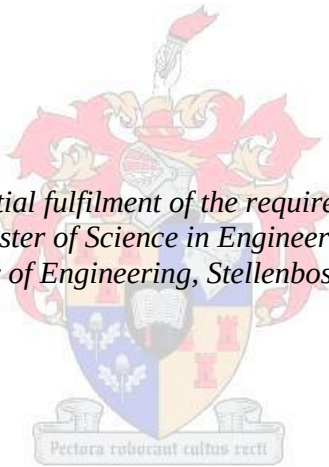
The Design and Development of an ADCS OBC for a CubeSat

By

Pieter Johannes Botma



Thesis presented in partial fulfilment of the requirements for the degree
Master of Science in Engineering
at the Faculty of Engineering, Stellenbosch University

Supervisor: Prof. W.H. Steyn
Department of Electrical and Electronic Engineering

December 2011

DECLARATION

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: December 2011

ABSTRACT

The Electronic Systems Laboratory at Stellenbosch University is currently developing a fully 3-axis controlled *Attitude Determination and Control Subsystem* (ADCS) for CubeSats. This thesis describes the design and development of an *Onboard Computer* (OBC) suitable for ADCS application. A separate dedicated OBC for ADCS purposes allows the main CubeSat OBC to focus only on command and data handling, communication and payload management.

This thesis describes, in detail the development process of the OBC. Multiple *Microcontroller Unit* (MCU) architectures were considered before selecting an ARM Cortex-M3 processor due to its performance, power efficiency and functionality. The hardware was designed to be as robust as possible, because radiation tolerant and redundant components could not be included, due to their high cost and the technical constraints of a CubeSat.

The software was developed to improve recovery from lockouts or component failures and to enable the operational modes to be configured in real-time or uploaded from the ground station. Ground tests indicated that the OBC can handle radiation-related problems such as latchups and bit-flips. The peak power consumption is around 500 mW and the orbital average is substantially lower. The proposed OBC is therefore not only sufficient in its intended application as an ADCS OBC, but could also stand in as a backup for the main OBC in case of an emergency.

OPSOMMING

Die Elektroniese Stelsels Laboratorium by die Universiteit van Stellenbosch is tans besig om ’n volkome 3-as gestabiliseerde oriëntasiebepaling en -beheerstelsel (Engels: *ADCS*) vir ’n *CubeSat* te ontwikkel. Hierdie tesis beskryf die ontwerp en ontwikkeling van ’n aanboordrekenaar (Engels: *OBC*) wat gebruik kan word in ’n *ADCS*. ’n Afsonderlike *OBC* wat aan die *ADCS* toegewy is, stel die hoof-*OBC* in staat om te fokus op beheer- en datahantering, kommunikasie en loonvragbestuur.

Hierdie tesis beskryf breedvoerig die werkswyse waarvolgens die *OBC* ontwikkel is. Verskeie mikroverwerkers is as moontlike kandidate ondersoek voor daar op ’n ARM Cortex-M3-gebaseerde mikroverwerker besluit is. Hierdie mikroverwerker is gekies vanweë sy spoed, effektiewe kragverbruik en funksionaliteit. Die hardeware is ontwikkel om so robuust moontlik te wees, omdat stralingbestande en oortollige komponente weens kostebeperkings, asook tegniese beperkings van ’n *CubeSat*, nie ingesluit kon word nie.

Die programmatuur is ontwikkel om van ’n uitsluiting en ’n komponentfout te kan herstel. Verder kan programme wat tydens vlug in werking is, verstel word en vanaf ’n grondstasie gelaai word. Grondtoetse het aangedui dat die *OBC* stralingverwante probleme, soos ’n vergrendeling (*latchup*) of bis-omkering (*bit-flip*), kan hanteer. Die maksimum kragverbruik is ongeveer 500 mW en die gemiddelde wentelbaankragverbruik is beduidend kleiner. Die voorgestelde *OBC* is dus voldoende as *ADCS OBC* asook hoof-*OBC* in geval van nood.

ACKNOWLEDGEMENTS

The author would like to thank and acknowledge the following for their contribution towards this project:

- **Prof W.H. Steyn** for his knowledge, guidance and patience throughout this project.
- All the people of the **ESL**, especially **Hanco Loubser**, **AM de Jager**, **Arno Barnard** and **Johan Arendse**, for their helpful inputs.
- My **family** and **friends**, especially **Elzaan Kotzé**, for their support and understanding during my long hours at work.
- **God**, for giving me the talents and opportunities to follow my dream.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

NOMENCLATURE

Abbreviations and Acronyms

- ADCS – Attitude Determination and Control Subsystem
- ADC – Analog to Digital Converter
- CMSIS – Cortex Microcontroller Software Interface Standard
- CPUT – Cape Peninsula University of Technology
- COTS – Commercially available Off-The-Shelf
- DAC – Digital to Analog Converter
- DMA – Direct Memory Access
- EPS – Electronic Power System
- ESL – Electronic Systems Laboratory
- HAL – Hardware Abstraction Layer
- I2C – Inter-Integrated Circuit
- IGRF – International Geomagnetic Reference Field
- ISR – Interrupt Service Routine
- LEO – Low Earth Orbit
- MCU – Microcontroller Unit
- OBC – Onboard Computer
- OS – Operating System
- RCO – Resistor-Capacitor Oscillator
- RTC – Real-Time Clock
- RTOS – Real-Time Operating System
- SAA – South Atlantic Anomaly
- SEE – Single Event Effects
- SEL – Single Event Latchup
- SEU – Single Event Upset
- SGP4 – Simplified General Perturbation no.4
- SPI – Serial Peripheral Interface
- SRAM – Static Random Access Memory
- SSTL – Surrey Satellite Technology Limited
- TID – Total Ionizing Dose
- TLE – Two Line Elements

- TMR – Triple Modular Redundancy
- UART – Universal Asynchronous Receiver/Transmitter
- USART – Universal Synchronous/Asynchronous Receiver/Transmitter
- XO – Crystal Oscillator

1 BACKGROUND

1.1 MISSION

The *Cape Peninsula University of Technology* (CPUT) has recently started working on a series of nanosatellite (CubeSat) missions which are detailed in [1]. Due to the importance of a robust *Attitude Determination and Control System* (ADCS) in most satellite missions, CPUT decided to collaborate with the University of Stellenbosch because of their experience in the ADCS field as well as satellite research in general. The University of Stellenbosch has been involved in the development of two satellites, namely the SUNSAT in 1999 [2] and the SumbandilaSat [3] in 2009. The first CubeSat by CPUT is to be a small (10 × 10 × 10 cm) satellite with a long antenna as its main payload and a relatively basic ADCS. The mission will be to calibrate the radar antenna patterns for the Hermanus Magnetic Observatory's antenna array in Antarctica. A second, slightly larger satellite (3U CubeSat) will subsequently be developed. The payloads for this satellite still have to be determined.

For the 3U CubeSat, the University of Stellenbosch will design a completely independent ADCS unit. This unit will control all the sensors and actuators, run all the algorithms and perform calculations in order to achieve the desired orientation which, through a high-level interface, can be set by the main *Onboard Computer* (OBC) and/or ground station. The two main reasons for implementing the ADCS in a separate unit are managing complexity and improving modularity.

1. **Complexity**

   The ADCS is a very complex subsystem responsible for updating sensor data and controlling actuators, while running multiple models (sun, orbit, etc.), estimators and control algorithms. Because of the large amounts of computations and data being handled, the ADCS tends to dominate an OBC's resources. In order to keep the main OBC free to react to mission-critical subsystems, such as power and communications, it is more desirable to implement the ADCS on a separate OBC.

2. **Modularity**

   Modularity refers to the ability to add, with minimal effort, a unit or module to a system that improves the overall ability of that system. The idea of a CubeSat as a standard satellite bus for which expansion boards of a different subsystem can be added to extend the functionality of the satellite fits very well into this definition of modularity. The

ADCS unit takes that same trend one step further by grouping all ADCS-related expansion boards (sensors and actuators) and having them controlled by a separate OBC. This unit can then be added easily to any CubeSat for which an ADCS is required.

Stellenbosch University's *Electronic Systems Laboratory* (ESL) has already designed a horizon and sun sensor, called CubeSense, which is used in the STRaND CubeSat from *Surrey Satellite Technology Limited* (SSTL) [4] and can be seen in Figure 1.1. The actuators, OBC and ADCS algorithms are still in development.

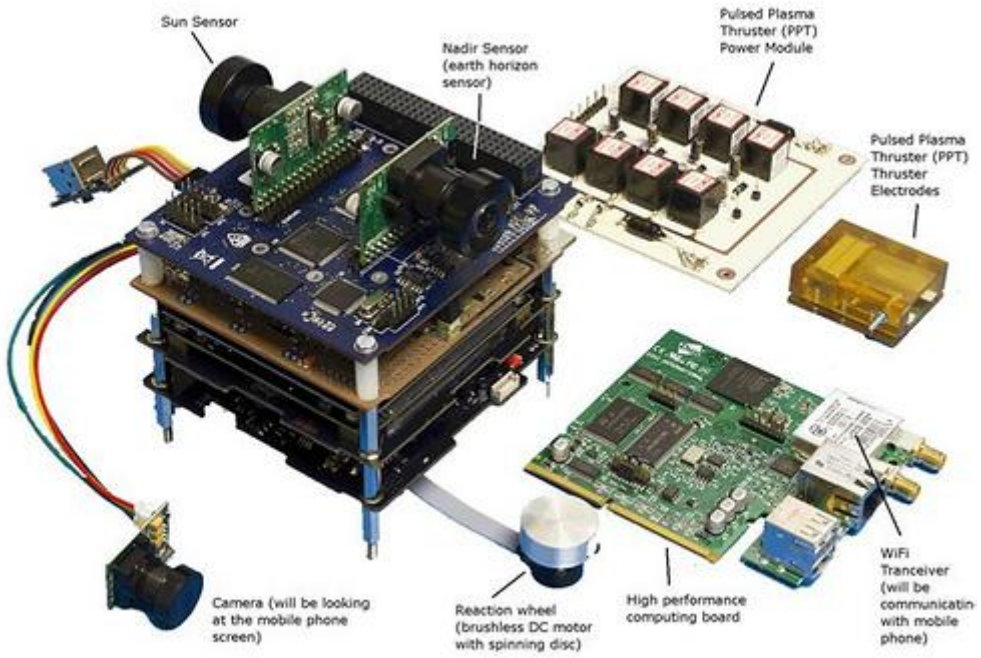


Figure 1.1: SSTL STRaND CubeSat. [4]

This thesis will document the design and development of the OBC required for the ADCS unit on the 3U CubeSat. The ADCS OBC will act as the interface between the main OBC and the orientation control of the satellite. The desired attitude controller and determination estimator of the satellite can be sent to the ADCS OBC via a telecommand. This will then be interpreted and the control algorithms together with sensor data will then compute the necessary output/commands for the actuators to achieve the desired attitude. The lack of space and power on a CubeSat impose strict limits on its design, which will be discussed later in Section 2.1.

1.2 CUBESAT

Traditional satellites tend to be large, complex and expensive systems. To maximise the value of a satellite, multiple payloads are fitted onto one bus. This forced designers to use redundant subsystems and radiation-hardened components to ensure reliability of the satellite which in turn increased the size and cost of the satellite. As the payloads continued to increase in number and complexity, the satellite bus and subsystems had to be redesigned for almost every mission. This made it difficult for academic institutions (such as universities) to start a satellite programme, because of the high cost and technical expertise that even a small satellite used to require. This increasing complexity and cost spiral is the main reason for the development of the CubeSat standard, according to [5].

The CubeSat approach is trying to change this by adhering to a satellite bus standard. A 1U CubeSat bus is roughly 10 × 10 × 10 cm and weighs around 1 kg. Figure 1.1 shows an example of an 1U CubeSat with all its subsystems, excluding the body-mounted solar panels. These CubeSat units can be fit together to create a 2U (10 × 10 × 20 cm), 3U (10 × 10 × 30 cm), etc. Examples of the chassis to contain these CubeSats are shown in Figure 1.2. There are two main advantages when working with CubeSats. Firstly, the standard size of the satellite structure has allowed for the design of a standard launch adapter (P-POD) which made it easier for CubeSats to piggyback on big satellite launches for a fraction of the price of a dedicated launch [6]. Secondly, it is possible to buy all the components and subsystems to create a space-ready CubeSat [7][8]. Therefore none of the components have to be specially made or developed; it is only necessary to add a payload to the CubeSat to do meaningful research.

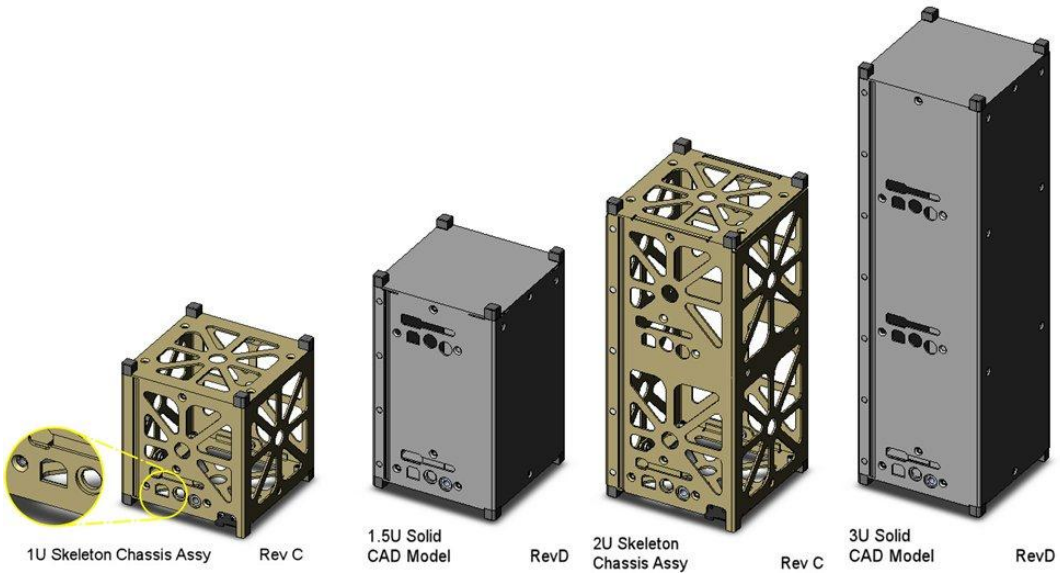


Figure 1.2: Examples of CubeSat Unit Sizes. [44]

The standard bus interface and protocol of a CubeSat makes it easier to design subsystems that can be used on multiple CubeSat missions. The use of *Commercially available Off-The-Shelf* (COTS) is encouraged to keep development costs to a minimum as well as to make use of the myriad of electronic components available on the market not necessarily aimed at the space industry. It is possible to manage the risk when not using space grade components on a satellite. This will be shown later in chapters 3 and 4.

The size of a CubeSat makes it a very affordable satellite bus to design and launch but it is this size that limits the weight and power usage when designing a subsystem for a CubeSat. Subsystems have to be very compact to fit into a CubeSat and the small solar panel surface area on the outside of the CubeSat chassis does not generate large amounts of electric power which results in very strict power budgets for each subsystem.

1.3 ADCS

The ADCS is responsible for the orientation of the satellite within its orbit. This subsystem follows the same principle as most control loops and is shown in Figure 1.3. Sensors on the satellite provide the latest measurements regarding its orientation relative to sun (sun sensor), stars (star tracker), earth (horizon sensor and magnetometer) and/or spin rates (gyroscope). The measured data is then compared to reference values set by the main OBC. If an error exists due to sensor noise, external forces on the satellite (drag and solar pressure) and other perturbations (gravity from earth, moon and sun), the satellite can be realigned through the use of actuators such as magnetic torque rods, reaction/momentum wheels and even thrusters.

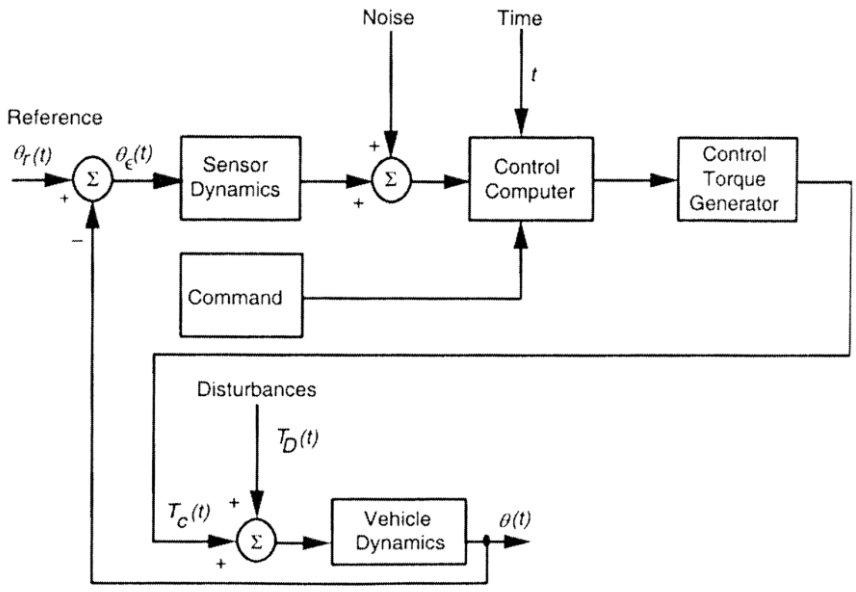


Figure 1.3: ADCS Control Loop for a Satellite. [7]

Being able to control the orientation of a satellite is very beneficial and might even be considered mandatory for some missions. The following are two common uses for an ADCS with examples:

1. Pointing a device or payload towards a target.

Examples:

- Pointing an imager at a target for longer exposure and therefore higher quality images.
- Pointing an antenna towards the ground station during an overpass to maximize transmission length and quality.

2. Spinning the satellite around an axis at a specified rate.

Examples:

- Spinning the satellite around the axis normal to the orbit plane (Y-Thompson [9]) to stabilize it against external disturbance forces.
- Spinning the satellite around the nadir axis (barbeque spin) to ensure that equal amounts of sunlight reach all the body-mounted solar panels (which also improves thermal stability).

For a more in depth discussion of the advantages and implementations of ADCS for a satellite, refer to [10], pages 354-380.

1.4 SPACE ENVIRONMENT

Space is a hazardous environment. The following section will highlight the major challenges when designing electronic equipment, such as the ADCS OBC, for space.

1.4.1 RADIATION EFFECTS

Stars emit various forms of charged particles, known as radiation, during its fusion process. On the earth's surface most of the radiation from space is diverted by the earth's magnetic field. Most of the time satellites in *Low Earth Orbit* (LEO) are within the safety of the magnetic field, except when passing over the *South Atlantic Anomaly* (SAA). The SAA (shown in Figure 1.4) is a region where radiation from the Van Allen belts as well as other charged particles enters the atmosphere. This radiation causes undesired effects in electronic equipment, especially semiconductor devices. A summary of these affects are explained below which can be found in more detail in [11] and [12], pages 214-221.

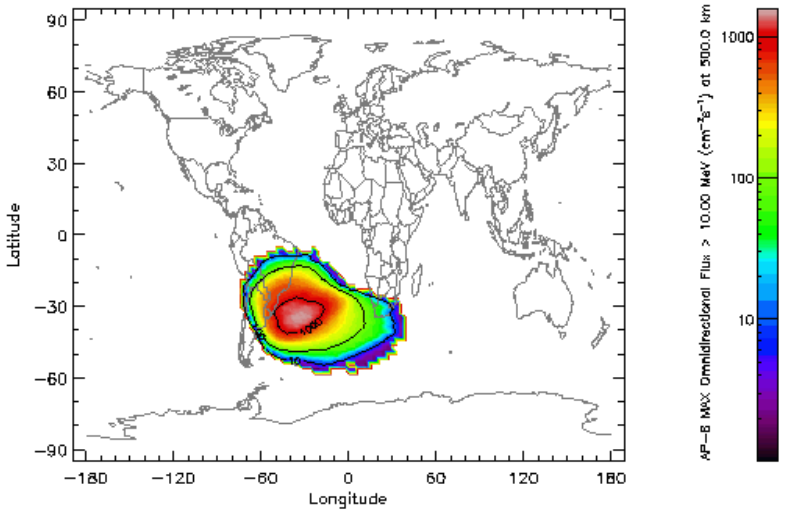


Figure 1.4: Proton Flux Simulation Showing the South Atlantic Anomaly. **[40]**

- **Single Event Upset**

  A *Single Event Upset* (SEU) occurs when a charged particle causes a change in the contents and/or state of a device without causing permanent damage to that device. [11] A common example is when the content of a memory cell is changed by a charged particle. This is referred to as a bit-flip. These upsets usually do not damage a device, but it could cause undesired effects within the operation of a device or system.

- **Single Event Latchup**

  A *Single Event Latchup* (SEL) occurs when radiation causes a parasitic transistor within the device to switch on and result in excessive current flow. [11] This excessive current flow may damage the device, due to the heat generated locally, if the latchup is not removed by means of power cycling (switching power on and off to device).

- **Total Ionizing Dose**

  The *Total Ionizing Dose* (TID) is the amount of radiation build-up a device can withstand before its operation is deemed unreliable. [11] The TID tolerance can be seen as a measure for determining the life expectancy of an electronic device. Some electronic devices have a radiation-hardened version which has a greater TID tolerance, but they are considerably more expensive. Some COTS do have a TID tolerance large enough to justify its use on a satellite with a short mission in LEO.

1.4.2 REMOTENESS

One of the major challenges when designing a satellite is to compensate for the fact that if a component or subsystem malfunctions in space, it cannot be brought in for repairs. Some satellites have the ability to reconfigure their software during operation, but when a serious hardware error occurs it usually results in the loss of the satellite. Implementing redundancy and radiation-hardened components are ways in which reliability can be designed into a satellite according to [11] and [12].

- **Redundancy**

  One form of redundancy is when a satellite has one or more dormant subsystems that can take over in case of a failure. A special case of redundancy called *Triple Modular Redundancy* (TMR) is when three subsystems/devices operate simultaneously and a controller chooses the result based on a majority vote.

- **Radiation- Hardened Components**

  One of the major reasons for a failure on satellites is components failing because of radiation (as explained in 1.4.1). Therefore the simplest way to incorporate reliability into a system is to use radiation-hardened components. Some electronic devices have radiation-hardened versions available from their manufactures, but they are much more expensive.

Other challenges for satellite design include thermal issues due to the lack of atmosphere which could cause massive temperature gradients (depending on which side is illuminated by the sun), micro meteorites, spacecraft charging, outgassing and many more which are described in [11], pages 22-27. These challenges will not be discussed in this thesis, because they do not directly influence the design of electronic subsystems, such as, in this case, an ADCS OBC.

1.5 DOCUMENT OUTLINE

The following section gives a short description of each chapter in this document.

1. **Background** – An introduction to the mission and topics related to the project.

2. **Hardware Design** – Details the requirement definition of the ADCS OBC and the system, subsystem and component level design.

3. **Software Development** – Discusses the software development of the ADCS OBC which includes low level and user level interfaces.

4. **Tests and Measurements** – Lists the results from the tests developed and measurements taken from ADCS OBC prototype.

5. **Conclusions and Recommendations** – Discusses the results of the tests and measurements with regards to the requirements and any recommended changes to the ADCS OBC.

6. **Summary** – A summary of the project.

2 HARDWARE DESIGN

This chapter will discuss the techniques followed during the hardware design of the ADCS OBC. Firstly the requirements definition is presented, which served as the standard by which the entire system was designed. The next section covers the process of selecting the microcontroller. This is important for the design, because selecting the appropriate microcontroller directly and indirectly affects the rest of the OBC. Following that, an overview of the OBC will be provided, looking at the different subsystems: why they are there and what they will do. A lower-level overview will then follow which will briefly describe how all the subsystems were implemented and which components were used.

2.1 REQUIREMENT DEFINITION

It is important to clearly define the requirements before starting any design. Requirements need to take into account not only the main objective of the design, but the challenges it presents as well. For this project the objective is to design an ADCS OBC for a CubeSat. The following requirements were defined while taking into account the aspects discussed in the previous chapter, such as the complexity of an ADCS, the limitations of a CubeSat and the harsh space environment:

2.1.1 PERFORMANCE

ADCS is a very complex subsystem. Therefore the OBC has to be powerful enough to handle all the complex computations (control algorithms, Kalman filters), precise control (floating-point and double data types) and large amounts of data (telemetry, models and sensor data).

2.1.2 EFFICIENCY

A CubeSat is very limited in power and space. This means that the OBC has to be very efficient and designed as cost effective as possible in terms of power usage and size. Low-power components should be used where possible and if a component or subsystem is not used it should be powered down to conserve energy. The OBC is also required to use less than 1 Watt power during peak operation.

2.1.3 RELIABILITY

On a satellite mission, reliability is always a major requirement because if a failure occurs in space it cannot be repaired and usually results in the end of the mission. The two most common implementations for reliability are to either use radiation-hardened components and/or include redundant components or subsystems. However, neither of these are an option

for the CubeSat. Radiation-hardened components are too expensive and redundancy takes up too much space.

The only way to ensure reliability is to design the OBC to be robust. Robustness is the ability to adapt and survive in case of an emergency. From a design point of view this means including error-detection measures that will respond by either trying to rectify or isolate these errors. The OBC should be robust on both the hardware as well as the software level in order to cope with the hazardous space environment.

2.2 MICROCONTROLLER SELECTION

The *Microcontroller* (MCU) is the most important component on an OBC. The features of the MCU usually dictate how the rest of the OBC is designed, because it acts as the interface between the majority of the components. Due to the ever-growing electronic market, especially in terms of efficiency, many different MCUs are available to choose from.

2.2.1 8-BIT VS. 16-BIT VS. 32-BIT MCUS

An 8-Bit MCU is used in designs where power usage is considered vital. Its small architecture allows it to control a system with minimal power usage, but also limits its mathematical capabilities, especially with large data types such as floats and doubles.

A 32-Bit MCU is mostly used for application purposes where large amounts of data are being processed and power is a secondary (or not even a) factor. It is very capable at math handling due to its larger registers and bus widths.

A 16-Bit MCU fills the slot between these two extremes. It is much better at math and floating-point handling, compared to 8-bit MCUs, while still maintaining efficiency and power consumption.

Taking the above into account, a 16-bit MCU would seem the logical choice for the ADCS OBC. However, a recent shift in the market has occurred. Manufactures, such as ARM [13] and Atmel [14], are providing 32-bit processors aimed at the low-power (8-bit / 16-bit) market. These MCUs do not only perform better, but their power usage is comparable (and even superior in some cases) to the 16-bit MCUs. This is possible because a 32-bit MCU will generally complete the same tasks and computations faster than a 16-bit MCU and therefore spend more time in an optimized “sleep mode” which results in less energy being used, as illustrated in Figure 2.1.

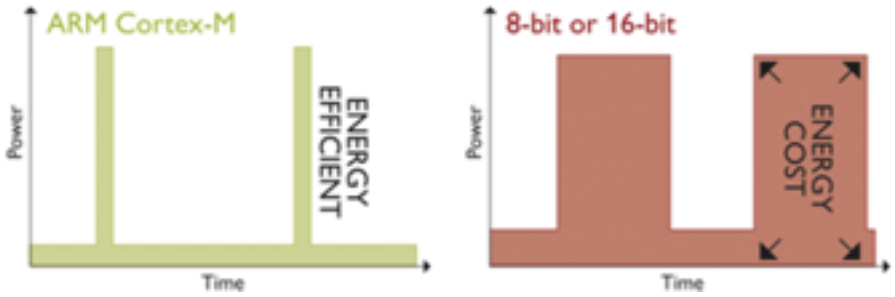


Figure 2.1: Cortex-M Energy Efficiency Solution. [10]

The new generation low-power 32-bit MCU was identified as an ideal candidate for the ADCS OBC. It is more than capable at handling the computational requirements of the ADCS algorithms but also includes enough energy-saving features which will be required on the limited power budget of a CubeSat.

2.2.2 MICROCONTROLLER COMPARISONS

Table 2.1 compares most of the current low-power 32-bit MCUs from various manufacturers, except for two: the MSP430 is a 16-bit MCU and AT91SAM7 is based on an older 32-bit MCU architecture.

The MSP430 is used in the CubeSat main OBC from Pumpkin (FM430 [15]) and is very competitive with regards to energy consumption. This, in fact, highlights how efficient some of these low-power 32-bit MCUs actually are as they have better active and sleep efficiency than the MSP430. Even though the 16-bit and 32-bit MCUs have comparable power usage, the 32-bit MCUs have a clear advantage regarding performance.

It is interesting to note the similar performance of the EFM32 Gecko, LCP13x and STM32L MCUs. They are all based on ARM's next generation low-power 32-bit architecture called the Cortex-M3 range. The Atmel AT91SAM7A3 is based on the older generation ARM7TDMI architecture and is also included to showcase the differences between the current (Cortex-M3) and previous (ARM7) generation of low-power 32-bit MCUs. The ARM7TDMI architecture has flight heritage in space, onboard the UTIAS SFL's CanX range of satellites [16], which will hopefully translate into the new generation of ARM MCUs.

Table 2.1: MCU Comparison (current measurements @ 3.3 V).

Manufacturer	MCU	Speed MHz	Performance DMIPS/MHZ	Active uA/MHz	Sleep uA
Energy Micro	EFM32 Gecko*****	32	1.25	**180**	**0.6**
STM	STM32L*****	32	1.04	230	10.4
NXP	LPC13x*****	72	-	236	30
Atmel	AT32UC3Lx	50	1.28	300	3.5
TI	MSP430F16x	8	1.5	330	1.1
Microchip	PIC32MX3/4	80	**1.56**	688	25
Freescale	MCF521	80	0.95	801	5.38
Atmel	AT91SAM7A3	60	0.9	1167	175

*** Cortex-M3-Based Architectures Microcontrollers**

Table 2.1 shows that the Cortex-M3-based MCUs are superior in terms of efficiency, closely followed by the AT32UC3L, which is based on the Atmel AVR architecture. These four MCUs have been evaluated further, as summarized in Table 2.2, based on the features of each MCU, which might be useful on an ADCS OBC. These include:

- **On-chip memory**

  This includes the amount of available SRAM and flash memory on the chip. This will be required for code and program/user data. The ADCS program works with large data types (float and double) and contains large models, arrays and variables. Therefore, more on-chip memory will allow for larger and more feature rich ADCS programs to be executed.

- **External Bus Interface**

  The *External Bus Interface* (EBI) is an interface that allows the MCU to extend its memory capabilities by accessing external memory. This external memory can usually be accessed by the user the same way in which the internal memory is accessed. This is useful when more memory might temporarily be required (telemetry data) or for storing multiple programs the MCU can execute.

- **Inter-Integrated Circuit**

  The *Inter-Integrated Circuit* (I2C) is a popular bus-based communication channel also referred to as “two wire interface”, because it only uses two IO lines for communication (clock and data). The I2C is the main communication channel on the CubeSat and links the different subsystems. It would therefore be beneficial for the MCU to include an

internal I2C controller, which would reduce the amount of chips on the PCB since an external I2C module will not be needed.

- **Serial Peripheral Interface**

  The *Serial Peripheral Interface* (SPI) is a point-to-point-based communication channel which can transfer data at high speeds. This is useful on a satellite when large amounts of data need to be transferred between two subsystems without clogging up the main communication bus.

- **Analog to Digital Converter**

  The *Analog to Digital Converter* (ADC) is a unit that samples analog signals (usually voltage) and converts it to a digital value. An ADC unit has a resolution which determines the accuracy of the digital value. ADCs are useful on a satellite because they can be used to measure currents and voltages for telemetry purposes.

Table 2.2: Comparison of MCU features.

Features	**AT32UC3L**	**EFM32**	**STM32L**	**LPC13xx**
Flash (kB)	64	128	128	32
SRAM (kB)	16	16	16	8
EBI	N	Y	Y	N
I2C	2	1	2	1
SPI	1	2	2	1
ADC (Resolution)	12-bit	12-bit	12-bit	10-bit

Taking the data in Table 2.2 into account, the EFM Gecko MCU was chosen because of its energy efficiency as well as offering all the features that could prove useful on an ADCS OBC for a CubeSat.

2.2.3 EFM GECKO MCU

This section will discuss the *EFM32 Gecko MCU* (EFM32G) in further detail, especially looking at its core architecture, energy management schemes, memory and bus system and, finally, peripherals. Figure 2.2 presents a block diagram of the EFM32 MCU on system level. The different colours indicate the different energy modes, which will be further explained later in this section.

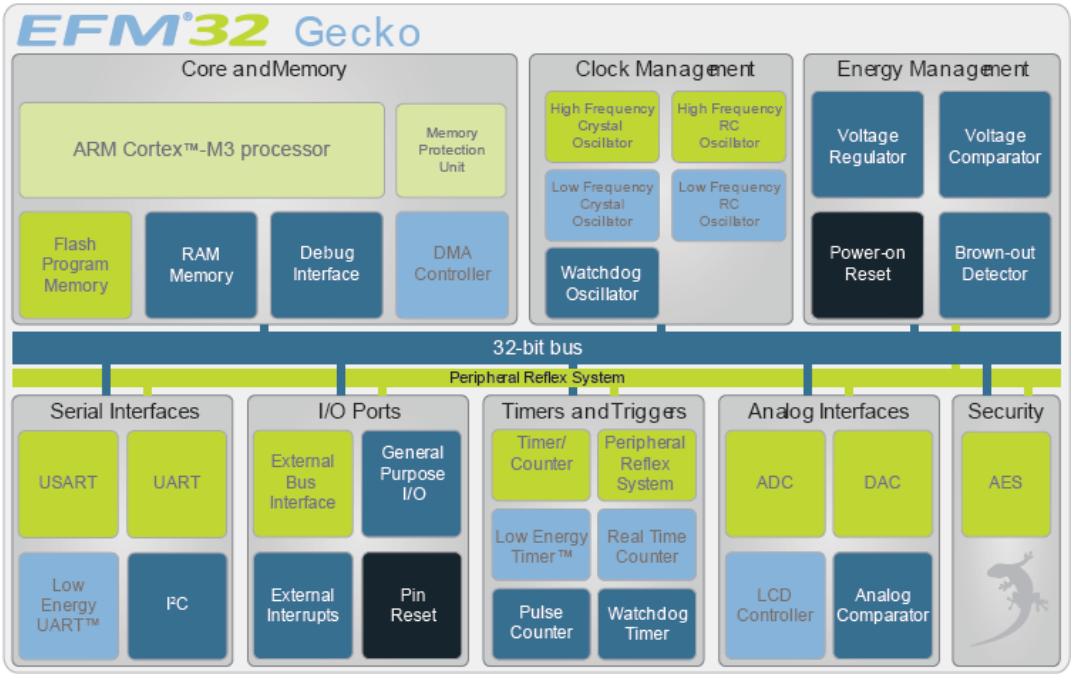


Figure 2.2: Diagram of EFM32 Gecko MCU. [12]

2.2.3.1 ARM CORTEX-M3 CORE

The ARM Cortex-M3 core, shown in Figure 2.3, is based on a high-performance processor core and designed to optimize efficiency. The following are some of the features included with the Cortex-M3 processor.

- **3-Stage Pipeline**

  Most processor instructions consist of three stages, namely the *instruction fetch* (IF), decode and execute stages. The use of a 3-stage pipeline enables the processing of different stages in different pipelines simultaneously. This generally ensures that the processor is completing an instruction per cycle instead of waiting three cycles, for all three stages, to complete a single instruction.

- **Harvard Architecture**

  The Harvard architecture uses different memory busses for code and data. On MCUs where there is no cache memory, the Harvard architecture effectively doubles the memory throughput, because the instruction and data (usually stored in separate memory locations such as flash and SRAM) can be fetched simultaneously.

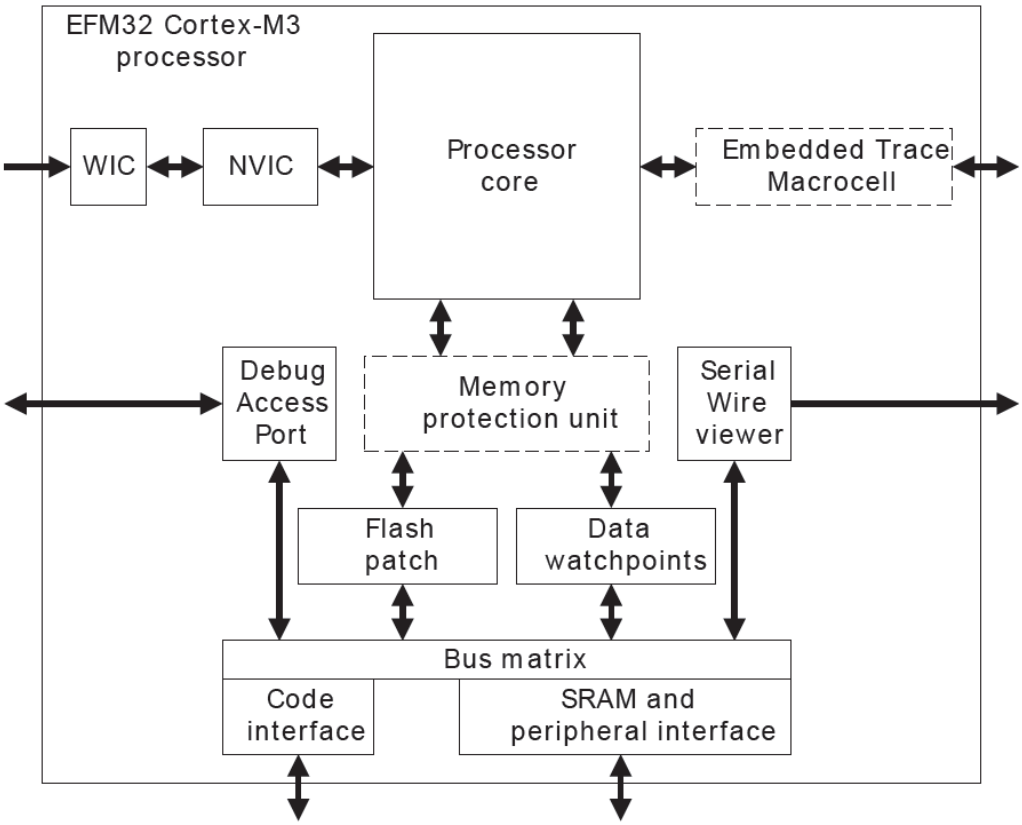


Figure 2.3: Cortex-M3 Processor Core. [17]

- **Hardware Division and Single-Cycle Multiplication**

  The Cortex-M3 processor core implements hardware division that is able to execute signed and unsigned division operations between two and 12 cycles. Together with a single cycle 32-bit multiplication, this allows for very efficient arithmetic computations, which are usually the most demanding responsibility of an MCU.

- **Interrupt Handling**

  The Cortex-M3 processor includes a *nested vector interrupt controller* (NVIC) and *wakeup interrupt controller* (WIC). These controllers allow the MCU to handle interrupts during deep sleep modes, which lowers its energy consumption.

- **Thumb-2 Instruction Set**

  The Thumb-2 instruction set is used by the Cortex-M3 processor and offers excellent code density. Even though the Cortex-M3 is a 32-bit MCU, the majority of its instructions are 16-bit [13]. The 32-bit registers and bus width decrease the program size

further because fewer instructions are required when working with larger data types (such as floating-point and double data types).

The Cortex-M3 includes many other features, such as a memory protection unit and extensive debugging capabilities, which can be studied further in the following references: [17] [18].

2.2.3.2 MEMORY AND BUS SYSTEM

The EFM32G uses an *AMBA High-performance Bus* (AHB) bus which allows its four bus masters to interact with the system through a memory-mapped address space. The four masters are:

- **I-Code** – for instruction fetches to code memory
- **D-Code** – for data and debug access to code memory
- **System** – for instruction fetches, data and debug access to system space
- **DMA** – for memory transfers to and from the entire memory space

Figure 2.4 is a block diagram of the Cortex-M3 memory and bus system. The four bus masters are shown on the left, with the system components on the right. The peripherals are connected to the AHB through the AHB-to-APB bridge. The system is memory mapped and the address space is shown in Figure 2.5.

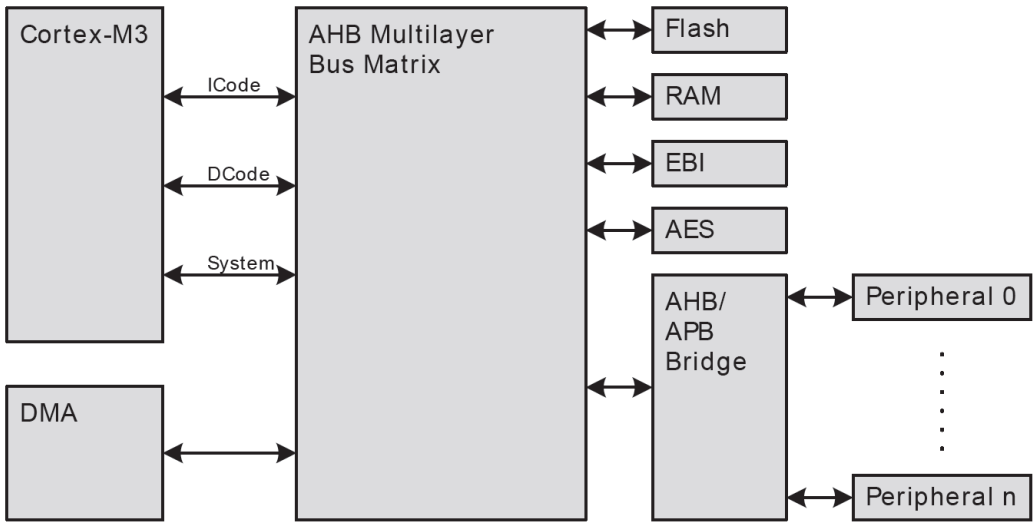


Figure 2.4: Cortex-M3 Memory and Bus System. [16]

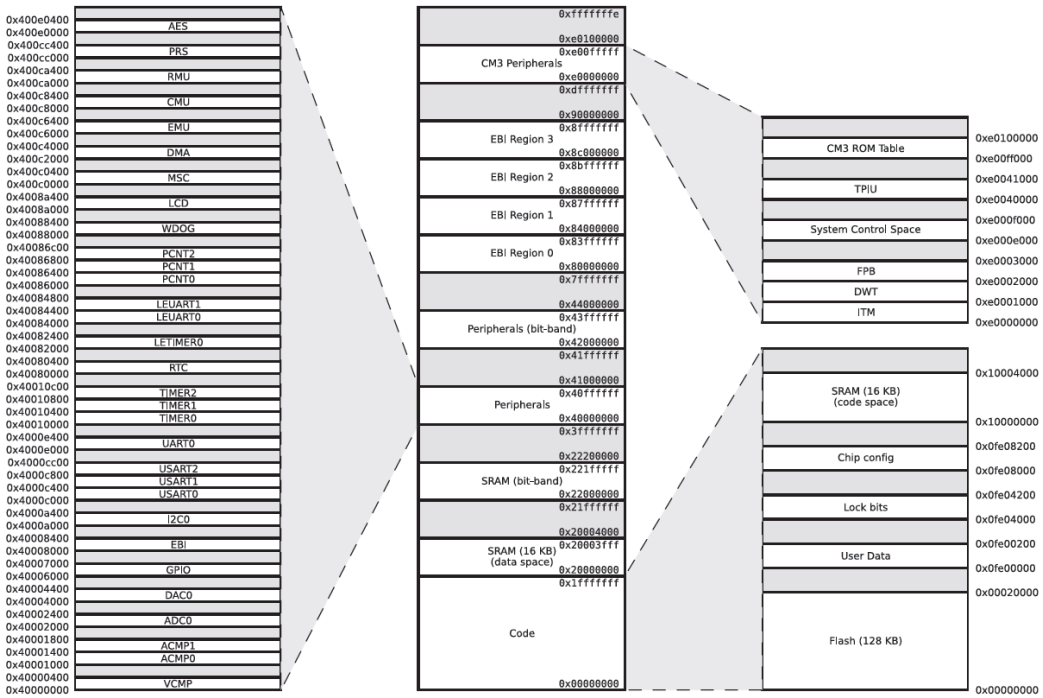


Figure 2.5: Cortex-M3 System Address Space. [16]

2.2.3.3 PERIPHERALS

The EFM32G includes a wide variety of peripherals as well as a few specially developed low-energy peripherals. These peripherals are the following:

1. **General Purpose Input/Output**

   The EFM32G includes up to 90 *General Purpose Input/Output* (GPIO) pins, depending on the package size. The EFM32G pins allow for multiple configurations of the peripherals to maximize the use of pins and peripherals. They also allow for up to sixteen asynchronous interrupt channels.

2. **Timers and Counters**

   The EFM32G includes three general purpose 16-bit timers. These timers can either be configured as counters, or used for input capture, output compare or *Pulse Width Modulation* (PWM). A watchdog timer and a separate low-energy 16-bit timer are also included, both of which can be used during sleep modes when most other peripherals are disabled.

3. Analog

The EFM32G includes a 12-bit ADC with a sample rate of one million samples per second. The ADC can choose as input one of eight external pins or six internal signals. A 12-bit DAC is included which can either be used in two single-ended modes or in differential mode. A voltage comparator can be used to monitor or compare eight external pins with three internal references. A supply voltage monitor and temperature sensor is included as well for monitoring purposes.

4. Communication

The EFM32G includes the UART communications protocol, which is usually used for debugging purposes. Two low-energy UARTs are also available that are able to operate independently from the main processor during sleep mode by means of the DMA controller. Two further USARTs are included as well and can each be configured either as a UART or an SPI port. The EFM32G further has an I2C module that includes address recognition, even during deep sleep modes.

This list is only a summary of the peripherals available on the EFM32G. For an in-depth look at the peripherals and features available on the EFM32G, please refer to [17].

2.2.3.4 ENERGY MANAGEMENT

The EFM32G has five well-defined energy modes, EM0 – EM4. A variety of wake-up triggers and low latency switching between these modes, allow for the maximum amount of time in the lowest possible energy mode which is key to saving energy. Table 2.3 provides a summary of the properties of the different energy modes available in the EFM32G.

Table 2.3: EFM32 Gecko Energy Mode Properties. **[17]**

Energy Mode	EM0 Run	EM1 Sleep	EM2 Deep Sleep	EM3 Stop	EM4 Shutoff
Current consumption	180 uA/Mhz	45 uA/MHz	0.9 uA	0.6 uA	20 nA
Wake-up time	0	0	2 us	2 us	163 us
Core	On				
HF peripherals	On	On			
LF peripherals	On	On	On		
Register & RAM retention	On	On	On	On	
Reset detectors	On	On	On	On	On

For a better understanding of which peripherals are available during a specific energy mode, compare Figure 2.6 with Figure 2.2. The peripherals are colour coded, where each colour represents the lowest energy mode in which the peripheral will respond to activity.

Figure 2.6: Energy Mode Indicator. [16]

2.3 SYSTEM OVERVIEW AND DESIGN

Figure 2.7 shows the ADCS OBC system in block diagram form. The MCU is the core of the OBC with four subsystems (power, monitoring, memory and communication) built around it. In the following sections each of these subsystems will be discussed in further detail with regards to their functions as well as their implementation.

2.3.1 MCU

As mentioned in the previous section, the MCU directly and indirectly affects the other subsystems in the following ways:

- **Power –** The MCU requires a stable power supply from the power subsystem.
- **Monitoring –** The MCU actively samples and processes the monitored channels with its ADC peripheral unit.
- **Memory –** The MCU is the only component responsible for storing and retrieving data from the external memory.
- **Communication –** The MCU connects to external subsystems (main OBC, sensors and actuators) where it either receives data from, or transmits data to.

The significance of the MCU in the OBC necessitates the implementation of extra reliability and accuracy in its design. An external watchdog will add reliability against undesired lock-ups and external real-time and high-frequency crystal oscillators will generate more precise clock signals.

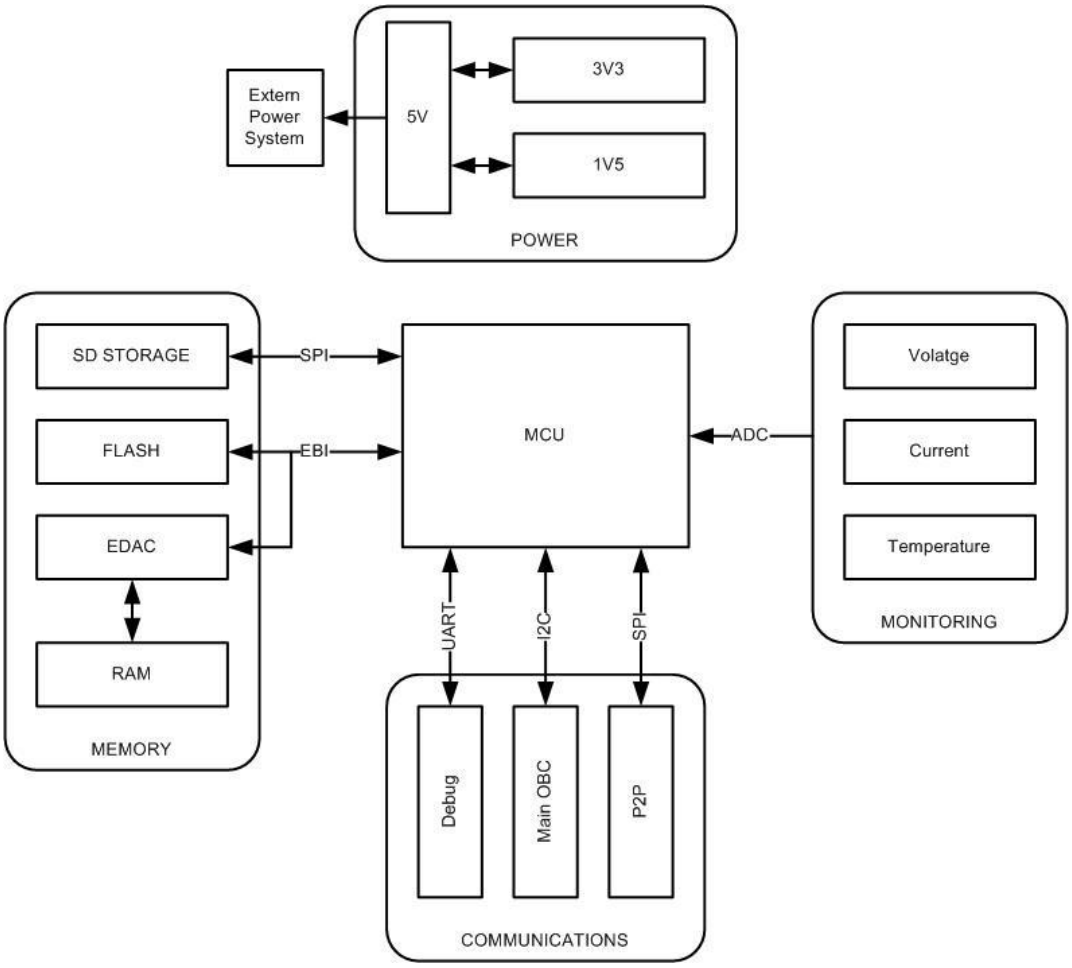


Figure 2.7: ADCS OBC Block Diagram.

2.3.1.1 WATCHDOG

An external watchdog is implemented on the OBC for the MCU. According to [19], the watchdog is a component with an internal countdown timer and is connected to the reset line of the MCU. Unless the watchdog is periodically toggled by the MCU (to reset the internal timer) it will cause the MCU to reset.

The MCU already includes an internal watchdog that should allow the MCU to reset in case the MCU becomes unresponsive due to a software error (runtime exception). Adding an external watchdog would add another level of reliability to the OBC with very little overhead (2.90 × 2.80 × 1.20 mm and 500 uA for the STWD100 [20]).

The watchdog used in this design is the STWD100 from ST Microelectronics [20]. The internal timer of the watchdog has to be reset within 1.6 s after the previous reset, which can easily be done when the real time clock generates an interrupt every second. If the internal

timeout is reached, the watchdog will reset the MCU for 210 ms which should allow the MCU to recover from its erroneous state.

2.3.1.2 CRYSTAL OSCILLATOR

Oscillators generate the signals that drive the MCU core and peripherals. The MCU includes internal *Resistor-Capacitor Oscillators* (RCOs). The high-frequency RCO has a configurable frequency range of 1 – 28 MHz and the real-time RCO has a frequency of 32.768 kHz. On a satellite many of the subsystems depend on the accuracy, precision and stability of the oscillator signals, also known as frequency stability. The following two examples illustrate the importance of frequency stability.

1. **Real-Time Clock Synchronization**

   For multiple OBCs it is important that the respective *Real-Time Clocks* (RTCs) are synchronized and increment at the same rate. If not, one OBC could schedule a task for another OBC at a time stamp that might have already elapsed according to its real-time clock. On an ADCS OBC, the need for an accurate RTC is even more important since some algorithms require the absolute time (SGP4) and a large time error will result in inaccurate ADCS control.

2. **Communication Synchronization**

   For communication peripherals a stable clock source on both the receiver and the transmitter is important. If the oscillator signals start to drift, the receiver might latch a data bit when the transmitter data is still changing its state. If the error is detected the data has to be retransmitted, or worse, the error is not detected and undefined data could cause a larger error somewhere else in the system.

These are two examples showing the importance for the use of stable oscillators. Unfortunately, the frequency stability of RCOs is sensitive to temperature, supply voltage and load variations, which are very difficult to control in the harsh space environment. The solution is to use *Crystal Oscillators* (XOs) which deliver better frequency stability even under varying conditions.

The OBC design includes the following two external crystal oscillators:

1. **High-Frequency Crystal Oscillator**

   A high-frequency, 32 MHz, oscillator is responsible for driving the core and all the high-frequency peripherals (ADC, EBI, I2C, etc). By using a *High-Frequency Crystal*

Oscillator (HFXO), the MCU core is also able to operate at its maximum frequency range of 32 MHz, which the internal RCO (maximum of 28 MHz) will not be capable of.

2. Low-Frequency Crystal Oscillator

A low-frequency, 32.768 kHz, oscillator is responsible for all the low-frequency peripherals including the real-time clock. A *Low-Frequency Crystal Oscillator* (LFXO) will make it easier to keep the RTCs synchronized between the main OBC and ADCS OBC.

For a detailed view of MCU and how it interfaces with the rest of the ADCS OBC refer to Appendix B.1.

2.3.2 EXTERNAL MEMORY SUBSYSTEM

The ADCS OBC includes an external memory subsystem. The MCU internally includes 256 kB of flash and 16 kB of SRAM memory. For an ADCS OBC, this might not be enough memory to implement all the required features. The OBC might need to have multiple programs for the bootloader to choose from, depending on the situation (safe mode, full operation, etc.). Large program stacks (multiple kilobytes as confirmed by testing) are required due to the complex ADCS algorithms, models, large data types and various telemetry related data. To ensure all of these features can be implemented, an external memory subsystem is added on the OBC, which is shown diagrammatically in Figure 2.8.

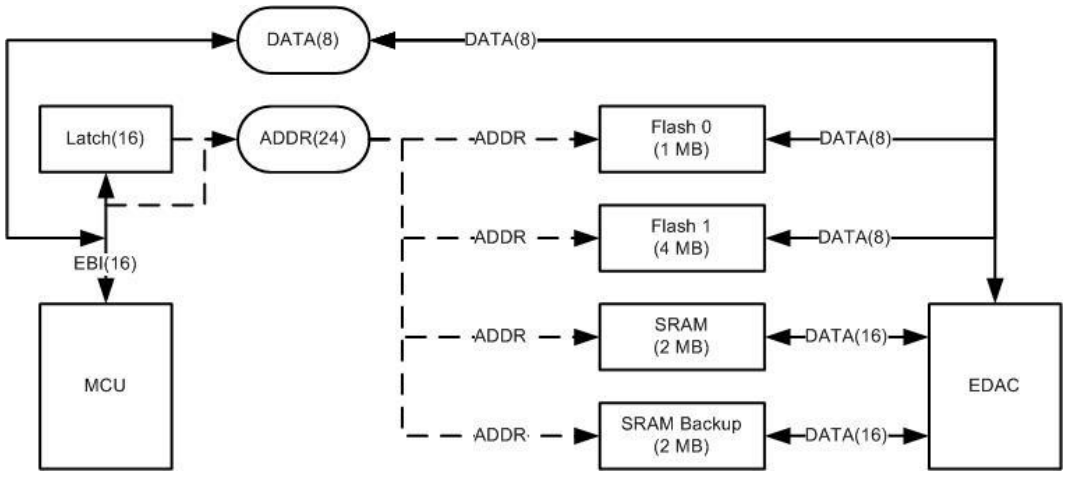


Figure 2.8: External Memory System of ADCS OBC.

2.3.2.1 MEMORY TYPES

Before the memory subsystem will be discussed in further detail, the differences between the SRAM and flash memory (the two main memory types used for most OBC designs) in terms

of performance and reliability will be looked at to understand the role they can fulfil and the challenges they will pose.

- **Flash**

  Flash memory is a non-volatile memory unit, in other words it retains its value even when not powered. It has very fast access speeds, but writing to flash memory is considerably slower than reading from it. Flash memory is also very resistant to radiation effects when being accessed, but it is susceptible to *Single Event Effects* (SEEs), such as SEU and SEL, when being programmed according to study done in [21]. Taking the above into consideration, flash memory is a very good storage candidate for program code. An MCU is programmed very seldom, therefore negating its weakness of slow programming time and SEEs. When continuously accessed during program execution it is very resistant to SEEs.

- **SRAM**

  Static Random Access Memory (SRAM) is a volatile memory unit. Unlike flash memory it retains its data only when powered. However, it has very fast read and write times which makes it an ideal candidate for storing program data which either rapidly changes their values or are merely temporarily allocated. Examples of these data types are variables used in the ADCS calculations, telemetry data temporarily stored before being transmitted to ground and sensor data which are periodically updated. Unfortunately SRAM is susceptible to radiation effects (SEEs) during both read and write operations [11]. To counteract the SEEs for this design an *Error Detection and Correction* (EDAC) module will be implemented for the SEUs and a robust power system will be implemented for the SELs.

2.3.2.2 EXTERNAL MEMORY REGIONS

The MCU includes an *External Bus Interface* (EBI) with four chip select lines which are used to access four external memory modules, each with a specific function.

1. **Flash 0 –** Safe Mode

   A flash memory module will be dedicated to storing a default safe mode operation program. This should be a simple program that will allow the satellite to make contact with the ground station, from where it will receive further commands. This should be the default program the ADCS OBC falls back on after a reset from an unresponsive state.

Because Flash 0 will only be storing one operating program, its capacity did not have to be as large as Flash 1 and was therefore chosen as one megabyte.

2. **Flash 1 –** Multiple Operating Programs

   An additional flash memory module will be used to store different programs which will execute after the safe mode program and the ground station have checked all the systems of the ADCS OBC. The reason for having multiple programs is that different programs are needed to handle different situations. Multiple programs can be used for reliability as well by storing the same program in three different memory locations as a form of triple mode redundancy. Extra programs can be uploaded from the ground station and stored in flash, either as corrections to existing programs or programs with different objectives. Because multiple programs will be stored on Flash 1, its capacity was chosen as four megabytes.

3. **SRAM –** Program Data

   An SRAM memory module is used for program data by the MCU. It is much safer to let the MCU store important data in the external memory, because of the safety measures implemented (EDAC and separate controllable power lines). If an SEU would occur while using the internal SRAM it can go unnoticed and could cause a serious error. If a SEL should occur while using the internal SRAM it would force the entire system (ADCS OBC) to reset. Therefore the internal SRAM will be powered down and only the external SRAM will be used. An SEU in the external SRAM will be detected by the EDAC and a SEL will prompt the MCU to try and recover the SRAM (by power cycling) before resetting the system. Besides the added safety of using the external SRAM, the internal SRAM is limited to 16 kB of memory while external SRAM can store much more data (2 MB is used for this design).

4. **SRAM –** Backup

   An extra SRAM memory module will be implemented as a backup due to the susceptibility of SRAM to radiation effects in space. This allows the MCU to reboot and use the extra SRAM module for the important data if a latchup should render the primary SRAM module unresponsive. This prolongs the time before it becomes necessary to use the internal SRAM of the MCU. This is important since the internal SRAM of the MCU has no form of EDAC and is therefore very susceptible to radiation effects. A SEL in the SRAM of the MCU could render the ADCS OBC useless.

2.3.2.3 EXTERNAL MEMORY INTERFACE

The EFM32G MCU used for the ADCS OBC includes an EBI which uses 16 GPIO lines for address and data transmissions to external asynchronous devices. This allows for a maximum 8-bit address and 8-bit data single-cycle throughput. To increase the range of the EBI, a multiplexed read/write operation can be implemented utilizing a latch. This allows for two additional EBI configurations: 16-bit address/data and 24-bit address/8-bit data. This design uses the 24-bit address/8-bit data configuration to maximize the memory map for each chip select line and therefore the maximum size of each memory module. The control of the latch and access to these external memories are fully automated and discussed during the driver implementations of the EBI.

2.3.2.4 ERROR DETECTION AND CORRECTION

An *Error Detection and Correction* (EDAC) subsystem is implemented to compensate for the susceptibility of SRAM to *Single Event Upsets* (SEUs) caused by the increased radiation level in space. The EDAC is very important since most of the data (not code) for the ADCS OBC is stored in SRAM and an undetected SEU (corrupt data) can cause undesired effects in the operation of the ADCS OBC.

Various forms of EDAC exist that can be implemented on an embedded system.

- **Software Based**

  A software-based EDAC system implements all the EDAC code within the MCU in the form of a driver between the program and memory. This removes the autonomous usage of the data stack by the MCU, which presents an extra overhead every time the SRAM is accessed for data. This is, however, very difficult to implement.

- **Hardware Based**

  Some manufacturers offer hardware implementations for flow-through EDAC systems, like the Atmel EDAC (29C516E [22]). These units are located between the MCU and SRAM on the data bus and work fast enough to enable them to encode and decode the data during a read or write cycle. However, they are usually expensive and inflexible which forces the memory design around these units.

- **FPGA**

  The FPGA-based EDAC system follows a hardware-based approach, but can be tailored to suit the requirements of a specific embedded design. A flow-through EDAC implementation can be designed that does not interfere with memory access. The FPGA

can also be used to implement other custom controllers needed on the OBC, such as an extra I2C controller, or port expander.

From the above descriptions, the FPGA-based EDAC system is the method best suited for this ADCS OBC design, due to its speed, flexibility and customization options. The ESL has experience in using FPGA-based EDAC systems in on-board satellite OBCs [23][24], which provided the necessary background and expertise to develop one for this project.

The EDAC system is implemented on an Igloo Nano (AGL030) FPGA from Actel [25]. This FPGA is designed for very low-power applications (efficiency) and the AGL030 version has no embedded SRAM which makes it less susceptible to radiation effects, ensuring reliability. The FPGA is also used on the CubeSense board design by the ESL [26].

2.3.2.5 SRAM BUS ISOLATION

Figure 2.9 shows the design for isolating the SRAM modules from the data and address bus. When an SRAM has been damaged due to a latchup and its power supply is turned off, the module can still draw current via the address/data bus lines and even affect the data on the bus. This happens because no supply current is available to drive the input/output pins of the SRAM module into high impedance. It is therefore important that the system is able to isolate the SRAM module completely from the address/data bus if it is powered down. This is achieved in this design by using bus switches which can be toggled by the MCU to either connect the SRAM module to the bus, or isolate it, by driving the ports to high impedance.

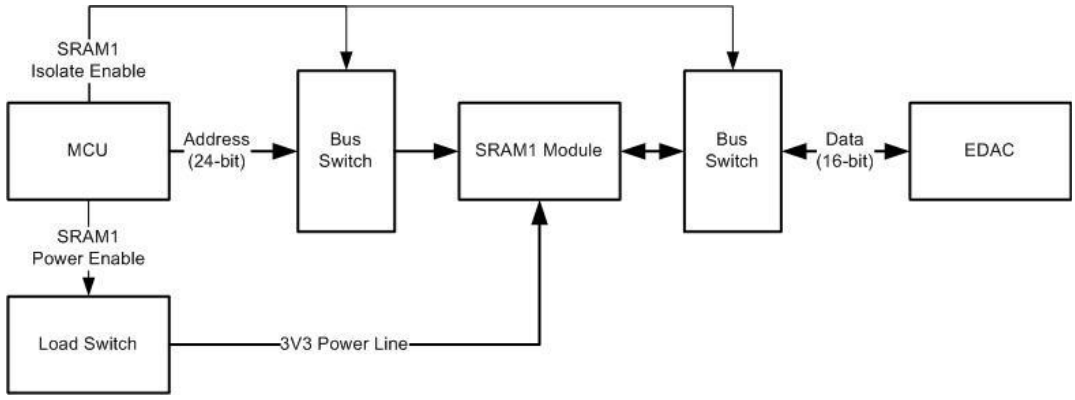


Figure 2.9: SRAM Isolation Design Diagram.

2.3.2.6 SD CARD STORAGE

The memory subsystem further includes a 2 GB microSD storage card. The SD storage is a non-volatile memory based on NAND flash technology and has exceptional capacity. The MCU uses the SPI interface to communicate with the microSD card. The SD storage offers

another form of long-term storage that could be used to log various telemetry data to be requested by the ground station at a later time. At the time of conducting this research, microSD cards do not have any flight heritage in space and is added as an experiment.

For a detailed view of Memory Subsystem and how it interfaces with the rest of the ADCS OBC refer to Appendix B.2.

2.3.3 POWER SUBSYSTEM

The power subsystem is responsible for providing all the components on the OBC with their required supply voltage and regulating the current during operation. The main *Electronic Power System* (EPS) supplies to all the subsystems a 5 V and 3.3 V power line which can be used to power most of the components on the ADCS OBC. However, a separate power system was specifically designed on the ADCS OBC to power the components. This improves reliability in two key areas, namely stability and control.

1. **Stability**

   When a power line is under heavy load (i.e. drawing heavy current) the supply voltage can drop due to undesired series resistance in the power line. This voltage drop can cause unwanted behaviour and even damage sensitive components. This can be prevented by regulating the required supply lines (3.3 V and 1.5 V) down from the 5 V power line supplied by the EPS. The output voltage will be very resistant to input voltage fluctuations as long as the input voltage drop remains above the specified dropout voltage of the regulator. The end result is a more stable supply voltage.

2. **Control**

   The advantage of designing a separate power system is that it supplies a means of control. The OBC can monitor the power system and detect if a too large current is being drawn or if a supply voltage is dropping too low. The OBC can then respond by power cycling the problematic power line to attempt to fix the latchup, or even switch it off completely. The OBC will then lose those components, but not the entire system.

Figure 2.10 shows the design of the power system in diagrammatic form, which will be broken down and discussed further below.

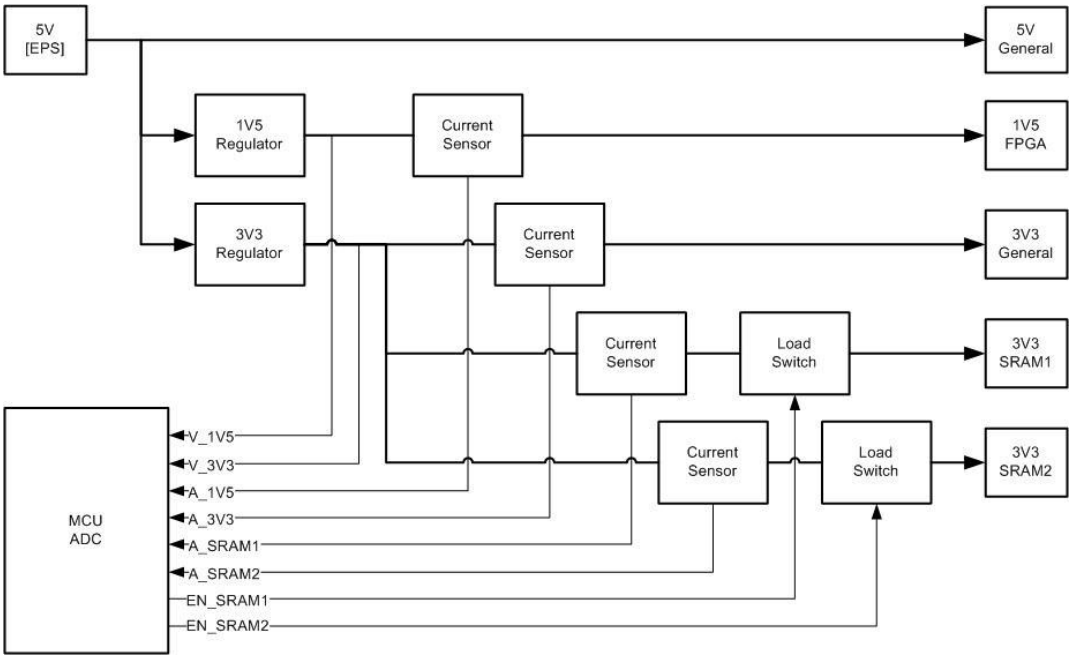


Figure 2.10: OBC Power System Block Diagram.

2.3.3.1 POWER SUPPLY LINES

A 5 V power line from the EPS is split into five power lines which are used in the ADCS OBC. These power lines are then used to power the various components.

- **General 5 V**

  This power line is the regulated 5 V power line supplied by the EPS. It is used to power the 5 V components on the OBC.

- **FPGA 1.5 V**

  This power line is regulated down from the 5 V power line supplied by the EPS. It is used to power the FPGA, which is the only component in the OBC that operates on a 1.5 V supply voltage.

- **General 3.3 V**

  This power line is regulated down from the 5 V power line supplied by the EPS. It is used to power the MCU and all other 3.3 V components, except the SRAM modules, which have their own power lines.

- **SRAM 3.3 V**

  Each SRAM module has its own power line. These power lines are regulated down from the 5 V power line supplied by the EPS. They power only their respective SRAM modules. The reason for this is that the power lines can individually be power cycled or disabled if a latchup should occur.

2.3.3.2 REGULATORS

The 3.3 V and 1.5 V lines are regulated using low-dropout linear regulators. The *Texas Instruments* (TI) TPS76733 [27] and TPS73215 [28] were used in this design for the 3.3 V and 1.5 V lines respectively. Both regulators offer exceptional voltage regulation at 0.01% change in output voltage over change in input voltage.

The 3.3 V and 1.5 V regulators have a current limit of 1 A and 250 mA respectively. Both of these limits are far higher than the required current according to preliminary power calculations. In the final version of the OBC, a regulator with a current limit closer to the nominal operating current can be selected. Therefore, if one of the components draws excessive amounts of current the supply voltage will drop and the brown-out-reset of the MCU will respond by resetting the OBC.

2.3.3.3 LOAD SWITCHES

Since SRAMs are the components on the OBC most susceptible to latchups, being able to switch the power to them on and off adds another layer of reliability. Power switches can be implemented with a simple PMOS circuit. However, a load switch from Fairchild Semiconductors (FPF2124 [29]) was chosen due to its current-limiting ability. For this design the current limit was configured to 200 mA, which is the rated latchup current for the Cypress Semiconductor SRAM (CY62167DV30 [30]) modules used in this design. When this current limit is reached and maintained for 10 ms the FPF2124 switches off the power line until it is toggled from an external controller (the MCU in the case of this design).

For a detailed view of Power Subsystem and how it interfaces with the rest of the ADCS OBC refer to Appendix B.5.

2.3.4 MONITORING SUBSYSTEM

Due to the harsh space environment, mentioned in section 1.4, the ability to continuously monitor the state of a system is very important. This allows the OBC to detect any abnormalities in the system which might be caused by faulty components or operations. These errors can either be fixed or even isolated before any serious damage is caused to the system.

The monitoring process should be sensitive and responsive: sensitive, in order for it to pick up the slightest change that could be a symptom of a larger problem and responsive, to be able to react quickly when a problem is detected.

As mentioned previously, the most common problem on an OBC is the susceptibility of SRAM to latchups due to radiation. This causes the SRAM to stop functioning as well as draw an excessive amount of current. This current, if not quickly dealt with, can damage the SRAM as well as other components on the OBC. A separate power line, with a current sensor and a load switch, was therefore implemented to try and isolate the SRAM modules in case of a latchup.

Figure 2.10, which illustrates the power system design, also shows how the monitoring system is implemented. The supply current and voltages are monitored which, from a hardware point of view, is the easiest way to detect any failures or malfunctions. The monitoring channels are as follows:

- **Voltage Sensors**

  The 3.3 V line and 1.5 V line is fed through a simple voltage divider circuit to the ADC unit of the MCU. The voltage measurements will be used for telemetry purposes. The MCU can also be configured to reset if the supply voltage drops below a certain threshold.

- **Current Sensors**

  The current from the general 3.3 V power line, 1.5 V FPGA power line and both SRAM power lines are monitored by current shunt sensors. These currents are sampled as voltages over output resistors and fed through an op-amp buffer to the ADC unit. This implementation of a current sensor is shown in Figure 2.11 and discussed in more detail in Appendix A.2.

- **Temperature**

  The MCU has an internal temperature sensor which will be used by the monitoring subsystem. The temperature measurements will be used mainly for telemetry purposes to assess the thermal situation in the ADCS unit and CubeSat.

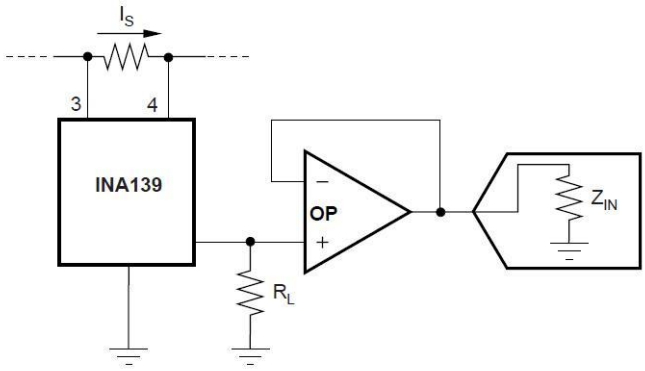


Figure 2.11: Current Sensor Implementation. [28]

All the above mentioned monitoring channels are sampled by the ADC unit of the MCU. The data gathered is stored for telemetry purposes, but could also be processed and allow the MCU to react autonomously to expected problems. Two examples used in this design are power cycling SRAM modules after a latchup and brown-out reset for the MCU.

2.3.5 COMMUNICATION SUBSYSTEM

The communication subsystem is responsible for allowing the ADCS to communicate with other subsystems on the CubeSat. These typically include the main OBC, sensors and actuators. The communication subsystem for this design is very simple from a hardware point of view because all the required communication peripherals are included with the EFM32G MCU. The following communication interfaces were implemented.

2.3.5.1 UART

The *Universal Asynchronous Receiver/Transmitter* (UART) is a simple point-to-point communication interface. Data is serially transmitted on one channel and received on another. Transmission is asynchronous in the sense that it can start at any time, but both transmitter and receiver should be set up with the same speed for operation.

In this design, the UART interface will be used mainly for debugging purposes. It can also be configured to program the MCU or merely used as a simple communication method for outputting data during the development of drivers and control modes.

2.3.5.2 I2C

The *Inter-Integrated Circuit* (I2C) is a high-speed bus-based communication interface. A master can access one of many slaves by using an address identifier to notify the correct slave of an incoming transmission. Through the use of arbitration, clock synchronization and

stretching, it is possible to allow for multiple bus masters if required. The I2C uses two IO lines for communication: a communal clock signal and a data bus line. The features of the MCU I2C controller is explained in more detail in [17].

In this design, the I2C bus is used as the communal bus where telecommands and telemetry data are transmitted to and from the main OBC to the rest of the CubeSat subsystems. In order for the ADCS OBC to communicate with its own subsystems (sensors and actuators) the I2C can either be shared, via time slots or arbitration, or a separate I2C bus can be used between the ADCS subsystems, which include the ADCS OBC, sensors and actuators.

2.3.5.3 SPI

The *Serial Peripheral Interface* (SPI) is a high-speed point-to-point communication interface. By synchronizing the communication between the transmitter and the receiver with a dedicated clock signal, transfer speed is only limited by the clock signal generated by the MCU. The SPI uses four IO lines for communication: a clock signal, a chip select signal, a transmit line and a receive line.

In this design, one SPI channel is used to communicate with the microSD card. Another could also be used for point-to-point access to a subsystem where large amounts of data transfer needs to take place. An example would be the ADCS OBC receiving an image file taken by a camera onboard the sun sensor or horizon sensor for telemetry or debugging purposes. Transmitting this large file over the communal shared bus (I2C in the case of most CubeSats) would unnecessarily occupy it for a long period of time and possibly prevent important communication to take place between the other satellite subsystems.

3 SOFTWARE DEVELOPMENT

The development of competent software is just as, if not more, important than the hardware design of an OBC. Software is responsible for instructing the different hardware components how and when to execute. The way in which these instructions are ordered (by means of a program) has to allow the system to achieve its intended purpose, which in the case of this project is the attitude and determination control of a CubeSat. Software programs are very complex and there are usually multiple implementations that achieve the same objective. It was therefore important that the software implementations for this project take into account the design requirements (performance, efficiency and reliability), because they apply to software as well as hardware.

This chapter starts from a low-level perspective by looking at the hardware abstraction layer supplied by ARM for its Cortex-M range architecture. A quick summary will then be given of all the drivers that were developed in order to utilize the hardware that was designed in the previous chapter. After that a section will be dedicated to the development of the error detection and correction algorithm and code that was implemented on the FPGA for the SRAM. Lastly, higher-level software will be discussed by looking at the operation of the bootloader and operating system with its control loop and background tasks.

3.1 HARDWARE ABSTRACTION LAYER

ARM uses a *Hardware Abstraction Layer* (HAL) called the *Cortex Microcontroller Software Interface Standard* (CMSIS) [13]. A HAL is low-level software that supplies a simple interface to access hardware features from various vendors or manufacturers. In the case of ARM, this standard is called the CMSIS and the interface allows access to the processor and peripherals on all microcontrollers based on the ARM Cortex-M architecture. Figure 3.1 shows the structure of the CMSIS and how it fits between user-developed code and Cortex-M processor-based MCUs.

The advantage of using a HAL is that it is backed by a large software community. Previously there were separate communities for each ARM-based microcontroller vendor. However, the Cortex-M community is now vendor independent, because all the software is based on the CMSIS. This greatly improves the ease with which code can be developed for a Cortex-M-based microcontroller due to the amount of code available and the ease at which it can be re-used. The EFM32G, the MCU used for this design, is a Cortex-M3-based MCU and its software is also based on the ARM CMSIS. Even though the EFM32G was released in

2009, a large pool of application code is already available due to the ease of developing code for the Cortex-M based MCUs, which can be found on their website [31].

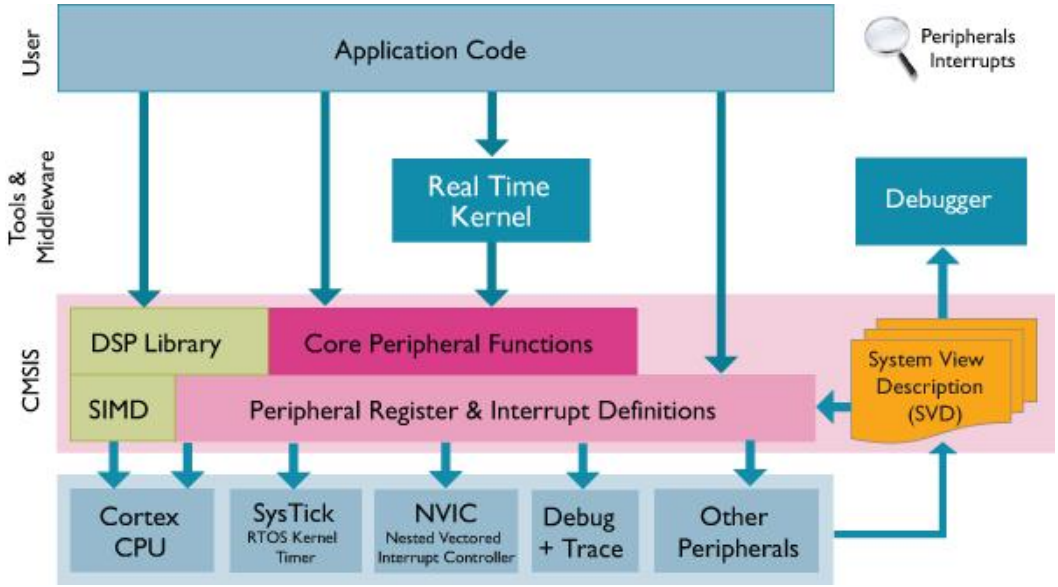


Figure 3.1: ARM CMSIS Structure. [9]

3.2 DRIVERS

A device driver is a piece of code that allows the operating system or user application to set up, access and utilize a specific hardware device and/or peripheral. Drivers are very useful since they allow an application to access the features of the OBC without any low-level knowledge such as registers, protocols and timings. The use of the CMSIS has greatly increased the ease with which drivers for the Cortex-M-based MCUs are developed, because of the simple and standard interface to low-level structures such as registers and interrupts.

For the EFM32G MCU, drivers for most of the peripherals and MCU features have already been developed. However, for this project, most of the drivers were either modified or redeveloped around the design requirements for the ADCS OBC in order to maximize the performance, efficiency and reliability of the peripherals, hardware components and the overall system. The following subsections are summaries of all drivers developed and used by the ADCS OBC in this project.

3.2.1 EXTERNAL BUS INTERFACE

The *External Bus Interface* (EBI) is responsible for interfacing the MCU with the external memory (flash and SRAM) on the OBC. The driver configures the settings that allow the MCU to read from and write to these external memories as well as the location of these

memories within the system memory map. After the driver has been set up, the external memory can be accessed by software the same way in which the internal memory is accessed.

The most important setting in the EBI is the synchronization of the read and write cycles between the MCU and the flash/SRAM. The timings between the different control signals (EBI_ALE, EBI_CSn, EBI_WEn, EBI_REn) and address/data bus (EBI_ADD[0:15]), shown in Figure 3.2 and Figure 3.3, can be configured in clock cycles of the MCU's high-frequency core clock. These timings need to be slack enough to allow both flash and SRAM to function within its operational limits, yet tight enough not to waste any MCU clock cycles and therefore power and performance.

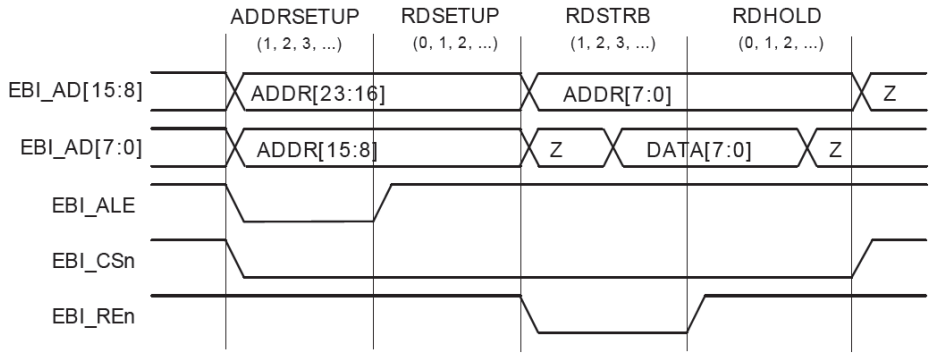


Figure 3.2: EBI Read Operation. [12]

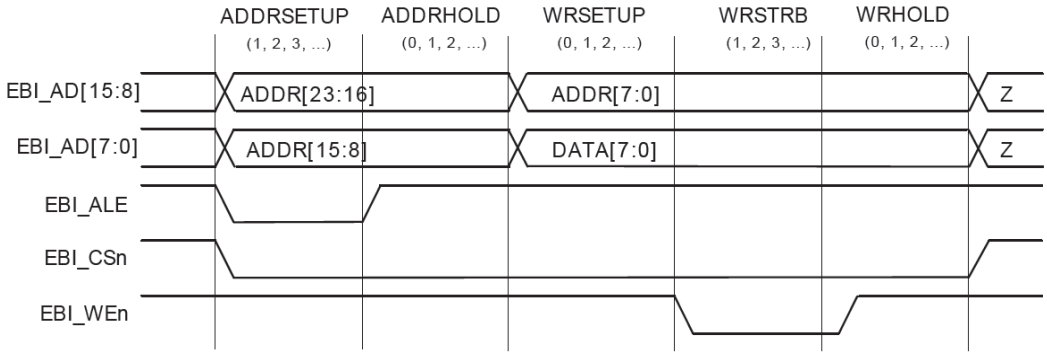


Figure 3.3: EBI Write Operation. [12]

3.2.2 DIRECT MEMORY ACCESS

The *Direct Memory Access* (DMA) is responsible for transferring data between peripherals and memory without the intervention of the MCU core. Some of the tasks on the OBC fit this description (monitoring and data transmission) and by utilizing the DMA capabilities of the

EFM32G, the MCU core can spend more time in a lower energy mode, which results in better efficiency.

For a DMA transfer to take place a channel has to be set up between the two involved parties (peripheral/memory and memory/peripheral) and a call has to be made that signals the start of the transfer. The DMA can also be configured to execute a special function (callback function) whenever a DMA transfer is complete. Figure 3.4 shows the implementation of the DMA in the UART driver during data transmission.

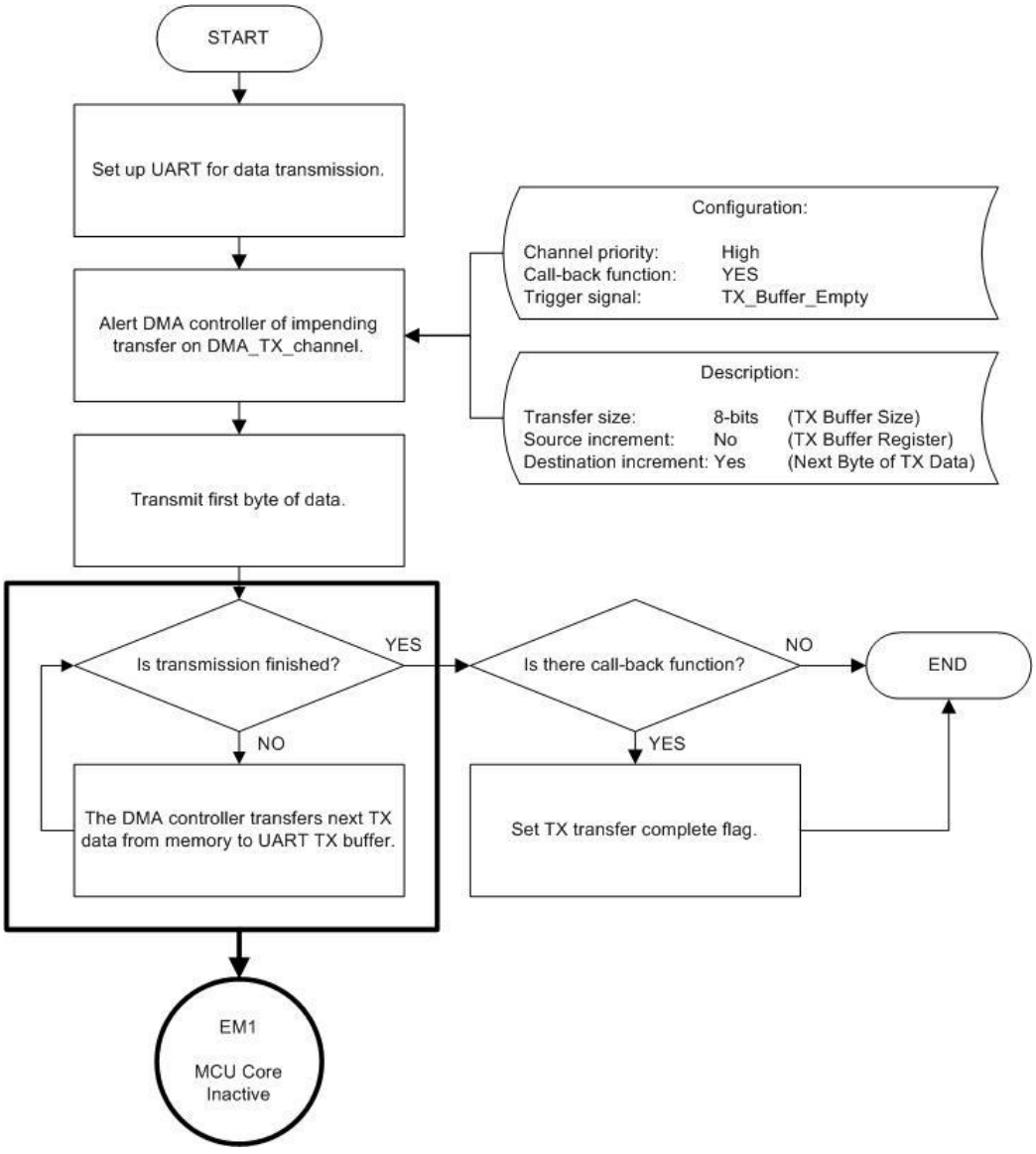


Figure 3.4: Flowchart of DMA Transfer for UART.

3.2.3 ANALOG TO DIGITAL CONVERTER

The ADC is responsible for sampling the analog channels for the monitoring subsystem. These channels include supply voltages (3.3 V, 1.5 V), load currents (3.3 V, 1.5 V, SRAM 1&2) and temperature (on-chip sensor).

The ADC is implemented on the ADCS OBC in the following two ways:

- **Periodic**

  Some of the ADC channels are sampled periodically. The sample period and channels to be scanned can be defined by a telecommand from the main OBC. After each period, the ADC is set up and scans the selected channels. The sampled values are then stored for telemetry purposes. The ADC is configured to sample at maximum frequency which allows it to finish as fast as possible. A DMA transfer is initialized before the scan, which allows the MCU to either process another task or to enter Stop Mode (EM1) to conserve energy.

- **Continuous**

  Some of the analog channels are continuously monitored with an analog comparator and compared to reference values. If this reference, or threshold, value is exceeded, an interrupt is generated and the MCU can react accordingly. An example of using the analog comparator on this OBC is for detecting a latchup in the external SRAM modules. The SRAM current for each module is continuously monitored. If the sampled current exceeds the rated SRAM latchup current, an interrupt is generated to the MCU which addresses the problem. This process is shown in Figure 3.5.

Table 3.1 shows the different functions supplied by the ADC driver, which are described in more detail in Appendix D.3.

Table 3.1: ADC Driver Functions.

Function Name	Description
BSP_ADC_Init	Initializes the clock, DMA channel and settings for the ADC.
BSP_ADC_Scan	Starts ADC scan of all channels.
BSP_ADC_IsScanComplete	Returns true if a scan is in progress, otherwise false.
BSP_ADC_GetAllData	Returns all the values of the latest ADC scan.
BSP_ADC_GetData	Returns the latest sampled value of the specified ADC channel.

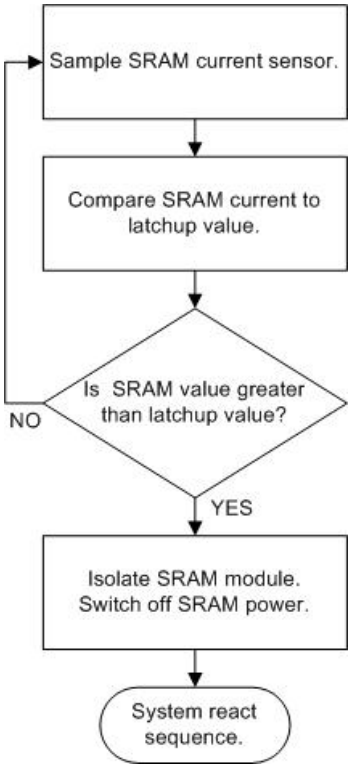


Figure 3.5: SRAM Latchup Detection Algoritm.

3.2.4 REAL-TIME CLOCK

The RTC allows the OBC to have an independent time keeping system. This is important on satellites, especially those with multiple OBCs, for avoiding conflicts. Implementing and synchronizing an accurate RTC onboard all the required subsystems (ADCS OBC, main OBC and ground station) will result in cooperation throughout the satellite.

The RTC in this design uses the external crystal oscillator due to the extra stability it offers over an RC oscillator. The RTC creates an interrupt every second which is used to tick the *Unix Time Counter* (UTC) and clock-calendar system, if enabled. The UTC timestamp is the time in seconds passed since 00:00 on 1 January 1970 [32]. These timestamps are popular in software programs because they allow time to be represented as one variable and not as days, hours, minutes, seconds, etc. which holds not as much significant value for a computer program as a human being.

Table 3.2 shows the different functions supplied by the RTC driver, which are described in more detail in Appendix D.1.

Table 3.2: RTC Driver Functions.

Function Name	Description
BSP_RTC_Init	Initializes the clock, counter and interrupt for RTC.
BSP_RTC_SetUnixTime	Sets the Unix time counter value.
BSP_RTC_GetUnixTime	Gets the Unix time counter value.
BSP_RTC_IncUnixTime	Increments the Unix time counter with a predefined value.

3.2.5 WATCHDOG

The OBC design includes two watchdogs, one internal and one external. Both watchdogs have to be periodically toggled within a certain time period; otherwise it will force the MCU to reset. This is an important reliability feature that safeguards the MCU against a software lockup, which will put the MCU, and therefore the OBC, in an unresponsive state. The MCU is the core of the OBC and in order for it to survive the harsh environment of space, as much reliability as possible should be implemented.

In this design, the two watchdogs are toggled in separate areas of the operating system (foreground and background). This prevents the MCU from entering a logic loop, which can occur if both watchdogs are toggled in the background by an interrupt while the foreground application is in an unresponsive state. This is illustrated in Figure 3.6 and Figure 3.7.

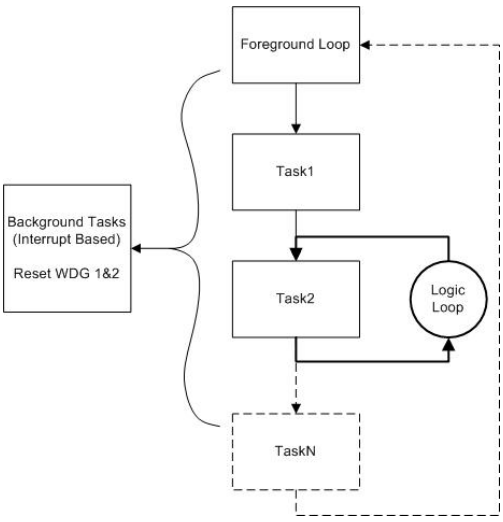


Figure 3.6: Logic Loop.

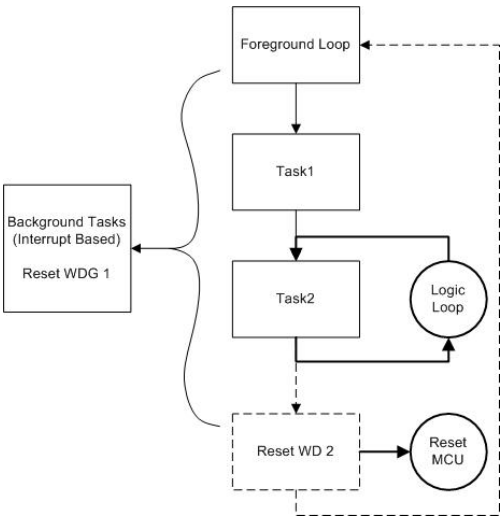


Figure 3.7: Logic Loop Prevention.

Table 3.3 shows the different functions supplied by the watchdog driver, which are described in more detail in Appendix D.2.

Table 3.3: Watchdog Driver Functions.

Function Name	Description
BSP_WDG_Init	Initializes both the internal and external clock and their timers.
BSP_WDG_ToggleExt	Toggles the external watchdog to reset its counter.
BSP_WDG_ToggleInt	Toggles the internal watchdog to reset its counter.

3.2.6 UART

The *Universal Asynchronous Receiver/Transmitter* (UART) is used in this design primarily for debugging purposes. Since the UART is only an interface for communication, it is ideal for testing inter-subsystem communication, protocols and synchronization because it is simple to implement and able to interface with a *Personal Computer* (PC) to emulate any other subsystem. PC emulation is very important because it allows the design to be debugged to a certain degree without having the full satellite system available.

For the OBC UART to communicate with the PC, its signals should be converted to RS232 voltage levels used by the serial port interface on the PC. An RS232-to-UART converter was designed and used only between the OBC and a PC. This was not implemented onboard the OBC, because it will not be required in the satellite and therefore only waste power. The debug drivers were developed to emulate communication with any subsystem (sensor, actuator and OBC) and to periodically or asynchronously output the OBC's telemetry to the PC.

3.2.7 SPI

The *Serial Peripheral Interface* (SPI) is used in this design to interface with the microSD card. It offers a high-speed synchronized link between the MCU and the SD storage. The MCU includes two other SPI interfaces that can be used for large point-to-point data transfers between the MCU and another subsystem, for example the transfer of images between the ADCS OBC and the CubeSense (sun and nadir sensor) for debugging purposes.

The microSD card is accessed by an application through the *File Allocation Table File System* (FATFS). The FATFS accesses the microSD card through a DiskIO driver which was supplied by Energy Micro. This DiskIO driver is a specially modified driver only intended for use by the FATFS. The interface between an application and the microSD card is shown in Figure 3.8.

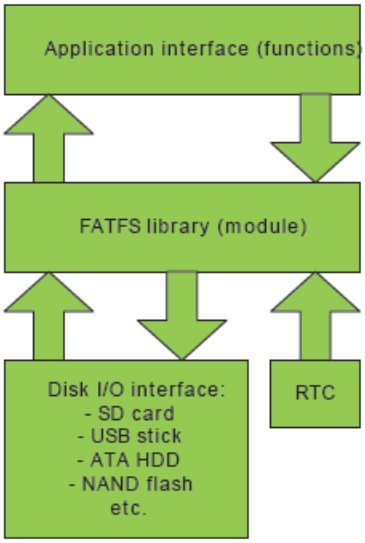


Figure 3.8: Interface between an Application and the MicroSD Card. **[45]**

Table 3.4 lists the different functions supplied by the microSD driver which are described in more detail in Appendix D.4.

Table 3.4: MicroSD Driver Functions.

Function Name	Description
BSP_MSD_Init	Initializes the SPI, microSD card and FATFS.
BSP_MSD_Write	Writes data to the LOG file on the microSD card.
BSP_MSD_Read	Reads data from LOG file into buffer variable.
BSP_MSD_Test	Executes a test program for the microSD card.

3.2.8 I2C

The *Inter-Integrated Circuit* (I2C) is used in this design to interface with the main communication bus, which is used mainly by the main OBC (master) to communicate with all the other major subsystems (slaves).

Because the ADCS OBC also has to interact with its own subsystems (sensors and actuators) the main I2C bus must allow for multi-master usage. The ADCS OBC already has the required features (arbitration and clock synchronization) to communicate simultaneously with the main OBC, but an easier solution would be to allow each OBC a specified time slice within which it has to communicate with its subsystems. Another option would be to have the ADCS OBC implement a separate I2C bus for communicating with its subsystems, while still acting as a slave on the main I2C bus for telecommands. The different I2C bus implementations are shown in Figure 3.9, Figure 3.10 and Figure 3.11.

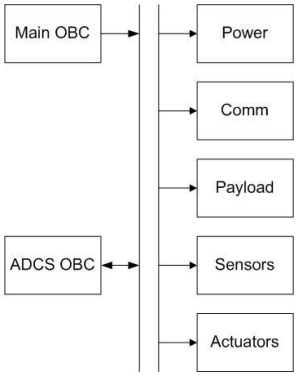


Figure 3.9: Multi Master Bus.

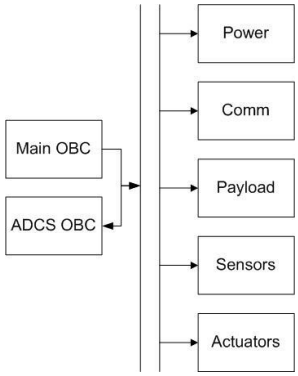


Figure 3.10: Time Shared Bus.

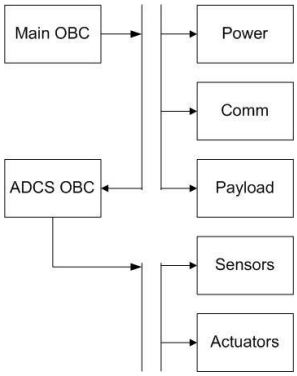


Figure 3.11: Separate Buses.

The multi-master bus configuration will allow the main and ADCS OBC access to all subsystems but it will be the most complex of the three configurations to implement due to potential bus conflicts. The time-shared bus configuration will be easier to implement and avoid any potential bus conflicts but will not maximise the throughput of the bus. A separate bus configuration will allow both main and ADCS OBC maximum throughput but at the cost of isolating some of the subsystems from the main OBC. Therefore, the I2C bus configuration will not be fixed for the ADCS OBC unit since the final implementation should take into account the requirements for and limitations of the CubeSat system and the mission for which it will be used. The eventual I2C driver will therefore depend on the bus configuration being used.

3.3 Error Detection and Correction

The *Error Detection and Correction* (EDAC) subsystem is implemented as a flow-through EDAC situated on the data bus between the SRAM and MCU. A flow-through EDAC has the advantage of converting the data (detecting and correcting) in real-time without affecting the read/write commands of the MCU. This simplifies the programming of the MCU and adds no memory access overhead. The EDAC design in this system uses linear block codes, which is implemented on an FPGA, to encode and decode data between the MCU and SRAM. When errors are detected and/or corrected, the FPGA signals the MCU which then reacts accordingly.

3.3.1 Linear Block Codes

A popular form of EDAC used on embedded systems is *Linear Block Codes* (LBC). Generally for LBCs a codeword $\boldsymbol{c}$ exists for every data word $\boldsymbol{d}$. These words can be represented as vectors:

$$\boldsymbol{d} = (d_1, d_2, \dots, d_k), \qquad \boldsymbol{c} = (c_1, c_2, \dots, c_n), \qquad (n > k) \tag{3.1}$$

The amount of data digits, *k*, divided by the amount of code digits, *n*, is known as the LBC's information rate.

A codeword $\boldsymbol{c}$ can be obtained by multiplying its data word $\boldsymbol{d}$ by a $(k \times n)$ generator matrix $\boldsymbol{G}$, using modulo-2 arithmetic. A special form of $\boldsymbol{G}$ exist which generates a codeword in systematic form. This type of generator matrix $\boldsymbol{G}$ consists of an $(k \times k)$ identity matrix $\boldsymbol{I}$ and a $(k \times m)$ matrix $\boldsymbol{P}$, where $m = n - k$. A codeword in systematic form is the same as its data word for the first *k* digits, while the remaining *m* digits are linear combinations of the data word, called the parity-check digits $\boldsymbol{c}_p$.

$$\boldsymbol{c} = \boldsymbol{dG} \tag{3.2}$$

$$= \boldsymbol{d}[\boldsymbol{I}_k(k \times k), \boldsymbol{P}(k \times m)] \tag{3.3}$$

$$= [\boldsymbol{d}, \boldsymbol{c}_p] \tag{3.4}$$

When a codeword **r** is retrieved, it is uncertain if and how many errors it contains. A $(n \times m)$ parity check matrix $\boldsymbol{H}$ exists for which the following condition holds:

$$\boldsymbol{cH}^T = 0 \tag{3.5}$$

If this condition is true for $\boldsymbol{r}$, then $\boldsymbol{r} = \boldsymbol{c}$ and the codeword does not contain any errors. If the codeword $\boldsymbol{c}$ is in systematic form it can be decoded by only using the first *k* digits of $\boldsymbol{c}$, which is equal to the data word $\boldsymbol{d}$.

$$\boldsymbol{rH}^T = 0 \tag{3.6}$$

$$\therefore \boldsymbol{r} = \boldsymbol{c} = [\boldsymbol{d}, \boldsymbol{c}_p] \tag{3.7}$$

$$\therefore \boldsymbol{d} = [\boldsymbol{r}_1, \boldsymbol{r}_2, \dots, \boldsymbol{r}_k] \tag{3.8}$$

If Equation (3.5) is not true for a received codeword $\boldsymbol{r}$, due to an error signal $\boldsymbol{e}$, the resultant row vector $\boldsymbol{s}$ in Equation (3.10) is known as the syndrome. If only one error is made, it can be corrected by comparing the syndrome $\boldsymbol{s}$ to the parity check matrix $\boldsymbol{H}$. The row vector which the syndrome corresponds to within $\boldsymbol{H}^T$ is the digit in codeword $\boldsymbol{r}$ that has to be corrected (i.e. flipped).

$$\boldsymbol{s} = (\boldsymbol{s}_1, \boldsymbol{s}_2, \dots, \boldsymbol{s}_m) \tag{3.9}$$

$$\boldsymbol{s} = \boldsymbol{rH}^T \tag{3.10}$$

$$\boldsymbol{s} = (\boldsymbol{c} \oplus \boldsymbol{e})\boldsymbol{H}^T \tag{3.11}$$

Because of Equation (3.5),

$$\boldsymbol{s} = \boldsymbol{e}\boldsymbol{H}^T \oplus 0 = \boldsymbol{e}\boldsymbol{H}^T \tag{3.12}$$

$$\boldsymbol{s} = [e_1 \quad e_2 \quad \dots \quad e_n] \begin{bmatrix} h_{11} & \cdots & h_{1m} \\ \vdots & \ddots & \vdots \\ h_{n1} & \cdots & h_{nm} \end{bmatrix} \tag{3.13}$$

A single bit error in digit i,

$$\boldsymbol{e} = [0, 0, \dots, e_i, \dots, 0] \tag{3.14}$$

$$\therefore \boldsymbol{s} = [h_{i1}, h_{i2}, \dots, h_{im}] \tag{3.15}$$

If more than one error has occurred in codeword $\boldsymbol{r}$, up to a certain limit, the syndrome $\boldsymbol{s}$ will not correspond to any of the row vectors in $\boldsymbol{H}^T$ but a non-zero value will indicate a detected error.

This is a very short description of linear code theory. For a more in-depth description, please refer to [33], page 729.

3.3.2 REED-MULLER CODE

Many LBCs exist which vary in their code lengths, parity generator matrices and the amount of detectable and correctable errors. For this design, the following three requirements were set for an LBC:

- **Reliability –** The LBC shall correct at least one error and detect as many as possible.
- **Performance** – The LBC shall generate systematic codewords which can easily be decoded using the first 8-bits of a codeword.
- **Efficiency –** The LBC shall have an information rate of fifty percent which corresponds to the MCU data bus (8-bits) over the SRAM data bus (16-bits) and therefore does not waste memory space.

The *Reed-Muller*- (RM) (1, 3) code [34] fulfils the above criteria as it can correct one error and detect up to three errors and because it has a systematic codeword generator matrix and an information rate of fifty percent. However, RM (1, 3) only produces an 8-bit codeword (four data digits and four parity check bits), while a 16-bit codeword (eight data digits and eight parity check bits) is required for optimal SRAM usage. This can be compensated for by splitting the received data word from the MCU into two 4-bit data words and implementing the RM (1, 3) LBC twice within the FPGA (shown in Figure 3.12). Two RM (1, 3) LBCs together will produce a 16-bit codeword and have an added advantage of being able to correct

up to two errors simultaneously, which a single LBC would not have been able to achieve without noticeable coding overhead.

3.3.3 IMPLEMENTATION

The EDAC is implemented on the FPGA as shown in Figure 3.12. Depending on the read /write command from the MCU, the data can flow through the FPGA in the following two ways:

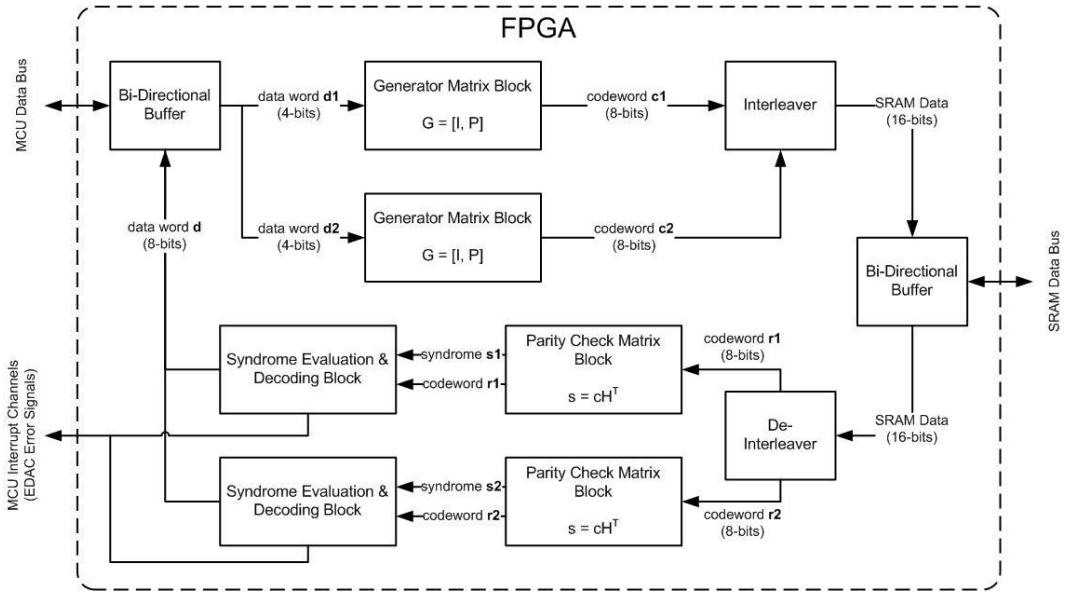


Figure 3.12: Error Detection and Correction Subsystem on FPGA.

1. Write Command

For a write command the data flows from the MCU data bus into the bidirectional buffer. The data is split up into two 4-bit channels, due to the Reed-Muller (1, 3) block length, and sent to the Generator Matrices ***G***. The resulting codewords ***c1*** and ***c2*** are interleaved together to produce a 16-bit data word which passes through the bidirectional buffer to be written in the SRAM.

2. Read Command

For a read command the 16-bit data word flows from the SRAM bus through the bidirectional buffer and into the deinterleaver. The resulting codewords ***r1*** and ***r2*** is multiplied by the parity check matrices ***H*** to produce the syndromes ***s1*** and ***s2***. These syndromes are then inspected to validate the codewords ***r1*** and ***r2***. If an error is detected and corrected, a signal is flagged for the MCU. If errors are detected but could not be corrected, a different signal is flagged for the MCU. If no errors are detected, the

codewords are decoded and passed through the bidirectional buffer onto the MCU data bus.

The function of the interleaver is to spread out the codewords ***c1*** and ***c2*** evenly throughout the SRAM data word. This is to reduce the likelihood of one high-energy particle causing multiple upsets (errors) within the same codeword and should subsequently increase the effectiveness of the EDAC subsystem.

3.3.4 ERROR HANDLING

The FPGA error signals (default high) are driven low according to Table 3.5 whenever an error is detected by the RM (1, 3) LBC. These signals are connected to the MCU and generate an interrupt when they change from their default value (1111b), i.e. on falling edge. The signal values are then interpreted and the MCU reacts accordingly.

Table 3.5: Error Signal Code Descriptions and MCU Reaction.

Error Signal Value	Error Signal Description	MCU Reaction
1111b	No errors detected	None
1110b / 1101 b 1011b / 0111b	One error detected and corrected	Memory Wash
1010b / 1001b 0110b / 0101b	Two errors detected and corrected	Memory Wash
00xxb / xx00b	Multiple errors detected and not corrected	Variable Integrity Check

xx = error values for other RM(1,3) code block

When an error signal is received by the MCU, it can respond to the error in the following ways:

- **Memory Wash**

  When one or two correctable errors are detected by the FPGA, the MCU will schedule a memory wash. A memory wash is when the MCU reads and re-writes all data on an entire memory module. This will clean all detectable and correctable errors which will prevent a build up of errors within a memory module. Memory washes can also be scheduled to run every few seconds even if no errors were detected on accessed memory. This will prevent the build up of errors in the memory module and especially bytes which might not be accessed as often.

- **Variable Integrity Check**

  When multiple uncorrectable errors are detected by the FPGA, the MCU will schedule a variable integrity check. Because the majority of data in SRAM are temporary variables created on the program stack, or variables updated every control loop cycle (sensor data, actuator commands), corrupt data will not necessary negatively affect the operation of ADCS. However, some data are considered critical, such as the operating parameter of the ADCS, and if an uncorrectable error is detected, a variable integrity check will be initialized that scans these critical variables. Each critical variable will have a copy which can be used for correction if it has become corrupt. If both the critical variable and its copy are corrupt, the ADCS OBC will have to be reset.

3.4 BOOTLOADER

The bootloader is a small program that runs every time the MCU resets. The function of a bootloader is to load a selected operating program into memory for execution. This allows the OBC to have more than one operating program. This has two major advantages for an OBC:

1. **Flexibility**

   An operating program is usually developed with a certain situation, responsibility and requirements in mind. By having multiple operating programs, each with a different focus, the OBC can better adapt to a situation by booting the appropriate program instead of developing one complex program to try and handle every situation. For this design, the OBC will have an operating program developed with ADCS as its focus while another backup program could allow the OBC to act as a main OBC in case of emergency.

2. **Reliability**

   When designing software programs, some logic errors or limitations only emerge during in-flight usage. The ability to upload operating programs from the ground station to the satellite after launch allows the developers to change the operation program to correct errors and compensate for limitations.

A bootloader adds flexibility and reliability to a system, but care must be taken to create a failsafe boot sequence. It is possible for a poorly designed bootloader to enter a state referred to as a “reboot loop” which lets the OBC continuously boot from a broken operating program without allowing any form of intervention from the outside.

3.4.1 PROGRAM TYPES

The OBC has two different types of programs which the bootlader can boot the MCU from:

- **Safe Mode**

  The safe mode program is a simple operating program that aims to put the satellite in a stable orientation using the minimum amount of sensors and actuators. This program is stored in a reliable memory module (Flash 0) separated from other operating programs. The safe mode program should be the default program the ADCS OBC falls back on when an unexpected reset or error occurs.

- **Nominal Programs**

  A nominal program is a more complex operating program developed to make full use of the ADCS OBC's subsystems and features to achieve its goal. More than one nominal program can be uploaded and stored on the OBC. The nominal programs are stored within a table structure on a flash memory module (Flash 1); each at a separate index. To boot from a specific nominal program, a boot index variable must be changed to the index of the desired program which the bootloader will then load into the internal flash memory of the MCU after a reset.

3.4.2 IMPLEMENTATION

The bootloader sequence proposed for this design is presented in Figure 3.13. After a reset, the bootloader starts by configuring the EBI of the MCU as both the safe mode program and nominal operating programs are stored in the external memory. The bootloader waits a few seconds for a telecommand from the main OBC which indicates which operating program to boot from. If no command is received, the bootloader reads a counter which counts the amount of times the bootloader has recently been executed. If this value exceeds a certain threshold, it indicates that the bootloader has entered a reboot loop.

A reboot loop is when the bootloader boots a faulty program which is eventually reset by the watchdog. After the reset the bootloader executes and reloads the faulty program which eventually resets again and again. This might occur when a program has a logic error which will not be detected by a CRC check, but will cause the program to stall and force the watchdog to reset the OBC. After each reset, the bootloader will increment the counter and an unexpected high value for the counter will indicate to the bootloader that it has entered a reboot loop. The bootloader will respond by booting the safe mode program until commanded otherwise from the main OBC. The counter can be reset via a telecommand from the main

OBC if the ground station has verified that the program is running correctly or a new program has been uploaded for execution.

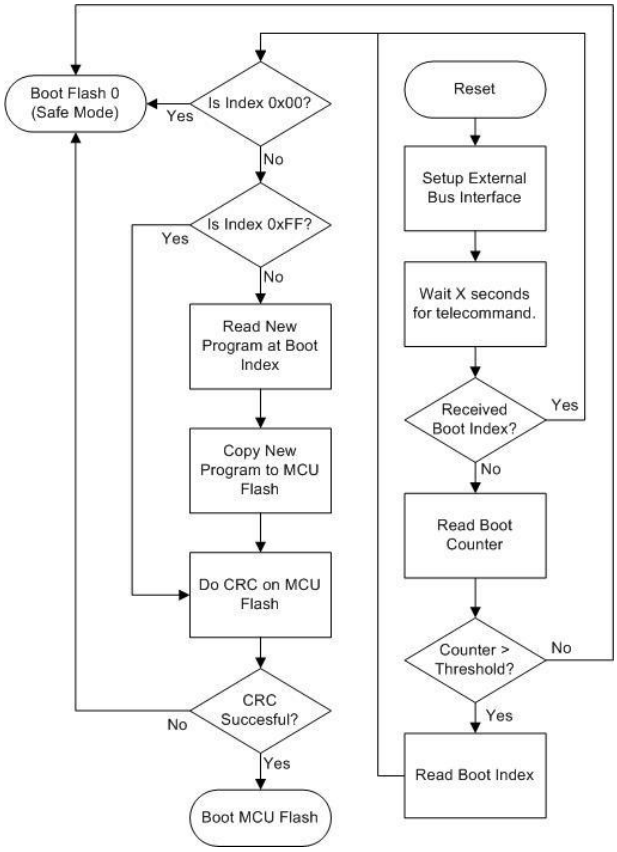


Figure 3.13: Proposed Bootloader Sequence.

If the counter has not exceeded the threshold value, it reads a special byte, in protected memory of Flash 1, which stores the boot table index of the next program to be loaded into the flash memory of the MCU for execution. The value of this index defines what the bootloader does next:

- **0xFF –** The bootloader executes the current program in the flash memory of the MCU.
- **0x00 –** The bootloder executes the safe mode program located in Flash 0.
- **0xXX –** The bootloader loads a new program into the flash memory of the MCU which is located in Flash 1 at index *XX* of the boot table.

After the program is loaded into the Flash of the MCU (or not in the case of 0xFF), a CRC check is done to verify the integrity of the program in internal MCU Flash. If this check is unsuccessful, which indicates that the program is corrupt, the bootloader boots the safe mode program.

3.5 OPERATING SYSTEM

The *Operating System* (OS) is responsible for managing the different tasks that need to be executed by the OBC. These tasks usually fall into one of two categories: foreground application and background services. The operating system, together with the ADCS application and background services, are shown in Figure 3.14.

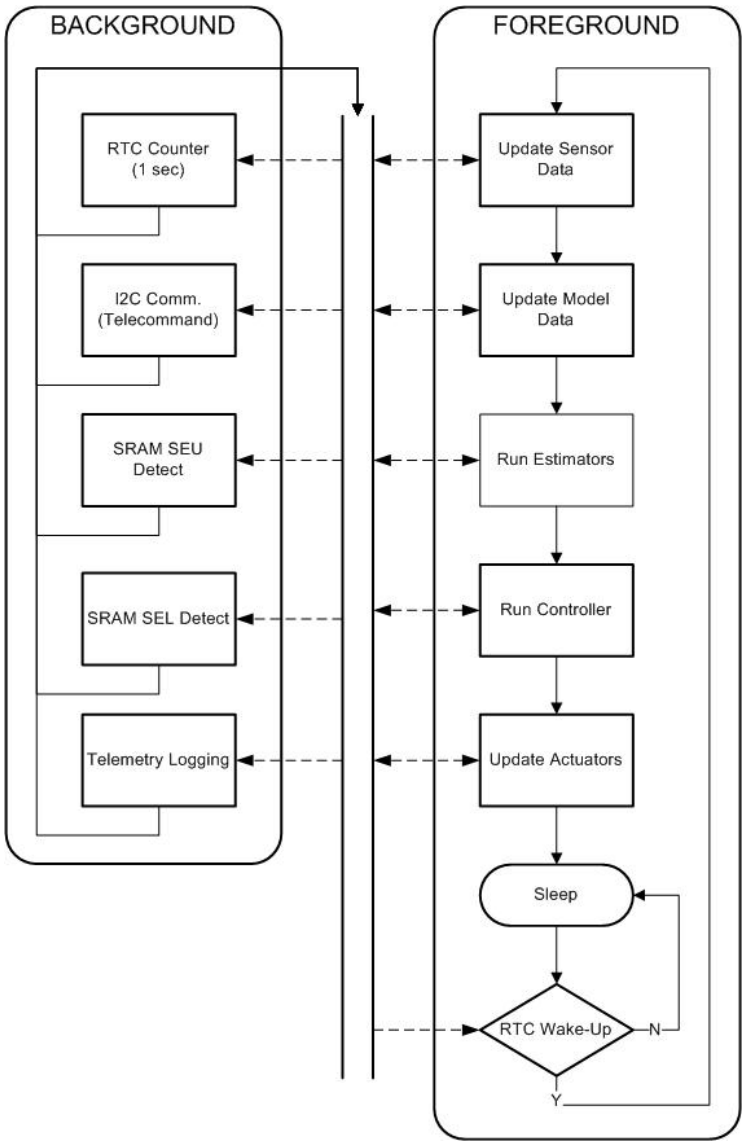


Figure 3.14: Operating System Flow Diagram for the ADCS OBC.

3.5.1 FOREGROUND APPLICATION

The foreground application is a collection of tasks which aim at achieving a single goal. These tasks can each be separately responsible for utilizing different subsystems or components of the OBC (communication, memory, core, etc.); however the result of all the foreground tasks should accomplish the same goal.

The foreground application for this design periodically executes the ADCS control loop. The control loop starts by reading the latest sensor values. From these measurements the control algorithms generate commands for the actuators which orientate the satellite accordingly. After the actuator commands are transmitted, the foreground application is put in sleep mode until it is scheduled to start again (more or less every second for this design). This is merely a summary of the control loop which is currently being developed, at time of writing, by another masters student in the ESL.

3.5.2 BACKGROUND TASKS/SERVICES

The background tasks, also known as background services, are additional tasks responsible for housekeeping, emergency responses and interrupts. These services do not directly contribute to the goal of the foreground application, but are nonetheless essential for the operation of the OBC.

The background services for this design gain access to the by means of an interrupt. The background service that generated the interrupt is then processed by running its *Interrupt Service Routine* (ISR). Each ISR was developed to execute as fast as. This helps the foreground application (main ADCS loop) to execute deterministically, i.e. no long unexpected delays due to interrupts. Short ISRs will also prevent the likelihood of data loss during communication since long delays may cause the IO interface to not be serviced before the next data packet arrives.

The background services for this design are as follows:

- **RTC Counter**

  The RTC counter generates an interrupt every second. The ISR of the RTC increments the UTC timestamp of the system, which indicates when some tasks, such as the control loop, are scheduled to start.

- **I2C Communication**

  The I2C communication generates an interrupt whenever it receives a data-packet. If the transmission is not dealt with immediately, the data could be dropped. This is undesirable since a telecommand from the main OBC could contain settings (orientation, schedules, etc.) that should be updated on the ADCS OBC as soon as possible.

- **SRAM SEU Detection**

  The SRAM SEU detection generates an interrupt when it receives a signal from the FPGA indicating an error has been detected or corrected. In case of a correction, the ISR logs the error and schedules a memory wash, described in Section 3.3.4. If multiple uncorrectable errors are detected within a word, the ISR logs the error and performs a variable integrity check, described in Section 3.3.4. If a critical variable is corrupt and cannot be corrected, the MCU is reset.

- **SRAM SEL Detection**

  The SRAM SEL detection generates an interrupt when the current from the SRAM power supply exceeds a certain threshold (latchup) value. The ISR immediately power cycles the SRAM module to attempt to fix the SEL. If the SEL persists the OBC will switch to the backup SRAM module for normal operation.

- **Telemetry Logging**

  The telemetry logger is a background service that periodically logs all OBC telemetry (voltages, currents, temperature, etc.) and ADCS telemetry (vectors and parameters). This information can be requested by the main OBC to be downloaded to the ground station for further inspection.

The above mentioned OS structure should more than suffice for the purpose of an ADCS OBC. This will be illustrated later during *Hardware In the Loop* (HIL) tests in section 4.2. As more complex control algorithms, sensors and actuators are implemented for the ADCS CubeSat unit, the foreground application, background services and protocol will expand, but not the manner in which they are managed and executed. However, in order for the ADCS OBC to stand in as a backup main OBC, a more robust OS structure has to be used.

4 TESTS AND MEASUREMENTS

It is important to thoroughly test all subsystems of an embedded design, even more so in the case of satellite electronics. This chapter will discuss tests that were designed for the ADCS OBC and show the results these tests obtained. Important measurements will then be shown indicating the performance and efficiency of the OBC during operation.

During the software development phase of the ADCS OBC many small driver tests were written to test out all the components on the OBC. The following tests were specifically developed to highlight the operation of the ADCS OBC as shown in Figure 3.14 in Section 3.5 on where the operating system is discussed. Short tests were developed for the different background services and a larger *Hardware In the Loop* (HIL) test was developed for the foreground application (control loop).

4.1 BACKGROUND SERVICES

4.1.1 RTC

The RTC was configured to generate an interrupt every second. The ISR of the RTC increments the UNIX time counter which is used to indicate the time of execution for certain scheduled tasks. The ISR also toggles the external watchdog to ensure it does not reset the MCU. Figure 4.1 shows the GPIO line used to toggle the external watchdog and the accuracy of the RTC which corresponded to one second when measured with an Oscilloscope.

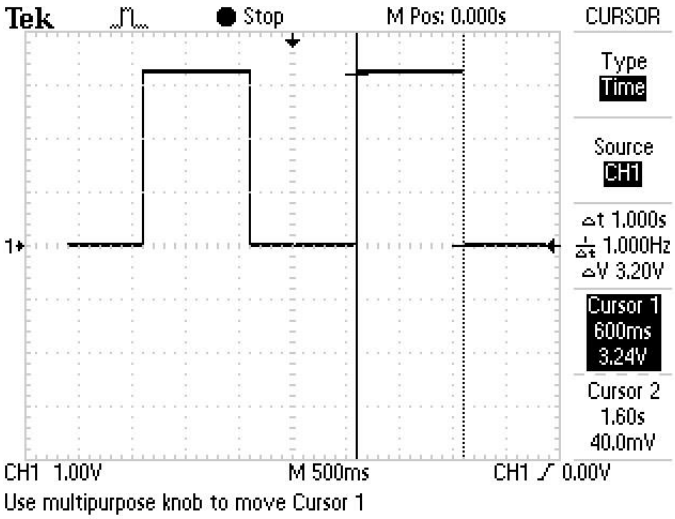


Figure 4.1: External Watchdog Toggle Line.

4.1.2 TELEMETRY LOGGING

The telemetry logging background service is a scheduled task that logs both ADCS telemetry and OBC telemetry which can be requested by the ground station for further inspection. The telemetry data is stored in the microSD card for this design.

A driver test was developed for the microSD card and the resultant text file message is shown in Figure 4.2.

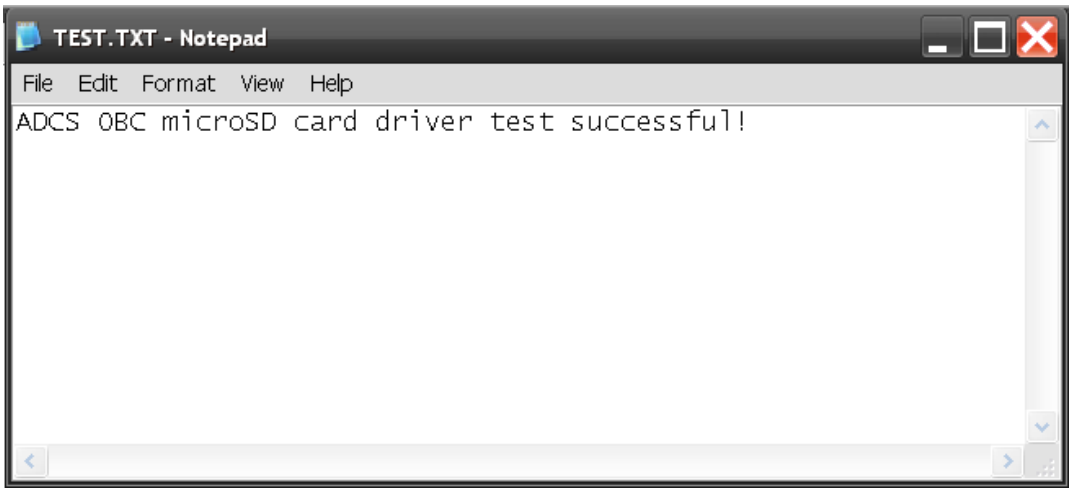


Figure 4.2: Text Retrieved from MicroSD Driver Test File.

To test telemetry logging on the ADCS OBC a test program periodically samples all the ADC channels used by the monitoring subsystem. The OBC telemetry data is then logged to the microSD card as shown in Figure 4.3. The test program stored all the data using string formatting in order to make the test results more readable. However, during satellite operation telemetry data will likely be stored in raw data formats (bytes, integers and double data types) to minimize the amount of data stored and transmitted on the satellite. The test program shows ADC values for current (milli-Amperes), voltage (milli-Volts) and on-chip temperature measurements every second.

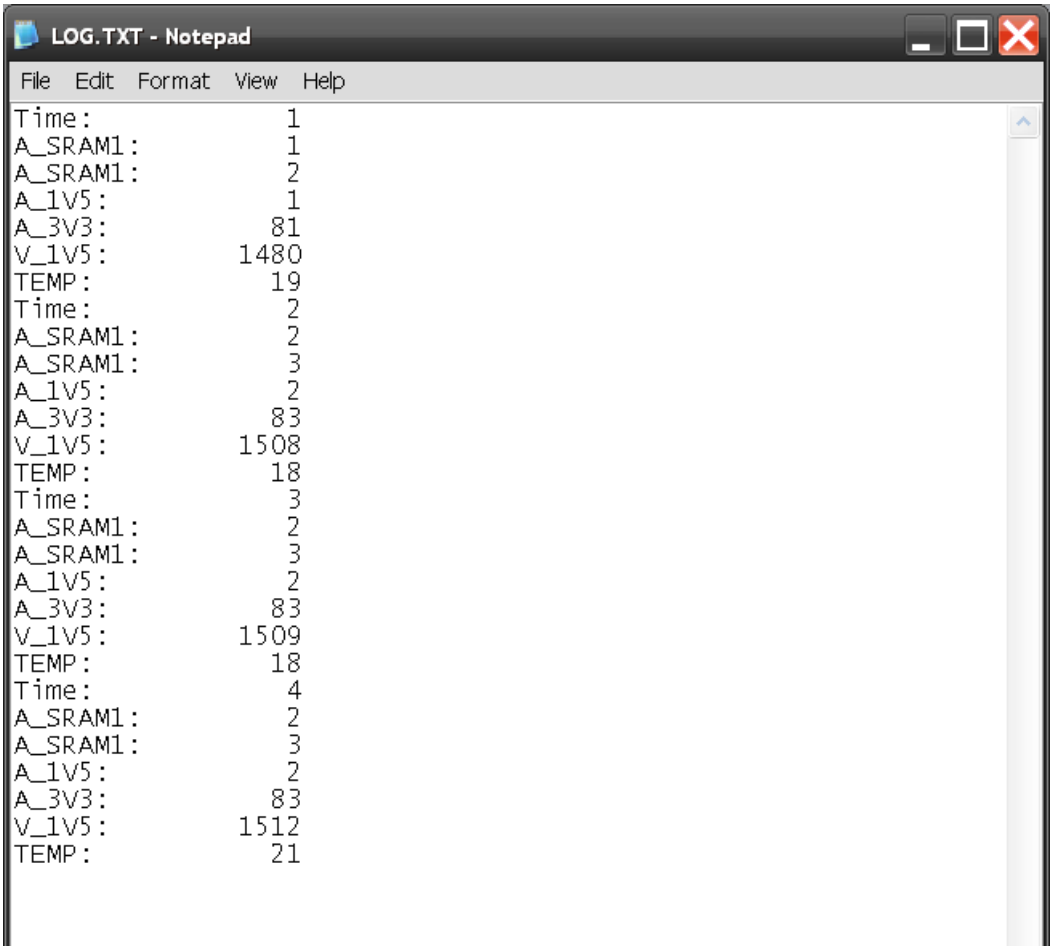


Figure 4.3: Text Retrieved from Telemetry Logging Test File.

4.1.3 SINGLE EVENT UPSETS

Before the SEU detection was tested, the EDAC first had to be verified. Test waveforms were generated by ModelSim (supplied with IDE for FPGA) based on the after-layout (i.e. real-world) operation of the FPGA.

The first line in the waveforms is the *chip-select* (nCS) line for the SRAM module. If the chip select is not asserted, the output/input bus of the FPGA is put in a high impedance state. The following two lines are the *encode* (enc) and *decode* (dec) control lines, which are directly mapped to the read and write control lines of the MCU EBI. When data is written, the encode line is asserted and when data is read, the decode line is asserted. The next line is the 8-bit data bus from the MCU followed by the 16-bit codeword bus from the SRAM modules. Lastly, the error signals generated by the EDAC subsystem are also shown to indicate if an error (SEUs) occurred and if it was correctable.

The following figures (Figure 4.4 – Figure 4.8) show the simulation results of the EDAC for different error conditions:

- **Encoding**

  Figure 4.4 shows the encoding process of the EDAC subsystem. A data word is supplied by the MCU data bus and is then converted to a codeword when the write control line is asserted. The codeword generated might not seem to be in systematic form, but it is, because the data word is split before being encoded and interleaved after being encoded.

- **Decoding –** No Errors

  Figure 4.5 shows the decoding process of the EDAC subsystem with no errors in the codeword. The codeword is the same as the one generated during the encoding process and the error signals indicate no errors.

- **Decoding –** One Error

  Figure 4.6 shows the decoding process of the EDAC subsystem with one error. The error is corrected (same data word as Figure 4.4) by the FPGA, which indicates the error to the MCU through the error signal lines.

- **Decoding –** Two Errors Correctable

  Figure 4.7 shows the decoding process of the EDAC subsystem with two errors. The errors are in separate codewords within the large codeword and both errors can therefore be corrected by the EDAC. The two errors are indicated to the MCU through the error signal lines.

- **Decoding –** Two Errors Uncorrectable

  Figure 4.8 shows the decoding process of the EDAC subsystem with two errors that cannot be corrected. The errors are in the same smaller codeword within the large codeword. The resulting data word is different from the one generated by the encoding process in Figure 4.4. However, the EDAC does detect the errors and generates the error signals to indicate to the MCU that the data retrieved from the SRAM module is corrupt.

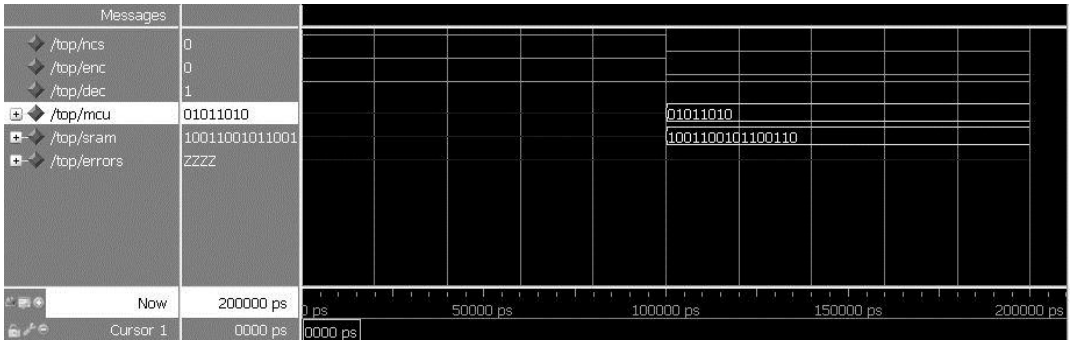


Figure 4.4: Encoding Process of EDAC.

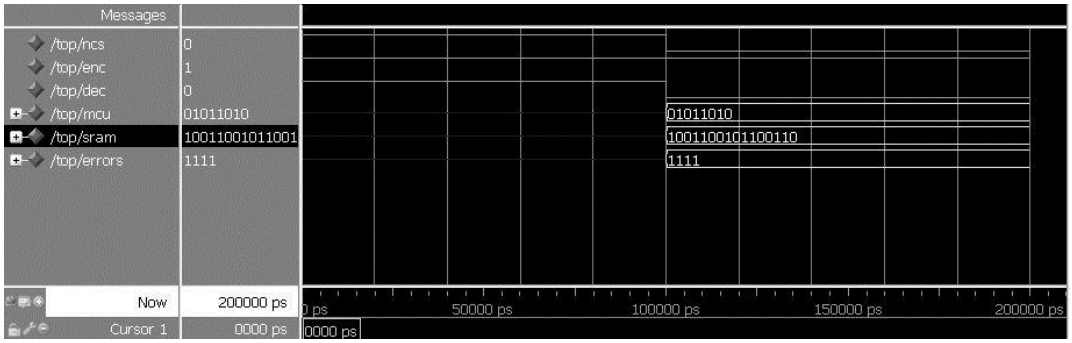


Figure 4.5: Decoding Process of EDAC with No Errors.

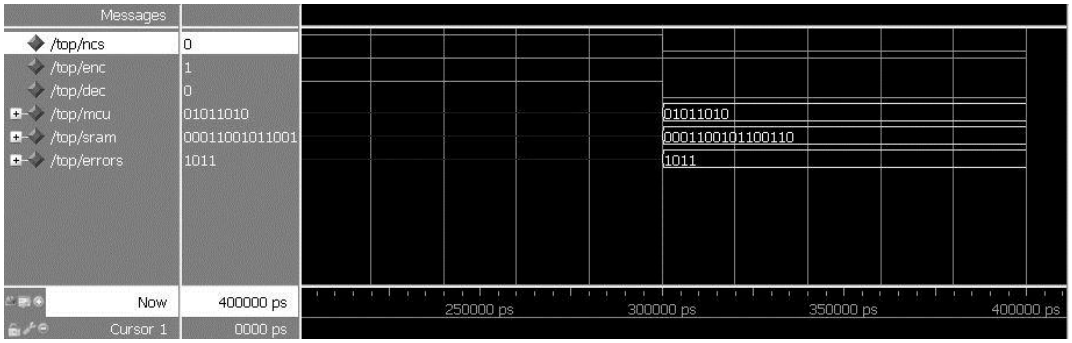


Figure 4.6: Decoding Process of EDAC with One Error.

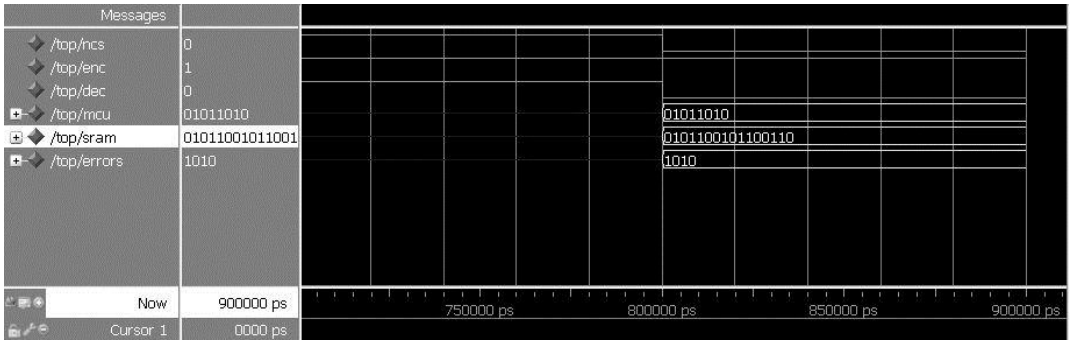


Figure 4.7: Decoding Process of EDAC with Two Errors.

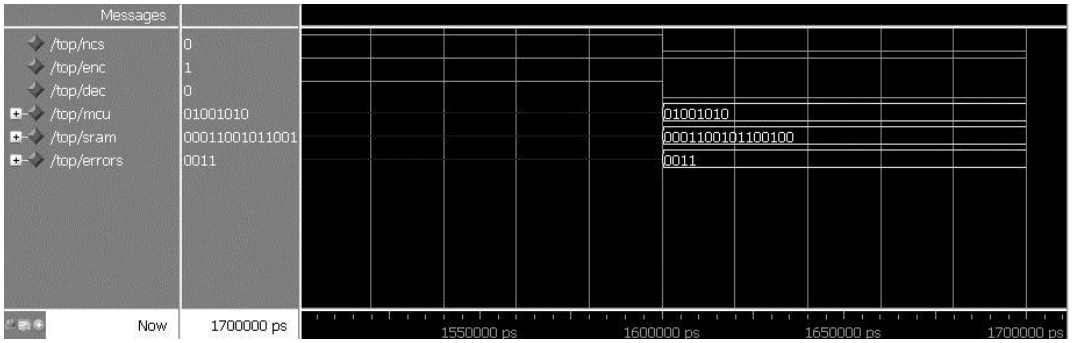


Figure 4.8: Decoding Process of EDAC with Uncorrectable Errors.

To physically test the SEU detection on the OBC a bit flip had to be simulated. The FPGA code was modified to manually flip a specified bit before writing it to the SRAM module. This same codeword was read by the MCU and the resulting error signals from the FPGA caused an interrupt on the MCU. The interrupt service routine can read the values of the error signals and respond by either scheduling a memory wash for a correctable error, or a variable integrity check for uncorrectable errors.

The latency involved with encoding and decoding the data between the MCU and SRAM databus was also sufficiently small enough that it was not necessary to add extra read and write hold cycles to a normal read and write cycles of the EBI. The EDAC implementation can therefore be seen as flow-through.

4.1.4 SINGLE EVENT LATCHUP

When a latchup occurs in an SRAM module, the module draws an excessive amount of current. This current spike can be used to detect the latchup by continually monitoring the SRAM current. The MCU can be configured to generate an interrupt whenever a threshold value is exceeded. The resulting ISR will then isolate the SRAM module from the address and data bus by turning off the disabling IO buffers and disabling the load switch supplying the

SRAM module. Figure 4.9 shows the SRAM supply voltage, from a load switch, toggled by the MCU with a test program.

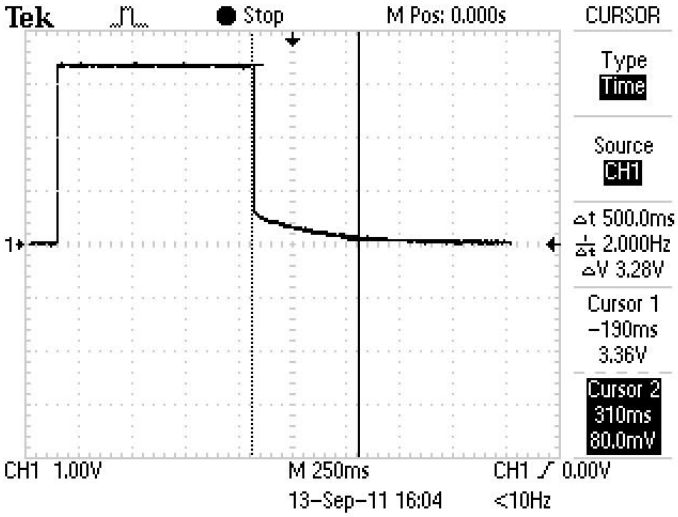


Figure 4.9: SRAM Supply Voltage Toggled by MCU.

4.1.5 COMMUNICATION

During the software development process of the ADCS OBC, the UART was used extensively for debugging purposes. Figure 4.10 shows the OBC telemetry data output to the PC through the UART.

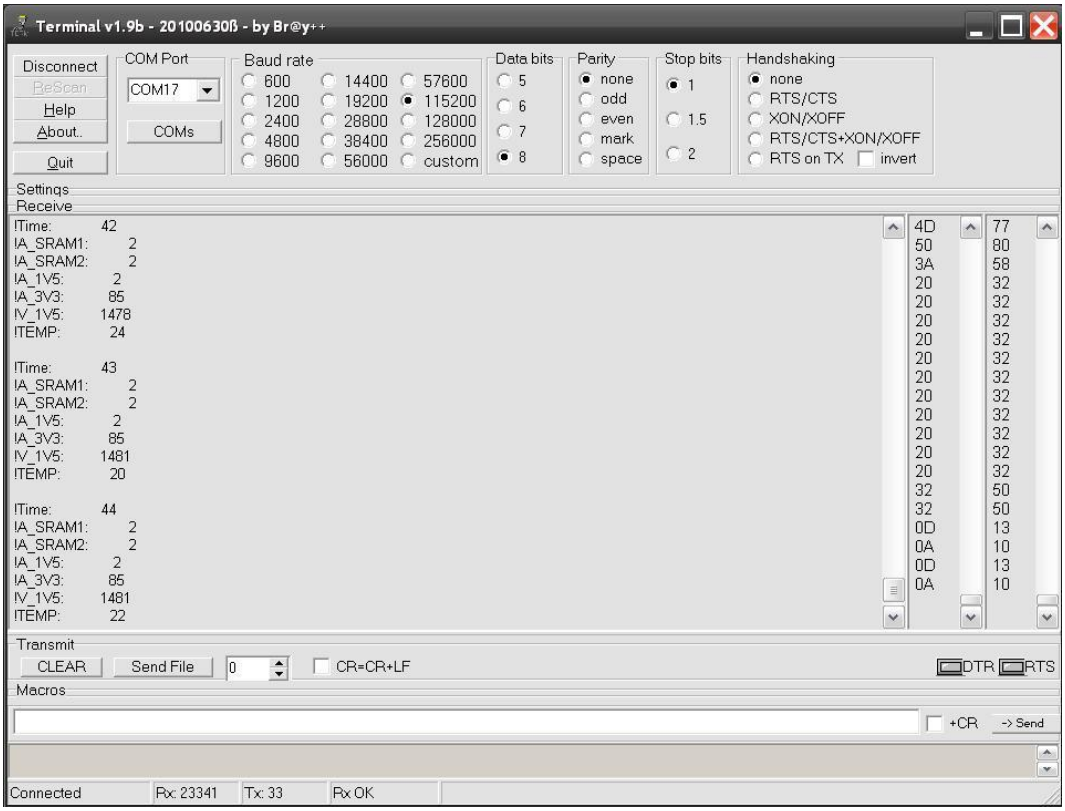


Figure 4.10: OBC Telemetry Data Output to UART.

The UART was used during the HIL tests to communicate with the PC. A small protocol was implemented for the UART to more accurately simulate in system operation. The HIL test will be discussed in more detail in the following section.

4.2 HARDWARE IN THE LOOP

A HIL test was devised to test the ADCS OBC performance under “real-world” conditions, or as close to it as possible. A HIL test simulates closed loop system operation and should receive realistic data inputs. The HIL test is an ideal method to test the performance and accuracy of the foreground application, i.e. its ability to perform the attitude determination and control for a CubeSat.

4.2.1 SETUP

The setup and flow of the HIL test is shown in Figure 4.11. The OBC was interfaced with a PC, which simulated all other needed ADCS subsystems, and a protocol was used to transmit and receive data between them. The OBC will request sensor data (generated by the PC) through the communication subsystem. The ADCS algorithms will then compute the needed actuator values. The ADCS telemetry (torque times, estimated rates etc.) will be transmitted to the PC which will be compared to similar values generated by the PC simulation program.

The accuracy of the ADCS OBC will be tested during the HIL test, but the computation time is also of importance. The faster the control loop can be executed, the more time can be spent in a lower energy mode and/or the more complex algorithms can be implemented. The control loop for the HIL test consists of the following ADCS algorithms:

- **SGP4 –** A *Simplified General Perturbation no.4* (SGP4) model calculates the satellite position vector using *Two-Line-Elements* (TLE) of the satellite orbit.
- **IGRF** – An *International Geomagnetic Reference Field* (IGRF) model calculates the magnetic field vector at the satellite's position.
- **Kalman Filter –** A Kalman filter is used to calculate the estimated angular body rates of the satellite.
- **BDOT** – A BDOT controller is used to calculate the required actuator torque times needed to detumble the satellite.

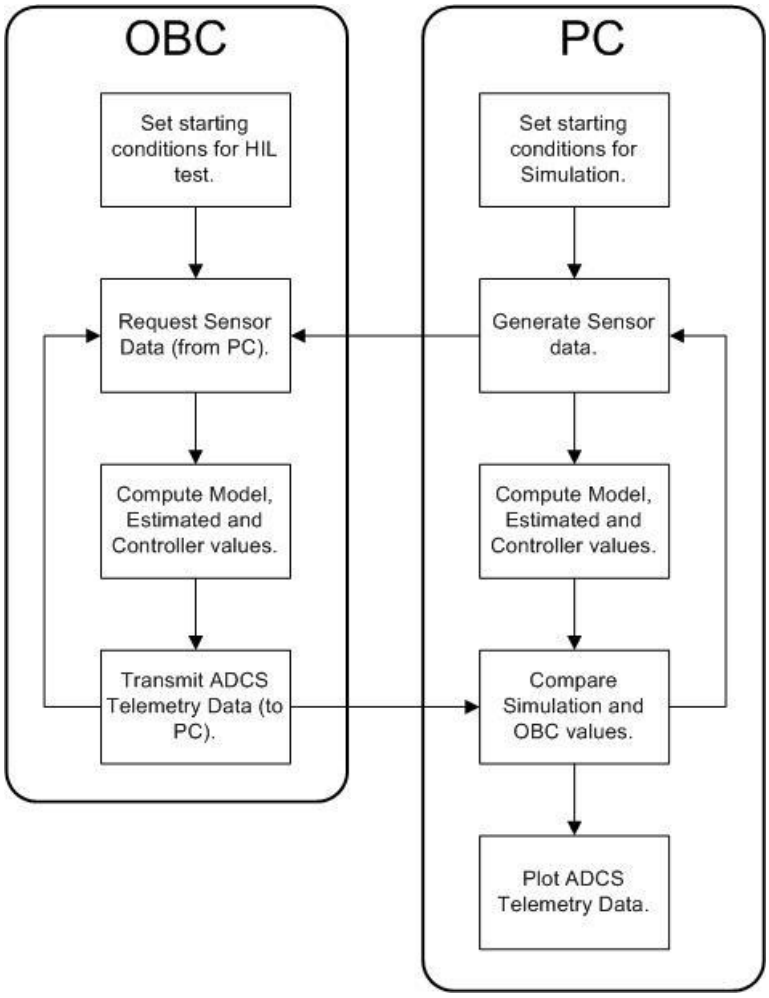


Figure 4.11: Hardware In the Loop Test Process.

4.2.2 PC COMMUNICATION PROTOCOL

The protocol used for the HIL test, shown in Table 4.1 and Table 4.2, consists of *Identification* (ID) and data packets. This is a very simple and specific protocol that was setup and used for the purpose of the HIL tests. It is loosely based on the protocol used on board SumbandillaSat for its ADCS OBC.

Table 4.1: Transmission Protocol from PC to OBC.

ID	Data length		Data content	Unit
$	3 × (short int)	6 bytes	Magnetometer readings	uTesla
&	1 × (char)	1 bytes	Controller mode	0-256
#	1 × (short int)	2 bytes	Reference Y-spin rate	m-deg/sec
%	1 × (char)	1 bytes	Controller sample time	0-256 (sec)
@	only identifier	0 bytes	Acknowledgement	

Table 4.2: Transmission Protocol From OBC to PC

ID	Data length		Data content	Unit
R	only identifier	0 bytes	Request sensor measurements	
T	3 × (short int)	6 bytes	Torquer rod on times	sec
W	3 × (short int)	3 bytes	Estimated body angular rates	m-deg/sec
N	3 × (double)	24 bytes	Magnetic torque	Nm

4.2.3 RESULTS

During the HIL test, the average execution time for a control loop iteration was calculated as 22 ms. For the CubeSat ADCS being developed at the ESL, a control loop only needs to be executed every one second. Even though the control loop for the HIL test was a simple detumbling procedure for a CubeSat, it still means the ADCS OBC requires very little of its allocated time (< 3% for the HIL test) to do the needed ADCS calculations. This will result in the OBC spending more of its time (> 97% for the HIL test) in a lower energy state, such as stop (EM1) or sleep (EM2) mode, which require less power.

In terms of accuracy the ADCS OBC also calculated the same values as simulated by the PC, which was expected since they both are running the same algorithms. In both Figure 4.12 (estimated body rates produced by the Kalman filter) and Figure 4.13 (actuator control values produced by the BDOT controller) the OBC-generated values (coloured dots) closely follow the PC-simulated-values (solid black lines).

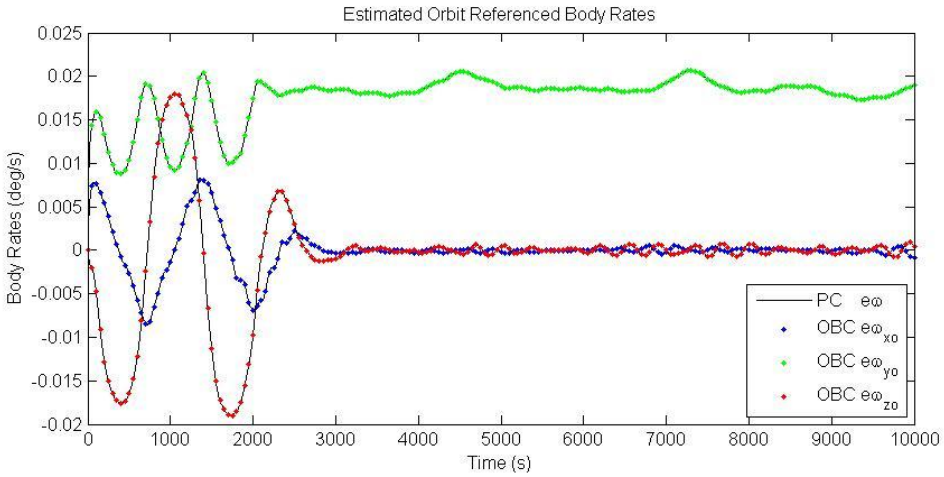


Figure 4.12: OBC and PC Estimated Body Rates for HIL Test.

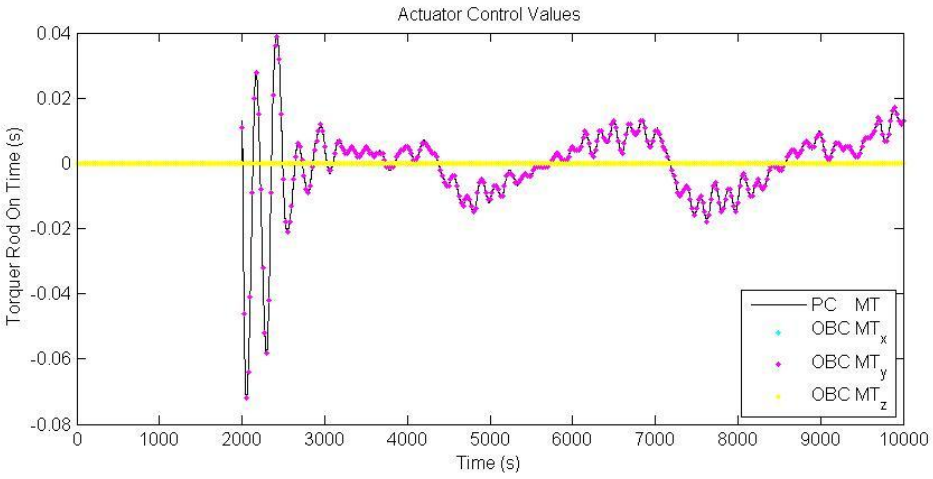


Figure 4.13: OBC and PC Actuator Control Values for HIL Test.

4.3 POWER CONSUMPTION

It is important for any OBC design to determine its power consumption under various operating conditions. This is even more important for an OBC in a CubeSat due to its limited power budget.

The power consumption of the ADCS OBC will be determined by measuring the current drawn from the power supply. Figure 4.14 shows the test setup that was used. Between the ADCS OBC and its power supply a 1.8Ω resistor is added in series. By measuring the voltage drop over this resistor, the current flowing through it can be calculated.

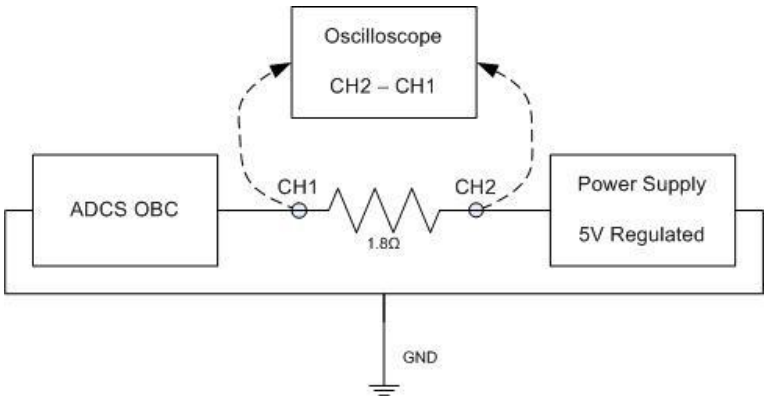


Figure 4.14: Test Setup for Power Consumption Measurements.

Different tests were written for the OBC to perform while its power consumption was measured. Some of the tests focused on the MCU under different conditions and some were focussed on the various peripherals. The results are shown in Table 4.3. The OBC draws a minimum power of 130mW during sleep mode and peaks at 435mW while accessing the

microSD card. The ADCS OBC power consumption should average somewhere between 170mW (EM0 – Run Mode) and 130mW (EM2 – Sleep Mode) most of the time since it will either be executing the control loop or sleeping.

Table 4.3: Power Consumption Test Results.

Test Type	**Voltage Drop mV**	**Current mA**	**Power mW**
Fibonacci Algorithm	61	34	170
while(1)	57	32	160
EM1	48	27	135
EM2	46	26	130
MSD (peak/avg)	156/110	87/62	435/310
UART	82	46	230
ADC	54	30	150
SRAM	55	31	155

5 CONCLUSIONS AND RECOMMENDATIONS

5.1 CONCLUSIONS

The tests in the previous chapter were designed to analyse and prove the capabilities of the ADCS OBC which were designed and developed during Chapters 3 and 4. It was important for these tests to reflect the design requirements of performance, efficiency and reliability in order to confirm that they have been met. The test results for each requirement can be summarized as follows:

- **Performance**

  The HIL test confirms the performance of the ADCS OBC by performing a typical ADCS control loop of updating sensor data, generating model and estimated values and computing actuator output. The requirement for a typical CubeSat ADCS (being developed in the ESL) is to execute a control loop every second. From the HIL test data, the ADCS OBC easily achieves this goal by executing the control loop in 22 ms.

  This short execution time has two implications for efficiency. Firstly, the MCU can sleep for the majority of the control loop cycle, which will reduce the average efficiency of the OBC. Secondly, the MCU can run at a lower frequency to reduce its peak power consumption, but at the cost of a longer execution time.

- **Efficiency**

  The current measurements during the different test programs show the efficiency of the ADCS OBC in terms of peak power consumption and average power consumption. As explained above, the MCU can be downclocked to decrease either the peak power consumption or the average power consumption. However, the ADCS OBC should still achieves a peak power consumption of less than half a Watt (435 for SD Card usage) and an average power consumption of less than two hundred milli-Watt, which is well within the expected power budget.

- **Reliability**

  The majority of the background services (watchdog toggling, SEU and SEL detection and telemetry logging) are implemented mainly for reliability purposes. Watchdogs prevent the MCU from entering an undesired state, SEU and SEL detection methods protect the OBC against SRAM failures due to radiation and OBC telemetry allows the OBC to be

monitored by users on the ground. Tests were designed to illustrate the operation of these background services which were developed to add an extra layer of reliability to the ADCS OBC.

These tests and measurements indicate that the ADCS OBC in this design achieved the requirements defined before the start of the project. However, some small alterations and improvements will be needed before the ADCS OBC in this design can be used within a CubeSat system.

5.2 RECOMMENDATIONS

This section will discuss recommendations regarding alterations and improvements that should be made to the ADCS OBC before it is implemented into a CubeSat.

5.2.1 LAYOUT

It is important to take into consideration the PCB layout of the ADCS OBC due to space constraints imposed by the standard size of a CubeSat structure. A standard PCB layout is shown in Appendix C.1.2. The PCB of the ADCS OBC was not designed according to this standard since it was considered a prototype. The prototype OBC has an external power connecter, extra pin-outs for unused MCUs, FPGA pins (used for debugging purposes) and larger dimensions (designed on a two layer PCB for reduced cost). If the ADCS OBC layout is done on a four-layer PCB without the debugging components, it should be able to conform to the standard layout proposed in Appendix C.1.2.

5.2.2 MICROCONTROLLER

During the hardware design of the ADCS OBC, the only available processor from Energy Micro was the EFM32Gecko (EFM32G), which was used in this design. In late 2010/early 2011 Energy Micro released the EFM32GiantGecko (EFM32GG [36]). This MCU is also based on the Cortex-M3 architecture, like the EFM32G, but has the following new features that would improve the ADCS OBC design.

- **Improved External Bus Interface**

  The external bus interface of the EFM32GG has the ability to translate read and write commands, depending on the bus width of the external memory. This means that the MCU autonomously converts the write command of 32-bit data into four write commands for an 8-bit external memory device. The major advantage this has for the ADCS OBC is that the MCU is now able to execute code directly from external memory. Currently the ADCS OBC needs to copy an operating program from external flash into the MCU flash

before booting from this program, which increases the likelihood of an SEU. With the EBI of the EFM32GG, the bootloader can boot directly from external flash.

- **Two I2C controllers**

  The EFM32GG has two I2C controllers compared to the EFM32G which only has one. Currently the ADCS OBC will only be able to implement the multi-master and time-shared I2C bus. The EFM32GG will however be able to implement the separate bus as well, by having one I2C controller act as a slave on the main I2C bus and the other controller act as a master on the bus connecting all the ADCS-related subsystems (sensors and actuators).

The EFM32GG is pin compatible with the EFM32G and should therefore require only minimal hardware design and layout changes. Due to the CMSIS the EFM32 drivers are built upon, very few changes to the drivers and operating system should occur. Not only should the EFM32GG be easy to implement, but its power consumption is roughly the same as the EFM32G, as shown in Table 5.1.

Table 5.1: EFM32 Gecko and Giant Gecko Comparison.

Microcontroller	**EM0**	**EM1**	**EM2**
EFM32 Gecko	180 uA/MHz	45 uA/MHz	0.9 uA
EFM32 Giant Gecko	200 uA/MHz	50 uA/MHz	1.2 uA

5.2.3 EEPROM

Electronic Erasable Programmable Read Only Memory (EEPROM) is a non-volatile memory that is inherently resistant to radiation effects such as SEUs and SELs. EEPROM is usually used to store important program code due to its radiation resistance. In this design, the safe mode program (described in Section 3.4.1) is stored in flash memory, which is still susceptible to SEUs. A more reliable design would be to store the safe mode program separately in an EEPROM module, such as the ATMEL EEPROM (AT28C64B [37]), which is virtually immune to SEUs. This will ensure the ADCS will always have a working backup program to fall back on. This EEPROM module can be accessed via the EBI of the MCU the same way as with flash and should be easily replaceable with one of the flash modules in the ADCS OBC design with only minimal modifications. An EEPROM module either has a longer read cycle or uses more current compared to a flash module (see Table 5.2), but the added reliability should be worth the trade-off.

Table 5.2: Flash and EEPROM Comparison.

Type		Size	Current	Read Cycle
Flash	S29AL008D [38]	1 MB	16 mA	70 ns
EEPROM	AT28BV25 [39]	256 kB	15 mA	200 ns
	AT28HC256 [40]	256 kB	80 mA	70 ns

5.2.4 PROTOCOL

A protocol is a set of rules which defines how data is transmitted between subsystems according to [10]. The rules define how information is encoded into transmission data. If the receiver uses the same protocol as the transmitter, the transmission data can be decoded successfully into meaningful information. It is important to have only one protocol on a bus to avoid any confusion when transmitting data between subsystems. Because the protocol is a standard set of rules, it is possible to develop a subsystem independently by only ensuring it adheres to the protocol used on the system bus.

Even though a simple protocol was defined for the HIL test in the previous chapter, it is highly recommended that a fully-fledged protocol should be developed for the ADCS OBC and its subsystem (sensors and actuators) in the ADCS unit. For a typical protocol a transmission consists of the following sections:

- **Header**

  The header section of a protocol contains fields which define how the transmission will occur. Examples of typical fields used in header sections are an address field to identify the sender/receiver, a size field to indicate the size of the data section, an indicator of the data format and an acknowledgement from the receiver. For the protocol used in the HIL test, the header section consists of one field, namely the ID byte.

- **Data**

  The data section of a protocol contains the actual data being transmitted. The data can either be sent all at once or sent as packets of smaller data, depending on the size of the data, bus and sender/receiver buffer.

- **Error Control**

  The error control section of a protocol usually contains a small amount of data which allows the receiver to check if the main data became corrupted due to transmission errors. A popular approach is to add a *Cyclic Redundancy Check* (CRC) at the end of a

transmission. The receiver can use the CRC to check for any corrupt data. If an error is detected, a retransmission can be initiated.

For this design a protocol was not created as it was not necessary for the design and development of an ADCS OBC. However, for the ADCS OBC to be integrated into the ADCS unit, a protocol should be defined and used for communication between the ADCS subsystems.

5.2.5 REAL-TIME OPERATING SYSTEM

A *Real-Time Operating System* (RTOS), as described by [41], is developed to be more adept at managing and executing multiple tasks. It uses a more advanced scheduling algorithm to ensure each task receives an equal or weighted amount of processing time from the MCU core. This scheduling produces a small overhead, but ensures all tasks can be processed closer to "real-time".

An RTOS is much more suited for the main OBC than the operating system used for the ADCS OBC. Instead of having one goal, like the satellite's ADCS does, the main OBC should manage the operation of many subsystems (payload, power, communication, ADCS, etc.) with almost equal importance. All these subsystems are interdependent and therefore if one is neglected, it adversely affects the entire satellite.

An RTOS was not used for the ADCS OBC because it is not crucial for its operation, but implementing the ADCS application on an RTOS would allow the OBC to adapt more easily to the role of main OBC in case of emergency. Energy Micro, the manufacturer of the MCU used in this design, has a source code example which shows an RTOS port (uC/OS-II from Micrium) running on the EFM32G [31]. This example can be used as a basis to create a RTOS tailored for the ADCS OBC.

6 SUMMARY

The goal of this project was to design and develop an ADCS OBC for a CubeSat. The process followed to achieve this will be summarized in this section.

The first step was to clearly define the design requirements for the OBC. The project goal can be broken into three parts and each part has an associated requirement attached to it. Firstly, the OBC should be able to perform complex ADCS-related tasks. This translated to **performance** as a requirement. Secondly, the OBC will be implemented on a CubeSat which has very limited resources. This translated to **efficiency** as a requirement. Lastly, the OBC will have to operate in space, which is a very harsh environment. This translated to **reliability** as a requirement. These requirements were considered throughout the hardware design as well as the software development of the ADCS OBC.

The core of the OBC hardware design is the MCU and care was taken in choosing the most appropriate MCU for the ADCS OBC. The EFM32Gecko from Energy Micro, based on the ARM Cortex-M3 architecture, was chosen due to its high performance and low power consumption. It also offered various on-chip peripherals which were used in the OBC design. A separate power system was designed to regulate the supply voltage and current to all the components. An external memory subsystem was designed which consists of flash, SRAM and a microSD card which stores code, data and telemetry. The OBC has a subsystem which monitors the current, voltage and temperature of the board for telemetry purposes, but it can also detect abnormalities and react accordingly. The OBC also includes various interfaces for communicating with external and internal subsystems (I2C, UART, EBI and SPI).

The software development commenced with creating and/or modifynig all the drivers needed to enable and access the different functions of the MCU and components on the ADCS OBC. Cortex-M3 based MCUs, such as the EFM32G, uses the CMSIS, which improves the turnaround time for developing drivers. The flow-through EDAC uses the Reed-Muller(1,3) linear block codes which were developed in VHDL and implemented on the FPGA. The bootloader algorithm was developed to ensure that the ADCS OBC will not be able to reboot into a faulty operating program. The operating system structure consists of a foreground application, the ADCS control loop, and background tasks which are responsible for the clock, communication, telemetry and error detection and correction.

Tests were developed to confirm the operation of the ADCS OBC. Individual tests were written for background tasks and a HIL test was designed to simulate in-system operation for the ADCS OBC foreground application. The results of the test were analysed in terms of the

design requirements defined at the beginning of the project. The conclusion reached after testing was that the ADCS OBC prototype justified the design by meeting the design requirements of performance, efficiency and reliability and that, with a few minor alterations and improvements, the ADCS OBC could successfully be implemented into a CubeSat.

A HARDWARE DESIGN DETAILS

The hardware design of the different components such as smoothing capacitors, pull up- and pull down resistors etc. were kept as close to the supplied hardware design guidelines supplied by each component's manufacturer. This ensured that all components functioned when the ADCS OBC was built. In some cases, the design choice could be made which altered the operation of certain components. These choices will be discussed in the following section.

A.1 HARDWARE DESIGN GUIDELINES

The following hardware design guidelines where used in the design of the ADCS OBC.

- **MCU** – "Application Note: Hardware Design Considerations".[42]
- **SRAM** – "SRAM System Design Guidelines". [43]
- **Voltage Regulator** – Datasheets [28], pages 11-13, and [27], pages 15-17.

A.2 CURRENT SENSOR DESIGN

In the current sensor design, as shown in Figure 2.11, the value of resistors R_S and R_L are designed to achieve an accurate output voltage for the range that includes normal operating conditions and a latchup within a SRAM module.

The output voltage is determined by the following equation[44]:

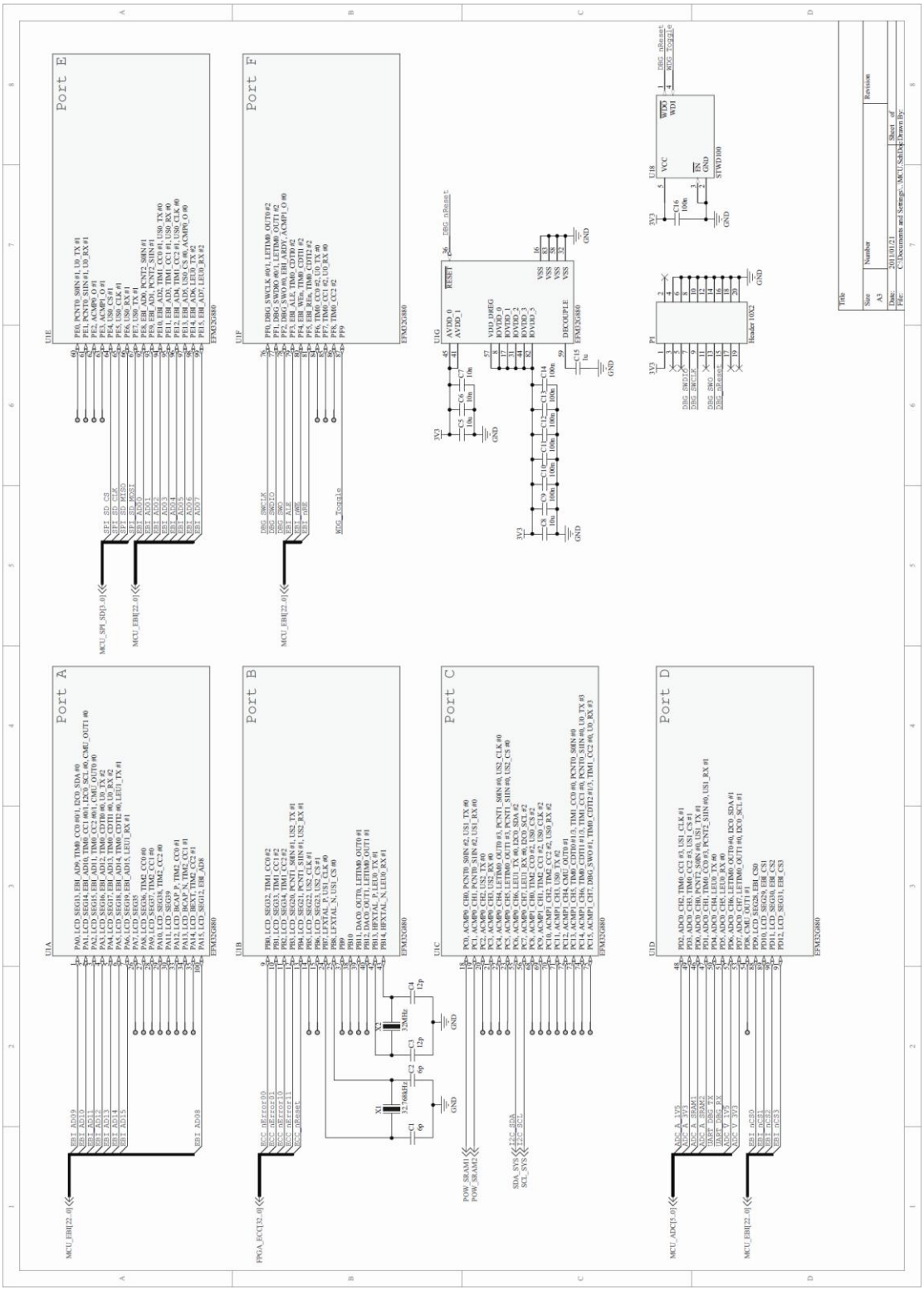
$$V_{OUT} = (I_S)(R_S)(1000\mu A/V)(R_L) \tag{A.1}$$

I_S would vary between 25mA at normal operating conditions and 200mA during a latchup from an SRAM module. For the INA139 current sensor used in this design the input voltage V_S is accurate if below 0.5 V. The resistor value for R_S was therefore chosen as 1 Ω which ensures an input voltage of less than 0.5 V under latchup conditions and does not allow a too large voltage drop on the power supply during normal operating conditions.

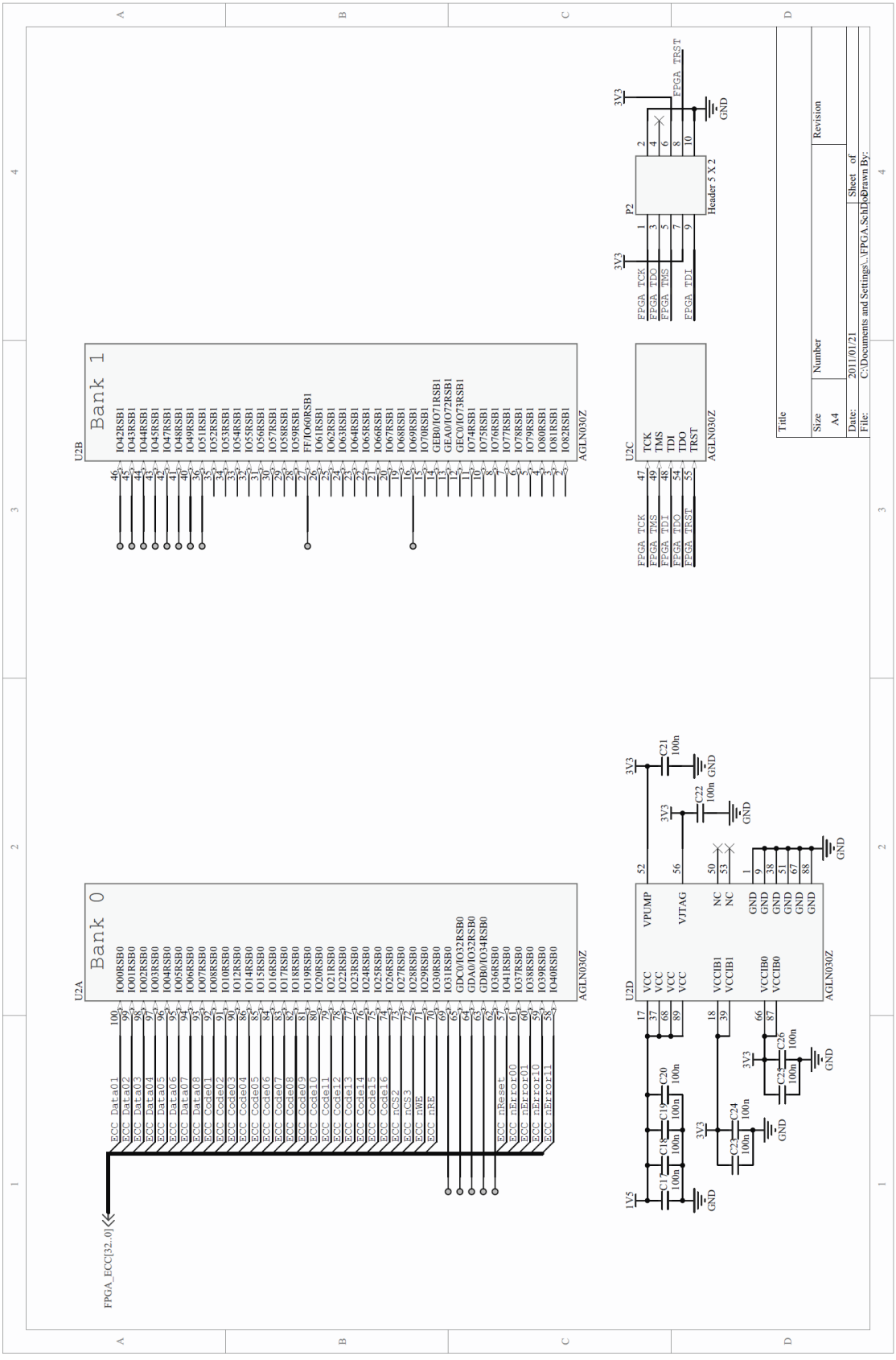
The output voltage of the INA139 is sampled by the ADC of the MCU. Therefore the output voltage range should be within the sample range of the ADC under normal operating conditions and during latchup. By choosing the value of R_L as 10 kΩ, equation (A.**1)** gives an output voltage of 250 mV for normal operating conditions (I_S = 25 mA) and 2 V during a latchup (I_S = 200 mA). For a 12-bit sample from the MCU ADC, the resolution is less than 1 mV which will be accurate enough for telemetry purposes.

B SCHEMATICS

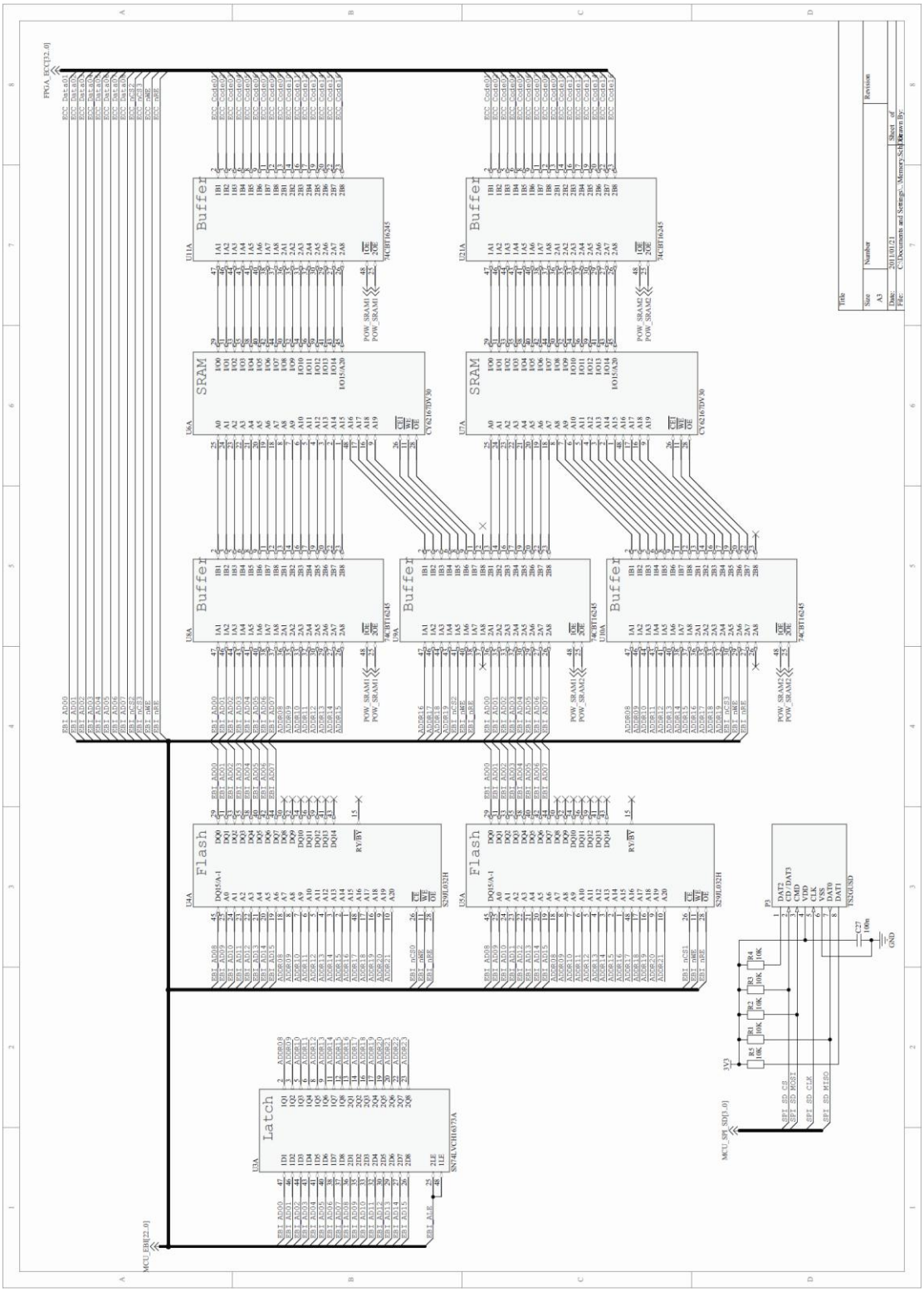
B.1 MCU



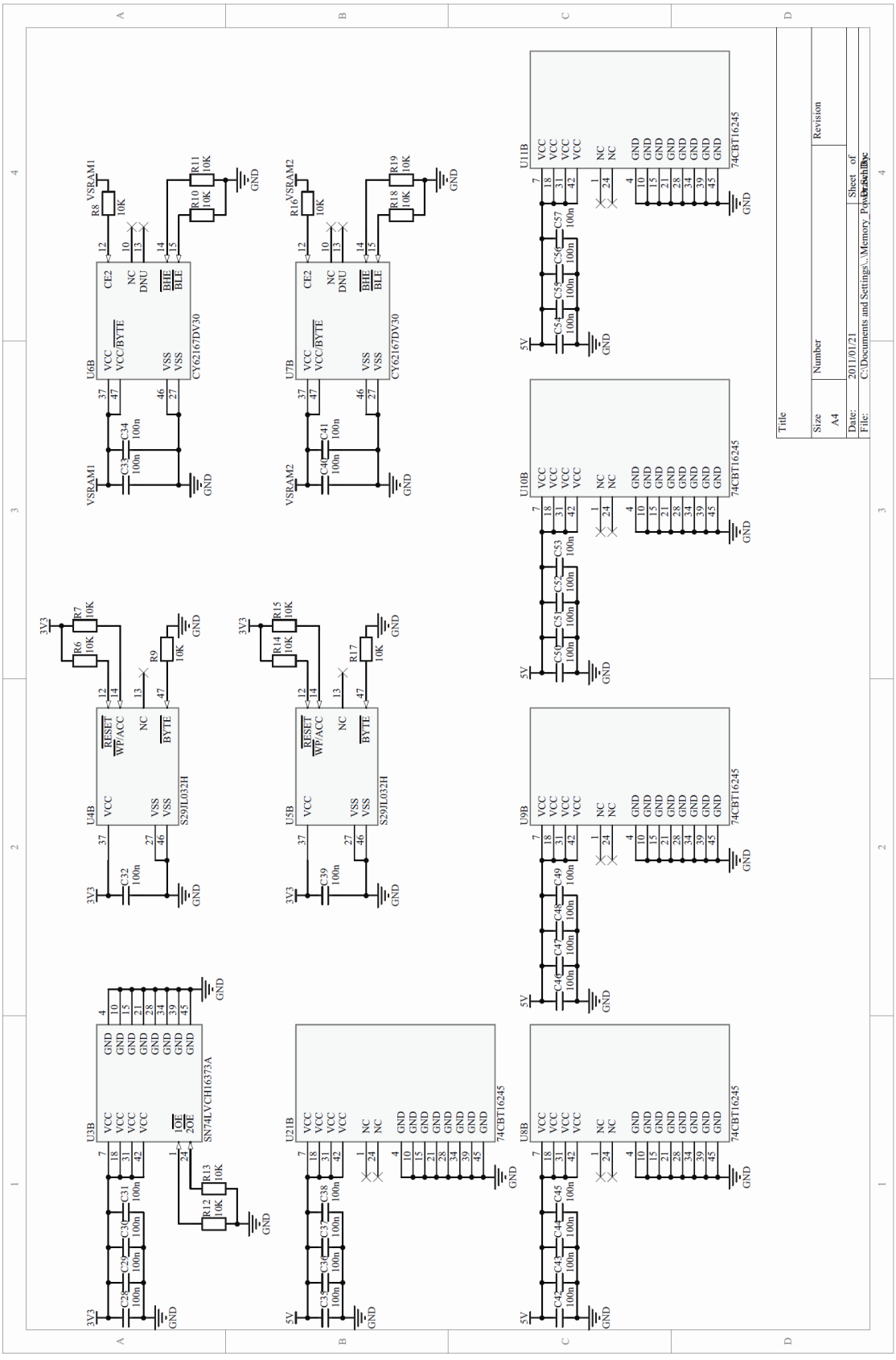
B.2 FPGA



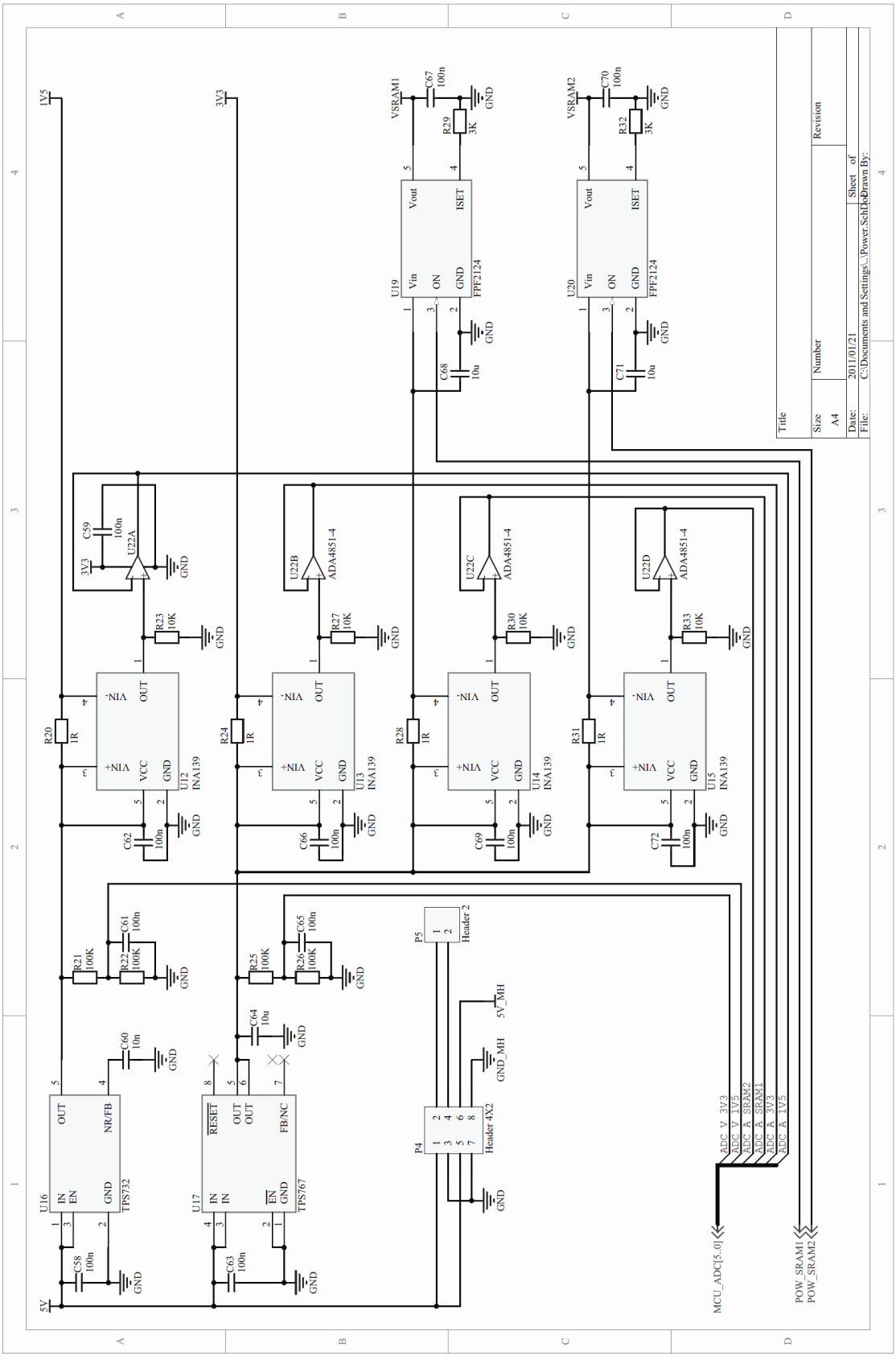
B.3 MEMORY BUS



B.4 MEMORY POWER



B.5 POWER SUPPLY



C PCB LAYOUT

C.1 CUBESAT LAYOUT STANDARD

When designing the layout for a CubeSat expansion board it is necessary to try and stay within the predefined specifications of the CubeSat standard, both electrically and mechanically, to ensure compatibility with other CubeSat subsystems and the CubeSat standard itself.

C.1.1 ELECTRICAL LAYOUT STANDARD (HEADER PIN ALLOCATION)

The CubeSat uses a stack through 104-pin main header that links most of the CubeSat subsystems. These shared pins allow CubeSat subsystems to easily integrate with each other with regards to power lines, communication lines and signals. At the moment no standard exists that clearly defines the functions of all the pins on the main header. However, some general guidelines do exist which are shown in Figure C.1. It is important to follow these guidelines when designing the ADCS OBC as it allows compatibility with most other CubeSat subsystems.

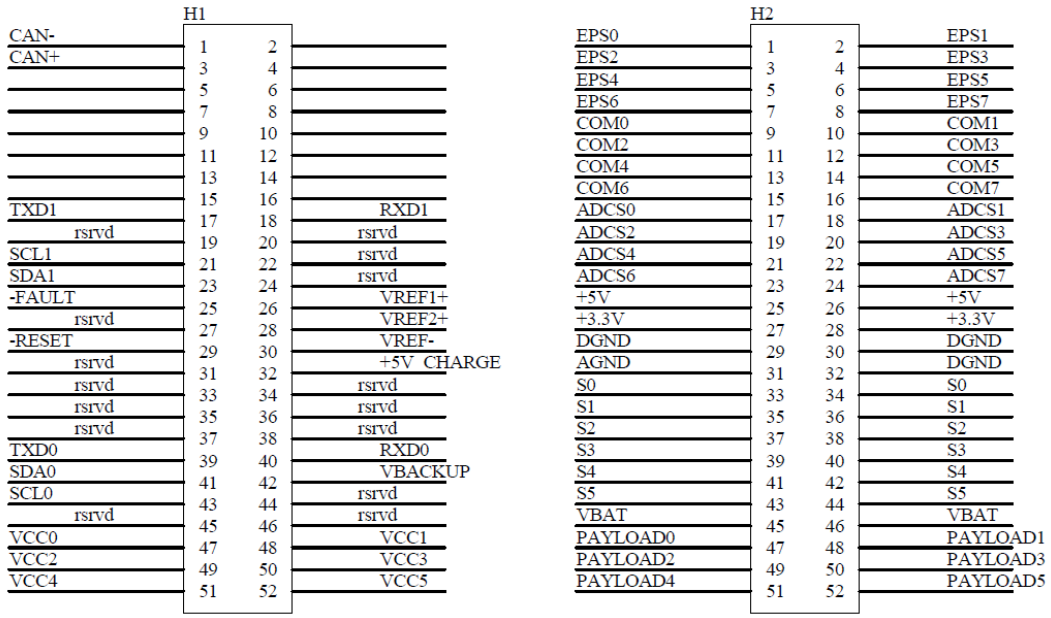


Figure C.1: CubeSat Proposed Electrical Layout Standard (PC/104 Based). [23]

C.1.2 MECHANICAL LAYOUT STANDARD (PCB DESIGN)

The CubeSat expansion boards are based on the mechanical layout of the PC/104 standard shown in Figure C.2. By adhering to this mechanical design when designing an expansion

board, it will easily integrate with the rest of the subsystem in the CubeSat. The greatest advantage of the mechanical standard of the CubeSat is that it makes use of the P-POD launcher which allows for inexpensive piggyback launches.

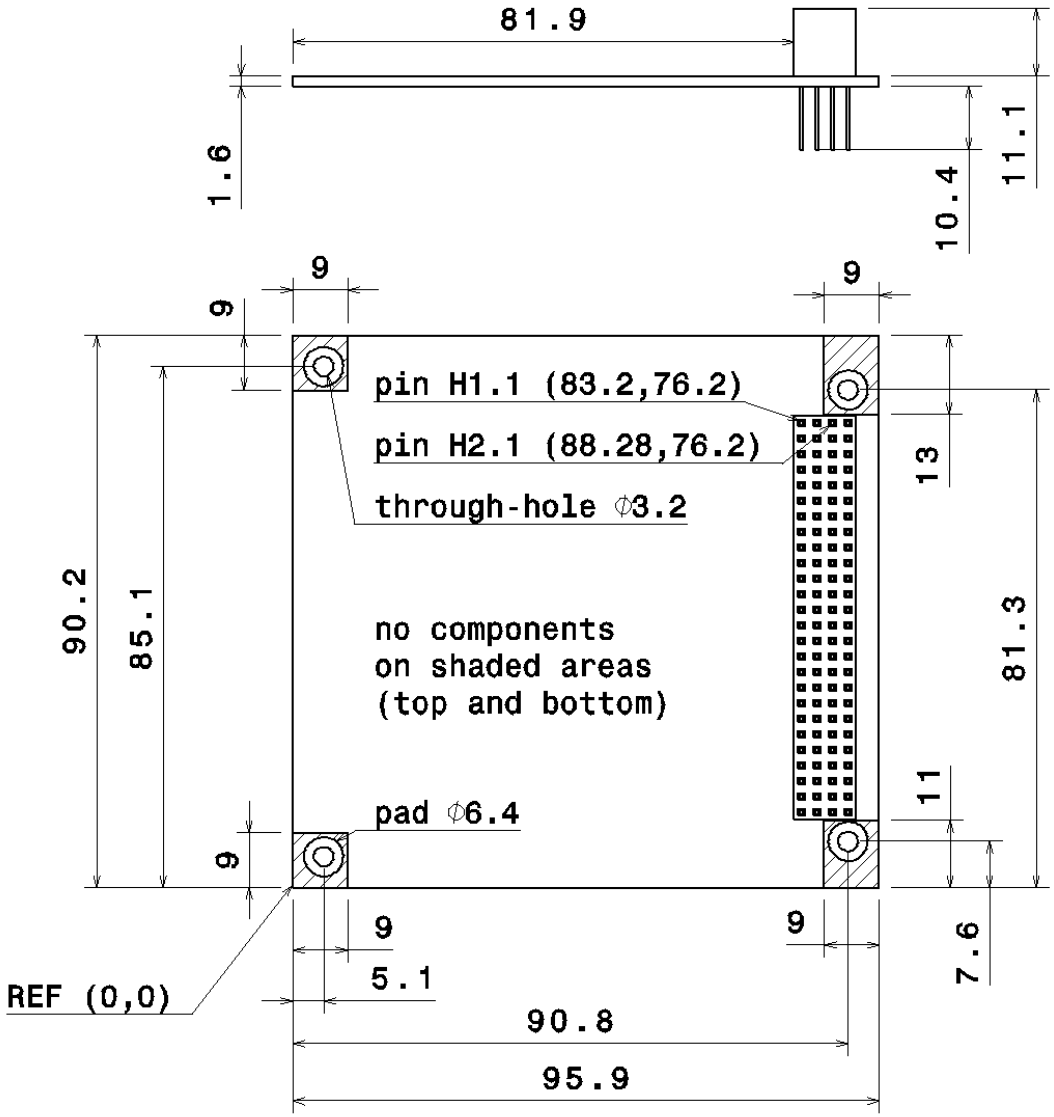
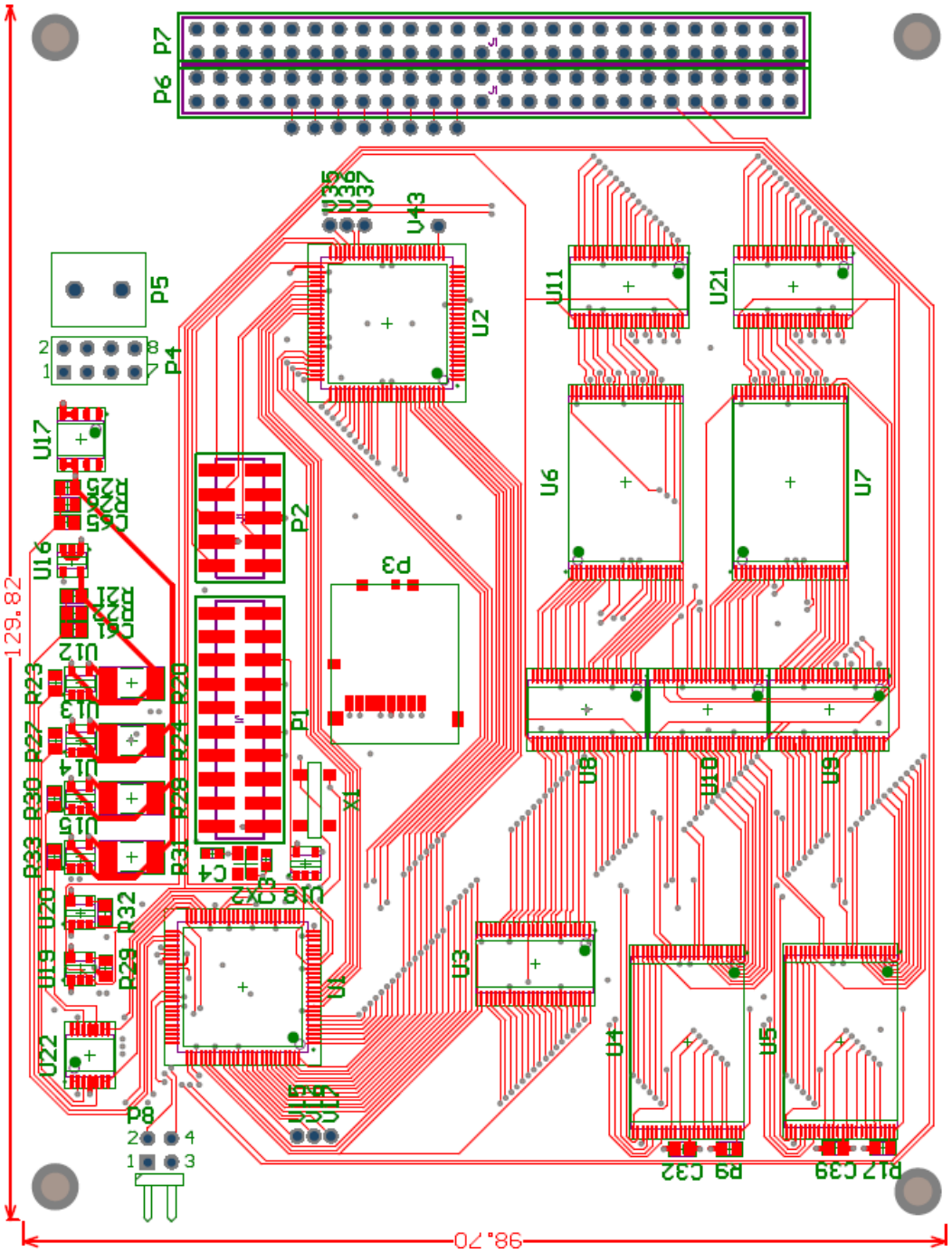


Figure C.2: CubeSat Mechanical Layout Standard (PC/104 Based). [23]

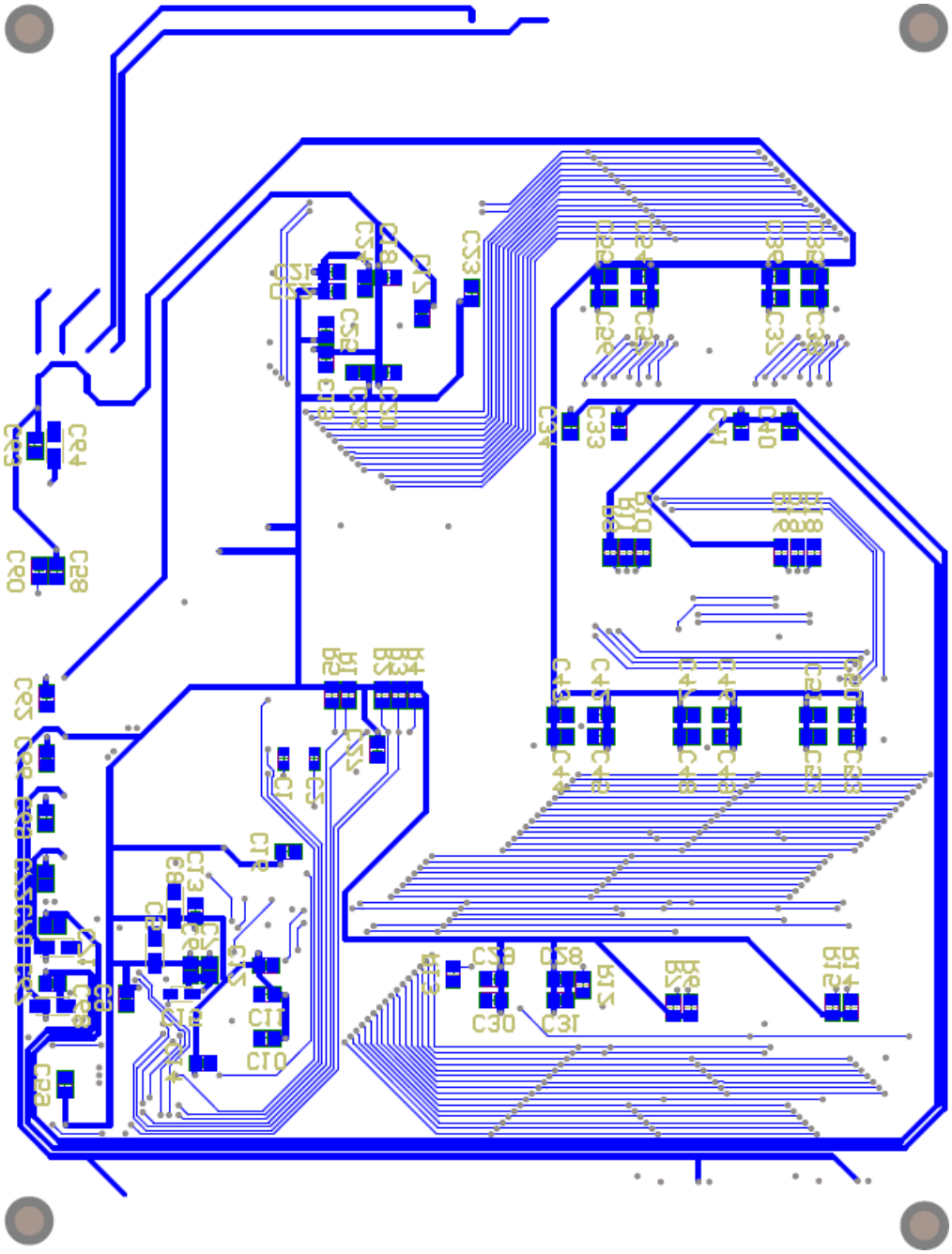
C.2 ADCS OBC LAYOUT

The following section will provide the physical layout of the ADCS OBC.

C.2.1 TOP LAYER



C.2.2 BOTTOM LAYER



C.2.3 ADCS OBC PHOTO

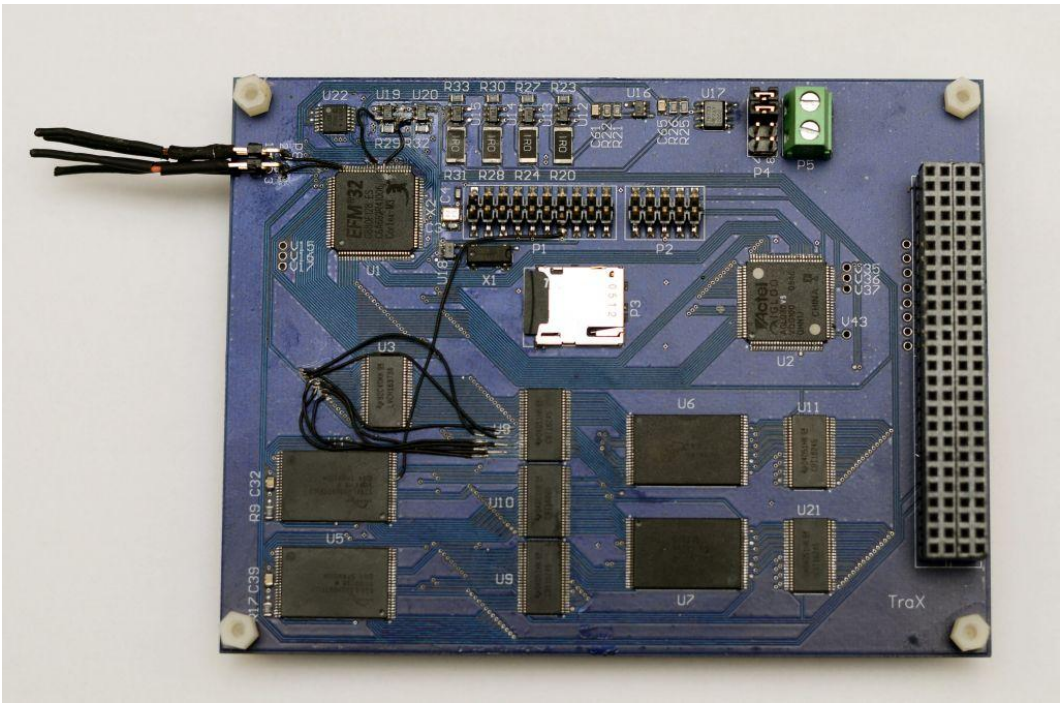


Figure C.3: ADCS OBC Prototype.

D DETAILED DRIVER DESCRIPTIONS

The following section will give a detailed description of defines, enums and functions developed for the ADCS OBC drivers, known as the *Board Support Package* (BSP).

D.1 REAL TIME CLOCK

The following is a detailed description of the RTC driver developed for the ADCS OBC.

Defines:

- **#define** BSP_RTC_IncSec 1

  Description: The time in seconds between each interrupt for the RTC.

Functions:

- **void** BSP_RTC_Init (**void**)

  Description: Initializes the RTC on the OBC by enabling its clock, setting the interrupt interval, enabling the interrupt and resetting the UNIX time counter to zero.
  Inputs: None.
  Returns: None.

- **void** BSP_RTC_SetUnixTime (**uint32_t** newUnixTime)

  Description: Sets the current UNIX time counter value to the given value.
  Inputs:
  - *newUnixTime*: New value for the RTC UNIX time counter.

  Returns: None.

- **uint32_t** BSP_RTC_GetUnixTime (**void**)

  Description: Gets the current UNIX time counter value.
  Inputs: None.
  Returns: Current UNIX time counter value.

- **void** BSP_RTC_IncUnixTime (**void**)

  Description: Increments the current UNIX time counter value with a predefined value (BSP_RTC_IncSec).
  Inputs: None.
  Returns: None.

D.2 WATCHDOG

The following is a detailed description of the watchdog driver developed for the ADCS OBC.

Defines:

- **#define** extWDGPort gpioPortF

  Description: The GPIO port group on the MCU the external watchdog input is allocated to.

- **#define** extWDGPin 8

  Description: The GPIO pin number within the port group on the MCU the external watchdog input is allocated to.

Functions:

- **void** BSP_WDG_Init (**void**)

  Description: The GPIO pin connected to the external watchdog is initialized and the internal watchdog settings and counter value is initialized.
  Inputs: None.
  Returns: None.

- **void** BSP_WDG_ToggleExt (**void**)

  Description: Toggles the external watchdog to reset it reset its timeout counter.
  Inputs: None.
  Returns: None.

- **void** BSP_WDG_ToggleInt (**void**)

  Description: Toggles the internal watchdog to reset it reset its timeout counter.
  Inputs: None.
  Returns: None.

D.3 ANALOG TO DIGITAL CONVERTER

The following is a detailed description of the ADC driver developed for the ADCS OBC.

Type Defines:

- **enum** ADC_Channel_TypeDef

  Description: The different ADC channels that are sampled on the ADCS OBC.
  Values:
  - *CURRENT_SRAM1* = 0
  - *CURRENT_SRAM2* = 1
  - *CURRENT_1V5* = 2
  - *CURRENT_3V3* = 3

Functions:

- **void** BSP_ADC_Init (**void**)

  Description: The ADC clocks are enabled, the DMA channel is configured and the ADC settings are configured.
  Inputs: None.
  Returns: None.

- **void** BSP_ADC_Scan (**bool** wait)

  Description: Starts the ADC scan of all channels on the ADCS OBC which are copied to memory by the DMA controller as they are sampled.
  Inputs:
  - *wait*: Indicates to the function whether it should return after ADC scan has started or wait for it to complete.

  Returns: None.

- **bool** BSP_ADC_IsScanComplete (**void**)

  Description: Returns true if an ADC scan is currently in progress otherwise false.
  Inputs: None
  Returns: True if an ADC scan is currently in progress otherwise false.

- **uint16_t*** BSP_ADC_GetAllData (**void**)

  Description: Returns a pointer to the data array containing the values of the latest ADC samples.
  Inputs: None
  Returns: Pointer to the data array containing the values of the latest ADC samples ADC.

- **uint16_t** BSP_ADC_GetData (**ADC_Channel_TypeDef** channel)

  Description: Returns the latest sampled value of the specified ADC channel.
  Inputs: None
  Returns: The latest sampled value of the specified ADC channel.

D.4 MICROSD

The following is a detailed description of the microSD driver developed for the ADCS OBC.

Defines:

- **#define** TEST_FILENAME "test.txt"

  Description: The filename of the text file created by the test program during microSD test BSP_MSD_TEST().

- **#define** LOG_FILENAME "log.txt"

  Description: The filename of the text file used to store all telemetry data.

Functions:

- **void** BSP_MSD_Init (**void**)

  Description: Initialized the SPI interface, microSD card and FAT filesystem.
  Inputs: None.
  Returns: None.

- **int** BSP_MSD_Write (**int8_t*** data, **int** size)

  Description: Writes the specified data to the LOG file.

  Inputs:

  - *data*: The data in bytes to be written to the LOG file.
  - *size:* The amount of bytes to be written to the LOG file.

  Returns: 1 if the writing process is successfully completed otherwise 0.

- **int8_t*** BSP_MSD_Read (**void**)

  Description: Reads data from log file into buffer variable.

  Inputs: None.

  Returns: A pointer to the buffer variable containing the data in the LOG file.

BSP_MSD_Read	**Reads data from log file into buffer variable.**

- **void** BSP_MSD_Test (**void**)

  Description: Executes a program to test the microSD card by initializing the microSD card, creating a file, writing a string to the file, reading a string from the file and closing the file. This function is used for debugging purposes.

  Inputs: None.

  Returns: None.

D.5 EXTERNAL BUS INTERFACE

The following is a detailed description of the EBI driver developed for the ADCS OBC.

Defines:

- **#define** EXT_SRAM_BASE_ADDRESS ((**uint8_t***) 0x88000000UL)

  Description: The pointer value of the base address for the external SRAM module.

Type Defines:

- **enum** EBI_SRAMSelect_TypeDef

Description: Used to indicate a specific SRAM module.

Values:

- *SRAM1* = 0
- *SRAM2* = 1

Functions:

- **void** BSP_EBI_Init (**void**)

  Description: Initialized the clocks, pins, timings and interrupts for the EBI.
  Inputs: None.
  Returns: None.

- **void** BSP_EBI_TogglePow (**EBI_SRAMSelect_TypeDef** module)

  Description: Toggles the power to a specified SRAM module.
  Inputs:
  - *module*: The SRAM module's power to be toggled.

  Returns: None.

- **void** BSP_EBI_ToggleBuf (**EBI_SRAMSelect_TypeDef** module)

  Description: Isolates a specified SRAM module from the address/data bus.
  Inputs:
  - *module*: The SRAM module to be isolated from the address/data bus.

  Returns: None.

D.6 UNIVERSAL ASYNCHRONOUS RECEIVER / TRANSMITTER

The following is a detailed description of the UART driver developed for the ADCS OBC.

Functions:

- **void** BSP_UART_Init (**void**)

  Description: Initializes the clock, DMA, pins and interrupt for the UART.
  Inputs: None.
  Returns: None.

- **void** BSP_UART_TxBuffer (**uint8_t** id, **uint8_t*** buffer, **int** size)

  Description: Transmits an ID byte and specified amount of data bytes with the UART.
  Inputs:

- *id*: The identification byte of the following data. The ID byte is used by the receiver to identify the incoming data.
- *buffer:* The data to be transmitted after the ID byte.
- *size:* The amount of bytes to be transmitted.

Returns: None.

- **bool** BSP_UART_IsTxComplete (**void**)

  Description: Returns true if no transmission is in progress.

  Inputs: None

  Returns: True if no transmission is in progress, otherwise false.

E EDAC DESIGN

E.1 EDAC FPGA IMPLEMENTATION

The detail design of the EDAC system which was implemented on the FPGA will be shown in the following section. The EDAC system consists of two Reed-Muller (1,3) linear code blocks which were implemented on the FPGA in VHDL code. The code for one RM(1,3) linear code block follows:

```
-- File:    rm_13.vhd
-- Desc:    A Reed-Muller (1,3) linear code block implementation.
Encodes a 4-bit data word in systematic form and decodes a 8-bit
codeword, detecting up to three errors and correcting up to one
error.
-- in:      enc, dec, nCS - control signals from the MCU.
-- out:     errors(0:3) - flag signal indicating if an error was
detected or corrected.
-- inout:   data - 4-bit data bus to the MCU bus.
--          code - 8-bit codeword to the SRAM bus.

library std;
library ieee;

use ieee.std_logic_1164.all;

entity rm13 is
port (
    enc,dec,nCS : in     std_logic;
    errors      : out    std_logic_vector(1 downto 0);
    data        : inout  std_logic_vector(3 downto 0);
    code        : inout  std_logic_vector(7 downto 0)
);
end rm13;

architecture behaviour of rm13 is

signal syndrome : std_logic_vector(0 to 3);

begin
    process(nCS, enc, dec, syndrome)
    begin
        -- encoding process
        if (enc = '0' and nCS = '0') then
            data <= "ZZZZ";
            errors <= "ZZ";
            code(0) <= data(0);
            code(1) <= data(1);
            code(2) <= data(2);
            code(3) <= data(3);
            code(4) <= data(0) xor data(1) xor data(2);
            code(5) <= data(0) xor data(1) xor data(3);
            code(6) <= data(0) xor data(2) xor data(3);
            code(7) <= data(1) xor data(2) xor data(3);
```

```
            syndrome <= "ZZZZ";

        -- decoding process
        elsif (dec = '0' and nCS = '0') then
            code <= "ZZZZZZZZ";
            syndrome(0) <= code(1) xor code(2) xor code(3) xor
code(7);
            syndrome(1) <= code(0) xor code(2) xor code(3) xor
code(6);
            syndrome(2) <= code(0) xor code(1) xor code(3) xor
code(5);
            syndrome(3) <= code(0) xor code(1) xor code(2) xor
code(4);

            -- indicate errors by checking syndrome
            case syndrome is
                -- no errors
                when "0000" =>
                    data <= code (3 downto 0);
                    errors <= "11";
                -- one error in data bits
                when "0111" =>
                    data(0) <= code(0);
                    data(1) <= code(1);
                    data(2) <= code(2);
                    data(3) <= code(3);
                    errors  <= "01";
                -- one error in data bits
                when "1011" =>
                    data(0) <= code(0);
                    data(1) <= not code(1);
                    data(2) <= code(2);
                    data(3) <= code(3);
                    errors  <= "01";
                -- one error  in data bits
                when "1101" =>
                    data(0) <= code(0);
                    data(1) <= code(1);
                    data(2) <= not code(2);
                    data(3) <= code(3);
                    errors  <= "01";
                -- one error in data bits
                when "1110" =>
                    data(0) <= code(0);
                    data(1) <= code(1);
                    data(2) <= code(2);
                    data(3) <= not code(3);
                    errors  <= "01";
                -- one error in parity check bits
                when "0001" =>
                    data <= code (3 downto 0);
                    errors  <= "10";
                -- one error in parity check bits
                when "0010" =>
                    data <= code (3 downto 0);
                    errors  <= "10";
                -- one error in parity check bits
                when "0100" =>
```

```
                    data <= code (3 downto 0);
                    errors  <= "10";
                -- one error in parity check bits
                when "1000" =>
                    data <= code (3 downto 0);
                    errors  <= "10";
                -- too many errors
                when others =>
                    data <= code (3 downto 0);
                    errors  <= "00";
            end case;

        -- high impedance state
        else
            data     <= "ZZZZ";
            code     <= "ZZZZZZZZ";
            errors   <= "ZZ";
            syndrome <= "ZZZZ";
        end if;
    end process;
end behaviour;
```

Two of these RM(1,3) LCB are implemented in the FPGA to encode the 8-bit MCU data bus into the 16-bit codeword bus of the SRAM. The top level implementation of the FPGA is shown in Figure E.1. The two RM(1,3) LBCs can be seen as well as how the codewords are interleaved before being put on the SRAM bus.

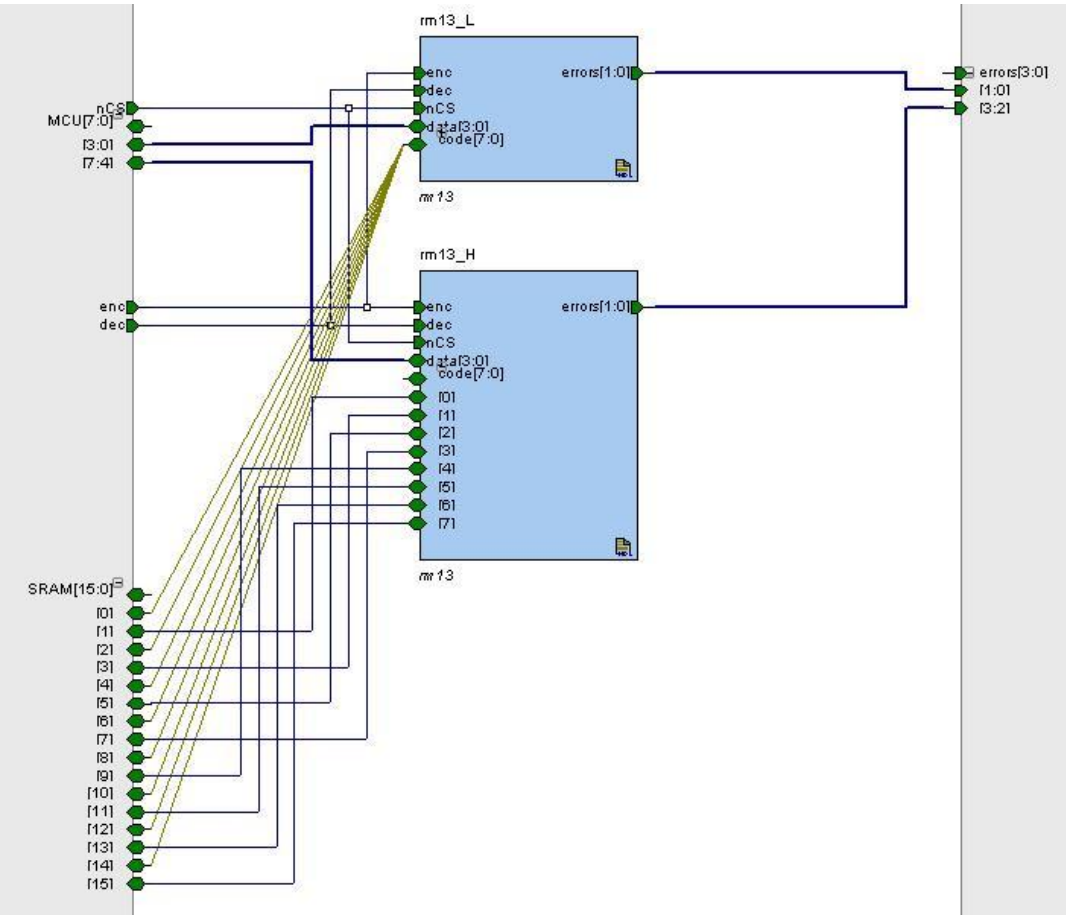


Figure E.1: Top Level Implementation of EDAC on FPGA

E.2 EXAMPLE

The following section will go through an example which will showcase the functioning of the EDAC subsystem.

1. Write / Encode

The MCU bus data, which is 8-bits, is split into two 4-bit data words which is separately fed into their respective RM(1,3) linear code blocks.

$$MCU\ data = 01010011b \tag{E.1}$$

$$\boldsymbol{d1} = 0101b, \qquad \boldsymbol{d2} = 0011b \tag{E.2}$$

The 4-bit data words are multiplied by the RM(1,3) generator matrix $\boldsymbol{G}$ in systematic form to generate the respective codewords ***c1*** and ***c2***.

$$\boldsymbol{G} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{bmatrix} \tag{E.3}$$

$$\boldsymbol{c} = \boldsymbol{dG} \tag{E.4}$$

$$\boldsymbol{c1} = 01011010b, \qquad \boldsymbol{c2} = 00111100b \tag{E.5}$$

The codewords c1 and c2 are interleaved before output to the SRAM data bus.

$$SRAM\ data = 0010011111011000b \tag{E.6}$$

2. Read / Decode

The SRAM bus data, which consists of 16-bits, is de-interleaved into two codewords ***r1*** and ***r2***.

$$SRAM\ data = \mathbf{1}010011111011000b \tag{E.7}$$

$$\boldsymbol{r1} = \mathbf{1}1011010b, \qquad \boldsymbol{r2} = 00111100b \tag{E.8}$$

Codewords ***r1*** and ***r2*** are then multiplied by their respective parity check matrix $\boldsymbol{H}$ to generate the syndromes ***s1*** and ***s2***.

$$\boldsymbol{H} = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{E.9}$$

$$\boldsymbol{s} = \boldsymbol{rH}^T \tag{E.10}$$

$$\boldsymbol{s1}' = 0111b, \qquad \boldsymbol{s2}' = 0000b \tag{E.11}$$

The syndromes are compared with the parity check matrix H to see if an error has occurred and if this error can be corrected. In this example a single bit error has occurred in codeword r1, which corresponds to the syndrome matching a row vector in $H^{T,}$ and no error in codeword r2, which corresponds to a zero syndrome.

The bit flip is in codeword r1 is identified and corrected and the corresponding error signals are flagged. The correct codewords are put together and output on the MCU data bus.

$$\boldsymbol{r1} = \mathbf{0}1011010b, \qquad \boldsymbol{r2} = 00111100b \tag{E.12}$$

$$\boldsymbol{d} = \boldsymbol{r1}[0:3] + \boldsymbol{r2}[0:3] = 01010011b \tag{E.13}$$

F SUPPORT FILES CD

The following CD includes all the support files used in the design and development of the ADCS OBC.

These include:

- Datasheets of all the components used for the ADCS OBC.
- Altium design files for the ADCS OBC.
- Source code for MCU and FPGA development for ADCS OBC.

BIBLIOGRAPHY

[1] CPUT, CubeSat Projects, http://active.cput.ac.za/fsati/public/index.asp?pageid=956, July 2011.

[2] Wikipedia, SUNSAT, http://en.wikipedia.org/wiki/SUNSAT, July 2011.

[3] Wikipedia, SumbandilaSat, http://en.wikipedia.org/wiki/SumbandilaSat, July 2011

[4] Surrey Satellite Technology LTD, Science & Exploration - STRaND Nanosatellite, http://www.sstl.co.uk/Divisions/Earth-Observation---Science/Science---Exploration/STRaND-nanosatellite/STRaND-1-Factsheet, October 2011.

[5] Mr. Hank Heidt, Prof. Jordi Puig-Suari, Prof. Augustus S. Moore, Prof. Shinichi Nakasuka, and Prof. Robert J. Twiggs, *CubeSat: A new Generation of Picosatellite for Education and Industry Low-Cost Space*, 14TH Annual/USU Conference on Small Satellites, North Logan, Utah, USA, 2000, pp. 1-2,6,18.

[6] Jordi Puig-Suari, Clark Turner, and William Ahlgren, *Development of the Standard CubeSat Deployer and a CubeSat Class PicoSatellite*, Aerospace Conference, Big Sky, Montana, USA, 2001, pp. 1-3,6.

[7] Clyde Space, http://www.clyde-space.com/cubesat_shop/cubesat_platforms/3u_platform, July 2011.

[8] ISIS. CubeSatShop. [Online]. http://www.cubesatshop.com/, October 2011.

[9] W. T. Thompson, *Spin stabilization of attitude against gravity torque*, J. Astronaut. Sci., vol. 9, no. 1, 1962, pp. 31-33.

[10] James R. Wertz, Wiley J. Larson, et al, *Space Mission Analysis and Design*, 3rd ed., James R. Wertz and Wiley J. Larson, Eds. California, USA: Microcosm Press, Kluwer Academic Publishers, 1999.

[11] H. Grobler, *Aspects Affecting the Design of a Low Earth Orbit Satellite On-Board Computer*, in Engineering Masters Thesis, University of Stellenbosch, 2000.

[12] Kenneth A. LaBel, Michele M. Gates, Amy K. Moran, et al. *Commercial Microelectronics Technologies for Applications in the Satellite Radiation Environment*, NASA/GSFC Radiation Effects & Analysis, http://radhome.gsfc.nasa.gov/radhome/papers/aspen.htm., October 2011.

[13] ARM, http://www.arm.com/products/processors/cortex-m/index.php, August 2011.

[14] ATMEL,

http://www.atmel.com/dyn/products/devices.asp?category_id=163&family_id=607&subfamily_id=2138, August 2011.

[15] Pumpkin, Flight Module FM430 Datasheet, Revision C, June 2008.

[16] eoPortal, CanX-6 (Canadian Advanced Nanosatellite eXperiment-6) / NTS, http://events.eoportal.org/get_announce.php?an_id=10000802, April 2008.

[17] Energy Micro, EFM32G Reference Manual, Revision 1.10, April 2011.

[18] Energy Micro, Cortex-M3 Reference Manual, Revision 1.00, February 2011.

[19] Wayne Hendrix Wolf, *Computers as Components*, 2nd ed. Montana, USA: Morgan Kaufmann Publishers, 2008.

[20] ST Microelectronics, Watchdog STWD100 Datasheet, Revision 5, 2008.

[21] Tetsuo Miyahira and Gary Swift, *Evaluation of Radiation Effects in Flash Memories*, in Military and Aerospace Applications of Programmable Devices and Technologies Conference, Maryland, 1998, pp. 1-4.

[22] ATMEL, EDAC 29C516E Datasheet, Revision E, 2007.

[23] Gregor Dreijer, *The Evaluation of an ARM-based On-Board Computer for a Low Earth Orbit Satellite*, Engineering Masters Thesis, University of Stellenbosch, 2002.

[24] Christiaan Johannes Petrus Brand, *The Developmnet of an ARM-Based OBC for a Nanosatellite*, Engineering Masters Thesis, University of Stellenbosch, 2007.

[25] Actel, IGLOO nano Low Power Flash FPGAs Datasheet, Revision 1.11, 2010.

[26] Hanco Evert Loubser, *The development of Sun and Nadir sensors for a solar sail CubeSat*, Engineering Masters Thesis, University of Stellenbosch, 2010.

[27] Texas Instruments, Voltage Regulator TPS767xx Datasheet, January 2004 Revision.

[28] Texas Instruments, Voltage Regulator TPS732xx Datasheet, 2008, May 2008 Revision.

[29] Fairchild Semiconductors, Load Switch FPF2123-FPF2125 Datasheet, Revision F, 2008.

[30] Cypress Semiconductors, SRAM CY62167DV30 Datasheet, Revision C, 2003.

[31] Energy Micro, http://www.energymicro.com/, October 2011.

[32] Wikipedia, Unix Time, http://en.wikipedia.org/wiki/Unix_time, August 2011.

[33] B. P. Lathi, *Modern Digital and Analog Comunication Systems*, Third Edition ed. New York, USA: Oxford University Press, Inc., 1998.

[34] Wolfram Alpha, Reed-Muller (1,3), http://www.wolframalpha.com/input/?i=reed-muller+1%2C+3, August 2011.

[35] Wikipedia, Non Blocking, http://en.wikipedia.org/wiki/Non-blocking_algorithm, September 2011.

[36] Energy Micro, EFM32GG Reference Manual, Revision 0.90, 2011.

[37] ATMEL, Parallel EEPROM (AT28C64B) Datasheet, October 2006 Revision.

[38] Spansion, S29AL008D Datasheet, Revision A.11, 2009.

[39] ATMEL, EEPROM AT28BV256 Datasheet, 2009 Revision.

[40] ATMEL, EEPROM AT28HC256 Datasheet, 2009 Revision.

[41] Wikipedia, RTOS, http://en.wikipedia.org/wiki/Real-time_operating_system, September 2011.

[42] Energy Micro, AN0002 - Hardware Design Considerations, Revision 1.31, 2011.

[43] Cypress Semiconductor, SRAM System Design Guidelines, October 2002 Revision.

[44] Texas Instruments, Current Sensor INA139 Datasheet, January 2003 Revision.

[45] Clyde Space LTD, User Manual: CubeSat 1U Electronic Power System and Batteries: CS-1UEPS2-NB/-10/-20, Issue A, 2010.

[46] ALLSPACE, http://www.cubesat.de/downloads/cbs.pdf, August 2011.

[47] NASA, Radiation Belts, http://radbelts.gsfc.nasa.gov/outreach/Radbelts3.html, October 2011.

[48] Olmo A. M. Franco, The Robotics Institute of Yucatan, http://www.triy.org/ENG/CubeSat_Intro.htm, September 2009.

[49] Energy Micro, AN0030 - FAT on SD Card, Revision 1.01, 2011.