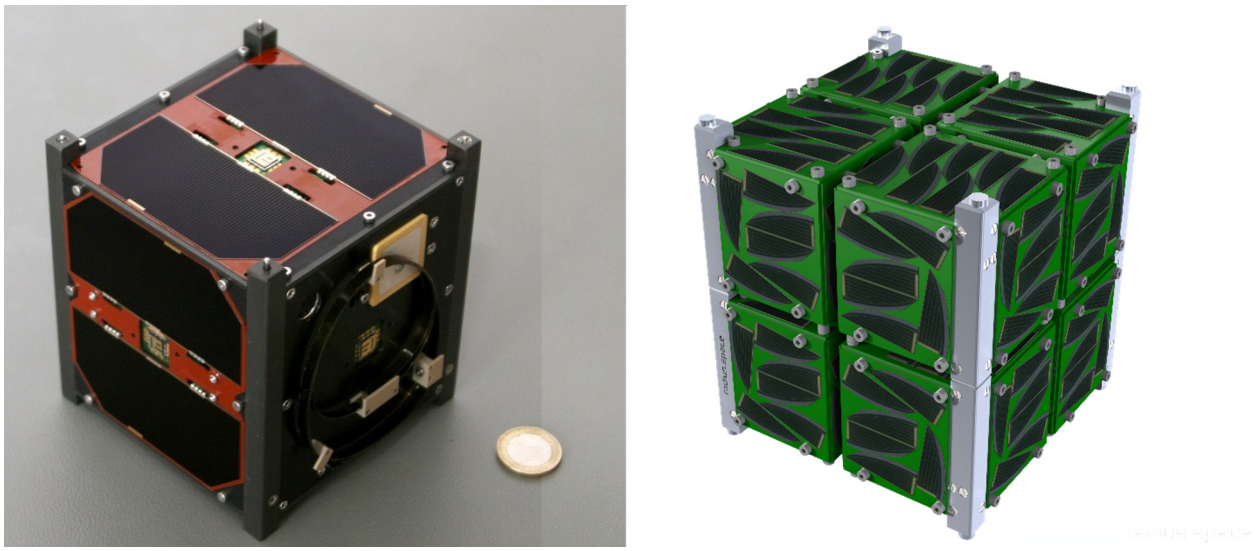

FIGURE 4: CUBESAT WITH £2 COIN (LEFT) [14] AND CUBESAT WITH 8 POCKETQUBES (RIGHT) [15]

2.2 Objectives

The foremost goal of this work is to design fault-tolerant, fault-detecting and self-healing algorithms for a MIC, which is controlling a nanosatellite. Furthermore, the developed software, which implements the algorithms, needs to work on different platforms and perform all the necessary on-board housekeeping, deployment of antennae and solar arrays etc. and, at the same time, be resistant to the most probable and most severe causes of failure in LEO. A typical life time for a nanosatellite in 300km altitude LEO is approximately one month [16], after which an orbital decay becomes too large and the Earth's gravitation forces the satellite to re-enter the atmosphere, where it burns up [17]. The orbital period of a spacecraft in LEO is about 90 minutes. For the purposes of this project, the estimated time for the MIC software to survive is 1 month. Another crucial assumption made in the scope of this project is that the MIC can turn on and off unpredictably due to the absence of a stable power supply, however, the up-time is assumed to be at least 1-2 seconds in order to let the MIC perform its core tasks. The software mostly consists of modules or methods to detect and handle specific errors, fix specific issues and use the MIC resources as required by a mission, thus, it is easy for a user to use and adjust the code as necessary.

To create such robust software, firstly, it is essential to define and quantify potential threats a satellite in LEO can be exposed to. Therefore, this paper reviews and discusses the potential problems, which are likely to occur to the MIC during the operation of COTS electronics in outer space and in LEO specifically, and solutions which are commonly used to deal with such problems. COTS electronics are not designed to function as parts of a satellite, hence, it is crucial to understand how suitable is the space environment for such equipment and what are the major challenges in managing consumer electronics in the space.

Next, when the risks and their consequences are known, the corresponding algorithmic solutions are developed as part of this project. One of the most crucial criteria is that the fault-tolerant and self-healing software has to be able to run reliably on the MIC and perform all the tasks as required. Thus, the potential problems, which can be foreseen, need to be dealt with appropriately and accounted for in code.

However, the robustness of the spacecraft software is very much dependant on the available memory allocated for the code. Embedded microcontrollers, which are used as nanosatellites' MICs, are generally very limited in terms of on-board storage and computing power. The characteristics are commonly measured in kilobytes and single- or double-digit MHz. Hence, during the software development the size of the code base is constantly monitored and it is understood that the code has to be as efficient as possible and as low-level as necessary, because otherwise all the important routines might not be able to fit into the satellite's embedded system. This means that all built-in or supported libraries, which could have been used in the code, are stripped down or rewritten to include only the necessary bits of code in order to save space. This requires a very good understanding of computer architecture, electrical engineering and chip architecture as it involves chip specific hardware register manipulations and knowledge of chip electronic design as well as various communication protocols such as I^2C.

2.3 Contributions

The main contribution of this project is a framework, that will serve as a basis for the development of satellites of the given size range. The framework provides working solutions for problems which can arise when a satellite resides in LEO and which can be handled programmatically. The satellite software is intended to be open-source and is available to anyone who might wish to use it as is, or develop it further and enhance it with additional functionality as they see fit.

The software developed for this project is part of a bigger project – PAMSAT (Parametric Advanced Membrane Satellite And Testbed), which will contribute to the world of space engineering through further exploration and exploitation of recently developed microsatellite standards via launching such a satellite into LEO and implementing the following functionalities:

- Deployment of TWIST (high performance thin film small satellite subsystems with a twist) solar panels [18] and evaluation of their performance (Figure 5).
- Installation of a ground station (GS) on the Imperial College London Electronic and Electrical Engineering (EEE) building, establishing a consistent radio connection between the GS and the spacecraft (Appendix 1).
- Demonstration of feasibility of carrying third-party payloads, relevant data collection and transmission of it to the GS.

FIGURE 5: TWIST THIN FILM SYSTEM DEPLOYMENT PROCESS DEMONSTRATION [19]

These research goals are not part of this project, however this work is closely tailored to fit the requirements of the PAMSAT project. PAMSAT, which the Imperial College London Adaptive Emerging Systems Engineering (AESE) research group is currently working on, is based on CubeSat and PocketQube standards with the final specifications currently being finalised.

Nevertheless, the framework developed in this project enables a spacecraft to perform on-board maintenance as required by the PAMSAT project. This implies that the software is fault tolerant

to the unpredictable space environment, can heal the stored code in cases of memory faults, can communicate with various on-board peripherals, performs all the tasks in the correct order regardless of power fluctuations, has predictable behaviour, can store data permanently, and is portable to different processor architectures with minimal modifications. In addition to this, the software fits into a very limited memory space as the computer for the role of PAMSAT MIC is provisioned to only have 32kb of Flash storage for the complete program code including the fault tolerance, fault recovery, code healing functionality, as described later in this paper, and any other third party code, which could be required by the mission.

This project will not only be useful as part of PAMSAT, but is also expected to significantly boost the amount of research in the area of nanosatellites and cost-efficient space exploration. Such an open-source framework will be helpful and save a significant amount of time for other space researchers, who desire to explore the cosmos with nanosatellites. The framework gives an insight into how to approach the most serious threats which a satellite is likely to be exposed to. Most importantly, the framework provides a complete and portable set of software solutions for regularly occurring faults, which can and are likely to happen to a satellite MIC. It can be transferred to any chip architecture and by design is supposed to keep MIC operational for significantly longer than one month.

3 Background

3.1 Hardware Specification

For the purposes of this project two different chip architectures were chosen for designing, developing and testing the software. Targeting software for at least two different chips is a good practice to avoid any potential risks of reduced portability in terms of very chip specific software solutions. It is also useful for redundancy purposes to have two MICs from different manufacturers in order to increase the chances of a space mission success. For example, in a case of an unlikely event, which has a fatal outcome one a specific type of microcontroller, there will still be one more operational MIC executing the code, which significantly enhances the probability of the spacecraft remaining operational.

The platforms that were considered as cores for the MIC, were the MSP430 and AVR ATmega328p (Table 1). The fault-tolerant software is designed for these two microcontrollers. It is very easy to notice that the memory resources on both chips are very scarce, which poses a serious limitation to the code flexibility.

MSP430	**AVR ATmega 328P**
Model: CC430F5137	Model: ATMEGA328P-PU
CPU: 16-bit, 8MHz	CPU: 8-bit, 16MHz
Flash: 32kb	Flash: 32kb
RAM: 4kb	RAM: 2kb
Voltage range: 1.8V to 3.6V	Voltage range: 1.8V to 5.5V
SPI bus: Yes	SPI bus: Yes
I2C bus: Yes	I2C bus: Yes
Example of real-life implementation: (provisional MIC based on MSP430 with solar panels)	Example of real-life implementation: (Arduino Uno board)
	

TABLE 1: HARDWARE SPECIFICATION [20]

By design, PAMSAT will be communicating with different on-board sensors and other electronic equipment using the I^2C data transfer protocol. I^2C (Inter Integrated Circuit Communications) protocol is used to exchange data between a designated microcontroller (Master) and peripheral devices (Slaves), including, but not limited to, other microcomputers, sensors, integrated circuits,

and is supported by most modern microcontroller architectures as it is commonly used in a broad variety of electronic devices (Figure 6) [21].

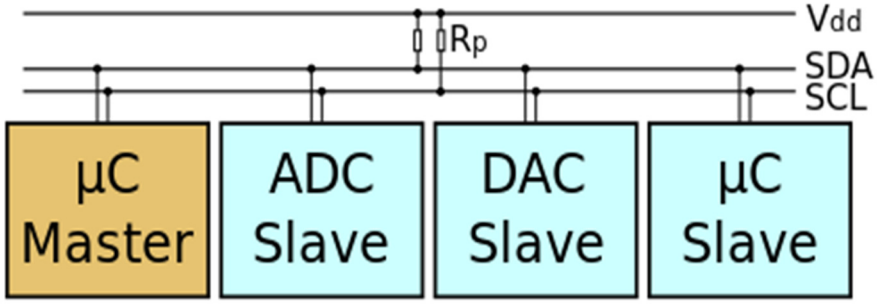

FIGURE 6: I2C MODEL [22]

The Integrated Development Environments (IDEs) used for this project to develop software for the microcontrollers were the Arduino IDE v1.6.9, Atmel Studio 7, Energia v0101E0017, and Code Composer Studio 6.1.3.

3.2 General Software Requirements

The core purpose of the software is to provide on-board maintenance of all satellite operations and functions in addition to having high levels of fault tolerance. Table 2 lists all the crucial tasks the software is required to be able to do.

Function	Description
Hardware pin manipulation	One of the most significant tasks of the PAMSAT project is to deploy antennae and solar panels, which are attached to the body of the spacecraft, folded and held under mild pressure against deployment with multiple pieces of fishing line, in a predetermined order. To achieve this, the MIC needs to send current for a continuous period of time through specific pins which are connected to burn wires to melt parts of the fishing line. Once they are melted, it will then let the radio antennae and solar panels expand. Additionally, running current through certain pins can enable the spacecraft to orientate itself by interacting with the Earth's magnetic field using magnetorquers.
I^2C data transfer	The satellite can have various sensors (gyroscope, accelerometer etc.), which gather different types of information. Therefore, the MIC should be able to exchange data with such sensors reliably and efficiently. The communication between the MIC and the sensors is performed using the I^2C data transfer protocol which only requires a two wire interface.
On-board data storage	The software has to save data reliably in the Flash memory of an on-board MIC's chip. Moreover, the software has to be able to retrieve the saved data from the memory and verify that the stored data is correct. If a piece of data becomes corrupt, the software should attempt to fix it and save it again. This is required to be able to track the performance at each stage, receive data from various peripherals, and keep variable values secure for the times when there is no power.

Function	Description
Ground station communication	The satellite is carrying a UHF radio and the software should provide the functionality for transmitting and receiving signals from a ground station using the on-board radio module. The satellite should be able to communicate to the Earth various data from its peripherals or any other information as required. Additionally, it should be able to process certain signals received from the ground station.
On-board timing	The spacecraft system is a soft real-time system. Due to a very unpredictable power supply it is impossible to ensure that every deadline is met while the spacecraft is in space. However, the ability to measure time intervals reliably is of high importance for mission success. Firstly, after being deployed into space, the spacecraft has to sleep for at least 30 minutes to ensure that it has drifted far enough to not jeopardise the carrier's mission. Secondly, various sleeping times are required between certain methods for additional stability because some on-board sensors have well defined calibration times. MICs may have different hardware restrictions and require time to react appropriately to certain changes of their registers' values.
Freeze detection	The MIC has to be able to detect entering an infinite loop or otherwise becoming frozen and not proceeding with code execution and perform a restart if such a problem occurs.

TABLE 2: GENERAL SOFTWARE REQUIREMENTS

All these requirements are included because they are very common for a space mission and, hence, have to be addressed with care and knowledge that the memory space and computing power of a nanosatellite are scarce resources as discussed in the previous section. In addition to these tasks, the software also has to manage various faults, which are discussed later in this paper. Therefore, since the provisioned MIC is limited to 32Kb Flash memory, the functions presented in Table 2 have to be as robust and light-weight as possible to minimise the risks of being buggy and maximise the amount of space for fault tolerance procedures and any other code as might be required by a space mission.

3.3 Space Environment Conditions and Impact

"The space environments differ from terrestrial environments in many ways" [23]. They are very tough and often have unpredictable influence on COTS. It is extremely difficult to model all possible scenarios of such environment on the Earth and satellites have to undergo thorough testing, which approximates space and reduces the risks of unexpected threats to a space mission. The risks are different depending on the orbit a satellite is launched into and since this project involves a satellite which is designed to be launched into LEO, the description presented here is for this particular orbit.

LEO has high radiation rates due to "high-energy solar particles and photons and charged particles trapped in Earths' magnetic field" [24]. The radiation there consists of mostly protons and electrons, while photons (X-rays, *gamma* rays) are more likely to be observed on higher orbits. The standard dose of proton and electron radiation a satellite consumes in LEO, which is below 500km, is same in both northern and southern hemispheres and is equivalent to 100-1,000 rad/year due to

particles trapped in *Van Allen* belts (Figure 7) [25]. In comparison to other higher inclinations, such a dose is approximately 10 times larger than what a satellite in LEO usually gets exposed to.

Radiation Source	Effects of Solar Cycle	Variations	Types of Orbits Affected
Trapped Protons	Solar Min - Higher; Solar Max - Lower	Geomagnetic Field, Solar Flares, Geomagnetic Storms	LEO, HEO, Transfer Orbits
Trapped Electrons	Solar Min - Lower; Solar Max - Higher	Geomagnetic Field, Solar Flares, Geomagnetic Storms	LEO, GEO, HEO, Transfer Orbits
Galactic Cosmic Ray Ions	Solar Min - Higher; Solar Max - Lower	Ionization Level, Orbit Attenuation	LEO, GEO, HEO, Interplanetary
Solar Flare Protons	During Solar Max Only	Distance from Sun; Outside 1 AU, Orbit Attenuation; Location of Flare on Sun	LEO (I>45°), GEO, HEO, Interplanetary
Solar Flare Heavy Ions	During Solar Max Only	Distance from Sun; Outside 1 AU, Orbit Attenuation; Location of Flare on Sun	LEO, GEO, HEO, Interplanetary

TABLE 3: COMMERCIAL MICROELECTRONICS TECHNOLOGIES FOR APPLICATIONS IN THE SATELLITE RADIATION ENVIRONMENT [26]

By default, commercial electronics have a radiation tolerance levels of 2,000 to 10,000 rad [27]. Therefore, assuming that a satellite weighs 1kg and stays on the orbit for 1 month, it can be irradiated with up to 83.3 rad in the worst case [28], which is significantly less that the minimum default radiation tolerance, of the on-board electronics. Hence, the risk of the electronics to degrade beyond operational levels and a satellite malfunctioning due to the absorbed radiation being larger than the Total Ionising Dose[1] (TID) that the satellite is able to withstand is insignificant. The Single Event Upset[2] (SEU) Error Rate, on the other hand for a satellite on LEO is typically equal to

$$10^{-5}\,\frac{errors}{bit\ per\ day}\ [27]$$

The amount of solar energy, which reaches a satellite in space, is noticeably larger than the amount of energy, which actually reaches the Earth's surface. The average amount of solar power, which reaches the Earth's surface is called the solar constant and is approximately 1,367 watts per square metre (W/m^2) [29]. The value varies slightly depending on the time of a year, since the Earth's orbit is lightly elliptical, but the variation is negligible for the purposes of this project. Of the solar energy, which reaches the surface of our planet, approximately 50% is being either absorbed by the atmosphere or reflected back into space [30]. However, taking into account the times of year and latitudes, there are areas on the Earth, which are receiving almost as much solar energy as a satellite would in space [32]. Therefore, in terms of amount of energy, the LEO environment is fairly similar to certain areas of the Earth's surface.

[1] Total Ionising Dose - caused by excessive radiation from the Sun. TID defines the quantity of free particles absorbed by and degrading a material.

[2] Single Event Upset – soft, non-destructive errors caused by ionizing radiation strikes that discharge the charge in storage elements, such as configuration memory cells, user memory, and registers [31]. Most often found as bit flips anywhere in the memory space.

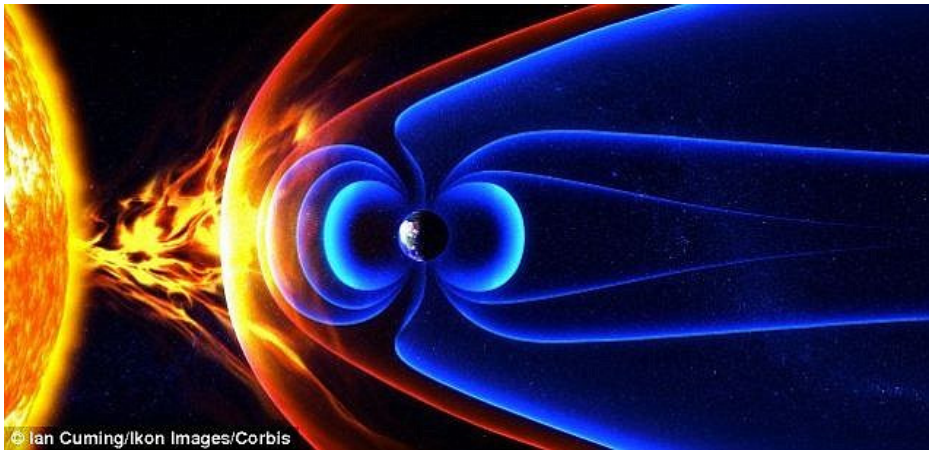

FIGURE 7: VAN ALLEN BELTS PROTECT EARTH FROM RADIATION [33]

The other issue in LEO is the temperature. Since space is a vacuum and there are not enough particles to absorb the heat from a heated object as well as there being no particles to share heat with the object, when it is in Earth's shadow, the temperature of a regular metal plate are said to cycle from –170°C to 123°C [27] depending on the object's orbit.

Therefore, in addition to evaluating a satellite's reliability in zero gravity conditions, it is required for this project to consider scenarios for potential problems, which may be caused by abnormal radiation, in terms of SEUs, and extreme temperature conditions, to develop an efficient fault-tolerant system.

3.4 Types of Risks in Space Environment

The spacecraft, which AESE group is working on, consists of COTS components. Hence, the on-board electronics are not designed for an environment like outer space and a great care has to be taken when evaluating potential threats to a nanosatellite. The list of theoretical causes of space mission failures is presented in Table 4.

Cause	Description
Total Ionizing Dose (TID)	If TID of an electronic component exceeds the component's tolerance level, it becomes faulty and recovery is not possible. However, as discussed above and assuming that the mass of the PAMSAT spacecraft is approximately 1kg, the satellite's components are not going to be in space for long enough for the TID to become significantly large. Even though the likelihood of degradation is small, TID is addressed where possible in the form of a faulty Flash memory gate.
Single Event Effect (SEE) & Single Event Upset (SEU)	SEEs have very low probability of happening [27], however, if they do occur the spacecraft can and will behave unpredictably. SEEs can occur as bit flips anywhere in the memory space of an electronic component. Such components can recover from SEEs programmatically if the error is detected correctly.

Cause	Description
Angular rotation	When the satellite leaves its deployer, it has no forces, which could limit its rotation and it has as much kinetic energy as the deployer provided it with. In an unlikely event of the deployer spinning the spacecraft too much, it can become impossible for the spacecraft to generate enough energy from the small on-board solar panels to melt the fishing line, which restrains the antennae and the bigger TWIST panels, if the angular rotation speed is too high and solar panels face the Sun for not long enough.
Temperature	Temperature of a metal plate in LEO can cycle from -170°C to +123°C [27]. Therefore, the software has to be protected from freezing and overheating of the on-board electronics. PAMSAT is unlikely to undergo extreme temperature shifts as the orbital period on LEO is 90 minutes, a satellite is rotating around its own axis and regularly entering the shadow of the Earth, hence, the heat has time to distribute and dissipate. Therefore, the satellite temperature fluctuates from -30 to 50 °C [33] and the temperature of its internal components normally varies from -5 to +30 °C [34], which is in normal operating temperature ranges for consumer electronics.

TABLE 4: POTENTIAL SOURCES OF FAULTS

We can conclude that in order to develop robust fault tolerant on-board maintenance software we have to consider the possible effects of TID, SEE and SEU. Moreover, we have to take into account the angular rotation and ensure that before the TWIST solar panels are deployed the fishing line melting task has a very high priority and can be executed in the time slot provided. The temperature fluctuations are not as dramatic as initially expected and therefore the software protection of the components from the temperature changes is out of the scope of this project. Next, an analysis of the causes of risks, which are likely to be caused by the problems discussed above, is required to determine the types of faults.

3.5 Types of Faults in the MIC Performance

To develop stable software, it is necessary to understand what type of influence the space can and is likely to have on the satellite's MIC and what are the potential consequences of the risks discussed in the previous section. It can sometimes be more difficult to detect an erroneous event during the program execution, than actually develop a solution to fix the error. However, it is possible to reduce this chance noticeably by accounting for the most likely, most detectable, most probable and most fixable errors [27]. The faults, which are dealt with in this project, are listed in the Table 5.

Everything can and may go wrong on a spacecraft. However, it is important to prioritise the detectable errors and develop a proxy solution for the undetectable and difficult-to-detect errors due to significant limitations on memory space and processing power. The priority for all the errors discussed above is high as the components are lacking any radiation tolerant protection and there is no on-board battery planned [6].

Problem	Likelihood	Consequences
I^2C bus jamming	Likelihood depends directly on amount of devices on the I^2C bus. If one of the devices becomes corrupt and occupies the I^2C bus, the whole bus becomes useless	Unable to communicate with on-board peripherals and exchange data; unable to perform any planned analyses of the environment and any peripheral adjustments when necessary
Bit flip on RAM	Likely due to SEE/SEU [35]	Corruption of a variable on stack and erroneous program input/output, which leads to unpredictable results.
Bit flip in saved data	Likely due to SEE/SEU [35]	A bit flip in I^2C slave address, other important constant values or various data, which was generated by the MIC itself leads to faulty code execution
Bit flip in program code	Likely due to SEE/SEU [35]	If the program code becomes corrupt, the processor can start executing commands, which can potentially corrupt whole memory space
Bit flip of program tracker[3]	Likely due to SEE/SEU [35]	The processor will start executing the program from a wrong line, which means that some code may remain not executed or executed twice. Depending on the shift in program tracker it may have no effect or corrupt some variable values
Corrupted ALU	Likely due to TID and SEE/SEU [35]	Corrupted ALU can lead to erroneous arithmetic computations and incorrect output
Corrupted watchdog	Likely due to TID and SEE/SEU [35]	Resets of the MIC at stochastic time intervals or the absence of such resets at all when they are required by the watchdog timer. Unpredictable behaviour
Corrupted registers	Likely due to TID and SEE/SEU [35]	Registers are unable to store temporary variable values reliably. Any computations, which use faulty registers, are going to be incorrect and faulty
Permanently damaged FLASH/RAM memory	Due to excessive TID and/or frequent usage of a specific memory location, it can either wear down (Flash) or degrade beyond its radiation resistance threshold	Constant errors in writing to or reading from a memory location. Unpredictable behaviour of a function call, which uses broken memory location.

[3] Program tracker is a variable, which holds checkpoint numbers for the program to jump to when it loads. Program tracker manages the code execution and ensures that the program runs in the cronological order

Problem	Likelihood	Consequences
Unstable power supply	The satellite certainly has some angular rotation, which is difficult to estimate. Such angular rotation has a direct influence on the satellite, since if solar cells are not facing the Sun, the MIC is not supplied with power. As it was mentioned before, the satellite orbits the Earth every 90 minutes, it will be in the Earth's shadow very often and since it does not have any battery on board, it is impossible to guarantee continuous power supply.	Interruptions at random times during the execution and restarts. If the restarts are frequent or the program execution is slow, it might be the case that the program will never reach certain areas of code. Therefore, it has to be able to continue from the point in code, where a stochastic restart happened.
Other problems	Likely due to TID and SEE/SEU [35]	Potential failure of the MIC and the space mission

TABLE 5: TYPES OF FAULTS IN SPACE

3.6 Fault Tolerance Methods

"Reliability is a critical design attribute for systems operating in remote or inhospitable environments" [36] and there has been done a lot of progress in improving a spacecraft's reliability. The progress involves both hardware and software improvement methods.

> "Satellites and other space systems typically employ rare and expensive radiation tolerant, radiation hardened or at least military qualified parts for computational and other subsystems to ensure reliability in the harsh environment of space." [37].

Therefore, radiation in space is less of a threat to more expensive space missions. However, for the purpose of this project the chosen electronic components are COTS and these do not have any radiation hardening. Thus, in addition to the existing default radiation resistance, the radiation effects are handled on the software level.

The supply of power is a major concern for a satellite in space and, thus, has to be treated with care. Spacecraft generally have four types of power sources as shown in Table 6.

Type	Power source
Battery	Store energy made on Earth and release it as electricity.
Solar panels	Convert light from the Sun into electricity.
Radioisotope thermoelectric generators (RTG)	Radioactive materials, encased in a sealed shell, will generate heat as they decay into non-radioactive materials. Heat is converted to electricity
Fuel cells	Store power in the form of separated oxygen and hydrogen. A thin membrane between the two elements harnesses the energy separated when the oxygen and hydrogen combine to form water.

TABLE 6: SPACECRAFT POWER SOURCES [38]

> "In space, batteries must work in both very hot and very cold conditions. They must withstand a lot of radiation from the Sun. They must work in a vacuum without leaking or blowing up. They must be rugged enough to withstand the severe vibrations of a rocket launch" [39].

Hence, due to high risks of malfunctioning or large physical dimensions of a power source, PAMSAT will only be carrying solar panels, which means that it will only work when not in eclipse. Therefore, a robust method of tracking the program's progress is required to enable the spacecraft to continue code execution from where it was interrupted by the absence of solar power.

To further enhance the stability of a spacecraft, the engineers often add redundancy to it as radiation hardening does not provide 100% protection against possible failures. Additional redundancy is very commonly used in fault tolerant control systems [40] to ensure stability of such systems. A system is said to be fault-tolerant if "its programs can be properly executed despite the occurrence of logic faults" [40] and this is essential for a spacecraft to have this property, as there is no human access to perform any fixes in a case of a fault. The types of fault tolerance approaches used for space applications are presented in Table 7.

Operational Environment	**Space**
Fault-Tolerant Approach	
Fault avoidance and fault intolerance	Radiation-hardened components
	Design diversity
	Safe system
Fault tolerance	Component-level redundancy
	Subsystem-level redundancy
	Multi-computer
	Retry
	Software reload

TABLE 7: SPACE DOMAINS [41]

Hardware redundancy uses radiation-hardened components, which duplicate the main system, to mask away the effects of malfunctioning electronics (multi-computer) (Appendix 2) [41]. The hardware redundancy level of PAMSAT project is still being specified at this stage and is out of the scope of this project. Software redundancy performs similarly by having duplicates of the program code in storage.

One of the common software fault tolerance methods is having three teams develop their own version of a program and have all three programs run simultaneously, which is similar to N-modular hardware redundancy, which is an extended version of Triple Modular Redundancy (TMR) [42]. Due to lack of resources, in the scope of this project the software is developed by one person and the software N-version fault tolerance technique is used [43]. This technique implies that the same code is stored in N different places and is always reconciled with itself during the running time to verify that the code is correct (software reload). Additionally, a great care is taken in the software design (safe system) and protection from freezes and fails is partially implemented through repetition (retry). The product of this project is stored in three different memory spaces in the MIC and is modified as necessary to run identically on two different chip architectures.

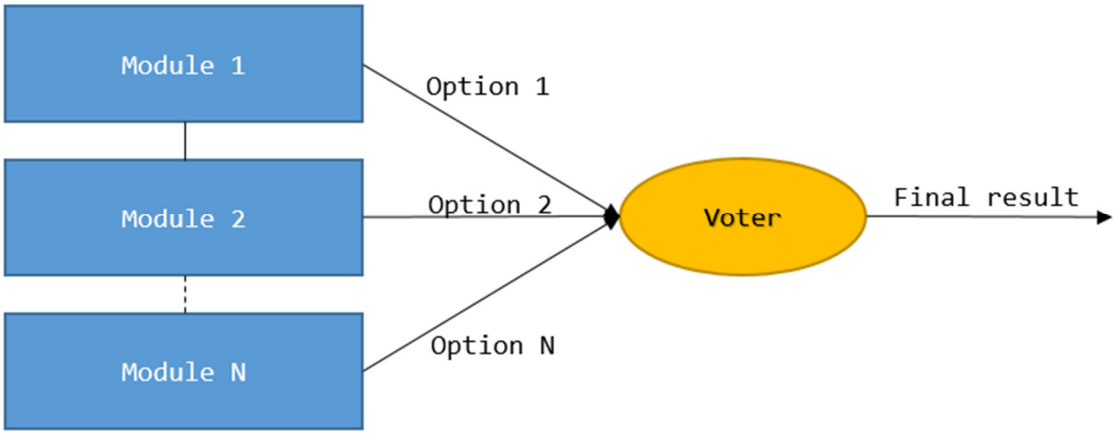

FIGURE 8: N-MODULAR REDUNDANCY CONCEPT

The multi-version methodology ensures that the code runs, but it does not protect the software from crashes, bugs or other design faults, which is part of a single-version redundancy technique [43]. "Many software faults are of latent type that shows up later" [40]. Therefore, the single-version fault tolerance method is used to develop the software for the spacecraft that has to be able to detect the most likely and predictable errors and adjust accordingly. Such errors are most likely to be due to TID and/or SEE, which are typically detected with checksums and can also be detected and corrected with Hamming code or Triple Modular Redundancy (TMR), which uses majority-voting system to fix errors [44]. The major differences of the three methods are summarised in Table 8.

	Checksums	Hamming Code	TMR
Error correction	Can detect up to n upsets per n-bit word if each upset is located in a distinct bit and only happened once in a checked memory segment (Table 9).	Corrects one single upset per word but does not correct the upset in the stored word. Upsets will accumulate if there is no extra logic to correct them.	Corrects up to n upsets per n-bit word if each upset is located in a distinct bit. It computes the correct value but it does not correct it. Upsets will accumulate if there is no extra logic to correct them.

TABLE 8: ERROR DETECTION AND CORRECTION METHODS [44] & AUTHOR

In addition to this, checking checksums is very fast and also TMR has a smaller time delay and does not need designated coders unlike Hamming Codes. The combination of these differences is the reason why checksums and TMR are preferred for this project as the code has to be able to detect and correct, if found, multiple errors quickly.

Later this paper presents a thorough description of how the error detection works and what is done if an error of a type as described above occurs. Some errors are difficult to detect and they cause program crashes. For such cases a general practice is to retry the operation by performing a system reset using a watchdog timer, which can identify a program crash [45].

	Before bit flip							
	bit 1	bit 2	bit 3	bit 4	bit 5	bit 6	bit 7	bit 8
byte A	1	1	0	0	1	0	1	0
byte B	0	1	0	1	0	1	0	1
byte C	1	1	1	1	1	1	1	1
byte D	0	0	1	0	1	1	0	1
checksum (XOR of all values)	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**1**

	After bit flip							
	bit 1	bit 2	bit 3	bit 4	bit 5	bit 6	bit 7	bit 8
byte A	1	1	0	0	1	**1**	1	0
byte B	**1**	1	0	1	0	1	0	1
byte C	1	**0**	1	1	1	1	1	1
byte D	0	0	**0**	0	1	1	0	1
checksum (XOR of all values)	**1**	**0**	**1**	**0**	**1**	**0**	**0**	**1**
correct checksum	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**1**
difference	**X**	**X**	**X**	**-**	**-**	**X**	**-**	**-**

TABLE 9: CHECKSUM ERROR DETECTION EXAMPLE

4 Design

For increased usability the framework developed in this project is very modular, which makes it easy to include, exclude or add various error-detecting and data transmitting modules. Therefore, separate methods, which either detect errors, fix them or both are discussed in this section.

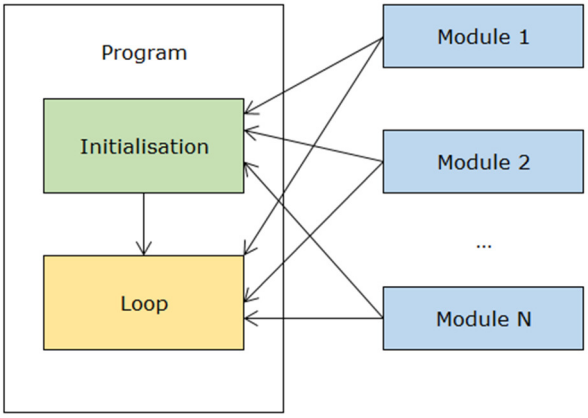

FIGURE 9: SIMPLIFIED FRAMEWORK DESIGN

4.1 Fault detection

To create a fault tolerant control system, we first need to know how to detect potential faults. However, not all faults are possible or convenient to detect in a system which is very limited in memory space and computing power. Therefore, some proxy solutions have to be developed. Summary detection mechanisms for faults which are likely to occur in space are presented in Table 10.

Problem	Fault detection method
I^2C bus jamming	In case of faults the program will be frozen on the I^2C method call. Watchdog will perform a reset. Every reset due to I^2C freeze is tracked
Bit flip on RAM	Three copies of the same variable are compared. Fault is detected if one copy is different
Bit flip in saved data	Three copies of the same variable/constant are compared. Fault is detected if one copy is different
Bit flip in program code	1) Fast method: check checksums of a sector of the program code and compare it to the stored checksums. If two values are different, then fault is detected 2) Slower method: perform TMR correction method on each byte in the code memory. TMR does the fault detection
Bit flip of program tracker	Three copies of the program trackers are compared. Fault is detected if one copy is different
Corrupted ALU	Basic arithmetic tests are performed. Fault is detected if results are not as expected
Corrupted watchdog	No detection possible. Fault prevention refreshes the watchdog register regularly
Corrupted registers	Specific values are put into and read from CPU registers. Fault is detected if values in the registers are not as expected
Unstable power supply	No fault detection possible. A program tracker tracks every step of the program progress
Other problems	Watchdog timer runs to detect any other errors, which might cause program freeze

TABLE 10: FAULT DETECTION MECHANISMS

These fault detection mechanisms will later help us successfully detect the faults, correct them accordingly, and perform code self-healing where appropriate.

4.2 Revised Software Requirement

In this section a more detailed software description is presented together with the fault healing procedures, which are triggered by one of the faults above. The description takes into account the potential dangers of operating in space discussed previously and provides a thorough outline of the logic flow in the on-board maintenance program as well as additional required test methods, which the program is incorporating. The logical flow of the program is displayed in Table 11.

Setup procedure as soon as MIC gets power	
Function	**Action if fault is detected**
Send radio signal to the ground station	Restart and retry
Only once: Partition the whole code memory space and calculate and store checksums for each segment	Set-up procedure; to be performed on the ground; no faults expected
Check if the deployment machine instructions are correct	Perform error correction (self-healing) with TMR [46]
Only once: Run current through specific pins to melt restraining fishing lines and deploy antennae and solar panels	Restart and retry
Check if the program machine instructions are correct	Perform error correction (self-healing) with TMR [46]
Check if the recovery function machine instructions are correct	Perform error correction (self-healing) with TMR [46]
Set-up watchdog timer as required	Restart and retry
Load saved variables into RAM and check if loading is successful	Store 3 copies of the variable and perform error correction with TMR [46] when variable is used
Check if the saved variables are correct	Perform error correction with TMR [46]
Check if constant values are correct	Performed together with the code error correction
Check if ALU works predictably	Restart and retry
Check if registers can store data	Wipe the registers' values and restart
Perform I^2C data bus transfer testing	Restart MIC and all the payloads. If multiple restarts do not help, disable payloads one by one or all at once depending on the hardware implementation
Resume the operations from where the program was last interrupted	

TABLE 11: SETUP PROCEDURE

Before the solar panels are fully deployed, it might be reasonable to skip the checksum step in order to save time, however the amount of time saved is insignificant. In addition to Table 12, there should be a looping cycle, which actually performs the functions the spacecraft is designed for. A

generic example of a looping cycle is presented in Table 12 and it can vary depending on a particular space mission.

Looping procedure once setup is complete	Description
Request data from on-board sensors	
Receive data from on-board sensors	
Save the data in RAM and/or Flash as applicable	Perform safe storing with appropriate tests. If memory locations are permanently damaged, shift the locations
Process the data	
Send the data via radio to the Earth	If fault detected, then restart and retry
Listen to commands from the Earth for a specific amount of time	Commands are coming in format "memory address" & "byte to be saved"

TABLE 12: LOOPING PROCEDURE

In the case of continuous power after the solar panels are deployed, the setup procedure will run only once every 90 minutes when the satellite emerges from the Earth's shadow. This is not frequent enough and to introduce more robustness some of the tests have to be run regularly. Thus, the safety checks as shown in Table 13 have to be performed from time to time in the looping procedure as well to ensure that MIC is working correctly. The frequency of checks depends on the space mission and the additional tasks the spacecraft is supposed to be performing, hence, the figures presented in Table 13 are not strict, are for general guidance only and should be used with care. Since memory occupies the biggest physical area of the MIC, it has to be checked more frequently than other parts of the microcontroller purely because if a fault occurs, it has higher probability of occurring in the code memory.

Methods to be called repetitively as safety checks	Frequency
Check if ALU works predictably	every 20th iteration
Check if registers can store data	every 20th iteration
Perform I^2C data bus transfer testing	every 10th iteration
Check RAM and Flash saving ability	every 10th iteration
Check if variables stored in RAM are correct	every use of a variable
Check if variables saved in Flash are correct	every 5th iteration
Check if the program machine instructions are correct	every 5th iteration
Check if program tracker is tracking the progress correctly	multiple times during 1 iteration

TABLE 13: REGULAR SAFETY CHECKS

Since the code is very limited in terms of space and has to be as fast as possible with the given processing speed, it puts significant constraints on the fault detection and fault tolerance mechanisms such that the latter ones have to be kept as simple and as quick as possible. This is why some of the faults are tolerated through restarting the whole system. The system evaluated in this project has only 32kB of memory and has to store three versions of code with additional functions for code self-healing. Hence, the functionality discussed above was carefully tailored to maximise

the number of tasks to be performed while minimizing the execution time and energy consumption. Therefore, the aforementioned software design is considered sufficient and is used to implement the fault tolerant control system (FTCS) on a microcontroller which is intended to act in a role of MIC on the PAMSAT spacecraft.

5 Implementation

This section provides comprehensive explanations of the method calls, which are described in the previous section, that are used to detect the errors and fix them. Some of explanations are presented in a form of pseudocode in order to make the understanding of the code easier and make the methods more transferable. The functionality presented here was implemented in C programming language with certain parts in assembly and tested on MSP430 microcontroller model CC430F5137.

The MSP430, like the majority of microcontrollers, has embedded Flash memory to store code. The Flash storage has a special characteristic, which has to be taken very seriously and which produces certain obstacles to the development of the nanosatellite reliable control system. Flash memory can only be erased and written in segments. For MSP430 code memory a segment is 512 bytes, but it depends on the chip architecture. Therefore, to erase or rewrite a byte of data somewhere in the memory it is necessary to temporarily store the whole segment of data somewhere else as shown later in this chapter. Otherwise, the data from that segment will be lost.

The requirement for the installation to make the program run as expected is that three versions of the program have to be stored in three separate locations in the chip's flash memory to enable it to reconcile and error-correct the machine regularly to make it fault tolerant against SEEs and SEUs. Additionally, three copies of the code recovery code have to be stored in the fourth separate location to ensure that the code does not erase itself.

5.1 Safe storage

One of the biggest challenges in making the whole system work is the ability to save and retrieve information reliably. If this functionality is working correctly, then it becomes easier to implement other methods and solutions.

In addition to the main 32kB of the code memory, the MSP430 has a section "Information Memory" (Appendix 3 & Appendix 4), which is an additional 512 bytes of Flash storage separated into 128 byte segments, which is a perfect location for storing the program tracker and other constants and variables. Most embedded microcontrollers have a similar type of memory, whether it will be a regular Flash memory or Electrically Erasable Programmable Read-Only Memory (EEPROM), which is easier to manage. Flash memory has a wear-out disadvantage [47], which means that logic gates that store information have a limited amount of program/erase cycles. To overcome this, every variable has its own range of memory addresses, which are used interchangeably to reduce the effect of wearing out. The program itself is designed in such a way that all the variables, which are needed, are known at compile time. Therefore, we are able to allocate a range of addresses for each variable of a size that corresponds to the frequency of usage of a given variable.

The saving function has a linear probing (rolling) method (Figure 10) where a variable is saved to the next memory address in relation to where it was saved before. Similarly, the loading method probes memory addresses until it finds the value. Then three copies of the value are loaded into RAM, compared, fixed and re-saved if necessary.

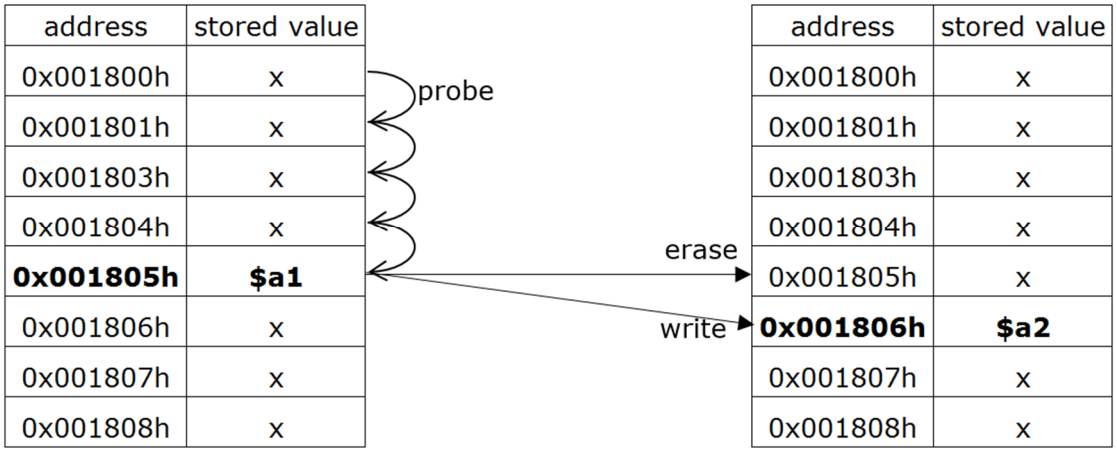

FIGURE 10: PROBING ERASE AND WRITE

Before a new value is being stored in the permanent storage the previous value is first erased to avoid any potential conflicts during the retrieving of the value. Then the new value is stored in the succeeding memory slot and the loading is attempted to ensure that the value has been stored correctly. If the loading fails multiple times, then the memory slot is shifted further and saving is executed again (Figure 9). This continues until a variable is stored in a healthy memory location (Algorithm 1).

The program tracker is the most frequently changing value and its correctness is vital for the space mission. It manages the program chronology and stores the number of a checkpoint, which is supposed to be executed next. For this project the tracker occupies ¼ of the variable memory, which is 128 bytes or one single information section completely. Thus, the whole segment is temporarily stored in RAM during the process of erasing and writing.

Algorithm 1: Loading a variable. Source: Author

```
1: //$size is 32 for the p. tracker to keep the values separated from each other
2: //$address is a predefined location for storing the value
3: unsigned char load ($address, $size) {
4:     unsigned char i = 0
5:     while i < $size {
6:           let the values at ($address + i), ($address + i + $size),
                   ($address + i + $size * 2) be $a1, $a2 and $a3
7:           if $a1, $a2 or $a3 is not equal to 255(default Flash value) {
8:                 if $a1, $a2 and $a3 are same
9:                       return $a1 //found the value, and it is stored correctly
10:                $answer = error_correct($a1, $a2, $a3)
11:                if $answer is not equal to 255
12:                      break //found the value, and it is not stored correctly
13:          }
14:          increment i
15:    }
16:    if i = $size //reached end of the allocated space
17:          return 255 //no stored values
18:    if $a1, $a2 or $a3 is not equal to $answer
19:          re-save the variable, which is different
22:    return $answer
21: }
```

error_correct($a1, $a2, $a3) performs TMR error correction between three values which should be equal and can fix up to one error per one bit, or 8 unique errors per byte (Algorithm 2). An example of how this function works is easily shown with a truth table (Table 14).

Algorithm 2: Error correction (Majority vote). Source: Author

```
1: Unsigned char error_correct($a1, $a2, $a3){ //3 values supposed to be the
                                                                    same
2:     return ($a1 & $a2 | $a1 & $a3 | $a2 & $a3)
3: }
```

	Byte							
	bit 1	bit 2	bit 3	bit 4	bit 5	bit 6	bit 7	bit 8
Correct value	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**0**
$a1 (assumed)	1	1	0	0	1	1	1	0
$a2 (assumed)	1	1	0	1	0	0	1	0
$a3 (assumed)	1	0	1	0	1	0	1	0
$a1 & $a2	1	1	0	0	0	0	1	0
$a1 & $a3	1	0	0	0	1	0	1	0
$a2 & $a3	1	0	0	0	0	0	1	0
($a1 & $a2 | $a1 & $a3 | $a2 & $a3)	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**0**

TABLE 14: ERROR CORRECTION TRUTH TABLE EXAMPLE (MAJORITY VOTING)

Such a loading mechanism is able to detect and correct up to 8 bit flips per byte given that the flips are in unique positions in the byte. The main logic is that if a bit is at 1 more than once, then the correct bit is 1, otherwise it is 0.

The saving procedure is implemented similarly to the loading – the code firstly finds the variable, which is already saved, then this variable is erased and the new one is saved in the subsequent memory location.

Algorithm 3: Saving a variable. Source: Author

```
1: save ($variable, $address, $size){ //$size is size of the sector, where $variable
                                                    can be saved
2:     unsigned char i = 0
3:     while i < $size {
4:           let the values at ($address + i), ($address + i + $size),
                 ($address + i + $size * 2) be $a1, $a2 and $a3
5:           if $a1, $a2 or $a3 is not equal to 255(default Flash value) {
6:                 if $a1, $a2 and $a3 are same
7:                       break //found the value, and it is stored correctly
8:                 $answer = error_correct($a1, $a2, $a3)
9:                 if $answer is not equal to 255
10:                      break //found the value, and it is not stored correctly
11:          }
12:          increment i
13:    }
14:    if i is less than $size //the latest value exists
15:          erase $a1, $a2 and $a3
16:    increment i
17:    if i is bigger than $size //i is beyond the allocated size boundaries
18:          i = 0
19:    store $value in the three subsequent memory locations determined by i
20: }
```

These two functions allow reliable saving and loading of any variables, however, the address space has to be predefined and is platform specific. The addresses of storage for MSP430, for example, vary between 0019FFh and 001800h. The functions are suitable for any type of TMR recovery including but not limited to code recovery, progress tracking, peripheral sensor data storage etc.

5.2 Code Self-Healing

The code self-healing method, which is implemented in this work, can be split into multiple distinct stages.

1. Parity byte checksums for each segment of code memory are stored in the MSP430 information memory. Parity byte checksums are calculated as XOR of all individual bytes in that memory segment. Hence, if a bit flip happens, then the newly calculated checksum will be different to the stored value, thus, a fault can be detected. The code memory is split into segments of 1024 bytes making it 32 parity bytes for 32kB of code. Three copies of the checksums are stored making it 96 bytes in total. This routine is only run once on the first initialisation of the program and should be performed on the ground to ensure that everything is set up correctly.
2. The microprocessor internal memory is split into four segments, call them A, B, C and D. Each of them is 8kB big. Segment A stores the main program code and segments B and C are storing the exact copy of the code in A. Section D is further split into $D1, D2, D3$ and $D4$. $D1, D2$ and $D3$ store the same code, which only runs through sectors A, B and C and fixes any code errors. A routine to correct the code in sections $D1, D2$ and $D3$ is stored in A, B and C. This design enables to overcome a potential risk of code erasing itself during execution. Such a risk exists because Flash memory is written in segments and if during the recovery procedure the segment, which is actually running, is erased, the PC register of the microcontroller will read a blank instruction and get frozen. After the watchdog performs a reset, RAM is refreshed and, hence, the code is lost permanently. The implementation requires two copies of the similar code to be stored in two distant memory locations and each such code does not make any changes to itself and does the healing of the other part of the code.
3. The error correction of code sections $D1, D2$ and $D3$ is run from section A. The method compares the versions of the code stored and if any discrepancies are found, then the TMR method fixes the differences. The method itself is a version of the $load()$ function, which takes the segment number as a parameter and attempts to read all the values from this segment with corresponding offsets to point at other sectors. Analogically the code from $D1, D2$ and $D3$ fixes any inconsistencies between A, B and C.
4. Since the probability of a bit flip in the code memory is small, it is too time and power consuming to run through a whole segment searching for potential errors. Therefore, as part of a regular maintenance routine firstly a checksum is re-calculated for a given segment, then

it is compared to the stored checksums and any inconsistencies initiate the code-healing procedure.

The code-healing procedures are run regularly and they vary between checking $D1, D2$ and $D3$ or specific segments in A, B and C. For example, at the end of the looping procedure the checksums for the beginning of the code memory are compared in order to make sure that the looping procedure can start again.

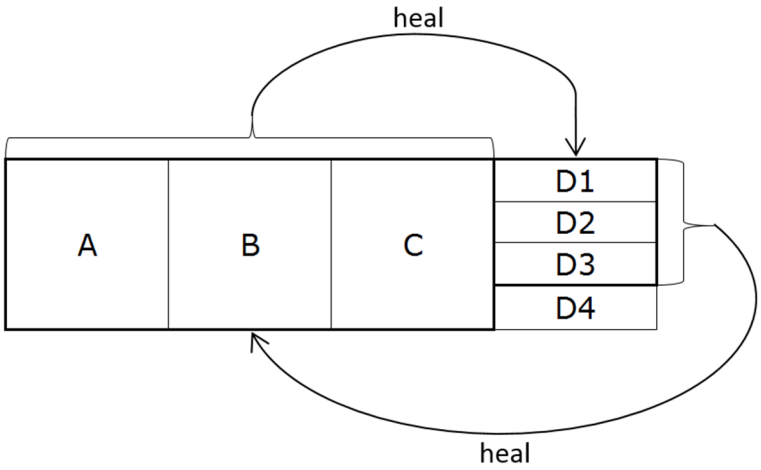

FIGURE 11: HEALING SEQUENCE

5.3 I2C Transmission

Communication between the MIC and the on-board sensors is crucial for the success of the PAMSAT mission. Therefore, extra attention is paid to the implementation of the I²C protocol on the aforementioned chip architectures. The major reason why standard I²C supported libraries could not be included is the limited memory space. The libraries contain functionality, which is not required for the PAMSAT mission and is very unlikely to be required in similar missions. To strip down the libraries to have only the required functionality requires a comprehensive understanding of how I²C works and how it is implemented in a chip. Both MSP430 and ATmega328P have the I²C hardware implementation built in. Therefore, the code has to perform very chip specific and very accurate registers' manipulations to launch the SCL bus and perform data bits transmissions via SDA.

Implementation of I²C varies significantly from architecture to architecture. After a thorough research it was possible to identify the registers and learn the runtime debugging messages, which are hardcoded by the manufacturers. All this information can generally be found in an extended version of documentation (datasheet) for a microcontroller. The debugging messages are necessary to detect whether a transmission is successful or faulty. The total I²C protocol then requires four distinct methods in a Master mode for the MIC (Algorithm 4).

In a case of I²C faults when any of the aforementioned procedures is stuck in an infinite loop waiting for a response, the whole system will be reset by its watchdog timer, which is enabled before the method call and is disabled or "fed"[4] afterwards to avoid unwanted restarts. The tailored code was tested for the memory efficiency on the MSP430 and ATmega328P. The test compares the total size of a simple program, which manages I²C data transmission with standard libraries, to a simple program, which performs exactly the same tasks, but does this with direct low-level register manipulation and is tailored for the space mission. The results are shown in Table 15 and the total amount of saved memory grows insignificantly with the number of times the I²C methods are used throughout the program.

[4] Feeding or kicking a watchdog are the terms used to say that the watchdog timer is reset and the countdown until system reset is restarted from the beginning

Algorithm 4. I2C data exchange. Source: Author

```
1:     void i2c_transmitInit($address){
2:           while (acknowledgment of success is not received) {
3:                 launch the I2C bus (engage pull-up resistor)
4:                 send start signal //such signal wakes up the peripherals
5:                 put $address into memory location for outgoing messages
6:                 set I2C to transmitting mode
7:                 initiate the transfer //generate transmission signal
8:                 wait for the slave to respond //Slaves respond with ack. bit
9:           }
10:    }
11:    void i2c_transmit ($message[N]){
12:          unsigned char i = 0
13:          while (i < N) {
14:                put $message[i] into outgoing box
15:                initiate the transfer
16:                wait for the transfer to happen
17:                if acknowledgment bit from the slave is not received
18:                      i -- //repeat the transfer of the same byte
19:                i++
20:          }
21:    }
22:
23:    void i2c_receiveInit($address) {
24:          while (acknowledgment of success is not received) {
25:                launch the I2C bus (engage pull-up resistor)
26:                send start signal
27:                put $address into memory location for outgoing messages
28:                set I2C to receiving mode
29:                initiate the transfer
30:                wait for the slave to respond
31:          }
32:
33:    }
34:    void i2c_receive($message [N]){
35:          unsigned char i = 0
36:          while (i < N) {
37:                initiate the delivery
38:                put received value into $message[i]
39:                wait for the transfer to settle
40:                if data is not received
41:                      i -- //no acknowledgment from the Master is sent
42:                i++
43:          }
44:    }
```

I2C data transfer protocol		
	Provided libraries	**Tailored methods**
MSP430	1,688 bytes	720 bytes
ATmega328P	1,454 bytes	672 bytes

TABLE 15: I2C SIZE COMPARISONS

5.4 Time Delay

The delay function is needed to time some of the processes on the MIC correctly. As the MIC is a soft real-time embedded system, knowledge of the exact real time is not essential. Moreover, it is impossible to continuously track the time due to inconsistent solar power supply. Therefore, for this project it is suffice to measure time in known factions of a minute. As it is mentioned above in this paper, since the memory space is a constraint, it was decided not to use the supplied delay functions, but to program lightweight ones specifically for PAMSAT. The delay functions implemented in this framework can measure one second and one millisecond intervals for 8MHz and 16MHz embedded processors (Algorithm 5).

Algorithm 5. Time delay. Source: Author

```
1: delayS ($time) { //delayMS is identical
2:     unsigned char i = 0
3:     while (i<$time) {
4:           perform dummy operation
5:           increment i
6:     }
7: }
```

All the values and the dummy operations in the delay functions are carefully calibrated separately for different chips and are stored as constants. When necessary, the time progress can also be saved to make sure that a certain procedure is run for at least a specific amount of time. For example, any transmissions of any signals by a spacecraft should not happen for at least 30 minutes after the spacecraft leaves its deployer.

5.5 CPU and ALU Tests

Errors in CPU registers and the ALU are not difficult to detect. However, the detection procedures may require very hardware specific assembly code. MSP430 has 12 general purpose registers and one ALU (Appendix 5). Therefore, the methods to detect faults are summarised in Table 16.

There is nothing software can do if CPU registers or ALU become permanently corrupted. However, permanent damage to the CPU core is unlikely and in most cases a restart solves a broad range of problems. Hence, a restart method is called if a fault is detected.

Location	Error detection method
CPU registers	Using assembly language, write known dummy values into all registers. Perform XOR on all values to reduce the computation speed. Compare the result with the manually calculated result. If the results are different, detect a fault. If the results are the same, then write another set of known dummy values into all registers. The second set of values is designed in such way that for each register it is checked that both 0's and 1's can be stored there. Perform XOR on all the registers again and compare it with the manually calculated value. If the results are different, detect a fault.
ALU	Perform multiple known arithmetic computations (summation, multiplication and division) using known values and compare it to the manually pre-computed results. If one of the calculations is not what is should be, detect a fault.

TABLE 16: CPU AND ALU FAULT DETECTION

5.6 Progress tracking

The program consists of many other different function calls, which use the functions described in this section so far and due to the high risk of power outages the progress has to be carefully tracked not to a repeat function call, which does not need to be repeated. Hence, all methods in the program also have progress tracking. To demonstrate the logic behind the usage of the program tracker, an example is shown in Algorithm 6.

Algorithm 6. Progress tracking (Radio signal). Source: Author

```
load($program_tracker1)
if $program_tracker1 is equal to the radio function checkpoint number {
    send_radio($message[N]){ //N is the length of the message
          load($program_tracker2) //initially it is 0
          for i from $program_tracker2 to N-1 {
                write message[i] into a corresponding register (for MSP430) OR
                      transfer message[i] via I2C to a radio chip
                load($sleep_timer) //initially it is 0
                while no acknowledgment that the transmission is complete {
                      sleep(0.5)
                      save($sleep_timer + 1)
                      if acknowledgement bit was not received in 30 seconds {
                            save($sleep_timer = 0)
                            restart the whole system
                      }
                }
                save(program_tracker2 + 1)//tells how many bytes is sent
          }
          save($program_tracker2 = 0)      //a whole message was sent
    }
    save($program_tracker1 + 1)      //the method is complete and we can move on
}
```

Similar logic applies to all other functions. This way ensures that the code is executed successively regardless of power outages. A function has to complete, before the next function runs. Of course, there exists a risk of a piece of code being run multiple times due to inconsistent power, but this risk is minimised through making sure that no distinct piece of code has to run for longer than 1 second without its progress being closely tracked.

The implementation of other methods (Appendix 6) is straightforward and self-explanatory. Overall, the program allows safe storage of all the required data and safe following of the program progress at all times. Thus, as long as a spacecraft has some supply of energy, even if it inconsistent, the program is able to fix bit flips caused by SEE/SEU, restart itself in case it enters an infinite loop or is frozen, and resumes execution from the point where it was interrupted. All the functionality described in this section was developed with the purpose of keeping it light-weight, easy-to-understand and make it as portable between various architectures as possible. A good example of code efficiency is shown in Table 18, which shows how beneficial it is to even remove simple *#define* macros from the code.

Set the third bit in a variable *abc* to zero	Size

`//Used in most libraries`
`#define _BV(bit)  (1 << bit)`
`#define cbi(sfr, bit)  (sfr &= ~(_BV(bit))`
`abc = cbi(abc, 3);`
	14 bytes

`//my simplification`
`#define CLEAR_BIT3  0b11111011`
`abc  &= CLEAR_BIT3;`
	4 bytes

TABLE 17: MEMORY EFFICIENCY EXAMPLE

6 Evaluation and Testing Results

The framework was implemented on the MSP430 chip architecture using the C programming language and a dummy fault-tolerant program, which performed a simple LED blink to demonstrate that it works reliably. The total size of the program, which is stored in section A is 2,828 bytes. Hence, the total amount of code memory occupied with the TMR methodology is:

$$2{,}512 * 3 = 8{,}484\ bytes$$

This means that the amount of space, which can be customized as required by space mission specifications is

$$8{,}000 - 2{,}828 = 5{,}172\ bytes$$

Considering how many faults can be detected and addressed with the code, which only takes 2,8kB of memory, it is reasonable to conclude that there is plenty of space for any modifications to this framework, if programmed with care. Hence, anyone, who chooses to use this framework, will be able to adjust it to their needs as they see fit without significant concerns about the space constraints.

The code-healing method, which heals A, B and C is stored in three copies in sections $D1, D2$, and $D3$ and in each section it only occupies 298 bytes. Therefore, in a case when the memory is not sufficient, it is possible to slightly expand the code memory by 1,702 bytes (2kB size of each of four D sections less 298 bytes) at the expense of sections $D1, D2$, and $D3$.

A variety of tests has been developed for the software and a large number of errors were imitated using the development environment Code Composer Studio 6.1.3 and different test routines. The tests, which cover the potential scenarios of errors and their results, are described in Table 19.

Test	Result
I^2C bus jamming	Program is frozen and restarted by watchdog timer
Bit flip in code memory at a random memory address	Error is corrected when found with regular checksum detection and code-healing
Bit flip in code memory at currently running code	The program is frozen and restarted by watchdog timer. Error is then corrected with regular checksum detection and code-healing
Bit flip in the code memory in error-fixing routines	Multiple restarts due to faulty code. Error is corrected when another code-healing code is running from the different memory segment
Bit flip in the first address of code memory when MIC is switched off	The program is frozen and no recovery
Bit flip in the first segment of code memory when MIC is switched off	The program is frozen. After the reset by watchdog, the self-healing is performed from another segment
Bit flip in the beginning of the code memory in the moment when looping cycle is coming to an end	Error is corrected when found with regular checksum detection and code-healing performed on the beginning of the code

Test	Result
Bit flip in the beginning of the code memory in the moment when looping cycle has just finished	The program is frozen. After the reset by the watchdog, self-healing is performed from another segment
Bit flip in RAM	Error is corrected as soon as the faulty variable is used
Bit flip in info memory	Error is corrected as soon as the faulty variable is used
Faulty value is stored in a CPU register	MIC performs restart when CPU registers checking routine is called and the fault is found
ALU cannot perform calculations correctly	MIC performs restart when ALU checking routine is called and the fault is found
Random power supply interruptions with 1-2 second intervals	The program continues from where it was interrupted after running a short setup

TABLE 18: MAIN TEST RESULTS

In Figure 12 a test function has changed the memory address 0x008064h from 0x40 to 0x4C. After the healing procedure runs from section *D*, it fixes the code back to 0x40.

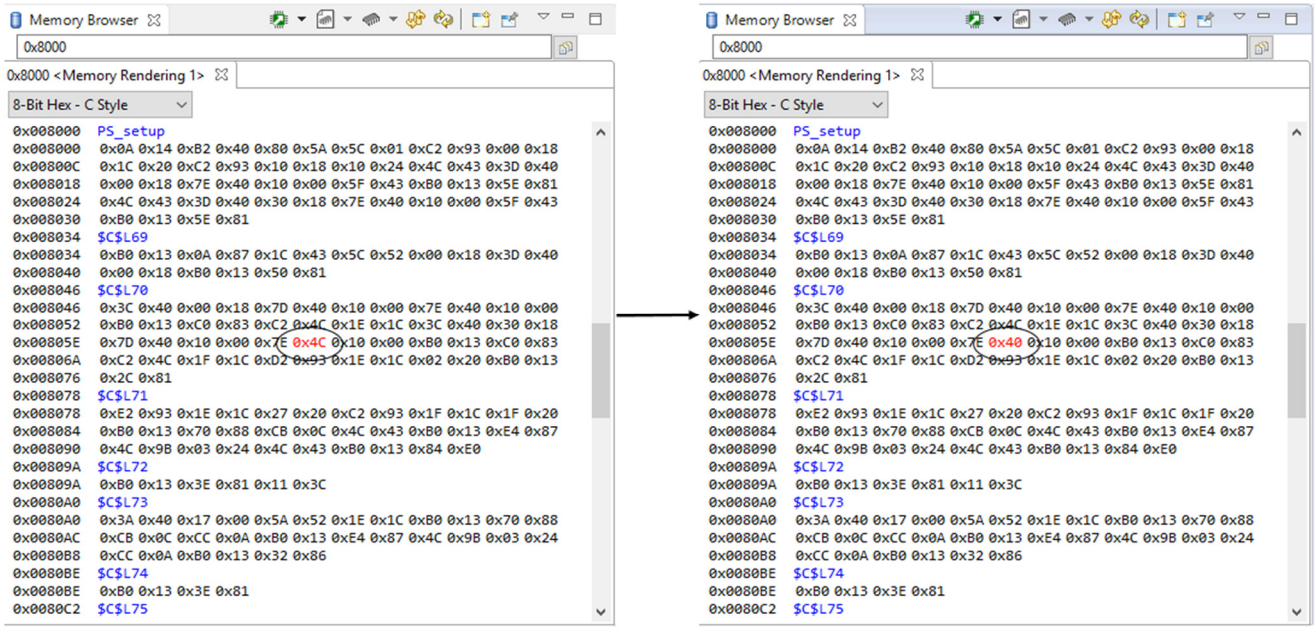

FIGURE 12: CODE SELF-HEALING EXAMPLE

Certainly, there is a fair amount of probability of an unrecoverable fault happening. Such a fault can only occur in the very beginning of the stored code and only during the times of absence of power supply to MIC. If a program managed to load successfully, it will check for faults and fix them in the subsequent segments of code memory. In the end of the program it checks for errors and bit flips, which might have occurred in the beginning of the code memory, and fixes them. Thus, the code is always in a circular loop of healing itself

7 Conclusion

The main aim of this project was to develop a framework for COTS microcontrollers, which are intended to be used in nanosatellite (such as CubeSat and PocketQube) space applications in the LEO space environment. To achieve this, firstly, the LEO environment had to be explored and thoroughly researched and the main threats to COTS electronics needed to be identified. The potential sources of faults and the types of faults had to be known to develop robust and stable algorithms, which would be able to deal with such errors. Additionally, other nanosatellite design limitations are discussed as part of this project. The framework is designed specifically for, but not limited to, LEO conditions and is presented in a form of fault tolerant, fault-detecting and self-healing software, which is uploaded to the MIC of a spacecraft. The software implements various error management methods, which are discussed and developed as part of this project, to achieve consistent and predictable execution of the program. This program is tailored to the needs of a bigger project – PAMSAT, which is described in the beginning of this paper. However, it was created with the intention to be open-source and available to other universities and researchers, if not as the core on-board spacecraft maintenance software, then as a guidance for developing similar fault tolerant software.

The amount of space research done using nanosatellites has been growing significantly in recent years (Figure 13) and more than 20% of them have size 1U or smaller (Figure 14). Moreover, the amount of planned launches for 2016 only is more than the amount of total nanosatellites launched to this day (Figure 13). Therefore, an obvious conclusion can be made that this area of research is rapidly gaining popularity and there is a need for a framework like the one developed as part of this project.

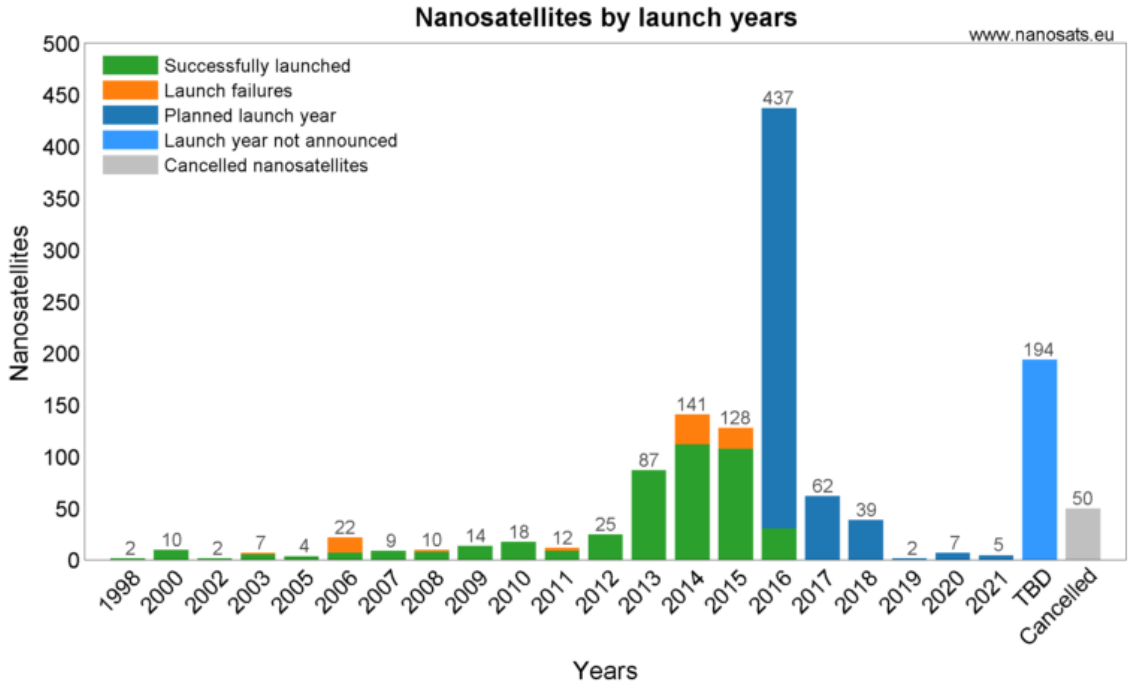

FIGURE 13: NANOSATELLITE LAUNCHES BY YEAR [48]

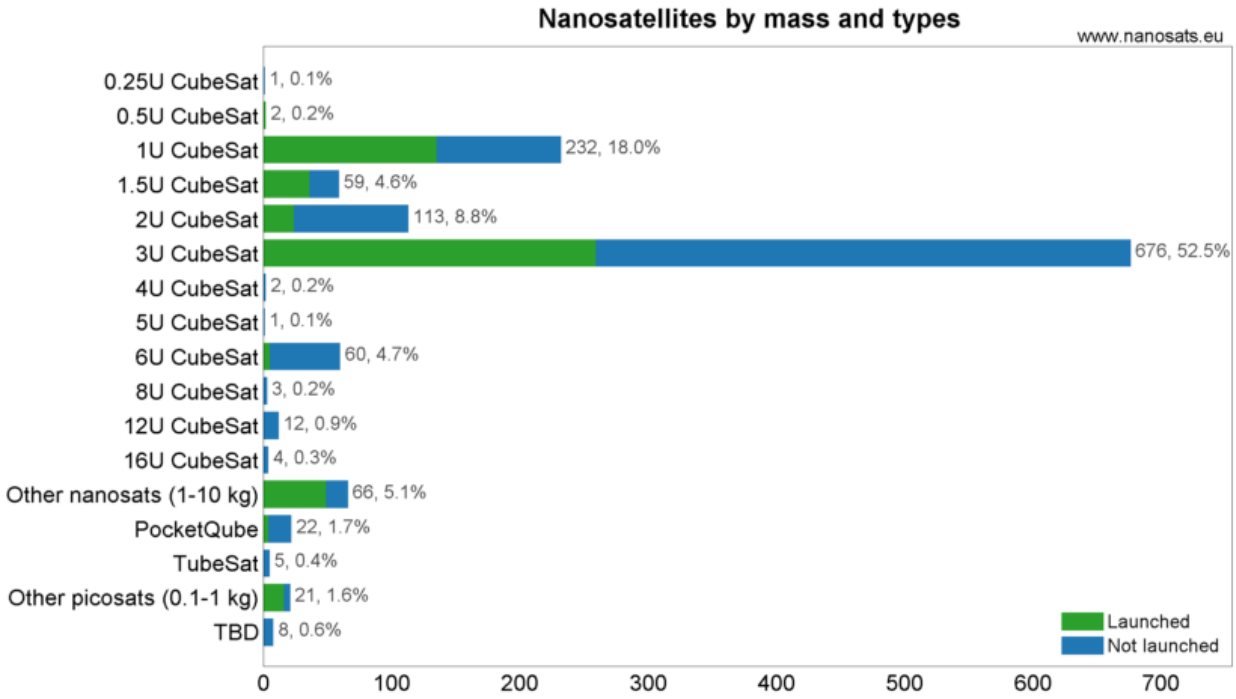

FIGURE 14: NANOSATELLITES BY MASS AND TYPES [48]

The thorough testing of the framework has been performed as part of this work and it can be said with confidence that the product of this research is a reliable and fault tolerant system with self-healing features and functionality required for a generic space mission (sensor communication, data storage, progress tracking etc.). In addition to this, the code base was very carefully designed and kept to a minimum in terms of speed of the code execution and the memory the code occupies on a microcontroller.

Therefore, it is possible to conclude that all the research objectives have been successfully achieved and, if applied correctly, this framework and the program should help researchers engineer noticeably easier, productive, and cost-efficient space missions, which use COTS electronic components, reliably and successfully.

7.1 Future Work

There is still plenty of potential for this project as it can be advanced even further in terms of providing fault tolerant control system for embedded microcontrollers like the ones looked at as part of this work.

Firstly, once the physical satellite is built as part of the PAMSAT project, it will be possible and is necessary to perform additional tests to establish the stability of the software and the framework in general. Such tests include, but are not limited to:

1. Vibration testing – theoretically, abnormal levels of vibration of the body of a spacecraft should not have any influence on the code, which is stored in MIC, and there is no evidence proving otherwise. Hence, it is difficult to forecast what kind of faults can occur in a microcontroller program during its launch and deployment in LEO. However, such testing is still required to minimize the risks of space mission failure.

2. Vacuum chamber testing – this test can imitate the LEO environment as closely as possible. In addition to a spacecraft being placed in a complete vacuum, it also is exposed to an imitation Sun and, hence, day and night conditions, which the satellite is most likely to encounter. Therefore, this test is crucial for the software as it will be possible to see the behaviour of it and its performance in a space-like environment.

Furthermore, alternative solutions to similar problems can be investigated and other possible faulty scenarios could be developed. Currently the biggest threat to the MIC is that a bit flip or similar error will happen in the beginning of the code memory, while the spacecraft is offline, and it will not be able to load up. The probability of such outcome is extremely small, but still present.

Overall, according to Moore's law, the performance and capacity of current COTS microcontrollers increases exponentially [49] from year to year. There is no doubt that future microcontrollers, which can be considered as the MIC of a nanosatellite, will outperform the current ones significantly. Therefore, they will be able to manage many more processes, handle more complicated faults programmatically, provide more options for storing the machine code and allow the spacecraft to exchange data with on-board peripherals and sensors using other protocols, such as Serial Peripheral Interface Bus (SPI) [50] and perhaps Universal Serial Bus (USB).

8 References

[1] Redd, Nola Taylor. How Big is the Universe? Space.com. [Online] 24. 12 2013. http://www.space.com/24073-how-big-is-the-universe.html.

[2] UCSB ScienceLine. [Online] 2015. [Citeret: 09. 06 2016.] http://scienceline.ucsb.edu/getkey.php?key=3775.

[3] Staff, Listverse. Listverse. [Online] Listverse Ltd., 31. 12 2009. [Citeret: 05. 09 2016.] http://listverse.com/2009/12/31/top-10-mysteries-of-outer-space/.

[4] Laboratory, Jet Propulsion. JPL NASA. [Online] 06 1991. [Citeret: 09. 06 2016.] http://www2.jpl.nasa.gov/files/educator/oursolar.txt.

[5] Tate, Karl. Distances Driven on Other Worlds. [Online] Space.com, 24. 03 2015. [Citeret: 10. 06 2016.] http://www.space.com/79-distances-driven-on-other-worlds.html.

[6] Johnson, Michael. www.pocketspacecraft.com. 01. 06 2016.

[7] JPL. http://www.jpl.nasa.gov/missions/?type=current. [Online] JPL NASA, 2016. [Citeret: 05. 09 2016.] http://www.jpl.nasa.gov/missions/?type=current.

[8] Virgin Galactic is back in the space race. [Online] Daily Mail, 19. 02 2016. [Citeret: 08. 09 2016.] http://www.dailymail.co.uk/sciencetech/article-3455294/Virgin-Galactic-space-race-Professor-Stephen-Hawking-unveils-Unity-spaceplane-blast-tourists-orbit.html.

[9] —. Where are the Voyagers? [Online] JPL NASA, 2016. [Citeret: 05. 09 2016.] http://voyager.jpl.nasa.gov/where.

[10] Bray, Nancy. Space Shuttle and International Space Station. [Online] NASA, 31. 07 2015. [Citeret: 05. 09 2016.] http://www.nasa.gov/centers/kennedy/about/information/shuttle_faq.html.

[11] SCHEDULE & PRICING. [Online] Spaceflight, 2016. [Citeret: 05. 09 2016.] http://www.spaceflight.com/schedule-pricing/.

[12] PocketQube. [Online] Wikipedia. [Citeret: 10. 06 2016.] https://en.wikipedia.org/wiki/PocketQube#cite_note-1.

[13] Scienceogram. The cost of space missions. [Online] Scienceogram UK, 2015. [Citeret: 05. 09 2016.] https://scienceogram.org/blog/2015/07/space-missions-cost-new-horizons/.

[14] A community-driven initiative to provide open source solutions for space and earth exploration. [Online] librecube.net/. [Citeret: 06. 09 2016.] http://librecube.net/.

[15] Machine shop for NewSpace. [Online] radiusspace.com/. [Citeret: 06. 09 2016.] http://www.radiusspace.com/.

[16] Li Qiao, Chris Rizos, Andrew G. Dempster. Analysis and Comparison of CubeSat Lifetime. Sydney : University of New South Wales, 2013.

[17] Daniel L. Oltrogge, Kyle Leveque. An Evaluation of CubeSat Orbital Decay. USA : AGI, 2011.

[18] Michael Johnson, James Bird, David Cava Pina, Benhur Johnson, Mathew Santer. TWIST. London : Imperial College London, 2015.

[19] James Bird, David Cava Pina, Benhur Johnson, Michael Johnson, Mathew Santer. TWIST - High performance thin film small satellite subsystems with a twist. London : Imperial College London, 2015.

[20] Castle, Alex. Know Your Arduino: A Practical Guide to The Most Common Boards. [Online] tested.com, 12. 06 2013. [Citeret: 08. 09 2016.] http://www.tested.com/tech/robots/456466-know-your-arduino-guide-most-common-boards/.

[21] NXP. I2C-bus specification and user manual. [Online] 04. 04 2014. [Citeret: 10. 06 2016.] http://www.nxp.com/documents/user_manual/UM10204.pdf.

[22] Wikipedia. I²C. [Online] [Citeret: 09. 06 2016.] https://upload.wikimedia.org/wikipedia/commons/3/3e/I2C.svg.

[23] Screening of nanosatellite microprocessors using californium single-event latch-up test results. Takahiro Tomioka, Yuta Okumura, Hirokazu Masui, Koichi Takamiya, Mengu Cho. 2016, Acta Astronautica, Årg. 126, s. 334-341.

[24] Radiation effects and cots parts in smallsats. Doug Sinclair, Jonathan Dyer. 2013. Small Satellite Conference.

[26] Kenneth A. LaBel, Michele M. Gates, Amy K. Moran. Commercial Microelectronics Technologies for Applications in the Satellite Radiation Environment. s.l. : Aerospace Applications Conference, IEEE, 1996.

[27] JSC. Space Radiation Effects on Electronic Components in Low-Earth Orbit. [Online] 01. 02 1999. [Citeret: 05. 09 2016.] http://llis.nasa.gov/lesson/824.

[28] Holbert, Dr. Total Ionizing Dose. [Online] ASU Electrical Engineering, 13. 02 2007. [Citeret: 05. 09 2016.] http://holbert.faculty.asu.edu/eee560/tiondose.html.

[29] William B. Stine, Michael Geyer. Power From The Sun. s.l. : www.powerfromthesun.net, 2001.

[30] Nielsen, Ron. Solar Radiation. Austrailia : http://home.iprimus.com.au/nielsens/, 2005.

[31] Single Event Upsets. [Online] ALTERA, 2016. [Citeret: 06. 09 2016.] https://www.altera.com/support/quality-and-reliability/seu.html.

[32] Insolation - weather. [Online] Renewable Energy Concepts. [Citeret: 05. 09 2016.] http://www.renewable-energy-concepts.com/solarenergy/solar-basics/insolation-weather.html.

[33] Earth is being protected by a 'Star Trek-style invisible shield': Scientists probe mysterious barrier blocking 'killer electrons'. [Online] Daily Mail, 26. 11 2014. [Citeret: 07. 09 2016.]

http://www.dailymail.co.uk/sciencetech/article-2850566/Earth-protected-Star-Trek-style-invisible-shield-Scientists-probe-mysterious-barrier-blocking-killer-electrons.html.

[33] Jonas Friedel, Sean McKibbon. Thermal Analysis of the CubeSat CP3 Satellite. San Luis Obispo : Aerospace Engineering Department, 2011.

[34] Flight Results of the Compass-1 Picosatellite Mission. Artur Scholz, Wilfried Ley, Bernd Dachwald, Engelbert Plescher, Jiun-Jih Miau, Jyh-Ching Juang. 191, 2010, AMSAT UK OSCAR NEWS, p. 21.

[35] Microsemi. Single Event Effects (SEE). [Online] Microsemi, 2016. [Citeret: 06. 09 2016.] http://www.microsemi.com/products/fpga-soc/reliability/see.

[36] Satellite Reliability: Statistical Data Analysis and Modeling. Jean-Francois Castet, Joseph H. Saleh. 5, 2009, Journal of Spacecraft and Rockets, Årg. 46, s. 1065-1076.

[37] Fault tolerance through redundant COTS components for satellite processing applications. I.V. McLoughlin, V. Gupta, G.S. Sandhu. 2003. Information, Communications and Signal Processing.

[38] What are possible power sources for spacecraft? [Online] qrg.northwestern.edu. [Citeret: 06. 09 2016.] http://www.qrg.northwestern.edu/projects/vss/docs/power/zoom-possible-powers.html.

[39] NASA. Space batteries! [Online] NASA, 09. 05 2011. [Citeret: 06. 09 2016.] http://spaceplace.nasa.gov/batteries/en/.

[40] Franke, Björn. Embedded Systems: Reliability & Fault Tolerance. Edinburgh : University of Edinburgh.

[41] Daniel P. Siewiorek, Priya Narasimhan. Fault-Tolerant Architectures for Space and Avionics Applications. Pittsburgh : Carnegie Mellon University, 2005.

[42] Dubrova, Elena. Software Redundancy. Fault-Tolerant Design. s.l. : Springer New York, 2013, s. 157-179.

[43] —. Software redundancy. Stockholm : Royal Institute of Technology.

[44] Analyzing Area and Performance Penalty of Protecting Different Digital Modules with Hamming Code and Triple Modular Redundancy. R. Hentschke, F. Marques, F. Lima, L. Carro, A. Susin, R. Reis. Washington : Universidade Federal do Rio Grande do Sul, 2002. 15th symposium on Integrated circuits and systems design.

[45] Badodekar, Nilesh. Mitigating Single-Event Upsets Using Cypress's 65-nm Asynchronous SRAM. s.l. : Cypress, 2016.

[46] Redundant System Basic Concepts. [Online] National Instruments, 11. 01 2008. [Citeret: 06. 09 2016.] http://www.ni.com/white-paper/6874/en/.

[47] Rouse, Margaret. NAND flash wear-out. [Online] TechTarget. [Citeret: 06. 09 2016.] http://searchsolidstatestorage.techtarget.com/definition/NAND-flash-wear-out.

[48] Nanosatellite Database. [Online] nanosats.eu, 15. 05 2016. [Citeret: 06. 09 2016.] http://www.nanosats.eu/.

[49] Moore's Law. [Online] Investopedia, 2016. [Citeret: 06. 09 2016.] http://www.investopedia.com/terms/m/mooreslaw.asp.

[50] Wikipedia. Serial Peripheral Interface Bus. [Online] [Citeret: 09. 06 2016.] https://upload.wikimedia.org/wikipedia/commons/f/fc/SPI_three_slaves.svg.

[51] Bin Suoa, Yong-sheng Cheng, Chao Zeng, , Jun Li. Calculation of Failure Probability of Series and Parallel Systems for Imprecise Probability. I.J. Engineering and Manufacturing. April 2012, s. 79-85.

[52] Instruments, Texas. MSP430™ SoC With RF Core. MSP430 Datasheet. s.l. : Texas Instruments, 2013.

9 Appendices

9.1 Appendix 1

FIGURE 15: EEE ROOFTOP SATELLITE 70CM AND 2M ANTENNAS #1

FIGURE 16: EEE ROOFTOP SATELLITE 70CM AND 2M ANTENNAS #2

FIGURE 17: EEE ROOFTOP SATELLITE 70CM AND 2M ANTENNAS #3

9.2 Appendix 2

Probability of failure of a redundant system.

The fault tolerance of a parallel system with multiple components is exponentially higher that of a single MIC. The failure likelihood of such system is shown in Equation 1, where P is the total probability of failure and P_i is a probability of failure of individual MIC [51].

$$P = \prod_{i=1}^{n} P_i$$

$$P = P_1^n \ , where \ P_i = P_j, i \neq j$$

EQUATION 1: PROBABILITY OF FAILURE OF PARALLEL SYSTEM WITH N COMPONENTS

9.3 Appendix 3

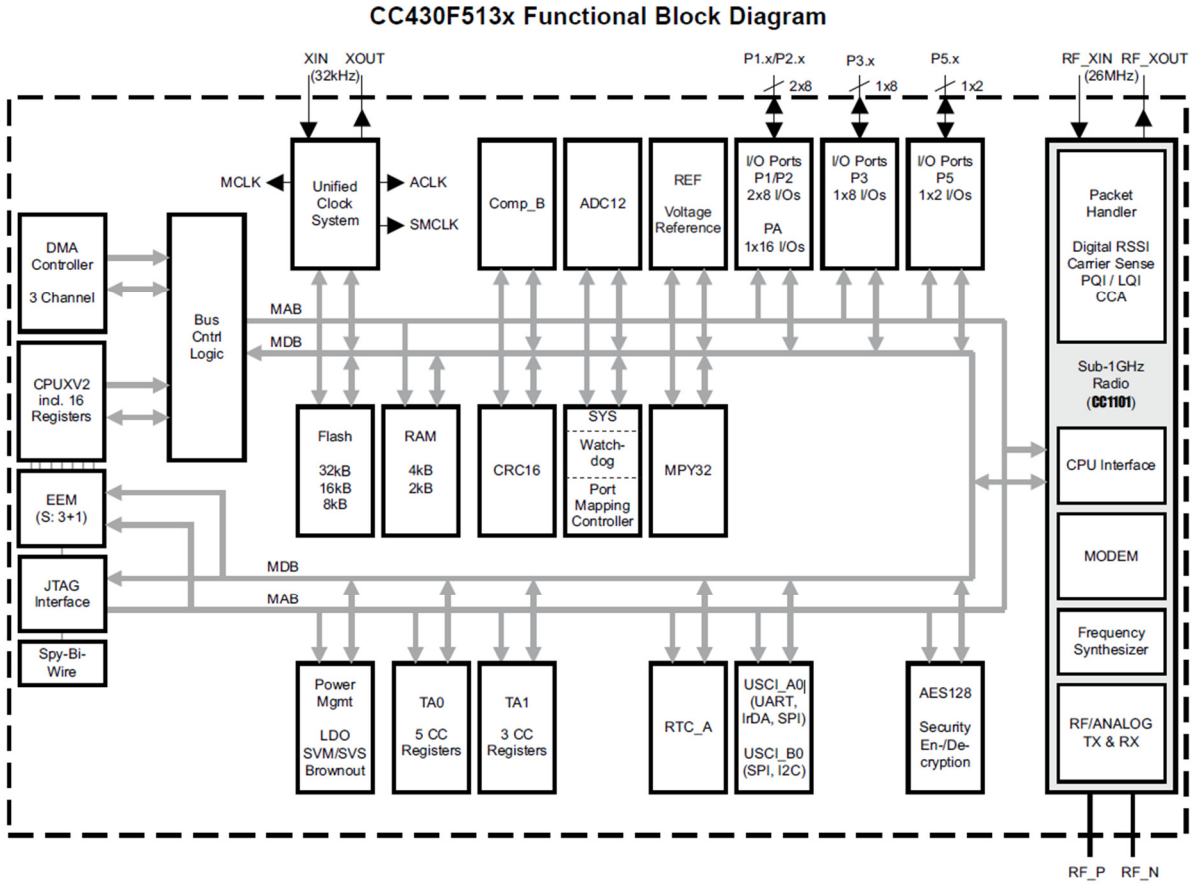

FIGURE 18: CC430F5137 FUNCTIONAL BLOCK DIAGRAM [52]

9.4 Appendix 4

		CC430F6137 CC430F6127 CC430F5137 (1)
Main Memory (flash)	Total Size	32kB
Main: Interrupt vector		00FFFFh to 00FF80h
Main: code memory	Bank 0	32kB 00FFFFh to 008000h
RAM	Total Size	4kB
	Sect 1	2kB 002BFFh to 002400h
	Sect 0	2kB 0023FFh to 001C00h
Device Descriptor		128 B 001AFFh to 001A80h
		128 B 001A7Fh to 001A00h
Information memory (flash)	Info A	128 B 0019FFh to 001980h
	Info B	128 B 00197Fh to 001900h
	Info C	128 B 0018FFh to 001880h
	Info D	128 B 00187Fh to 001800h
Bootstrap loader (BSL) memory (flash)	BSL 3	512 B 0017FFh to 001600h
	BSL 2	512 B 0015FFh to 001400h
	BSL 1	512 B 0013FFh to 001200h
	BSL 0	512 B 0011FFh to 001000h
Peripherals		4 KB 000FFFh to 0h

FIGURE 19: MSP430 MEMORY MAP [52]

9.5 Appendix 5

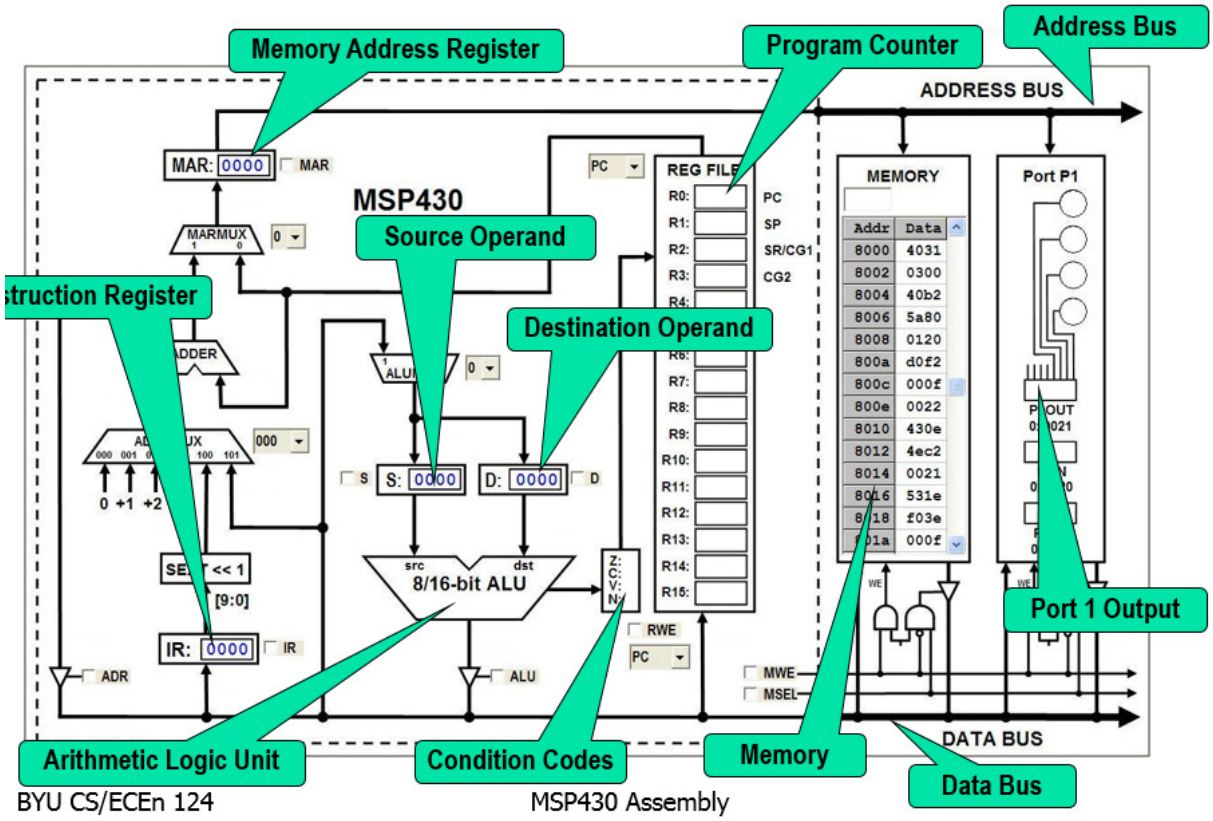

FIGURE 20: MSP430 ARCHITECTURE [52]

9.6 Appendix 6

Setup procedure on every loading	
Function	**Description**

Check if the deployment machine instructions are correct	1: code_check(N){ //N is the number of the segment to be checked
	2: let the values at $address_A + N*SECTOR_SIZE,
	$address_B + N*SECTOR_SIZE,
	$address_C + N*SECTOR_SIZE be $a1, $a2 and $a3
	3: while $a1 < $address_B + N*SECTOR_SIZE {
	4: if $a1, $a2 and $a3 are not same {
	5: $correct_code =
	error_correct($a1, $a2, $a3)
	6: re-save $correct_code in all locations,
	which have discrepant values
	7: }
	8: increment $a1, $a2 and $a3
	9: }
	10: }

Only once: Run current through specific pins to melt restraining fishing lines and deploy antennae and solar panels	1: run_current($pin){
	2: //Pin addresses are hardware dependent
	3: //and need to be predefined in advance
	4: set the $pin to Output
	5: set the $pin to High //enable voltage at that pin
	7: wait for X seconds { // to ensure that the fishing lines melted
	8: if the wait was interrupted (power outage)
	9: re-set the X
	10: //once the solar panels are deployed, the power
	//supply will be more consistent and last for X
	//seconds
	11: set the $pin to Low
	12: save $program_tracker in the permanent memory to reflect that this method is complete
	13: }

Check if the program machine instructions are correct	Similar to the check of the deployment sequence
Set-up watchdog timer as required	A hardware specific value with the timer configuration is written into a watchdog register. Watchdog is
Load saved variables into RAM and check if loading is successful	1: load_variable($variable, $address){ 2: $variable = load($address) 3: 4: if($variable != load($address)) 5: try again 6: }
Check if the saved variables are correct	Function implemented in loading and/or saving a variable

Function	Description
Check if constant values are correct	Constants are hardcoded in the program code. Hence, they are checked with the program code
Load the most recent program tracker	Same as loading a variable from a specific memory address. Memory address is reserved for the program tracker
Perform I2C data bus transfer testing	Initialisation of the transmission performs all the required tests of the I2C bus
Resume the operations from where the program was last interrupted	Resuming works through loading the $program_tracker and then locating the line of code, which should be executed for the corresponding $program_tracker. After each operation the $program_tracker is incremented to reflect the progress

Looping procedure once setup is complete

Function	Description

Request data from on-board sensors	`1: i2c_request($address){`
	`2: i2c_transmitInit($address)`
	`3: i2c_transmit($message)`
	`4: //message is device specific signal to begin transmission`
	`5: }`

Receive data from on-board sensors	`1: i2c_receive_data($address, $buffer){`
	`2: i2c_receiveInit($address)`
	`3: i2c_receive ($buffer)`
	`4: }`

Process the data	Depends purely on the payload. At this stage payload has not been confirmed yet. The data is stored in $buffer and is ready to be processed
Send the data via radio to the Earth	Same as transmitting a radio signal in the beginning of the program

Listen to commands from the Earth for a specific amount of time	`1: receive_radio(){`
	`2: //radio device specific`
	`3: initialise radio module to receive signals`
	`4: check every X seconds`
	`5: for Y seconds`
	`6: if anything was received`
	`7: if received{`
	`8: format the message into "address" + "data"`
	`9: save($data, $address)`
	`10: }`
	`11: }`

TABLE 19: REMAINING PSEUDOCODE