POLITECNICO DI TORINO

Department of Electronics and Telecommunications

Master Degree in Electronic Engineering

Development of a payload for the characterization of commercial microcontrollers to radiations

Supervisors

Prof. Claudio Sansoè

Prof. Leonardo Maria Reyneri

Candidate

Arturo Guadalupi

July 2015

to my girlfriend who gives me support and helps me during this long journey together

and to my family who always trust in me

Acknowledgements

I want to say thanks to my family, all the people and all the friends who were my companions during these five years in the Politecnico. This was a journey made up of tears, joy and laughs, and without support very difficult to bear. Thanks to my colleagues of Officine Arduino for the opportunity they gave me to study and work at the same time and for the doors they opened me. A special thanks to my liuther friend Andrea Capurso for the help, support and trust in my audio related skills. Thanks to professor Leonardo Reyneri and professor Claudio Sansoé for their help during my thesis period. Thanks to the music, my guitar and the tubes for maintaining my mental sanity. Thanks to Doctor Who, Game Of Thrones and all the TV series and animes watched during the exams sessions in these five years. Finally, I want to say thanks to all the people who always have envied my results, because while I was happy of my trophies, yours rage encouraged me more and more.

Arturo Guadalupi

Summary

The main scope of this thesis is to prepare the base for the use and the radiation characterization of the new Texas Instruments' FRAM micro-controllers within the Modular Architecture for Satellites (AraMIS) developed by the Politecnico di Torino. These kind of micro-controllers seem to be very appealing for space applications based on Commercial Off The Shelf (COTS) components because of their intrinsically radiation hardened structure and their low power consumption compared with standard FLASH based one.

The idea of using ferroelectric materials to store digital can be dated back to 1952, but it was practically implemented only starting from the 80s because the needed advanced technology to develop them wasn't available before. FRAM based micro-controllers are instead available on the market since about one year and an half. The ferroelectric RAM memory, known as FeRAMF or FRAM, is **conceptually** similar to the DRAM cell, but there is an important difference that lies in the dielectric of the storage capacitor: while DRAM cells use a layer of standard linear material, the dielectric of a FeRAM cell is made of ferroelectric material, usually lead (Pb) Zirconate Titanate (PZT).

Using a ferroelectric dielectric leads to a different behavior of the cell compared with a DRAM one, leading to many advantages especially for what concern the overall power consumption in read/write cycles. Furthermore, the material exhibits two stable polarization conditions and it's possible to switch between them by means of an electric field with opposite polarity. Since the polarization will be kept after the applied field is removed, it is possible to link the polarization state to a logic state and so these materials can be used to build a **non volatile** memory device. No periodic refresh is so necessary to keep the information, like in a DRAM memory.

The reading process is destructive: it is not possible to read the content of a cell without actually clearing it, because of the way the information is stored in the device.

To know which of the possible polarization states the dielectric holds, the only way is to write a new value to the cell with the bit-line pre-charged but in high impedance state and depending on the previous polarization, this process will or won't produce a voltage pulse out of the bit-line. Read and write cycles require basically the same operations and can both be completed in times in the order of tens of nanoseconds and without using high voltage charge pump like in FLASH memories.

The three main design parameters of the electronic systems of small satellites are:

- power consumption;
- physical dimensions;
- radiation environment behavior.

The electric power in the satellite comes from solar panels, which are necessarily of small dimensions because of the mechanical structure, leading to few Watt of average power to cover all the needed functions. It is so necessary to make the best use of any mW of available power. Furthermore launch costs are directly proportional to the mass of the system, so it is absolutely necessary to reduce as much as possible dimensions and mass of the electronic system. We said that FeRAM memories are RAM devices, meaning that read and write procedures do not differ significantly and random write is possible without the need of a previous erase of a cell, but they are also non volatile, we can so for sure state that this leads to save power. In fact in DRAM devices most of the power is used by the refresh procedure otherwise the stored informations are lost. Furthermore the refresh process leads to a decreasing in the overall speed performances. At the moment there are no big FRAM memory available on the market, but in any case, memory requirements of small satellites are normally compatible with the size of available FeRAM, except for imaging payloads if local storage of a certain number of images is mandatory.

Because the FRAM cell stores the state as a PZT film polarization, an alpha hit have a very small possibility to cause a change in the polarization. FRAM terrestrial Soft Error Rate (SER) is not even measurable. This "radiation resistant" characteristic of FRAM makes it attractive for use in several medical applications and space one.

Keeping in mind the concepts exposed above, this work is focused on the development of a payload tile for the AraMIS structure called 1B521 Radiation Characterization Payload whose aim is to introduce the usage of new FRAM micro-controllers within the AraMIS nano-satellite structure and characterize them for low cost space applications in therms of radiations. In particular it is requested to characterize the use of a FRAM micro-controller (MSP430FR6989) in therms of Total Ionizing Dose (TID), Single Event Effect (SEE), like Single Event Upset (SEU) and Single Event Latch-up (SEL), power efficiency and reliability in general. No scientific data coming from real space experiments or terrestrial simulations (using for example particles accelerators) are available at the moment. It is furthermore requested to show the efficiency of the AraMIS' developed software hardening library in order to have a direct comparison between a standard compiled code and an hardened one.

The AraMIS radiation-hardening technique, is based on the use of appropriate C++ classes from the hardened data (**Hdata**) package developed in house, which can be used in a common C++ program instead of standard data type. For instance, a short can be substituted by the so-called **TripleShort**, which automatically and transparently stores three copies of the same value and votes or recovers data whenever required. A normal C++ program can so still be compiled by modifying only the data type definitions. This makes possible to reuse software algorithms and procedures which have already been validated and tested without any specific effort apart from redefining data types drastically reducing the development time.

This thesis has to be considered an user guide manual about the developed payload tile, and a base for future developments on FRAM microcontrollers within the AraMIS nano-satellite structure. The first part of the developed work in-fact makes possible to introduce and start using any kind of FRAM micro-controllers that belongs to the family MSP430FRxxxx without an heavy effort. All the hardware-dependent choice that have been made are explained and the software commented in order to be easily understandable and useful for feature developments.

Here a little overview about the structure of the thesis.
Chapter 1 gives an introduction about the space radiation environments, its interaction with the electronics and the used shielding techniques.

Chapter 2 gives an overview about the FRAM technology and some concept about their pro and cons about their use in space applications.

Chapter 3 and *Chapter 4* give an overview about the UML approach in the AraMIS structure and how it is organized.

Chapter 5 explain the design of the developed PCB and what hardware has been chosen in order to give support to the developed software.

Chapter 6 shows the software structures behind the designed tile, how to it communicates with the OBC and the type of tests that are executed.

Chapter 7 gives an overview about the tests that has been made to validate the work and the reached results. What can be done in the feature to improve the what have been done is also mentioned.

Contents

List of Figures

CHAPTER 1

The space radiation environment and electronics

The space radiation environment can have serious effects on spacecraft electronics. Transiting cosmic rays of galactic, solar origin and their interaction with the earth's magnetic field is considered because these effects will limit system endurance and reliability. Transient effects from individual high-energy protons or cosmic rays can in fact disrupt system operation irreversibly causing system faults that can be very dangerous. The internal radiation environment can be described in terms of shielding the high-energy electrons, protons, and cosmic rays of the external environment: the exposure levels can be presented in terms of ionizing radiation dose and particle fluence for comparison to electronic component damage susceptibility. Transient effects can be described in terms of particle flux for assessment of the potential frequency or probability of critical effects in the electronics and of particular importance are the limits in shielding effectiveness for high energy electrons, protons, and cosmic rays. The interactions between the space radiation environment and the spacecraft electronics include consider the external surfaces as well as the internal electronics:

- External surfaces include degradation of solar cells or charging of dielectric material, which can lead to arc-discharges. For these reasons characterize the free-field electron and proton environments as a function of particle energy and time are important;
- Internal spacecraft radiation environment is defined by particle transport through

the spacecraft structure and if used shielding that protect sensitive electronic piece-parts like memories. Important effects:

- Performance degradation due to the energy deposition by accumulated ionization in the semiconductor materials;
- Atomic displacement damage due to high-energy protons;
- Transient effects resulting due to the interaction of a single cosmic ray or high-energy proton.

1.1 The trapped radiation environment

The earth's natural radiation environment consists of electrons, protons, and heavy ions. In particular this particles are:

- Trapped by the earth's magnetic field;
- Transiting through the domains of the earth's artificial satellites.

As the earth sweeps through the solar wind, a geomagnetic cavity is formed by the earth's magnetic field, as shown in fig.1.1, which defines the magnetosphere.[2]

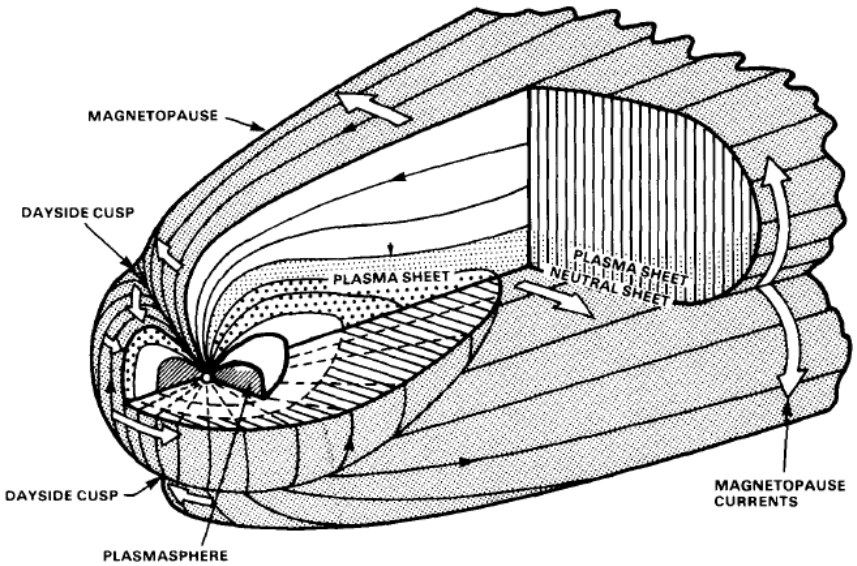

Figure 1.1: Geomagnetic cavity

The total magnetic field of the magnetosphere is due to two sources:

1. Internal field: caused by convective motion in the molten nickel-iron core of the planet, and by a residual permanent magnetism in the earth's crust;

2. External field: due to the sum-total effect of currents and fields set up in the magnetosphere by the solar wind.

Of particular importance for space applications, is the South Atlantic Anomaly (SAA)). This is primarily the result of the offset of the dipole term of the geomagnetic field by approximately 11° from the earth's axis of rotation, and displacement of about 500 km toward the Western Pacific [2]. The effect is an apparent depression of the magnetic field over the coast of Brazil where the Van Allen belts (see 1.2) reach lower altitudes, going till the atmosphere. The SAA is responsible for most of the trapped radiation received in Low Earth Orbit (LEO) (orbits used by nano-satellites). In contrast, on the opposite side of the globe, the Southeast-Asian Anomaly displays correspondingly stronger field values, and the trapped particle belts are located at higher altitudes [2].

1.2 Van Allen radiation belts

The Van Allen Radiation belts were discovered in 1958 by a group of United States scientists under the direction of Dr. James Van Allen so from here their names. The belts are two zones encircling the earth in which a relatively large numbers of high-energy charged particles is present; mainly protons and electrons trapped within the belts by the earth's magnetic field.

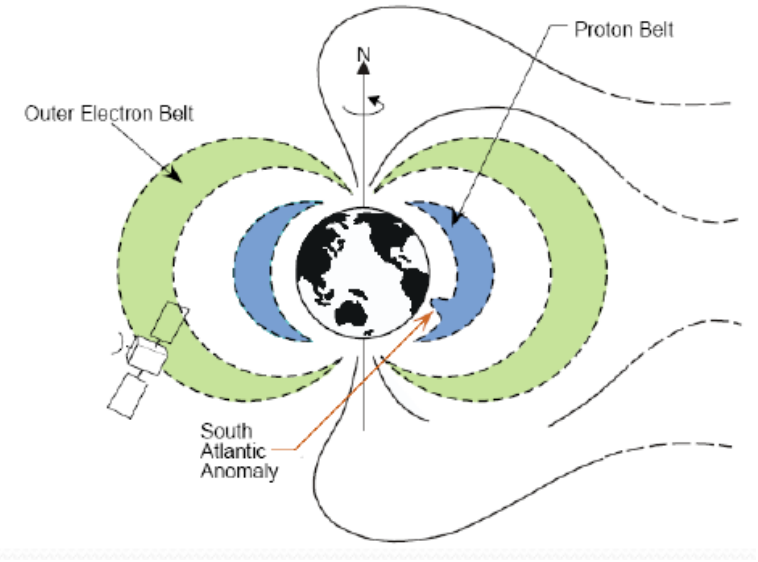

Figure 1.2: Van Allen Radiation Belts

1.3 Trapped radiation domains

Above the dense atmosphere, the earth's magnetic field is populated with trapped electrons, protons, and small amounts of low energy heavy ions [2]. These particles follow and are scattered by the magnetic field lines. Fig.1.3 illustrates the spiral, bounce, and drift motion of the trapped particles.

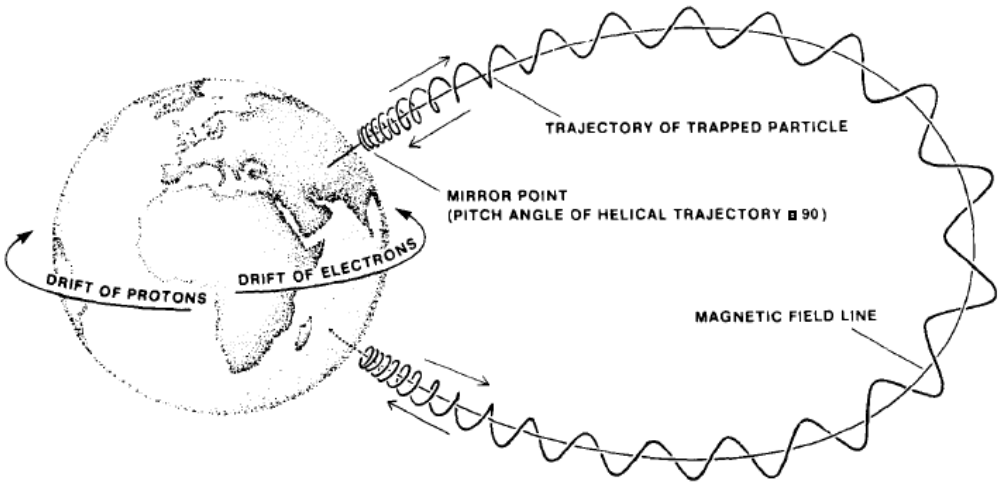

Figure 1.3: Motion of trapped particles

The magnetosphere can be divided into five domains for particle species populating

or visiting, as shown in fig. 1.4:

1. Solar flare protons;
2. Trapped protons;
3. Outer zone electrons;
4. Inner zone electrons;

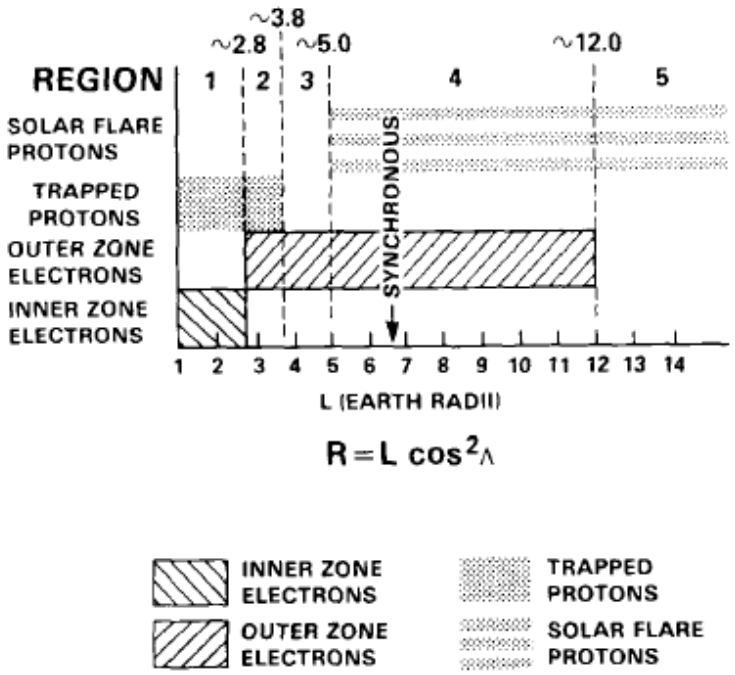

Figure 1.4: Charged particles distribution in the magnetosphere

The strong dependence of trapped particle flux can be expressed by means of the Mcllwain L parameter, defined as a dimensionless ratio of the earth's radius, approximately equal to the geocentric distance of a field line in the geomagnetic equator. The domains listed above can so be mapped using the so called dipole field equation:

$$R = L \cdot cos^2 \Lambda$$

Where R is defined as the radial distance while Λ as the invariant latitude.

1.4 Radiation belt effects on spacecraft and personnel

Energetic particles of about 1 Me will steadily degrade electronics, optics, solar panels, and other critical systems because are able to break chemical bonds, disrupt crystalline and molecular structures, and cause localized charge effects. Spacecraft systems operating in Earth orbit must be hardened to withstand this radiation environment, and typically their electronics must be designed with several layers of redundancy, incurring significant expense and additional mass (this is a problem for nano-satellites where the mass is a critical design parameter). The radiation particles also pose a significant threat to personnel and other biological systems in Earth orbit since they are able to pass through tissue ans so the can ionize water and proteins,leading to cellular damage.

1.5 Emerging radiation

In interacting with spacecraft materials, the electrons and protons of the trapped radiation belts are modified in intensity by shielding, and in character through the production of secondary radiation [2]. The most significant secondary radiation known so far, is the bremsstrahlung (aka braking radiation), produced by the fact that electrons decelerate when the contact the spacecraft surface. The bremsstrahlung intensity depends linearly on the atomic number of the spacecraft material and on the square of the initial electron energy [2]. Bremsstrahlung is very penetrating, and thus difficult to attenuate using standard spacecraft materials (like aluminum to reduce the mass).

1.5.1 Electrons and Bremsstrahlung

Fig. 1.5 and fig. 1.6 show the emerging electron and bremsstrahlung spectra behind spherical aluminum shielding for the incident environment of a 500 km circular orbit of 60° inclination [2]. As can be easily seen, the trapped electrons are very effectively attenuated and, increasing the aluminum thickness, even at the highest electron energies are attenuated. The bremsstrahlung flux levels for energies above 40 keV are not significantly affected by any of the aluminum shields because like we previously said

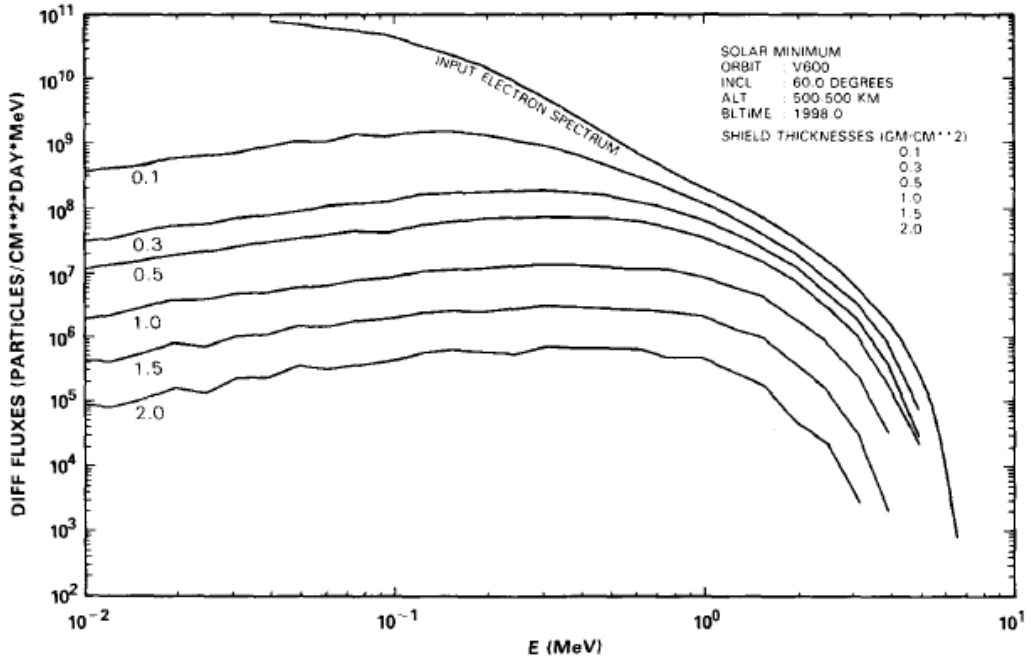

Figure 1.5: Emerging electrons spectra behind spherical aluminum shield

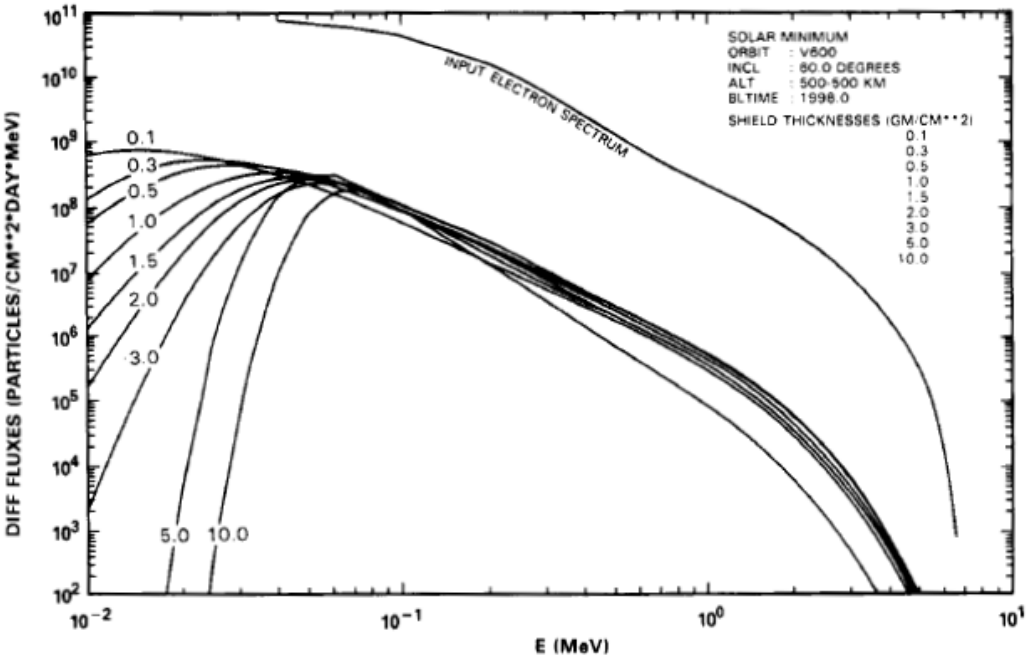

Figure 1.6: Energing bremsstrahlung behind spherical aluminum shield

they are very penetrating.

1.5.2 Trapped protons

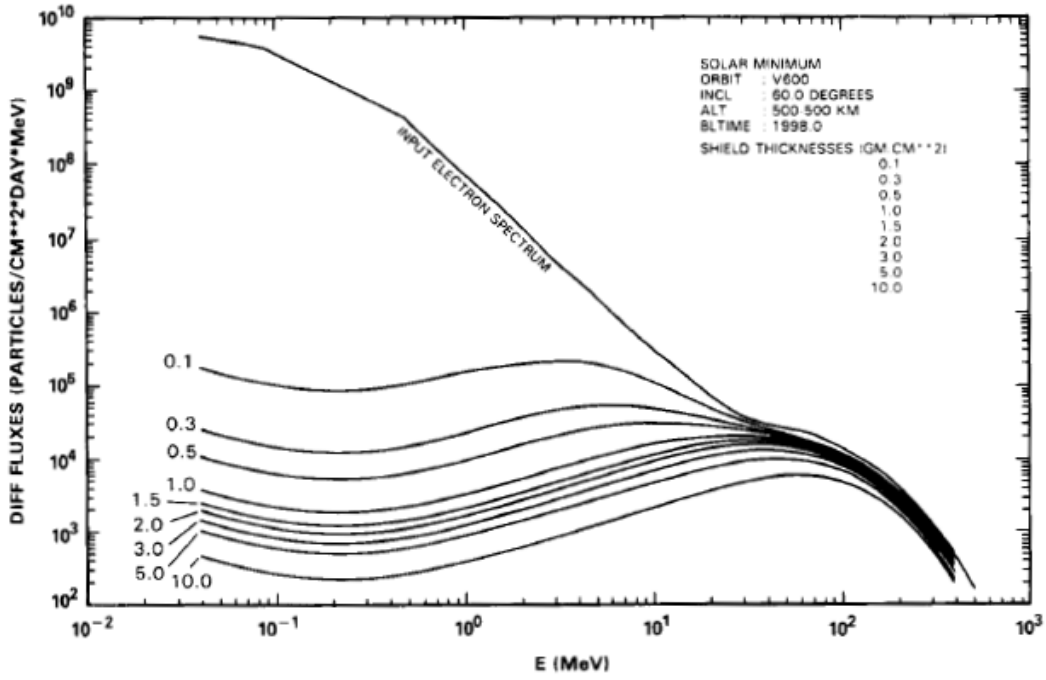

Figure 1.7: Emerging trapped proton spectra behind spherical aluminum shield

Fig. 1.7 shows the emerging proton spectra behind spherical aluminum shields for the 500 km circular, 60° inclination orbit [2]. As shown, the aluminum shielding is very effective for the low energy protons while absolutely ineffective for the high energy protons. The shielding ("hardening") of the proton spectra provides little help in reducing potential proton-induced Single Event Upset (SEU).

1.5.3 Variables affecting dose evaluation

Obtaining estimates of the dose on a given component of the internal electronics in a spacecraft is a complex process involving several variables that directly affect the results. These variables include:

- Primary environment definition;
- Description of the input spectra;
- Contributions from secondary particles and photons.

Four areas stand out that are of particular concern to shielding and transport evaluations. These are completely independent from, and unrelated to, the definition of the spacecraft-encountered radiation environment:

1. Shield geometry and shielding analysis technique;
2. Sshield material composition;
3. Target (i.e., component) composition (e.g., package, passivation, metalization and semiconductor of a complex microcircuit);
4. Dose units.

Fig. 1.8 better illustrates what is said above.

The energy deposition in the internal electronics can be measured in units of rads (material). A radiation absorbed dose (rad) is defined as 100 ergs of energy deposition per gram of absorber material, without reference to the nature of the energy deposition [2].

Generally, space radiation transport and dose calculations use idealized aluminum shielding configurations (i.e solid spheres, semi-infinite slabs etc.). This approximation allows parametric analysis of dose attenuation, exploration of the consequences of environmental uncertainties, and identification of the shielding required for a given spacecraft.

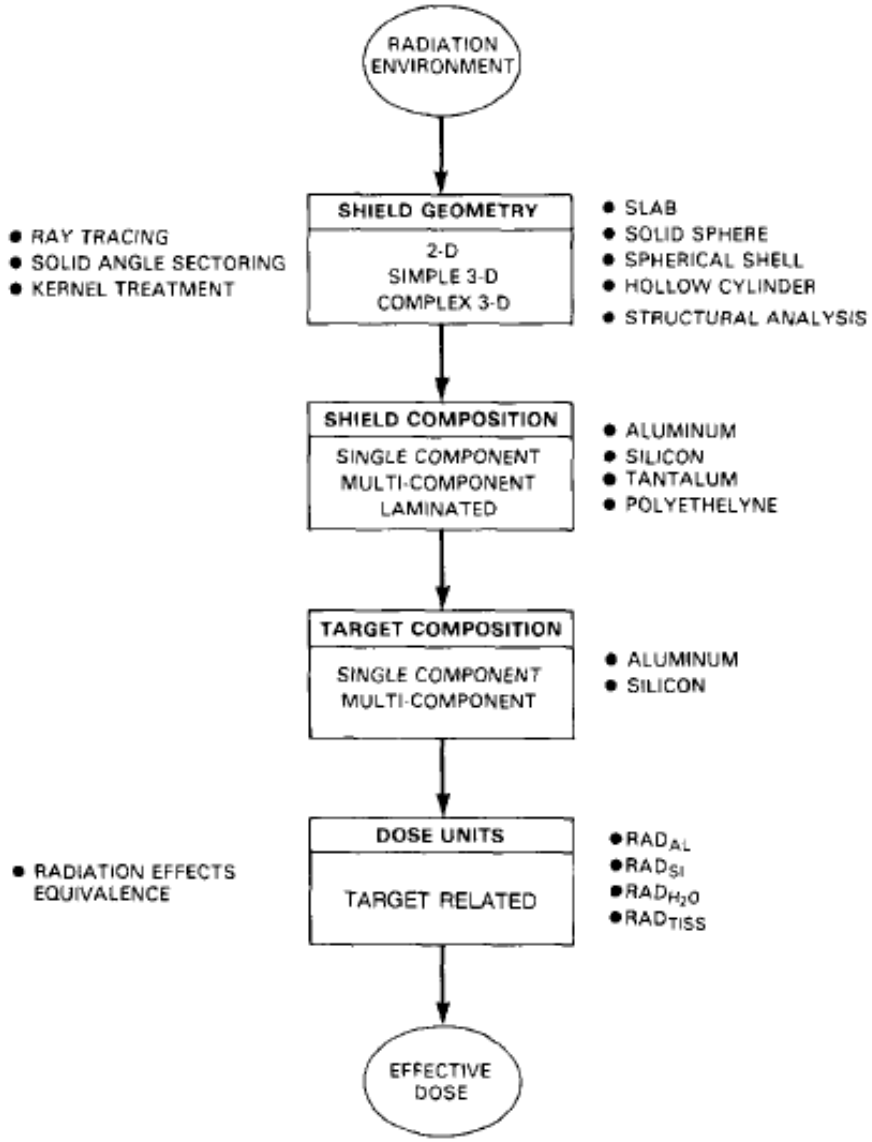

Figure 1.8: Shielding areas

1.6 Permanent damage susceptibility

The basic permanent damage mechanisms in semiconductor devices due to exposition to high-energy electrons and protons is a atomic displacement. Failure levels resulting from proton-induced displacement damage can be as low as $1^{10}p/cm^2$ for circuits (especially analog one) who are very sensitive.

1.7 Single event susceptibility

The high energy protons of the trapped space radiation environment can cause single event effects in modern semiconductor electronics [2]. It results that for electronics with "typical" shielding (about 2mm of aluminum), the single event upset rate could be as high as 0.1 upsets l bit/day for very susceptible circuits, but it decreases drastically for less susceptible one.

1.8 Transiting radiation

The transiting radiation in the space environment is composed by a solar contribution and a galactic contribution. Each is composed of high energy protons and heavy ions. In terms of the spacecraft electronics, the dominant effects are those associated with the ionization tracks of single particles, as well as the effects of total accumulated ionization [2].

1.8.1 Solar cosmic rays

This typer of rays can be divided in two macro categories:

1. Solar cosmic protons: disturbed regions on the sun sporadically emit bursts of energetic charged particles into interplanetary space (called Solar Energetic Particle (SEP)). The emission of protons from the SEP event can last as long as several days.

   SEP event cen be furthermore divided in ordinary (OR) events and anomalously-large (AL) events (quite rare). For spacecraft of mission durations greater than one year, OR event fluences are not significant, because probability theory predicts the occurrence of at least one AL event, even for a confidence level as low as 80%.

2. Solar Heavy Ions: during major solar events, the abundance of some heavy ions may increase rapidly by three or four orders of magnitude above the standard galactic background, for periods of several hours to days leading to an increasing, in therms of frequency, of single event effects within the spacecraft electronics.

1.8.2 Galactic cosmic rays

The region outside the solar system in the outer part of the galaxy is believed to be filled uniformly with cosmic rays. These consist of about 85% protons, about 14% alpha particles, and about 1% heavier nuclei. The galactic cosmic rays range in energy to above 10 GeV per nucleon [2].

In fig.1.9 the spectral distributions for hydrogen, helium, carbon, and oxygen ions is shown.

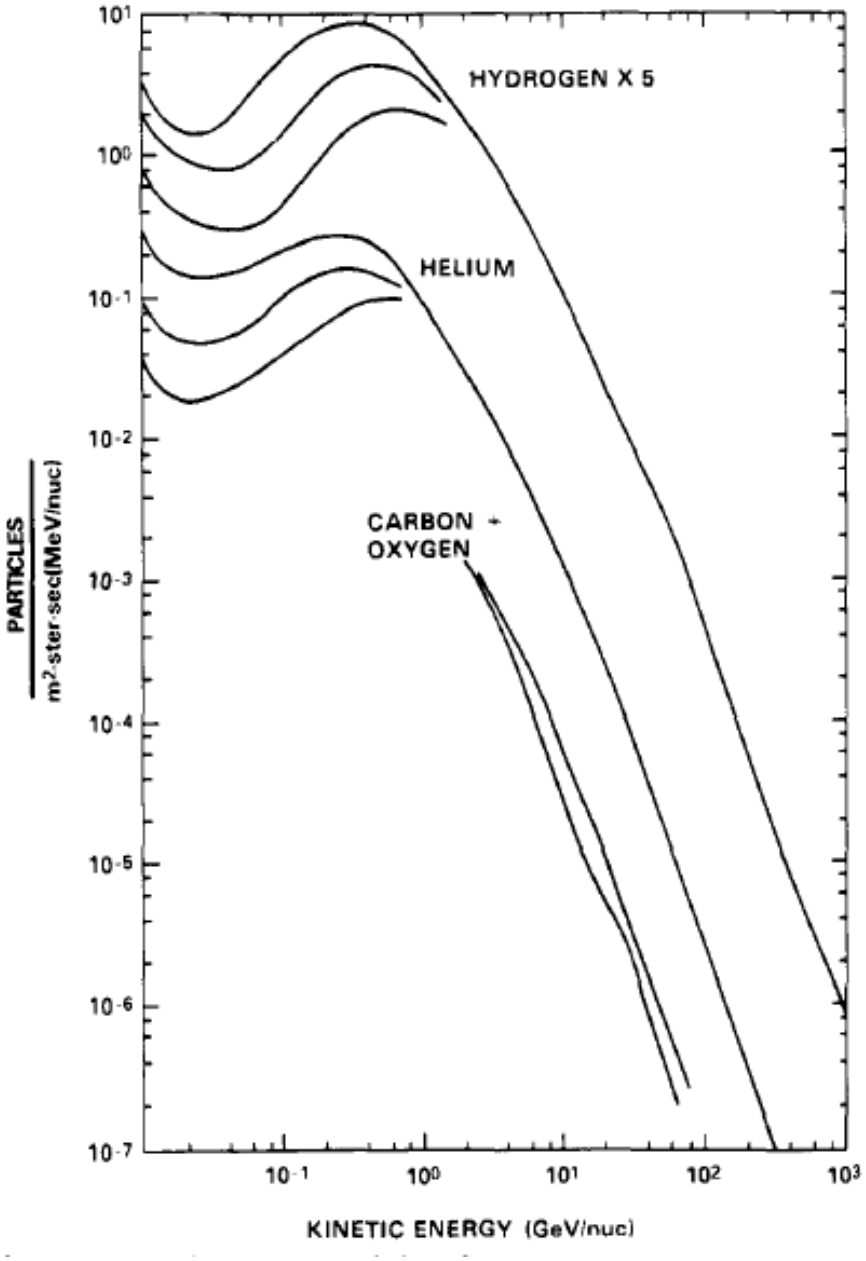

Figure 1.9: Cosmic ray spectral distribution

1.8.3 Geomagnetic shielding

Geomagnetic shielding effects on geocentric missions are usually evaluated with simple rigidity considerations, for economy reasons, and because of substantial diurnal variations in the cutoff latitudes associated with geomagnetic tail effects), and storm-induced changes [2].

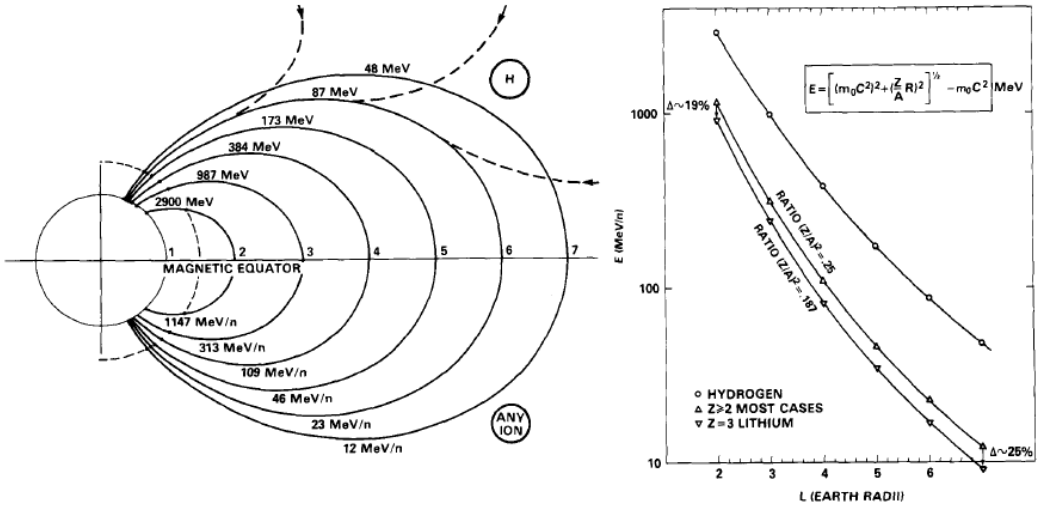

Figure 1.10: Total energy required to penetrate the magnetosphere

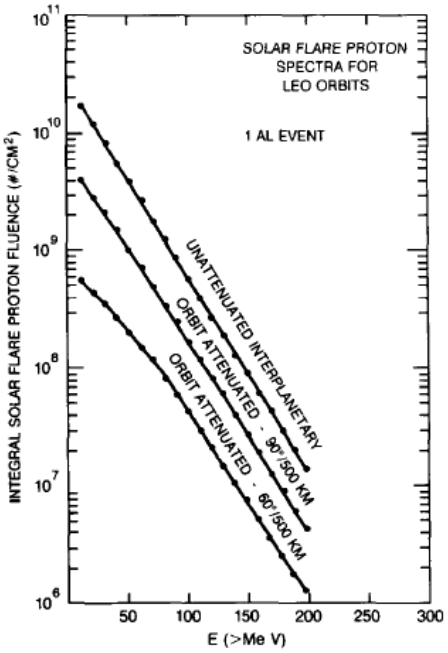

Figure 1.11: Magnetospheric attenuation of solar flare protons for LEO

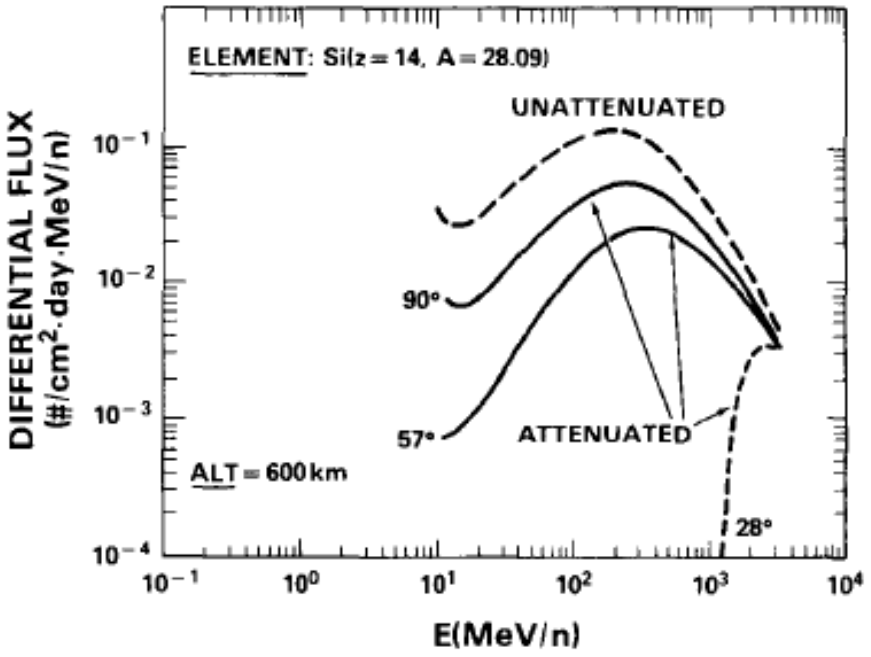

Figure 1.12: Magnetospheric attenuation of cosmic rays for LEO

1.9 Transiting radiation transport, shielding and energy deposition

1.9.1 Emerging radiation spectra

1. Solar Flare Protons: particularly relevant to single particle event effects in the electronics is the Linear Energy Transfer (LET) in silicon, defined as the energy deposition per unit length in the active region of the semiconductor device. The LET spectrum for one AL event is shown in fig.1.13.

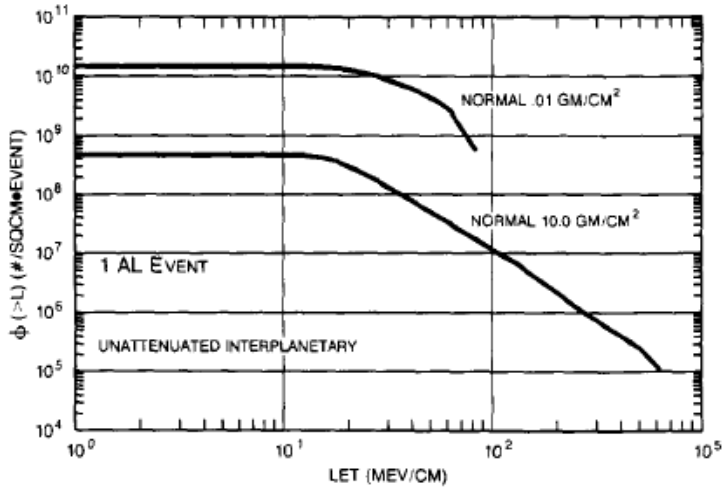

Figure 1.13: Integral LET spectra for solar flare protons behind spherical aluminum shields

2. Galactic Cosmic Rays: fig.1.14 shows the unattenuated interplanetary spectra for silicon cosmic ray ions, the magneto-spherically attenuated orbit-integrated spectra incident on the surface of the spacecraft, and the shielded spectra of emerging particles behind selected thicknesses of spherical aluminum geometries for an orbit of 57° inclination and 600 km altitude [2]. The LET spectrum is important in defining the energy deposited by a single particle, and subsequent single event effects in the spacecraft electronics.

Heavy ions who pass through shielding material are responsible of nuclear reactions that provide a source of secondary radiation, both prompt and delayed.

Several important features are illustrated by the curve in fig.1.15.

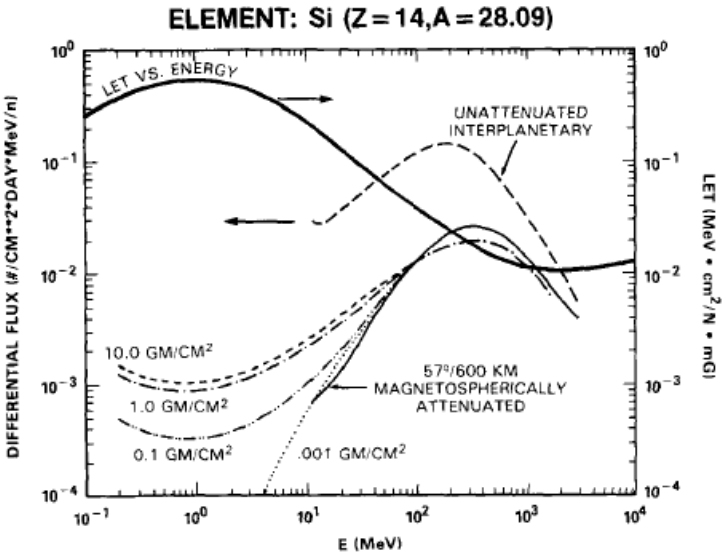

Figure 1.14: Galactic cosmic ray spectra (solar min.) emerging behind spherical aluminum shields

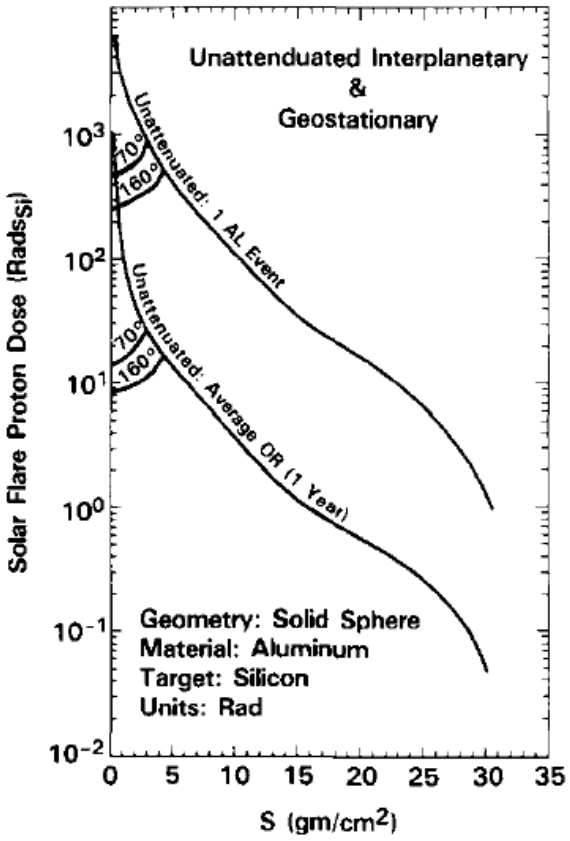

Figure 1.15: Solar flare proton dose

- First, there is substantial attenuation by the earth's magnetic field of all particles in the energy range of 10-10 000 MeV per nucleon;
- Second, there is an insignificant effect of material shielding in the energy range from about 90 to 10 000 MeV;

It is very important to note that there is no substantial decrease in flux even for aluminum shielding of $10 grams/cm^2$ (approximately 1.5 inches). Increasing shield thicknesses, the population of high energy ions decreases slightly, but with a resultant increase in the low energy (0.8-50 MeV/nuclear) ions. Since the LET increases with decreasing energy in this range (heavy solid curve) the presence of the shield actually increases the severity of the environment to the internal electronics.

1.9.2 Ionizing radiation dose

In general, the ionizing radiation dose from the transiting radiation environment is not significant compared to that of the trapped radiation environment [2]. Particle fluxes from energetic solar flares are heavily attenuated by the geomagnetic field, which prevents their penetration to low orbital altitudes and inclinations but in Geostationary Earth Orbit (GEO), the geomagnetic shielding is practically ineffective.

1.9.3 Single event susceptibility of electronics

Single event upset effects in electronics from the transiting space radiation environment may be the result of either the energetic solar flare protons or cosmic rays. In general, the single event upset rate due to transiting protons is small compared to that due to cosmic rays, except for the occurrence of an AL. To cover the occurrence of an AL during the spacecraft mission, both the expected duration and fluence of the AL must be considered in the electronics design. For the cosmic ray component of the transiting space radiation environment, the definition of the LET spectrum of the internal radiation environment is a fundamental basis for characterization of component susceptibility. Observed effects from single heavy high energy ions include memory bit upset, microprocessor errors, CMOS latch-up and burnout in power MOSFETs, and electrically-erasable PROMS. The probability of latch-up or burnout is much less than that of memory bit upset or logic

errors, but the consequences to system operation may be much more severe. Generally, cosmic-ray-induced single event effects dominate proton-induced single event effects both at altitudes below 1000 km and above 4000 km for 60° circular orbits. For orbits of lower inclinations, the cosmic rays are shielded by the earth's magnetic field, causing the cosmic ray upset level to decrease compared to the proton upset rate. On the other hand, for orbits of higher inclinations, the relative upset rate of the cosmic rays increases. The variations in the spacecraft orbit, space radiation environment, and device susceptibility should be considered in estimating specific cosmic ray/proton upset levels in support of spacecraft electronics design. The specification of the internal electronics environment should include the time-dependent proton flux and energy spectrum, the cosmic ray LET spectrum, and the cosmic ray spectrum by particle species and energy spectrum. The actual cosmic ray spectrum can be a valuable supplement to the LET spectrum in those cases where more detail is necessary to support experimental characterization in ground-based laboratory facilities.

1.10 Radiation effects summary

Summarizing all the informations we get the classification in fig.1.16.

Radiation Source	Particle Type	Primary Effects
Trapped radiation belts	Electrons Protons	Ionization damage; Ionization, SEE.
Galactic Cosmic Rays	High-energy, heavy, charged particles	SEE
Solar Flares	Electrons Protons Lower-energy, charged particles	Ionization Ionization, SEE SEE

Figure 1.16: Summary of radiation effects

CHAPTER 2

FRAM technology and space applications

2.1 Introduction to FRAM technology

2.1.1 The FeRAM cell

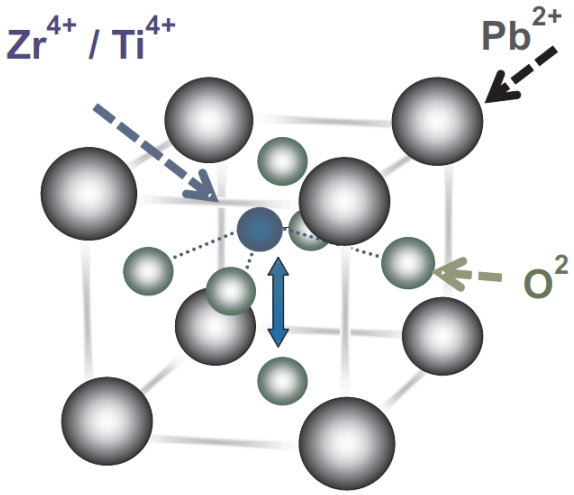

Figure 2.1: PZT FeRAM cell

The idea of using ferroelectric materials to store digital data date is dated 1952. However it was practically implemented only starting from the 80s. The ferroelectric RAM cell, known as FeRAM of Ferroelectric Random Access Memory (FRAM), can be conceptually associated to the DRAM cell, in that a single capacitor stores one bit of information and the cell is connected to a memory column via a single pass transistor. The big difference lies in the dielectric of the storage capacitor. The DRAM cells in fact,

use a layer of standard linear material, while the dielectric of a FeRAM cell is made of ferroelectric material, usually lead (Pb) Zirconate Titanate (PZT).

The use of this ferroelectric dielectric makes the cell behave very differently from a DRAM cell. On one side, the dielectric constant of ferroelectric materials is very high, this is the reason why it is possible larger capacitors using a smaller space can be made. Furthermore, it points out that the material exhibits two stable polarization conditions, it is so possible to switch between them by means of an electric field with opposite polarity. Since the polarization will be kept after the applied field is removed, it is possible to link the polarization state to a logic state and that will be maintained also in absence of power supply, meaning that the FRAM cell is a non volatile and that no refresh is necessary to keep the information in the memory. We have to underline that in a DRAM cell the capacitor has one of the electrodes grounded; in the FRAM cell the corresponding electrode is connected to a so called drive-line.

Like for DRAMs, in a FRAM cell, the reading process is destructive. This means that it is not possible to read the content of a cell without actually clearing it. This is de to the way the information is stored in the device. To know which of the possible polarization states the dielectric holds, the only way consists in writing a new value to the cell with the bit-line pre-charged but in high impedance state. Depending on the previous polarization, this process will or won't produce a voltage pulse out of the bit-line resulting so in a reading. We can so easily understand that read and write cycles require basically the same operations and can both be completed in times if the order of tens of nanoseconds.

The first manufacturer who introduced FRAM technology in micro-controllers is Texas Instruments (TI). The key features of these type of micro-controller are described in 2.1.2.

2.1.2 FRAM overview in MSP430FRxx devices

TI offer a family of micro-controllers (MSP430FRxxxx) that uses the FRAM as storing element instead of the usual FLASH memory. Some of the key attributes of this family are:

- The embedded FRAM on MSP430 devices can be accessed (read or write) at up to

a maximum speed of 8 MHz. Above 8 MHz, wait states are used when accessing it;

- Writing and reading the FRAM requires no setup or preparation such as pre-erase before write or unlocking of control registers (unless the Memory Protection Unit (MPU) is used to protect the FRAM against write access);
- FRAM is not segmented and each bit is individually erasable, writable, and addressable;
- FRAM does not require an erase before a write (like in FLASH memories);
- FRAM write accesses are low power, because writing to FRAM does not require a charge pump;
- FRAM writes can be performed across the full supply voltage range of the device;
- FRAM write speeds can reach up to 8 MBps with a typical write speed of approximately 2 MBps. The high speed of writes is inherent to the technology and is aided by the elimination of the erase bottleneck that is prevalent in other non-volatile memory technologies. In comparison, typical flash write speed including the erase time is approximately 14 kBps [7].

2.1.3 Writing to FRAM

Storing a 1 or 0 (writing to FRAM) requires polarizing the crystal in a specific direction using an electric field. This makes FRAM very fast, easy to write to, and capable of meeting high endurance requirements [7].

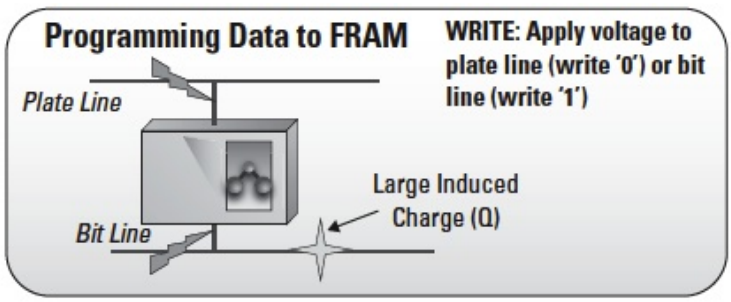

Figure 2.2: Write process in FRAM cell

2.1.4 Reading from FRAM

Reading from FRAM requires applying an electric field across the crystal similar to a write. Depending on the state of the crystal, it may (or may not) need to be re-polarized, thereby emitting a large or small induced charge. This charge is compared to a known reference to estimate the state of the crystal. In the process of reading the data, the crystal that is polarized in the direction of the applied field loses its current state. Therefore, every read is accompanied by a write-back to restore the state of the memory location [7].

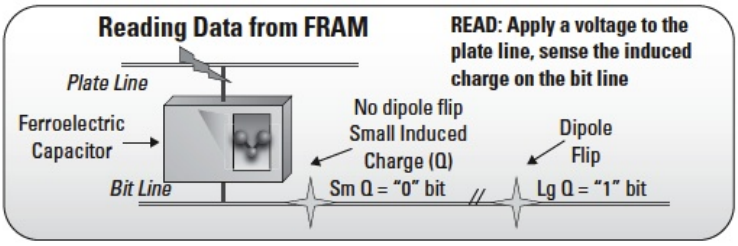

Figure 2.3: Read process in FRAM cell

2.1.5 Data retention

When data retention tests are executed, the primary goal is to ensure a ten year lifetime at $85°C$. To ensure this specification in FRAM two mechanisms regarding retention reliability must be considered.

Thermal Depolarization

The integrity of the information stored in an FRAM cell is directly proportional to the amount of polarization that the cell is capable to maintain. When exposed to high temperatures, FRAM cells loose their ability to stay polarized and are unable to store information for as long as the high-temperature condition last. This phenomenon is called thermal depolarization and it is referred to a reduction of the spontaneous polarization that occurs as the ambient temperature of the ferroelectric material increases towards the Curie temperature [1]. In the case of the specific composition of material

[1]In physics and materials science, the Curie temperature (T_C), or Curie point, is the temperature where a material's permanent magnetism changes to induced magnetism. The force of magnetism is

used in the MSP430FRxxxx devices, the Curie or transition temperature at which the FRAM cell is completely depolarized is approximately $430°C$. A qualitative graph that explains this phenomenon is reported in fig.2.4.

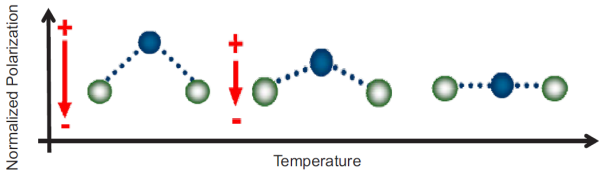

Figure 2.4: Reduction of polarization with temperature in a FRAM Cell

Imprint

Ferroelectric memories experience an effect in which data in one logic state can strengthen when the memory cell is exposed to high temperatures over long periods. This effect (the stabilization of polarization into a particular state) is known as imprint. However, imprinting also weakens the ability of the FRAM cell to store the complementary state data [7].

Unlike thermal depolarization (see 2.1.5), imprinting is permanent and is not lessened with a reduction in temperature. Note that while complete thermal depolarization occurs at the transition temperature of $430°C$ (FRAM Curie Temperature), imprinting occurs at lower temperatures, such as when data is written to FRAM and the material is subjected to a high-temperature bake for a long duration.

2.1.6 Magnetic fields

A common misconception is that ferroelectric crystals contain iron or are ferromagnetic or have similar properties. The term "ferroelectric" refers to similarity of the graph of charge plotted as a function of voltage to the hysteresis loop (BH curve) of ferromagnetic materials. Ferroelectric materials are not affected by magnetic fields.

determined by magnetic moments.

2.1.7 Electric fields

The FRAM memory cell operates by applying a switched voltage to sense and restore the data state. The ferroelectric film PZT is about $70nm$ thick. If the device is placed in a $50kV$ field at $1cm$, it is not possible to produce more than $1V$ across the ferroelectric film. As a practical matter, FRAM devices are impervious to external electric fields.

2.2 COTS, nano-satellites and space applications

As we already said and shown (see 1), when selecting components for space missions, the key issue is the reliability. Electronic systems designed for spacecrafts are normally built using space qualified components which are devices undergo special treatment to conform to specs identical or similar to military standards.

These devices have a very high costs, unreachable for regular commercial or scientific space missions. This is the reason why from national or international space agencies have budgets allowing the designers to work only with space qualified Commercial Off The Shelf (COTS) components. If COTS are used there are some critical point to take into account:

- radiation: at ground level, the atmosphere constitutes an effective shield to incoming space radiations. Outside the atmosphere the radiations levels are much higher and impose severe limitations on electronics (see 1).

- pressure: no atmosphere is present in orbit. This fact creates two main consequences: pressure is very low and power dissipation through convection is impossible. The low pressure limits the use of devices with liquid components (like electrolytic capacitors) and it is necessary to check that the packages of electronic components do not emit dangerous gases and do not break during depressurization phase, so out-gassing and off-gassing tests are necessary [1].

- temperature: even if outside temperature can be extreme in light and darkness, inside small satellites it can be demonstrated that the temperature isn't a big issue because it remains in the range $-10°C$ to $20°C$, so normal devices rated for automotive use are well suited for operation inside a satellite [1].

- vibration: heavy vibrations are normal during the launch phase of the mission. Again, automotive devices are normally designed to sustain this vibration level [1].

We can for sure state that, however, the main concern in using COTS is the space environment because the others specs are reasonably met. Summarizing what we said before, radiations in space come from different sources:

- The Sun is the main emitting body to be considered;
- Background cosmic rays;
- The electromagnetic field of the Earth plays a significant role in shielding incoming particles so the radiation level will be different depending on the orbital parameters of the spacecraft.

The damages produced by such a kind of radiations, can be divided in two macro categories:

1. Cumulative effects of the dose received, known as Total Ionizing Dose (TID);
2. Effects of a single particle hitting the device Single Event Effect (SEE);

In particular for MOSFET devices the main problem comes from a gradual shift in the threshold voltage. Above a certain TID (measured in $krad(Si)$), its shift is so high that the device cannot switch anymore, causing a functional failure of the circuit [1].

SEE create several failure depending on various parameters (i.e device, technology), but the particles responsible for SEE are mainly ions and protons. When an high energy particle hits the surface of a silicon chip, part of its energy is transferred to the chip as electric charge. The amount of energy transferred is called LET measured in $MeV \cdot cm^2 \cdot mg^{-1}$.

The main effects are:

- SEU: recoverable error that appears mostly in memory devices. The high energy particle hits the sensible area of the memory (i.e capacitor in a DRAM) and transferring an amount of charge sufficient to alter the stored value. This damage is called soft error, meaning that it isn't permanent and it is sufficient to rewrite the memory to restore correct behavior;

- Single Event Latch Up (SEL): potentially destructive error typical of CMOS logic because a parasitic Silicon Controlled Rectifier (SCR) is present. If the particles has enough energy to put theSCR in conduction, it remains ON until the device is switched OFF. When this device is ON, it creates a low resistance path between power supply and ground. The current can be very high, creating an hot-spot that can turn in a permanent damage;

- Single Event Functional Interrupt (SEFI): erratic behavior of a complex circuit due to the consequences of the impact of a single particle. It is similar to SEU, but the affected area, instead of being a simple memory cell, is a Finite State Machine (FSM) or other sequential circuit which is forced into an unwanted state by the event. The error may persist until the next reset or may be recovered at some time. In the case of a memory device, a SEFI occurring in the control part of the device can lead to reprogramming a big area of the matrix (typically a whole row) [1];

- Single Event Gate Rupture (SEGR) and Single Event Burnout (SEB): these damages occur when a particle hits the active area of a power MOSFET transistor under certain bias conditions, creating a physical damage, such as oxide breakdown for SEGR, overheating due to large currents for SEB, that prevents normal operation of the device. Low power devices such as memories are not subjected to SEB, but SEGR has been reported if a particle hits an Electrically Erasable Read Only Memory (EEPROM) or a Flash memory during the erase procedure, due to the relatively high voltages used during such operation [1].

Testing one device for SEE typically means exposing it to a flux of heavy ions, characterized by a specific LET, for a specified amount of time and repeated for different LETs. As can be easily imagined, this process is very expansive because this kind of tests can only be performed in cyclotron. Alternative lower cost methods include the of laser pulses or a small radioactive sources based on Californium 252 which emits ions of different LET, but in this case it is difficult to relate test results to more rigorous cyclotron methods. Once a device is characterized for TID and SEE behavior, knowing the expected radiation environment at the programmed orbit, it is possible to predict which errors can be expected during operation and what is the relevant error rate [1].

2.2.1 Advantages of using FRAM in space applications for nano-satellites

The three main design parameters of the electronic systems of small satellites are:

- power consumption;
- physical dimensions;
- radiation environment behavior.

The electric power in the satellite comes from solar panels, which are necessarily of small dimensions, leading to few watts of average power to cover all the needed functions, so it is necessary to make the best use of any mW of available power. Launch costs are directly proportional to the mass of the system, so it is mandatory to reduce as much as possible dimensions and mass of the electronic systems [1].

As previously stated, FRAM memories are RAM devices, meaning that read and write procedures do not differ significantly and random write is possible without the need of a previous erase of the cell (like for example in FLASH devices), but they are also non volatile, so that information is not lost after removing power supply.

We can therefore compare FRAMs with RAMs and FLASH devices. As we already observed, the structure of the FRAM cell is very similar to the DRAM one, it doesn't require the refresh procedure, leading to a save of power. In fact in DRAM devices most of the power is used by the refresh procedure. FRAM cells are bigger than DRAM one, so it is not possible to use FRAM memories to store huge amount of data at the moment. In any case, memory requirements of small satellites are normally compatible with the size of available FRAM devices, except for imaging payloads if local storage of a certain number of images is mandatory.

Since the FRAM stores the information polarizing a PZT film, an alpha hit is very unlikely to cause a change of the polarization. FRAM terrestrial SEU is not even measurable. This "radiation resistant" characteristic of FRAM makes it attractive for use in several emerging medical applications and space one [1].

Comparison between FRAM, FLASH and EEPROM

Let's now make some comparison between FRAM, FLASH and EEPROM.

- Read operations in FRAM and FLASH devices are equivalent both in speed and power requirements;
- Write operations in a FLASH memory are quite complex:
  1. First phase is a page erase, which takes a time in the order of tens of milliseconds;
  2. Second write operation of the new values. Even to rewrite a byte, one full page has to be erased and the unmodified cells have to be rewritten in place.

  The erase procedure requires a high supply voltage (negative for erase, positive for write) which is internally generated by the device using a charge pump circuit. EEPROM devices can be reprogrammed on a single byte basis, speeding up the write process when a single random byte has to be altered, but the need for high voltages is the same as for Flash devices and the operation can be completed in tens of microseconds [1];
- The number of write cycles that can be sustained by a Flash or EEPROM device is in the order of 10^4 to 10^5, while the FRAM memories can be written more than 10^{12} times, with 10^{16} cycles being claimed by for example TI.
- Several tests performed on the FRAM cells show that the SEU response is very good, definitely better than the one of most Flash or DRAM devices, indicating that this technology is very appealing for space applications.

As a summary, FRAM devices are attractive for use in small spacecrafts as a replacement for both RAM and Flash memories when the size of the memory is small, because of the ease of use, non volatility (when compared to RAM), the low power requirements, the speed advantage in the write process and the endurance, when comparing with Flash memories [1].

2.2.2 Possible hardening solutions

The problems highlighted in the previous sections, due to the ionization problems, can be overcome in two different ways.

1. The first is to design a FRAM memory specifically for space applications (very expensive solution);

2. The second is to use COTS devices taking specific system design measures to prevent the failures due to the CMOS logic surrounding the memory array (cheaper solution but whose requires hardware redundancy).

Obviously the first solution is preferable, but it involves high development and production costs and time, so it is not affordable when designing low cost spacecrafts such as university satellites. Going a little bit into details, create a rad-hard version of a FRAM memory means use a rad-hard CMOS process to build row and column decoders and the read/write control logic. This type of processes normally use Silicon On Insulator (SOI) or Silicon On Sapphire (SOS) techniques. As an alternative, the addition of an epitaxial layer to the substrate of a standard CMOS process can lead to improved performances, at least about SEL sensitivity. Since it is not difficult to add a ferroelectric layer to a rad-hard CMOS process, this way is feasible, but the associated costs are very high [1].

The alternative of using COTS devices is very attractive and is feasible in the case of FRAM memories because of the characteristics of these devices (see 2.2.1). The risks of data corruption or physical damage come from the standard CMOS logic surrounding the memory array.

To prevent the risk of loss of the device due to latch-up, it is possible to monitor the supply current and to switch off the chip in case of overload, indicating SEL occurrence. This procedure has to be done very carefully in CMOS logic, because it is not normally sufficient to remove power supply to be sure to reset the parasitic structure responsible for latch-up. The structure of the input circuitry of CMOS devices always includes clamp diodes, so even removing power supply it is possible to continue to supply the chip via inputs at logic high state [1].

Soft errors are the second problem to address. The non volatility of the information stored in the FRAM is a great help in this respect. Soft errors can only occur when the memory is powered, but our devices need power supply only when it is necessary to read or write information, not to maintain internal data. This suggests a strategy for SEU and SEFI effects mitigation: the device is powered only during read or write operations, switched off otherwise. This strategy is possible only if the memory stores

data which are to be seldom read or written, not if the device is used to store the active CPU program.

A second strategy for important data, such as the backup copy of processor program can be achieved separate copies on multiple devices, furthermore the data are associated with strong error detection Cyclic Redundancy Check (CRC) codes, so that it is possible to detect if what is stored in a device was corrupted by SEU/SEFI. Corrupted data are regenerated from the other copies so the system integrity can be guaranteed.

CHAPTER 3

The UML approach

Let's now introduce an approach to Unified Modelling Language (UML) as a high level specification, description and documentation language. The purpose of this approach is to obtain a complete development flow for mixed-systems able to produce, on one side, documentation always close to real project implementation and, on the other, a fast and reliable method for reducing time- to-market in developing these objects. The design of all the major subsystems of AraMiS has been carried out by using UML. Initially developed in 1995 for designing software, the UML was optimally adapted to the description of systems made of both hardware and software. It is based on the representation of entities involved in the system functioning and all interactions among them. There are many advantages of using this language with most important being:

- Make easier the project understanding, even by people external to the project, thanks to a graphical/conceptual representation of the elements that make the system (components, subsystems, signals, functions...) starting from a high level description to a specific one;
- Simplify and improve the description of system functionalities and the specification definition providing a common basis in the approach of designing the units forming the whole system;
- Make exportable the system building blocks (which are independent form each other) such that they can be re-used in other projects so implementing the modularity concept.

Among all possible utilities that UML offers, there are certain types of diagrams used in UML including, but not limited to use case diagram, class diagram and sequence diagram.

3.1 UML diagrams

3.1.1 Use case diagram

Use case diagrams show main function of the system (use cases) and the entities that are outside the system (actors). Use case diagrams show how the class and objects of the class relate and hierarchical associations and object interaction between classes and objects. These diagrams allow us to specify the requirements of the system and show interaction between system and external actors. These diagrams are the starting point in the system modelling and consist of actors and use cases [6].

Actors

Actors are generic entities, human users, other systems or the external environment, which interact with the system under design and implements one or more use cases. They are usually shown as sketched people with a short name which identifies the role in the system. They are associated with a detailed documentation. The list of actors is fundamental to understand all entities which might interact with the system. The actors are very fundamental entities and missing an actor will miss all the interfaces and functions associated with it. Therefore they are critical in the design and the designers need to put more time in identifying the possible actors during early stages of design. Let us imagine a situation where the designer forgot to add an actor "tester" in the early stage of the design. The possible consequences may be:

- There might not be possibility of test connector to test the satellite;
- There will be no internal access node for debugging;
- No software will be added for detailed testing;
- There might not be special satellite mode to allow testing before launch, etc.

Use case

The major work of the actors is to interact with different subsystems of the satellite. The main concerning points are:

- What does and actor expect from a system?
- What does a system expect from an actor?

All this is detailed by Use Cases, which are usually described by an oval with a name which shortly describes it. Building up the list of use cases means starting to specify the functions of the satellite or its subsystem and therefore thinking to the mission. There exist several kinds of relations between use cases and actors including generalization, inclusion, extensions, associations etc. This section details the use cases of the magnetic attitude subsystem of the AraMiS architecture. The use case diagram defines the functional specifications of the project and is shown in fig.3.1 for magnetic torque actuator subsystem [6].

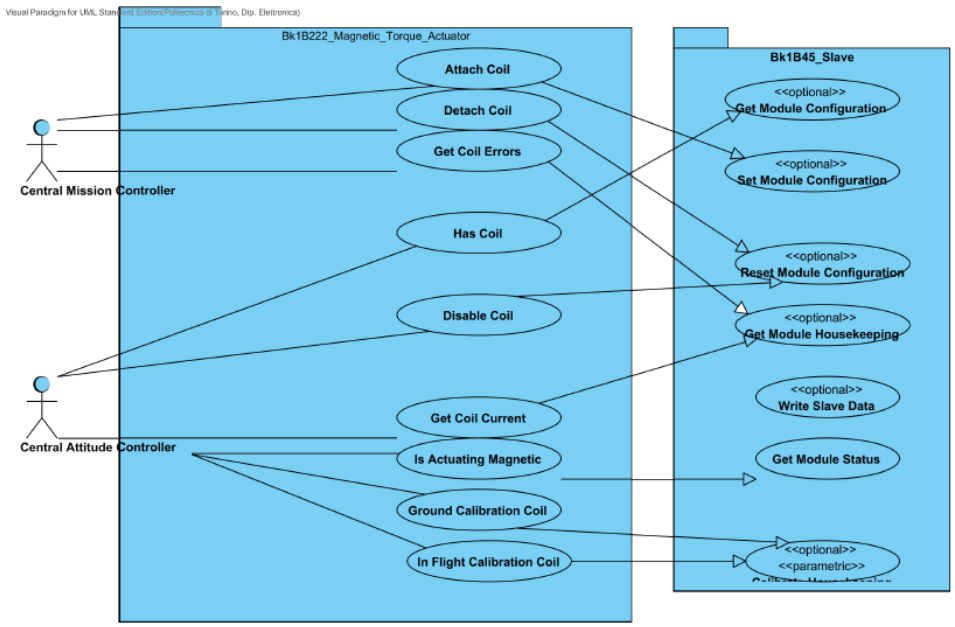

Figure 3.1: Use case example

The central mission controller and central attitude controller actors interact with the system and perform many tasks. The central mission controller is an entity (likely

a software routine running on the onboard computer) in charge of satellite supervision, fault detection and management and emergency management whereas the central attitude controller is an entity (likely a software routine running on the On Board Computer (OBC) in charge of managing the attitude control subsystem in nominal operation. Fault and emergency handling are left to the Central Mission Controller. All other use cases implement most of the magnetic actuation and control functions such as attach/ detach coil, get voltage and current levels of coil and housekeeping sensors etc [6].

3.1.2 Class diagram

The objects have tendency to know things i.e. they have attributes and they do things i.e. they have methods. All objects of the same type are represented by a class. In UML notion, classes are depicted as boxes with three sections, the top one indicates the name and stereotype of the class, the middle one lists the attributes of the class, and the bottom one lists the methods. Each object in UML classes can either be associated with hardware (HW), software (SW), an analog (ANA) implementation etc. depending on the stereotype of each class. Each stereotype is labelled with a specific colour. Each subclass has objects which contain different attributes and operations. UML classes and objects are used to specify any electronic, mechanical, software element of a system. The attributes of the class store data for the class. The attributes can either be constant, therefore representing characteristics common to all objects of that type variable, therefore storing time-variable data which are part of the class. Classes in turn can be made of one or more other classes (hierarchical structure). The class diagram of attitude and orbit control subsystem has been elaborated as a test case. The magnetic attitude control system together with the inertial attitude control system is used to accomplish the desired rotation to the satellite, by sending commands directly from the ground [6].

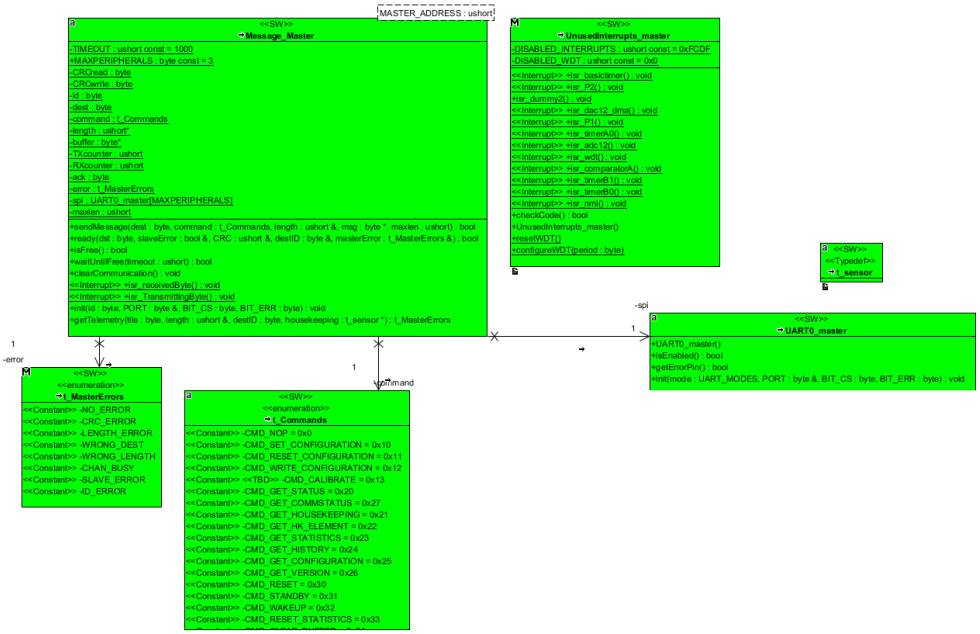

Figure 3.2: Class diagram example

3.1.3 Sequence diagram

Sequence diagrams describe how structural elements communicate with one another and time sequence of events. Time increases vertically from top to bottom. fig.3.3 shows the sequence diagram of simple protocol data exchange [6].

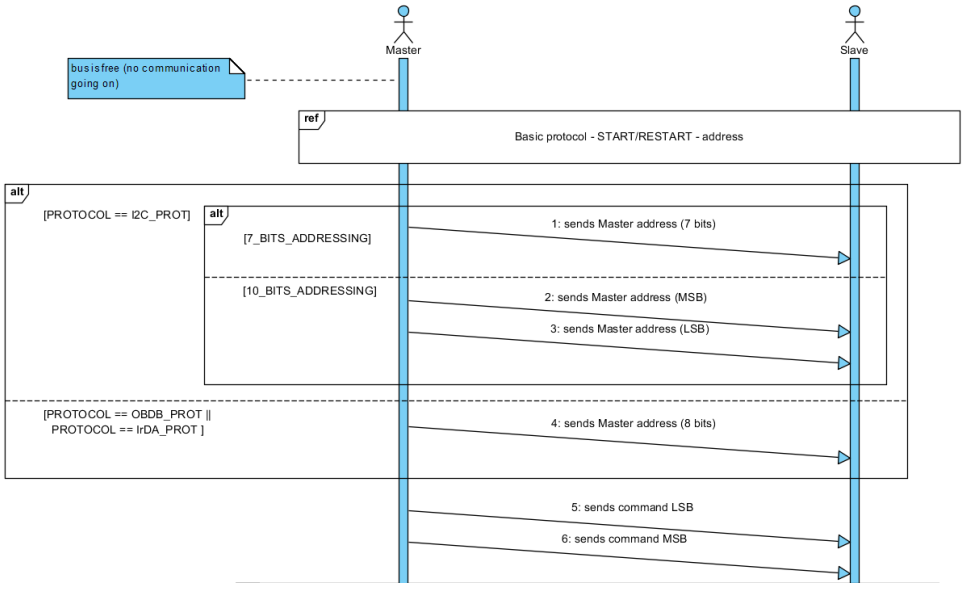

Figure 3.3: Sequence diagram example

CHAPTER 4

Small satellites and the AraMIS concept

4.1 Introduction

Small satellites market has considerably grown over the last decade. This has been made possible due to the availability of low cost launch vectors. This cost reduction has made it feasible for universities and small industries to enter the satellite market. In 2001, Professor Robert Twiggs at Stanford University, USA, in collaboration with Professor Jordi Puig-Suari at California Polytechnic State University, have defined and implemented the standard for small satellite called CubeSat. CubeSat identifies the standard for small satellite with dimensions 10×10×10 cm3 and a maximum mass of 1 kg, having a structure adapted to the launcher Pico-satellite Orbital Deplorer (POD) . Another feature of CubeSat is the use of low-cost commercial components called COTS. These features greatly help in the reduction of cost and development time. In addition, the weight reduction allows the use of less expensive launchers. Satellites based on CubeSat standard have made it possible for potential clients to buy and assemble their own satellites. This (CubeSat) standard laid the foundation for several projects of nano-satellites by many universities and SMEs throughout the world. Different universities across Europe such as University of Wurzburg in Germany, the Norwegian University of Science and Technology and Italian universities including; the University of Rome La Sapienza , the University of Trieste and Politecnico di Torino. Practically, any artificial satellite of low mass and low volume can be considered as a small satellite. However, the definition can be extended to any system designed with the small satellite philoso-

phy. This may include features such as commercial off the-shelf components, modular systems, less redundancy, open sourcing, incremental missions, etc. Over the past few years, small satellites have revolutionized the landscape of space exploration. They facilitate quicker, cost effective and reliable access to the space. This provides a potential opportunity to have smaller project groups and encourages new actor to develop their capabilities in the space domain. Small satellites are encouraging spacecrafts to test and try novel methods and technologies (e.g. open source hardware and software, formation flying), which might not be under the purview of large scale satellites This remains the reasons as to why small satellites have been considered a disruptive technology by numerous space mission experts. Small satellite programs are particularly attractive since they are "affordable". There shall be no surprises in the near future, if more and more developing countries, groups from the academic world or even small teams of space enthusiasts develop their own space mission based on small satellites. The small satellite platform is catering to new actors such as developing countries, students, and amateurs.

4.2 AraMIS concept

Modular architecture for Satellites (AraMiS) is a project that wants to take further this concept and create a true modular architecture. The design approach of AraMiS architecture is to provide low-cost and high performance space missions with dimensions larger than CubeSats. The feature of AraMiS design approach is modularity. These modules can be reused for multiple missions which helps in significant reduction of the overall budget, development and testing time. One has just to reassemble the required subsystems to achieve the targeted specific mission. Design reuse is the rationale behind the AraMiS project is to have a modular architecture based on a small number of flexible and powerful modules which can be reused as much as possible in various missions. This architecture is intended for different satellite missions, from small systems weighing from 1kg to larger missions. The fig.4.1 depicts a number of configurations that show the potential capabilities of the proposed architecture. Modularity has been implemented in different ways. From the mechanical perspective, larger satellite structures can be conveniently realized by combining several small modular structures. The modularity concept has also been intended from electronic standpoint. Most of the

internal subsystems are developed in such a manner they can be composed together to enhance performance. One such example is the power management subsystem. In conventional missions: to get maximum solar power, solar cells are mounted on all the available surfaces but their number can be different in various missions, thus requiring redesign each time this system. This new modular approach makes use of a standard module which can be replicated many times to fit mission requirements.

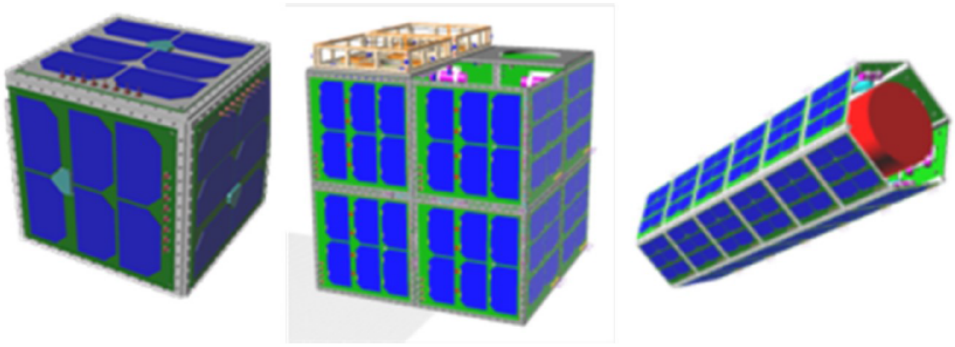

Figure 4.1: AraMIS structure examples

4.2.1 AraMiS Satellite Subsystems

The AraMiS satellites can achieve the desired flexibility level by combination of several subsystems together. The main subsystems of this architecture are:

- Mechanical;
- Power management;
- Attitude determination and control;
- Telecommunication;
- Payload;

Mechanical subsystem

The mechanical subsystem is the backbone of a satellite. It provides physical structure to the satellite and holds in place all the subsystem together and also protect them from environment conditions. The main material used for building the AraMiS structure is

Aluminum, used in particular for its light weight. The structure skeletal is made by metallic square rods while the power management and telecommunication subsystems are mounted on thin panels that are screwed to the rods. The power management tiles cover the satellite with solar panels mounted on the external face. The number of these tiles mainly depends on satellite size and power requirement. This provides a degree freedom to mission designers since size and generated power can be increased by simply adding more modules. All the tiles are connected on the external faces of the satellite and the payload is mounted inside which can be altered by the mission requirement [5].

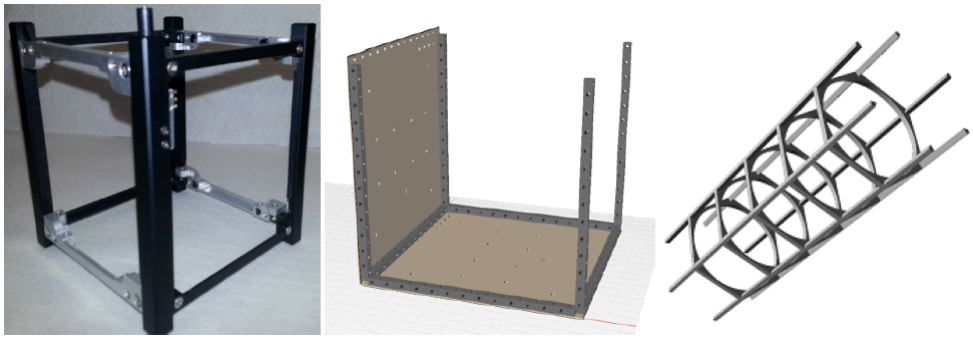

Figure 4.2: AraMIS mechanical structure

Power management subsystem

The power management subsystem is responsible for generating, storing and delivering power to all the other satellite subsystems. It is one of the most critical subsystem as a failure here can lead to shutting down everything. Fault tolerance is an important parameter and most of the design solutions were selected for this reason. Conventionally, power management is mission dependent which requires ad-hoc development for the specific needs. This tends to increase overall system cost and testing time. For this reason the AraMiS project uses modular power management system that can be adapted for various missions [5].

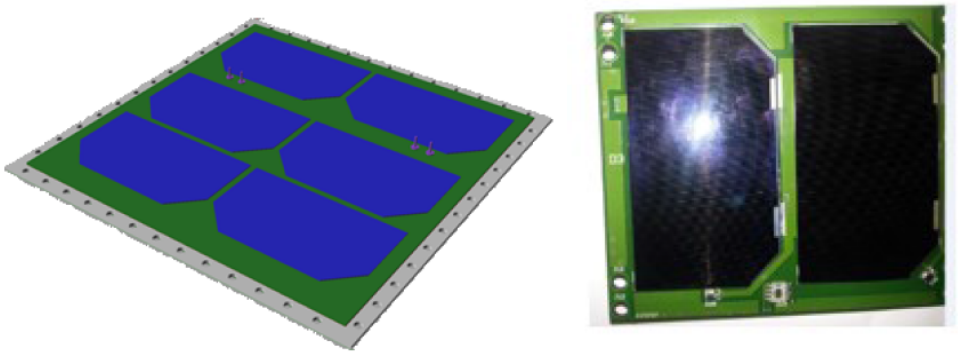

Figure 4.3: Solar panels used in AraMIS

Attitude determination and control subsystem

This subsystem is mainly responsible for sensing and modifying satellite orientation for keeping the tile subsystems pointing at their targets. Attitude control can be performed in passive or active way: passive attitude control is usually achieved by mounting a permanent magnet in the satellite which acts as a compass in the Earth magnetic field. This system is extremely simple and consumes no power. The main drawback is lack of spin control due to the variable Earth magnetic field. Active control is performed using controlled actuators that modify satellite attitude on OBC commands. In AraMiS, attitude control is automatically performed by the satellite using magnetorquer and reaction wheels. For attitude determination, three types of attitude sensors are used: magnetic, spin and Sun sensors. These sensors consist of COTS components which were selected on the basis of small dimension, light weight and low power consumption while achieving better performances [5].

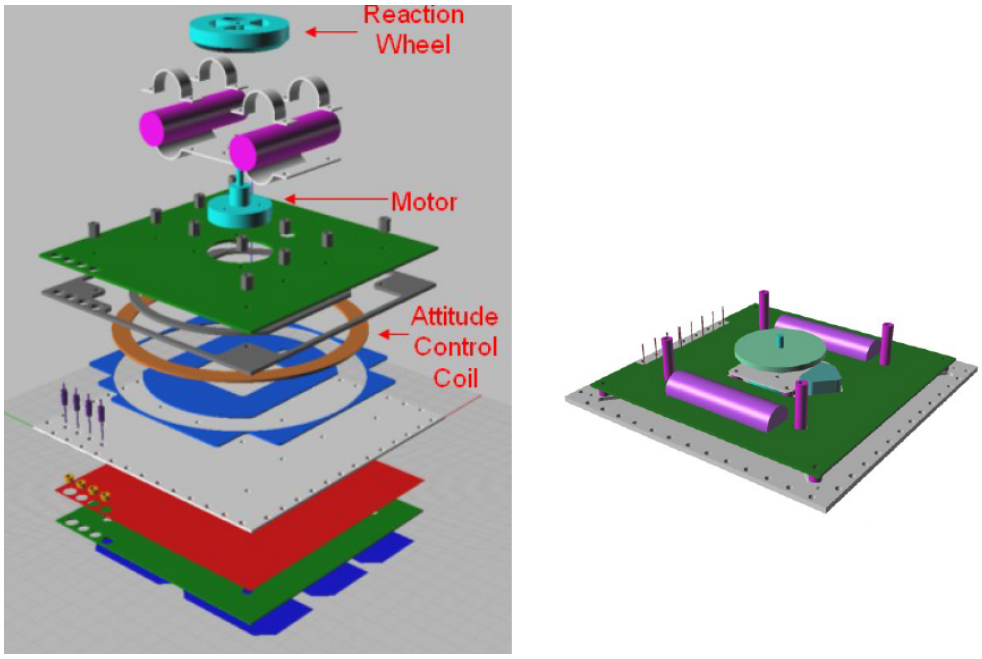

Figure 4.4: AraMIS ADCS modules

Telecommunication subsystem

The AraMiS telecommunications subsystem follows the modularity concept. There is a basic telecommunication tile that is provided in a standard AraMiS satellite. In case of special applications, dedicated tiles, can be added to meet mission criteria. This module is used to receive command and control packets from the ground station and to send back telemetry and status information. The bandwidth needed to exchange this kind of information is usually low, so the RF link was designed for low speed and low power. The module has been designed using COTS components which were selected to achieve good fault tolerance level. There are two different frequency bands used for satellite and ground communication: the UHF 437MHz and the S-band 2.4 GHz. To reduce occupied bandwidth, both channels are implemented using halfduplex protocol, sharing the same frequency for downlink and uplink [5].

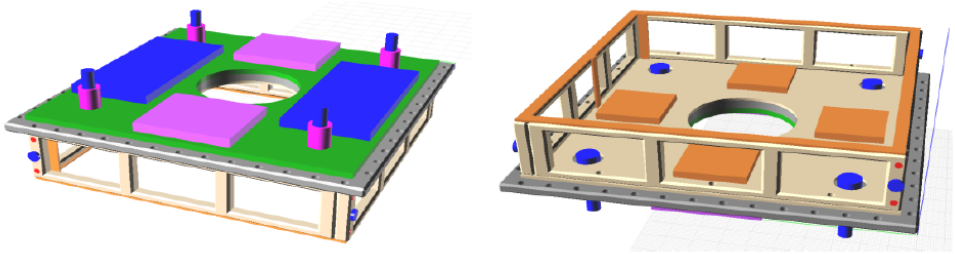

Figure 4.5: Telecommunication module

Payload

The payload is heavily mission dependent and the architecture was developed to allow high flexibility on it: the main requirements that the AraMiS architecture poses on the payload is its compatibility with the tile power distribution and data bus. Different payloads can be fitted in the various configurations but mechanical fixtures should be developed to connect them to the mechanical structure [5].

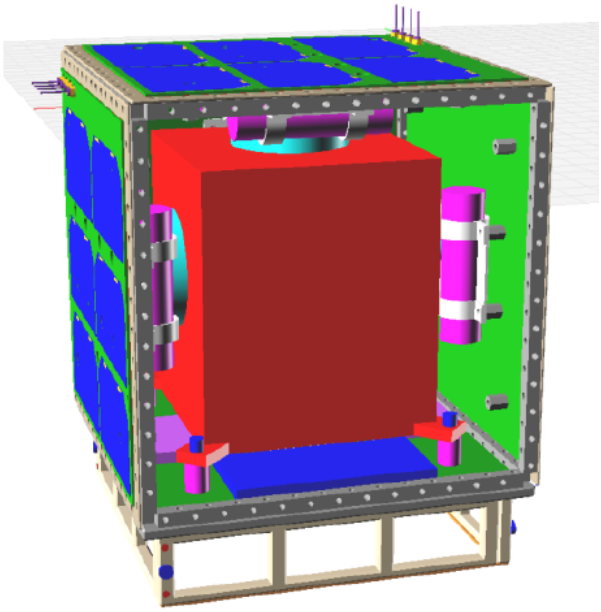

Figure 4.6: An example payload in the AraMIS structure

CHAPTER 5

1B521 Radiation Characterization Payload : Hardware

This chpater is intended to be read beside the developed Menthor Graphics DX-Designer project.

5.1 Hardware specifications

- Power consumption: less than $800mW$;
- Power supply voltage: $3.3V$ compatible with the AraMIS Power Distribution Bus (PDB);
- Physical Printed Circuit Board (PCB) dimensions: $9x9cm$;

5.2 AraMIS Power Distribution Bus

The PDB is based on a proprietary technology for distributed power power management. It is used to supply all the tiles in the structure, so in order to understand how it works a very quick overview of its functionalities will be given.

It is a fully distributed power management system which allows a high degrees of modularity and flexibility in spacecraft (re)configuration.

One can add as many primary sources, energy storages and loads without the need to design, reconfigure or resize the whole EPS.

It can be used as an effective alternative to traditional EPS, which are designed ad-hoc according to spacecraft requirements and power budget.

Basic concepts:

- The more loads are present or the higher is power demand, the more primary sources and/or energy storages can be added and the system self adapts to the new configuration;
- It intrinsically supports Point of Load conversion;
- It allows having one energy storage for each load or group of them;
- Significant increase in fault tolerance through distributed redundancy;
- Reduces wear of energy storage and allows dedicating specific energy storage for emergency.

It distributes power generation, storage and distribution tasks over a variable number of subsystems allowing flexible management of modularity.

Six types of element interact with the PDB:

1. one or more primary sources, which can autonomously source power without prior charging;
2. one or more optional battery sources, which can source power only upon previous charging;
3. one or more battery chargers, capable of sinking and storing energy for battery sources;
4. usually several loads, which sink power for their nominal operation;
5. optionally active shunts, purposely placed to sink overabundant energy or for active thermal control;
6. one or more compulsory over-voltage protectors; aimed at limit bus voltage in particular situations;

All elements except over-voltage protectors can be disabled and enter in idle mode (no energy exchange) for energy saving or emergency recovery.

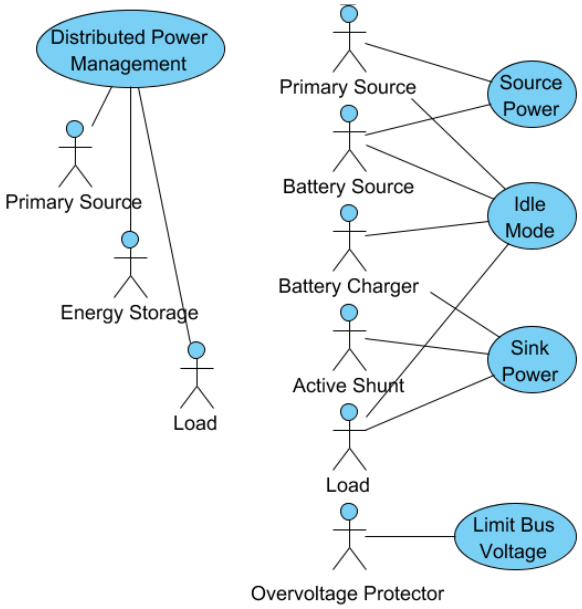

Figure 5.1: PDB actors

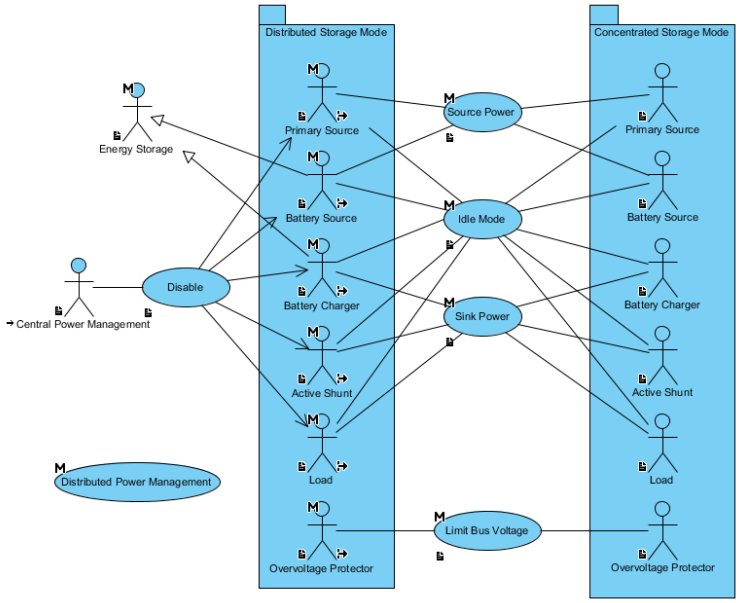

Figure 5.2: PDB use cases

5.3 Hardware overview

In this and in the following sections, an overview about the hardware structure will be given dividing the study in blocs presented in fig.5.3. The board presents two identical micro-controllers MSP430FR6989 that run the same program but compiled in two

different ways: hardened and not hardened. This choice has been done in order to characterize the micro-controller under test and at the same time the AraMiS hardening software library. For short the two on-board micro-controllers will be called μC_1 and μC_2.

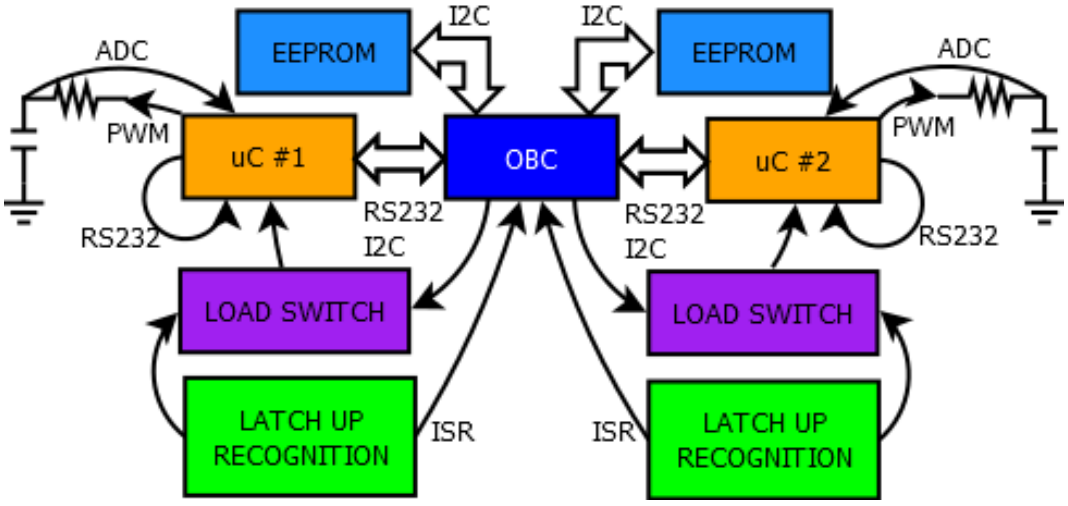

Figure 5.3: Hardware blocks scheme

Each micro-controller can communicate with the OBC in two ways:

- using the Recommended Standard 232 (RS232) protocol;
- using the Inter Integrated Circuit (I2C) protocol;

As already said a design choice the RS232 is used as the first communication option while the I2C is used as backup one.

5.3.1 MSP430FR6989 overview

Here a quick overview about the key features of the used microcontroller will be given.

- 16-Bit RISC Architecture up to 16 MHz Clock;
- Wide Supply Voltage Range (1.8 V to 3.6 V);
- LPM current:
  - Active Mode: $100\mu A/MHz$;

  - Standby: $0.4\mu A$;
  - Real-Time-Clock: $0.35\mu A$;
  - Shutdown: $0.02\mu A$;

- FRAM characteristics:
  - 128KB;
  - Fast Write at 125 ns per Word (64KB in 4 ms);
  - Unified Memory = Program + Data + Storage in one single space;
  - 10^{15} write endurance;
  - Radiation Resistant and Nonmagnetic;

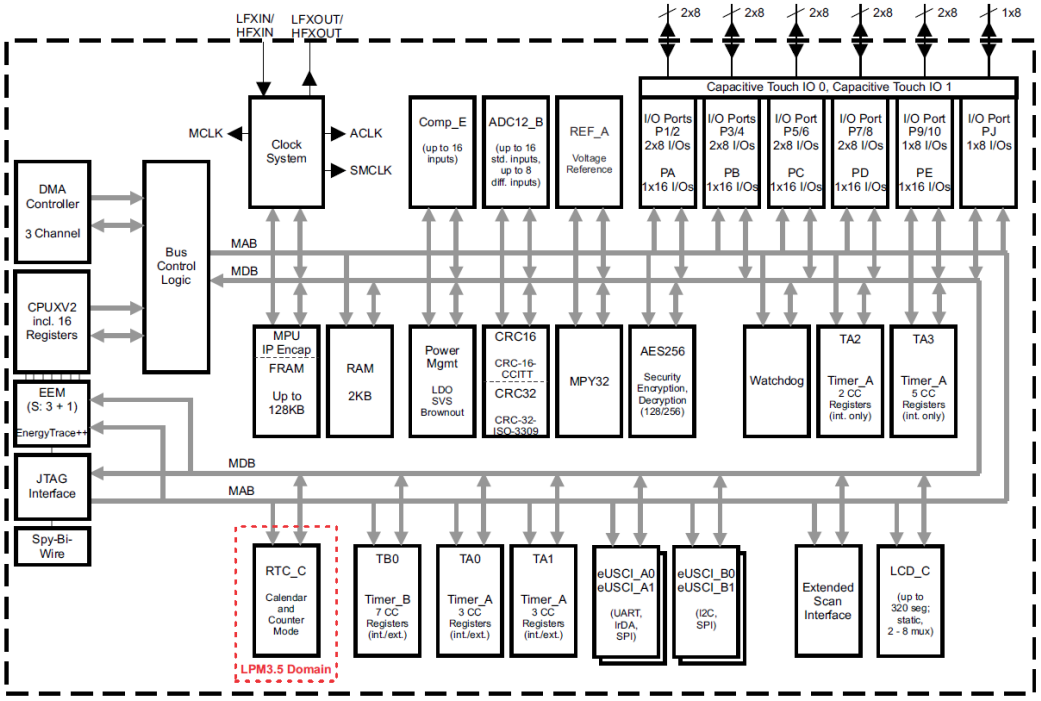

Figure 5.4: Micro-controller's block scheme

5.3.2 Micro-controller's modules

The pins of the micro-controller are grouped in modules. Each module consists of 10 pins (D0 to D9) and any of them has a specific function. In the table reported in fig.5.5,

the column pin indicates the pin position on the chosen connector, while the letters form A to H indicate the corresponding module. The couple of modules A and B, and C and D share the same pins for the I2C communication. Since the chosen micro-controller has only enhanced Universal Communication Interface (eUSCI) A0, A1, B0 and B1, module E, F, G and H don't have the possibility to wire Universal Asynchronous Receiver Transmitter (UART)s, I2Cs and SPIs communication.

Conn	Pin	A	B	C	D
D0/RX/SOMI	11	P2.1/UCA0SOMI /UCA0RXD	P7.5	P5.5/UCA1SOMI/ UCA1RXD	P4.2
D1/TX/SIMO	9	P2.0/UCA0SIMO /UCA0TXD	P7.6	P5.4/UCA1SIMO/ UCA1TXD	P4.3
D2/SCL/SOMI	7	P1.7/UCB0SCL	P1.7/UCB0SCL	P4.0/UCB1SCL	P4.0/UCB1SCL
D3/SDA/SIMO	5	P1.6/UCB0SDA	P1.6/UCB0SDA	P4.1/UCB1SDA	P4.1/UCB1SDA
D4/CLK	3	P2.2/UCA0CLK	P2.3/UCA0STE	P3.6/UCA1CLK	P3.7/UCA1STE
D5/PWM	1	P6.5/TB0.1	P7.2/TA0.1	P7.3/TA0.2	P3.3/TA1.1
D6/A0	12	P1.0/A0	P1.2/A2	P8.6/A4	P8.4/A6
D7/A1	10	P1.1/A1	P1.4/A3	P8.5/A5	P9.0/A7
D8/ID/INT	4	P1.4	P1.5	P3.0	P3.1
D9/EN/PWM2/INT	2	P2.4/TB0.3	P2.5/TB0.4	P2.6/TB0.5	P2.7/TB0.6

	Pin	E	F	G	H
D0/RX/SOMI	11	P6.0	P7.0	P8.0	P5.6
D1/TX/SIMO	9	P6.1	P7.4	P8.1	P5.7
D2/SCL/SOMI	7	P6.2	P6.2	P8.2	P8.2
D3/SDA/SIMO	5	P6.3	P6.3	P8.3	P8.3
D4/CLK	3	P5.0	P5.1	P5.2	P5.3
D5/PWM	1	P7.1/TA0.0	P6.6/TB0.2	P10.1/TA0.0	P10.1/TA1.0
D6/A0	12	P9.1/A8	P9.2/A10	P9.4/A12	P9.6/A14
D7/A1	10	P9.2/A9	P9.3/A11	P9.5/A13	P9.7/A15
D8/ID/INT	4	P3.2	P3.4	P3.5	P4.4
D9/EN/PWM2/INT	2	P4.5/TA1.0	P4.6/TA1.1	P4.7/TA1.2	P6.4/TB0.0

Notes:

a. ***Odd modules:*** *Modules A, C, E, G*
b. ***Even modules:*** *Module B, D, F, H*
c. D2 and D3 shared by pairs
d. MSP430FR6989 has only USCIA0, USCIA1, USCIB0, USCIB1

Figure 5.5: Microcontroller's modules and wiring

For this specific applications only the modules A and C are used for both the micro-controllers.

Bk1B4223W_Tile_Processor_8M_FRAM

In fig.5.6 the connections between the micro-controller, its basic components (i.e decoupling capacitors, crystals) and the modules buses are reported. The values of the rails decoupling capacitors were choose according to the device's datasheet. In order to speed up feature developing, using this micro-controller, all its needed connections have been wired in a reusable block called **Bk1B4223W_Tile_Processor_8M_FRAM**. In this way if a new design based on this micro-controller has to be started, these connections are already present.

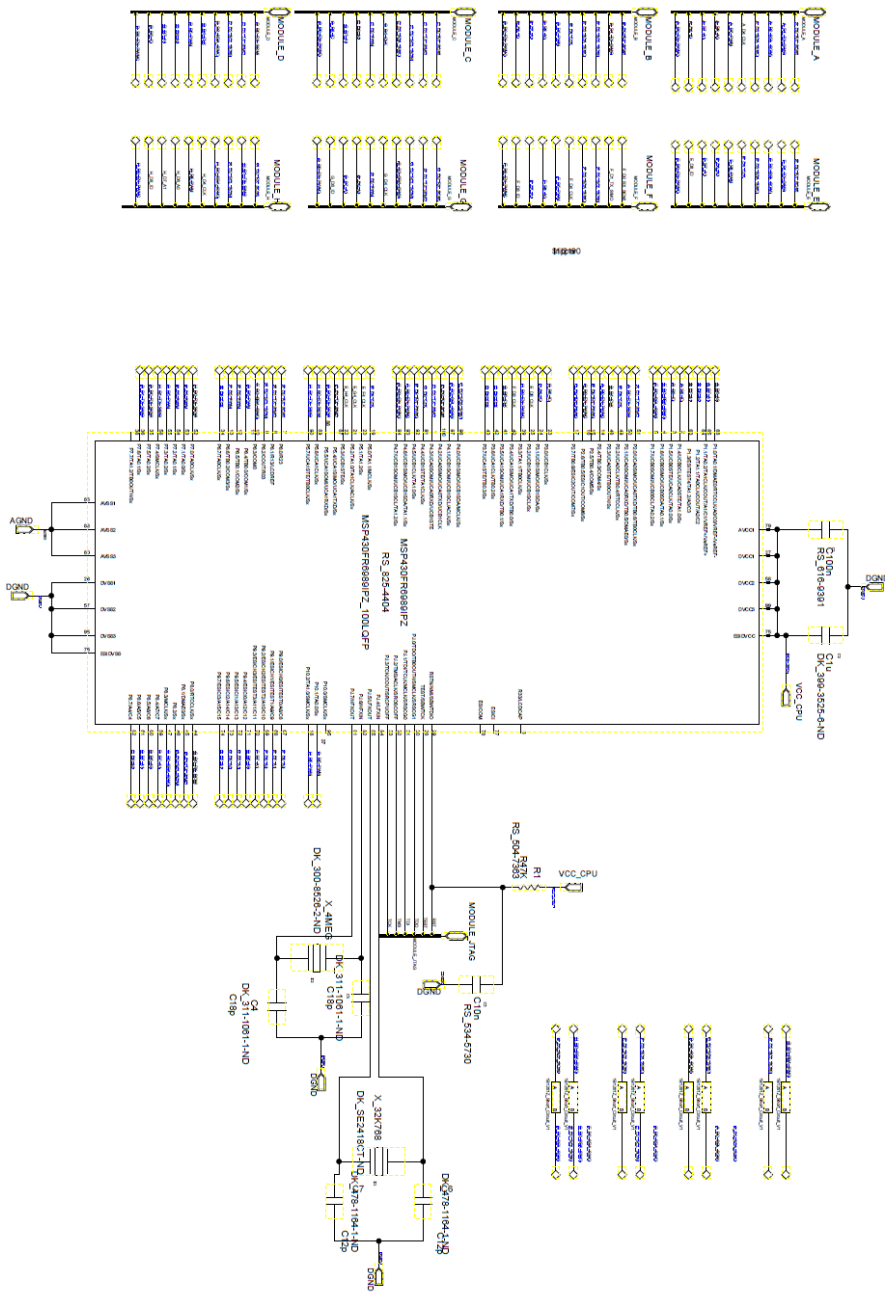

Figure 5.6: μC connections

Module A wiring

The module A is used in order to communicate and interact with the OBC and it's wired in the following way.

- D0/RX/SOMI and D1/TX/SIMO: used for the RS232 communication;
- D2/SCL/SOMI and D3/SDA/SIMO: used for the I2C communication;
- D4/CLK: used by the OBC to reset the target;
- D5/PWM: used with the D4/CLK to enter in the BSL mode;
- D6/A0: used both by the target and by the OBC to sense the current sensor output voltage;
- D7/A1: as will be discussed later on this pin is used to jump to the second FRAM memory copy of the firmware;
- D8/ID/INT: used in order to sense the output of the comparator in the Latch-up recognition circuit;
- D9/EN/PWM2/INT: used to cut the power supply of the target acting on the load switch enable of the target.

Module C wiring

The module C is intended for self test operation. This choice has been made in order to test the micro-controller's normal operation is affected by the space environment trying to use the more resources as possible at the same time (see 6.6.1) and see 6.6.1).

- D0/RX/SOMI and D1/TX/SIMO: are wired in a loop-back mode;
- D5/PWM: used to generate a Pulse Width Modulation (PWM) signal;
- D6/A0: filtering the PWM signal generated by D5/PWM by means of an RC Low Pass Filter (LPF) the ADC operation is checked;
- D7/A1 and D9/EN/PWM2/INT: used as D5/PWM and D6/A0. This wiring has been made in order to have the possibility to test the also the differential input operation of the ADC.

5.3.3 RC Low Pass Filter

The RC LPF is used in order to extract the average value of the generate PWM signal and use the ADC in order to check the correct behavior of the two peripherals. Since the reading is used only as a general check it is not necessary to use more complicated filter techniques. The cutting frequency at which the RC filter mus act can be computed in the following way.

The ADC operates on 12 bits at 3.3V so an LSB corresponds to:

$$V_{LSB} = \frac{3.3V}{2^{12}} = 805.66\mu V$$

We also know that, since an RC filter is a first order filter, it has a slope of $-20dB/dec$. Fixing an acceptable uncertain on the ADC's read of 5 LSBs we can find the attenuation factor:

$$G = 20log\frac{5 \cdot 805.66 \cdot 10^{-6}V}{3.3V} = -58.26dB$$

Approximating it with $60dB$ we have to place the RC LPF three decades before the PWM's frequency.

The Module C D5/PWM PWM is associated to the TIMER A0 while the Module C D9/EN/PWM2/INT is associated with the TIMER B0. The period values of the two timers are fixed and defined by the used CPU_DESCRIPTOR. In this case they are respectively equal to:

$$f_{PWM_{A0}} = MCLK\_FREQ[Hz] \cdot TIMERA0\_PERIOD[\mu s]/8/1000000$$

$$f_{PWM_{B0}} = MCLK\_FREQ[Hz] \cdot TIMERB0\_PERIOD[us]/8/1000000$$

Where:

- Factor 1000000 is to compensate $Hz \cdot \mu s$. Division by 8000000 is split into two divisions: $64(>> 6) \cdot 15625$
- $TIMERA0\_PERIOD = 1000$ defined in CPU_DESCRIPTOR_DEFAULT_FR
- $TIMERB0\_PERIOD = 2000$ defined in CPU_DESCRIPTOR_DEFAULT_FR

Clocking the micro-controller at $8MHz$ we obtain:

$$f_{PWM_{A0}} = 1kHz$$

$$f_{PWM_{B0}} = 2kHz$$

We are now able to find out the filter's components value. We have to remember that electrolytic capacitors are forbidden in space applications because the electrolyte can easily evaporate or explode in absence of pressure. So fixing $R_{A0} = 150k\Omega$ we can easily find C_{A0} as:

$$C_{A0} = \frac{1}{2\pi f_{PWM_{A0}} R_{A0}} = 1nF$$

Doing the same with $f_{PWM_{B0}}$ and $R_{B0} = 100k\Omega$ we get:

$$C_{B0} = \frac{1}{2\pi f_{PWM_{B0}} E_{B0}} = 1.5nF$$

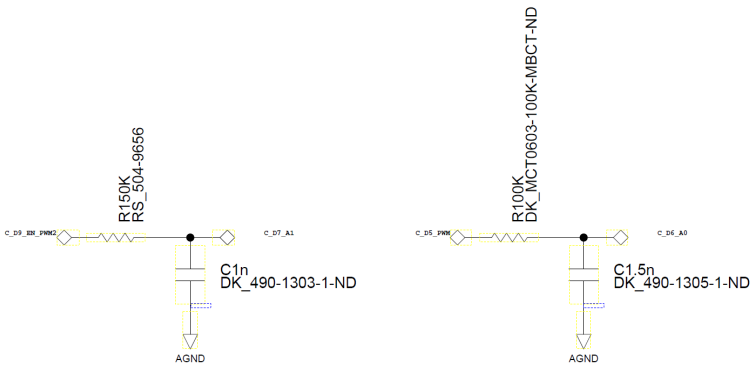

Figure 5.7: Designed RC filters

5.3.4 Load Switch

The load switch schematic is reported in fig.5.8. It simply consist of a p-MOS and an n-MOS that are used to supply or not the load attached to the OUT port. Putting a logical high voltage on the gate of the n-MOS cause its drain voltage to go down (it goes in triode) and so does the p-MOS. The absorbed current of this circuit can be easily calculated. When the n-MOS is ON the absorbed current I_{EN} is equal to:

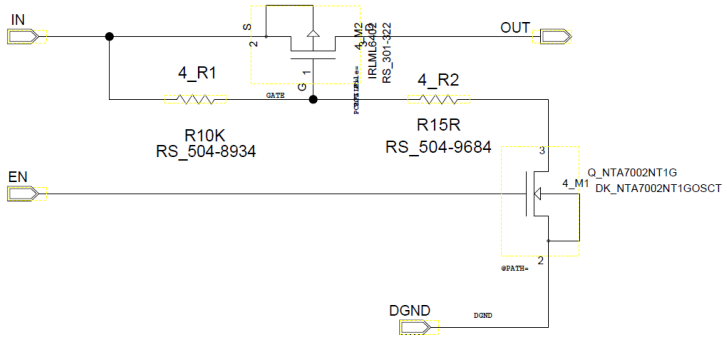

Figure 5.8: Load switch schematic

$$I_{EN} = \frac{V_{IN}}{R_1 + R_2 + R_{ON_n}}$$

Neglecting the contribute due to R_{ON_n} (since it is very low) and supposing $V_{IN} = 3.3V$ we get:

$$I_{EN} = 329\mu A$$

5.3.5 Latch-up recognition circuit

If a latch-up occurs the power supply has to be cut-off, and the event to be counted. In case of a latch-up, the affected micro-controller starts to absorb a very huge amount of current, that is for sure greater than the absorbed current in Active Mode (AM). The latch-up recognition circuit has to drive the load switch (see 5.3.4) and give feedback to the OBC of the event. If the micro-controller is affected by a latch-up event, it can't count the event since it stops working normally so, the OBC counts this type of events. The output of the latch-up recognition circuit is connected to module A D8/INT (that is an interrupt pin for the OBC) so a routine interrupt can be used. The OBC has also to pull down all the pins connected to the micro-controller in order to avoid a current flow through the clamping diodes. This circuit consist of two blocks:

1. current sensor;

2. non inverting comparator;

Bk1B32G_Config_Current_Sensor

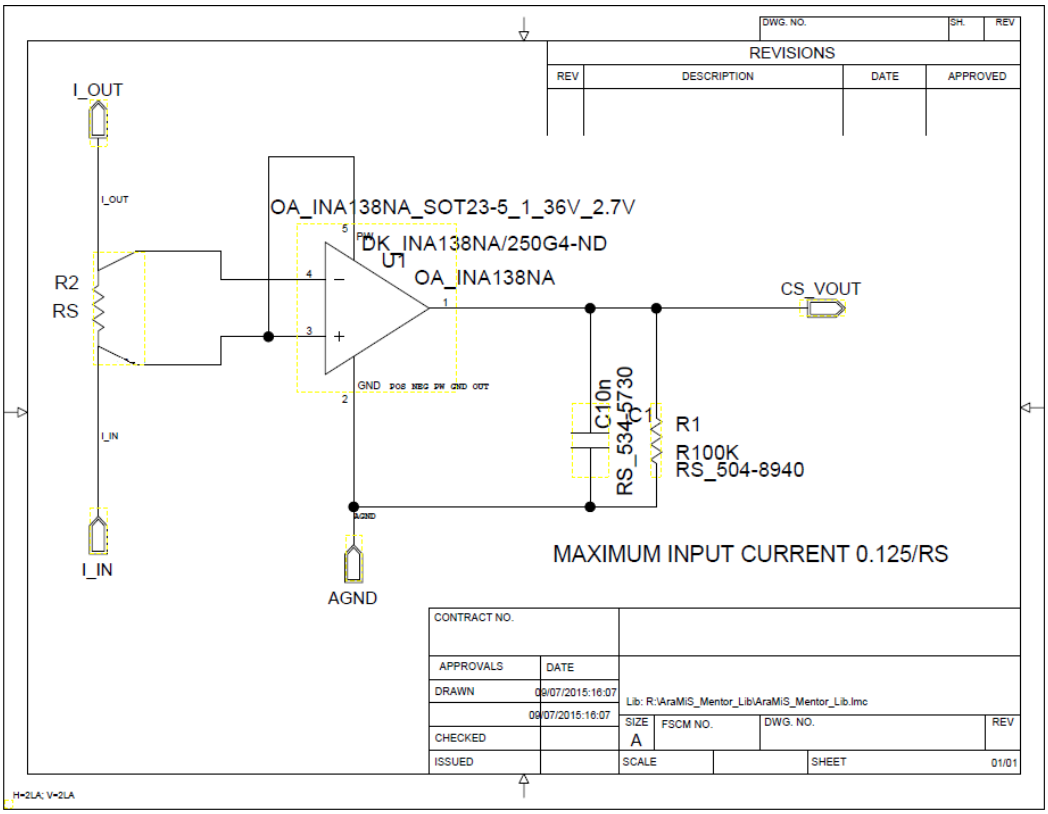

Figure 5.9: Current sensor schematic using INA138 IC

The current sensor has been designed using an INA138 (High-Side Measurement Current Shunt Monitor) Integrated Circuit (IC). It's output voltage is given by the following relationship.

$$V_o = \frac{R_S \cdot R_L \cdot I_S}{5k\Omega}$$

The output of the current sensor is also sent to one analog input of the device in order to monitor the current consumption. It is in fact possible to read these value sending a special command to the desired micro-controller (see 6.2.2). The **Bk1B32G_Config_Current_Sensor** is designed in order to change the value of only one resistance (R_S) to get the desired

The MSP430FR6989 datasheet states that in AM the micro-controller absorb a current:

$$I_{AM} = \frac{103\mu A}{MHz}$$

Clocking it at $8MHz$ (design choice in order to avoid FRAM's wait states as indicated in the device's datasheet) we obtain that the maximum absorbed current in AM $I_{MAX_{AM}}$ is:

$$I_{MAX_{AM}} = 8MHz \cdot \frac{103\mu A}{MHz} = 0.824mA$$

The indicated current absorption in AM takes into account the consumption of all the micro-controller's peripherals, but in order to take into account peaks of absorption we can multiply it by three to be sure that only latch-up phenomena will be detected.

$$I_{MAX} = 3 \cdot I_{MAX_{AM}} = 2.5mA$$

Choosing that at this input current corresponds an output voltage $V_o = 1.5V$ we an find the product $R_L \cdot R_S$ as:

$$R_L \cdot R_S = \frac{5k\Omega \cdot V_o}{I_S} = \frac{5k\Omega \cdot 1.5V}{2.5mA} = 3 \cdot 10^6$$

Having:

$$R_L = 100k\Omega$$

We obtain:

$$R_S = 30\Omega = 33\Omega_{(E12value)}$$

Using this value of R_S the maximum voltage drop we get across it is equal to:

$$V_{R_S} = R_S \cdot I_{MAX} = 25mV$$

Which is good because the datasheet suggests to have a maximum differential voltage of $80mV$.

Using these values to compute again the corresponding output voltage we obtain that:

$$V_o = \frac{R_S \cdot R_L \cdot I_S}{5k\Omega} = \frac{30\Omega \cdot 100k\Omega \cdot 2.5mA}{5k\Omega} = 1.65V$$

which is good too because we over-sized it.

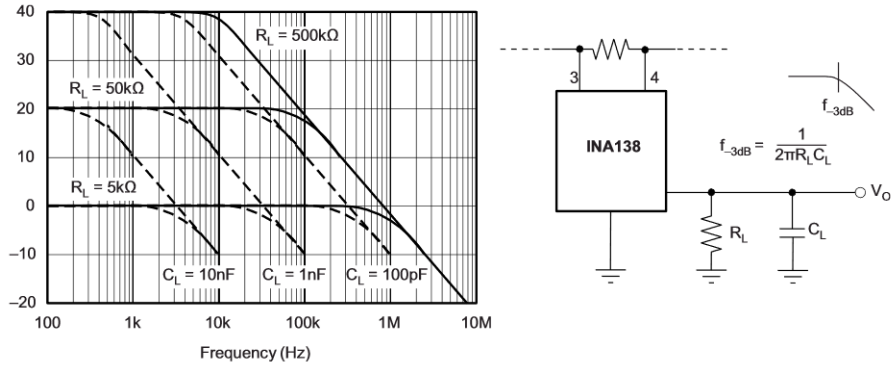

Figure 5.10: INA138 frequency response as function of R_L

For what concern the output filtering, as can be easily seen in fig.5.10.

We can easily compute the f_{-3dB} obtained using the chosen components as:

$$f_{-3dB} = \frac{1}{2 \cdot \pi RC} = \frac{1}{2 \cdot \pi 100k\Omega \cdot 10nF} = 159.23Hz$$

Non inverting comparator

This block has to effectively drive the load switch if the fixed absorbed current threshold is exceeded, cutting off the micro-controller power supply. The comparator simply consist of a voltage divider and an Operational Amplifier (OP-AMP). Since on the developed board there is the need of two latch-up recognition circuit (so two non inverting comparators), an OPA2703 has been chosen as operational amplifier because it offers two OP-AMPs. In fig.5.11 a principle schematic of a non-inverting compactor is reported. In our case V_i represents the output of the current sensor described by 5.3.5 whileV_H the power supply voltage supplied by the AraMIS PDB (see 5.2) equal to $3.3V$.

The reference voltage, that we have on the inverting terminal of the operational amplifier, is equal to V_H scaled down by the factor $\frac{R_1}{R_1+R_2}$. As found before (see 5.3.5) the threshold voltage in order to act on the load switch should be $1.52V$, so in order to have a comparator that works as desired, the following relationships have to be satisfied:

$$V_H \cdot \frac{R1}{R_1 + R_2} > 1.52V$$

Since we previously obtain that for the chosen current threshold, we get an output voltage of $V_o = 1.65V$ and since the in our case $V_H = 3.3V$ we choose:

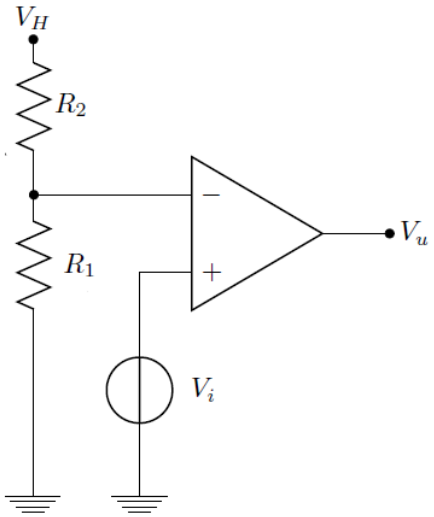

Figure 5.11: Non inverting comparator

$$R_1 = R_2 = 10k\Omega$$

Operation principle

Now we have have all we need for the latch-up recognition circuit.

The load switch (see 5.3.4) cuts off the supply if on its enable pin it has a logic zero. The enable is driven high by the OBC (using the D9_EN_PWM2 pin of module A) if it wants to have the payload on, but it can also be driven down by the means of the non inverting comparator. So in case of a latch-up the power is removed.

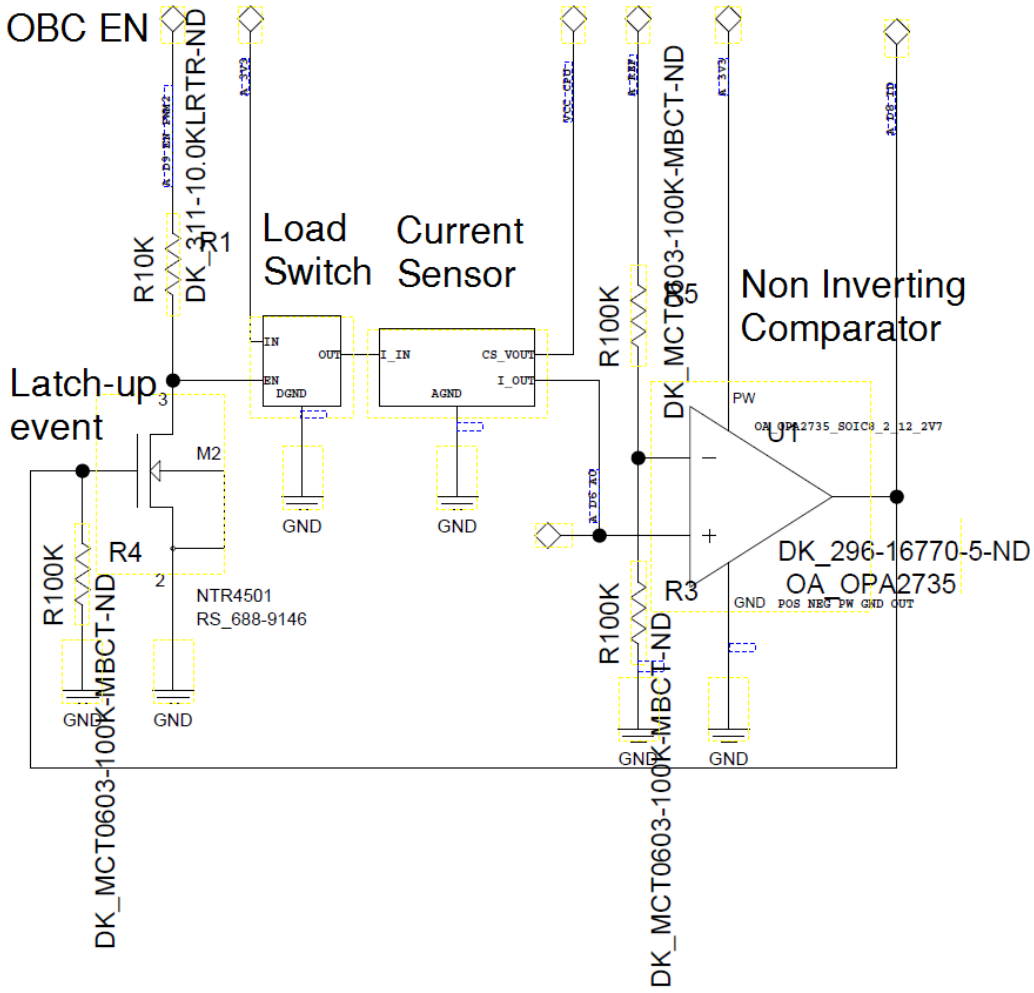

Figure 5.12: Latch-up recognition circuit

5.3.6 EEPROM I2C Memory

In the unlucky case of failure of the double main program redundancy (see 6.7.1), an I2C memory can be used in order to restore the micro-controller's firmware. The OBC can in fact access to this memory in which N copies of the firmware are present and by means of the Boot Strap Loader (BSL) functionality (see 6.4) can write it to the target. A first approximation about the dimension that this memory must have can be made in the following way: the on-board FRAM has a dimension of 128kB. If as a worst case we consider that we use all the memory to store the program we have to guarantee that at least four back-up copies of the firmware should be available. This is a very bad approximation because the error rate is proportional to the used area, so the program

have to use a very little memory area in order to lower the error rate. By means of these considerations, we can choose:

$$Memory > 4 \cdot 128kB = 512kB$$

Inserting this parameter together with the fact that the memory must be an I2C one, the catalog has a very poor offer. The most of the I2C are in-fact EEPROM ones and they don't exceed the 512kB. So the the chosen memory is an 24LC512 from Microchip.

Microchip 24LC512 characteristics

This memory has the following key characteristics:

- Low-Power CMOS Technology:
  - Active current 400 uA, typical;
  - Standby current 100 nA, typical;
- I2C interface;
- 100 kHz and 400 kHz Clock Compatibility;
- Page Write Time 5 ms max;
- Self-Timed Erase/Write Cycle;
- Hardware Write-Protect;
- More than 1 Million Erase/Write Cycles;
- Data Retention > 200 years;
- Temperature Ranges:
  - Industrial: -40°C to +85°C;
  - Automotive: -40°C to +125°C;

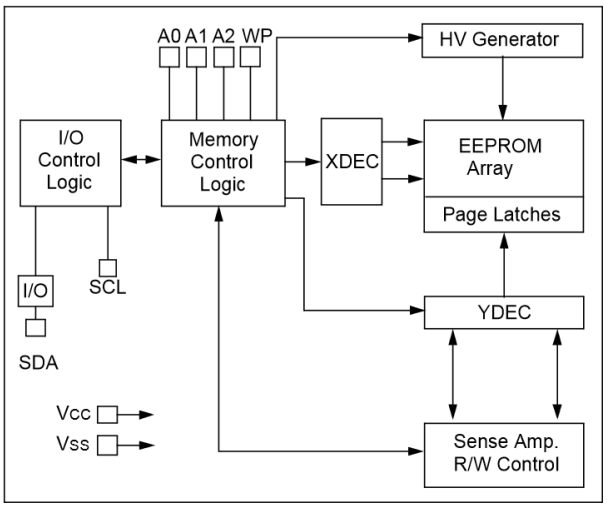

Figure 5.13: 24LC512 blocks scheme

Address selection

The address selection of the memory can be done using the pins A0, A1, A2. In order to make it the most configurable as possible, 0Ω resistors have been used in order to easily drive them to V_{CC} or GND like showed in fig.5.14.

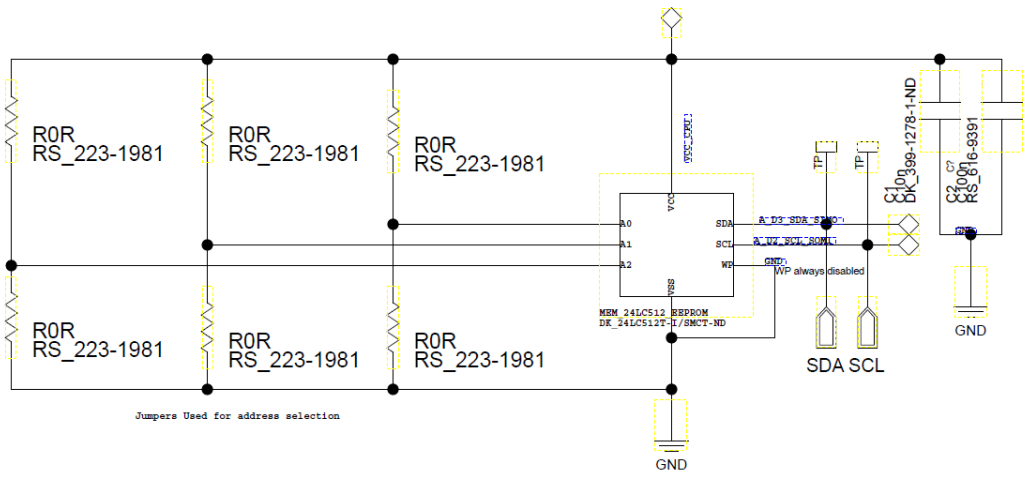

Figure 5.14: Schematic of 24LC512 address selection

5.3.7 Connectors

On the board there are four connectors:

1. MOLEX - 8 pins PicoBlade header : used to program μC_1 using a JTAG programmer;
2. MOLEX - 8 pins PicoBlade header : used to program μC_2 using a JTAG programmer;
3. Bk1B4811W Module Interface Receptacle used to wire the μC_1 MODULE A;
4. Bk1B4811W Module Interface Receptacle used to wire the μC_2 MODULE A;

5.4 Power consumption

Now we have described all the hardware we need we can make an estimation of the total current absorption.

- microcontrollers: $I_{\mu C_1+\mu C_2} = 2 \cdot 824\mu A = 1.5mA$;
- I2C memories: $I_{Memory_1+Memory_2} = 2 \cdot 400\mu A = 0.8mA$;
- load switches : $I_{LS_1+LS_2} = 2 \cdot 329\mu A = 0.66mA$;

The total current consumption is so equal to:

$$I_{TOT} = I_{\mu C_1+\mu C_2} + I_{Memory_1+Memory_2} + I_{LS_1+LS_2} = 2.17mA$$

$$P_{TOT} = I_{TOT} \cdot 3.3V = 7.16mW$$

The obtained power consumption is well lower than that one imposed by th specifications. Considering the ratio in fact we get a power lower of

$$P = P_{specs}/P_{TOT} = \frac{800mW}{7mW} \cdot 100 = 114.3$$

times.

5.4.1 Comparison with MSP430F5438A

If for example an MSP430F5438A is used as micro-controller on the same board we can make a comparison between the two equivalent power consumptions. In AM the micro-controller mentioned above has a current consumption:

$$I_{AM} = \frac{230\mu A}{MHz} \cdot 8MHz = 1.84mA$$

Evaluating again the power we get:

$$I_{TOT} = I_{\mu C_1 + \mu C_2} + I_{Memory_1 + Memory_2} + I_{LS_1 + LS_2} = 5.14mA$$

$$P_{TOT} = I_{TOT} \cdot 3.3V = 16.97mW$$

We so obtained a power saving (in percentage):

$$P_{saved_{\%}} = \frac{P_{MSP430F5438A}}{P_{MSP430FR6989}} = 237\%$$

using a FRAM micro-controller.

5.5 Complete schematic

The design is hierarchical so it is divided in blocks. Putting all the main block together (see 5.3.2, 5.3.3, 5.3.4, 5.3.6) and considering that two identical microcontrollers are present on the PCB the complete schematic can be summarized in the following way.

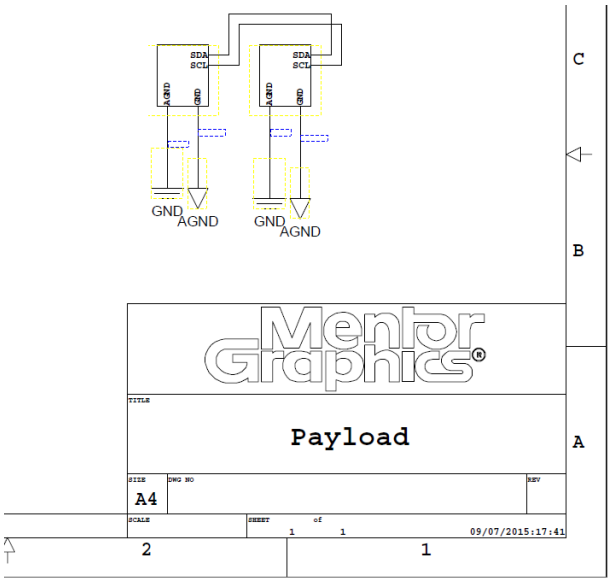

Figure 5.15: Top level schematic

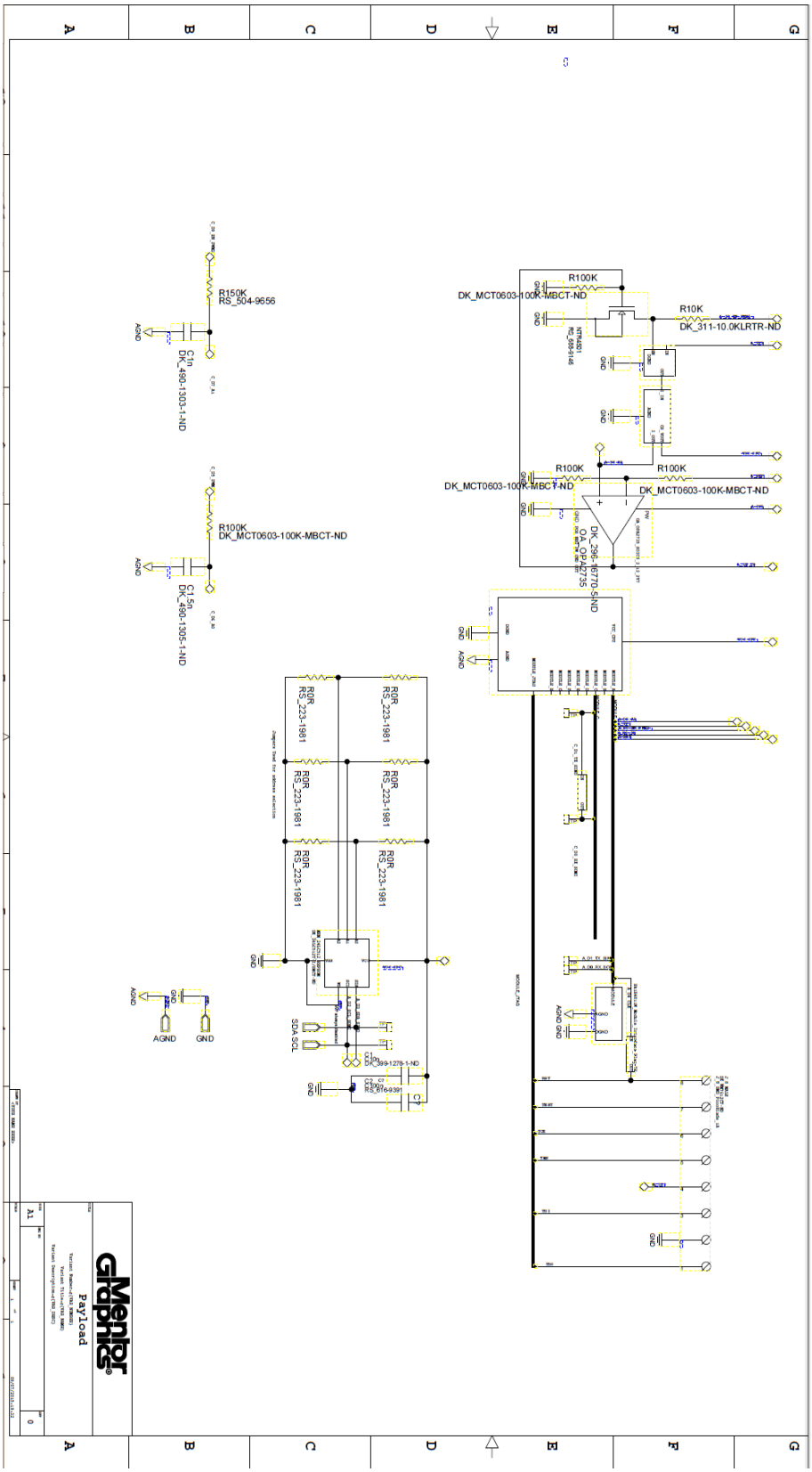

Figure 5.16: Complete schematic for each microcontroller

5.6 PCB layout

Here the layout of the developed PCB is presented. All the components have been positioned on the TOP layer. The routing has been done preferably on the TOP layer in order to have a more homogeneous BOTTOM ayer used as ground plane. The whole developed PCB is reported in fig.5.17. The dimensions of the board are $90mmx90mm$ in order to be compliant with the given specs.

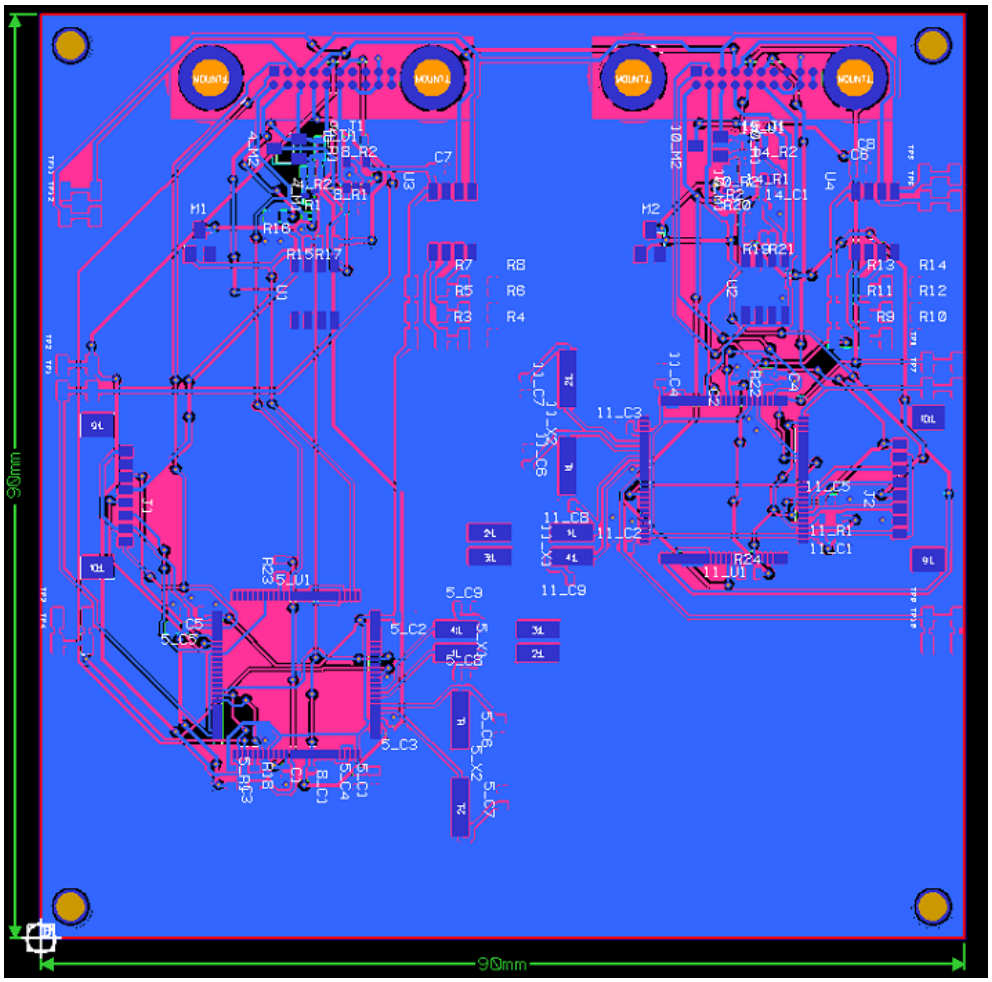

Figure 5.17: Whole PCB view

5.6.1 Connectors quotes

In fig.5.18 some useful quotes about the positioning of the connectors is reported.

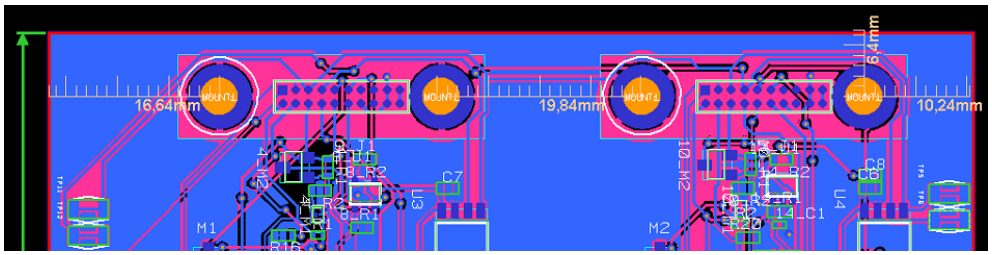

Figure 5.18: Connectors positioning quotes on the PCB

5.6.2 TOP layer

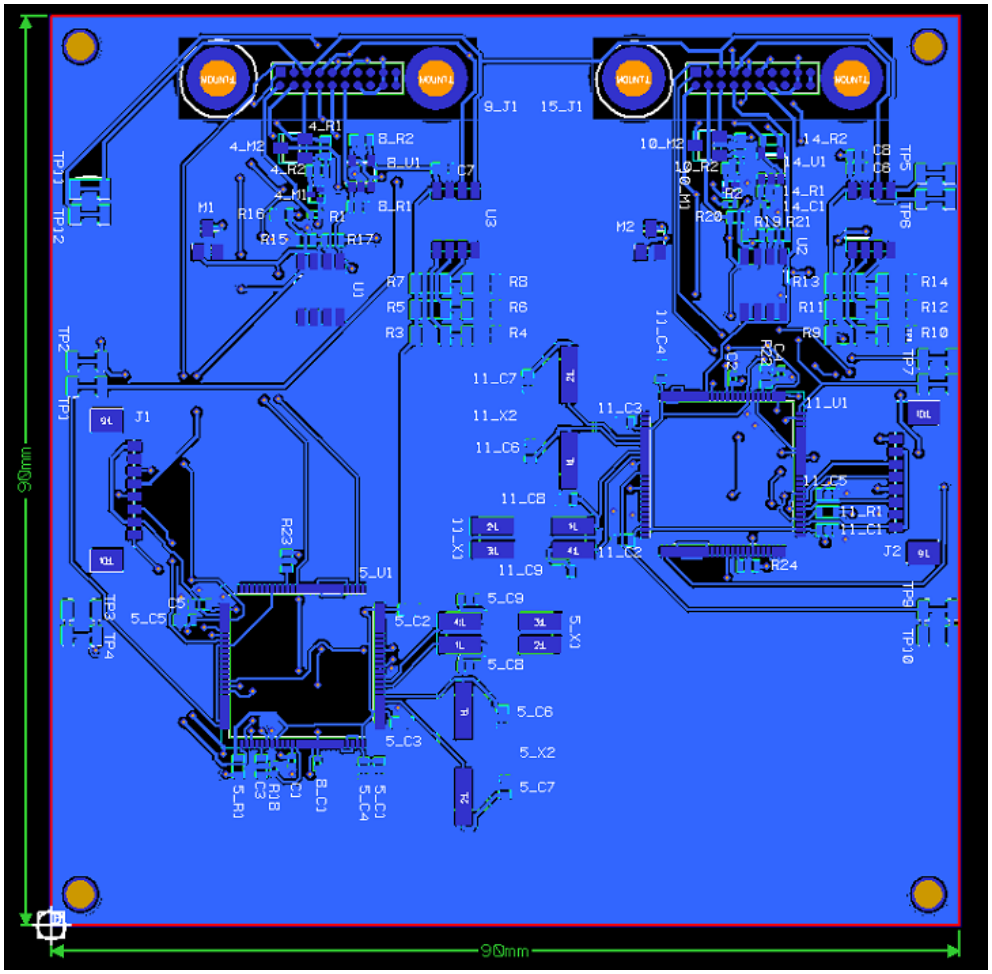

Figure 5.19: PCB TOP layer view

5.6.3 BOTTOM layer

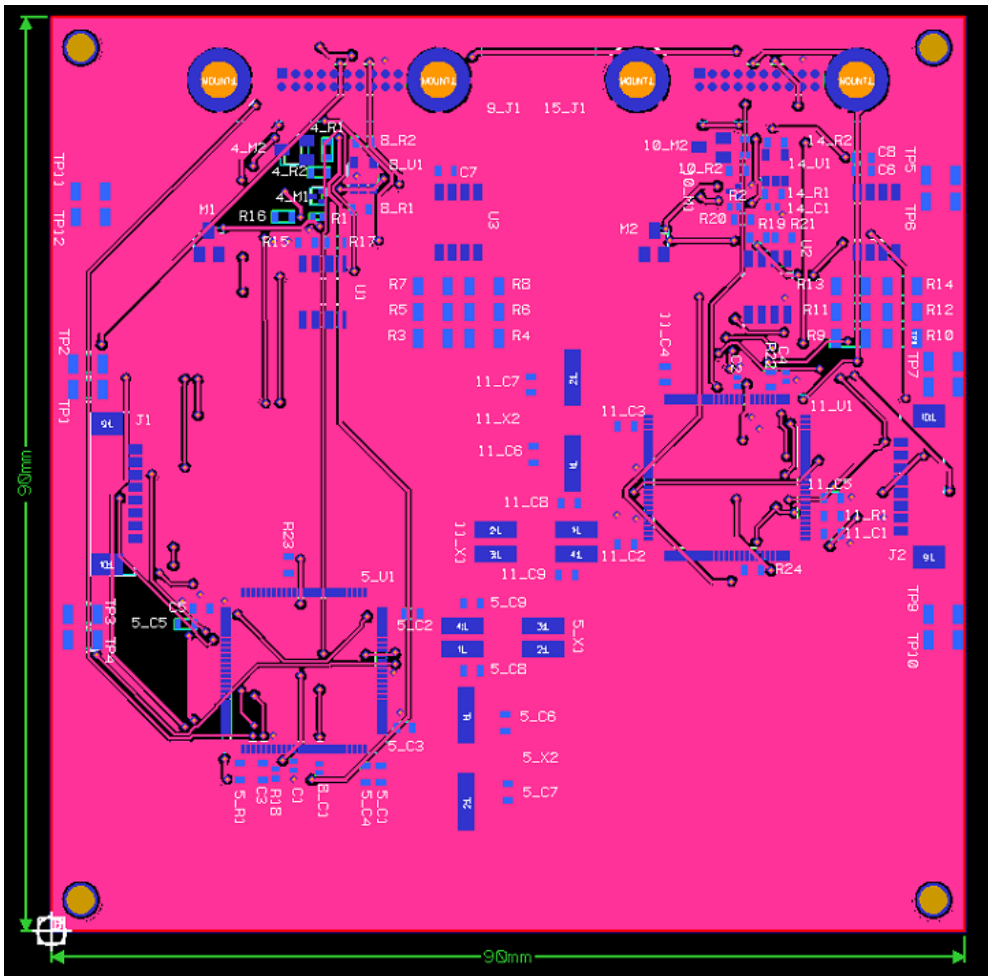

Figure 5.20: PCB BOTTOM layer view

5.7 Test points positioning

The test points have been positioned on all the used communication channels. In this way it is possible to easily use a protocol sniffer if it is needed to monitor the data exchange. They are positioned on the boarders of the board in order to be easily accessible. even if the board is mounted in the whole structure.

CHAPTER 6

1B521 Radiation Characterization Payload : Software

This chapter is intended to be read looking at the developed use case diagrams and class diagram that can be found in the UML repository. The developed software is in-fact commented in this chapter in theoretical therms, so the software implementation is available only in the UML repository. The developed code is also commented within all the software developed class in order to better understand all the operations and why they are made.

6.1 Software specifications

- Compatibility with AraMIS UML standards;
- Comparison with AraMIS software hardening library;
- Introduce TI's FRAM microcontrollers in the AraMIS structure and test them;
- Code compatible with the AraMIS UML standards.

6.2 AraMIS facilities

In this section the used facilities of the AraMIS project will be described, pointing out the key features.

6.2.1 Software hardening

The software programs running on commercial off the shelf processor are more prone to radiation. They may suffer from sensitivity to ionizing radiations, therefore to radiation induced SEE. SEUs are transient faults caused by the ionization of a single charged particle and typically cause bit-flips which is an undesired change of state in the content of a storage element. Most software-based approaches are aimed at detecting faults, some of them apply redundancy to high-level source code by means of automatic transformation rules, whereas some others use instruction redundancy at low level (assembly code) in order to reduce the code overhead and performance degradation, and improve the detection rates. The AraMiS software was developed using a high level software radiation-hardening technique completely developed in house. The radiation hardening technique aim at protecting the on-board computer and on-board data handling functions from SEEs (mostly SEUs), in particular:

- Bit-flips and data corruption in data storage;
- Corruption of time and spacecraft status and configuration registers (e.g. subsystem enable/disable, configuration word, calibration data, etc.), which might potentially be harmful to the whole system;
- Corruption of peripheral configuration registers in the micro-controller (e.g. interrupt enables and flags, configuration words of Analog to Digital Converter (ADC)s, UARTs and Timers, etc.).

The radiation-hardening technique is based on the use of appropriate C++ classes from the hardened data (**Hdata**) package developed in house, which can be used in a standard C++ program instead of standard data type. For instance, a short can be substituted by the so-called **TripleShort**, which automatically and transparently stores three copies of the same value and votes or recovers data whenever required. A normal C++ program can still be compiled by modifying only the data type definitions. This allows reusing software algorithms and procedures which have already been validated and tested without any specific effort apart from redefining data types. For instance, the piece of code on the left column of fig.6.1 can be changed to the code in the right column by changing only the first line.

Normal program	Hardened program

short a=3, b=5;
short c;
c = a+b;
	TripleShort a=3, b=5;
	short c;
	c = a+b;

Figure 6.1: Software hardening example

The instruction **a = 3**, **b = 5**; automatically stores the values **3** and **5** into three replicas **a** and **b**, respectively. The operation **a + b** automatically sums up each replica of **a** with the corresponding replica of **b**. The assignment **c =** to a variable which has been chosen to be non-replicated (a standard **short**) automatically votes and stores the result into **c**. This simple approach to TMR is made possible by using a set of software classes developed internally, one of which is partially visible in the fig.6.2 **TripleData**. A similar approach can also be applied to replicate and protect microcontroller and peripheral configuration register, by means of the specialized class **TripleConfigByte** shown in fig.6.2, which periodically and autonomously refreshes configuration registers starting from three replicas of register content which are hidden into the class, yet preserving the volatility of certain configuration bits. All the software of AraMiS, including all the drivers of the subsystems, has been developed using the **Hdata** package. The radiation hardening technique was tested by injecting random background faults (i.e. emulating single event upsets) in configuration registers and data memory cells. The results are quite promising and yet to be published. The presented technique is intrinsically more redundant because most of the functions are intrinsically triplicated [6].

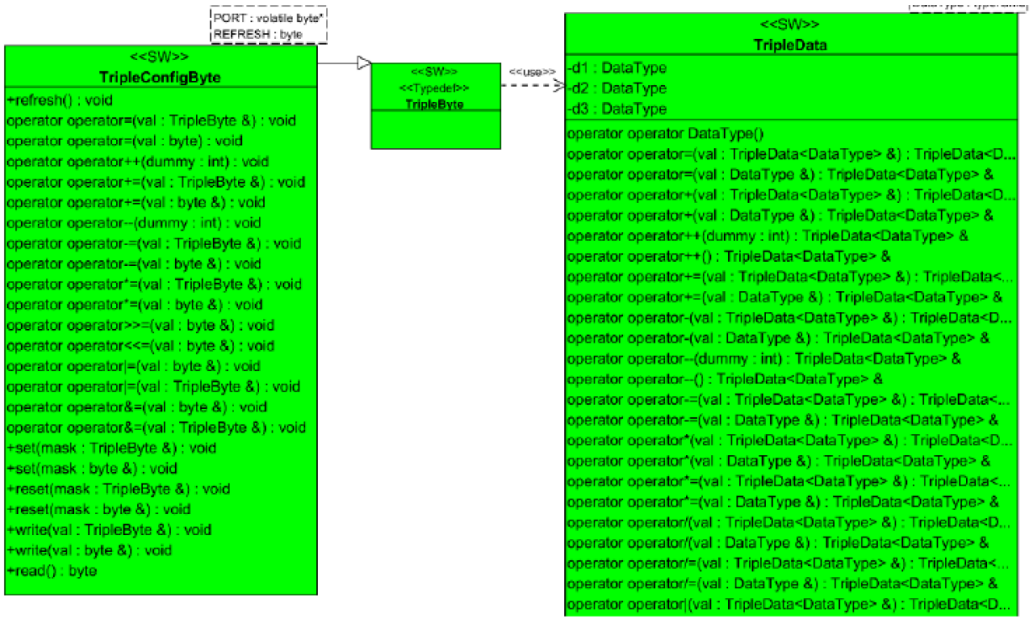

Figure 6.2: Software hardening class

Registers volatile bits

The software hardening has a **refresh()** function who is able to read all the registers of a peripheral and update the three local copy (see 6.2.1) accordingly. Some bits of these registers are continuously written by the micro-controller it self (i.e Interrupt Service Routine (ISR)) so these bits has to masked. This is done defining a mask for each register in the hardened header.

In fig.6.3 a complete use case diagram of the housekeeping management is reported.

6.2.2 Housekeeping

The housekeeping is responsible to get important data used to make statics, or to point out important event that can compromise the normal operation of the system. Using function as **getSatus()** it is possible to exchange sensible data like for example, in the case of the developed payload, the results of the tests described in 6.6. This is done because the protocol showed in 6.2.3, has an intrinsic communication timeout, so it is not possible to have the bus busy until a test is running (it can last too much). Furthermore, if the software is hardened (see 6.2.1) or not the needed time to get the results of a test is quite different.

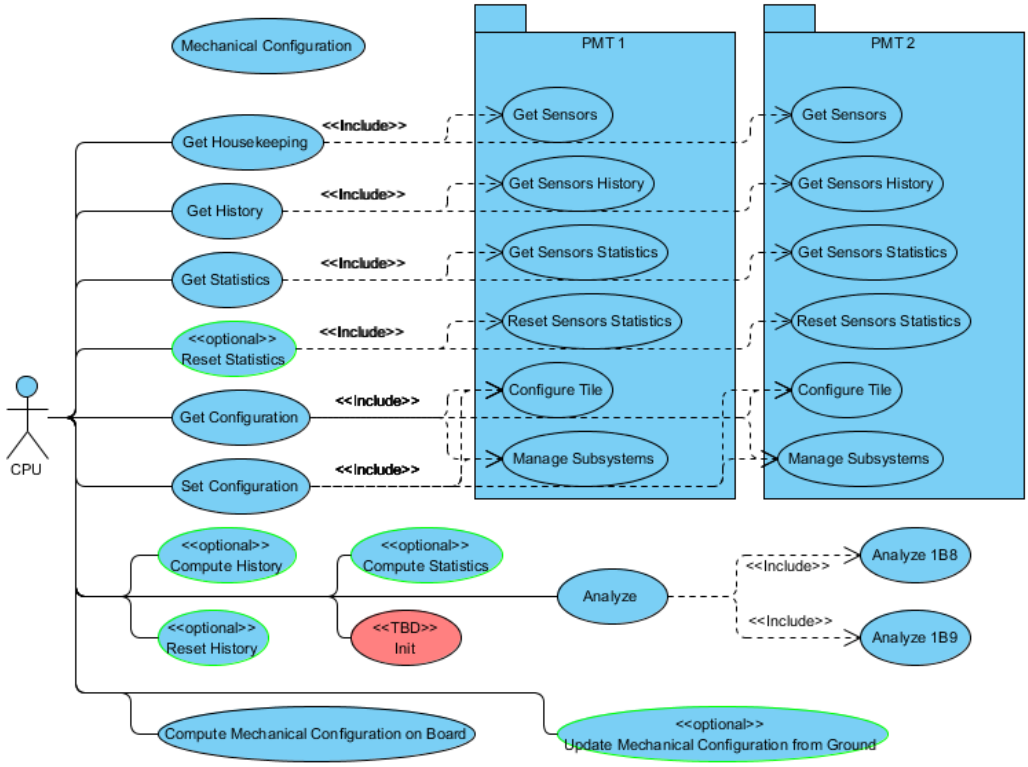

Figure 6.3: Housekeeping management use case diagram

6.2.3 Basic Communication Protocol

The used communication protocol is the 1B45 implemented in the AraMiS bus communication. A complete UML use case diagram of what the protocol can manage and who are the actors is reported in fig.6.4, while the used methods to ensure the communication between the OBC and the target are reported in fig.6.5

Let's identify the actors:

- Error indicator: a digital output indicating that an error has occurred. It is set to active whenever an error is detected like, for instance:
  - CRC error in messages from the CPU actor;
  - a nack received from the CPU actor;
  - no buffer available.
- Tester: the person (together with the appropriate equipment) in charge of testing

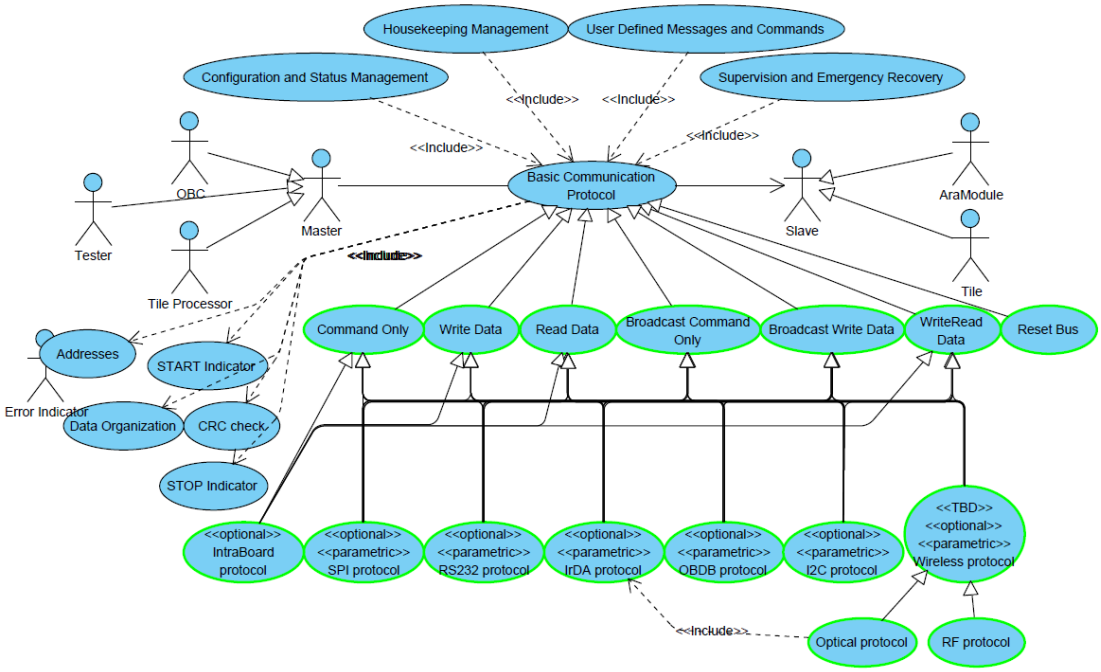

Figure 6.4: 1B45 complete basic communication protocol

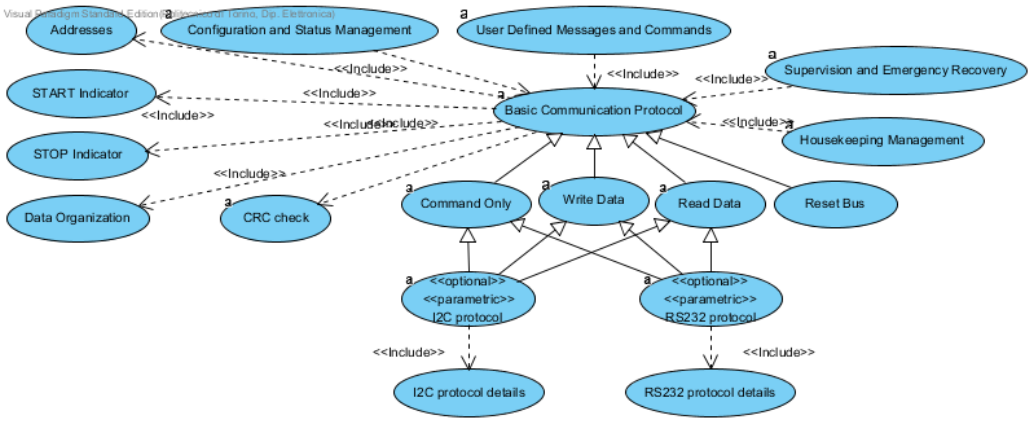

Figure 6.5: 1B45 used basic communication protocol

any part of the system (namely, both the subsystems, the tiles and the whole satellite);

- Tile processor: the processor hosted on each Tile, which is in charge of handling all Tile functions and subsystems;
- Master: any processor willing to communicate (exchange data and commands)

with a Slave. The Master communicates with the Slave via either of the foreseen protocols. The Master is communication master, that is, the element which starts communication, either read or write, while the Slave is the element which responds accordingly. Slave addressing depends on the protocol used. The Master, via the interface, can at least:

1. send Designer-defined messages to the Slave (namely, a Write Data operation);
2. receive Designer-defined messages from the Slave (namely, a Read Data operation);
3. send Designer-defined commands to the Slave (namely, a Command Only operation);
4. acquire Designer-defined housekeeping information from the Slave (e.g. internal voltages, currents, temperatures);
5. set the Slave to sleep mode or wake it up

- Slave: is any processor capable to communicate (exchange data and receive commands) with a Master, when requested by the latter. The Slave communicates with the Master using any of the foreseen protocols. The Master is communication master, that is, the element which starts communication, either read or write, while the Slave is the element which responds accordingly. Slave addressing depends on the protocol used. The Slave, via the interface, can at least:

  1. receive Designer-defined messages from the Master (namely, a Write Data operation);
  2. send Designer-defined messages to the Master (namely, a Read Data operation);
  3. receive Designer-defined commands from the Master (namely, a Command Only operation);
  4. acquire Designer-defined housekeeping information and send them to the Slave (e.g. internal voltages, currents, temperatures; see use case Housekeeping Management);

5. go to sleep mode or wake up, upon Master request.

- the AraModule is any module which can be plugged on on each Tile, containing individual elementary satellite functions;
- the Tile is a generic tile of the AraMiS satellite;

For what concern the various use cases instead:

- Addresses: the addresses of Master and Slaves must be unique in the whole system. They can range from 1 to 254 (0xFE). Address 255 (0xFF) is used by either Broadcast Command Only or Broadcast Write Data. Address 0 is forbidden. Address 1 is usually associated with Master. Addresses differ from Tile_ID, as the latter represents the type of Tile and usually there may be more than one Tile with the same Tile_ID in each spacecraft, while the Tile address must be unique.
- WriteRead Data: the protocol to issue a command which first writes between 1B to 256B of Designer's defined data from the Master to one Slave, then reads between 1B to 256B of Designer's defined data from the Slave to the Master, as defined by the Slave address. Up to 255 Slaves can be addressed separately. It is the Designer's responsibility to ensure that data to be read is already available in the Slave when the command is issued.
- Stop Indicator: Communication is terminated by the Master with a stop operation, which can either be:
  - IrDA protocol, OBDB protocol, RS232 protocol: no action. Data transfer is based on length of data. In case of packet corruption, the beginning of next packet is clearly defined by the following un-escaped STX data;
  - I2C protocol: it sends STOP flag (Master $\rightarrow$ Slave), that is, by rising SDA line when SCK is high (invalid data bit);
  - SPI protocol, it sends STOP flag (Master $\rightarrow$ Slave), that is, by rising CS line;
  - Wireless protocol: still TBD.

- Start Indicator: All data transfers are initiated by the Master only when the bus is free, that is, all previous communications are terminated or after an appropriate timeout (Max delay to slave answer (sends CRC (LSB) (Master → Slave) → returns slave ID (Slave → Master))) Communication starts with a START Indicator, which can either be:

  - IrDA protocol, OBDB protocol, RS232 protocol: it sends start byte (STX) (Master → 1B45), that is, ASCII STX (byte 0x02). Since STX identifies unequivocally the start of a new message, any subsequent data in the packet which matches STX shall be escaped (preceded) by an ASCII ESC (0x1B). Similarly a data which matches ESC shall also be preceded by another ESC. Therefore STX alone means start of transmission, the sequence ESC+STX means a data byte equal to STX (no start of message), the sequence ESC+ESC means a data equal to ESC. In an escaped sequence, the second byte must follow immediately (within at most max escape time) the ESC byte;
  - I2C protocol: it sends START flag (Master → 1B45), that is, by lowering SDA line when SCK is high (invalid data bit);
  - SPI protocol: it sends STOP flag (Master → Slave), that is, by rising CS line;
  - Wireless protocol: still TBD.

- Data Organization: When transferring data between the Master and the Slaves, data transfer takes places in bytes, according to data organization of most serial protocols. Only exception is the IntraBoard protocol, as this transfers data between elements of the same processor and requires no serial communication. The following criteria shall be followed to properly sequence data:

  - within each byte transfer, either the MSBit or the LSB it is transferred first, depending on the chosen low level protocol; see details on each of them to find out what endiannes is used;
  - when data longer that 1B have to be transferred, the least significant byte (LSByte) has to be transferred first. When stored, the LSByte has to be

stored at the lowest address. most Significant Byte (MSByte) has to be transferred last and stored at the highest address;

- for floating point data, the LSByte of mantissa is transmitted first (stored at lowest address) while exponent is transferred last (stored at highest address);
- for vectors, the first element (vector[0]) has to be transferred first (saved at lowest address(es));
- for matrices, the order is the row-wise, namely following: matrix[0][0], matrix[0][1], ..., matrix[0][N-1], matrix[1][0], matrix[1][1] etc.;
- for data shorter than 1B (e.g. bool), either two or more data are packed into 1B, or data are right-aligned. For instance a bool true is stored as a 0x01. In case of negative data, sign shall be extended as much as necessary, to maintain it when data is read byte-wise.

- Reset Bus: the protocol used to reset any on-going data exchange in case of failures. It can only be used by the Master in case of failures and fault recovery, as it interrupts any ongoing data exchange. The Master can Reset Bus by sending a STOP Indicator to the Slave. Since IrDA protocol, OBDB protocol and RS232 protocol do not have an explicit STOP Indicator, they cannot Reset Bus, so the Master shall wait until there is no more data traveling on the bus, that is, after an appropriate timeout (min bus timeout).

- CRC module: produces a signature for a given sequence of data values. The signature is generated through a feedback path from data bits 0, 4, 11, and 15 (see Figure 12-1). The CRC signature is based on the polynomial given in the CRC-CCITTBR polynomial

  Identical input data sequences result in identical signatures when the CRC is initialized with a fixed seed value, whereas different sequences of input data, in general, result in different signatures. Initial seed is 0xFFFF for the 1B45 Basic Communication protocol. Once all data have been processed through the CRC check, the value stored inside the register is added at the end of data for error protection.

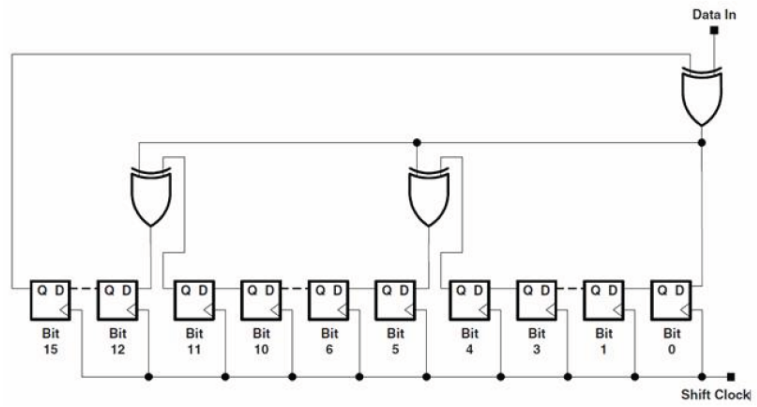

Figure 6.6: LFSR CRC-CCITT Standard, Bit 0 is the MSB of the result

- Configuration and Status Management: This is a group of use cases aimed at transferring either:

  – status information:

  statusRegister: CS_REDUNDANCY[LENGTH_STATUS] from one of many Tiles to the only Master;

  – configuration information:

  configRegister: CS_REDUNDANCY[LENGTH_CONFIG] to/from the only Master from/to one of many Tiles.

  As a rule of thumb, the status contains a 16-bits (model element not found) for each AraModule present on the Tile, plus one word (status[0]) for the status of Inter Tile Communication, plus any other mission-dependent status information which might be defined by the Designer. The word status[0] also contains lastError: t_LastError (shared by all AraModule) which stores additional details on the last error reported by each AraModule. The (optional) configuration contains a 16-bits (model element not found) for each AraModule present on the Tile, plus any other missiondependent status information which might be defined by the Designer. For other types of non-standard AraModules, the Designer shall specify both the meaning of each status and configuration bit and (when used) the lastError: t_LastError codes.

The Basic Communication Protocol supports communication between one Master (usually, either the OBC or the Tile Processor) and one or more Slave(s) (either a Tile

or an AraModule). The Designer shall make use as much as possible of the support from this protocol and its support for his specific module(s). It implements several basic functions for the AraMiS architecture, which are grouped in at least four groups:

1. Configuration and Status Management
2. Housekeeping Management
3. User Defined Messages and Commands
4. Supervision and Emergency Recovery

The protocol is then implemented in a number of physical implementations: SPI protocol, RS232 protocol, IrDA protocol, OBDB protocol, I2C protocol, Wireless protocol and IntraBoard protocol, which differ for the details of the physical support and data rate. The basic communication protocol is half-duplex that allows a communication between one and only one Master and one or more Slaves.

The following actions are supported:

- Write Data: when a Master wants to transfer up to 256B of data to a Slave;
- Read Data: when a Master wants to read up to 256B of data from a Slave;
- Command Only: when a Master wants to deliver a data-less command to a Slave;
- Broadcast Write Data: when a Master wants to transfer up to 256B of data to all Slaves;
- Broadcast Command Only: when a Master wants to deliver a data-less command to all Slaves;
- WriteRead Data: when a Master wants to transfer up to 256B of data bidirectionally to/from a Slave.

Most data transfers contain, from the Master to the Slave:

- an appropriate START Indicator;
- the nature of the START Indicator depends on the actual protocol chosen;

- an 8-bit Master address;
- an 8-bit Slave address to address a specific Slave; a 16-bits command;
- an 8-bit data length field (only for Write Data, WriteRead Data and Broadcast Write Data);
- data (1B to 256B; only for Write Data, WriteRead Data and Broadcast Write Data);
- a 16-bit CRC check. CRC algorithm is a CRC-16 of all bytes (including command/ID and length fields);
- an appropriate STOP Indicator; the nature of the STOP Indicator depends on the actual protocol chosen;

While from the Slave to the Master:

- an 8-bit Slave ID to identify the Slave type;
- an 8-bit data length field (only for Read Data and WriteRead Data);
- data (1B to 256B; only for Read Data and WriteRead Data);
- a 16-bit CRC check. CRC algorithm is a CRC-16 of all bytes (including command/ID and length fields)

If an error occurs (either wrong CRC or wrong length or no memory available, etc.):

- the Slave internally sets an ErrorFlag : bool and does not send any answer;
- by calling the Get Module Status use case, a Slave can read details on the last error and clear the **ErrorFlag : bool**.

6.3 Micro-controller's driver

The 1B521 Radiation Characterization Payload is the first AraMiS tile that uses a microcontroller different from the one used by all the other tiles. This is because the FRxxxx family has been introduced on the market since almost one year and so there

wasn't the possibility to include it before in the AraMiS project. For this reason, the first step of the development of the project was to write down and implement all the function that are used in the project for the used micro-controller. It has paid particular attention about the hardened header (see 6.2.1) because what was a **TripleConfigByte** in the previous used microcontrollers is a **TrypleConfigWord** in the MSP430FRxxxx family.

Here an overview about all the developed drivers and their functionalities will be given.

6.3.1 Ports

The ports driver are essentially used to configure the microcontroller's I/O as input/output port and enable the external interrupts request on the pins and port that have this feature. On the MSP430FR6989 an external interrupt can be placed on each pin of the port from 1 to 4.

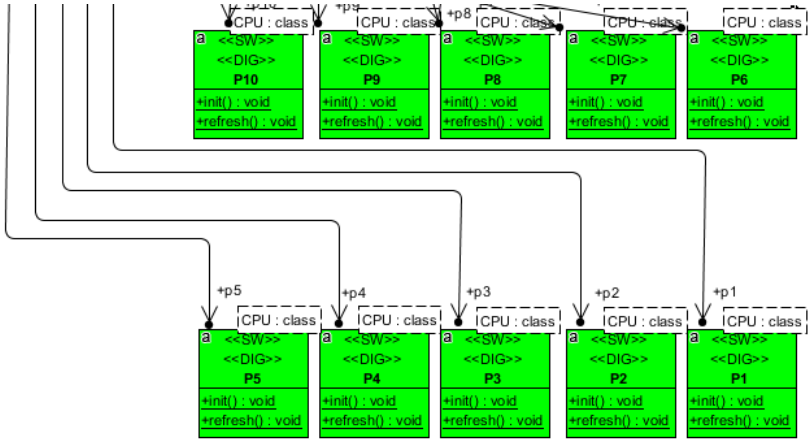

Figure 6.7: Ports P1 to P10 class

init()

This function is used to associate the microcontroller's registers associated with this peripheral to the respective hardened one.

refresh()

Refresh the configuration of the respective hardened copies of the configuration registers.

6.3.2 WDT

These drivers allow the user to use the Watch Dog Timer (WDT) functionalities. When WDT is enabled, the code shall periodically use the **reset() : void** periodically before the deadline defined by **configure(period : byte) : void)** function is reached, otherwise the WDT reaches end of count and reboots CPU. When WDT is disabled, there is no need to periodically **reset() : void** and CPU execution continues without reboots.

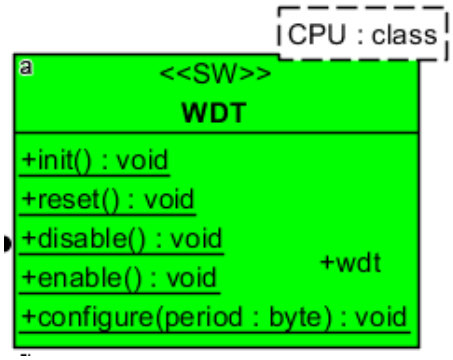

Figure 6.8: WDT class

init()

Initialize all the peripheral used data.

reset()

Resets WDT. If it is enabled, **reset() : void** must be called by the user periodically, namely before the WDT triggers a reboot.

disable()

Stop the WDT operation.

enable()

Start the WDT operation.

configure()

Configures WDT for counting SMCLK ticks depending on the value of period (0-3).

6.3.3 CRC16

These drivers allow the user to use the CRC functionalities. The CRC module produces a signature for a given sequence of data (see 6.2.3).

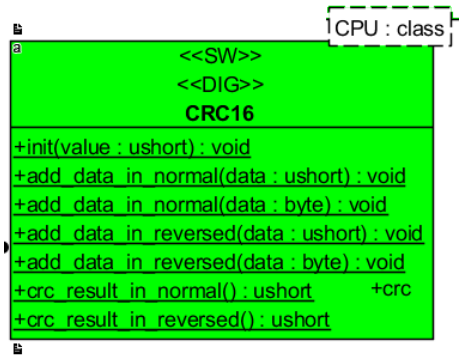

Figure 6.9: CRC class

init()

Initialize all the peripheral used data.

add_data_in_normal()

Add data to compute the signature starting from Most Significant Bit (MSB).

add_data_in_reversed()

Add data to compute the signature starting from Least Significant Bit (LSB).

crc_result_in_normal()

Get the signature starting from the MSB.

crc_result_in_reversed

Get the signature starting from the LSB.

6.3.4 RTC

These drivers allow the user to use the Real Time Clock (RTC) functionalities. It is a clock that keeps track of the current time. It can be used to program periodic actions and works using an external clock source crystal at the frequency of 32768Hz.

Figure 6.10: RTC class

init()

Initialize all the peripheral used data and set the desired date and hour in the format yyyy/mm/dd/wd, hh:mm:ss.

reset()

Reset data to the default configuration equal to 2015/01/01/1 00:00:00.

start()

Starts the RTC counting.

disable()

Disable the RTC counting.

setYear()

Change the stored year.

setMonth()

Change the stored month.

setDay()

Change the stored day.

setDayOfTheWeek()

Change the stored day of the week.

setDayAlarm()

Allow to generate an interrupt when the chosen day is matched.

setHourAlarm()

Allow to generate an interrupt when the chosen hour is matched.

setMinutesAlarm()

Allow to generate an interrupt when the chosen minutes are matched.

readYear()

Read the stored year value.

readMonth()

Read the stored month value.

readDay()

Read the stored day value.

readDayOfTheWeek()

Read the stored day of the week value.

readHour()

Read the stored hour value.

readMinutes()

Read the stored minutes value.

readSeconds()

Read the stored seconds value.

6.3.5 Timer A0, Timer A1, Timer B0, Timer B1

These drivers allow the user to operate with the four timers module that are present on the microcontroller. We will focus our discussion only on one timer (TIMERA0) because the operation is the same and the code is differentiated one each other only by the registers name and the associated port as can be easily seen in fig.6.11.

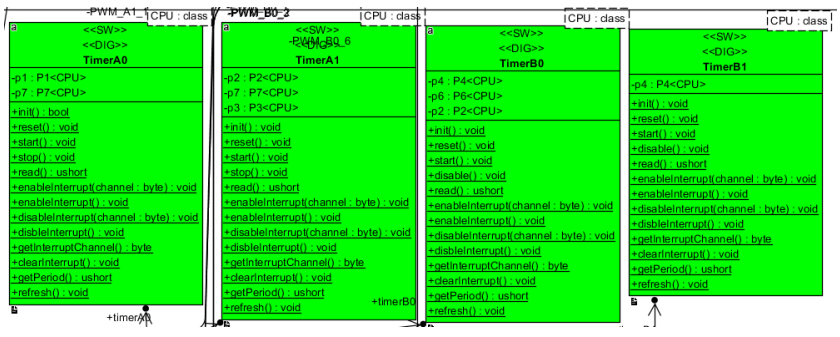

Figure 6.11: Timer A0, Timer A1, Timer B0, Timer B1 classes

init()

Initializes Timer to have clock from SMCLK, to go in stop mode, no capture, and to generate interrupt (when enabled) at rate **1e6/CPU::TIMERA0_ PERIOD** Hz; interrupt disabled. After **init()**, the user shall call start() in order to activate the timer.

For MSP_430_Family, **1e6/CPU::TIMERA0_ PERIOD < CPU::SMCLK_FREQ / 2e16 / 8**. It usually returns true, except when timer period is not achievable with the given clock frequency.

reset()

Resets Timer counting and all pending interrupt flags. It does not affect whether the timer is running or not, it only resets its count value. If timer is running, the first interrupt is generated at the end of counting.

start()

Starts (or restarts) Timer counting by setting it in up mode. Count value is not reset.

stop()

Stops Timer counting by setting it in stop mode. Count value is not affected.

read()

Reads Timer counting. Counting starts from 0, therefore an immediate call to **read()** right after **reset()** will return either 0 or a very small number.

enableInterrupt()

Enables the following interrupts of Timer:

- capture/compare channel 0, for channel=0; this will trigger interrupt vector TIMER0 _A0_VECTOR;
- capture/compare channel 1/2/3/4/5/6, for channel = 1/2/3/4/5/6, respectively. This will trigger interrupt vector TIMER0_A1_VECTOR;
- timer overflow, for channel = 0x7; this will trigger interrupt vector TIMER0 _A1_VECTOR;

If an interrupt is pending, this is immediately triggered. Routine **getInterruptChannel()** can be used to discriminate among capture/compare channels 1 through 7 and timer overflow.

disableInterrupt()

Disables interrupts of Timer. If the interrupt is disabled within an interrupt service routine, the interrupt flag also has to be cleared.

getInterruptChannel()

Returns the channel of the currently pending interrupt associated with vector TIMER0 _A1_VECTOR:

- 0 for no interrupt pending;
- 1/2/3/4/5/6 for capture/compare channel 1/2/3/4/5/6, respectively. Note that capture/compare channel 0 is associated with a different interrupt vector (TIMER0 _A0_VECTOR);
- 7 for Timer overflow;

This routine returns a correct value independently of the interrupt being enabled or not. If the interrupt is enabled, the corresponding interrupt service routine is also called. The latter shall explicitly clear interrupt flag (**clearInterrupt()**) before exiting, otherwise the same routine will be improperly called again upon exit.

clearInterrupt()

Clears interrupt flag. Must be called at the end of the interrupt service routine, otherwise interrupt service routine is called endlessly. If other interrupts are pending, another interrupt service routine is immediately triggered.

getPeriod()

Returns period of interrupt triggering, in us.

refresh()

Refreshes TMR variables against radiation-induced effects or other soft errors.

6.3.6 PWM_A0, PWM_A1, PWM_B0

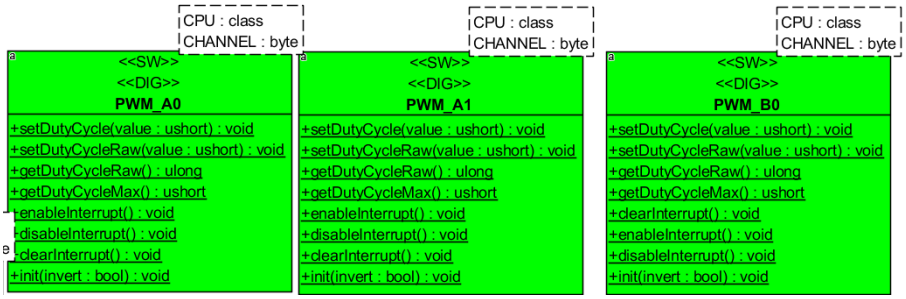

Figure 6.12: PWM_A0, PWM_A1, PWM_B0 classes

init()

Initializes output pin for CHANNEL of PWM_A0. The associated Timer MUST be already initialized. When invert is false, the PWM output is not inverted, namely a 10% dutycycle keeps output high 10% of the time and 90% low. When invert is true, the PWM output is inverted, namely a 10% dutycycle keeps output low 10% of the time and 90% high.

setDutyCycle()

Sets duty cycle of output signal of channel number CHANNEL of PWM_A0 as close as possible to the value given by value/2e16. For instance, when:

- value = 0, duty cycle = 0;
- value = 0x7FFF, duty cycle = 0.5;
- value = 0xFFFF, duty cycle = 1;

setDutyCycleRaw()

Sets duty cycle of output signal of channel number CHANNEL of PWM_A0 in raw format, between 0 and the max count value of the associated Timer (which is available using **getDutyCycleMax()** function). For instance, when:

- value = 0, duty cycle = 0;
- value = 0.5 *getDutyCycleMax() : ushort, duty cycle = 0.5;
- value = getDutyCycleMax() : ushort, duty cycle = 1;

getDutyCycleRaw()

Returns actual duty cycle (in raw format) of output signal of channel number CHANNEL of PWM_A0. For instance, when:

- duty cycle == 0, returns 0;
- duty cycle == 0.5, returns 0.5 *getDutyCycleMax();
- duty cycle == 1, returns getDutyCycleMax();

getDutyCycleMax()

Returns the max value for the **setDutyCycleRaw(value)** function.

enableInterrupt()

Enables interrupt when Timer count has reached the value set by either **setDutyCycle(value)** or **setDutyCycleRaw(value)** for output CHANNEL.

disableInterrupt()

Disables interrupt when Timer count has reached the value set by either **setDutyCycle(value)** or **setDutyCycleRaw(value)** for output CHANNEL.

clearInterrupt()

Clears interrupt flag. Must be called at the end of the interrupt service routine, otherwise interrupt service routine is called endlessly. If other interrupts are pending, another interrupt service routine is immediately triggered.

6.3.7 Processor

These drivers are essentially used to set the different Low Power Modes (LPM) of the micro-controller.

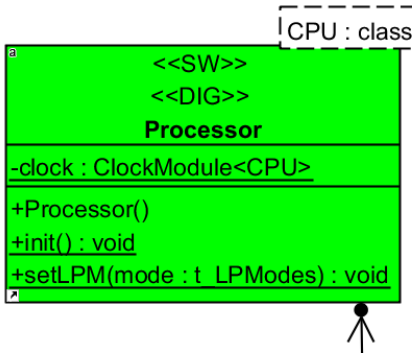

Figure 6.13: Processor class

init()

Initialize all the peripheral used data.

setLPM()

Sets processor into one of the LPM.

6.3.8 ADC

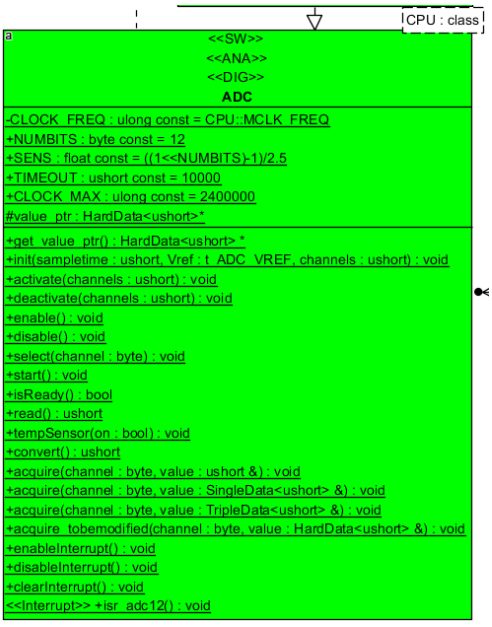

Figure 6.14: ADC class

init()

Initializes ADC for all following conversions. Arguments are:

- sampletime : indicates the desired sample time of the sample&hold circuit, in microseconds. Only a limited number of values are permitted, depending on the processor and its clock frequency;
- Vref : the source of reference voltage;
- channels : a word to specify which ADC channels are used: each bit is associated with a different channel; bit 0 with channel 0, up to bit 15 which is associated with channel 15. Each bit shall be set/reset to activate/deactivate the corresponding channel, respectively.

activate()

Activates one or more ADC channels for all following conversions. The ADC must already be initialized before calling this **activate(channels)**. Arguments are:

- channels : a word to specify which ADC channels are to be activated: each bit is associated with a different channel; bit 0 with channel 0, up to bit 15 which is associated with channel 15. Each bit shall be set to activate the corresponding channel. All other channels are not touched.

deactivate()

Deactivates one or more ADC channels for all following conversions.

The ADC must already be initialized before calling this **activate(channel)**.

Arguments are:

- channels : a word to specify which ADC channels are to be deactivated: each bit is associated with a different channel; bit 0 with channel 0, up to bit 15 which is associated with channel 15. Each bit shall be set to activate the corresponding channel. All other channels are not touched.

enable()

Turns on ADC and its reference. Conversion is NOT started. It shall be started with either **start()** or **acquire(channel, value)** or equivalent functions.

disable()

Turns off ADC and its reference and disables conversion.

select()

Selects the input channel to the ADC, for all following conversions. The input channel is identified by argument channel (from 0 to a Processor-dependent value; often either 0-7 or 0-15).

NOTE: the use of **select(channel) + start() + read()** operations is NOT compatible with the use of **acquire(channel, value)** operation.

start()

Holds input voltage from the last chosen channel, starts conversion and exits immediately. The channel to convert must be previously selected by means of the **select(channel)** operation. The user shall then wait until the **isReady()** operation returns true before calling **read()**, otherwise unpredictable results may occur.

NOTE: the use of **select(channel) + start() + read()** operations is NOT compatible with the use of **acquire(channel, value)** operations.

isReady()

Returns true when the ADC has terminated conversion; false otherwise. Reading converted value does not reset the returned conversion status.

read()

Returns converted data from ADC, in the range 0 to 2eNUMBITS-1. 0V converts to 0, while full scale converts to 2eNUMBITS-1. Full scale depends on the Vref parameter in **init(sampletime, Vref, channels)**. Data conversion should have been started by

means of the **start()** operation. The user shall then wait until the **isReady()** operation returns true before using this operation, otherwise unpredictable results may occur.

NOTE: the use of **select(channel) + start() + read()** operations is NOT compatible with the use of **acquire(channel, value)** operations.

tempSensor()

Enable the internal temperature sensor.

convert()

Holds input voltage from the last chosen channel, starts conversion, waits until end of conversion, then returns converted value. Waiting is interrupted after TIMEOUT const internal units (actual delay is not predictable).

NOTE: the use of **convert()** operation is NOT compatible with the use of **acquire(channel, value)** operations.

acquire()

Starts hold and conversion of input channel defined by the channel parameter. It enables interrupt for ADC. At the end of conversion, ADC automatically calls interrupt service routine **isr_adc12()** to transfer converted result into the location defined by the parameter value.

enableInterrupt()

Enables interrupt at the end of ADC conversion.

disableInterrupt()

Disables interrupt at the end of ADC conversion and clears the corresponding interrupt flags.

clearInterrupt()

Clear the interrupt flags.

6.3.9 Clock Module

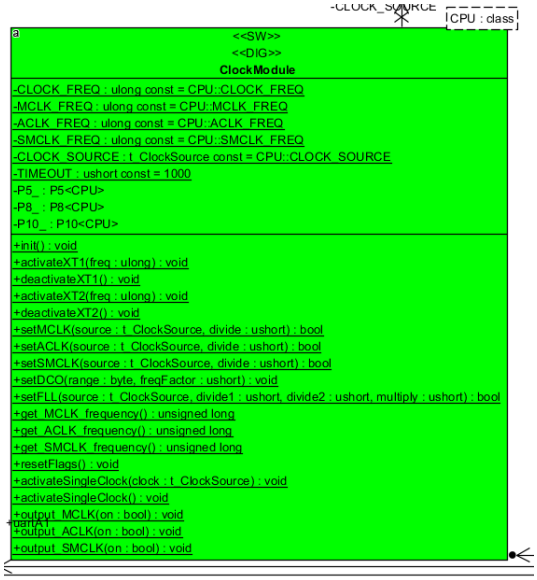

Figure 6.15: Clock Module class

init()

Initialize all the peripheral used data.

activateXT1()

Activates first oscillator (XT1) and configures it to operate at frequency freq, in Hz.

deactivateXT1()

Deactivates first oscillator (XT1).

activateXT2()

Activates second oscillator (XT2) and configures it to operate at frequency freq, in Hz.

deactivateXT2()

Deactivates first oscillator (XT2).

setMCLK()

Selects source for MCLK (CPU main clock) from the source defined by argument source and sets clock division factor to divide. Clock frequency (in Hz) is stored into attribute MCLK_FREQ. Argument divide must be a power of 2; otherwise, the function returns false; the function returns false also for unsupported clock sources or in case of clock faults. In all other cases, it returns true.

setACLK()

Selects source for ACLK (CPU auxiliary clock) from the source defined by argument source and sets clock division factor to divide. Clock frequency (in Hz) is stored into attribute ACLK_FREQ. Argument divide must be a power of 2; otherwise, the function returns false; the function returns false also for unsupported clock sources or in case of clock faults. In all other cases, it returns true.

setSMCLK()

Selects source for SMCLK (CPU secondary main clock) from the source defined by argument source and sets clock division factor to divide. Clock frequency (in Hz) is stored into attribute SMCLK_FREQ. Argument divide must be a power of 2; otherwise, the function returns false; the function returns false also for unsupported clock sources or in case of clock faults. In all other cases, it returns true.

setDCO()

Configures DCO to operate in frequency range given by parameter range, with frequency factor given by parameter freqFactor. For further details, read the processor manuals.

get_MCLK_frequency()

Returns frequency of MCLK, in Hz.

get_ACLK_frequency()

Returns frequency of ACLK, in Hz.

get_SMCLK_frequency()

Returns frequency of SMCLK, in Hz.

resetFlags()

Reset all the faults flags.

activateSingleClock()

Activate a specific clock indicating the source.

output_MCLK()

If on==true, MCLK is output on the corresponding pin. If on==false, MCLK is not output and the corresponding pin is available as a normal IO, configured as an input.

output_ACLK()

If on==true, ACLK is output on the corresponding pin. If on==false, ACLK is not output and the corresponding pin is available as a normal IO, configured as an input.

output_SMCLK()

If on==true, SMCLK is output on the corresponding pin. If on==false, SMCLK is not output and the corresponding pin is available as a normal IO, configured as an input.

6.3.10 UART A0, UART A1

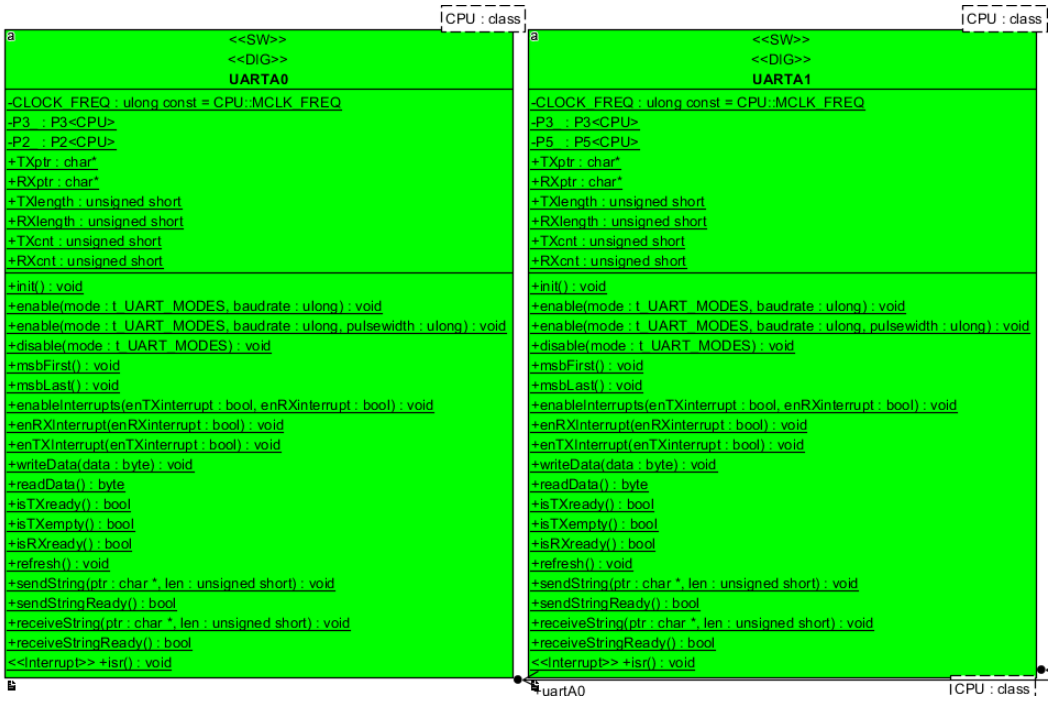

Figure 6.16: UART A0, UART A1 classes

init()

Initializes UARTA0. Clears all registers. Stops any ongoing transmission. Disables interrupts.

enable()

Activates communication using the protocol defined by mode, with baudrate defined by baudrate and 8-bit data width. It also configures I/O pins accordingly. No action starts until data is either sent or received.

disable()

Activates communication using the protocol defined by mode, with baudrate defined by baudrate and 8-bit data width. It also configures IrDA pulse width to pulsewidth. It also configures I/O pins accordingly. No action starts until data is either sent or received.

msbFirst()

Configures the UART to transit MSB first.

msbLast()

Configures the UART to transit MSB lasst.

enableInterrupts()

Enables (if true) or disables (if false):

- UART TX interrupt (enTXinterrupt);
- UART RX interrupt (enRXinterrupt);

writeData()

It reads and returns received data from UART data register. The UART has to be enabled to allow reception (first call to **init()**, then to **enable(mode, baudrate)**). The caller shall first check that a byte has been received and not yet read, by means of the **isRXready()** operation, which must return true. If not, the user shall wait. If no data has been received yet, it returns an unpredictable value.

The user shall therefore first verify that transmitter is ready by means of **isTXready()** which must return true. If not, the user shall wait.

readData()

It reads and returns received data from UART data register. The UART has to be enabled to allow reception (first call to **init()**, then to **enable(mode, baudrate)**). The caller shall first check that a byte has been received and not yet read, by means of the **isRXready()** operation, which must return true. If not, the user shall wait. If no data has been received yet, it returns an unpredictable value.

The user shall therefore first verify that transmitter is ready by means of **isTXready()** which must return true. If not, the user shall wait.

isTXready()

Returns true if transmitter buffer is ready to receive a new byte for transmission; false otherwise.

isTXempty()

Returns true when all data in the TX buffer and in the TX shift register have been sent; false otherwise. The caller shall wait until this routine returns true before calling **disable(mode)**, otherwise data transmission gets interrupted before completion.

isRXready()

Returns true if receiver buffer is full, that is, a byte has been received but not yet read; false otherwise.

refresh()

Refreshes TMR variables against radiation-induced effects or other soft errors.

sendString()

Sends len bytes of the string of data pointed by ptr (independently of string termination), one byte at a time using the UART, which must be properly initialized and enabled (first call to **init()** then to **enable(mode, baudrate)**). This routine configures transmission, then it immediately exits. Transmission continues by means of the interrupt service routine **isr()**.

sendStringReady()

Returns true when the transmission initiated by **sendString(ptr, len)** is finished, that is, exactly len bytes have been transmitted.

receiveString()

Receives len bytes of data and stores them into the string pointed by ptr (independently of string termination), one byte at a time using the UART, which must be properly initialized and enabled (first call to **init()** then to **enable(mode, baudrate)**). This

routine configures reception, then it immediately exits. Reception continues by means of the interrupt service routine **isr()**.

receiveStringReady()

Returns true when the reception initiated by **receiveString(ptr, len)** is finished, that is, exactly len bytes have been received.

6.3.11 UART B0, UART B1

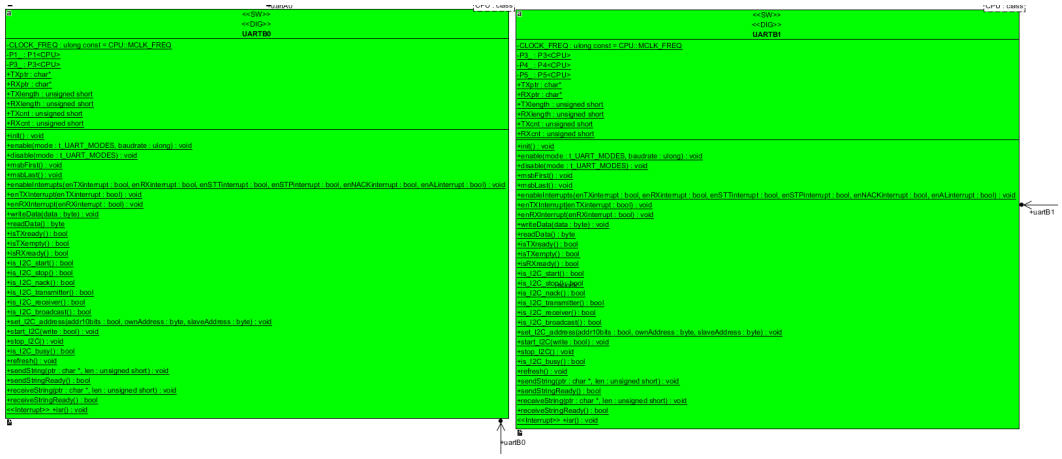

Figure 6.17: UART B0, UART B1 classes

init()

Initializes UARTB0. Clears all registers. Stops any ongoing transmission. Disables interrupts.

enable()

Activates communication using the protocol defined by mode, with baudrate defined by baudrate and 8-bit data width. It also configures I/O pins accordingly. No action starts until data is either sent or received.

disable()

Deactivates and aborts any ongoing data exchange. It also configures I/O pins to their normal digital I/O function, in particular as inputs. The mode must match the value used when enabling.

msbFirst()

Configures the UART to transit MSB first.

msbLast()

Configures the UART to transit LSB first.

enableInterrupts()

Enables (if true) or disables (if false):

- TX interrupt;
- RX interrupt;
- Start interrupt;
- Stop interrupt;
- Nack interrupt;
- Arbitrary lost interrupt;

writeData()

It writes the value of data into the UARTB0 data buffer. Data is automatically sent when next data exchange starts, that is: for the master, as soon as the previous transmission has terminated; for the SPI slave, as soon as the master sends its first clock bit. If no data is written in time, the UART sends an unpredictable value. If a new data is written to the buffer before the previous has been moved to transmission register, the previous data is lost. The user shall therefore first verify that transmitter is ready by means of **isTXready()** which must return true. If not, the user shall wait. The UART has to be enabled to allow transmission.

readData()

It reads and returns received data from UART data register. The UART has to be enabled to allow reception. The caller shall first check that a byte has been received and not yet read, by means of the **isRXready()** operation, which must return true. If not, the user shall wait. If no data has been received yet, it returns an unpredictable value. The user shall therefore first verify that transmitter is ready by means of **isTXready()** which must return true. If not, the user shall wait.

isTXready()

Returns true if transmitter buffer is ready to receive a new byte for transmission; false otherwise.

isTXempty()

Returns true when all data in the TX buffer and in the TX shift register have been sent; false otherwise.

isRXready()

Returns true if receiver buffer is full, that is, a byte has been received but not yet read; false otherwise.

is_I2C_start()

When the UART is enabled in I2C_SLAVE_MODE, it returns true if the last received byte (or double byte for 10-bit addressing) was the I2C addressing field (first byte or two bytes after I2C start flag) and the received address matches the slaveAddress argument of the last **set_I2C_address(addr10bits , ownAddress, slaveAddress)**. When the UART is either not enabled or enabled in any other mode, it returns false

is_I2C_stop()

When the UART is enabled in I2C_SLAVE_MODE, it returns true after reception of an I2C stop flag; false otherwise. When the UART is either not enabled or enabled in any other mode, it returns false

is_I2C_nack()

When the UART is enabled in I2C_MASTER_MODE, it returns true after reception of an I2C NACK (ACK bit = 1) from slave; false otherwise; When the UART is enabled in I2C_SLAVE_MODE, it returns true after reception of an I2C NACK (ACK bit = 1) from master; false otherwise. When the UART is either not enabled or enabled in any other mode, it returns false

is_I2C_transmitter()

When the UART is enabled in I2C_MASTER_MODE, it returns true when configured to transmit by **start_I2C()**; false otherwise. It also returns false when the master has lost arbitration; When the UART is enabled in I2C_SLAVE_MODE, it returns true upon reception of a valid START, I2C address matches slave address (as set by **set_I2C_address(addr10bits, ownAddress, slaveAddress)**) and master requests a read from the slave; false otherwise. When the UART is either not enabled or enabled in any other mode, it returns false

is_I2C_receiver()

When the UART is enabled in I2C_MASTER_MODE, it returns true when configured to receive by **start_I2C()**; false otherwise. It also returns true when the master has lost arbitration; When the UART is enabled in I2C_ SLAVE_MODE, it returns true upon reception of a valid START, I2C address matches slave address (as set by **set_I2C_address(addr10bits, ownAddress, slaveAddress)**) and master is writing into the slave; false otherwise. When the UART is either not enabled or enabled in any other mode, it returns false

is_I2C_broadcast()

When the UART is enabled in I2C_SLAVE_MODE, it returns true upon reception of a General Call address; false otherwise; When the UART is either not enabled or enabled in any other mode, it returns false

set_I2C_address()

Sets I2C address for both transmission and reception: if addr10bits is true, sets 10 bits addressing mode; is false, sets 7-bits addressing mode ownAddress defines the 7/10 bits addressing of the I2C master; address is right-aligned; the address can be any 7/10 bits value, except the I2C general address (0x0) slaveAddress defines the 7/10 bits addressing of the I2C slave; address is right-aligned; the address can be any 7/10 bits value, except the I2C general address (0x0) This function can not be called during an active I2C communication.

start_I2C()

Starts (or restarts) I2C communication from master side, by sending: a START bit for I2C; slave address (either 7 or 10 bits) plus transmit flag, with transmit flag set to either: write, if write==true or read, if write==false It does NOT wait until end of transmission. The user shall either wait for interrupt or poll **isTXready()** before sending first data byte.

stop_I2C()

Stops transmission and sends a STOP bit for I2C.

is_I2C_busy()

Check if the communication is hold by someone.

refresh()

Refreshes TMR variables against radiation-induced effects or other soft errors.

sendString()

Sends len bytes of the string of data pointed by ptr (independently of string termination), one byte at a time using the UART, which must be properly initialized and enabled. This routine configures transmission, then it immediately exits. Transmission continues by means of the interrupt service routine **isr()**.

sendStringReady()

Returns true when the transmission initiated by **sendString(ptr : char *, len)** : void is finished, that is, exactly len bytes have been transmitted.

receiveString()

Receives len bytes of data and stores them into the string pointed by ptr (independently of string termination), one byte at a time using the UART, which must be properly initialized and enabled (first call to (model element not found) then to (model element not found)). This routine configures reception, then it immediately exits. Reception continues by means of the interrupt service routine **isr()**.

receiveStringReady()

Returns true when the reception initiated by **receiveString(ptr : char *, len)** is finished, that is, exactly len bytes have been received.

6.4 Boot Strap Loader (BSL)

The developed payload uses the BSL micro-controller's functionality in order to reload the firmware if it is needed due to some unrecoverable fault. Here a quick overview about what a BSL is and how to use it will be given.

6.4.1 What is a BSL?

A BSL, also called boot-loader is the first program which executes (before the main program) whenever a system is initialized. It has a dedicated small section in the ROM of the controller, and is executed first when it is initialized. The boot-loader program can access any of inbuilt peripherals like USB, USART, CAN, SPI, etc. The incoming data data and this is used to write the non-volatile memory of the micro-controller. The boot-loader can be inserted into a controller by using an external or any conventional burner and then depending on the type of boot-loader the controller starts responding to the interface. So whenever the controller is reinitialized the program counter jumps to the boot-loader section and then it waits there for the instruction, which is fed from

external device. In case there is no boot-loader the program counter will go on 0000H (starting position of the non-volatile memory) and start executing the instructions which are written in the memory of the device.

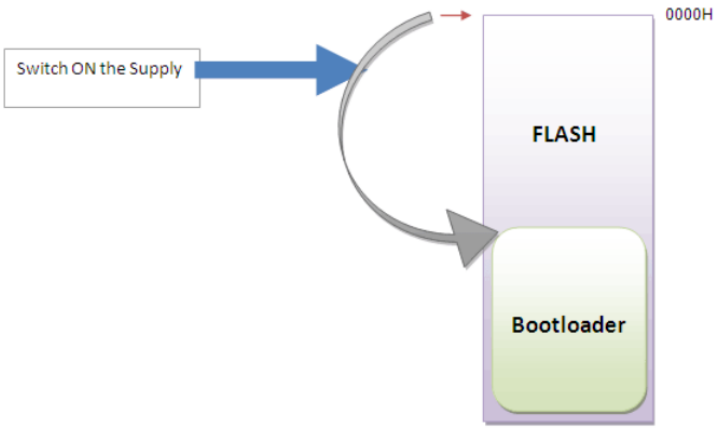

Figure 6.18: BSL operating principle

If the program counter enters the boot-loader section then after executing the it, there must be an instruction used to force the program counter to go to 0000H. Mostly the boot-loader resides in the bottom most area of the ROM but there are some cases it can be configured in the top.

6.4.2 BSL in MSP430FR6989

The BSL of the MSP430FR6989 has the following characteristics:

- Small footprint (1 to 3 flash sectors)
- Supports USI, USCI, and eUSCI peripherals
- Optional support for SMBus (protocol and clock timeout)
- Different communication protocols vary in complexity and size
- Different options allow for different levels of robustness
- Optional dual-image support
- Allows for use of all interrupts in application

- Configurable entry sequence
- Optional validation of application using CRC-8 or CRC-CCITT

A basic BSL program is provided by TI and resides in ROM at memory space 01000h through 017FFh. The BSL supports the commonly used UART protocol with RS232 interfacing, allowing flexible use of both hardware and software. To use it, a specific BSL entry sequence must be applied to the RST/NMI and $TEST$ pins. A correct entry sequence causes $SYSBSLIND$ to be set. A bootstrap-loading session can be exited by continuing operation at a defined user program address or by applying the standard reset sequence. Access to the device memory via the BSL is protected against misuse by a user-defined password. Two BSL signatures, BSL Signature 1 (memory location 0FF84h) and BSL Signature 2 (memory location 0FF86h) reside in FRAM and can be used to control the behavior of the BSL. Writing 05555h to BSL Signature 1 or BSL Signature 2 disables the BSL function and any access to the BSL memory space causes a vacant memory access[1]. Most BSL commands require the BSL to be unlocked by a user-defined password. An incorrect password erases the device memory as a security feature. Writing 0AAAAh to both BSL Signature 1 and BSL Signature 2 disables this security feature. This causes a password error to be returned by the BSL, but the device memory is not erased. In this case, unlimited password attempts are possible [9].

6.4.3 Device start-up sequence

After power up, the device first checks the BSL signature. If the appropriate values are there, the device calls the BSL protect function.

[1]Vacant memory is non-existent memory space. Accesses to vacant memory space generate a System Non-Maskable Interrupt (SNMI) when enabled (VMAIE = 1). Reads from vacant memory results in the value 3FFFh. In the case of a fetch, this is taken as JMP $. Fetch accesses from vacant peripheral space result in a Power Up Clear (PUC). After the boot code is executed, it behaves like vacant memory space and also causes an Non Maskable Interrupt (NMI) on access.

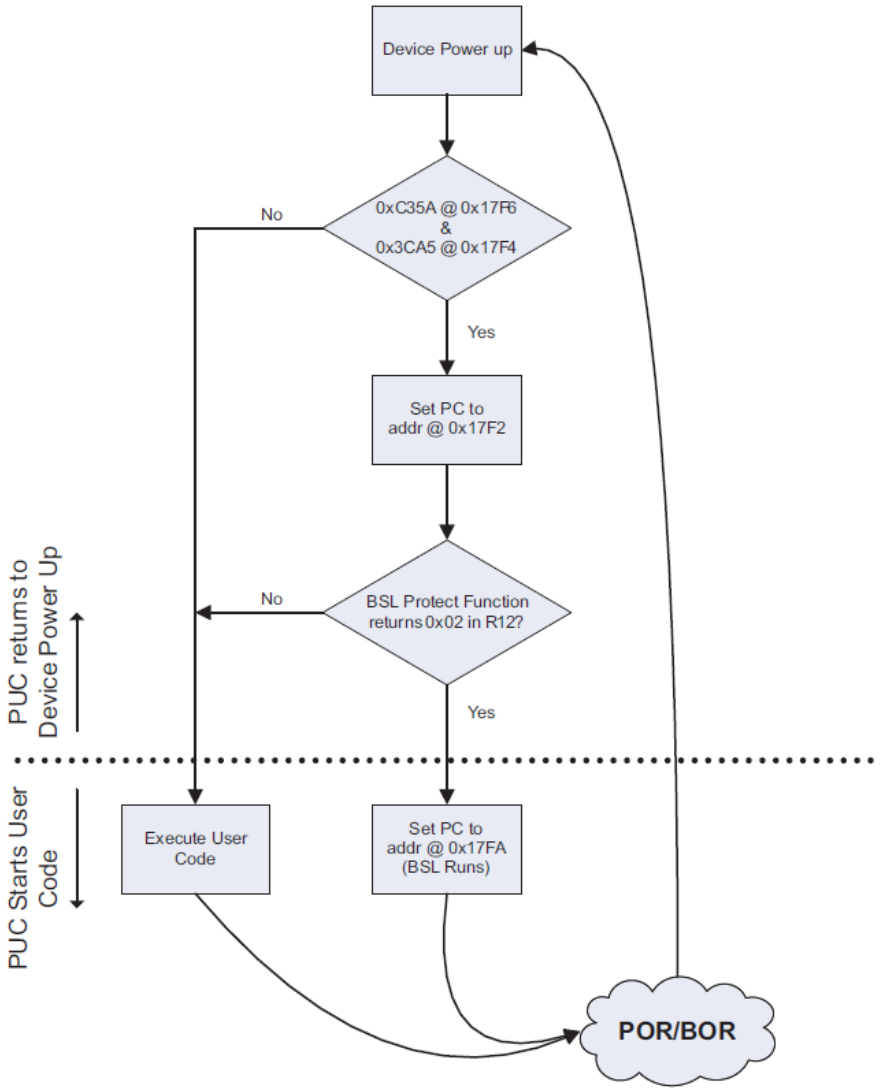

Figure 6.19: Device start-up sequence

6.4.4 BSL Protect Function

The BSL Protect function is called after each BrownOut Reset (BOR) and before user code is executed. There is no time and functionality limit placed on this code; however, if this function does not return, it renders the device totally unresponsive. Additionally, excessively long delays in the return functions could lead to problems during debug. At its most basic, the BSL Protect Function should perform two essential functions:

- Protecting the BSL memory;
- Determining whether the BSL or user code should be executed after exiting the

BSL Protect function

The BSL Protect function is called with the stack pointer set to a default location, which is dependent on the device used. Changing the stack pointer or manipulation of the stack pointer data values will most certainly lead to a unresponsive device. In this case nothing, not even reprogramming, will be possible anymore. On some devices the stack pointer space is very limited and extensive use of the stack pointer within the BSL Protect function could lead to memory overflows. To ensure proper behavior, the stack access should be limited or the stack pointer should be moved to another location. To make sure the device returns from the BSL Protect function correctly, the stack pointer needs to be restored before returning [9].

6.4.5 BSL entry sequence

Applying an appropriate entry sequence on the RST and TEST pins forces the microcontroller to start program execution at the BSL RESET vector located at the address FFFEh. The default position of this vector is FFFEh; it is used if TEST is kept low while RST rises from low to high like showed in fig.6.20.

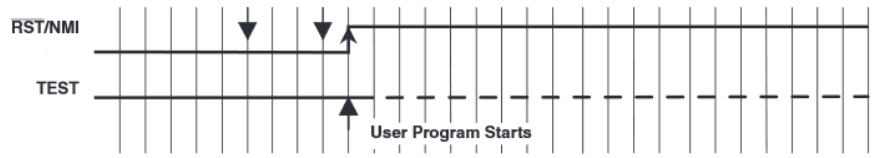

Figure 6.20: Standard RESET sequence

The BSL program execution starts when the TEST pin has received a minimum of two positive transitions and if TEST is high while RST rises from low to high like showed in fig.6.21.

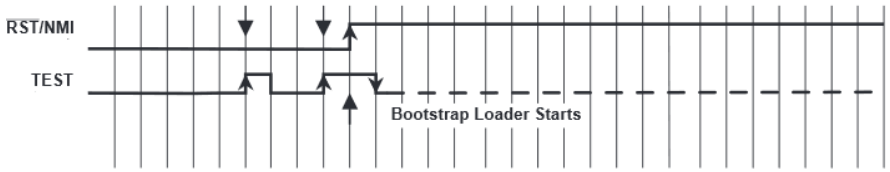

Figure 6.21: BSL entry sequence

6.5 FRAM partitioning and security

The developed firmware include a test of the memory (see 6.6.1). Since we want to test the program and the data memory, independently one from the other, a partitioning is suggested.

6.5.1 Memory organization

Fig.6.22 shows how the memory of the used microcontroller is organized.

		MSP430FRxxx9(1)	MSP430FRxxx8(1)	MSP430FRxxx7(1)	MSP430FRxxx6(1)
Memory (FRAM) Main: interrupt vectors and signatures Main: code memory	Total Size	127 KB 00FFFFh–00FF80h 023FFFh–004400h	95 KB 00FFFFh–00FF80h 01BFFFh–004400h	63 KB 00FFFFh–00FF80h 013FFFh–004400h	47 KB 00FFFFh–00FF80h 0FF7Fh–004400h
RAM	Sect 1	2 KB 0023FFh–001C00h	2 KB 0023FFh–001C00h	2 KB 0023FFh–001C00h	2 KB 0023FFh–001C00h
Boot memory (ROM)		256 B 001BFFh–001B00h	256 B 001BFFh–001B00h	256 B 001BFFh–001B00h	256 B 001BFFh–001B00h
Device Descriptor Info (TLV)		256 B 001AFFh–001A00h	256 B 001AFFh–001A00h	256 B 001AFFh–001A00h	256 B 001AFFh–001A00h
Information memory (FRAM)	Info A	128 B 0019FFh–001980h	128 B 0019FFh–001980h	128 B 0019FFh–001980h	128 B 0019FFh–001980h
	Info B	128 B 00197Fh–001900h	128 B 00197Fh–001900h	128 B 00197Fh–001900h	128 B 00197Fh–001900h
	Info C	128 B 0018FFh–001880h	128 B 0018FFh–001880h	128 B 0018FFh–001880h	128 B 0018FFh–001880h
	Info D	128 B 00187Fh–001800h	128 B 00187Fh–001800h	128 B 00187Fh–001800h	128 B 00187Fh–001800h
Bootstrap loader (BSL) memory (ROM)	BSL 3	512 B 0017FFh–001600h	512 B 0017FFh–001600h	512 B 0017FFh–001600h	512 B 0017FFh–001600h
	BSL 2	512 B 0015FFh–001400h	512 B 0015FFh–001400h	512 B 0015FFh–001400h	512 B 0015FFh–001400h
	BSL 1	512 B 0013FFh–001200h	512 B 0013FFh–001200h	512 B 0013FFh–001200h	512 B 0013FFh–001200h
	BSL 0	512 B 0011FFh–001000h	512 B 0011FFh–001000h	512 B 0011FFh–001000h	512 B 0011FFh–001000h
Peripherals	Size	4 KB 000FFFh–000020h	4 KB 000FFFh–000020h	4 KB 000FFFh–000020h	4 KB 000FFFh–000020h
Tiny RAM	Size	26 B 000001Fh–000006h	26 B 000001Fh–000006h	26 B 000001Fh–000006h	26 B 000001Fh–000006h
Reserved (ROM)	Size	6 B 000005h–000000h	6 B 000005h–000000h	6 B 000005h–000000h	6 B 000005h–000000h

(1) All address space not listed is considered vacant memory.

Figure 6.22: MSP430FR6989 Memory Organization

What we have to divide is the FRAM memory so addresses from $0x23FFF$ to $0x004400$. The micro-controller also has 2kB of RAM that will also be under test.

6.5.2 Memory layout partitioning

FRAM does not require a pre-erase in which every write to FRAM is non-volatile. However, there are some minor trade-offs in using FRAM instead of RAM that may

apply to a subset of use cases. One of the differences, on the MSP430 platform, is the FRAM access speed. This is in fact limited to $8MHz$ (in order to avoid unwanted wait states), whereas, SRAM can be accessed at the maximum device operating frequency. Since FRAM memory can be used as universal memory for program code, variables, constants, stacks, and so forth, the memory has to be partitioned for the application in order to optimize power saving. IAR Embedded Workbench® for MSP430 IDEs, can be used to set up an application's memory layout to make best-possible use of the underlying FRAM depending on the application needs. These memory partitioning schemes are generally located inside the IDE-specific linker command file. By default, the linker command files will typically allocate variables and stacks into SRAM. And, program code and constants are allocated in FRAM. These memory partitions can however be moved or sized depending on the application needs.

Program code and constant data

Both program code and constant data should be allocated in FRAM just like it would be done in a standard FLASH based microcontroller. Furthermore, to ensure maximum robustness and data integrity, the MPU feature should be enabled for those regions such that they are protected against write accesses. Doing this prevents accidental modification that could result from possible errant write accesses to those memory regions in case of program failures (software crash), buffer overflows, pointer corruption, and other types of anomalies.

Variables

Variables are allocated in SRAM by the default linker command files.

MSP430FRxxxx devices would typically have 2KB of SRAM. If the variable size is too large to fit in SRAM, the linker command file can be modified or C-language **#pragma** directives can be used to allocate specific variables or structures in FRAM memory. Aside from SRAM memory constraint, another reason you would use FRAM for variables is to decrease start-up time.

Software stack

FRAM can be used for the stack in a typical application, it is recommended to allocate the stack in the on-chip SRAM because it can be accessed at full-speed with no wait-states independent of the chosen CPU clock frequency (MCLK). Since in most applications the stack is the most frequently accessed memory region, this helps ensure maximum application performance. Likewise, since SRAM memory accesses are even lower power than FRAM write accesses, allocating the stack in SRAM also yields to lower active power consumption numbers. Last but not least, the contents of the stack does not need to be preserved through a power cycle, in most if not all use cases, since the application code performs a cold start and re-initializes the basic C runtime context anyways.

6.5.3 Memory partitioning in IAR

The tool-chains available for MSP430 all ship with linker command files that define a default memory setup and partitioning, typically allocating program code and constant data into FRAM, and variable data and the system stack to SRAM [11]. C compilers have extensions that allow to locate selected variables and data structures into FRAM as described above ans so to utilize the benefits of using FRAM for persistent data storage without any further considerations regarding memory partitioning or modifications of the linker command files. Due to the nature of the FRAM being equally usable for both code, constant and variable data storage, the task of partitioning the memory can be typically left to the linker [11]. If the application data is for example located into FRAM through the use of compiler extensions, the space available for program code automatically reduces by the amount that is consumed by such variables, as the linker places all its output segments into the same "pool" of FRAM. It is also possible to limit certain linker sections to specific fixed memory regions in order to enable an easier manual setup of the MPU module. Customization of the memory partitioning typically involves making modifications to a project-specific linker command file that is based off the default file that ships with the IDE. While some changes may seem intuitive and obvious, it is highly recommended to obtain a good working knowledge of the linker and its command files by consult the linker documentation. Here a procedure that shows

how to do that will follow.

IAR linker configuration

In IAR, modifying the linker command (.xcl) file is not needed if variables are located in FRAM through the use of the __**persistent** attribute. However, if it is desired to locate variables declared as __**noinit** into FRAM, then a minor modification can be made to accommodate this by moving the **DATA16_N** and **DATA20_N** segment assignments in the linker command file from RAM into the FRAM region. If a customized linker command file is still required, a copy of lnk430xxxx.xcl needs to be made. The following steps outline how to create a custom IAR linker command file.

- Navigate ti the IAR installation directory folder;
- Make a copy of the link430xxxx.xcl file to your local project and rename the filename, if needed (a backup copy is always a good idea);
- Open the new copy of the .xcl file and customize it;
- Configure the IAR project to point to the customized linker command file like in fig.6.23;

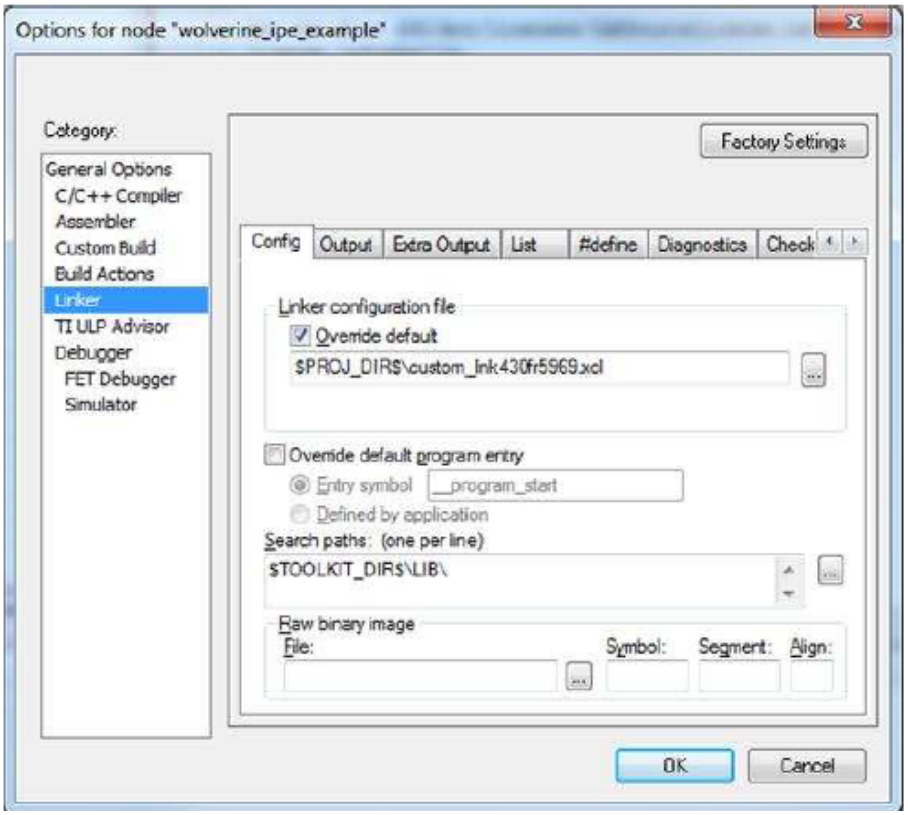

Figure 6.23: Override Linker Command File in IAR

6.5.4 Use compiler extensions for FRAM

This section outlines how to leverage built-in compiler extensions to locate specific variables in FRAM so that their values can be preserved during power cycles or periods of any length where the system is completely powered down. Locating variables in FRAM through either persistent or no-init mechanisms discussed here also helps to reduce the application wake-up time and with this its energy consumption as those variables will not get initialized by the C startup routine [11].

Compiler extensions in IAR

In IAR, two C language extension attributes named **__persistent** and **__no_init** are provided that facilitate the use of FRAM for data storage.

For persistent storage functionality in IAR, variables can be declared using the **__persistent** attribute. Variables declared with this attribute are allocated into the **DATA16_P** and **DATA20_P** linker memory segments, which the default IAR linker

command files (.xcl) automatically locate in FRAM. Below shows an example of a variable x declared such that it is not initialized during C startup and automatically allocated in FRAM memory. Furthermore, similar to the behavior in CCS, this variable only gets initialized by the debug tool chain during the initial code download but not at application startup or runtime.

```
1 __persistent unsigned int x = 5;
```

Similarly, no-init storage functionality also exists in IAR through the use of the **__no_init** attribute. Declaring variables with this attribute causes them to be allocated into the **DATA16_N** and **DATA20_N** linker memory segments. And also unlike in the case of **__persistent**, variables declared as **__no_init** will not get allocated into FRAM by default. If such functionality is required, a minor modification to the linker command file is needed [11].

```
1 __no_init unsigned int x = 5;
```

6.5.5 FRAM protection and security

FRAM is easy to write so application code, constants, and some variables residing in FRAM need to be protected against unintended writes that may result from invalid pointer accesses, buffer overflows, and other anomalies that could potentially corrupt your application. It is available a built-in MPU that monitors and supervises memory segments as defined in software to be protected as read, write, execute or a combination of them. Before protecting the memory, the FRAM memory needs to be partitioned. To partition, understanding the program size and types of memory segments after program linking is important to decide how each memory segments are protected. This information is generally located in the project map file that is generated during application build and gets populated to an IDE-specific output folder. The following sections describe an example of how variables, constants, and program code can be protected using the MPU. The configuration can be performed automatically by the MPU, or can be done manually for maximum flexibility.

Linker MAP file in IAR

The first step is to analyze the linker-generated map file in order understand the start and size of the memory segments that constitute the application firmware image: **constants**, **variables**, **no-init**, **persistent**, and **program code**. IAR does not generate the map file by default. This feature needs to be enabled by checking *Generate linker listing* box under the *Project Options*, as shown in fig.6.24.

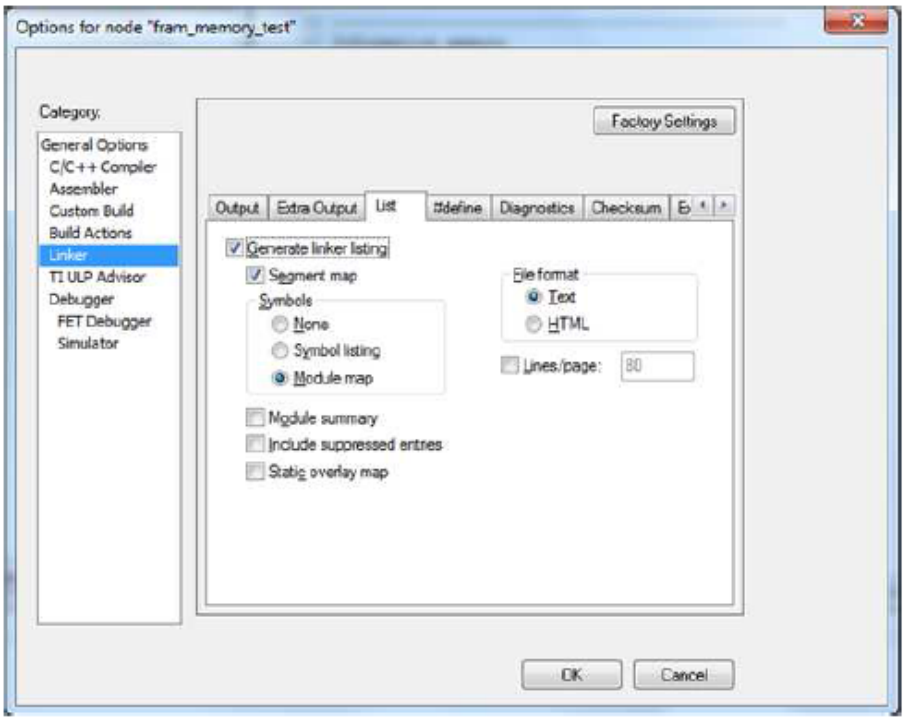

Figure 6.24: Generate MAP file in IAR

If enabled, the map file scan be found in the project after a compile process. Open up the map file and analyze it for the following segment names.

Segment Name	Memory Region	Recommended Protection Type (if in FRAM)
DATAxx_Z	Data initialized to zero	Read and Write
DATAxx_I	Initialized data	Read and Write
DATAxx_N	Data defined using `__no_init`	Read and Write
DATAxx_P	Data defined using `__persistent`	Read and Write
DATAxx_HEAP	Heap used by 'malloc' and 'free'	Read and Write
DATAxx_C	Constants	Read only
CODE	Program Code	Read and Execute

Figure 6.25: MAP file example

Manual MPU configuration in IAR

The MPU can be configured to protect three different memory segments in software. Each segment can be individually configured to read, write, execute, or a combination of them. Most applications would have some form of variables that should be protected as read and write, constants to be read only, and program code should be read and execute only.

Memory Region	Protection Type	MPU Segment
Variables	Read and Write	Segment 1
No-init	Read and Write	Segment 1
Persistent	Read and Write	Segment 1
Constants	Read only	Segment 2
Program Code	Read and Execute	Segment 3

Figure 6.26: MPU memory segmentation

Once the starting address for the application's read and write, read only, and read and execute segment has been identified from the generated map file. Now it is time to determine and configure the segment boundaries for the MPU. Do keep in mind that the smallest MPU segment size allocation is 1KB or 0x0400. In now proposed an example in which the application uses only 5-bytes of constant array, 2-bytes used for persistent variable, and remainder is application code. Therefore, the linker should allocate this example application in which 1KB for variables and 1KB for constants.

Memory Region	Protection Type	MPU Segment	Example Memory Partition
Variables	Read and Write	Segment 1	0x4400 – 0x47FF
Constants	Read only	Segment 2	0x4800 – 0x4BFF
Program Code	Read and Execute	Segment 3	0x4C00 – 0xYYYYY

Figure 6.27: MPU memory segmentation example

Once the memory segmentation has been decided for segment 1, 2, and 3 there are two registers to define how the segment boundaries are configured: Memory Protection Unit Segmentation Border 1 (**MPUSEGB1**) and Memory Protection Unit Segmentation Border 2 Register (**MPUSEGB2**). Before writing to the register, the address needs to be shifted to the right by 4 bits.

In IAR a new C-file has to be created with the name **low_level_init.c**. This file would need to be included in your project. IAR's equivalent function to enable the execution

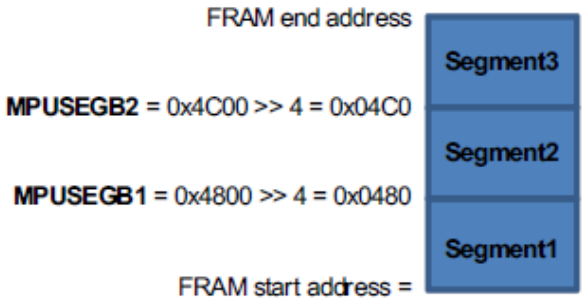

Figure 6.28: Address to be written in MPUSEGBx registers

of application code as soon as the device starts up is **int __low_level_init(void)**. The following code snippet example shows an equivalent MPU configuration for IAR.

```
1
  #include "msp430.h"
3 int __low_level_init(void)
  {
5 //Insert your low-level initializations here
  WDTCTL = WDTPW+WDTHOLD;
7
  //Configure MPU
9 MPUCTL0 = MPUPW;   //Write PWD to access MPU registers
  MPUSEGB1 = 0x0480; //B1 = 0x4800; B2 = 0x4C00
11 MPUSEGB2 = 0x04c0; /Borders are assigned to segments

13 /* Segment 1 { Allows read and write only
     Segment 2 { Allows read only
15   Segment 3 { Allows read and execute only*/

17 MPUSAM = (MPUSEG1WE | MPUSEG1RE | MPUSEG2RE | MPUSEG3RE |
     MPUSEG3XE);
  MPUCTL0 = MPUPW | MPUENA | MPUSEGIE; /* Enable MPU protection */
19
  /* MPU registers locked until BOR
21
  * Return value:
```

```
23  *
    * 1 - Perform data segment initialization.
25  * 0 - Skip data segment initialization.
    */
27
    return 1;
29  }
```

IDE WizardBased MPU configuration in IAR

IAR's IDE option of configuring the MPU via the MPU Wizard is located under **Project Options** $->$ **General Options** $->$ **MPU/IPE** as depicted in fig.6.29. Enable the MPU by checking the Support MPU box. Once enabled, the IAR toolchain automatically determines which segments are code, constants, and variables to establish how the MPU partitions should be configured [11].

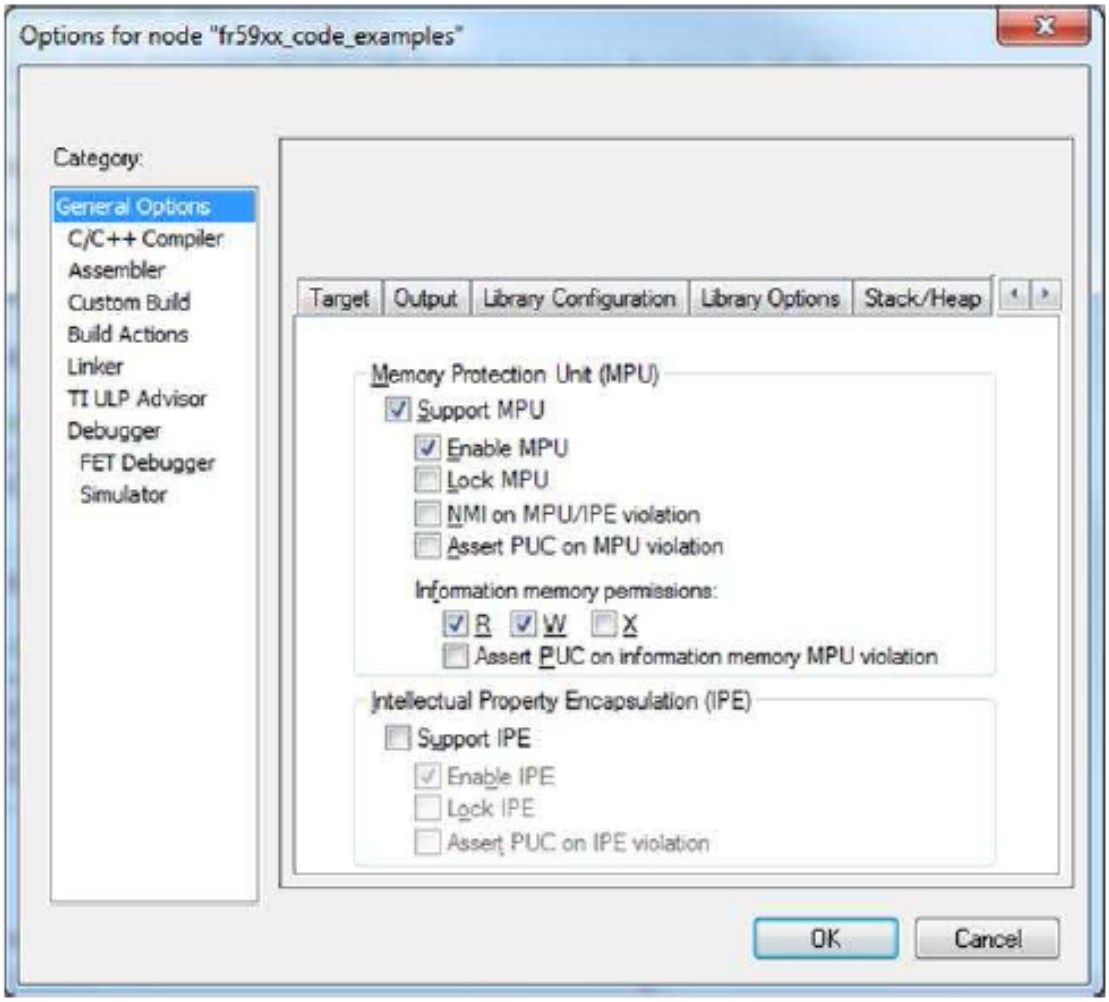

Figure 6.29: MPU IAR wizard

6.6 Firmware Overview

The developed software consist of different tests and a command based protocol in order to interact with the two micro-controllers. Specific commands that μC_1 and μC_2 can understand are so implemented.

6.6.1 Available commands and tests

The commands are unique data that the payload can understand and that generate a payload actions. Here the list of the available commands (see fig.6.30)and the actions that they define will be illustrated

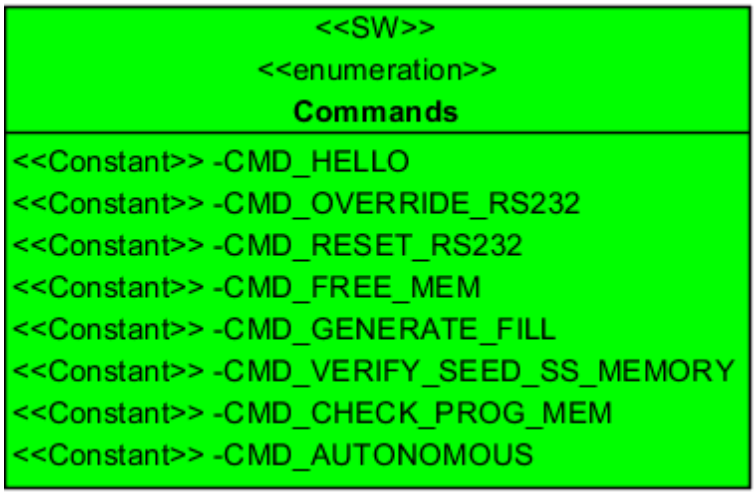

Figure 6.30: Available commands

CMD_HELLO

The OBC tests the communication channels using CMD_HELLO.

1. The OBC tests the RS232 and I2C to understand the peripherals status of the target.
   (a) The OBC sends an CMD_HELLO keyword on the RS232 channel and waits (with a timeout defined by the 1B45 protocol) for an CMD_HELLO as answer.
   (b) The OBC sends an CMD_HELLO keyword on the RS232 channel and waits (with a timeout defined by the 1B45 protocol) for an CMD_HELLO as answer.
2. If there is no response for the RS232 or for the I2C it has to be restored using the I2C communication channel sending a CMD_RESET_RS232 command or the RS232 can be overrided using a $CMD\_OVERRIDE\_RS232$ command.
3. If both the channels are out of service, the target is reset.
4. operation 1, 2, & 3 are repeated for μC_2.

CMD_OVERRIDE_RS232

As already said, the RS232 is intended to be used as primary communication interface while the I2C is intended to be used as backup in order to don't have the possibility to loose the payload so easily and left the target reset as last chance to restore the communication. However using the $CMD\_OVERRIDE\_RS232$ command it is possible to disable the RS232 and use the I2C as unique communication channel.

CMD_RESET_RS232

When this command is received the target writes all the default configuration in the UARTA0 registers. This command is intended to be used if it is not possible to ensure a communication using the RS232 and as a try to restore the communication.

CMD_FREE_MEM

This command is used to understand the dimension of the available FRAM and RAM that can be tested wit the $CMD\_GENERATE\_FILL$ test.

1. The OBC sends the CMD_FREE_MEM command;
2. The target sends the value of the variable $LastStartAddress$ that indicates the beginning of available FRAM in $buffer[0]$ (low side) and $buffer[1]$ (high side);
3. The target sends $LastStopAddress$ that indicates the end of available FRAM in $buffer[2]$ (low side) and $buffer[3]$ (high side);
4. The target sends the value of the variable $LastStartRAMAddress$ that indicates the beginning of available RAM in $buffer[4]$ (low side) and $buffer[5]$ (high side);
5. The target sends $LastStopRAMAddress$ that indicates the end of available RAM in $buffer[6]$ (low side) and $buffer[7]$ (high side);

CMD_GENERATE_FILL

The chosen micro-controller execute a memory test program to detect SEUs. The deep of the memory under test can be chosen using a $STARTADDRESS$ and a $STOPADDRESS$.

1. The OBC sends the $CMD\_GENERATE\_FILL$ command;
2. The OBC sends a 16 bit seed (**unsigned int**) to the target and it is stored in the variable *seed*;
3. The OBC sends a $STARTADDRESS$ and a $STOP\_ADDRESS$. If they are in the available defined range for a RAM or a FRAM test they are stored accordingly. Otherwise an error occurred.
4. The received seed is used by the target in order to generate an 32 bits (unsigned long) pseudo random pattern to fill the memory in the range defined by RAM or FRAM.

The RAM depth is defined by **StartRAMAddress : unsigned long*text**, **StopRAMAddress : unsigned long***. If two valid in-range addresses are received they are stored in **LastStartRAMAddress : unsigned long***, **LastStopRAMAddress : unsigned long***.

The RAM depth is defined by **StartAddress : unsigned long*** and **StopAddress : unsigned long***. If two valid in-range addresses are received they are stored in **LastStartRAMAddress : unsigned long*** and **LastStopRAMAddress : unsigned long***.

CMD_VERIFY_SEED_SS_MEMORY

The OBC tests the memory filled using the command $CMD\_GENERATE\_FILL$.

1. The OBC sends a $CMD\_VERIFY\_SEED\_SS\_MEMORY$ command;
2. The OBC sends the seed and it is stored in the variable $OBCseed$;
3. The OBC sends $STARTADDRESS$ and it is stored in the variable $OBCstartAddress$;
4. The OBC sends $STOPADDRESS$ and it is stored in the variable $OBCstopAddress$;
5. The target verifies local seed and SS. If some parameter is wrong it is re-written. If SS are not compatible with the range an error occurred;

6. The target generates again the sequence and compares word per word each memory location counting the number of mismatching bits updating the variable $numberOfMismatchingBits$;

7. The target sets the a status word called **statusWord : byte** according to the comparisons:

   (a) bit1 seed ok;

   (b) bit2 start ok;

   (c) bit3 stop ok;

CMD_PROG_MEM

The OBC tests the target program memory content.

1. The OBC sends the $CMD\_CHECK\_PROG\_MEM$ command;

2. The target reads the program-only area and computes its CRC;

3. The target compares the computed CRC and $storedCRC$ variable;

4. If the CRCs are equal the target set the flag $HK :: statusRegister[1]$;

5. If the CRCs are not equal the target the target reset the flag $HK :: statusRegister[1]$;

CMD_AUTONOMOUS

The OBC tells to the target micro-controller that it has to run a self test program. The program tests various peripherals.

1. The OBC sends the $CMD\_AUTONOMOUS$ command;

The Autonomous Tests consists of:

- Verify Seed, SS & Memory: In Autonomous Tests this is repeated using **seed : unsigned int**, **LastStartAddress : unsigned long*** and **LastStopAddress : unsigned long*** previously used.

Test PWM: One Output or Two Output PWM signals are generated with a chosen duty cycle;

Test ADC: By means of an LPF the PWM generated by Test PWM is read by the ADC and the read value compared with the equivalent average voltage value that the PWM generate. The data can be also read in differential mode if a Two Output PWM is generated;

Test UART: The data read by Test ADC is sent in loopback over the UART and the received value compared with the sent one;

Test RTC: The RTC time is read at the beginning of Autonomous Tests and again at the end of one cycle of Autonomous Tests in order to understand how long this test long and to check that the time is reasonable. Otherwise it means that an error occurred. and the RTC is reset;

This type of test is very important because it uses a lot of peripherals so if an error occurs it can be easily detected. An UML use case diagram summary of the available test is reported in fig.6.31.

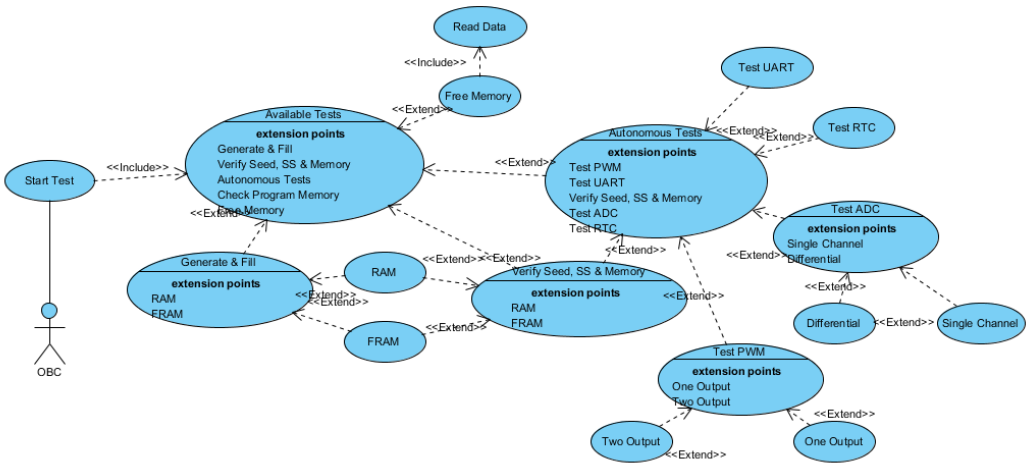

Figure 6.31: Available tests use case diagram

6.6.2 How to get test results

As we already said, the results can be read using the housekeeping module described in 6.6. In particular depending on a index within the one listed in fig.6.32 it is possible to

get the results of a test.

Figure 6.32: Available housekeeping indexes

HK_CURRENT

Read the output of the current sensor and give back the value to the OBC.

HK_VOLTAGE

Read the output of the internal voltage sensor and give back the value to the OBC.

HK_TEMPERATURE

Read the output of the internal temperature sensor and give back the value to the OBC.

HK_SSS

Gives back the result of the **verifySeedSSMemory()** test (see 6.6.1). In particular which one among **seed**, **StartAddress** and **StopAddress** are wrong.

HK_MismatchingBits

Gives back the result of the **verifySeedSSMemory()** test (see 6.6.1). In particular the computed number of mismatching bits.

HK_MismatchingWords

Gives back the result of the **verifySeedSSMemory()** test (see 6.6.1). In particular the computed number of mismatching words.

HK_FirstWrong

Gives back the result of the **verifySeedSSMemory()** test (see 6.6.1). In particular the first found wrong address.

HK_LastWrong

Gives back the result of the **verifySeedSSMemory()** test (see 6.6.1). In particular the last found wrong address.

HK_ORWords

Gives back the result of the **verifySeedSSMemory()** test (see 6.6.1). In particular the OR of the EXOR of all the found mismatching words.

HK_Reboots

Gives back the number of the executed reboots.

6.7 Code redundancy

In the described application, the code stored in the microcontroller is the most critical factor to take into account. If the code is lost because of non-reparable radiation effect, the board cannot be started and used for its purpose. This is the reason why as already discussed, there is the necessity to provide different layers of redundancy.

6.7.1 Double main program

The first layer of redundancy of the payload's code consist of a double memorization of the main program in the FRAM. This can be done declaring the main in a function declared *static inline* and call it two times. This type of declaration creates a copy of

the function every time it is called. The selection on which copy has to be chosen is made reading a pin as first thing at the very beginning of the firmware. The probability that this instruction can be affected by some radiation effect is indeed very low because it is proportional to the memory size and this is very low.

6.7.2 I2C EEPROM

As already described the I2C EEPROM is used to have the possibility to restore the firmware in the unlucky event of failure. The OBC using the I2C reads reads the firmware on the EEPROM memory and downloads it on the micro-controller using the BSL functions.

6.8 Software class: PayloadCommandsAction

All the operations described in 6.6.1 ans 6.6.2 are enclosed int the the developed software class called **PayloadCommandsAction** showed in fig.6.33.

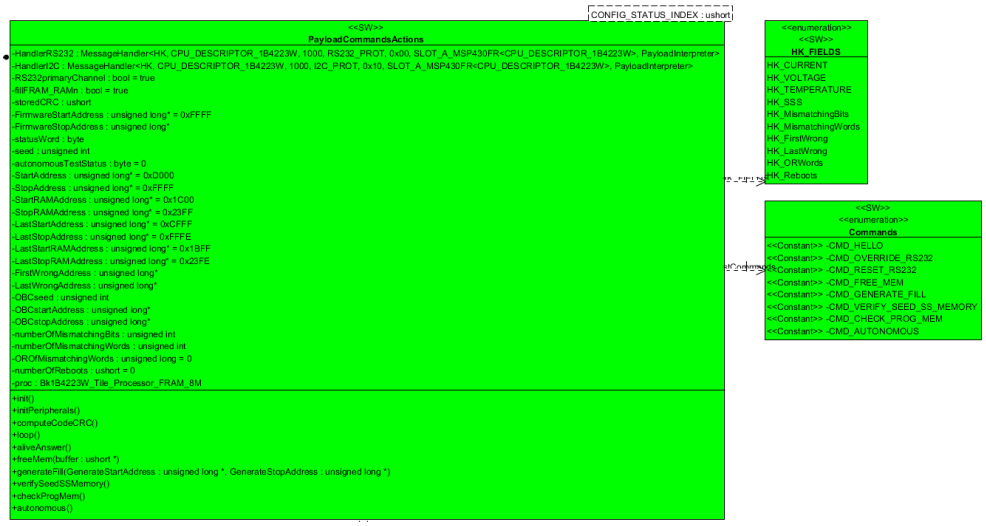

Figure 6.33: PayloadCommandsActions class

init()

Function who check the **D7/A1** pin of the module A (see 5.3.2) in order to jump or not to the second copy of the main program (see 6.7.1).

6.8.1 initPeripherals()

Function used to configure all the used mico-controller's peripherals.

6.8.2 computeCodeCRC

Function used to compute the CRC of the program memory and stores the result in *storedCRC* (see 6.6.1).

6.8.3 loop()

Is the equivalent of a **while(1)** in a firmware. In this function the flag related to available test are checked in order to understand which test has to be executed (see 6.9.1).

6.8.4 aliveAnswer()

Function used to test the communication channels (see 6.6.1).

6.8.5 freeMem()

Function who returns the available memory for the **generateFill()** test (see 6.6.1). Arguments are:

- **GenerateStartAddress : unsigned long*** : the used start address for the generation;
- **GenerateStopAddress : unsigned long*** : the used stop address for the generation;

6.8.6 generateFill()

Function used to fill the memory area under test (see 6.6.1).

6.8.7 verifySeedSSMemory()

Function used to verify the memory under test (see 6.6.1).

6.8.8 checkProgmem()

Function used to check the CRC of the program memory (see 6.6.1).

6.8.9 autonomous()

Function who implement the autonomous test (see 6.6.1).

6.9 Software class: PayloadInterpreter

This class deals with the protocol commands interpretation and with the housekeeping management.

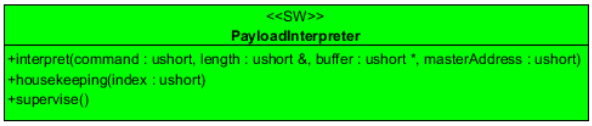

Figure 6.34: PayloadInterpreter class

6.9.1 interpret()

The interpret functions comes from the code developed for the Basic Communication Protocol described in 6.2.3. This function is able to decode the data sent on the chosen communication channel and act as a consequence. It is used to decode the test commands described in 6.6.1 and start the respective test.

6.9.2 housekeeping()

Depending on the index, given to the call of this function, gives back the respective data like described in 6.6.2.

6.9.3 supervise()

This function is called by the housekeeping management in order to check the communication between the payload and the housekeeping object.

CHAPTER 7

Test, feature developments and conclusions

7.1 Software testing

Since the developed drivers are absolutely necessary for the The developed code has been tested using a TI development board. In particular 100 pins FRAM microcontrollers have their own development board called MSP430 100-Pin IPZ Target board (fig.7.1) since from previous hardware releases supply pins are in different positions.

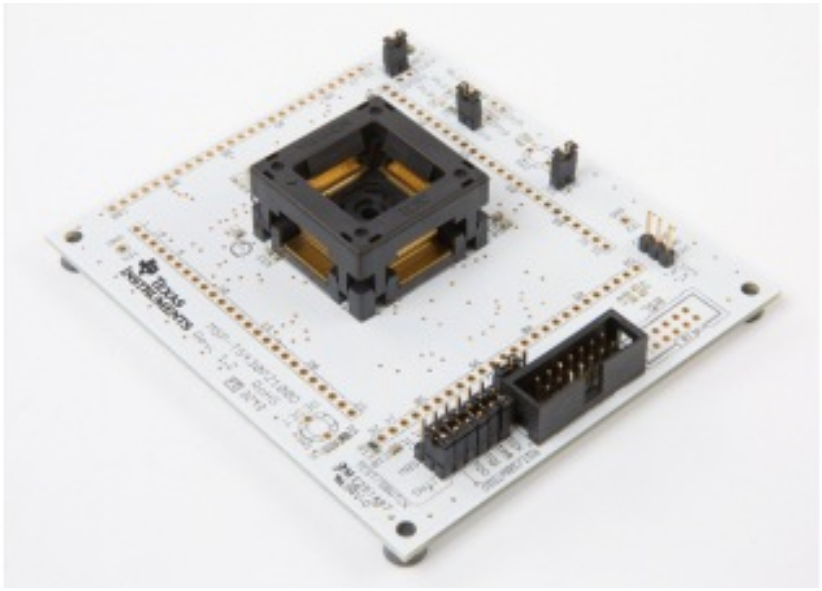

Figure 7.1: MSP430 100-pin target board

It is a standalone 100-pin ZIF socket target board used to program and debug the MSP430 in-system through the JTAG interface or the Spy Bi-Wire (2-wire JTAG) pro-

tocol. The development board supports the MSP430FR6989 FRAM device in a 100-pin LQFP package (MSP430FR6989).

This board has been used in order to test all the written micro-controller's drivers and validate their functionalities. The used IDE, for the developing and debug was IAR Embedded Workbench. In order to test software procedures like the one described in 6.6.1 the debug functionality has been used changing by hand the stored values in memory.

7.2 Hardware testing

For what concern the hardware testing, it was done making all the designed blocks described in 5 on a breadboard and checking all the blocks individually. Once the individual block works, the whole system was linked in order to be sure that nothing went wrong.

7.3 Feature developments

Feature developments of this project, can include:

- Writing more specific software tests in order to validate the full functionality of a peripheral;
- If necessary, add the needed hardware to support the new developed tests;
- Generally speaking FRAM based micro-controllers seems to be more reliable than FLASH based one used in the AraMIS project, so can be thought to substitute the used micro-controller on the other tiles. This can be easily done, because since the developed drivers uses the same methods for each micro-controller, the substitution results in changing only the layout of the microcontroller section (that is a reusable block like explained in 5.3.2).

7.4 Conclusions

In conclusion, we can for sure state that the aim of introducing the MSP430FRxxxx family in the AraMIS project has been reached. Like said in 7.3 this micro-controller can be used in new tiles within the AraMIS structure and this will be for sure an advantage because, like showed in 5.4, the current absorption is very low leading to an huge save of power which is critical in these type of applications. Furthermore the developed board can be used as a payload for incoming space missions and so scientific data can be collected and shared with the community.

Acronyms

AM Active Mode . . . 62

BOR BrownOut Reset . . . 118

BSL Boot Strap Loader . . . 67

COTS Commercial Off The Shelf . . . 32

CRC Cyclic Redundancy Check . . . 38

EEPROM Electrically Erasable Read Only Memory . . . 34

eUSCI enhanced Universal Communication Interface . . . 56

FRAM Ferroelectric Random Access Memory . . . 27

FSM Finite State Machine . . . 34

GEO Geostationary Earth Orbit . . . 25

I2C Inter Integrated Circuit . . . 54

IC Integrated Circuit . . . 63

ISR Interrupt Service Routine . . . 80

LEO Low Earth Orbit . . . 10

LET Linear Energy Transfer . . . 23

LPF Low Pass Filter . . . 59

LPM Low Power Modes . . . 99

LSB Least Significant Bit . . . 92

MPU Memory Protection Unit . . . 29

MSB Most Significant Bit . . . 92

NMI Non Maskable Interrupt 117

OBC On Board Computer 42

OP-AMP Operational Amplifier 65

PCB Printed Circuit Board 51

PDB Power Distribution Bus 51

PDB Power Distribution Bus 51

POD Pico-satellite Orbital Deplorer 44

PUC Power Up Clear 117

PWM Pulse Width Modulation 59

PZT lead (Pb) Zirconate Titanate 28

rad radiation absorbed dose 16

RS232 Recommended Standard 232 54

RTC Real Time Clock 93

SAA South Atlantic Anomaly 10

SCR Silicon Controlled Rectifier 34

SEB Single Event Burnout 34

SEE Single Event Effect 33

SEFI Single Event Functional Interrupt 34

SEGR Single Event Gate Rupture 34

SEL Single Event Latch Up 34

SEP Solar Energetic Particle 18

SEU Single Event Upset 15

SNMI System Non-Maskable Interrupt 117

SOI Silicon On Insulator 37

SOS Silicon On Sapphire 37

TI Texas Instruments 28

TID Total Ionizing Dose 33

UART Universal Asynchronous Receiver Transmitter 56

UML Unified Modelling Language 39

UML Unified Modelling Language 39

WDT Watch Dog Timer 91

Bibliography

[1] COTSlaudio Sansoé, Maurizio Tranchero, *Use of FRAM Memories in Spacecrafts*, 2010.

[2] E. G. Stassinopoulos, James P. Raymond, *The Space Radiation Environment for Electronics*, 1988.

[3] D. R. Barraclough, R. M. Harwood, B. R. Leaton, and S. R. C. Malin, *A model of the geomagnetic field at epoch 1975*, 1975.

[4] E. G. Stassinopoulos, J. H. King, *Empirical solar proton model for orbiting spacecrafts applications.*

[5] A. Haider *Telecommunication Subsystem Design for Small Satellite*

[6] M. R. Mughal *Onboard Communication Systems for Low Cost Small Satellites*

[7] TI Application Report, *MSP430 FRAM Quality and Reliability - SLAA526A*, 2014.

[8] TI Application Report, *Migrating from the MSP430F5xx and MSP430F6xx Family to the MSP430FR58xx/FR59xx/68xx/69xx Family - SLAA555B*, 2014.

[9] TI Application Report, *5xx and 6xx Bootstrap Loader Customization - SLAA450C*, 2013.

[10] TI Application Report, *MSP430 Programming Via the Bootstrap Loader - SLAU319I*, 2013.

[11] TI Application Report, *MSP430 FRAM Technology - How To and Best Practice - SLAA628*