



UNIVERSITEIT • STELLENBOSCH • UNIVERSITY
jou kennisvennoot • your knowledge partner

The Development of an ARM-based OBC for a Nanosatellite

by

Christiaan Johannes Petrus Brand



*Thesis presented at the University of Stellenbosch in partial
fulfilment of the requirements for the degree of*

Master of Science in Engineering
(Electronic Engineering with Computer Science)

Department of Electrical Engineering
University of Stellenbosch
Private Bag X1, 7602 Matieland, South Africa

Study leader: Prof P.J. Bakkes

December 2007

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature:

C.J.P. Brand

Date:



Abstract

Next-generation nanosatellites are becoming a very cost effective solution to gain access to space. Modern manufacturing technology together with low power low cost devices makes the development of nanosatellites, using standard industrial components, very attractive. A typical nanosatellite will have only one microprocessor, capable of performing all the computing tasks onboard the satellite - housekeeping, AODC (Attitude and Orbit Control) and instructing the different payloads aboard the satellite.

One of the major requirements was to choose a processor from a dominant manufacturer in the market that will still be available for future satellite missions. Just as the 8051 dominated the 8-bit market, the ARM7 processor is fast becoming a market leader in the segment for 16-bit applications. ARM processors has also been used much in handheld devices in recent years - which emphasize the low power requirements and stability of these processors in embedded applications.

This thesis investigates the different processors that are currently available. A complete system design is done, taking into account all the different modules needed onboard a very small Low Earth Orbit (LEO) satellite. Finally, some test results are given showing how this system can be reliably used onboard a nanosatellite in future.

Opsomming

Moderne nanosatelliete se lae koste maak dit 'n baie aantreklike oplossing om toegang tot die ruimte te verkry. Hedendaagse vervaardigingstegnieke tesame met goedkoop, lae drywing komponente maak die ontwikkeling van nanosatelliete, deur gebruik te maak van alledaagse industriële komponente, baie aantreklik. 'n Tipiese nanosatelliet beskik oor slegs een mikroverwerker wat instaat is om al die verwerking op die satelliet te behartig - Orientasie en Wentelbaan beheer, algemene onderhoud, en die instruksies na, en van kameras en ander stooreenhede.

Een van die vereistes was dat die mikroverwerker wat gekies word, steeds beskikbaar sal wees in die afsienbare toekoms. Soos die 8-bis mark gedomineer is deur die 8051 verwerker, is die ARM7 verwerker vinnig besig om 'n markleier in die 16-bis segment te word. ARM verwerkers is deesdae volop te vind in battery-aangedrewe handtoestelle: dit beklemtoon juis die lae kragverbruik en stabiliteit van hierdie verwerkers in die *toegewyste* mark.

Hierdie tesis ondersoek die verskillende verwerkers wat tans beskikbaar is. 'n Volledige stelselontwerp word gedoen waarin al die verskillende modules wat benodig word op 'n aanboord rekenaarstelsel, behandel word. Laastens word 'n evaluering van die stelsel gedoen en toetsresultate toon aan dat hierdie stelsel in die toekoms betroubaar op 'n nanosatelliet gebruik kan word.

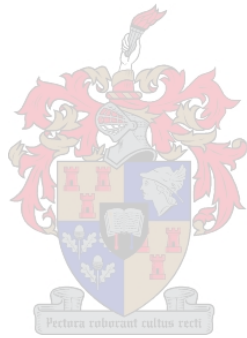
Acknowledgements

The author would like to thank the following people for their contribution towards this project.

- My supervisor, **Professor P.J. Bakkes**, for his guidance throughout this project.
- The ZA-SAT administrators, especially **Professor W.H. Steyn** who made this opportunity possible.
- All the people of the ESL, specifically Niel Muller, for all the help with the *challenging side projects*.
- L^AT_EX for the excellent typesetting of this document.
- **My family** for all their support.
- **Karlien**, for understanding all the long nights in the laboratory.
- **God**, who assisted me through out it all.

Contents

Declaration	i
Abstract	ii
Opsomming	iii
Acknowledgements	iv
Contents	v
Abbreviations	viii
List of Figures	x
List of Tables	xii
1 Introduction	1
1.1 Background	1
1.2 Nanosatellite Design	2
1.3 An ARM7-Based Onboard Computer	2
1.4 Document Outline	3
2 Selection of hardware	4
2.1 Onboard Computer Requirements	4
2.2 Processor	5
2.3 FPGA	6
3 Background on the Atmel ARM	8
3.1 Overview of the Atmel ARM AT91SAM7A2	8
3.2 Power Consumption	12



3.3	Memory Interface	15
3.4	General Analog and Digital Interfaces	19
3.5	Communication Interfaces	20
3.6	Reliability	23
4	Detailed Design of the OBC	25
4.1	Design Overview	25
4.2	Physical Printed Circuit Board	26
4.3	Power Supply	27
4.4	Communications Interfaces	30
4.5	Actel ProAsic Plus FPGA	33
4.6	Memory System	39
4.7	Miscellaneous	40
5	Software	43
5.1	JTAG Development Environment	43
5.2	Software Development	44
6	Tests and Measurements	57
6.1	Memory System and EDAC	57
6.2	Peripherals	60
6.3	Measurements	62
7	Conclusions and Recommendations	64
7.1	Conclusions	64
7.2	Recommendations	65
	Appendices	67
	A Physical PCB Layout	68
	B Design Schematics	70
	C Peripheral Software Support	71
C.1	CAN Bus	71
C.2	UART Communication	75
C.3	PIO Controller	80
C.4	Analog to Digital Conversion	82
C.5	Software LVDS Interface	86

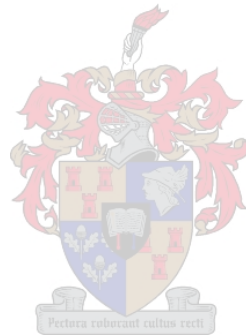
<i>CONTENTS</i>	vii
D LVDS to SPI Design	88
E LVDS Output	89
F AT91SAM7A2 Complimentary Hardware Design	91
G Actel FPGA Design	94
Bibliography	100



Abbreviations

- A/D - Analog to Digital
- AODCS - Attitude Determination and Control System
- ARM - Advanced RISC Machines
- CAN - Controller Area Network
- CFI - Common Flash Interface
- COTS - Commercial Off The Shelf
- DMA - Direct Memory Access
- EBI - External Bus Interface
- EDAC - Error Detection and Correction
- ESA - European Space Agency
- FIFO - First In First Out
- IGRF - International Geomagnetic Reference Field
- LEO - Low Earth Orbit
- LLC - Logical Link Control
- LVDS - Low Voltage Differential Signalling
- MMU - Memory Management Unit
- MSD - Mass Storage Device
- NASA - National Aeronautics and Space Administration

- OBC - Onboard Computer
- PIO - Programmed Input / Output
- RISC - Reduced Instruction Set Computing
- RTC - Real Time Clock
- SEE - Single Event Effects
- SEL - Single Event Latchup
- SEU - Single Event Upset
- SPI - Serial Peripheral Interface
- SRAM - Static Random Access Memory



List of Figures

1.1	Block Diagram of Typical Nanosatellite [15]	2
3.1	AT91SAM7A2 Block Diagram [3]	9
3.2	AT91SAM7A2 Clock Manager [3]	11
3.3	Standard Read Cycle	17
3.4	Early Read Cycle	18
3.5	Write Cycle	18
3.6	Write Cycle with 2 Wait States inserted	19
3.7	ISO/OSI Representation of a CAN Node [3]	21
3.8	A Typical CAN Data Frame	22
4.1	Block Diagram of different OBC subsystems	26
4.2	Current Monitoring of PCB	28
4.3	Voltage Monitoring of PCB	29
4.4	Nanosatellite LVDS Data Link	33
4.5	EDAC Block Diagram	37
4.6	OBC Memory System	39
5.1	Representation of hardware abstraction layer	47
5.2	Received bit sequence from LM70	50
5.3	Flow Diagram of Software SPI / LVDS Driver	51
5.4	Read-Only SPI Connections between peripheral and AT91SAM7A2	51
5.5	TimeKeeper Register Map [13]	52
5.6	TimeKeeper Read Mode Sequence [13]	53
5.7	TimeKeeper Write Mode Sequence [13]	53
6.1	EDAC Encoding of data word	58
6.2	EDAC Decoding of data word	58

6.3 EDAC Decoding of invalid data word 58

6.4 EDAC Writing Sequence 59

6.5 EDAC Writing Sequence (zoomed in) 59

6.6 Data received by SPI (software emulation) on AT91SAM7A2 61

A.1 Schematic Representation of PC Board Layout for Nanosatellite
OBC 68

A.2 PC Board Layout for Nanosatellite OBC 69

A.3 Photo of Nanosatellite OBC 69

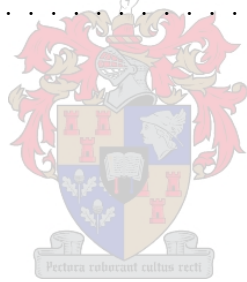
C.1 Partition of the bit time 72

D.1 Block diagram of LVDS to SPI link 88

E.1 LVDS Output using MAX9157 90

F.1 Reset Controller for AT91SAM7A2 91

F.2 PLL RC Filter Circuit 93



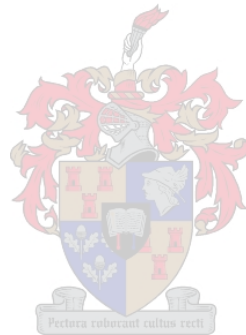
List of Tables

2.1	Processors reviewed for use on OBC (current measurements @ 3.3 V)	7
3.1	Typical Current Consumption for the AT91SAM7A2 [3]	13
3.2	Power Management Blocks for on chip peripherals [3]	14
3.3	Memory Map in Reboot Mode	16
3.4	Memory Map in Remap Mode	16
3.5	Peripheral Memory Mapped I/O Addresses	17
4.1	Mechanical PC Board Specifications	27
4.2	ARM7 UPIO Connections	41
5.1	Functions provided by CAN driver	47
5.2	Functions provided by USART driver	48
5.3	ARM7 UPIO pins configured as outputs at boot time	49
5.4	Functions provided by UPIO driver	49
5.5	Functions provided by RTC driver	53
5.6	Functions provided by Analog to Digital converter	54
5.7	Functions provided by hardware SPI	55
5.8	Functions provided by software LVDS receiver	56
6.1	Current consumption of OBC	62
C.1	CAN Mode Register [0x064]	71
C.2	CAN Mode Register	72
C.3	CAN Control Register [0x060]	73
C.4	USART Mode Register [0x064]	76
C.5	USART Baud Rate Generator Register [0x088]	77
C.6	USART Receiver Time Out Register [0x08C]	77

LIST OF TABLES

xiii

C.7	USART Transmit Holding Register [0x084]	78
C.8	ADC Mode Register [0x064]	83
C.9	ADC DC Conversion Mode Register [0x068]	84
G.1	Pin assignments for Actel ProAsic APA075	95



Chapter 1

Introduction

1.1 Background

A satellite consists of a number of subsystems, each performing its own dedicated task. The primary function of an Onboard Computer (OBC) aboard a satellite, is to facilitate the communication between these different subsystems. The subsystems are normally physically placed on different modules and need to be connected by a reliable data bus. In a nanosatellite design, space is often a limiting factor, and some of the subsystems are integrated directly into the OBC. In this case, software communication between the subsystems replace the physical links but the subsystems are still required to function independently of each other.

To satisfy this requirement, a powerful microprocessor capable of running multiple processes simultaneously will be needed. Running processes will also have to be isolated from each other so that one misbehaving subsystem does not cause a critical failure of other systems.

In recent years the focus has shifted from space qualified components, to commercial off-the-shelf (COTS) hardware that can be slightly adapted or modified to provide adequate reliability in space. These modifications normally involves an extra layer of complexity for error checking and correction since these commercial components are susceptible to radiation errors.

1.2 Nanosatellite Design

Because of the physical small size of a nanosatellite (30cm x 30cm x 20cm) many of the tasks that are normally performed by other subsystems need to be handled by the onboard computer. These subsystems will most likely have their own unique supporting hardware (for example: reaction wheels in the case of the AODCS), but the primary processing will be done onboard the OBC. A Block Diagram showing the different subsystems onboard a typical nanosatellite, is shown in Figure 1.1 [15].

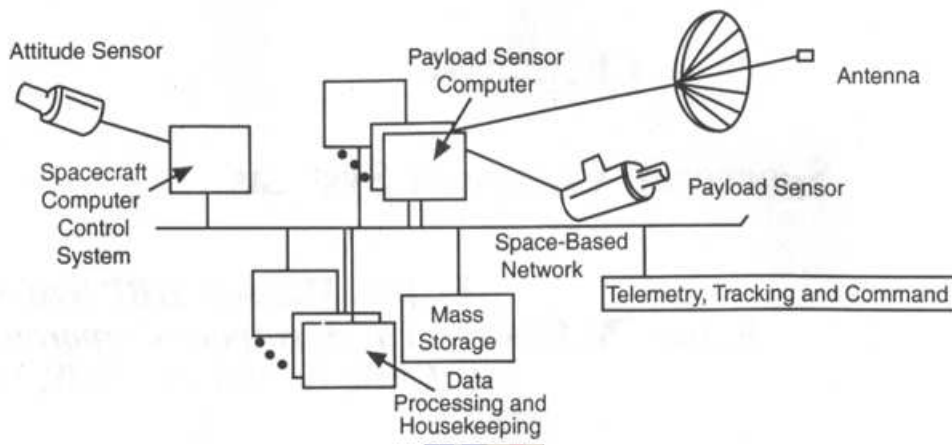


Figure 1.1: Block Diagram of Typical Nanosatellite [15]

1.3 An ARM7-Based Onboard Computer

In the past the 8-bit market was dominated by the Intel 8051 microprocessor. Many OBCs was based on designs using this, and some even featured the more powerful 80386 processor. Real estate on a nanosatellite is much more of a concern than on a traditional microsatellite. We will typically only have space for one microprocessor that will need to take care of all the processing tasks onboard our satellite. Some design constraints and requirements will be set in the chapters that follow, but it is quite clear that we have a huge choice when it comes to the selection of an appropriate microprocessor.

To narrow this choice down, we have decided to primarily look at microprocessors available from Advanced RISC Machines (ARM). As the 8051 overshadowed the 8-bit embedded market, the ARM7 processor currently dominates the 16-bit market segment for embedded processors. This thesis will cover the design and development of an OBC for a LEO nanosatellite based on an ARM7 processor.

1.4 Document Outline

A condensed outline of the document structure is set out here.

- **Chapter 1** introduces the subject of small satellite engineering and gives a broad overview of this thesis.
- **Chapter 2** describes the OBC processor requirements, and the selection of hardware is done.
- **Chapter 3** provides detail functionality of the chosen processor.
- **Chapter 4** goes through the detail hardware design of the OBC.
- **Chapter 5** gives an overview of the AT91SAM7A2 programming model and development of drivers for all the peripherals onboard the OBC.
- **Chapter 6** lists all the tests and measurements performed on the board.
- **Chapter 7** is the concluding chapter which summarizes the design. Areas where more testing might be necessary is discussed and suggestions for possible future modifications are made.

Chapter 2

Selection of hardware

A Satellite OBC consists of different subsystems and therefore selection of multiple components needs to take place. In most cases there are more than one component available to do the job, each with their own advantage- and disadvantages.

2.1 Onboard Computer Requirements

Before any hardware selection can be done, we need to determine the requirements of the onboard computer aboard a nanosatellite. This is summarized as follows:

- Computational Performance
This is generally measured in MIPS. This will be the only processor onboard the nanosatellite capable of performing complex tasks: House-keeping, AODCS and communications.
- Low Power Consumption
Components that has been proven in other battery operated handheld devices (Cellphones, PDAs) will be given preference, since power onboard a satellite is a scarce resource.
- Low Voltage Components
To improve (lower) power consumption further, low voltage components (3.3 V) are favoured above the traditional 5 V systems.

- Availability of components
Components used in this design needs to be commercially available for at least the next three years.
- Programmable Memory Management Unit (MMU)
Memory Protection and paging supported in hardware will be preferable, since misbehaving processes accessing memory outside its scope can easily be identified and stopped without causing instability to the operating system.
- Memory Error Detection and Correction
Cache memory is especially susceptible to SEU's caused by radiation. Memory onboard the satellite should consist only of SRAM and Flash based memories. Since SRAM is also prone to errors caused by radiation, some detection and correction hardware should be implemented.
- I/O Interfaces
Communication with other subsystems onboard the satellite is very important. Dedicated communication channels to the modems, and a high speed link to a storage device also needs to be implemented. For compatibility with devices developed at SunSpace, a LVDS data link is also a requirement.
- History of successful use in a Low Earth Orbit
Previous use of this processor onboard a LEO satellite is ideal.

2.2 Processor

Because of the proven reliability and the feature-rich flavours in which they are available, we will choose an ARM-based processor for the processing unit of this satellite. After the review of several ARM processors (as shown in Table 2.1), the processor which best fitted our requirements set out in the previous section, was the *AT91SAM7A2* ARM7 based processor from Atmel. It has an external bus interface which allows for 6MB of external memory (Asynchronous SRAM and Flash memory are supported), which is controlled by an MMU capable of memory protection and paging. Power consumption in the *active*-state is extremely low at only 900 $\mu\text{A}/\text{MHz}$ [3]. The power of all the peripherals integrated into the processor are controlled by an Advanced

Power Management Unit which allows the system to be modularly powered up and down as different modules are needed. Support for a communications bus architecture called the Controller Area Network (CAN) is also provided by the processor in the form of four different CAN-controllers. High speed communication can be done by using the provided Serial Peripheral Interface (SPI) clocked at the core frequency of the processor (30 MHz).

It was determined that the most computationally complex operation the processor onboard the nanosatellite would need to deal with, is the IGRF modelling for the AODCS system. A simulation was done with the actual IGRF function by using a *AT91SAM7A2* development board. The IGRF models need to be computed once every second, and the ARM7 took 50 ms to run the model. This computes to a utilization of about 5% for one of the most complex tasks onboard the satellite.

An ARM7-based processor has also been used in the design for the Canadian Can-X2 satellite, which is due to be launched in a few months.

2.3 FPGA

Single Event Upsets (SEUs) caused by radiation corrupts the data stored in SRAM cells. Ideally we want to be able to transparently detect and correct errors in our external SRAM. The complete design of this system will follow later, but we will need to implement this design in a FPGA (Field Programmable Gate Array). Some FPGA's use SRAM to store their internal configuration, but because the device itself will then be vulnerable to radiation effects, we will need to make use of a Flash Based FPGA. The Error Detection and Correction code is quite small, and will easily fit in the smallest FPGA available to us. We decided to use the industrial version of the Actel Proasic Plus 75 000 gate FPGA. This version of the FPGA can operate in temperatures between -40°C to $+85^{\circ}\text{C}$. This is the same range as for the *AT91SAM7A2* used in this design and should be more than adequate for use onboard a passive temperature controlled LEO satellite.

CPU	Manufacturer	CAN	I2C	EBI	RTC	Consumption	MIPS
AT91SAM7A1	Atmel	Yes	No	Yes	No	900 μ A/MHz	36
AT91SAM7A2	Atmel	Yes	No	Yes	No	80mA	30
AT91SAM7A3	Atmel	Yes	No	No	No	100mA	27
AT91SAM7xX	Atmel	Yes	No	No	Yes	100mA	27
MAC7111	Freescale	Yes	Yes	Yes	Yes	100mA	40
MAC7115	Freescale	Yes	Yes	Yes	Yes	100mA	40
MAC7116	Freescale	Yes	Yes	Yes	Yes	100mA	40
MAC7131	Freescale	Yes	Yes	Yes	Yes	100mA	40
MAC7134	Freescale	Yes	Yes	Yes	Yes	100mA	40
MAC7135	Freescale	Yes	Yes	Yes	Yes	100mA	40
MAC7136	Freescale	Yes	Yes	Yes	Yes	100mA	40
LPC2210	Philips	No	Yes	Yes	Yes	100mA	30
LPC2212	Philips	No	Yes	Yes	Yes	100mA	30
LPC2214	Philips	No	Yes	Yes	Yes	100mA	30
LPC2220FBD144	Philips	No	Yes	Yes	Yes	100mA	30
LPC2220FET144	Philips	No	Yes	Yes	Yes	100mA	30
LPC2290	Philips	Yes	Yes	Yes	Yes	100mA	30
LPC2292	Philips	Yes	Yes	Yes	Yes	100mA	30
LPC2294	Philips	Yes	Yes	Yes	Yes	100mA	30
LH75400	Sharp	Yes	Yes	No	Yes	100mA	25
LH75401	Sharp	Yes	Yes	No	Yes	100mA	25
STR710FZ1	ST Micro	Yes	Yes	Yes	Yes	110.6mA	59
STR710FZ2	ST Micro	Yes	Yes	Yes	Yes	110.6mA	59
TMS470R1A288	Texas Instruments	Yes	Yes	Yes	Yes	115mA	48
TMS470R1A384	Texas Instruments	Yes	Yes	Yes	Yes	115mA	48
TMS470R1B1M	Texas Instruments	Yes	Yes	Yes	Yes	110mA	48

Table 2.1: Processors reviewed for use on OBC (current measurements @ 3.3 V)

Chapter 3

Background on the Atmel ARM

As stated in the previous chapter, Atmel's implementation of the ARM7TDMI core, the AT91SAM7A2, was identified as the best candidate for a nanosatellite computer application. This chapter will give some insight on the features of this processor and the reasons behind its selection.

3.1 Overview of the Atmel ARM AT91SAM7A2

The AT91SAM7A2 can be divided into 4 parts, namely the Processing Core, Advanced Memory Controller, Clock Manager and Peripherals. Figure 3.1 shows the block diagram of the AT91SAM7A2.

3.1.1 Processing Core

The AT91SAM7A2 processor is an implementation of the ARMv4T architecture. Microprocessor architectures traditionally have the same width for instructions and data. In comparison with 16-bit architectures, 32-bit architectures exhibit higher performance when manipulating 32-bit data, and can address a large address space more efficiently. 16-bit architectures typically have higher code density (smaller code) than 32-bit architectures, but approximately half the performance [4].

The Thumb instruction set from ARM implements a 16-bit instruction set on a 32-bit architecture to effectively get the smallest code footprint with

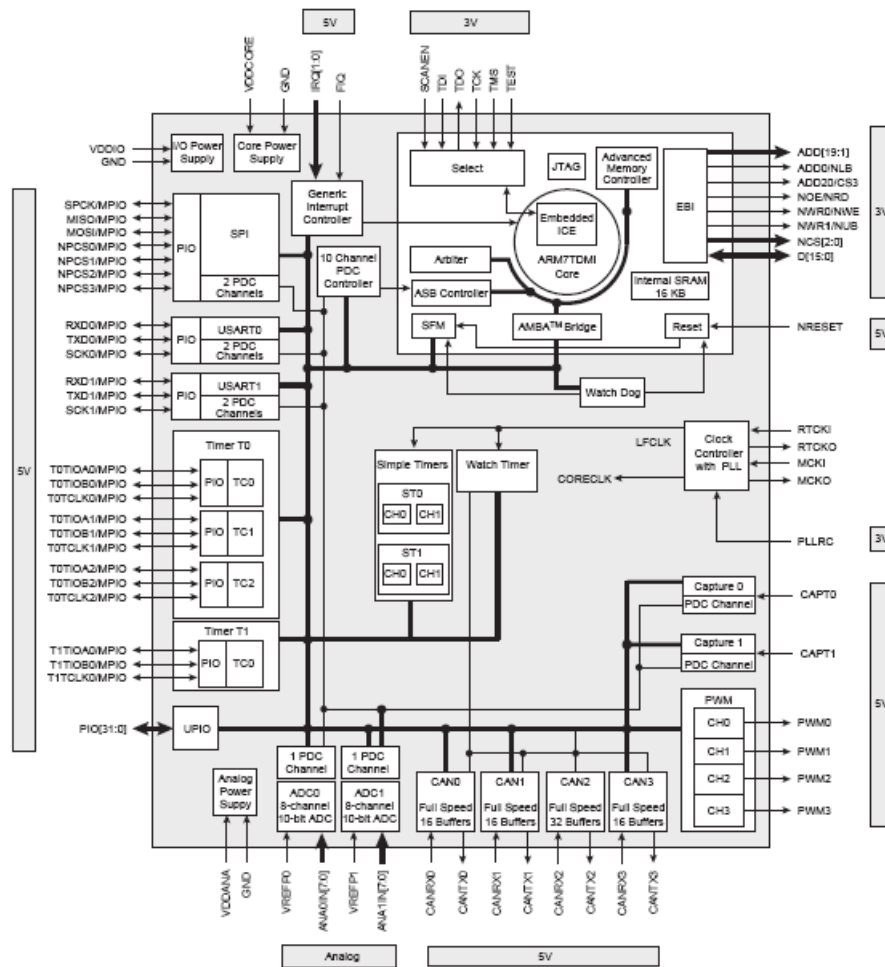


Figure 3.1: AT91SAM7A2 Block Diagram [3]

the highest performance. The Thumb instruction set is a subset of the most commonly used 32-bit ARM instructions. Thumb instructions are therefore only 16-bits long but have a corresponding 32-bit instruction that has the same effect. Thumb code is typically only 65% the size of full ARM code. This holds a significant advantage for a nanosatellite application where memory is an expensive commodity. It is also possible to switch between Thumb and full 32-bit ARM code in runtime, should the need arise.

3.1.1.1 Instruction Pipeline

The instruction pipeline consists of three stages, namely Fetch, Decode and Execute. This allows several operations to take place simultaneously.

3.1.1.2 Cache Memories

The AT91SAM7A2 has no cache memory - this fits our initial requirement as radiation in a Low Earth orbit can corrupt the data in the cache [5] and cause software malfunctions.

3.1.1.3 Memory Access

The processor has a Von Neumann architecture, which means that a single memory space is occupied by both instructions and data. Only load, store and swap instructions can access data from memory. Data can be either 8-, 16- or 32-bits wide, and must be aligned to their respective boundaries. 16 kBytes of internal SRAM are provided.

3.1.1.4 Reset Controller

A Reset input to the ARM7TDMI core is also provided which will cause the processor to restart execution from its boot address.

3.1.2 Advanced Memory Controller

The AT91SAM7A2 provides support for different types of memories connected to any of the three chip select lines. Wait states and the width of the data can be independantly configured for each of the lines. ROM, SRAM and NOR based Flash devices are easily supported by this interface, called the External Bus. This is the physical layer for connecting external memory devices to the processor. 20 address- and 16 data lines are provided. Due to the size of the EBI, only 6 MB of external memory is possible.

Protected mode access is also provided whereby a certain process that access memory outside its allowed scope causes an interrupt which can then be handled to close the misbehaving process, without causing any instability to the operating system. User and Supervisor modes are provided for these functions.

3.1.3 Clock Manager

The microcontroller provides a 32.768 kHz oscillator, a 2 to 6 MHz oscillator, a programmable PLL (2 to 20 times) and a programmable master clock divider. The clock management is done through the clock manager, and this allows the user to select between Low Power, Slow and Operational modes. Figure 3.2 shows how the different clocks on the microprocessor are constructed using the 6 MHz, and 32.768 kHz oscillators as reference. Details on the different power modes will be covered in detail in Section 3.2.1.

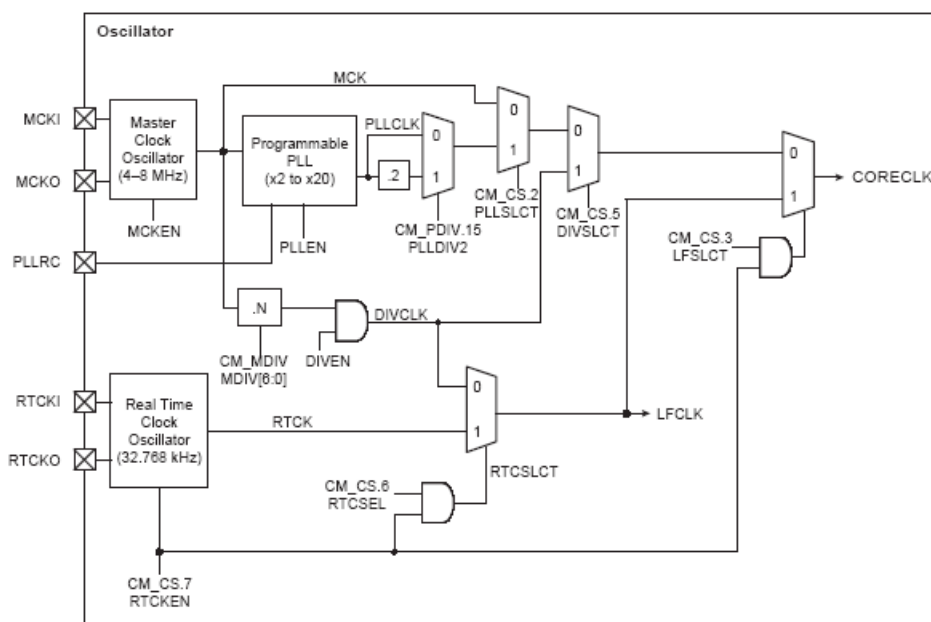


Figure 3.2: AT91SAM7A2 Clock Manager [3]

3.1.4 Peripherals

A Peripheral Data Controller is provided to facilitate DMA (Direct Memory Access) transfers from peripherals to memory, or from memory to peripherals. 10 DMA channels is provided for transfers to and from:

- 2 Capture Sources
- 2 Analog-to-Digital Converters

- 2 Serial Peripheral Interfaces
- 4 Universal Synchronous / Asynchronous Receivers / Transmitters

The following onboard peripherals are also provided:

- 4 Timer Channels
- A Watchdog Timer
- 4 Pulse Width Modulation Channels
- 4 Controller Area Network controllers
- A Generic Interrupt Controller
- 32 General Input / Output Pins

3.2 Power Consumption

A nanosatellite has to rely upon solar energy to satisfy all its power requirements and when the satellite goes into solar eclipse, batteries has to take over the task [15]. Solar panels has a limit on the amount of electricity that can be generated. Batteries also have a limited lifetime based on the number of charge and discharge cycles. Therefore power consumption has to be optimized in order for the satellite to be able to function in space for as long as possible.

A Major part of the selection criteria for a processor was that it should perform very well under these stringent power conditions. Processors used in battery powered handheld devices typically have low power consumption; primarily because of the very advanced power saving systems they employ.

3.2.1 Operating Modes

The clock manager is used to select between the different power modes of the processor: Operational, Slow and Low Power.

3.2.1.1 Operational Mode

During Operational Mode the master clock oscillator (MasterClock) and the PLL are enabled. The system clock is given by the following equation, CoreClock = $\alpha \times$ MasterClock (where α is between 2 and 20). Clearly the system clock can be adjusted as needed without rebooting the system. The PLL multiplier can be changed as the computational needs onboard the satellite fluctuates. The low frequency clock can be used for peripherals and can be selected as either RTCK (32.768 kHz) or MasterClock / β (where β is between 2 and 256). The typical power consumption of the microprocessor is shown in Table 3.1. Typical power consumption in this mode for the core is given by 900 μ A/MHz. Power consumption for the PLL is frequency independent and given as 4.95 mW.

Peripheral	Consumption [μ A/MHz]
Peripheral Data Controller	160
Unified Parallel Input Output	40
Universal Sync / Async Receiver / Transmitter	110
Serial Peripheral Interface	60
General Purpose Timer (3 Channels)	150
General Purpose Timer (1 Channel)	40
Analog to Digital Converter	20
CAN 16 Channels	210
CAN 32 Channels	280
Simple Timer	40
All Modules	1650

Table 3.1: Typical Current Consumption for the AT91SAM7A2 [3]

3.2.1.2 Slow Mode

In this mode, the PLL is deactivated and CoreClock = MasterClock / β . The low frequency clock can be used in exactly the same way as in Operational Mode. For the following configuration: $V_{DDCORE} = 3.3$ V, MasterClock = 4 MHz, $\beta = 256$ and CoreClock = 15.625 kHz the power consumption for the core is given as 3.76 mW.

3.2.1.3 Low Power Mode

The master clock oscillator, PLL and internal divider is switched off. Only the real time oscillator is enabled and both CoreClock and Low Frequency clock (for peripherals) is governed by this. In this mode, if all peripherals are disabled (by using the Power Management Block), the power consumption is only 792 μ W.

3.2.2 Power Management Block

In order to manage power consumption on an embedded device with as many peripherals as the AT91SAM7A2, a power management block is provided to switch the peripheral clocks on and off. The Power Management Block and Power Management controller operates completely independent of each other. When a device's peripheral clock (and/or PIO clock) is disabled, the clock on that device is immediately stopped. When the clock is re-enabled the peripheral controller resumes exactly where it left off. Displayed in Table 3.2 is a list of modules with powersave capabilities.

Module	Power Management Block Present
AMC (Advanced Memory Controller)	No
SFM (Special Function Mode)	No
Watchdog	No
Watch Timer	No
USART (Serial Interface)	Yes
CAN (Controller Area Network)	Yes
SPI (Serial Peripheral Interface)	Yes
ADC (Analog to Digital)	Yes
GPT (General Purpose Timer)	Yes
PWM (Pulse-Width Modulation)	Yes
UPIO (Unified Parallel IO)	Yes
CAPT (Capture Channels)	Yes
Simple Timer	Yes
CM (Clock Manager)	No
PMC (Power Management Controller)	Yes
PDC (Peripheral Data Controller)	No
GIC (General Interrupt Controller)	No

Table 3.2: Power Management Blocks for on chip peripherals [3]

3.2.3 Power Management Controller

Further optimization of power consumption can be done by completely disabling the ARM core clock. The current instruction is finished before the clock is disabled, and the clock can be enabled by any interrupt, or a hardware reset. This should be used with care onboard a satellite as the absence of any interrupts will cause the processor to remain in sleep mode indefinitely.

3.3 Memory Interface

The External Bus Interface (EBI) is the connection between off-chip memory and the Memory Controller of the ARM7. The controller allows for 6 MBytes of external address space. This is divided into three sections, selectable by chip selects zero through two. Each chip select line can be individually configured for different wait states, width, and byte or word access.

Once the AT91SAM7A2 reboots, the ARM core is in *reboot*-mode and the system starts executing code at address `0x00000000`. The device connected here should be 16-bits wide, and is by default configured with 8 wait states with Byte Write Access (BAT). This means that the NRD (Not Read) signal is used for reading, and two signals (NWR0 (Not Write 0) and NRW1) is used for writing. Thus, only 16-bit words can be read.

Only external memory on chip select zero is accessible in this mode, as shown in Table 3.3. Once the *remap* command is issued, all memory is available, and the revised memory map is shown in Table 3.4. The precise mapping of memory inside the *External Memories* block is done at boot time, and will be discussed in Section 5.2.1.1.

All the peripherals are also addressed using Memory Mapped I/O, and the respective addresses is shown in Table 3.5.

3.3.1 External Bus Interface Timings

Two types of read access cycles are possible on the EBI of the AT91SAM7A2: the standard read protocol, and the early read protocol. The latter increases EBI performance by allowing a faster timing on the EBI to be used.

Memory Space	Application	Abort
0xFFE00000 - 0xFFFFFFFF	Peripheral Devices.	No
0x00400000 - 0xFFDFFFFFF	Reserved.	Yes
0x00300000 - 0x003FFFFFF	Internal RAM 16 kBytes (repeated).	No
0x00200000 - 0x002FFFFFF	Reserved (Read as '0').	No
0x00100000 - 0x001FFFFFF	Reserved.	Yes
0x00000000 - 0x000FFFFFF	External Memory on CS0.	No

Table 3.3: Memory Map in Reboot Mode

Memory Space	Application	Abort
0xFFE00000 - 0xFFFFFFFF	Peripheral Devices.	No
0x80000000 - 0xFFDFFFFFF	Reserved.	Yes
0x40000000 - 0x7FFFFFFF	External Memories (up to 3).	Yes (when outside page)
0x00300000 - 0x3FFFFFF	Reserved.	Yes
0x00100000 - 0x002FFFFFF	Reserved (Read as '0').	No
0x00000000 - 0x000FFFFFF	Internal RAM 16 kBytes (repeated).	No

Table 3.4: Memory Map in Remap Mode

3.3.1.1 Standard Read Protocol

This is the default read protocol employed on the AT91SAM7A2 and it implements a read cycle where the Not Read (NRD/NOE) line is active during the second part of the read cycle. This allows enough time for the completion of any previous access and allows the address and chip select outputs to settle before attempting a new memory access. The respective NCS is set low with the appropriate address at the beginning of the cycle, with NRD only going low in the second half of the memory access as shown in Figure 3.3.

3.3.1.2 Early Read Protocol

A more sophisticated approach to reading data from the EBI involves setting NRD at the beginning of the read cycle. When continually reading from the same memory device, NRD remains active as shown in Figure 3.4. This allows more time for the memory device to obtain the required data (and also more time for our EDAC unit to check the data). An extra data float wait state is needed in some cases to avoid contention on the EBI. After a **read access**, a data float wait state gives more time for the external memory to release the

Peripheral	Address	IRQ
AMC	0xFFE00000	-
SFM	0xFFF00000	-
Watchdog	0xFFFA0000	2
Watch Timer	0xFFFA4000	3
USART0	0xFFFA8000	4
USART1	0xFFFAC000	5
CAN3 (16 Channels)	0xFFFB0000	6
SPI	0xFFFB4000	7
CAN1 (16 Channels)	0xFFFB8000	8
CAN2 (32 Channels)	0xFFBFC000	9
ADC0 (8 Channels, 10-bit)	0xFFFC0000	10
ADC1 (8 Channels, 10-bit)	0xFFFC4000	11
GPT0 (3 Channels)	0xFFFC8000	12, 13, 14
GPT1 (1 Channel)	0xFFCC0000	18
PWM	0xFFFD0000	19
CAN0 (16 Channels)	0xFFFD4000	20
UPIO (1 Channel)	0xFFFD8000	21
Capture CAPT0	0xFFDC0000	22
Capture CAPT1	0xFFE00000	23
Simple Timer ST0	0xFFE40000	24
Simple Timer ST1	0xFFE80000	25
Clock Manager	0xFFEC0000	-
PMC	0xFFFF4000	-
PDC	0xFFFF8000	-
GIC	0xFFFFF000	-

Table 3.5: Peripheral Memory Mapped I/O Addresses

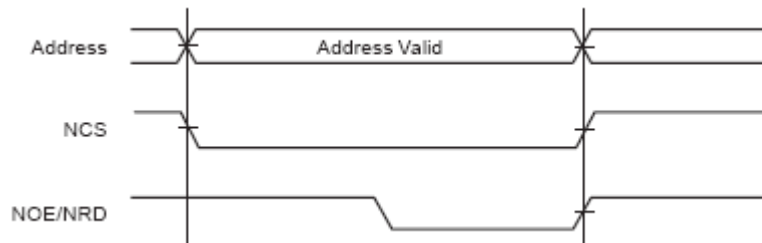


Figure 3.3: Standard Read Cycle

data bus. After a write access, a data float wait state will give more time to the EBI to release the data bus. The data float output time can be individually configured for each chip selectable memory device. Data Float wait states are asserted in between accesses. The wait state insertion depends strongly on the previous- and next access and whether it is/was a read or a write access on the same chip select line, or not. It is computed by looking at the Data Float Output Time (t_{DF}) of each external memory device as programmed into the AMC_CSR register.

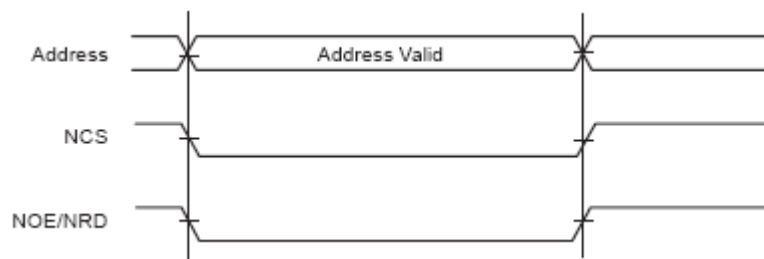


Figure 3.4: Early Read Cycle

3.3.1.3 Writing Protocol

Writing data to the external memory devices works in much the same way as the Standard Read Protocol. The address is first placed on the bus together with the data to be written. The respective NCS line is then pulled low, and remains low for the remainder of the cycle. After half a clock period, the NWE line is also set to a logic '0' after which the memory device clocks the data in. The write protocol is shown in Figure 3.5.

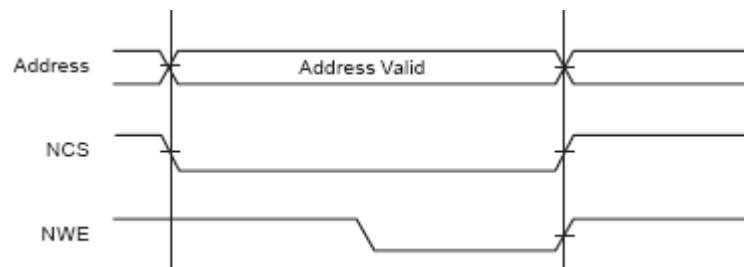


Figure 3.5: Write Cycle

3.3.1.4 Wait States

When using slower memory devices (such as FLASH) together with faster SRAM-based devices on the asynchronous data bus, the slower memory might not be able to keep up with the nominal bus speed. In this case, wait states can be inserted which effectively *slows down* access to these devices by prolonging a read- or write cycle. A write cycle with with two wait states included, is shown in Figure 3.6.

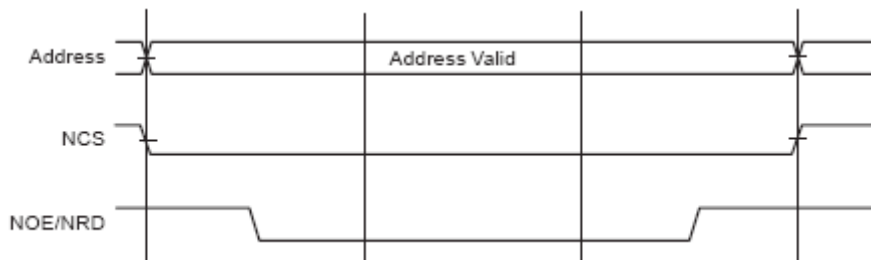


Figure 3.6: Write Cycle with 2 Wait States inserted

3.4 General Analog and Digital Interfaces

A fair number of general input / output pins are provided on the AT91SAM7A2. When all other peripherals are in use, 32 dedicated digital IO pins remain. These can be grouped into buses for providing Programmed IO (PIO) communication to other devices, or used individually to sense and respond to other devices onboard the nanosatellite. Chapter 4 will be dedicated to the design of a typical nanosatellite OBC and will show where such pins will be used.

Two 10-bit Analog to Digital converters, each with 8 independent channels, are also included on the processor. This allows the processor to directly interface to real world situations using sensors, since values encountered around us is very seldom discreet. This means that no additional hardware will be needed to convert data from analog sensors. Typically temperature, voltage and current sensors have analog outputs.

3.5 Communication Interfaces

A number of communication controllers are already provided on the AT91SAM7A2. This simplifies the hardware design of a satellite OBC tremendously, as a large part of the job of an OBC is to provide and coordinate communications with other subsystems onboard the satellite. Not all of the interfaces onchip are equally useful in a nanosatellite satellite system so some conversions are done to utilize the peripherals provided fully as well as provide adequate communication mediums for other subsystems.

3.5.1 Universal Synchronous / Asynchronous Receivers / Transmitters

The AT91SAM7A2 provides two USARTS. Each receive / transmit channel is individually connected to the Peripheral Data Controller for transfers directly to- and from memory without the need for processor intervention. The USARTS can operate in both Synchronous and Asynchronous modes and supports data framing which allows them to be directly connected to a modem for data downlink. In data framing mode the start character(s) are automatically detected in hardware which frees up the software from performing this time consuming task.

3.5.2 CAN Controllers

The Controller Area Network is a serial communications protocol that supports a bus architecture with distributed real-time control with a very high level of security. Communications with CAN is possible at rates up to 1 Mbit/s. Adhering to the transparency standards set out in the ISO/OSI [11] reference model, the CAN protocol has been subdivided into two different layers, namely the Data Link Layer and the Physical Layer. The Data Link Layer is responsible for the Logical Link Control, and Medium Access Control sublayers as shown in Figure 3.7.

3.5.2.1 Logical Link Control (LLC)

The tasks of the LLC sublayer is to determine which messages received are to be accepted, providing services for data transfer and remote data requests

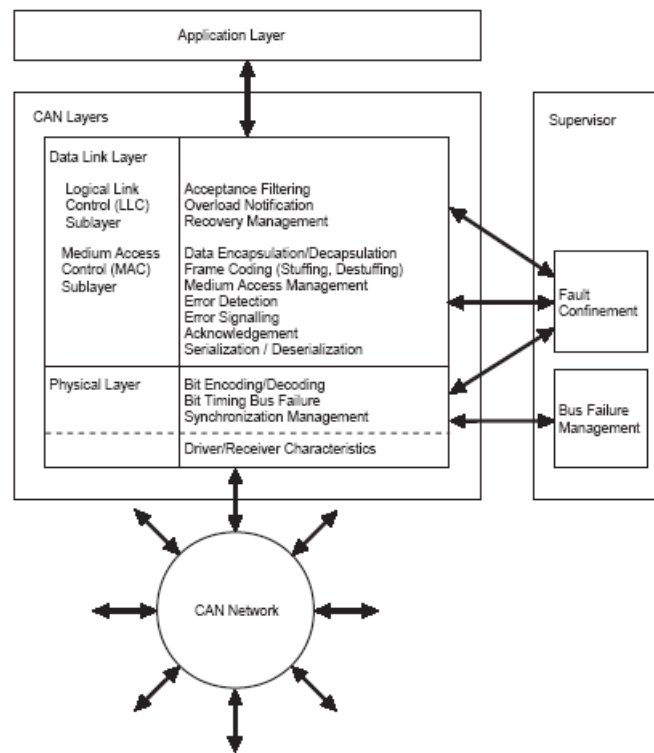


Figure 3.7: ISO/OSI Representation of a CAN Node [3]

and to provide the means for error handling and management in the event of overloading.

3.5.2.2 Media Access Control (MAC)

The MAC layer is primarily responsible for the transfer protocol: controlling framing, performing arbitration and error checking. The MAC layer is responsible for determining if the bus is free to start transmitting or if another transmission has just started and it is necessary to abort the current message. This is done by using a scheme called bitwise arbitration, whereby each CAN message is identified using a 11 (or 29) bit identifier.

Data transmission on the line is performed by sending data, NRZ (Non-return to Zero) encoded, over the CAN bus. A dominant bit is sent on the line by setting $CANH = 2.5\text{ V}$ and $CANL = 0.5\text{ V}$. A recessive bit is created by setting $CANH = 2.3\text{ V}$ and $CANL = 2.3\text{ V}$.

When a node sends a message, it also monitors the line to see if the message that it is sending is indeed the one on the line. As soon as a difference between the message that is sent and the state of the bus is detected, the node backs off and assumes another station with a lower identifier is transmitting. This ensures that only the station with the lowest identifier will have access to the bus, and the other stations back off for a certain interval and then tries again.

This scheme insures that the network will be collision free and the packet with the lowest identifier will always get through undisturbed. This makes CAN very suitable for real time applications since prioritizing of packets are inherently supported with the lowest-identifier-always-wins scheme. The arrival time for a high priority (low identifier) message is therefore always bound.

A typical CAN data frame is shown in Figure 3.8.

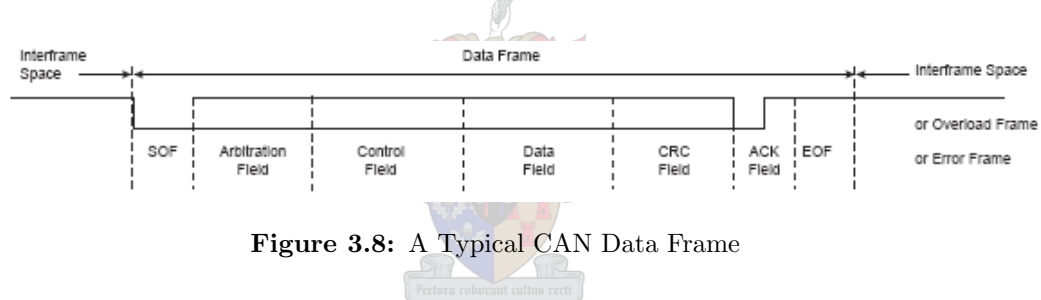


Figure 3.8: A Typical CAN Data Frame

3.5.3 Serial Peripheral Interface

The SPI Interface is a general serial bus which can operate in both master, and slave modes. The only real difference between the two is that the AT91SAM7A2 **supplies** the clock in master mode, and **accepts** it in slave mode. Due to the nature of the design of the Peripheral Data Controller, direct-to-memory transfers are only possible when using the SPI bus in master mode. The interface consists of a Data Input, Data Output, Clock and four SPI select lines. It is therefore possible to have up to 4 devices on the SPI bus, and selecting between them by using the NPCS (Not Peripheral Chip Select) lines. In master mode the clock of the SPI bus can be set in software, up to the maximum of CORECLK (30 MHz). The clock is only running while

transmitting (or receiving) data, and the size of the transceived data is always 16 bits.

3.6 Reliability

With the design of a LEO nanosatellite using commercial-off-the-shelf-components two opposing forces come into play. The first, is obviously the need for reliability: Even when building a satellite from commercial components, it is still a very costly exercise and the risk for failure should be kept to an absolute minimum. The second important factor is that of cost: Using space-approved radiation hardened components will almost certainly improve our reliability, but well exceeds the size of our available budget. The next section will highlight some of problems we might encounter in space with possible ways of solving them.

3.6.1 Temperature- and radiation tolerance

External parts of a satellite can typically vary between -263°C to over 100°C [7]. Finding components that will survive this extremes are practically impossible, so other ways of passive temperature control are employed on nanosatellites. This involves painting the satellite with thermal insulating materials and also spinning the satellite around one axis to keep the temperatures inside more stable. Temperatures of between -15°C to 45°C can be obtained by using these methods which is certainly a bit more hospitable to electronic components [7].

Commercial components normally have temperature specifications in the range of 0°C to 70°C , with their industrial counterparts specified at -40°C to 85°C . The typical price difference between a commercial and industrial component is roughly double, so, depending on the availability, it could be a worthwhile investment.

In a Low Earth Orbit there is significantly more radiation activity than experienced on the earth's surface, primarily due to the particles trapped in the Van Allen belts. The Van Allen belt is a torus of electrical charged particles around the earth, held in place by the earth's magnetic field. The

effect of radiation on electrical components (such as integrated circuits) are categorized in the following sections.

3.6.1.1 Single Event Latchup (SEL)

This happens when a semiconductor device no longer responds to its input signals. Excessive current flow may be a result of a SEL and if it was not damaged by this, it may be restored to a functional state by power cycling.

3.6.1.2 Single Event Upset (SEU)

This is categorized by an unwanted change in state inside a memory device. Normally this is regarded as a ‘soft error’ and may be corrected by using an error detection and correction (EDAC) scheme. SEUs cause no damage to a device.

3.6.1.3 Single Event Effects (SEE)

SELs and SEUs are both examples of SEEs and the rate at which SEEs occur is used to measure the sensitivity of a device to radiation effects.

A software watchdog is a piece of hardware that is continually polled by a software application. As soon as a SEE occurs which results in a program malfunction that impairs the working of the satellite, the software does not poll the watchdog anymore, which causes a complete system reboot. This is generally enough to temporarily get rid of the effects of SEUs. Power cycling is the only effective way to get rid of SELs.

3.6.2 History of Space Use

Unfortunately since there is so much different ARM7 based microprocessors on the market, the chances of our specific one being used onboard another satellite is extremely slim. The CanX-1 nanosatellite from the University of Toronto was launched in June 2003, and utilized an ARM7TDMI processor. Unfortunately more information on the precise model being used could not be obtained from them, and even on the image of their OBC the name of the processor is blurred. Unfortunately the CanX-1 never answered from space, but the CanX-2 (to be launched in June 2007) also employs an (presumably) Atmel ARM7 processor.

Chapter 4

Detailed Design of the OBC

To successfully evaluate the performance of the AT91SAM7A2 in a nanosatellite environment, a complete onboard computer system based on this processor was developed. The design of the board outlined in this chapter aims to provide support for all the standard devices typically operated onboard a very small satellite. Because of this, it should be usable as an OBC for future nanosatellite missions with very little (if any) changes to the design. Unlike previous OBC projects in the ESL (Electronic System Laboratory) at Stellenbosch University, this is **not** a development board but rather a prototype for a general purpose OBC and adheres to the mechanical and electrical specifications given by SunSpace for peripherals used onboard their satellites.

4.1 Design Overview

The first step in the design of an OBC is determining what services should be provided by the OBC - this is normally given in the functional specification. In our case a very general functional specification is given, since we need to provide support for nanosatellites with a wide range of missions:

- Provide a low-speed redundant communications bus for information exchange between peripherals.
- Provide a high-speed communications interface for communication to the payload.
- General Housekeeping tasks need to be performed.

- Provide the ability to run user processes (AODCS, Image processing, etc.).

A Block diagram of the complete OBC design (Figure 4.1) and a diagram depicting the layout of the PC board (Figure A.1 and Figure A.2) is shown. Each subsystem is described in detail in the sections that follow.

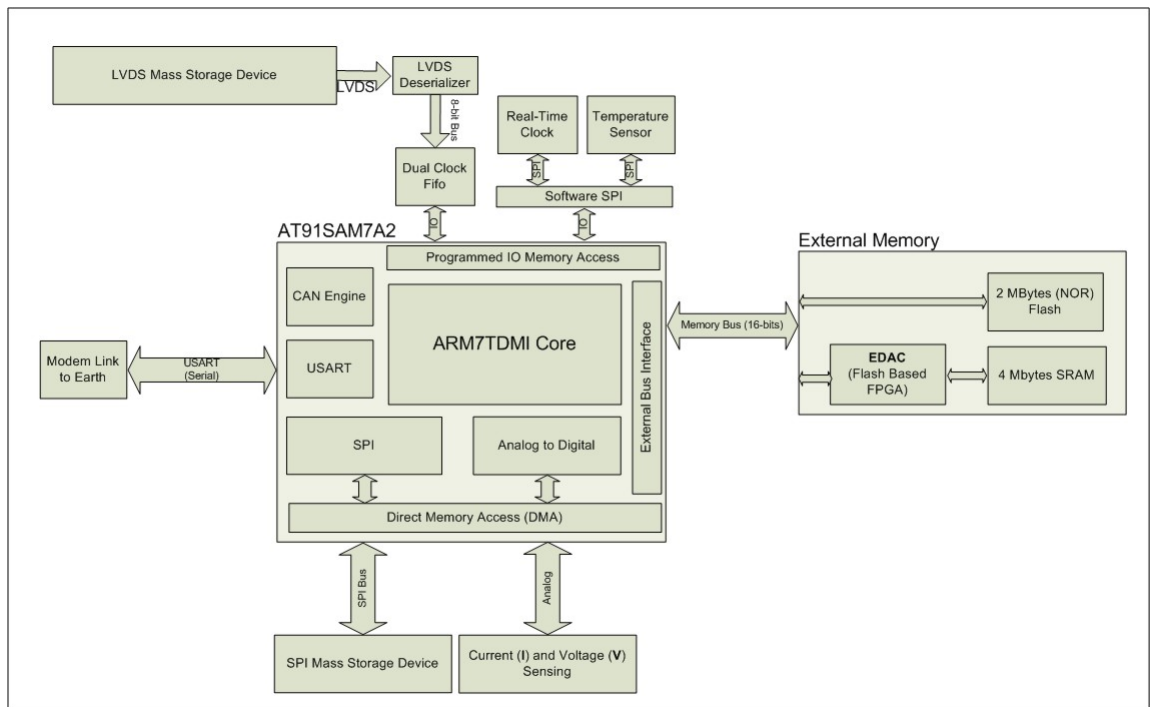


Figure 4.1: Block Diagram of different OBC subsystems

4.2 Physical Printed Circuit Board

To be able to integrate with other parts of a nanosatellite, certain size and volume specifications must be adhered to. At the start of the project, detailed design information was obtained from *SunSpace* in their *PCB Mechanical Development Specifications* [1] document. A brief overview of the specifications is summarized in Table 4.1.

Item	Constraint
PC Board Material	FR4
Number of layers	4 or more
Thickness	≥ 1.6 mm
Copper Thickness	≥ 35 μm
PC Board Dimensions	EuroCard 6U (160 mm)

Table 4.1: Mechanical PC Board Specifications

4.3 Power Supply

For the correct operation of the microprocessors and other supporting hardware onboard our OBC a very stable power supply was needed. Since the bus voltage onboard a small satellite can vary from anything between 14 and 28 Volts DC, support needs to be provided for this range of voltages.

Since different devices with different voltage requirements are used on board our OBC, support has to be provided for both 2.5 V and 3.3 V devices. The Actel APA075 FPGA requires 2.5 V for its core, while the AT91SAM7A2 and all other devices requires 3.3 V. The FPGA draws a maximum of 10 mW from the 2.5 V source, while the calculation for the 3.3 V source is shown in Equation 4.3.1.



$$W_{3.3v} + W_{2.5v} = W_{total} \quad (4.3.1)$$

$$158.4mW + 12.5mW = 170.9mW$$

Therefore a 500 mA converter for each of the power supplies (3.3 V and 2.5 V) is more than capable of providing the necessary current for the operation of our OBC, as shown in Equation 4.3.2.

$$W_{3.3v} = 158.4mW = 3.3V \times 48 \text{ mA} \quad (4.3.2)$$

$$W_{2.5v} = 12.5mW = 2.5V \times 5 \text{ mA}$$

Linear regulators are normally the easiest to use, but since their efficiency is very low compared to switch mode regulators, the latter will be used. Power

is limited onboard a satellite and unnecessary heating caused by linear regulators (which is a severe problem because of the lack of cooling by convection) makes them a very poor choice.

The R-78XX Series by Recom is a switch mode regulator with a very high efficiency and requires no external components, except for a filter capacitor on the output. To monitor the voltage output and current consumption of the 2.5 V and the 3.3 V supplies, 4 of the Analog to Digital channels on the AT91SAM7A2 are used as shown in Figure 4.2 and Figure 4.3.

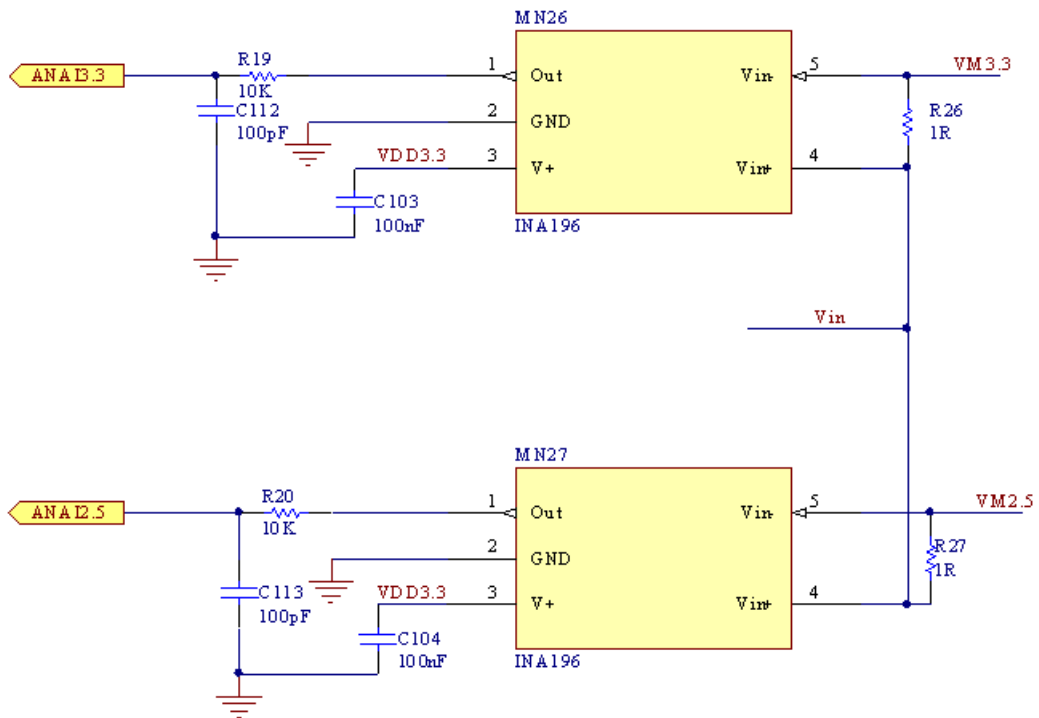


Figure 4.2: Current Monitoring of PCB

The INA196 from Texas Instruments is used to convert the current through the 1 Ω resistor to a voltage, which is then fed to one of the A/D pins on the microprocessor. The 1 Ω is chosen because it provides for enough resolution once the A/D conversion is complete, and a very small amount of power is dissipated in the device because of its low resistance. The INA196 is on the

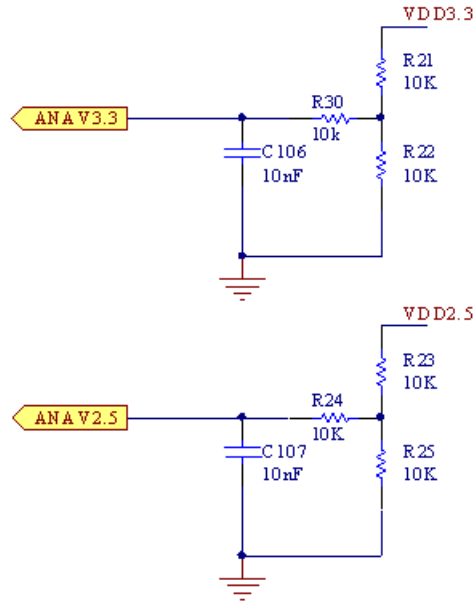


Figure 4.3: Voltage Monitoring of PCB

high-voltage (input) side of the regulators so that the voltage drop across them does not affect the 2.5- and 3.3 voltages used by the very sensitive microprocessors.

The voltages are divided by using a resistor voltage divider network, and connected to one of the A/D converter pins. A current limiting resistor is also used in each instance for protection of the A/D input. The value for this resistor can easily be determined by considering the leakage current of the AT91SAM7A2 Analog to Digital input pads.

According to the datasheet [3], the input leakage current is 90 nA. This means that with a 10 k Ω resistor, the voltage drop will be 900 μ V. The 10-bit A/D converter on the AT91SAM7A2's smallest voltage step is given by Equation 4.3.3. This means that a 10 k Ω resistor will make no noticeable difference to the voltage discretized. If the pad of the converter should fail and short to ground, only about 1 mW of energy would be wasted; which is acceptable.

$$3.3V/2^{10} = 3.2mV \quad (4.3.3)$$

4.4 Communications Interfaces

The AT91SAM7A2 microprocessor has a plethora of on-chip communications peripherals already built-in which eliminates the need for external controllers. All communications will be provided by the microprocessor, with the exception of the LVDS data link which will need to be managed externally.

4.4.1 CAN Interface

The microprocessor provides four on-chip CAN channels. Two of these channels are used to provide primary and redundant communications bus on the satellite. Messages destined for any number of the subsystems onboard the nanosatellite will be transmitted using this data path.

The two CAN channels are identical and each consists of 16 *mailboxes* where messages to/from other subsystems can be held until ready to be processed by the running operating system. Each channel consists of a TTL-level Send- and Receive pin-pair on the microprocessor. This has to be converted to the differential signalling system used by CAN-devices. To fulfil this requirement, an SN65HVD230 3.3 V CAN Transceiver from Texas Instruments is used. It supports communication speeds of up to 1 Mbps and has a *sleep* pin which, when activated, can lower current consumption to 40 nA.

4.4.2 SPI Bus

A Serial Peripheral Interface is also available on the microprocessor. This bus consists of a Master-out-Slave-In (MOSI), Master-in-Slave-out (MISO), Clock and four chip select lines. The processor can operate the SPI bus in either Slave or Master mode.

In Slave mode the driving of the chip select and clock lines are not done by the microprocessor. A big drawback is that Direct Memory Access (DMA) is not supported in this mode, and software will have to do all the transferring from data from/to the bus.

Master mode is the preferred mode of operation: The microprocessor controls the chip select lines and can individually speak to up to four devices

connected to the same bus. The clock is generated by the microprocessor itself, and speeds of up to 30 Mbps is possible. DMA access is supported, which makes this the preferred way of moving large data blocks in- and out of the memory without having to commit all the processing resources to the task. In a nanosatellite implementation, the SPI bus is ideally suited for an imager or mass storage device where large amounts of data needs to be transceived without impacting the performance of the OBC.

The SPI bus is also suited for the transmission of data between other peripherals directly related to the OBC, such as an external Real Time Clock and Digital Temperature Sensor. Unfortunately the transmission speeds of these devices is much lower than the speed at which we would like to run the SPI bus and we would also like to keep the bus available for imaging and data storage modules that transmits large amounts of data at a time. Because of this, a decision was made to rather implement the SPI communication for these low-speed SPI devices in software since DMA is not a requirement (because of the small, infrequent amounts of data being sent), and it will free up the high-speed hardware bus for modules that really need it.

4.4.3 Serial Interface

The microprocessor has two Universal Synchronous / Asynchronous Receiver / Transmitters (USARTs). They will be configured in Asynchronous mode and provide communication to one or two MODEMs (Modulator/Demodulator) for down- or uplink to earth. The communication speed can be changed from 300 bps through to 12 Mbps.

Because the wiring distance from the OBC to the modems will typically be less than a metre we decided not to use a RS232 transceiver, but rather keep the voltage levels as TTL. Interference and cable resistance should not be a problem, and RS232 transceivers (such as the common MAX232) uses charge pumps to convert TTL levels to RS232 which will just result in a higher power consumption in a system that cannot really afford it.

4.4.4 RS-485 Data Line

A single differential RS-485 type signal needs to be sent from the OBC each second for time synchronization between subsystems onboard the satellite. One I/O pin on the processor is used for this purpose, which is in turn connected to a SN65HVD32 RS-485 transceiver. Only the transmit pin on the transceiver will be used since this bus will only be used for this specific time-keeping application.

4.4.5 Low Voltage Differential Signalling (LVDS) Interface

A common way of sending large amounts of data to an OBC is to use a LVDS interface. The primary way of transceiving data on this OBC is to use the high speed SPI bus described earlier, but an LVDS interface is also provided for compatibility with other, already developed, satellite peripherals.

LVDS uses a two-wire differential signalling system which is very good at rejecting interference even at high frequencies. Typically a normal serial data transmission is not just *level converted* with an LVDS transceiver (as is the case with RS232, RS485 and CAN) to LVDS levels.

In standard LVDS satellite peripherals (such as these designed by Sun-Space), a LVDS Serializer (DS92LV1021) and Deserializer (DS92LV1212A) is used to pack the parallel data outputs into a LVDS datastream. A parallel interface with a 10-bit output running at 1 MHz is thus converted to a serial 12 MHz LVDS signal (two extra bits, start and stop, are added to the data as padding), which is transmitted over a LVDS link, deserialized, and again produced as a 1 MHz 10-bit data byte.

Together with the 10-bit data input to the serializer, a clock also needs to be provided. In order for the deserializer to lock, this same clock needs to be available to it as well. For this, the clock is **also** transmitted via its own LVDS data path, using a standard LVDS signal level converter (transmitter).

When designing a LVDS data link to send data from an imager to the OBC another problem becomes apparent: If the LVDS device sending the data (Mass Storage Device or Imager) is sending data to the OBC and also providing a clock, data might be lost if the OBC gets held up in other tasks

with a higher priority and does not have time to service the incoming data.

For this reason a 8 kB Dual Clock FIFO (first in first out) buffer (IDT72V251L15PFI) is placed between the deserializer and the Atmel microcontroller. Data is written into the FIFO by the deserializer and read from it by the micro. Both the microcontroller and the deserializer supplies its own separate clock.

If the OBC and the remote device sending the data is not precisely synchronized, a buffer overflow will occur at some point and some of the data will be lost. To counter this, another single LVDS link is set up, but this time in the opposite direction of the clock and data flow. This line will alert the remote transmitting device that the FIFO is almost full, and it must temporarily stop sending data. The FIFO used in the design has a flag that is set as soon as the FIFO is *almost full* which can be converted to a LVDS signal easily by using a LVDS transmitter (MAX9157). The complete LVDS data link is shown in Figure 4.4.

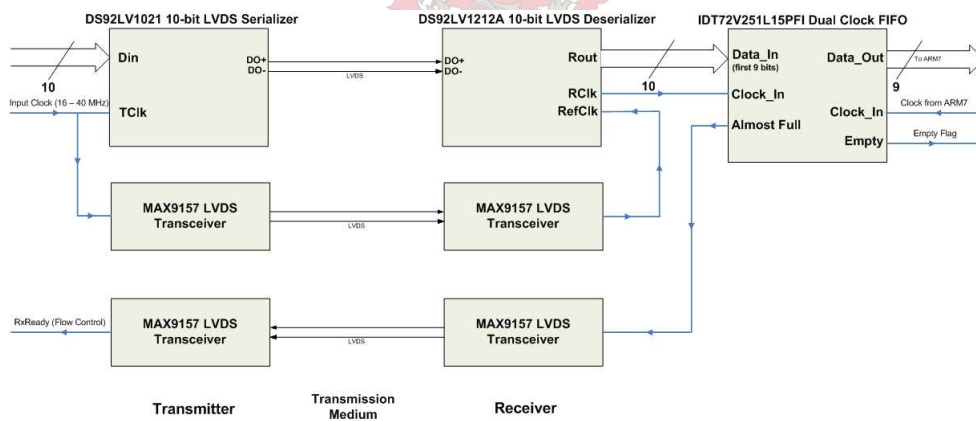


Figure 4.4: Nanosatellite LVDS Data Link

4.5 Actel ProAsic Plus FPGA

A FPGA was added to the design primarily for providing the necessary support for a transparent VHDL [16] based Error Detection and Correction (EDAC) [8] system for asynchronous SRAM. Commercial EDACs are available, but

they are expensive, and no changes can be made to the algorithm they are using.

The goal was to design a *black box* type EDAC that can be reused in other systems using asynchronous memories susceptible to corruption. Ideally you should be able to place it on your data bus without the microprocessor or the memory being aware of its existence. A secondary use for the FPGA was also to provide memory decoding functions for all the memories onboard the OBC.

The Actel ProAsic Plus FPGA was primarily chosen because of the fact that it uses Flash Based memory cells to store its configuration. This type of memory is immune to particle effects caused by radiation. FPGA's like the Altera family uses SRAM, which is in itself susceptible to radiation corruption. A development board for this FPGA was already available at the time, and the algorithm developed performed excellent on this platform. The Actel ProAsic Plus is a fairly old and proven FPGA and provides more than enough speed at its maximum clock rate of 180 MHz.

4.5.1 Error Detection and Correction

The most common used way of detecting and correcting memory errors on the fly, is to use a slightly adapted version of Hamming code. The foremost reason for this being that check bit generation and error checking needs to be performed in real time, and a computationally expensive code will result in the memory running at a lower speed than is ideally possible.

Standard Hamming codes are capable of correcting a single bit error or detecting two bit errors, but not capable of doing both simultaneously, as it cannot distinguish between the two [12]. You may choose to use Hamming codes as an error detection mechanism to catch both single and double bit errors or to correct single bit error. This is accomplished by using more than one parity bit, each computed on different combination of bits in the data. By using a slightly adapted Hamming code, it is possible to extend the algorithm so error detection and correction can be done simultaneously.

Hamming code generates parity bits for certain groups of memory bits. The number of parity bits needed for a given amount of memory-bits, is given

by Equation 4.5.1.

$$d + p + 1 \leq 2^p \quad (4.5.1)$$

Where d is the number of data bits and p is the number of parity bits. The result of appending the computed parity bits to the data bits is called the Hamming code word. The size of the code word $c = (d + p)$ and a Hamming code word is described by the ordered set (c, d) .

A Hamming code word is generated by multiplying the data bits by a generator matrix \mathbf{G} using modulo-2 arithmetic. Modulo-2 arithmetic is performed digit by digit on binary numbers. Each digit is considered independently from its neighbours: numbers are not carried or borrowed.

This modulo-2 multiplication's result is called the code word vector (c_0, c_1, \dots, c_n) , consisting of the original data bits and the calculated parity bits [8].

The generator matrix \mathbf{G} used in constructing Hamming codes consists of \mathbf{I} (the identity matrix) and a parity generation matrix \mathbf{A} (Equation 4.5.2).

$$\mathbf{G} = [\mathbf{I}|\mathbf{A}] \quad (4.5.2)$$

Evaluating Equation 4.5.1 for our case where we have a 16-bit data bus we have $p = 5$. This means that we need 5 extra parity bits per 16 data bits for the equation to hold. The smallest SRAM package available to us has a data bus width of 8. It is important to try to keep the address bus mapping at a one to one relationship between the 16 data bits and 8 parity bits to avoid extra computations which will slow down the memory system. By utilizing these extra 3 bits available inside the 8-bit wide SRAM blocks, we will be able to extend our error detection capabilities quite dramatically.

The NASA Office of Logic Design [9] supplies a very good Hamming-based EDAC software library which has been developed at ESA and the University of Surrey. Although not completely applicable to this design, the parity generation matrices can be used to obtain a real time *flow-through* EDAC system which will be able to detect and correct all single bit errors, and detect up to 8 bit errors in a 16-bit data word. This system will add 8 bits of parity overhead to each 16-bit data word.

$$A = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \quad (4.5.3)$$

The data and parity, \mathbf{m} , stored in the memory is made up out of the data bits (\mathbf{d}) and parity bits (\mathbf{p}) and is given by Equation 4.5.4.

$$\mathbf{m} = \mathbf{d} \times \mathbf{G} \quad (4.5.4)$$

All the columns of \mathbf{A} are selected so each column is unique. The parity bits (p_0, p_2, \dots, p_n) represents parity calculations of eight distinct subsets of \mathbf{d} . Validating the received code word \mathbf{r} , involves multiplying it by a parity check to form \mathbf{s} , the syndrome or parity check vector.

$$\mathbf{H} = [A^T | \mathbf{I}] \quad (4.5.5)$$

$$\mathbf{s} = \mathbf{H} \times \mathbf{r} \quad (4.5.6)$$

If all elements of \mathbf{s} are zero, the code word was received correctly. If \mathbf{s} contains non-zero elements, the bit in error can be determined by analyzing which parity checks have failed, as long as the error involves only a single bit.

For instance if $\mathbf{s} = [00110101]$, that syndrome matches to the fourth column

in \mathbf{H} that corresponds to the fourth bit of \mathbf{r} : the bit in error.

A block diagram of the EDAC system implemented in VHDL is shown in Figure 4.5. The complete FPGA pinouts and VHDL code is shown in Addendum G.

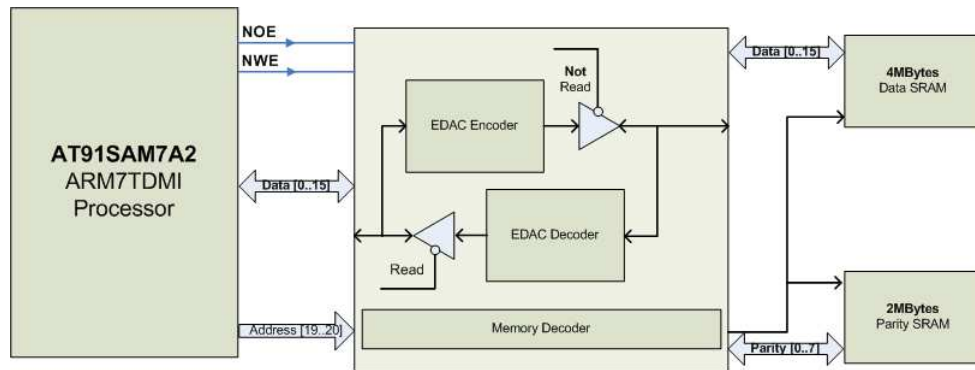


Figure 4.5: EDAC Block Diagram

4.5.2 Memory Decoding

The AT91SAM7A2 has support for 3 different banks of memory, 2 MBytes each. The NOR-based flash memory used in this design is easily available in a 2 MByte package, but the largest Asynchronous SRAM module is only 512 kBytes. While addressing the 16-bit wide SRAM data modules, 8 bits of parity data needs to be saved also. For this reason, the following two modules were chosen:

- **Data Store** : 256 kBytes \times 16-bits = 512 kBytes
- **Parity Store** : 512 kBytes \times 8-bits = 512 kBytes

A one-to-one address mapping between the data and parity devices is now possible, which means that the EDAC does not have to be aware of the addresses that data is written to. Because of the fact that we need 4 SRAM data modules and 2 SRAM parity modules per Chip Select line on the microprocessor, some form of memory decoding is necessary.

The 21 address lines which are available ($2^{21} = 2$ MBytes of memory space) are connected to the SRAM data memory devices:

- A_0 : Not Lower Byte (NLB)
- $A_1 - A_{18}$: Data Lines on 256 kByte module
- $A_{19} - A_{20}$: Used as Chip Select Lines (via FPGA)

and also to the SRAM parity devices:

- A_0 : Not Lower Byte (NLB)
- $A_1 - A_{19}$: Data Lines on 512 kByte module
- A_{20} : Used as Chip Select Lines (via FPGA)

The VHDL implementation of the memory decoder for the OBC is shown below:

```
ARCHITECTURE Decoder OF MemDec IS
BEGIN
```

```

    ncs0 <= not(not(address(20)) and not(address(19)) and not(ncs1_in));
    ncs1 <= not(not(address(20)) and (address(19)) and not(ncs1_in));
    ncs2 <= not((address(20)) and not(address(19)) and not(ncs1_in));
    ncs3 <= not((address(20)) and (address(19)) and not(ncs1_in));

    npcs0 <= ncs0 and ncs1;
    npcs1 <= ncs2 and ncs3;

    ncs4 <= not(not(address(20)) and not(address(19)) and not(ncs2_in));
    ncs5 <= not(not(address(20)) and (address(19)) and not(ncs2_in));
    ncs6 <= not((address(20)) and not(address(19)) and not(ncs2_in));
    ncs7 <= not((address(20)) and (address(19)) and not(ncs2_in));

    npcs2 <= ncs4 and ncs5;
    npcs3 <= ncs6 and ncs7;

END Decoder;
```

4.6 Memory System

A nanosatellite typically needs three different random access memory storage *banks*. The scheme that was used to realize the three different banks onboard this OBC, is shown in Figure 4.6.

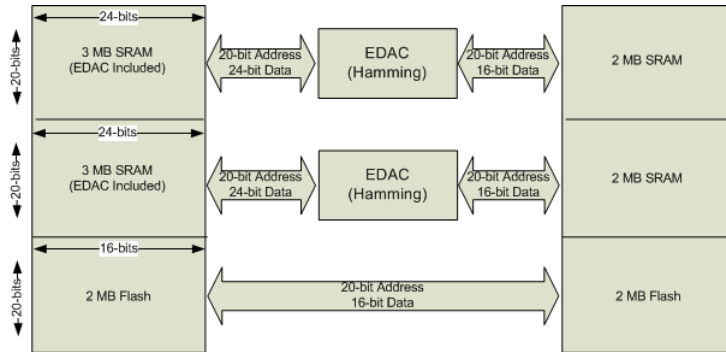


Figure 4.6: OBC Memory System

4.6.1 Flash

The first bank needs to contain non-volatile memory with the boot-code and operating system and all the initial parameters required for a cold system startup. NOR type FLASH memory is chosen on the nanosatellite board, which provides random read access to any of the 16-bit wide blocks. Reading from these devices is typically very fast (10 MHz) but the writing of data is done by using the CFI and speeds are about ten times slower. The OBC provides 2 MBytes of NOR flash memory using an Atmel AT49BV163D.

4.6.2 SRAM

SRAM memory provides very fast random access in either reading or writing modes. In a typical nanosatellite environment, the operating system kernel and any other processes will be copied from flash memory into SRAM where it will reside during normal operation. Random access reading and writing at high speeds (typically 30 MHz for this design) will be possible.

The third bank of memory also consists of SRAM, and will primarily be used for storing temporary data being processed by the OBC. An example of this is images transferred from the imager directly into memory via SPI. From here, compression algorithms can be run and the data sent to earth via a modem downlink.

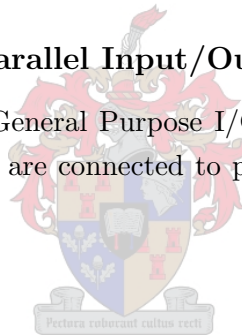
The OBC has 2 banks of 2 Mbyte EDAC protected SRAM. This is provided by 8 x 256 kBytes of SRAM memory with a data bus width of 16-bits.

4.7 Miscellaneous

Some other application specific devices not directly supported by the ATM91SAM7A2 processor is also needed for correct operation of the nanosatellite OBC. The detail of how the microprocessor will interact with these devices is now discussed.

4.7.1 ARM Unified Parallel Input/Output Interface

The AT91SAM7A2 has 32 General Purpose I/O pins which can all be controlled independantly. They are connected to peripherals as shown in Table 4.2.



4.7.2 Real Time Clock

Certain processes aboard a satellite needs to be automated and scheduled to perform at specific times [14]. Rotation of a camera to earth for imaging at a specific time is just one of these events. Even if the microprocessor onboard the satellite needs to reboot due to a software or temporary hardware error the time information has to stay current. For this we use the M41ST95W Real Time Clock.

The RTC has a SPI interface to the AT91SAM7A2 which is implemented in software on the microprocessor. Initially the correct date and time information can be uploaded to the OBC from a groundstation on earth. From here on, the RTC will keep the correct time even in case of a complete system restart. The RTC also has an additional (optional) connector for a backup battery in case the primary batteries onboard the satellite completely discharges.

UPIO Pin(s)	Peripheral Attached
0	EDAC Disable
1..2	EDAC Error[0..1]
3..4	CAN[0..1] Powersave
5	TimeSync (RS485 Pulse)
6	Connected to FPGA
7	LVDS Rx Not Locked
8	LVDS Tx Clock
9	LVDS Tx Data
10	Real Time Clock Rx
11	Real Time Clock Tx
12	Real Time Clock Clock
13	Real Time Clock Enable
14	Temperature Rx
15	Temperature Clock
16	Temperature Enabled
17..24	LVDS RX[0..7]
25	PowerSave LVDS RX Deserializer
26	LVDS RX Clock (Output)
27	LVDS RX Read Enable (Output)
28	LVDS RX Dual FIFO Not Empty (Input)
29	LVDS RX Not Reset FIFO (Output)
30	LVDS RX Not Write Enable (Output)
31	CORECLK (Connected to FPGA)

Table 4.2: ARM7 UPIO Connections

Pectora roburant cultus recti

4.7.3 Digital Temperature Sensing

To keep our satellite component costs down, our OBC makes use of COTS components. Most of these components have commercial or industrial environmental operating ratings (temperature, humidity) which, when exceeded, could damage the device permanently. Therefore it is imperative that these conditions be continuously monitored and not exceeded.

Temperature monitoring is done by the LM70 from National Semiconductor. It is a SPI digital temperature sensor connected to the AT91SAM7A2 via a software SPI link. When a temperature surge is detected the microprocessor can be directed to go into power save mode, dissipating less power and thus generating less heat.

4.7.4 Current- and Voltage Measurements

In addition to the environmental operating ratings, all the devices onboard the OBC has specific voltage and current requirements. A malfunctioning DC-to-DC converter could have disastrous effects and cause the destruction of components on the board.

Also, a microprocessor suddenly consuming exorbitant amounts of power could be the result of a Single Event Latchup. It may be possible to recover from this state by simply tripping and restoring power to the affected subsystem.

Voltage monitoring on the board is done by sampling the 3.3 V and 2.5 V outputs of the DC-to-DC converters. The converter outputs are first divided exactly in two and then fed to the A/D input of the microprocessor. A 10 k Ω resistor is also added to protect the power supply circuit in case the I/O input of the microprocessor breaks down and shorts to earth.

Current monitoring is done in much the same way: The current is first converted into a voltage by measuring the voltage drop across a 1 Ω resistor. The INA196 from Texas Instruments is used to measure this voltage drop and amplify the resulting difference. This is done **before** the DC-to-DC converters (using the raw bus input voltage) to ensure that the resulting voltage drop from the resistor does not influence the 2.5 V and 3.3 V supply voltages. The resulting amplified difference from the INA196 is then connected to a A/D input on the microprocessor for sampling.

Chapter 5

Software

The development of hardware is rarely done without providing some software support as well. The successful completion of a project is normally measured by software tests designed to check every part of the design. Good debugging tools supporting your hardware is also a necessity with the complex processors and peripherals being used today. This chapter will briefly explain all the software being used [2], and developed. The complete software toolkit and applications that were developed is available in CD-ROM form.

5.1 JTAG Development Environment

Both the Actel ProAsic FPGA and the AT91SAM7A2 microprocessor has support for a JTAG (Joint Test Action Group) IEEE 1149.1 compliant debugging interface. With JTAG it is possible to do in-system debugging, set breakpoints, view registers, and even view and change data inside memory connected to the devices at runtime. Programming of the FPGA is also supported via the JTAG link.

5.1.1 Actel ProAsic FPGA

Development of the VHDL code for the Actel ProAsic FPGA was done inside the Libero IDE v7.0 environment. This application is primarily only a frontend for the Synthesizer (SynPlify Pro), Simulation Package (ModelSim) and their own layout- and STAPL (Standard Test and Programming Language) file generator, Libero Designer. The STAPL file is then loaded into FlashPro for programming via the standard JTAG interface. An adapter PC Board was

designed and manufactured to convert the 26 pin micro header used on the FlashPro programmer to the standard, more robust, 26 pin header used on the OBC.

5.1.2 Atmel AT91SAM7A2 ARM7 Microprocessor

There is a multitude of ARM-compliant JTAG debugging tools available for developing applications based on the ARM7 architecture. We decided to use the *Academic Package* available directly from ARM, which contains:

1. ARM Multi-ICE hardware interface for JTAG to PC communications
2. MetroWerks CodeWarrior v1.2 (IDE and Compiler)
3. ARM eXtended Debugger (connects to Multi-ICE for in-system debugging)

Semihosting is a very useful function provided by Multi-ICE, where input and output directed to the standard IO of the AT91SAM7A2 (via `printf` and `scanf`) is redirected to the ARM Debugger on the PC. This makes development and debugging much easier, especially when still developing drivers and no other means of IO is available for debugging purposes.

5.2 Software Development

To mark the successful completion of a hardware design project, certain aspects of the finished design must be measured and verified. In order for this to be done, the peripherals must be set up correctly and receive the correct commands. This is done with driver software.

5.2.1 Bootloader

As soon as the ARM microprocessor comes out of reset-mode, it looks for its first instruction to execute at address `0x00000000`. This is where the bootloader needs to be located. The AT91SAM7A2's Chip Select 0 (address `0x00000000` before **remap**) line is connected to a 2 MByte Flash memory module. The bootloader is an application written in ARM assembly language and will be discussed below.

5.2.1.1 Chip Select Configuration

We first have to specify which memories are attached to the microprocessor. This is done by loading the correct values into the AMC Chip Select registers. We have three external memories which need to be configured. When setting up the AT91SAM7A2 Chip Selects for this board, we only have a choice between either 1 or 4 MByte pages. In our case 4 MBytes is chosen, which means that the last 2 MBytes of each memory location will contain a copy of the first 2 MBytes:

1. **CS0:** 2 MBytes of Flash Memory @ $0x40000000$
2. **CS1:** 2 MBytes of SRAM Memory @ $0x48000000$
3. **CS2:** 2 MBytes of SRAM Memory @ $0x48400000$

5.2.1.2 Set REMAP mode

The second task, is to set the processor to **remap** mode. This will activate all the Chip Select lines as configured in the Chip Select Configuration section of the bootloader. After **remap**, the bootloader code will not be situated at $0x00000000$ anymore, but at $0x40000000$. Therefore, directly after remap, we also need to jump to the new address of the bootloader.

5.2.1.3 Stack Configuration

At this point, we need to set up the stacks for all the different processor modes. During the setup process, all IRQ and FIQ requests are masked. The five modes defined, are given as:

1. Supervisor Mode
2. Undefined Mode
3. Abort Mode
4. FIQ (Fast Interrupt reQuest) Mode
5. IRQ (InterRupt reQuest) Mode

5.2.1.4 Copy data and branch to C Code

All program code currently located in Flash Memory is now copied to SRAM and the interrupt vectors are mapped to internal RAM (located at `0x00000000`). At this point, the General Interrupt Controller is initialized:

1. All interrupts are disabled and cleared.
2. Reset source mode- and source vector registers.
3. Initialize Spurious Interrupt Vector handler.
4. Validate all interrupt levels.

The only thing left at this point, is to switch into User Mode, and branch to the C code `main()` function.

5.2.2 Drivers

The goal of device drivers is to provide an abstraction layer between the Operating System, and the physical hardware devices that it will be interacting with. The operating system has no knowledge about the specific signals needed to make a peripheral work. The message is simply passed from the operating system to the device driver, which translates the message into something the device can understand.

Although the operating system for this Onboard Computer has not been designed yet, the device drivers developed here will allow it to interact with all the different subsystems without the need for detailed knowledge about each device.

5.2.2.1 CAN Bus

Four CAN Controllers are provided on the AT91SAM7A2. Only two of these are used; one as a primary CAN bus, and another as a secondary, backup bus. To obtain the necessary CAN bus voltages, a SN65HVD230D 3.3 V CAN transceiver is used. This device converts the 3.3 V TTL levels from the AT91SAM7A2 into CAN bus compatible levels. Each of these devices are provided with a *Powersave* pin which, when pulled low, enables the device. This pin is connected to the UPIO (Unified Parallel I/O Controller) on the AT91SAM7A2, which makes it possible to enable or disable the transceiver

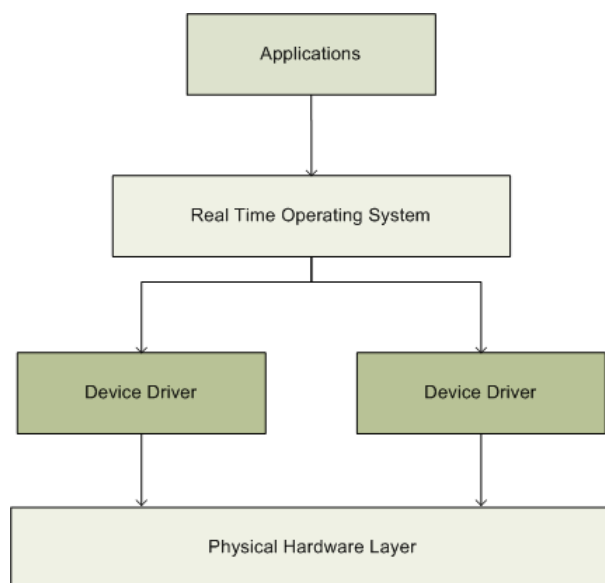
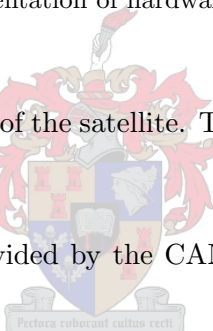


Figure 5.1: Representation of hardware abstraction layer

depending on the current state of the satellite. The transceivers are connected to pins 3 and 4 of the UPIO.

A list of the functions provided by the CAN driver is listed below, with full details in Appendix C.



Function Name	Description
CSP_CANInit	Initializes the CAN Channel and enables clock.
CSP_CANClose	Disables the clock of the CAN Channel.
CSP_CANConfigInterrupt	Configure CAN Interrupts in General Interrupt Controller.
CSP_CANChannelConfigInterrupt	Configure Interrupts inside CAN Module.
CSP_CANEnable	Enables CAN Module.
CSP_CANDisable	Disables CAN Module.
CSP_CANReceive	Set up a CAN Channel to receive messages addressed to it.
CSP_CANTransmit	Transmit a message on a CAN channel.

Table 5.1: Functions provided by CAN driver

5.2.2.2 UART Communications

The AT91SAM7A2 provides two *Universal Synchronous / Asynchronous Receiver / Transmitter* with support for LIN (Local Interconnect Network) and

SmartCard ISO7816-3 protocols. Aboard the OBC the two USARTS will be used in Asynchronous mode for connection to modems for down/uplink to/from earth. The voltage levels supplied by the OBC is TTL 3.3 V and not RS232 as would be expected. Reasons for this are given in the hardware development chapter.

Function Name	Description
CSP_USARTInit	Initializes the USART Channel and enables clock.
CSP_USARTClose	Disables the clock of the USART Channel.
CSP_USARTConfigInterrupt	Configure Interrupts inside the USART Module.
CSP_USARTEnable	Enables USART Module.
CSP_USARTDisable	Disables USART Module.
CSP_USARTReceive	Set up USART to Receive Bytes.
CSP_USARTTransmit	Send a data byte using USART.
CSP_USARTtxData	Send a null-terminated string over USART.

Table 5.2: Functions provided by USART driver

5.2.2.3 Unified Parallel Input/Output Pins

The AT91SAM7A2 has 32 General Purpose I/O pins. Before the UPIO block of the microprocessor can be used, it must first be initialized by setting up the *Multi Driver (Open Drain) Status* and Input/Output status of every pin.

In the case of our OBC, no pins should be set to Multi Driver, and the pins shown in Table 5.3 are set as outputs (all others are implicitly defined as inputs).

5.2.2.4 Temperature Sensor

The OBC is equipped with a LM70 Digital Temperature Sensor from National Semiconductor. It uses a standard SPI interface to communicate to the AT91SAM7A2. SPI communications to the LM70 will be done in software without the use of the hardware SPI module onboard the OBC.

To receive temperature data from the LM70, a clock needs to be supplied. A complete receive/transmit communication phase will consist of 32 clock cycles, with the first 16 as receive, and the next 16 as transmit. We will only be receiving temperature data from the LM70, so we only need 16 clock cycles.

UPIO Pin(s)	Peripheral Attached
0	EDAC Disable
3..4	CAN[0..1] Powersave
5	TimeSync (RS485 Pulse)
8	LVDS Tx Clock
9	LVDS Tx Data
11	Real Time Clock Tx
12	Real Time Clock Clock
13	Real Time Clock Enable
15	Temperature Clock
16	Temperature Enabled
26	LVDS RX Clock (Output)
27	LVDS RX Read Enable (Output)
29	LVDS RX Not Reset FIFO (Output)
30	LVDS RX Not Write Enable (Output)

Table 5.3: ARM7 UPIO pins configured as outputs at boot time

Function Name	Description
CSP_PIOInit	Initializes the UPIO block and enables clock.
CSP_PIOClose	Disables the clock of the UPIO block.
CSP_PIOConfigInterrupt	Configure Interrupts inside the UPIO Module.
CSP_PIOGetStatus	Get Status of pins attached to the UPIO Module.
CSP_PIOClear	Set UPIO Pin to '0'.
CSP_PIOSet	Set UPIO Pin to '1'.

Table 5.4: Functions provided by UPIO driver

To receive data from the LM70, the \overline{CS} line needs to be held low while transmitting a clock signal. The data being received will consist of (Figure 5.2):

- One Sign Bit
- 10 Temperature Bits
- 3 High Bits
- 2 High Impedance Cycles

A device driver was written for obtaining this data from the LM70 chip. It is meant to be run inside a timer on a multi-process operating system as long as the timer is not faster than 6.25 MHz [10]. A flow diagram illustrating

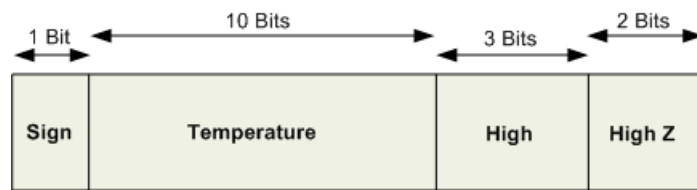


Figure 5.2: Received bit sequence from LM70

the the generic software SPI driver that was developed is shown in Figure 5.3 together with Figure 5.4 that shows the SPI connections to the AT91SAM7A2.

5.2.2.5 Realtime Clock

The AT91SAM7A2 does not come equipped with a Real Time clock embedded. In order to perform scheduled tasks at specific moments in time we need to have a stable time reference. The M41ST95W from ST Micro Electronics is a digital SPI compatible Real Time Clock with an embedded crystal.

The SPI interface to the RTC will also be done in software, since the hardware SPI on the AT91SAM7A2 is already in use. Since high speed communication to/from the RTC is not necessary, this will not be a problem.

Before the RTC can be used, it must first be initialized [13]. When the RTC is cold-booted, the HT (Halt Update) bit is automatically set. Before timekeeping can commence, this first has to be cleared.

Read and Write Cycles

The RTC is a bit more complex than the LM70 previously discussed, and will also need one or more write cycles before reading can commence.

The Read- and Write Mode Sequences for the RTC are displayed in Figure 5.6 and Figure 5.7.

The RTC considers the first bit sent to it after \overline{CS} goes low. If this bit is a '1' one or more *write* cycles will occur. If however, this bit is a '0', one or more *read* cycles is to follow.

In the simplest case, a write to $0x0C$ to clear the HT bit is all that is necessary to start the RTC. After this, the reading of any register can be done.

The driver functions that were provided for use with the RTC is shown in Table 5.5.

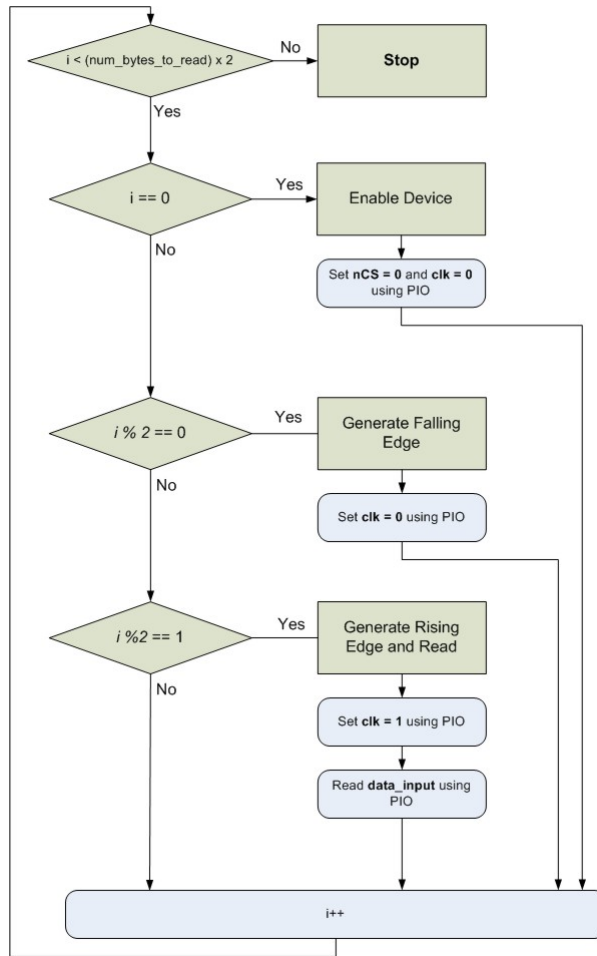


Figure 5.3: Flow Diagram of Software SPI / LVDS Driver

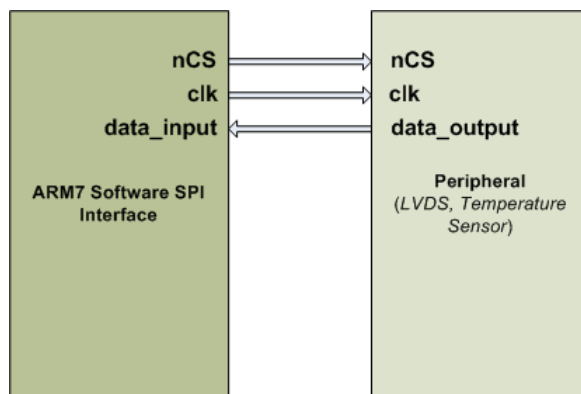


Figure 5.4: Read-Only SPI Connections between peripheral and AT91SAM7A2

Addr									Function/Range BCD Format	
	D7	D6	D5	D4	D3	D2	D1	D0		
00h	0.1 Seconds				0.01 Seconds				10ths/100ths of Seconds	00-99
01h	ST	10 Seconds			Seconds				Seconds	00-59
02h	0	10 Minutes			Minutes				Minutes	00-59
03h	CEB	CB	10 Hours		Hours (24 Hour Format)				Century/Hours	0-1/00-23
04h	TR	0	0	0	0	Day of Week			Day	01-7
05h	0	0	10 Date		Date: Day of Month				Date	01-31
06h	0	0	0	10M	Month				Month	01-12
07h	10 Years				Year				Year	00-99
08h	OUT	FT	S	Calibration					Control	
09h	WDS	BMB4	BMB3	BMB2	BMB1	BMB0	RB1	RB0	Watchdog	
0Ah	AFE	SQWE	ABE	AI 10M	Alarm Month				AI Month	01-12
0Bh	RPT4	RPT5	AI 10 Date		Alarm Date				AI Date	01-31
0Ch	RPT3	HT	AI 10 Hour		Alarm Hour				AI Hour	00-23
0Dh	RPT2	Alarm 10 Minutes			Alarm Minutes				AI Min	00-59
0Eh	RPT1	Alarm 10 Seconds			Alarm Seconds				AI Sec	00-59
0Fh	WDF	AF	0	BL	0	0	0	0	Flags	
10h	0	0	0	0	0	0	0	0	Reserved	
11h	0	0	0	0	0	0	0	0	Reserved	
12h	0	0	0	0	0	0	0	0	Reserved	
13h	RS3	RS2	RS1	RS0	0	0	0	0	SQW	

Keys: S = Sign Bit
 FT = Frequency Test Bit
 ST = Stop Bit
 0 = Must be set to '0'
 BL = Battery Low Flag (Read only)
 BMB0-BMB4 = Watchdog Multiplier Bits
 CEB = Century Enable Bit
 CB = Century Bit
 OUT = Output level
 AFE = Alarm Flag Enable Flag
 RB0-RB1 = Watchdog Resolution Bits
 WDS = Watchdog Steering Bit
 ABE = Alarm in Battery Back-Up Mode Enable Bit
 RPT1-RPT5 = Alarm Repeat Mode Bits
 WDF = Watchdog flag (Read only)
 AF = Alarm flag (Read only)
 SQWE = Square Wave Enable
 RS0-RS3 = SQW Frequency
 HT = Halt Update Bit
 TR = t_{REC} Bit

Figure 5.5: TimeKeeper Register Map [13]

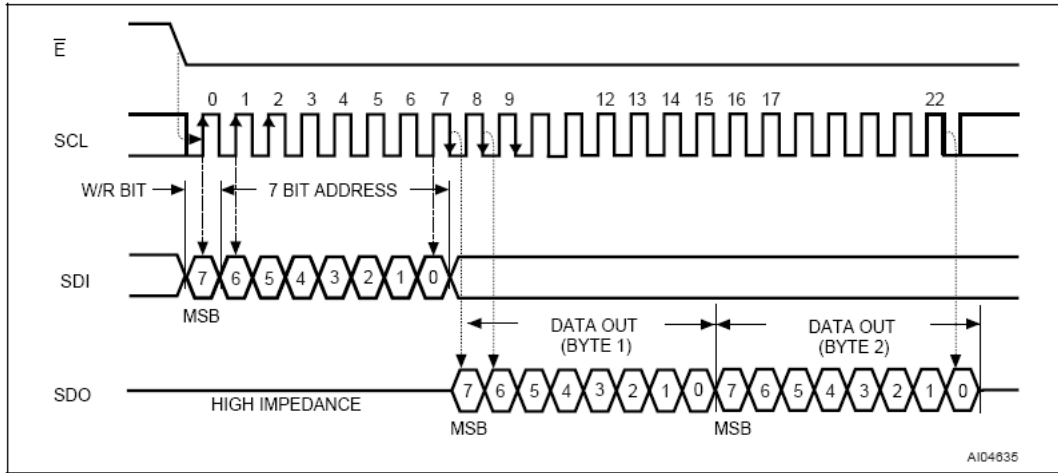


Figure 5.6: TimeKeeper Read Mode Sequence [13]

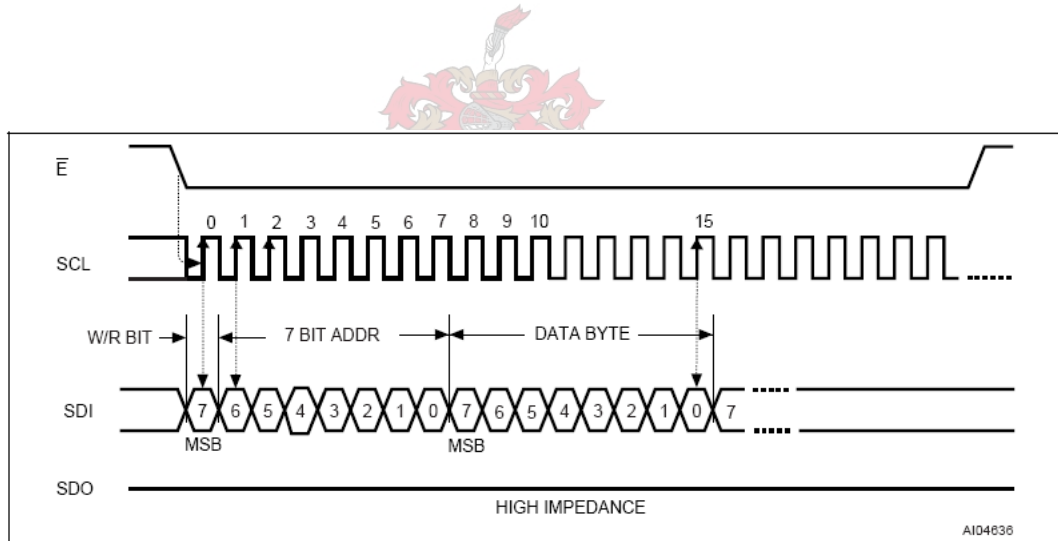


Figure 5.7: TimeKeeper Write Mode Sequence [13]

Function Name	Description
ReadRTC	Enable the RTC and read 8 bits from a specified address.
WriteRTC	Enable the RTC and write 8 bits to a specified address.

Table 5.5: Functions provided by RTC driver

It should be noted that it is also possible to *Burst Read* or *Burst Write* from/to the RTC by just continuously writing / reading data without sending an address again. The register address on the RTC is automatically incremented in this mode. Currently the driver provided only supports the reading / writing of one byte at a time.

5.2.2.6 Analog to Digital: Current and Voltage Sensing

To measure the current consumption of the OBC and voltage output of the switch mode regulators, the embedded A/D controller on the AT91SAM7A2 is used.

The microcontroller has two different sampling modes. The first one, is one-shot mode where a single sample is taken and converted. The second one, is a continuous mode where the conversion is done completely independant by the A/D peripheral. Data is transferred from the A/D converter directly into memory using DMA (the Peripheral Data Controller). This is the mode that we will be using.

The driver functions provided for the A/D conversion is shown in Table 5.6.

Function Name	Description
CSP_ADC8CInit	Initializes the ADC block and enables clock.
CSP_ADC8Cclose	Disables the clock of the ADC block.
CSP_ADC8CConfigInterrupt	Configure Interrupts inside the ADC Module.
CSP_ADC8CEnable	Enable the A to D Converter.
CSP_ADC8CDisable	Disable the A to D Converter.
CSP_ADC8CStartConversion	Start Conversion on A to D Channel.
CSP_ADC8CStopConversion	Stop Conversion on A to D Channel.
CSP_PDCStartRx	DMA Function called by <i>ADC8CStartConversion</i> .

Table 5.6: Functions provided by Analog to Digital converter

5.2.2.7 High Speed Serial Peripheral Interface

The hardware Serial Peripheral Interface supplied on the AT91SAM7A2 can be used at a bitrate of up to 30 MBits / s. It is also tied in to the Periph-

eral Data Controller of the AT91SAM7A2 which makes large block transfers without the need for software intervention from the microprocessor, possible. The onchip SPI has support for 4 devices, selected by 4 SPI CS lines.

Typical uses for the SPI Bus will be the transferring of large data blocks (for example, images) from a Mass Storage Device or Camera.

The software interface provided for the SPI module is given here:

Function Name	Description
CSP_SPIInit	Initializes the SPI block and enables clock.
CSP_SPIClose	Disables the clock of the SPI block.
CSP_SPIConfigureCS	Configure the Chip Select Lines of the SPI Module.
CSP_SPIConfigInterrupt	Configure Interrupts inside the SPI Module.
CSP_SPIEnable	Enable the SPI Interface.
CSP_SPIDisable	Disable the SPI Interface.
CSP_SPIReceive	Set up the address at which the received data will be stored.
CSP_SPITransmit	Set the address and length of data to transmit via PDC.

Table 5.7: Functions provided by hardware SPI



5.2.2.8 LVDS Data Link

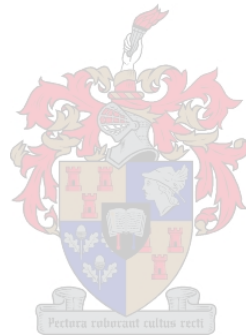
All the peripherals on the OBC is driven *from* the AT91SAM7A2. This has the big advantage that certain clock cycles can be *delayed* if the OBC is busy inside another time critical function with a high priority.

This will make the realization of a real time operating system using this board much easier, since the peripherals (LVDS, SPI, Temperature, Real Time Clock) will just wait for the next clock cycle to be supplied by the board before continuing the data transfer.

Three functions are provided for downloading data from the Dual Clock FIFO. The IDT72V201 has a buffer of 4 KBytes and a read/write cycle time of 10 ns. Before any transfers to or from the buffer can commence, the read and write modules first has to be enabled. When there is data in the buffer, the *Not Empty* flag will be asserted, and we can start retrieving data until this flag is cleared.

Function Name	Description
CSP_InitLVDS	Enables the receive and transmit blocks of the FIFO.
CSP_isNEmptyLVDS	Provides the status of the FIFO.
CSP_DoLVDS	Retrieves one data byte from the FIFO.

Table 5.8: Functions provided by software LVDS receiver



Chapter 6

Tests and Measurements

No hardware project can be deemed successful without the completion of tests to validate the design. This chapter will outline the specific tests that were done which proved the initial design constraints.

6.1 Memory System and EDAC

The 2 MBytes of Flash Memory was tested by reading the manufacturer ID from the chip, and then using the Common Flash Interface (CFI) to write a test pattern to the module which was successfully read back.

A test pattern was written to the 4 MBytes of SRAM and was successfully read back to the system with EDAC enabled and disabled. Simulation of the VHDL EDAC system implemented in the FPGA, using ModelSim 6.3c, shows the design to be successful. A Test pattern (1010101111000101b) was written to memory (as shown in Figure 6.1). This same pattern was then read back from memory (Figure 6.2) and then read back with one bit altered (Figure 6.3). In the last case, the bit was corrected, and the **Error** signal was set to 01.

With the EBI running at 30 MHz, each clock cycle lasts 33.3 ns. The AT91SAM7A2 asserts the NWE (Not Write) signal after 50% of the clock cycle is completed. This means that the EDAC effectively has 16.6 ns in which it has to perform the error checking and correction. Using a logic analyzer, the precise timings were evaluated and both the Read, and Write cycles took a maximum of 10 ns to complete. Figure 6.4 and Figure 6.5 show a write cycle

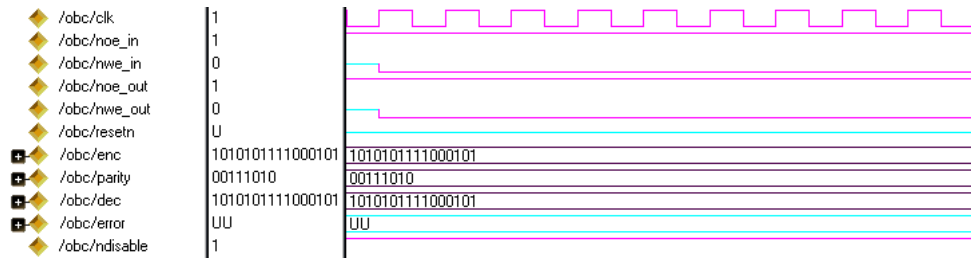


Figure 6.1: EDAC Encoding of data word

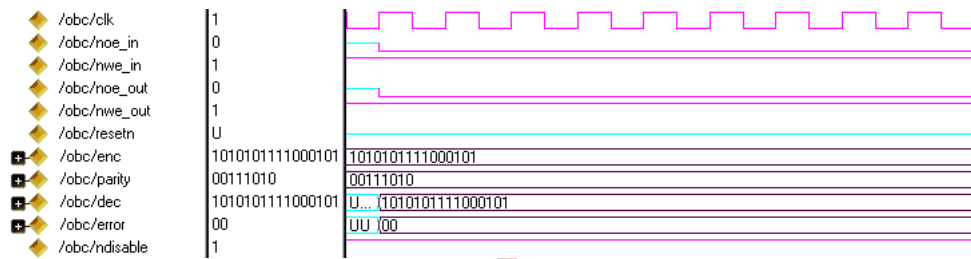


Figure 6.2: EDAC Decoding of data word

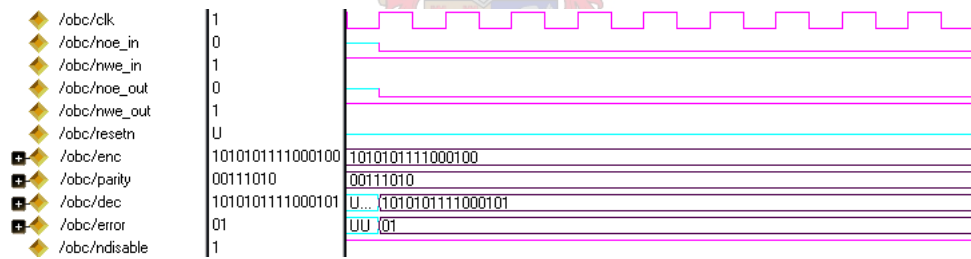


Figure 6.3: EDAC Decoding of invalid data word

with the EDAC enabled. $D_0 - D_7$ are connected directly to the AT91SAM7A2, while $D_8 - D_{15}$ are connected to the SRAM. It is very clear from Figure 6.4 that the data is slightly delayed by the EDAC unit. By looking at Figure 6.5 the delay of 10 ns can easily be seen.

At this point no error was detected yet by the EDAC system (as would be expected) while working in the laboratory. One of the outputs of a memory module containing data was shorted to ground and the correction was visible using a logic analyzer. The check-bit data stored in the check-bit-memory was also analyzed and proved to be correct.

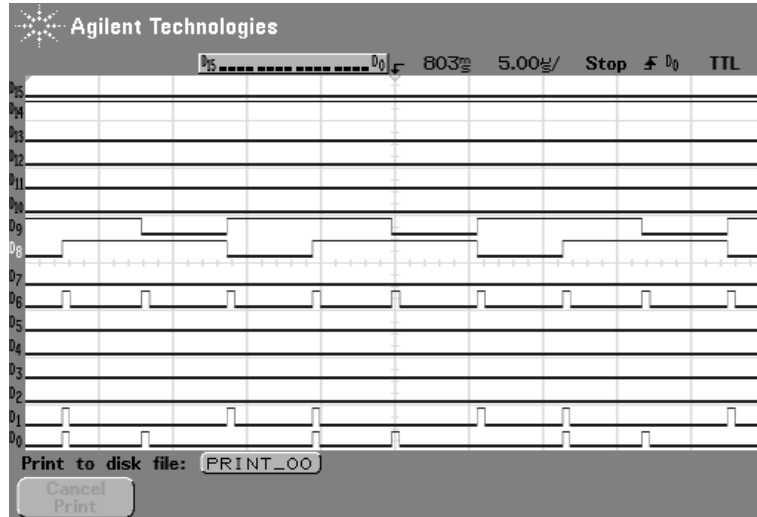


Figure 6.4: EDAC Writing Sequence

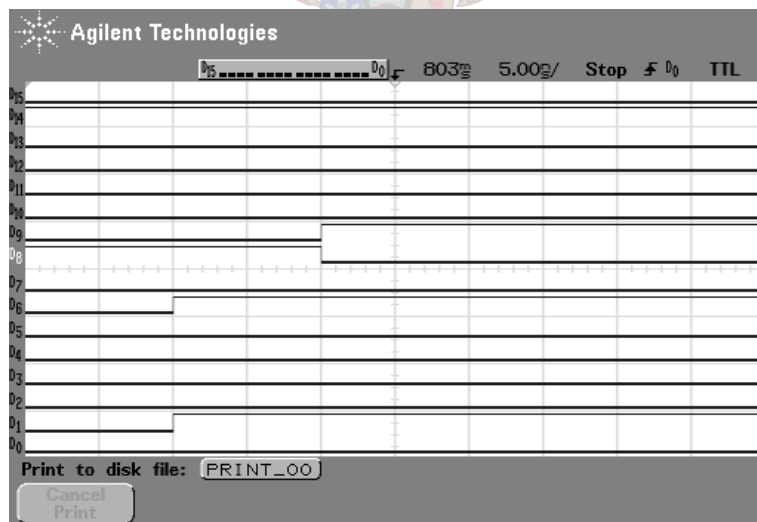


Figure 6.5: EDAC Writing Sequence (zoomed in)

6.2 Peripherals

CAN Controller Messages were successfully sent and received at 100 kbps by/from the OBC by using a PeakCAN USB module connected to a PC.

USART Controller Data bytes were sent and received at 9600 baud from the OBC by using a RS232 transceiver module to convert the 3.3 V TTL levels to RS232 PC compatible levels.

LM70 Temperature Controller Temperature data was received from the LM70 module by using the driver developed for it. Values in °C were recorded by taking the data read from the device, and dividing it by four [10]. Differences in degrees measured could easily be seen by warming the board up slightly with a heat gun.

Real Time Clock The Real Time Clock was started by writing a '0' to the HT (Halt Update) bit. Timing data was read from the RTC by reading data from registers at $0x00$ and $0x01$ [13]. Once the Halt Update bit was reset, the RTC started counting. In a satellite environment, once the satellite has been deployed and the OBC switched on, the correct time should be uploaded to the RTC, after which the HT bit should be cleared. The data received from the RTC by our software SPI interface is shown in Figure 6.6

Hardware SPI with Direct Memory Access (DMA) An 8051 evaluation board was used to generate 8 bytes of SPI data which was directly written into SRAM on the OBC using the SPI functions discussed in the previous chapter. The OBC is set as SPI master, generating a clock and sending a 8-bit *instruction sequence*, where after the 8051 (simulating a Mass Storage Device as SPI Slave) send the 8-bytes of data.

Analog-to-Digital Conversion The ADC0 channel of the AT91SAM7A2 was set up in continuous capture mode with the first 4 lines (In0 - In3) enabled. The resulting data that was received was the 3.3 V and 2.5 V (divided in two by a voltage divider) and the current transformed to a voltage by the INA196. The data was sent over USART0 for validation.

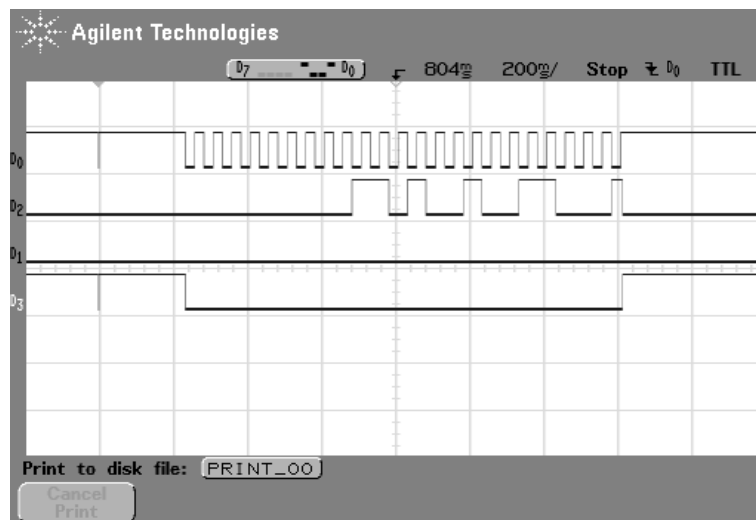


Figure 6.6: Data received by SPI (software emulation) on AT91SAM7A2

RS-485 Time Pulse The RS-485 time pulse is a signal lasting 1ms which is supplied at the start of each second. This allows other subsystems on the satellite to be in precise synchronization with the OBC at all times. UPIO[5] on the AT91SAM7A2 is connected to the input of a RS-485 transceiver which converts the standard TTL 3.3 V into a RS-485 compatible bus voltage level. Another RS-485 transceiver was connected to the OBC, and could pick up the transitions on the bus successfully.

LVDS Mass Storage Device Link The primary connection for a Mass Storage Device or Camera onboard the nanosatellite should be the hardware SPI bus. For compatibility with other systems using a LVDS link, an LVDS deserializer and dual clock FIFO was also incorporated into this design. 8-bit Data sent by using a 8051 Microcontroller was serialized with an LVDS serializer. This data was then sent to the OBC, where the data was deserialized and stored temporarily in the Dual Clock FIFO.

Only software transfer (involving the AT91SAM7A2 microprocessor) of the data from the dual clock FIFO to SRAM is currently supported, so moving large amounts of data may prove to be very computationally expensive. That is why a dual clock FIFO was used: Whenever the AT91SAM7A2 is idle, it can transfer data from the FIFO at its own pace. It does not have to

necessarily keep up with the clock of the LVDS deserializer.

When the dual clock FIFO is almost full, the almost full flag will be raised and data transfer from the MSD will temporarily stop.

6.3 Measurements

6.3.1 Power Consumption

The design constraint placed on the power consumption of the OBC is that it should not consume more than 1.5 Watt of power with all systems operating. The startup current of the board is quite high, at 166 mA measured at a bus voltage of 14 Volts. Current drawn from a source under different operating conditions is shown in Table 6.1.

CAN	SPI	USART	Processor	Memory	I [3.3 V]	I [2.5 V]	I [14 V]
Off	Off	Off	Idle	No	34 mA	5 mA	38 mA
On	Off	Off	Idle	No	43 mA	5 mA	42 mA
On	On	On	Idle	No	47 mA	5 mA	46 mA
On	On	On	Memory Access	Read/Write	48 mA	5 mA	48 mA
On	On	On	Floating Point	Read/Write	48 mA	5 mA	48 mA
On	On	On	Halted	No	40 mA	5 mA	30 mA

Table 6.1: Current consumption of OBC

6.3.2 Processing Capabilities

Before starting this project there was one restriction on the speed of the microprocessor: A Complex IGRF mathematical function needs to be executed each second onboard the satellite. The fact that the AT91SAM7A2 does not have a math coprocessor was a concern, because the IGRF functions work with floating point numbers - this will have to be emulated in software which can be very computationally expensive. The IGRF function was tested on the evaluation board of the AT91SAM7A2 before making a decision about which microprocessor to use. The evaluation board executed the function in 50ms while running at a Core Clock frequency of 20 MHz.

The OBC designed runs at a frequency of 30 MHz and takes approximately 35 ms to execute the function. There is still more than enough processing time left for other processes and housekeeping. The processor is thus more than fast enough to cope with the software applications used in orbit.



Chapter 7

Conclusions and Recommendations

7.1 Conclusions

After successful testing and validation of the initial design constraints, it is proven that this system conforms to all specifications set out in the beginning of this document.

7.1.1 Reliability

Although this board performs excellent in a laboratory environment, it would be interesting to subject the OBC to some environmental and radiation tests to see how it holds up under these real-world space conditions.

The Actel ProAsic FPGA has gone through numerous radiation tests for use aboard an experimental payload on **Sumbandilasat** due to launch in June 2007 and has passed them all with flying colours.

7.1.2 Power Consumption

The complete OBC board draws less than 700 mW from the power system when all peripherals are enabled. Since the switch-mode power converters are only operating at about 20% capacity, the efficiency is poor. Using a regulator from another manufacturer could prove to make the OBC power system much more efficient than it currently is. The power consumption of all the individual peripherals (such as SRAM, LVDS Deserializer, etc.) could not be measured

since the board does not make provision for such a measurement. The total OBC power consumption is well within the 1 Watt limit defined for typical OBC power consumption.

7.1.3 AT91SAM7A2 as a OBC

Although the AT91SAM7A2 has no hardware floating point unit, running at 30 MHz, the processor performs excellent and can complete even the most demanding floating point function with time to spare. With the cost of the industrial specification processor at around R70.00 each, it makes for a very economical choice on an OBC.

7.2 Recommendations

During this project, a few aspects were identified that could be improved further. These are highlighted in the following section.

7.2.1 Development Board Layout

Due to a miss-read of the Actel FPGA Mechanical specifications, the footprint for the FPGA on the PC Board was about 1.5 mm too big on either sides. With some clever solder work from *Johan Arendse*, the pads were extended so that the FPGA could fit. The footprint was readjusted and no other errors on the board were spotted.

7.2.1.1 Single Event Latchups

A Single Event Latchup occurs as a result of radiation accumulating inside a semiconductor. At this point the device no longer responds to input signals. Excessive current flow may be a result of a SEL and power cycling can normally reset this device to a functional state.

Although a INA196 current sensor is provided on the OBC, it is recommended that the output of this sensor is also fed to a level comparator, which can trigger a *trip* signal to the power system of the nanosatellite. Power can then be automatically restored after a few seconds, when the effects of the SEL should have subsided.

7.2.2 LVDS Data Link

7.2.2.1 LVDS (using DMA via SPI)

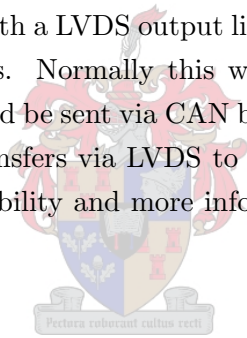
Due to the fact that the SPI bus is the primary datapath for Mass Storage Devices, the LVDS link (provided for compatibility) does not have a DMA link into the memory of the OBC. If this becomes a requirement at some point, the LVDS deserializer can be exchanged for a standard LVDS receiver. This can be buffered by one of the input/output pin pairs on the dual clock FIFO. The FIFO can in turn be connected to the SPI bus, which will allow serial DMA access (at a lower speed than with the serializer / deserializer pair) to the LVDS data. A proposal for this design is given in Appendix D.

7.2.2.2 LVDS Output

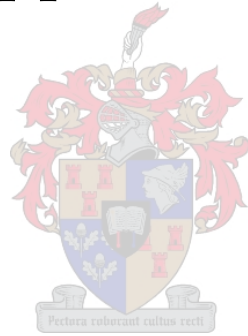
The OBC is also equipped with a LVDS output link to send data and messages to LVDS compatible devices. Normally this would not be used onboard a satellite, since messages would be sent via CAN bus to the respective modules, which would initiate the transfers via LVDS to the OBC. The LVDS output is only provided for compatibility and more information on it can be seen in Appendix E.

7.2.3 OBC Design

To conclude: This design illustrates the successful use of an ARM7TDMI processor incorporating all the peripherals typically required by a nanosatellite.



Appendices



Appendix A

Physical PCB Layout

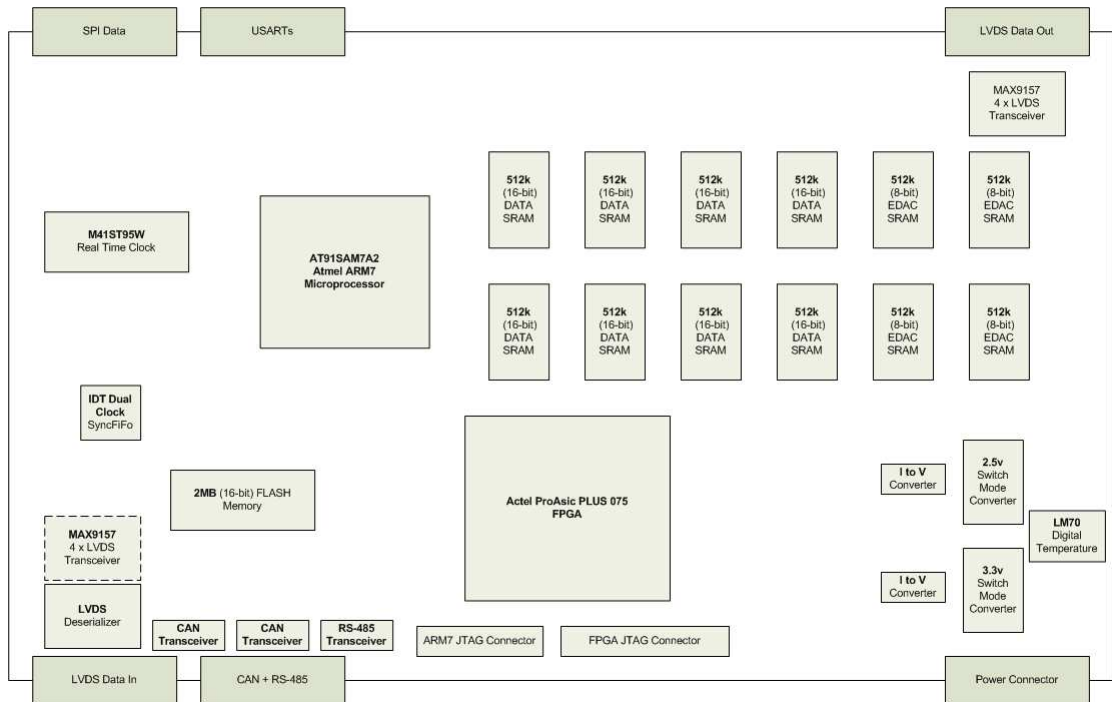


Figure A.1: Schematic Representation of PC Board Layout for Nanosatellite OBC

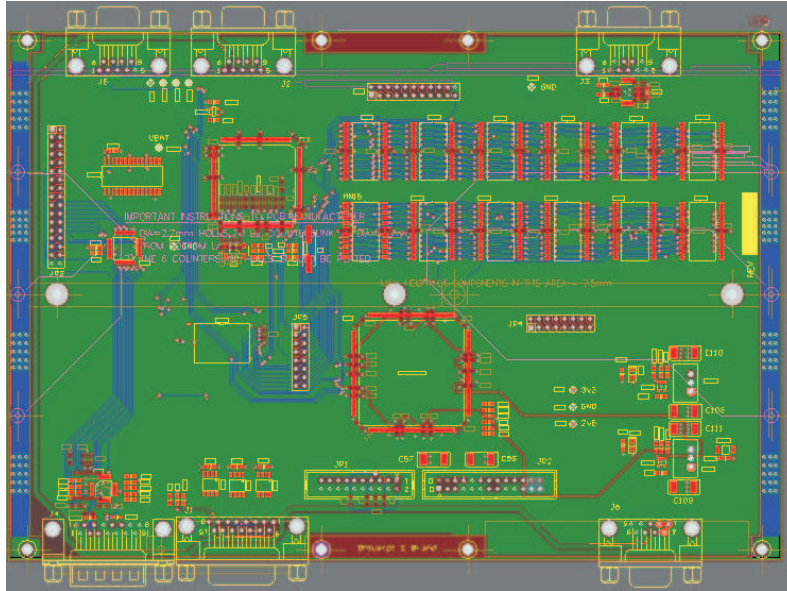


Figure A.2: PC Board Layout for Nanosatellite OBC

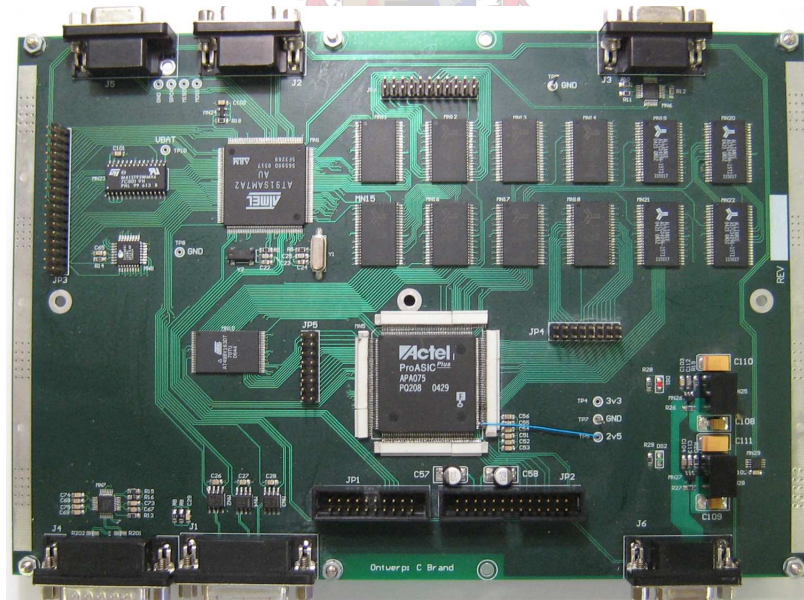
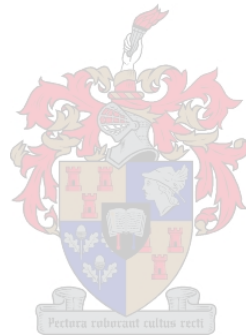


Figure A.3: Photo of Nanosatellite OBC

Appendix B

Design Schematics

Please see the included CD-ROM for the design schematics (DXP 2004).



Appendix C

Peripheral Software Support

C.1 CAN Bus

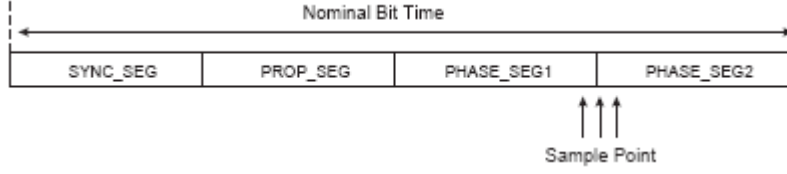
Before any communication on a CAN bus can commence it first has to be initialized using the correct parameters. This is specified by setting the Mode register.

Table C.1: CAN Mode Register [0x064]

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	PHSEG2[2:0]			-	PHSEG1[2:0]		
15	14	13	12	11	10	9	8
-	SMP	SJW[1:0]		-	PROP[2:0]		
7	6	5	4	3	2	1	0
-	-	BD[5:0]					

BD[5:0]: Time Quantum Period These bits are used to determine the time quantum t_{CAN} . The time quantum, together with the synchronization segment (always equal to one time quantum), propagation segment (t_{PRS}) and two phase segments (t_{PHS1} and t_{PHS2}) define the nominal bit time. The bus speed is given as the inverse of the nominal bit time.

$$t_{CAN} = \frac{BD[5:0] + 1}{CORECLK} \quad (C.1.1)$$

**Figure C.1:** Partition of the bit time

$$t_{PRS} = t_{CAN} \times (PROP[2:0] + 1) \quad (C.1.2)$$

$$t_{PHS1} = t_{PHS2} = t_{CAN} \times (PHSEG[2:0] + 1) \quad (C.1.3)$$

The CAN controller resynchronizes on each edge of the transmission. The t_{SJW} value defines the maximum number of CAN clock cycles a bit period may be shortened or lengthened. A typical value for t_{SJW} (as given by the AT91SAM7A2 datasheet) is $t_{CAN} \times 3$.

$$t_{SWJ} = t_{CAN} \times (SWJ[1:0] + 1) \quad (C.1.4)$$

For a bus speed of $100kbps$, sampled once every sample point:

$$(t_{SYNC} + t_{PRS} + t_{PHS1} + t_{PHS2}) \times t_{CAN} = \frac{1}{100kbps} \quad (C.1.5)$$

Symbol	Value
BD[5:0]	14
PROP[2:0]	6
SWJ[1:0]	2
SMP	0
PHSEG1[2:0] = PHSEG2[2:0]	5

Table C.2: CAN Mode Register

After initialization the CAN channel needs to be enabled. Writing a 1 into the CHANEN bit of the CAN Control Register will enable the channel so that it can send or receive data.

Table C.3: CAN Control Register [0x060]

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
-	OVDIS	OVEN	ABDIS	ABEN	CANDIS	CANEN	SWRST

To actually send data on the CAN bus the CAN channel, ID of the message, the actual data, and control bits needs to be specified. This is done by writing to the CAN Channel- Identifier, Data, and Control registers of the channel in question.

The data is automatically sent as soon as the control register receives the correct values. The transmission will be resent on the bus until at least one node acknowledges the receipt of the data.

Short description of selected software functions

void CSP_CANInit (CSP_CAN_T *const can, U32_T mode)

Description: Switches on the clock and reset the registers.

Inputs:

- **can*: Pointer to CAN structure.
- *mode*: Configure the CAN mode register.

Returns: None.

void CSP_CANClose (CSP_CAN_T *const can)

Description: Switches off the clock.

Inputs: **can*: Pointer to CAN structure.

Returns: None.

void CSP_CANConfigInterrupt (CSP_CAN_T *const can, U32_T int_mode, U32_T int_mask, U32_T callback)

Description: Configure CAN Interrupts.

Inputs:

- **can*: Pointer to CAN structure.
- *int_mode*: Configure the priority level and source type.
- *int_mask*: Configure which interrupt bits are activated.
- *callback*: Function called through the assembler interrupt handler.

Returns: None.

CSP_CANEnable (CSP_CAN_T *const can)

Description: Enable CAN (12 μ s is necessary at 30 MHz after an enable command to transfer and receive data).

Inputs: **can*: Pointer to CAN structure.

Returns: None.

void CSP_CANDisable (CSP_CAN_T *const can)

Description: Disable CAN.

Inputs: **can*: Pointer to CAN structure.

Returns: None.

void CSP_CANReceive (CSP_CAN_T *const can, U8_T channel, U32_T id, U32_T mask, U16_T control)

Description: Configure a CAN channel to receive messages from the network.

Inputs:

- **can*: Pointer to CAN structure.
- *channel*: Receiver channel [0-31].
- *id*: Configure the identifier and the remote request.
- *mask*: Configure the mask and the remote request.
- *control*: Configure the number of bytes to receive and the control bits.

Returns: None.


```
void CSP_CANTransmit (CSP_CAN_T *const can, U8_T channel,
U32_T id, U8_T *data, U16_T control)
```

Description: Configure a CAN channel to transmit message on the CAN network.

Inputs:

- **can*: Pointer to CAN structure.
- *channel*: Transmitter channel [0-31].
- *id*: Configure the identifier and the remote request.
- **data*: Pointer to an array where data packet sent is stored.
- *control*: Configure the number of bytes to send and the control bits.

Returns: None.

A simple example to transmit two bytes of data on the primary CAN bus (@ 100kbps) is given below (assuming that the CAN transceivers are not in powersave mode):

```
CSP_CAN_T* myCAN;
U8_T data[2] = {0x33, 0x33};

myCAN = ((CSP_CAN_T*) CAN0_BASE_ADDRESS);

CSP_CANInit(myCAN, 0x55260E);
CSP_CANEnable(myCAN);
CSP_CANTransmit(myCAN, 0, 0x01, data, 0xC2);
```

C.2 UART Communication

Before being able to send or receive data on a USART, it first has to be initialized by setting some parameters. Initialization of a USART consists of setting the Mode Register, together with the Baud Rate Register and Receiver Time Out Register.

Settings concerning the configuration of the USART in Asynchronous mode is given by:

Table C.4: USART Mode Register [0x064]

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	CLKO	MODE9	SMCARD
15	14	13	12	11	10	9	8
CHMODE[1:0]		NBSTOP[1:0]		PAR[2:0]			SYNC
7	6	5	4	3	2	1	0
CHRL[1:0]		USCLKS[1:0]		SENDDTIME[1:0]		-	LIN

LIN : Local Interconnect Network Mode

0: USART does not support LIN mode

USCLKS[1:0] : Baud Rate Generator Input Clock

00: CORECLK is used as the Input Clock to the Baud Generator

CHRL[1:0] : Character Length

11: 8-bit Character Length

SYNC : Synchronous Mode Select

0: Operating in Asynchronous Mode

PAR[2:0] : Parity Type

100: No parity

NBSTOP[1:0] : Number of Stop Bits

00: 1 stop bit

CHMODE[1:0] : Channel Mode

00: Normal Mode: The USART Channel operates as a Rx/Tx USART

SMCARD : Smart Card Protocol

0: The USART is not in Smart Card Mode

MODE9 : 9-bit Character Length

0: CHRL defines character length



Table C.5: USART Baud Rate Generator Register [0x088]

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	CLKO	MODE9	SMCARDPT
15	14	13	12	11	10	9	8
CD[15:8]							
7	6	5	4	3	2	1	0
CD[15:0]							

CD[15:0] : Baud Rate Generator

$$BaudRate = \frac{SelectedClock}{16 \times CD[15 : 0]} \quad (C.2.1)$$

Table C.6: USART Receiver Time Out Register [0x08C]

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
TO[15:0]							
7	6	5	4	3	2	1	0
TO[7:0]							

TO[15:0] : Receiver Timeout

When a new character is received the timeout counter is reloaded with TO[15:0].

$$TimeOutDuration = TO[15 : 0] \times 4 \times BitPeriod \quad (C.2.2)$$

Sending data :

Data is sent by writing the character to the Transmit Holding Register.

Receiving data :

The last byte received is held in the Transmit Holding Register. It is possible

Table C.7: USART Transmit Holding Register [0x084]

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	TXCHR[8]
7	6	5	4	3	2	1	0
TXCHR[7:0]							

to set up the interrupt register so that an interrupt is generated each time a character is received.

Short description of selected software functions

void CSP_USARTInit (CSP_USART_T *const usart, U32_T mode, U16_T baudrate, U8_T time_out, U8_T time_guard)

Description: Switch on the clock, reset the registers and configure the USART module mode and the PDC RX/TX lines.

Inputs:

- **usart*: Pointer to USART structure.
- *mode*: Configure the USART mode.
- *baudrate*: Configure the Baud Rate Generator.
- *time_out*: Configure the receiver time-out.
- *time_guard*: Configure the transmit time-guard.

Returns: None.

void CSP_USARTClose (CSP_USART_T *const usart)

Description: Reset and switch off the clock.

Inputs: **usart*: Pointer to USART structure.

Returns: None.

void CSP_USARTConfigInterrupt (CSP_USART_T *const usart, U32_T int_mode, U32_T int_mask, U32_T callback)

Description: Configure USART Interrupts.

Inputs:

- **usart*: Pointer to USART structure.
- *int_mode*: Configure the priority level and source type.
- *int_mask*: Configure which interrupt bits are activated.
- *callback*: Function called through the assembler interrupt handler.

Returns: None.

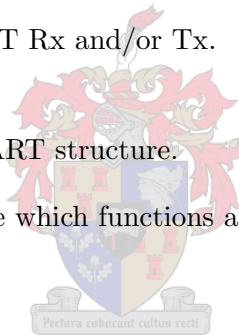
void CSP_USARTEnable (CSP_USART_T *const usart, U32_T enable_mask)

Description: Enable USART Rx and/or Tx.

Inputs:

- **usart*: Pointer to USART structure.
- *enable_mask*: Configure which functions are enabled.

Returns: None.



void CSP_USARTDisable (CSP_USART_T *const usart, U32_T disable_mask)

Description: Disable USART Rx and/or Tx.

Inputs:

- **usart*: Pointer to USART structure.
- *disable_mask*: Configure which functions are disabled.

Returns: None.

void CSP_USARTReceive (CSP_USART_T *const usart, U8_T *data, U16_T length)

Description: Configure USART to receive data. The data will be directly transferred into memory using the PDC.

Inputs:

- **usart*: Pointer to USART structure.
- **data*: Pointer to an array where data packet received will be store.
- *length*: Number of bytes to receive.

Returns: None.

void CSP_USARTTransmit (CSP_USART_T *const usart, U8_T *data, U16_T length)

Description: Configure USART to transmit data directly from memory.

Inputs:

- **usart*: Pointer to USART structure.
- **data*: Pointer to an array where data packet to be sent is stored.
- *length*: Number of bytes to transmit.

Returns: None.

A simple example to transmit a data byte over USART0 (@ 9600bps) is given below:

```
CSP_USART_T* myUSART;
U8_T data_test = 0x41;

myUSART = ((CSP_USART_T*) USART0_BASE_ADDRESS);

CSP_USARTInit(myUSART,0x8C0,195,0,0);
CSP_USARTEnable(myUSART,0x50);
CSP_USARTTransmit(myUSART,&data_test,1);
```



C.3 PIO Controller

A Short description of the functions provided by the software PIO controller, is described below.

void CSP_PIOInit (CSP_PIO_T *const pio, U32_T output_pio, U32_T multidriver_pio)

Description: Switch on the clock, reset the registers and configure the PIO module mode.

Inputs:

- **pio*: Pointer to PIO structure.
- *output_pio*: Configure which pins are set as outputs.
- *multidriver_pio*: Configure which pin are configured as Multi-Driver (open drain).

Returns: None.

void CSP_PIOClose (CSP_PIO_T *const pio)

Description: Reset and switch off the clock.

Inputs: **pio*: Pointer to PIO structure.

Returns: None.

void CSP_PIOConfigInterrupt (CSP_PIO_T *const pio, U32_T int_mode, U32_T int_mask, U32_T callback)

Description: Configure PIO Interrupts.

Inputs:

- **pio*: Pointer to PIO structure.
- *int_mode*: Configure the priority level and source type.
- *int_mask*: Configure which interrupt bits are activated.
- *callback*: Function called through the assembler interrupt handler.

Returns: None.

U32_T CSP_PIOGetStatus (CSP_PIO_T *const pio)

Description: Get PIO Pin Data Status

Inputs: **pio*: Pointer to PIO structure.

Returns: 32-bit value of pin status.

```
void CSP_PIOClear (CSP_PIO_T *const pio, U32_T pio_mask)
```

Description: Set the PIO to low level.

Inputs:

- **pio*: Pointer to PIO structure.
- *pio_mask*: Configure which pins are set to low level.

Returns: None.

```
void CSP_PIOSet (CSP_PIO_T *const pio, U32_T pio_mask)
```

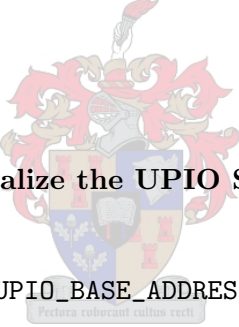
Description: Set the PIO to high level.

Inputs:

- **pio*: Pointer to PIO structure.
- *pio_mask*: Configure which pins are set to high level.

Returns: None.

A simple example to initialize the UPIO Structure



```
CSP_PIO_T* myPIO;
myPIO = ((CSP_PIO_T*) UPIO_BASE_ADDRESS);

//Initialize as outputs (nDISABLE = EDAC and CAN PowerSave, TimeSync, RTC and Temp)
CSP_PIOInit(myPIO,0x1B839,0x0);

//Set CAN powersave off (0) and EDAC off, EDAC is bit 1
CSP_PIOClear(myPIO,0x18);

/* Set EDAC Disabled (bit 1) and Temp_Clock = 1, and RTC_Clock = 1
   and Temp_E = and RTC_E = 1 */
CSP_PIOSet(myPIO,0x1B001);
```

C.4 Analog to Digital Conversion

Before any sampling from the Analog to Digital Converter can commence, it first has to be set up correctly.

When initializing the ADC Controller, the mode register needs to be set up.

Table C.8: ADC Mode Register [0x064]

31	30	29	28	27	26	25	24	
-	-	-	-	-	-	-	-	
23	22	21	20	19	18	17	16	
-	-	-	-	CONTCV	NBRCH[2:0]			
15	14	13	12	11	10	9	8	
STARTUPTIME[7:0]								
7	6	5	4	3	2	1	0	
-	STOPEN	-	PRLVAL[4:0]					

PRLVAL[4:0]: Preload Value

$$ADC_{clk} = \frac{CORECLK}{4 \times PRLVAL[4:0]} \quad (C.4.1)$$

For an ADC_{clk} value of 500 kHz, $PRLVAL[4:0] = 15$

STOPEN : Stop Enable

1: Conversion can be stopped by setting bit TEND

STARTUPTIME[7:0] : Startup Time

This value indicates the number of Master Clock periods to make $4\mu s$ needed for stabilization of the converter circuitry. For $CORECLK = 30 \text{ MHz} \Rightarrow STARTUPTIME[7:0] = 120$.

NBRCH[2:0] : Number of Conversions

We have 4 pins in total that needs to be converted: 2 x Current Monitoring (3.3 V and 2.5 V) 2 x Voltage Monitoring (3.3 V and 2.5 V) $NBRCH[2:0] = 011$

CONTCV : Continuous Conversion

1: ADC converts as much inputs as specified by NBRCH[2:0] in the order given by ADC_CMRR and repeats.

The DC Conversion mode register (shown in Table C.9) specifies in what order to do the conversion.

Table C.9: ADC DC Conversion Mode Register [0x068]

31	30	29	28	27	26	25	24
-	CV8[2:0]			-	CV7[2:0]		
23	22	21	20	19	18	17	16
-	CV6[2:0]			-	CV5[2:0]		
15	14	13	12	11	10	9	8
-	CV4[2:0]			-	CV3[2:0]		
7	6	5	4	3	2	1	0
-	CV2[2:0]			-	CV1[2:0]		

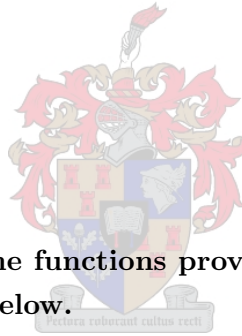
It is set up as:

CV1[2:0] = 000

CV2[2:0] = 001

CV3[2:0] = 010

CV4[2:0] = 011



A Short description of the functions provided by the software PIO controller, is described below.

CSP_ADC8CInit (CSP_ADC8C_T *const adc, U32_T mode)

Description: Switch on the clock, reset the registers and configure the ADC module mode and the PDC RX line.

Inputs:

- **adc*: Pointer to ADC structure.
- *mode*: Configure the ADC mode.

Returns: None.

CSP_ADC8CClose (CSP_ADC8C_T *const adc)

Description: Reset and switch off the clock.

Inputs: **adc*: Pointer to ADC structure.

Returns: None.

CSP_ADC8CConfigInterrupt (CSP_ADC8C_T *const adc, U32_T int_mode, U32_T int_mask, U32_T callback)

Description: Configure ADC Interrupts.

Inputs:

- **adc*: Pointer to ADC structure.
- *int_mode*: Configure the priority level and source type.
- *int_mask*: Configure which interrupt bits are activated.
- *callback*: Function called through the assembler interrupt handler.

Returns: None.

CSP_ADC8CEnable (CSP_ADC8C_T *const adc)

Description: Enable ADC.

Inputs: **adc*: Pointer to ADC structure.

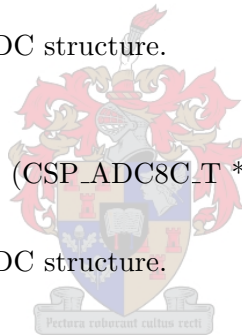
Returns: None.

CSP_ADC8CDisable (CSP_ADC8C_T *const adc)

Description: Disable ADC.

Inputs: **adc*: Pointer to ADC structure.

Returns: None.



CSP_ADC8CStartConversion (CSP_ADC8C_T *const adc, U16_T *data, U32_T order_conversion, U8_T nb_conversion)

Description: Configure ADC module, start conversions on 10 bit ADC in the chosen order and transfer the result directly to memory.

Inputs:

- **adc*: Pointer to ADC structure.
- **data*: Pointer to the address where the converted values will be stored.
- *order_conversion*: Configure the order of conversion (eg: CV1 to CV8).
- *nb_conversion*: Number of conversion [0-7] to be carried out.

Returns: None.

CSP_ADC8CStopConversion (CSP_ADC8C_T *const adc)

Description: Stop Conversion

Inputs: *adc: Pointer to ADC structure.

Returns: None.

C.5 Software LVDS Interface

A Short description of the functions provided by the software LVDS receiver, is described below.

InitLVDS (CSP_PIO_T *const PIO_data)

Description: Enable read and write access to the FIFO.

Inputs: *PIO_data: Pointer to PIO structure.

Returns: None.

isNEmptyLVDS (CSP_PIO_T *const PIO_data)

Description: Will return a false if FIFO buffer is empty.

Inputs: *PIO_data: Pointer to PIO structure.

Returns: True (buffer not empty) or False (buffer empty).

DoLVDS (CSP_PIO_T *const PIO_data)

Description: Returns a data byte from FIFO each time this function is called.

Inputs: *PIO_data: Pointer to PIO structure.

Returns: Data byte read from FIFO.

Sample code using LVDS functions to retrieve all data from buffer:

```
// Global Variables
CSP_PIO_T* myPIO;
myPIO = ((CSP_PIO_T*) UPIO_BASE_ADDRESS);

char lvds_buffer[100];
char lvds_buffer_count = 0;
char lvdsTemp;

// Inside timer
```

```
void timerInterrupt
{
    while (isNEmptyLVDS(myPIO));
    {
        lvdsTemp = DoLVDS(myPIO);
        if ((lvds_buffer_count % 2) == 1) lvds_buffer[lvds_buffer_count] = lvdsTemp;
        lvds_buffer_count++;
    }
}
```



Appendix D

LVDS to SPI Design

Using the LVDS data link to send large amounts of data to the OBC is possible, but since the transfers are done in software, it might keep the processor from successfully performing other tasks. The solution is to implement the LVDS bus with a DMA data path, allowing it to directly read/write from/into memory without any processor intervention.

By using only one of the nine available bits onboard the dual clock FIFO, and keeping the LVDS data as a serial data stream, the bits can be written directly into memory using SPI. Figure D.1 shows a schematical representation of the design.

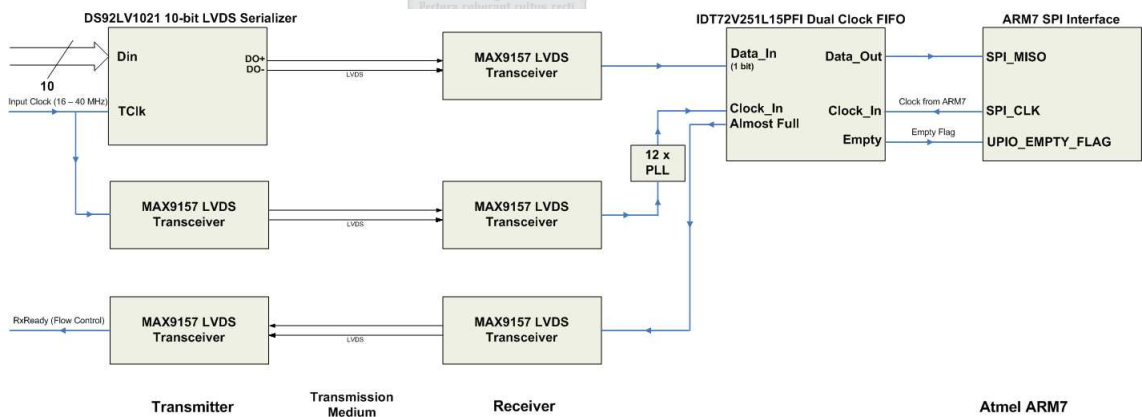


Figure D.1: Block diagram of LVDS to SPI link

Appendix E

LVDS Output

This OBC is developed with a primary (SPI) and secondary (LVDS) path for receiving large blocks of data from a camera or mass storage device. The transfer is usually initiated by sending a CAN message to the respective peripheral, which then starts sending the data over the dedicated data channel. Since SPI already has a two-way architecture, transfers can be initiated using this medium as well, instead of using a CAN message on peripherals not fitted with a CAN transceiver. A LVDS transmitter is also provided on the OBC for sending messages and data to a LVDS capable device that does not have the ability to receive these via CAN [6].



Two UPIO pins on the AT91SAM7A2 is used - for *clock* and *data out* respectively. This is then converted to LVDS voltage levels using a MAX9157 as shown in Figure E.1.

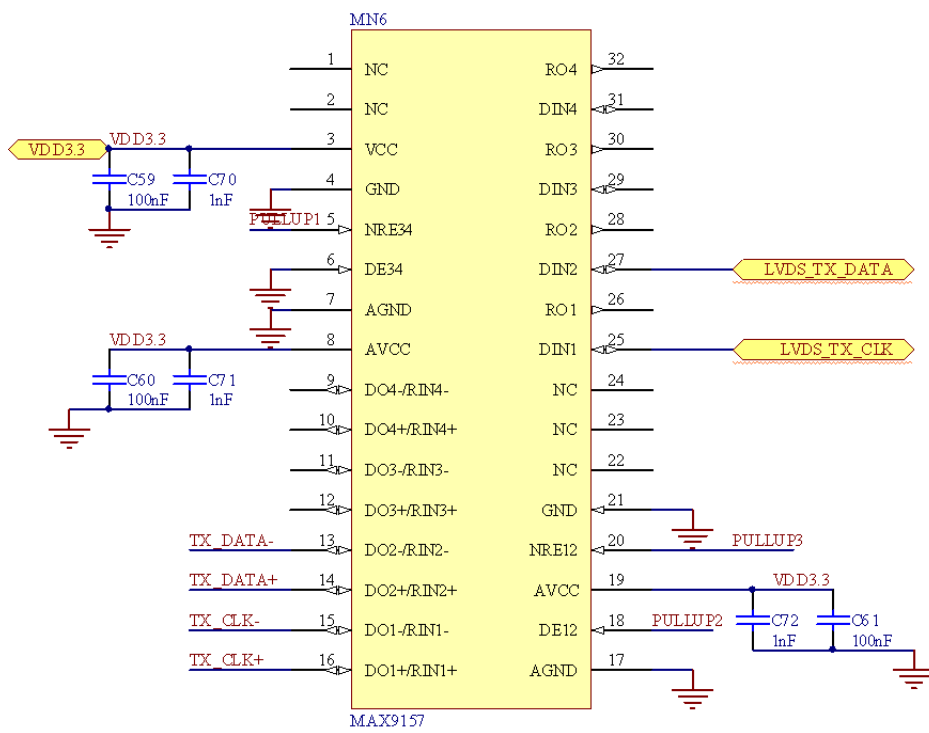


Figure E.1: LVDS Output using MAX9157

Appendix F

AT91SAM7A2

Complimentary Hardware Design

To properly reset the AT91SAM7A2 microprocessor, a reset signal of at least $1 \mu\text{s}$ must be maintained. To insure this, a MAX6315 reset controller (which applies a $1 \mu\text{s}$ reset signal) is used as shown in Figure F.1.

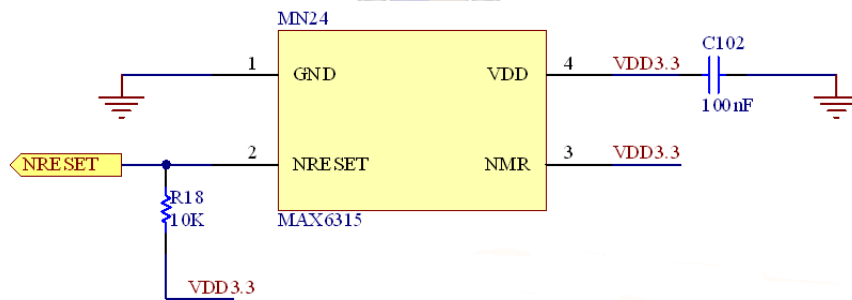


Figure F.1: Reset Controller for AT91SAM7A2

For each of the 32.768 kHz and 6 MHz oscillators, a crystal needs to be connected. The recommended capacitor for the 32.768 kHz and 6 MHz crystal is respectively 22 pF and 47 pF. According to the datasheet of the AT91SAM7A2 [3], a series resistor of 10 k Ω is necessary for the 32.768 kHz crystal.

The PLL has connections for an external RC filter (Figure F.2). The optimum response with a simple RC filter is obtained when: (Equation F.0.1 = 0.707).

$$0.4 < \sqrt{\left(\frac{K_0 \times I_P}{n \times (C_3 + C_4)}\right)} \times \frac{R \times C_4}{2} < 1 \quad (\text{F.0.1})$$

Where:

- K_0 is the PLL V_{CO} gain (typically 105.106 Hz / V)
- I_P is the peak current delivered by the charge pump into the filter (typically 250 mA)
- n is the division ratio of the divider (the PLL multiplication factor)
- stability can be improved with an additional capacitor C_3 . The value of C_3 must be chosen so that:

$$4 < \frac{C_4}{C_3} < 15 \quad (\text{F.0.2})$$

$$\sqrt{\left(\frac{K_0 \times I_P}{n \times (C_3 + C_4)}\right)} \leq \frac{\xi \times f_{CKR}}{5} \quad (\text{F.0.3})$$

Where:

- f_{CKR} is the PLL input frequency (6 MHz)
- ξ is the phase jitter (200 ps typically)

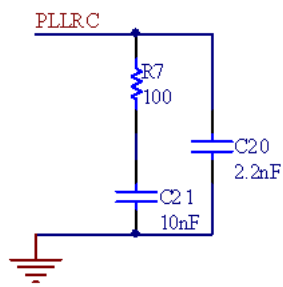


Figure F.2: PLL RC Filter Circuit

Appendix G

Actel FPGA Design

The complete pin assignment for the FPGA is shown in Table G.1. The VHDL code for the Hamming-based Error Detection and Correction module are also shown.

```
FUNCTION hamming_enc(DataOut : Std_Logic_Vector(0 to 15))
RETURN Std_Logic_Vector IS
  Variable CheckOut : Std_Logic_Vector(0 to 7);
BEGIN

  -- Check bit generator output
  CheckOut(0) :=      DataOut(0) xor DataOut(4) xor
                    DataOut(5) xor DataOut(6) xor
                    DataOut(7) xor DataOut(8) xor
                    DataOut(12) xor DataOut(13);
  CheckOut(1) :=      DataOut(1) xor DataOut(4) xor
                    DataOut(6) xor DataOut(8) xor
                    DataOut(9) xor DataOut(10) xor
                    DataOut(11) xor DataOut(14);
  CheckOut(2) := not (DataOut(0) xor DataOut(1) xor
                    DataOut(2) xor DataOut(3) xor
                    DataOut(5) xor DataOut(9) xor
                    DataOut(12) xor DataOut(15));
  CheckOut(3) := not (DataOut(0) xor DataOut(1) xor
```

Pin	Description
2..15	Data_ARM[0..13]
18..19	Data_ARM[14..15]
20..21	Address[19..20]
23	NWE_ARM (Not Write)
31..32	Error_ARM[0..1]
33	ncs1_in (Chip Select 1)
34	ncs2_in (Chip Select 2)
35	nDisable (Disable EDAC)
37	NOE_ARM (Not Read)
114..121	Parity[7..0]
124	NWE_SRAM
125	NOE_SRAM
137	Data_SRAM[15]
139	Data_SRAM[14]
140	Data_SRAM[13]
143..155	Data_SRAM[12..0]
158	npcs1
159	npcs3
160	npcs2
161	npcs0
163	ncs7
164	ncs6
165	ncs5
166	ncs3
167	ncs4
168	ncs2
169	ncs1
172	ncs0



Table G.1: Pin assignments for Actel ProAsic APA075

```

                                DataOut(2) xor DataOut(3) xor
                                DataOut(4) xor DataOut(10) xor
                                DataOut(13) xor DataOut(14));
    CheckOut(4) :=                DataOut(2) xor DataOut(5) xor
                                DataOut(7) xor DataOut(8) xor
                                DataOut(9) xor DataOut(10) xor
                                DataOut(11) xor DataOut(15);
    CheckOut(5) :=                DataOut(3) xor DataOut(6) xor
                                DataOut(7) xor DataOut(11) xor
                                DataOut(12) xor DataOut(13) xor
                                DataOut(14) xor DataOut(15);
    CheckOut(6) :=                DataOut(1) xor DataOut(2) xor
                                DataOut(4) xor DataOut(5) xor
                                DataOut(7) xor DataOut(8) xor
                                DataOut(11) xor DataOut(13);
    CheckOut(7) := not (DataOut(1) xor DataOut(2) xor
                        DataOut(3) xor DataOut(5) xor
                        DataOut(11) xor DataOut(12) xor
                        DataOut(13) xor DataOut(15));

    RETURN CheckOut;
END hamming_enc;

```

```

FUNCTION hamming_dec(DataIn : Std_Logic_Vector(0 to 15);
    CheckIn: Std_Logic_Vector(0 to 7))
RETURN Std_Logic_Vector IS
    variable Parity: Std_Logic_Vector(0 to 7); -- Generated parity
    variable Syndrome: Std_Logic_Vector(0 to 7); -- Syndrome
    variable DataCorr: Std_Logic_Vector(0 to 15);
    variable Err: Std_Logic_Vector(0 to 1);
BEGIN

    -- Check bit generator
    Parity(0) := DataIn(0) xor DataIn(4) xor
                DataIn(5) xor DataIn(6) xor

```

```

                                DataIn(7) xor DataIn(8) xor
                                DataIn(12) xor DataIn(13);
Parity(1) := DataIn(1) xor DataIn(4) xor
                                DataIn(6) xor DataIn(8) xor
                                DataIn(9) xor DataIn(10) xor
                                DataIn(11) xor DataIn(14);
Parity(2) := not (DataIn(0) xor DataIn(1) xor
                                DataIn(2) xor DataIn(3) xor
                                DataIn(5) xor DataIn(9) xor
                                DataIn(12) xor DataIn(15));
Parity(3) := not (DataIn(0) xor DataIn(1) xor
                                DataIn(2) xor DataIn(3) xor
                                DataIn(4) xor DataIn(10) xor
                                DataIn(13) xor DataIn(14));
Parity(4) := DataIn(2) xor DataIn(5) xor
                                DataIn(7) xor DataIn(8) xor
                                DataIn(9) xor DataIn(10) xor
                                DataIn(11) xor DataIn(15);
Parity(5) := DataIn(3) xor DataIn(6) xor
                                DataIn(7) xor DataIn(11) xor
                                DataIn(12) xor DataIn(13) xor
                                DataIn(14) xor DataIn(15);
Parity(6) := DataIn(1) xor DataIn(2) xor
                                DataIn(4) xor DataIn(5) xor
                                DataIn(7) xor DataIn(8) xor
                                DataIn(11) xor DataIn(13);
Parity(7) := not (DataIn(1) xor DataIn(2) xor
                                DataIn(3) xor DataIn(5) xor
                                DataIn(11) xor DataIn(12) xor
                                DataIn(13) xor DataIn(15));

-- Syndrome bit generator
Syndrome(0) := Parity(0) xor CheckIn(0);
Syndrome(1) := Parity(1) xor CheckIn(1);
Syndrome(2) := Parity(2) xor CheckIn(2);
Syndrome(3) := Parity(3) xor CheckIn(3);

```

```

Syndrome(4) :=      Parity(4) xor CheckIn(4);
Syndrome(5) :=      Parity(5) xor CheckIn(5);
Syndrome(6) :=      Parity(6) xor CheckIn(6);
Syndrome(7) :=      Parity(7) xor CheckIn(7);

-- Bit corrector
DataCorr          := DataIn;                                -- uncorrected default

case Syndrome is                                           -- bit error correction
  when "10110000" =>                                       -- single data error
    DataCorr(0)    := not DataIn(0);
    Err            := "01";
  when "01110011" =>                                       -- single data error
    DataCorr(1)    := not DataIn(1);
    Err            := "01";
  when "00111011" =>                                       -- single data error
    DataCorr(2)    := not DataIn(2);
    Err            := "01";
  when "00110101" =>                                       -- single data error
    DataCorr(3)    := not DataIn(3);
    Err            := "01";
  when "11010010" =>                                       -- single data error
    DataCorr(4)    := not DataIn(4);
    Err            := "01";
    when "10101011" =>                                       -- single data error
      DataCorr(5)  := not DataIn(5);
      Err          := "01";
  when "11000100" =>                                       -- single data error
    DataCorr(6)    := not DataIn(6);
    Err            := "01";
    when "10001110" =>                                       -- single data error
      DataCorr(7)  := not DataIn(7);
      Err          := "01";
    when "11001010" =>                                       -- single data error
      DataCorr(8)  := not DataIn(8);
      Err          := "01";

```



```

        when "01101000" => -- single data error
            DataCorr(9) := not DataIn(9);
            Err         := "01";
        when "01011000" => -- single data error
            DataCorr(10) := not DataIn(10);
            Err          := "01";
        when "01001111" => -- single data error
            DataCorr(11) := not DataIn(11);
            Err          := "01";
        when "10100101" => -- single data error
            DataCorr(12) := not DataIn(12);
            Err          := "01";
        when "10010111" => -- single data error
            DataCorr(13) := not DataIn(13);
            Err          := "01";
        when "01010100" => -- single data error
            DataCorr(14) := not DataIn(14);
            Err          := "01";
        when "00101101" => -- single data error
            DataCorr(15) := not DataIn(15);
            Err          := "01";
    when "10000000" | "01000000" | "00100000" |
        "00010000" | "00001000" | "00000100" |
        "00000010" | "00000001" => -- single parity error
        Err         := "01";
    when "00000000" => -- no errors
        Err         := "00";
    when others => -- multiple errors
        Err         := "11";
    end case;

return (DataCorr & Err);

END hamming_dec;

```

Bibliography

- [1] Allie, S., PCB Mechanical Development Specifications , Sun Space and Information Systems, 2006. 26
- [2] Atmel Application Group, AT91SAM7A2 Software Library, Atmel, 2004. 43
- [3] Atmel Corporation, AT91SAM7A2 Datasheet, Atmel Electronic Document, 2004. x, xii, 5, 9, 11, 13, 14, 21, 29, 91
- [4] Atmel Corporation, ARM7 TDMI Programmer's Model, Atmel Electronic Document, 1999. 8
- [5] Barnard, A., Feasibility of using an ARM Processor in a Micro Satellite On-board Computer, University of Stellenbosch, 2001. 10
- [6] Dreijer, G., The evaluation of an ARM-based on-board Computer for a low earth orbit satellite, University of Stellenbosch, 2003. 89
- [7] Grobler, H., Aspects affecting the design of a low earth orbit satellite on-board computer, University of Stellenbosch, 2000. 23
- [8] Moon, T.K., Error Correction Coding, Wiley, 2005. 33, 35
- [9] NASA Office of Logic Design, A error correcting code for critical memory applications, <http://www.klabs.org/richcontent/Papers/Hans/Edac/>, 2000. 35

- [10] National Semiconductor, LM70 Digital Temperature Sensor Datasheet, National Semiconductor, 2006. 49, 60
- [11] Peterson, L.L., Computer Networks: A systems approach, Morgan Kaufmann, 2003. 20
- [12] RAD University, Hamming Code: Explained, http://www2.rad.com/networks/1994/err_con/hamming.htm, 1994. 34
- [13] ST Micro Electronics, M41ST95W Datasheet, ST, 2004. x, 50, 52, 53, 60
- [14] Tsai, J.J.P., Distributed Real-Time Systems, Wiley, 1996. 40
- [15] Wertz, J.R., Space Mission Analysis and Design, Space Technology Library, 1999. x, 2, 12
- [16] Zwolinski, M., Digital System Design with VHDL, Prentice Hall, 2003. 33

