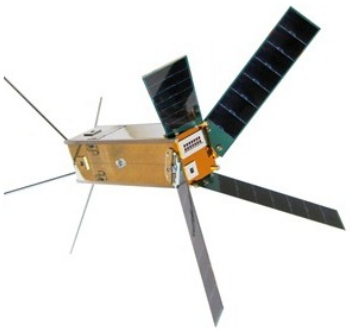
Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2012

MSc THESIS

Fault-Tolerant On-Board Computer Software for the Delfi-n3Xt Nanosatellite

Alexander Franciscus Cornelis (Sander) van den Berg



CE-MS-2012-03

Abstract

Fault-tolerant On-Board Computer (OBC) software for the Delfi-n3Xt nanosatellite is needed in order to minimize the risk of failures that may occur while the satellite is operating in space. Failures may be OBC specific, but failures that affect the state of the entire satellite and influence the health of the data bus may occur as well. Some failures that may occur on the I^2C data bus can have a very large impact on the health of the satellite. The failure cases in which the I^2C data line or I^2C clock line is being pulled low for a longer period of time make communication over the I^2C bus impossible. The I^2C bus recovery mode that is implemented in the OBC, together with the I^2C recovery mechanism that applies to the whole satellite, provides a way to resolve failure cases like these. The failure cases on the I^2C bus with less disastrous impacts may result in data inconsistencies and time-outs and are handled by the OBC as well. The I^2C data bus performance analysis for Delfi-n3Xt shows a bit error rate of at most $4 \cdot 10^{-9}$, which fulfills the requirement that specifies that the bit error rate must be 10^{-6} or less.
Apart from failures on the I^2C data bus, failures may occur internally in the OBC hardware or software. Since the OBC controls the whole satellite, a permanent failure in the OBC hardware or software may result in a non-functional satellite. The OBC software is designed and implemented in such a way that it can not become in an undefined state for longer than 8 seconds. Besides that, the OBC assures that transfers over the I^2C bus never take longer than 30ms. This improves reliablity and performance. Furthermore, clever routines that save flash memory erase cycles were designed and developed in order to increase the lifetime of the flash memory.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Fault-Tolerant On-Board Computer Software for the Delfi-n3Xt Nanosatellite

Including I^2C bus failure and performance analysis

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Alexander Franciscus Cornelis (Sander) van den Berg
born in Noordwijkerhout, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Fault-Tolerant On-Board Computer Software for the Delfi-n3Xt Nanosatellite

by Alexander Franciscus Cornelis (Sander) van den Berg

Abstract

Fault-tolerant On-Board Computer (OBC) software for the Delfi-n3Xt nanosatellite is needed in order to minimize the risk of failures that may occur while the satellite is operating in space. Failures may be OBC specific, but failures that affect the state of the entire satellite and influence the health of the data bus may occur as well.

Some failures that may occur on the I^2C data bus can have a very large impact on the health of the satellite. The failure cases in which the I^2C data line or I^2C clock line is being pulled low for a longer period of time make communication over the I^2C bus impossible. The I^2C bus recovery mode that is implemented in the OBC, together with the I^2C recovery mechanism that applies to the whole satellite, provides a way to resolve failure cases like these. The failure cases on the I^2C bus with less disastrous impacts may result in data inconsistencies and time-outs and are handled by the OBC as well. The I^2C data bus performance analysis for Delfi-n3Xt shows a bit error rate of at most $4 \cdot 10^{-9}$, which fulfills the requirement that specifies that the bit error rate must be 10^{-6} or less.

Apart from failures on the I^2C data bus, failures may occur internally in the OBC hardware or software. Since the OBC controls the whole satellite, a permanent failure in the OBC hardware or software may result in a non-functional satellite. The OBC software is designed and implemented in such a way that it can not become in an undefined state for longer than 8 seconds. Besides that, the OBC assures that transfers over the I^2C bus never take longer than 30ms. This improves reliablity and performance. Furthermore, clever routines that save flash memory erase cycles were designed and developed in order to increase the lifetime of the flash memory.

Laboratory : Computer Engineering
Codenumber : CE-MS-2012-03

Committee Members :

Advisor:	Dr. ir. A.J. van Genderen, CE, TU Delft
Advisor:	Ir. J. Bouwmeester, SSE, TU Delft
Chairperson:	Dr. K.L.M. Bertels, CE, TU Delft

Dedicated to my beloved family and friends, and in particular to my parents for their endless love and support

Contents

List of Figures **x**

List of Tables **xi**

Acknowledgements **xiii**

1 Introduction **1**
1.1 Problem Statement . . . 2
1.2 Thesis Objectives . . . 3
1.3 Thesis Organization . . . 4

2 Background **7**
2.1 Delfi-n3Xt Mission . . . 8
2.1.1 Educational Objective . . . 8
2.1.2 Technology Demonstration Objective . . . 8
2.1.3 Nanosatellite Bus Development Objective . . . 9
2.1.4 Advancements . . . 9
2.2 Delfi-n3Xt Payloads and Subsystems . . . 10
2.2.1 Micro Propulsion Payload . . . 10
2.2.2 Tranceiver Payload . . . 11
2.2.3 Communications Subsystem . . . 12
2.2.4 Attitude Determination and Control Subsystem . . . 12
2.2.5 Electrical Power Subsystem . . . 13
2.2.6 Structure, Mechanisms and Thermal Control Subsystems . . . 14
2.2.7 Command and Data Handling Subsystem . . . 15
2.3 Delfi-n3Xt CDHS Hardware . . . 16
2.3.1 On-Board Computer Hardware . . . 17
2.3.2 Delfi Standard System Bus Hardware . . . 17
2.4 I^2C Communication Basics . . . 18
2.4.1 Hardware Setup . . . 18
2.4.2 Start and Stop Conditions . . . 19
2.4.3 Write Operation . . . 19
2.4.4 Read Operation . . . 20
2.4.5 Sampling vs. Edge-Triggered Interrupts . . . 21
2.4.6 Data Transfer Example . . . 21
2.4.7 Clock Stretching . . . 23
2.4.8 Data Transfer Speeds . . . 23
2.5 Summary . . . 24

3 I²C Bus Analysis **25**
3.1 I²C Bus Failure Analysis . . . 25
3.1.1 Start Condition Failures . . . 25
3.1.2 Slave Request Failures . . . 26
3.1.3 Read/Write Bit Failures . . . 27
3.1.4 Slave Acknowledgement Failures . . . 28
3.1.5 Data Byte Transfer Failures . . . 29
3.1.6 Stop Condition Failures . . . 30
3.1.7 Slave Device Missing Clock Pulses . . . 31
3.1.8 Data Line Pulled Low Indefinitely . . . 32
3.1.9 Clock Line Pulled Low Indefinitely . . . 33
3.1.10 I²C Failures Summary . . . 33
3.2 I²C Bus Performance Analysis . . . 35
3.2.1 Bit Error Rate . . . 36
3.2.2 Test Setup . . . 36
3.2.3 Test Procedure . . . 37
3.2.4 Software Verification . . . 38
3.2.5 Performance Results . . . 39
3.3 Summary . . . 40

4 On-Board Computer Software Design **41**
4.1 OBC Software Requirements . . . 41
4.1.1 Subsystem Communication Requirements . . . 41
4.1.2 Fault-Tolerant Software Requirements . . . 43
4.1.3 Telecommanding Requirements . . . 44
4.1.4 Data Acquisition Requirements . . . 46
4.1.5 Monitoring Requirements . . . 48
4.2 OBC Software Architecture . . . 48
4.2.1 OBC Software Layers . . . 48
4.2.2 OBC Module Overview . . . 49
4.3 OBC Service Layer Software Design . . . 51
4.3.1 Clock Source Module . . . 52
4.3.2 Programmable Interval Timers . . . 53
4.3.3 Flash Memory Controller . . . 56
4.3.4 Analog to Digital Converter . . . 59
4.3.5 I²C Controller . . . 61
4.3.6 Watchdog Timer . . . 65
4.4 OBC Application Layer Software Design . . . 68
4.4.1 Boot Mode . . . 69
4.4.2 Delay Mode . . . 72
4.4.3 Deployment Mode . . . 73
4.4.4 Main Mode . . . 75
4.4.5 I²C Bus Recovery Mode . . . 84
4.5 Summary . . . 85

5 On-Board Computer Software Implementation **87**
5.1 Service Layer Software . . . 87
5.1.1 Clock Source Module . . . 87
5.1.2 Programmable Interval Timers . . . 89
5.1.3 Flash Memory Controller . . . 91
5.1.4 Analog to Digital Converter . . . 94
5.1.5 I^2C Controller . . . 96
5.1.6 Watchdog . . . 101
5.2 Application Layer Software . . . 103
5.2.1 Boot Mode . . . 103
5.2.2 Delay Mode . . . 107
5.2.3 Deployment Mode . . . 108
5.2.4 Main Mode . . . 110
5.2.5 I^2C Recovery Mode . . . 114
5.3 Summary . . . 115

6 Conclusions **117**
6.1 Summary . . . 117
6.2 Contributions . . . 119
6.3 Future Work . . . 120

Bibliography **124**

A OBC Source Code Directory Listing **125**

List of Figures

1.1 *Architectural overview of Delfi-n3Xt on the subsystem level* 1

2.1 *Two nanosatellites that are part of the Delfi program* 7
2.2 *Tranceiver payload from ISIS BV* 11
2.3 *Architectural overview of the Delfi-n3Xt Communications Subsystem* . . . 12
2.4 *Prototype of the ADCS orthogonal reaction wheel assembly* 13
2.5 *Architectural overview of the Global EPS* 13
2.6 *Example of the rod system of Delfi-C^3* 14
2.7 *Functional overview of the DSSB protection circuit* 15
2.8 *Delfi-n3Xt On-Board Computer hardware PCB layout* 16
2.9 *I^2C single-master, multiple-slave setup* 18
2.10 *Start condition and stop condition* 19
2.11 *Data transfer from master to slave* 20
2.12 *Data transfer from slave to master* 21
2.13 *Example of a data transfer over the I^2C lines* 22
2.14 *Example where the slave device stretches the clock* 23

3.1 *Start condition with SDA high at time t_1 and SDA low at time t_2* 25
3.2 *Bit flips in a start condition on the I^2C lines between times t_1 and t_2* . . 26
3.3 *Stop condition with SDA low at time t_1 and SDA high at time t_2* 30
3.4 *Bit flips in a stop condition on the I^2C lines between times t_1 and t_2* . . . 31
3.5 *Switch to pull low the SDA line intentionally* 38

4.1 *High level architectural overview of the OBC* 49
4.2 *Module overview of the OBC service layer software* 50
4.3 *Module overview of the OBC application layer software* 51
4.4 *Activity flow for configuring the MSP430 clock system of the OBC* 53
4.5 *16-bit timer A operating in up mode and generating interrupts* 54
4.6 *16-bit timer A operating in up/down mode and generating interrupts* . . . 54
4.7 *Activity flow for the configuration of a timer* 55
4.8 *Segmentation of the MSP430F1611 flash memory* 56
4.9 *Activity flow for reading data from flash memory* 57
4.10 *Activity flow for writing data to the flash memory* 58
4.11 *Activity flow for erasing (part of) the flash memory* 58
4.12 *Temperature sensor transfer function for the on-chip temperature sensor* . 59
4.13 *Activity flow for the ADC to read out temperature* 61
4.14 *Activity flow for initializing the I^2C peripheral in master mode* 62
4.15 *Activity flow for reading data from the I^2C bus in master-receiver mode* . 62
4.16 *Activity flow for writing data on the I^2C bus in master-transmitter mode* 63
4.17 *Activity flow for the initialization procedure for an I^2C slave device* . . . 64
4.18 *Activity flow for handling I^2C interrupts for reading data in slave mode* . 65
4.19 *Activity flow for handling I^2C interrupts for writing data in slave mode* . 65

4.20 *Activity flow for the transitions between operational modes* 68
4.21 *Activity flow for the execution of the boot mode* 69
4.22 *The content of the encoded boot counter that represents a value of 0* . . . 71
4.23 *The content of the encoded boot counter that represents a value of 1* . . . 71
4.24 *The content of the encoded boot counter that represents a value of 256* . . 71
4.25 *Activity flow for the delay that may be needed before deployment* 73
4.26 *Activity flow for deploying the solar panels using the primary resistors* . . 74
4.27 *Activity flow for deploying the solar panels using the secondary resistors* . 75
4.28 *The last part of the deployment mode* 76
4.29 *Activity flow of the main loop* 77
4.30 *Activity flow for checking and executing a telecommand* 83
4.31 *Activity flow for the I^2C bus recovery mode* 84

5.1 *The BCSCTL1 register that is used to configure the clock system* 87
5.2 *The BCSCTL2 register that is used to configure the clock system* 88
5.3 *The TACCTL register that is used to set the timer threshold value* 89
5.4 *The TACTL register that is used to configure timer A* 90
5.5 *The FCTL2 register that is used to initialize the flash timing generator* . . 91
5.6 *The FCTL1 register that is used to control the flash memory controller* . . 93
5.7 *The FCTL3 register that is used to control the flash memory controller* . . 93
5.8 *The ADC10CTL0 register that is used to configure the A/D converter* . . 94
5.9 *The ADC10CTL1 register that is used to configure the A/D converter* . . 95
5.10 *The ADC10MEM register that is used to store the A/D conversion result* 96
5.11 *The U0CTL register that configures the MSP430F1611 USART peripheral* 97
5.12 *The I2CTCTL register that configures and drives the I^2C peripheral* . . . 97
5.13 *The I2CPSC, I2CSCLH and I2CSCLL registers that configure the bus speed* 98
5.14 *The I2CIFG register that holds the interrupt status flags* 99
5.15 *The I2COA register that holds the 7-bit slave address* 100
5.16 *The I2CIE register that enables or disables I^2C interrupts* 100
5.17 *The WDTCTL register that is used to configure the watchdog timer* 101

List of Tables

2.1 $T^3\mu$PS characteristics and specifications 10
2.2 Delfi-n3Xt tranceiver payload specifications 11

3.1 I^2C failures with their causes, impacts and resolve steps 35

4.1 Selectable watchdog expiration times using the 32768 Hz clock source . . 67
4.2 I^2C recovery reset sequence and timings 86

Acknowledgements

First of all, I would like to thank Jasper Bouwmeester and Arjan van Genderen for providing me this thesis assignment. I really appreciate their supervision, advises and especially their time and efforts they have put into reading and reviewing this thesis and all the developed source code.

Furthermore I want to thank the entire Delfi-n3Xt team as well, with in particular the embedded software engineers and electrical engineers. I really liked the fruitful discussions during the Delfi-n3Xt progress meetings and the one-to-one conversations with some of the team members. They gave me the opportunity to learn a lot more about embedded systems and electronics in general. Besides that, I am very grateful that working on Delfi-n3Xt offered me opportunities to gain experiences in working with real flight hardware and software in a cleanroom environment. It is an experience that will definitely help me in my future professional carreer in the space industry.

Those I will not forget to mention are my relatives, and in particular my parents. They always motivated me to work and study hard to achieve my goals. They supported me in all possible ways in order to successfully complete my academic carreer. The writing of this thesis was definitely not possible without their endless love and support. Last but not least, I thank my friends for having a great time with me during my whole academic carreer. The good times with them definitely gave me the power and strength to go on with my studies.

Alexander Franciscus Cornelis (Sander) van den Berg
Delft, The Netherlands
August 28, 2012

1 Introduction

Delf-n3Xt is a nanosatellite that is under development at Delft University of Technology. It will be launched in September 2012 and it consists of various subsystems that together form the complete satellite. Part of the Delfi-n3Xt satellite is the On-Board Computer (OBC). The OBC is the central control unit of the satellite. It controls the operational mode of the satellite and it initiates all data transfers that take place over the satellite data bus [4].

The OBC, together with the Delfi Standard System Bus (DSSB), is part of the Command and Data Handlig Subsystem (CDHS) of the satellite. Besides the CDHS, various other subsystems are present, as will become clear in the next chapter. The OBC, combined with the DSSB, is able to autonomously control and synchronize all the subsystems that are part of the satellite. The architectural overview of Delfi-n3Xt on the subsystems level is shown in Figure 1.1

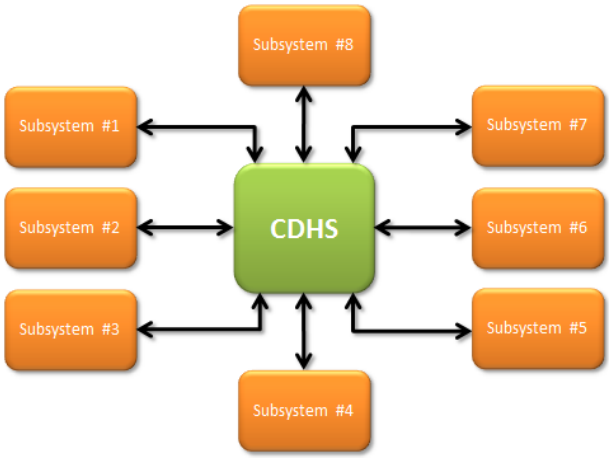


Figure 1.1: *Architectural overview of Delfi-n3Xt on the subsystem level*

The internal communication between the OBC and all the other subsystems within Delfi-n3Xt take place over the I^2C data bus. Bidirectional communication is possible, but the OBC always initiates the data transfers. The OBC transmits data to a subsystem in order to let the requested subsystems execute a desired task. The data that is received from the subsystems is used to form the telemetry data frame. This telemetry data frame consists of payload data [12] and housekeeping data that must be send to the earth by one of the radios that are present in Delfi-n3Xt. Each radio that is present in Delfi-n3Xt is considered as a subsystem on its own.

From an architectural point of view, the DSSB is part of the CDHS. Physically the DSSB circuitry is present on all the subsystems. The DSSB circuitry consists of a control unit that can be commanded by the OBC over the data bus. The DSSB control unit can be commanded to turn on or off the subsystem on which it is present. These on/off switching functionalities of the DSSB circuitry, together with the commanding functionalities of the OBC, form the autonomous control capabilities of the satellite.

The remaining part of this chapter is organized in three sections. In Section 1.1 the problem statement is defined. The main goal and objectives of this thesis are described in Section 1.2. Finally, in Section 1.3, a detailed overview of this thesis is outlined.

1.1 Problem Statement

Because of its central position, the OBC is a critical system within the satellite. A permanent failure in the OBC means the end of the Delfi-n3Xt mission. This is because the OBC is responsible for the data acquisition of all the payload data and housekeeping data from all the powered subsystems. When the OBC is unable to acquire these data and forward it to one of the radios, no data will be transmitted from the satellite to the ground station. Acquiring data and analyzing these data on the ground is what the mission is all about and it is mendatory in order to succeed the mission.

As already mentioned earlier, data transfers take place over the I^2C data bus [24]. When the I^2C data bus fails, communication between the OBC and the other subsystems is not possible anymore. This means that payload data and housekeeping data of the other subsystems can not be acquired anymore by the OBC. Therefore the I^2C data bus is also a critical part of the satellite system. There are various kind of failures that may occur on the I^2C bus, each having one or more different causes. Due to this, the satellite system is not failure free. A fault-tolerant implementation ensures that failures like these are properly handled and solved by some kind of recovery procedure.

Other critical systems within the satellite are the electrical power system [26] and the radio [30]. The electrical power system makes sure the satellite can be powered properly and the radio is needed for communications between the satellite and the ground station. A permanent failure in one of these systems means, just as in the case of the OBC, end of the Delfi-n3Xt mission. This is not acceptable, so each of these critical systems must be redundant [9] in order to avoid a single point of failure (SPOF). This has consequences for the OBC software, since the OBC is also a critical system and hence it must be redundant. To prevent SPOFs in the OBC, a second OBC is present. The OBC software must be designed and implemented such that when the first OBC fails in the execution of its operations, the second OBC takes over. The OBC is also responsible for switching between the radios when one of the radios fail. To handle the above described failures, a fault-tolerant implementation of the OBC software is needed. Also a thorough understanding and analysis of the possible I^2C failure cases is necessary. The need for redundancy and a fault-tolerant OBC introduces a certain degree of complexity in the system.

1.2 Thesis Objectives

In the previous section, the various problems that exists in designing and implementing a fault-tolerant OBC are described. Compared to the Delfi-C^3 nanosatellite, several improvements can be made in the CDHS for Delfi-n3Xt. These improvements will become clear in the following chapters.

The main goal of this thesis is to deliver a fault-tolerant flight software implementation for the Delfi-n3Xt OBC that shows improvements in the performance and reliability of the CDHS compared to Delfi-C^3 [4]. In order to manage this main goal, several objectives have been defined.

The following objectives are part of this thesis:

- **OBC software requirements analysis.**
  All the requirements for the OBC software must be listed and extensively analysed. This is necessary in order to understand the needs of the software that must be designed and implemented. There is already a requirements and configuration item list [19] in which most of the OBC software requirements are stated. The OBC software requirements must be rationalized and if needed new requirements must be added to the requirement list.

- **I^{2}C bus performance analysis.**
  An important improvement compared to Delfi-C^3 will be the improvement in performance of data transfers over the I^{2}C bus. An extensive I^{2}C bus performance analysis must be performed in order to judge whether or not improvements have been made compared to Delfi-C^3. Besides the improvements compared to the Delfi-C^3 satellite, the I^{2}C bus performance analysis will show results on the bit error rate of the data bus. The results show whether or not the performance of the I^{2}C bus meets the requirement that specifies the maximum allowable bit error rate.

- **I^{2}C bus failure analysis.**
  The health of the Delfi-n3Xt satellite highly depends on the health of the I^{2}C bus. As already discussed in the previous section, critical failures may occur when the operation of the I^{2}C bus fails. In order to get insights in the different failure cases that may occur on the I^{2}C bus, an extensive I^{2}C bus failure analysis must be performed. The result of the analysis is a list with the most significant failure cases that may occur on the I^{2}C bus, including their causes, impacts and resolve steps.

- **OBC service layer software design**.
  The OBC service layer software must consist of software modules that drive the needed microprocessor hardware peripherals. The design of the service layer software must define these service layer software modules together with their functionalities and activity flows. Each service layer software module must be mapped to the corresponding hardware peripheral of the microprocessor.

- **OBC service layer software implementation**.
  The OBC service layer software modules must implement the functionalities that are defined and described during the design of the OBC service layer software. These software modules must contain a low level implementation that is close to the OBC hardware. The low level procedures that are part of the service layer software modules must execute read and write operations from and to the registers of the microprocessor. Configuration of the microprocessor registers will result in the execution of the desired actions by the hardware peripherals of the microrprocessor.

- **OBC application layer software design**.
  The OBC application layer software must consist of software modules that make use of the OBC service layer software modules in order to successfully execute higher level tasks that control the data flow within the CDHS and the rest of the satellite. The design of the application layer software must define the application layer software modules together with their functionalities and activity flows.

- **OBC application layer software implementation**.
  The software modules that are part of the OBC application layer software must contain a higher level implementation. The application layer software modules must implement the operational modes of the satellite and the data flow control within the satellite system. These application layer software modules will execute procedures that are implemented in the service layer software modules.

- **Software testing**.
  The OBC software must be extensively tested. In the beginning of the project this will consist of unit tests in order to test the functionality of the service layer software modules independently. Later on when the application layer software modules are ready the testing consist of data flow tests in order to check the correctness of the data that flows in and out of the OBC. In the last phase of the project several integration tests will be performed in which other subsystems of the satellite are involved.

1.3 Thesis Organization

In this chapter, the various problems that exists in designing and implementing a fault-tolerant CDHS and the objectives for this thesis were defined and described. This thesis is divided in five chapters. The thesis is organized in such a way that the reader is taken along the analysis, design and implementation phases of the fault-tolerant OBC software development process.

Chapter 2 describes all the required background information about Delfi-n3Xt. In this chapter descriptions are given about the Delfi-n3Xt mission, the Delfi-n3Xt payloads and subsystems and the Delfi-n3Xt CDHS hardware. Besides this background information about Delfi-n3Xt, it is also explained how communication over the I^2C bus works.

Chapter 3 presents the complete I^2C bus failure analysis in which possible I^2C bus failure cases are described together with their causes, impacts and resolve steps. The I^2C bus performance analysis that is performed on engineering models of the CDHS hardware is also given in this chapter.

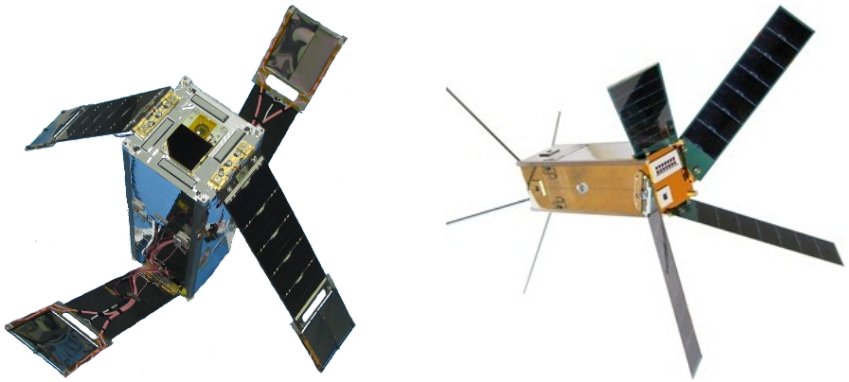
The design of the OBC software is explained in Chapter 4. In this chapter the OBC software requirements are listed and rationalized and an overview of the OBC software architecture is given. Furthermore, the OBC service layer software design and OBC application layer software design are discussed and worked out. These designs describe the functionalities of all the defined software modules.

Chapter 5 describes the implementation of all the OBC software modules. This includes the software modules for the service layer and the application layer. The implementation of the service layer software modules shows how the registers of the microprocessor are manipulated in order to let the hardware peripherals of the microprocesser execute the desired tasks. The implementation of the application layer software modules shows how the different operational modes of the satellite are implemented and how data flows within the satellite are initiated.

Conclusions on the software development process and the developed OBC software are drawn in Chapter 6. The chapter also gives a summary of this thesis and a description of my personal contributions to the OBC software and other parts of the Delfi-n3Xt project. Finally, the future work that is needed to complete the OBC software is described.

2 Background

The Delfi program is a nanosatellite development line at the faculty of Aerospace Engineering, department of Space Systems Engineering, at Delft University of Technology. It currently consists of two nanosatellite projects; the Delfi-C^3 project and the Delfi-n3Xt project (renderings shown in Figure 2.1).



(a) Delfi-C^3 (b) Delfi-n3Xt

Figure 2.1: *Two nanosatellites that are part of the Delfi program*

Delfi-C^3 was the first nanosatellite in the Netherlands and the Delfi-C^3 project has been started in 2004 by students and staff members [10]. It is a 3-unit CubeSat and it was launched on 28 April 2008 from India aboard a PSLV-C9 rocket. The Delfi-C^3 nanosatellite is shown in Figure 2.1a. Delfi-n3Xt is the successor of Delfi-C^3 and the project was started in November 2007. Once again, it is also designed by students and supported by staff members. Just like Delfi-C^3, Delfi-n3Xt is a 3-unit CubeSat and it is planned for launch in September 2012. This nanosatellite is shown in Figure 2.1b.

In this chapter, the background information that is needed to understand the other chapters of this thesis is described. First, the Delfi-n3Xt mission is discussed in Section 2.1 by describing the mission objectives and the advancements. The Delfi-n3Xt payloads and subsystems are described in Section 2.2. The background information that is needed on the CDHS hardware is given in Section 2.3. This includes the OBC and DSSB hardware. In Section 2.4, the I^{2}C communication basics are explained. Section 2.5 summarizes this chapter.

2.1 Delfi-n3Xt Mission

The Delfi-n3Xt mission consists of the three general Delfi objectives; the educational objective, technology demonstration objective and nanosatellite bus development objective. The mission statement for the Delfi-n3Xt mission [3] is as follows:

"Delfi-Next shall be a reliable triple-unit CubeSat of TU Delft which implements substantial advances in 1 subsystem with respect to Delfi-C3 and allows technology demonstration of 2 payloads from external partners from 2012 onwards".

Substantial advancements have been made in the Attitude and Determination Control Subsystem (ADCS). This subsystem, together with the other subsystems and the two payloads from the external partners are described in Section 2.2. The remainder of this Section describes the three general Delfi objectives and the advancements of the Delfi-n3Xt mission with respect to the Delfi-C^3 mission. The educational objective is described in Section 2.1.1. In Section 2.1.2 the technology demonstration objective is discussed. The nanosatellite bus development objective is described in Section 2.1.3 and the advancements with respect to Delfi-C^3 are given in Section 2.1.4.

2.1.1 Educational Objective

Since the Delfi nanosatellites are designed and developed by students at Delft University of Technology, one of the obvious objectives of the Delfi program is education. The Delfi program gives TU Delft students, as well as international students and students from other educational institutions, the opportunity to work on a real satellite design and engineering project. Students working on Delfi-n3Xt will improve their skills in systems engineering, research, teamwork, scientific writing, communication, mechanics, astrodynamics, control and simulation, electronics and software.

The goal of this objective is to optimally prepare students for any further careers in the space industry. The students will acquire hands-on experience with all aspects of the development of a real spacecraft.

2.1.2 Technology Demonstration Objective

Another aim of the Delfi program is to perform technology demonstrations and/or qualifications of micro-technologies that are designed for space applications. Onboard Delfi-n3Xt, there will be two experimental payloads:

- A micro-propulsion system developed by TNO in cooperation with Delft University of Technology and University of Twente.
- An in-orbit configurable, high-efficient transceiver platform developed by ISIS BV in cooperation with Delft University of Technology and SystematIC BV.

Whether or not these payload experiments work as expected can be determined by analyzing the telemetry data of the satellite that is received at the ground station once the satellite is fully operational and flying in space.

2.1.3 Nanosatellite Bus Development Objective

One other important objective is the development of the nanosatellite bus. The nanosatellite bus is an important part of the satellite since it connects all the subsystems with each other. The aim is to design the bus system in such a way that it is capable for more advanced technology demonstrations that require more data throughput and more power. If the satellite bus is capable of handling higher data rates and more data throughput, Delfi can create spin-offs for the space sector for more advanced scientific or even commercial missions.

2.1.4 Advancements

As already mentioned, Delfi-n3Xt will consist of several advancements compared to its predecessor Delfi-C^3. The major advancements will be on several subsystems that belong to Delfi-n3Xt. A detailed description about the subsystems is given in the next Section. For now, only the advancements are listed. These advancements belong to the following subsystems:

- **Attitude and Determination Control Subsystem**
  For Delfi-n3Xt, the Attitude and Determination Control Subsystem (ADCS) should have significant advancements compared to that of the Delfi-C^3. The ADCS for Delfi-n3Xt [25] will provide more advanced capabilities for nanosatellites because of its full three-axis active control. Delfi-C^3 has a passive magnetic attitude control system which is not suitable for more advanced functionalities.

- **S-band transmitter**
  The Delfi-n3Xt nanosatellite will be expanded with an experimental S-band transmitter. The S-band transmitter (also known as the STX subsystem) allows higher downlink data rates which might be useful for future missions that need higher data rates.

- **Electrical Power Subsystem**
  The electrical power subsystem (EPS) of the Delfi-C^3 satellite did not had the capability of energy storage. The EPS for Delfi-n3Xt will have onboard energy storage in the form of a battery [26].

- **Command and Data Handling Subsystem**
  The OBC, which is part of the CDHS, will be single point of failure (SPoF) free. This is the reason why the OBC is redundant. In Delfi-C^3, the backup for operations by the single OBC was a distributed autonomous operation by the local microcontrollers of the other subsystems. However, for the more advanced Delfi-n3Xt mission this becomes way too complicated. This redundancy feature, together with the aim for a significant lower bit error rate (BER), makes the CDHS of Delfi-n3Xt more robust and more fault-tolerant [5] than the CDHS of the Delfi-C^3.

The designed fault-tolerant OBC software and the improvements in the BER [6] are one of the major things that are discussed in the remaining chapters of this thesis.

2.2 Delfi-n3Xt Payloads and Subsystems

The Delfi-n3Xt nanosatellite consists of various payloads and subsystems. In this Section all of these payloads and subsystems are shortly described. Section 2.2.1 gives background information about the micro propulsion payload. A description about the tranceiver payload is given in Section 2.2.2. In Section 2.2.3 the communication subsystem is explained. The attitude determination and control subsystem is discussed in Section 2.2.4, followed by the electrical power subsystem in Section 2.2.5. Finally, descriptions of the Delfi-n3Xt structure, mechanisms and thermal control subsystems are given in Section 2.2.6 and the already introduced CDHS is discussed in Section 2.2.7.

2.2.1 Micro Propulsion Payload

Future nanosatellite missions may consist of multiple nanosatellites that fly in formation or act as a swarm. Such missions require relative orbit control which can only be accomplished by a propulsion system onboard the satellites. The T$^3\mu$PS is such a propulsion system [21] and it is designed and engineered by TNO, together with the TU Delft and the University of Twente. The main problem for propulsion systems designed

Characteristic	Specification
Specific impulse	$>$ 30 s
Thrust	6 — 100 mN
Propellant mass	0.3 g per CGG, 2.4 g in total
Total mass	120 g
Dimensions	90 mm $\cdot$ 90 mm $\cdot$ 35 mm
Power consumption (per mode)	measuring mode: 63 mW
	thrusting mode: 335 mW
	ignition mode: 10.6 W

Table 2.1: T$^3\mu$PS characteristics and specifications

for nanosatellites is the contraints in the technical budgets such as volume, mass and power. In order to deal with these constraints, the T$^3\mu$PS is based on cold gas generators (CGGs). The specifications of the T$^3\mu$PS are listed in Table 2.1. Delfi-n3Xt will demonstrate the micropropulsion system by performing the following experiments:

- Ignition of multiple CGGs.
- Thrusting at various levels with pulse-width-modulated duty cycling.
- Determination of the leakage through continuous plenum pressure measurements during the nominal measurements mode of the micropropulsion system.
- Thrust determination by measuring the pressure drop during thrusting and indirectly by minor orbit changes measured on the ground by radar tracking.
- General housekeeping measurements such as temperature and power consumption.

2.2.2 Tranceiver Payload

The Delfi-n3Xt tranceiver payload (Figure 2.2) from ISIS BV is an experimental, high efficient radio that is in-orbit configurable. The radio is very flexible since the characteristics of the radio like data rate, modulation, output power and data protocol are configurable through software. This makes the radio usable for nanosatellite missions with radio requirements that meet the specifications of the radio.

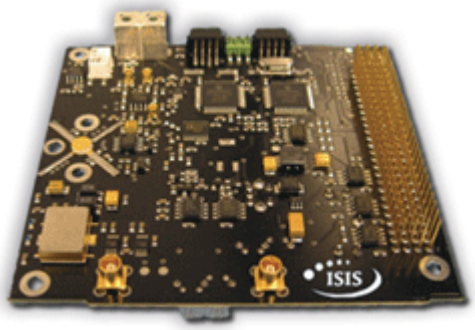


Figure 2.2: *Tranceiver payload from ISIS BV*

The tranceiver payload will be optimized such that it can be used with the very limited available power within the Delfi-n3Xt nanosatellite. The total power consumption of the tranceiver payload contributes for a significant part to the total power budget that is available for Delfi-n3Xt [16]. The power amplifier in the transmitter section of the radio is the electrical component that consumes most of the power [2]. The tranceiver payload will use a switching mode power amplifier that is being developed in cooperation between TU Delft, ISIS BV and SystematIC BV. The switching mode power amplifier can yield an efficiency of above 80%. The specifications of the tranceiver are listed in Table 2.2. This table shows that the transmitter consumes over 1.5W of power. For Delfi-n3Xt, the

Characteristic	Specification
Frequency bands	VHF downlink, UHF uplink
Supported data rates	1200, 2400, 4800 and 9600 bps
Supported modulation	(A)FSK, BPSK, CW, QPSK, MFSK
Supported protocols	AX.25, CW, ISIX, DelfiX
Total mass	90 g
Dimensions	90 mm · 90 mm · 20 mm
Power consumption	receiver: 255 mW
	transmitter: 1580 mW

Table 2.2: Delfi-n3Xt tranceiver payload specifications

data rate will be set to 2400 bps, the modulation to AFSK for UHF uplink and BPSK for VHF downlink and the protocol will be set to AX.25.

2.2.3 Communications Subsystem

The Delfi-n3Xt communications subsystem (COMMS) consists of three radios, phasing circuitry and antennas. An overview of this subsystem is shown in Figure 2.3.

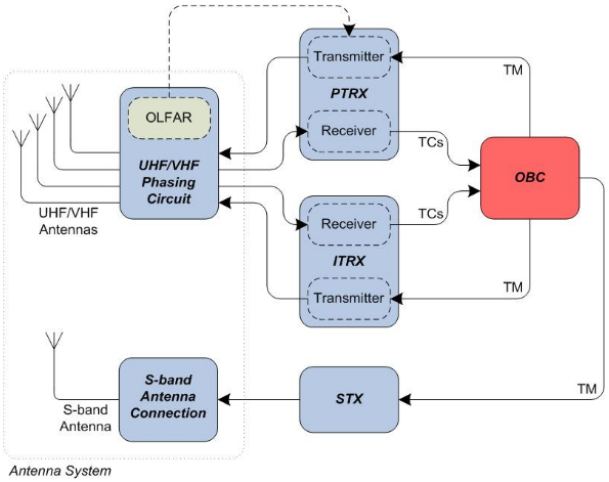


Figure 2.3: *Architectural overview of the Delfi-n3Xt Communications Subsystem*

The PTRX is the primary tranceiver which consists of a transmitter and a receiver. The ITRX is the experimental high efficiency tranceiver payload from ISIS BV that acts as a back-up radio for the PTRX. The ITRX was already described in Section 2.2.2. The experimental S-band transmitter can be used as transmitter only. Furthermore all the radios must be commanded by the OBC before they transmit data to the ground station.

2.2.4 Attitude Determination and Control Subsystem

The Attitude and Determination Subsystem (ADCS) consists of an attitude determination part and an attitude control part. Besides that, various kinds of hardware are present like sensors, actuators and a microcontroller that executes the determination and control algorithms [25]. The ADCS is experimental and the subsystem will demonstrate the following functionalities:

- Detumbling of the satellite
- Three-axis stabilization and pointing of the satellite with an accuracy of 3° with respect to the sun vector, velocity vector, magnetic field line and nadir.
- Slewing manoeuvre for ground station tracking of the S-band antenna with a 5° accuracy

In order to control the satellite, the ADCS consists of three magnetorquers and three reaction wheels. The three reaction wheels control the attitude of the three axes of the satellite [14]. A prototype of an orthogonal reaction wheel assembly is shown in

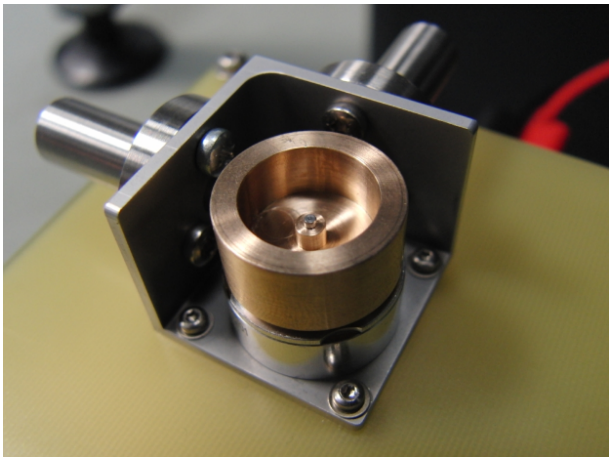


Figure 2.4: *Prototype of the ADCS orthogonal reaction wheel assembly*

Figure 2.4. The magnetorquers must dump the momentum of the spacecraft and the reaction wheels. Basically the magnetorquers are simple electric coils that will give a moment when an electric current flows through the coils [31]. Making use of the Earth its magnetic field, the magnetorquers can create a torque in the axes that are orthogonal to the magnetic field line of the Earth. The control algorithm in the microcontroller will be either a basic proportional-integral-derivative (PID) control algorithm or a more complex linear-quadratic regulator (LQR) control algorithm [25].

2.2.5 Electrical Power Subsystem

The Electrical Power Subsystem (EPS) must deliver a sufficient amount of power to all the other subsystem of the satellite. Basically, this subsystem consists of solar panels that generate power, a battery that can store the excessive amount of generated power by the solar panels and a couple of DC/DC converters that will convert the 12V input bus voltage to 3.3V and 5V output voltages that are used by the other subsystems [26]. An overview of the EPS is given in Figure 2.5.

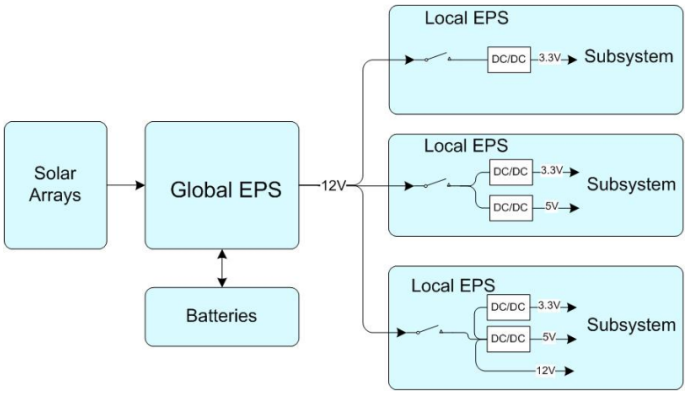


Figure 2.5: *Architectural overview of the Global EPS*

2.2.6 Structure, Mechanisms and Thermal Control Subsystems

The structural subsystem (STS), mechanical subsystem (MechS) and thermal control subsystem (TCS) have strong interfaces with one another and therefore they are all described in this single section. The STS consists of an inner structure and an outer structure and the outer structure is compliant with the standard triple-unit CubeSat dimensions [11]. The inner structure consists of 4 rods on which PCBs can be mounted on top of one another. This forms a stack of all the subsystem electronics. In Figure 2.6 an example of a rod system is shown.

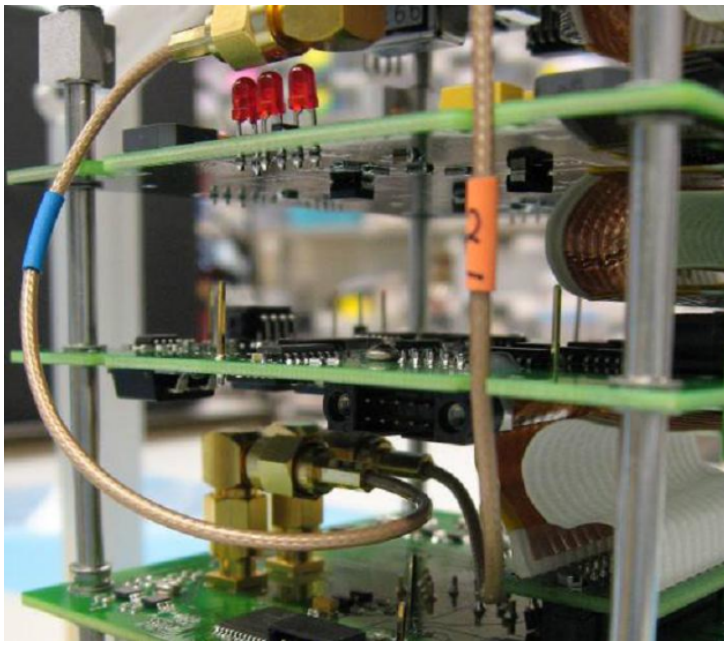


Figure 2.6: *Example of the rod system of Delfi-C*^{3}

Part of the MechS is the deployment machanism. This deployment mechanism is responsible for reliable deployment of the solar panels and the antennas of the radios. Deployment of a solar panel or antenna is done by letting a current flow through a resistor on which a thin wire is applied that holds the solar panel or antenna in its folded position [29]. Eventually, when a large enough current flows through a resistor for a longer period of time, the resistor will heat up until the wire burns. When the wire that holds the solar panel or antenna in its folded position burns, the attached solar panel or antenna will be deployed. The deployment mechanism is redundant in case one of the mechanisms fails. In total there are 4 solar panels and 4 antennas so each deployment board consists of 8 resistors that must be burned. For each resistor a power of 1.92W is required for approximately 12 seconds in order to burn it.

The Delfi-n3Xt thermal control is aimed to be passive by the use of heat radiators, heat sinks and thermal isolation. An analysis has been performed in order to check whether or not all the systems stay within their nominal operating temperatures [17].

2.2.7 Command and Data Handling Subsystem

As already mentioned before in the introduction, the Command and Data Handling Subsystem (CDHS) consists of the redundant OBC and the Delfi Standard System Bus (DSSB). The next section gives background information about the OBC and DSSB hardware details so they will not be discussed here. The operations and functionalities of the OBC are already described in Chapter 1. In Figure 2.7, the functional overview of the DSSB protection circuitry is shown.

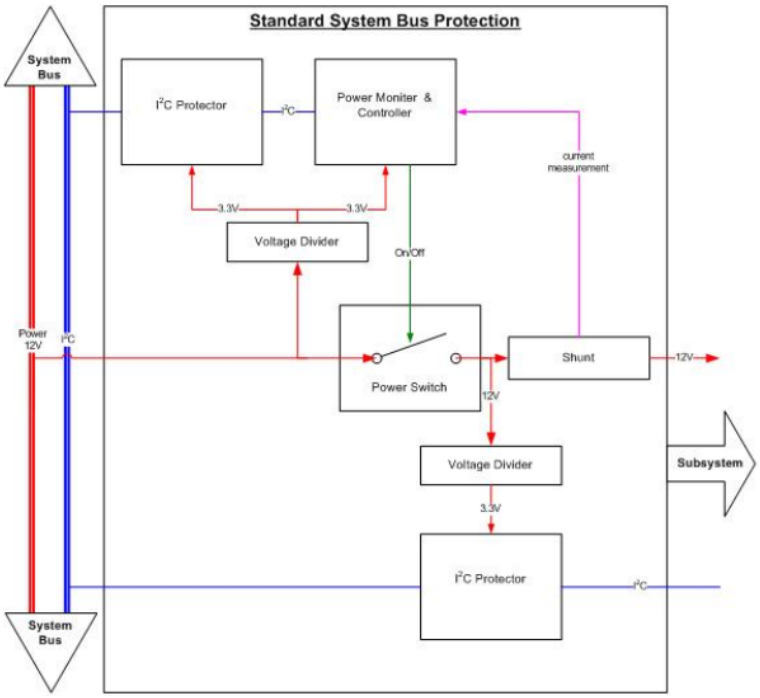


Figure 2.7: *Functional overview of the DSSB protection circuit*

The DSSB protection circuit is a combination of hardware and software that is able to detect the occurrence of short circuits and isolate the subsystem on which a short circuit occurred [7]. The DSSB protection circuit consists of a power monitor/controller and a shunt resistor that are together able to detect a short circuit that may have happened on the subsystem to which the DSSB protection circuit belongs. Since the DSSB protection circuit is present on all subsystems, all subsystems can be powered off by the DSSB protection circuit when a short circuit occurs in the subsystem. This DSSB functionality is an important step in the design and development of a fault-tolerant satellite.

The I^2C protector, as its name says, protects the I^2C lines for high currents or voltages on the lines that may destroy the I^2C hardware peripherals of I^2C devices that are connected to the data bus [7].

2.3 Delfi-n3Xt CDHS Hardware

The CDHS hardware consists of the redundant OBC hardware and the DSSB hardware. The Delfi-n3Xt redundant OBC hardware PBC layout is shown in Figure 2.8. The upper part of the PCB consists of the redundant OBC hardware, with on the left side the primary OBC and on the right side the secondary OBC (the two OBCs are seperated by the seperation line in the middle of the PCB). The secondary OBC must take over all the OBC operations in case of a failure in the primary OBC.

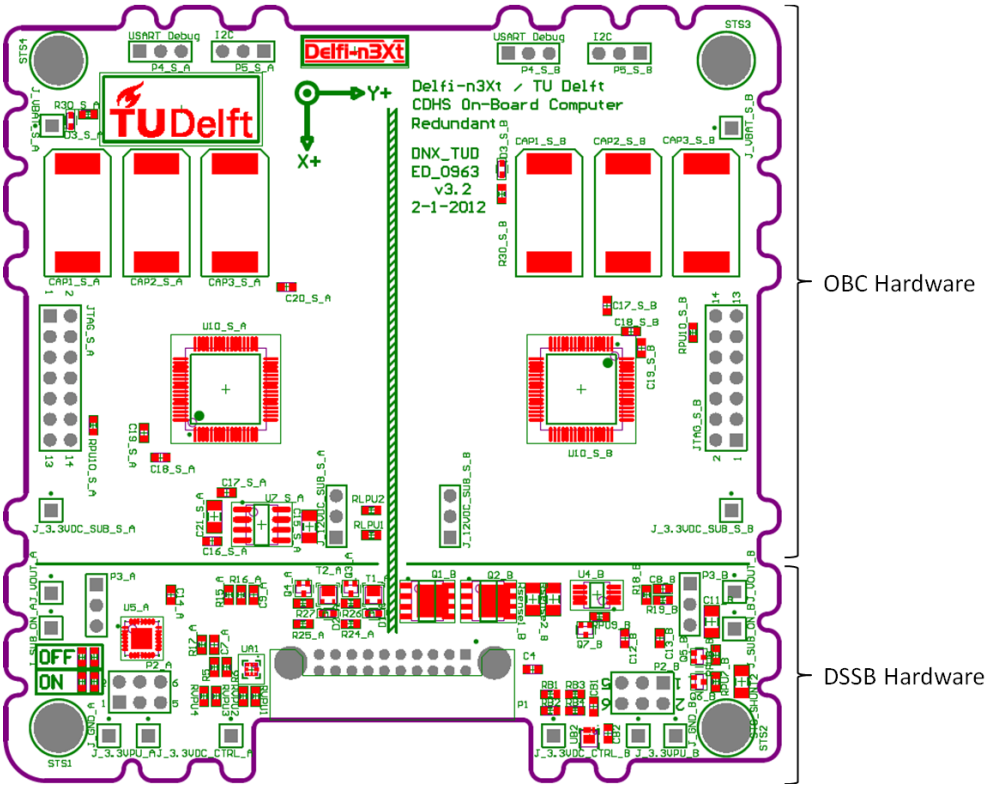


Figure 2.8: *Delfi-n3Xt On-Board Computer hardware PCB layout*

The lower part of the PCB consists of the double DSSB hardware; one DSSB circuit for the primary OBC and one DSSB circuit for the secondary OBC. With these two different DSSB circuits, the primary and secondary OBCs can be controlled independently. Furthermore, the different DSSB circuits make it possible to read out the operating voltage and the current consumption of the two OBCs.

In the remainder of this section the OBC hardware and DSSB hardware are further described into details. Section 2.3.1 gives more detailed information about the OBC hardware. A more detailed description about the DSSB hardware is given in Section 2.3.2.

2.3.1 On-Board Computer Hardware

The OBC hardware consists of various passive and active electrical components and a PCB with traces that connect all the components. The most important components that form the OBC hardware for a single OBC are the following components:

- One MSP430F1611 microcontroller [23]
- One 8 MHz crystal
- One Elapsed Time Counter
- Two 32768 Hz crystals
- Three supercapacitors

The MSP430F1611 microcontroller is a 16-bit reduced instruction set computer (RISC) that has an on-chip temperature sensor, a configurable clock source module and many other hardware peripherals that can be used for a wide range of applications. The hardware peripherals present on the MSP430F1611 are a flash memory controller, a DMA controller, a watchdog timer, two ordinary 16-bit timers, an USART peripheral supporting UART, SPI and I^2C modes, a comparator, a 10-bit and 12-bit analog to digital converter and a 12-bit digital to analog converter. Furthermore, the MSP430F1611 can operate at a clock frequency of at most 8 MHz. An 8 MHz external crystal will be used to drive the microcontroller chip and most of its peripherals. An external 32768 Hz crystal will be used to drive the watchdog timer peripheral.

The Elapsed Time Counter (ETC) is a 44-bit counter that maintains the amount of time that the device operates from main and/or backup power. The elapsed time can be read out by the microcontroller through a parallel interface [8]. The ETC is driven by an external 32768 Hz crystal. The three supercapacitors are used as backup power for the ETC and they ensure that the ETC keeps on working when the main power fails.

2.3.2 Delfi Standard System Bus Hardware

The DSSB hardware also consists of many passive and active electrical components and traces that connect all the components. The DSSB hardware is part of the PCBs of all subsystems, as can be seen in Figure 2.8 where the DSSB hardware is part of the OBC board. The most important components for the DSSB hardware are:

- One Atmel ATmega88PA microcontroller
- One 3.3V DC/DC converter
- Several transistors

The Atmel ATmega88PA microcontroller will be used to switch on or off the subsystem and to read out the operating voltage and the current consumption of the subsystem. The DSSB microcontroller and the rest of the subsystem operate on a DC voltage of 3.3V, so the 12V DC system bus voltage must be converted to a 3.3V DC voltage. The transistors are used for switching the power on or off for the subsystem, and cutting off the I^2C lines to the subsystem if necessary.

2.4 I²C Communication Basics

I²C stands for Inter-Integrated Circuit and is designed and developed by Philips. It allows serial communication between two or more devices (integrated circuits). Only two lines are needed to connect the devices; the data line and the clock line [24]. This section describes the basics of I²C communication.

2.4.1 Hardware Setup

There are two kinds of I²C devices: the master device and the slave device. Communication between master and slave devices takes place over two lines: the I²C Serial Clock (SCL) line and the I²C Serial Data (SDA) line. The master device controls the SCL line and initiates data transfers to or from a slave device over the SDA line.

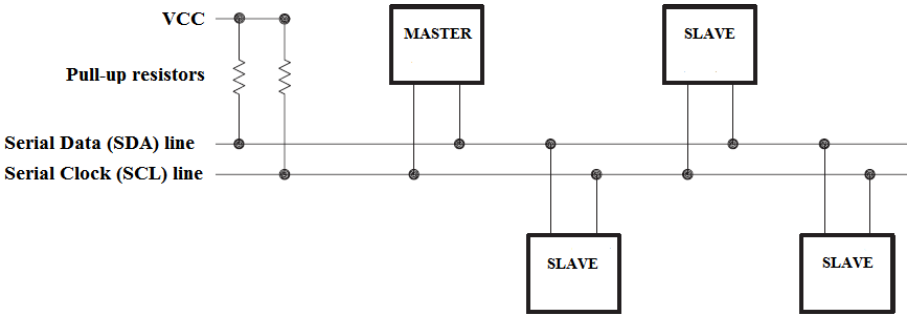


Figure 2.9: *I²C single-master, multiple-slave setup*

It is possible to have multiple master devices connected to the bus. This multi-master architecture is not considered in this thesis since it is specified in the On-Board Computer requirements in Section 4.1 that only one master device is allowed to control the bus. The architecture of a single-master, multiple-slave setup is shown in figure 2.9. The devices on the bus are also connected to V_{cc} and GND (not shown in the figure).

Each device that is connected to the bus must have an own unique address. Using this addressing scheme the master device is able to contact a specific slave device and initiate a read or write operation. An address is 7 bits wide and a total of 112 devices can be connected to the I²C bus (some addresses are reserved and cannot be used). There is also a 10-bit addressing mode which allows one to connect more devices to the bus [24]. However, this generates an overhead of 1 byte for each data transfer compared to the 7-bit addressing mode. To have a reliable communication the clock speed of the microcontrollers that act as master and slave devices should be at least 10 times the I²C bus speed [19]. Note the pull-up resistors between the bus lines and the common-collector voltage supply line (V_{cc}). These pull-up resistors are needed because the I²C bus is an open-collector design. This means that when the I²C bus is in the idle state, the bus lines are pulled high to V_{cc} (meaning a logical 1). The I²C devices that are connected to the bus can pull the bus lines low (meaning a logical 0) or just do nothing with the bus lines (meaning a logical 1).

2.4.2 Start and Stop Conditions

There are two kinds of data transfer operations: write operations and read operations. Both are more elaborated in the subsequent paragraphs of this section. Each operation starts with a start condition and stops with a stop condition, as shown in Figure 2.10.

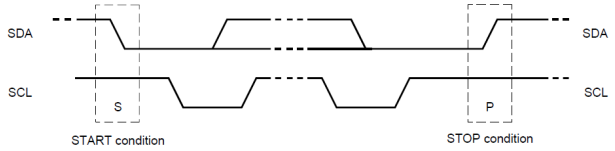


Figure 2.10: *Start condition and stop condition*

Start and stop conditions are generated by the master device. A high to low transition on the SDA line while the SCL line is high is known as a start condition. A low to high transition on the SDA line while the SCL line is high is known as a stop condition. The bus is busy after a start condition and free after a stop condition.

2.4.3 Write Operation

The master always initiates a transfer, so a write operation means that the master device writes data on the bus and a slave device reads data from the bus. With such an operation the master device operates in master transmitter mode and the slave device in slave receiver mode. To achieve this the following will happen:

- The master sends a start bit (start condition).
- The master sends the 7-bit address of the slave device.
- The master sends a bit with the logical value 0, representing a write operation.
- The slave responds with an acknowledge (ACK) bit (logical 0) if it exists on the bus.

If an ACK bit with a logical value of 0 (active low) is received from the slave device the master knows that the slave is present and that the slave is ready to receive data. If an ACK bit is not received (the master reads a logical value of 1 because the slave did not pull low the SDA line) the master knows that the slave is not present and that it is not possible to write data to the slave device. Once the master device received the ACK bit with a logical value of 0 it can continue with sending data:

- The master sends a data byte on which the slave responds with an ACK bit (logical value of 0).
- The master continues sending data bytes as specified in the previous step until it is done.
- The master sends a stop bit (stop condition), this is also the case when the slave not acknowledges.

The data transfer from master to slave is shown graphically in Figure 2.11 [24]. Here it can be seen how the SDA bus line is used for the data transfer. Note the value of the R/$\overline{W}$ bit. It equals 0, representing a write operation.

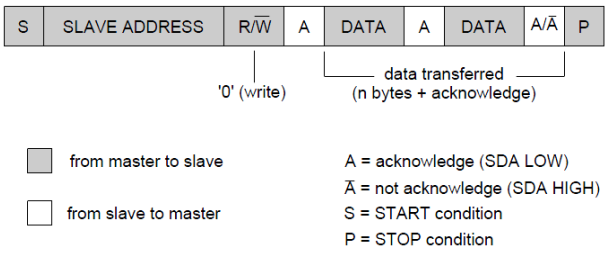


Figure 2.11: *Data transfer from master to slave*

2.4.4 Read Operation

A read operation means that the master device reads data from the bus and a slave device writes data on the bus. With such an operation the master device operates in master receiver mode and the slave device in slave transmitter mode. To achieve this the following will happen:

- The master sends a start bit (start condition).
- The master sends the 7-bit address of the slave device.
- The master sends a bit with the logical value 1, representing a read operation.
- The slave responds with an acknowledge (ACK) bit (logical 0) if it exists on the bus.

If an ACK bit with a logical value of 0 (active low) is received from the slave device the master knows that the slave is present and that the slave is ready to send data. If an ACK bit is not received (the master reads a logical value of 1 because the slave did not pulled the SDA line low) the master knows that the slave is not present and that there is no need to read data from the bus. Once the master device received the ACK bit with a logical value of 0 it can continue with receiving data:

- The slave sends a data byte on which the master responds with an ACK bit (logical value of 0).
- The slave continues sending data bytes as specified in the previous step until it is done. The master acknowledges all the bytes, except for the last byte.
- The master sends a stop bit (stop condition). This is always the case, even if one or more of the previous steps fail.

The data transfer from slave to master is shown graphically in Figure 2.12 [24]. Here it can be seen how the SDA bus line is used for the data transfer. Note the value of the R/$\overline{W}$ bit. It equals a logical 1, representing a read operation.

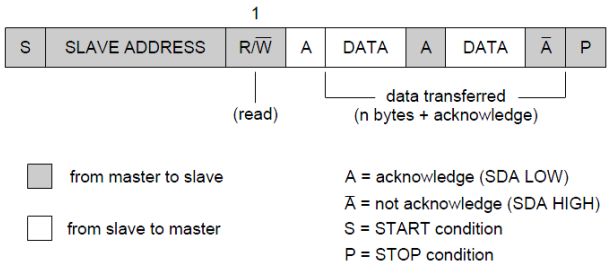


Figure 2.12: *Data transfer from slave to master*

2.4.5 Sampling vs. Edge-Triggered Interrupts

An I^2C controller can be implemented in software or in hardware. Most modern microcontrollers have a hardware I^2C module on-chip. I^2C controllers that are implemented in software use a sampling technique to determine what happens on the SDA and SCL lines. The sampling rate must be at least twice as high as the I^2C data bus rate for reliable communication. Any sampled data can be stored and processed further. I^2C controllers that are implemented in hardware usually use edge-triggered interrupts to determine what happens on the I^2C bus lines. In this case the I^2C controller reacts on a rising edge (a low to high transition) or a falling edge (a high to low transition) on both bus lines. Usage of a hardware I^2C controller that is based on edge-triggered interrupts is less error prone compared to the usage of I^2C controllers implemented in software. One advantage of hardware I^2C controllers is that they are thoroughly tested by the manufacturer of the chip. A disadvantage of sampling is that data may be sampled after the occurrence of a bit flip on one of the lines. The impact of bit flips is further described in Section 3.1, which is about I^2C communication failures. The Delfi-n3Xt OBC its microcontroller consists of an I^2C hardware peripheral which is edge-triggered.

2.4.6 Data Transfer Example

As already mentioned before, I^2C communication works over two lines: the serial data line (SDA) and the serial clock line (SCL). Each clock pulse on the SCL line corresponds to a bit that is transferred over the SDA line. Using the clock line the master and slave devices remain sync to one another. The master generates the clock pulses over the SCL line and the slave device can read data from or write data to the SDA line on each clock pulse. The state machines in the hardware of the I^2C peripherals of the devices should count the number of clock pulses on the SCL line, such that data can be read from or written to the SDA line correctly.

An example of a data transfer over the I^2C bus is shown in Figure 2.13. In the figure one is able to see a small data transfer of only 1 byte from the master device to a slave device. The data transfer takes place from left to right. In the very left side of the figure both the SDA and SCL lines are high.

Suddenly the SDA line makes a transition from high to low while the SCL line is high. This happens because the master wants to initiate a data transfer. To do this the master pulls low the SDA line while the SCL line remains high. With this action the master

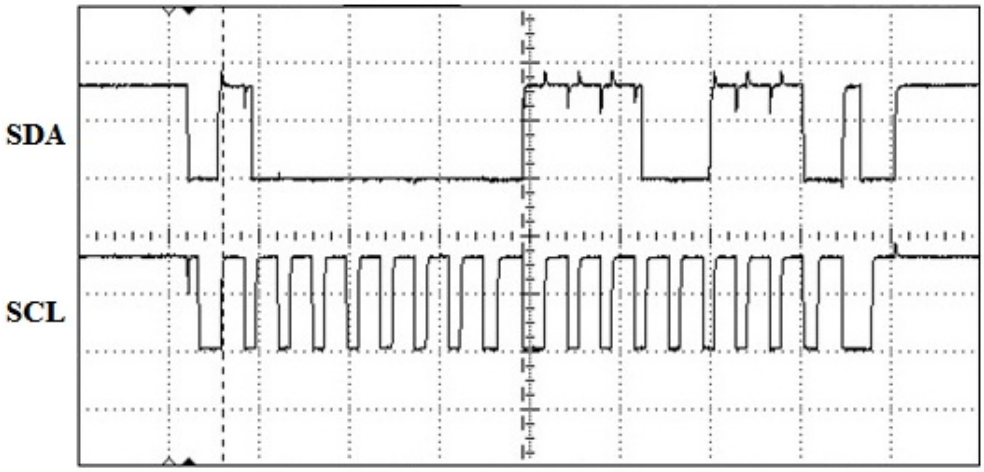


Figure 2.13: *Example of a data transfer over the I^2C lines*

issues a start condition; all the slave devices that are connected to the I^2C bus will detect this, such that they know that the master would like to perform a data transfer. Next, the master sends out clock pulses over the SCL line (nominally at a rate of 100kHz) and puts data on the SDA line. The first 7 data bits on the SDA line correspond to the first 7 clock pulses on the SCL line and represent the slave address with which the master would like to communicate. One can see that during the first clock pulse the SDA line is high and during the next 6 clock pulses the SDA line is low. This means that the master wants to communicate with a slave device with address 1000000 in binary (0x40 in hexadecimal).

The next clock pulse (i.e. clock pulse number 8) corresponds to the R/$\overline{W}$ bit which indicates whether the master would like to read data from the slave or write data to the slave. In the example shown in Figure 2.13 the SDA line is low during the clock pulse that corresponds to the R/$\overline{W}$ bit, indicating that the master wants to write data.

The ninth clock pulse corresponds to the acknowledgement (ACK) bit. The SDA line is pulled low by the slave device with address 0x40 during the ninth clock pulse. This means that the slave device with address 0x40 is present on the bus.

At this point the master knows that the slave is ready to receive data from the master. Now the master will issue 9 more clock pulses; 8 clock pulses for the data byte and 1 clock pulse for the acknowledge bit. The SDA line is high on clock pulses 1, 2, 3, 6, 7 and 8 and low on clock pulses 4 and 5. This means that a data byte with a value of 11100111 in binary has been transferred from the master to the slave. On the ninth clock pulse of the data byte transfer the slave device pulls low the SDA line in order to let the master know that it received the data byte properly. After this acknowledgement the slave releases the SDA line such that is becomes in a high state.

The master now ends the transfer by pulling low the SDA line and release the SDA line while the SCL line is in the high state, as can be seen in the very right side of Figure 2.13. This represents a stop condition. The slave device is able to detect this stop condition such that it knows that the transfer is finished.

2.4.7 Clock Stretching

The I²C specification allows devices that are connected to the I²C bus to stretch the clock. This feature is useful when slow devices are connected to the I²C bus. Slower devices might not be able to handle and process data at the rate at which the master generates clock pulses. It also might be the case that a slave device is busy with other things such that the handling of I²C interrupts have to wait. When this happens the slave device can pull low the SCL line until it is ready to go on with the processing of I²C events.

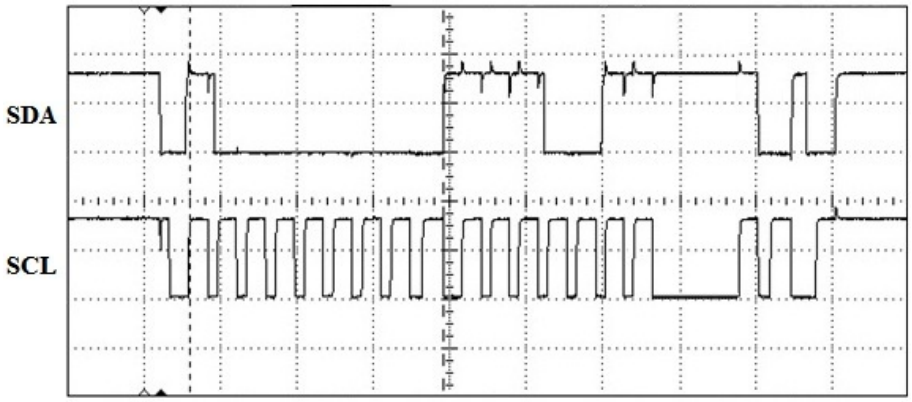


Figure 2.14: *Example where the slave device stretches the clock*

The data transfer example shown in Figure 2.13 is also shown in Figure 2.14, except that the slave device now stretches the clock after clock pulse number 16 of the transfer. When the slave stretches the clock, the SDA line remains in the same state as it was one clock pulse earlier. In this particular example the SDA line remains in the high state. In Figure 2.14 one can see that after a while the slave device releases the SCL line. This means that the slave device processed the I²C interrupt event and is ready for further data reception.

2.4.8 Data Transfer Speeds

In theory there are no restrictions to the data transfer speed for data transfers over the I²C bus. The main thing is that when one likes to perform data transfers at a desired rate, two things should be met:

- The master device must be able to generate clock pulses and process data at the desired rate.
- The slave devices connected to the bus must be able to process data at the desired rate.

In practice, most I²C peripherals have a data transfer speed restriction in order to facilitate reliable data transfers. In the I²C specification it is stated that the CPU clock frequency of devices connected to the I²C bus must be at least 10 times higher than

the clock frequency of the I2C bus [24]. Furthermore the I^2C specification specifies the following standardized bus speeds:

- Normal mode, 100kHz (100kbit/sec)
- Fast mode, 400kHz (400kbit/sec)
- Fast mode plus, 1MHz (1Mbit/sec)
- High speed mode, 3.4MHz (3.4Mbit/sec)

However, the master device is in principle free to select any frequency at which data will be transferred.

2.5 Summary

The background information that is required to understand the remainder of this thesis is presented in this chapter. This includes background information about the Delfi program, the Delfi-n3Xt nanosatellite mission, the payloads and subsystems of Delfi-n3Xt, the CDHS hardware and the I^2C bus communication basics.

The Delfi program is a nanosatellite development line that currently consists of two satellites: the already launched Delfi-C^3 and the Delfi-n3Xt which is planned for launch in September 2012. The Delfi-n3Xt nanosatellite is currently under development at the Faculty of Aerospace Engineering, department of Space Systems Engineering, Delft University of Technology. The Delfi-n3Xt mission has three objectives: education, technology demonstration and nanosatellite bus development. Several significant advancements have been made in the ADCS, S-band transmitter, EPS and CDHS compared to Delfi-C^3.

Delfi-n3Xt consists of various payloads and subsystems. The two payloads are the micro propulsion payload and the tranceiver payload. The other subsystems are the communications subsystem, the attitude determination and control subsystem, electrical power subsystem, structural subsystem, mechanical subsystem, thermal control subsystem and command and data handling subsystem which are all shortly introduced and described in this chapter.

The On-Board Computer hardware of Delif-n3Xt consists of a MSP430F1611 microcontroller, an 8 MHz crystal, an elapsed time counter, two 32768 Hz crystals, three supercapacitors and various other passive and active electrical components. The Delfi Standard System Bus hardware also consist of several passive and active electrical components. The most important components for the DSSB are the Atmel ATmega88PA microcontroller, the 3.3V DC/DC converter and several transistors that are used to cut off the power lines or I^2C lines to the subsystem.

For I^2C communication, one master device and one or more slave devices are needed. Each I^2C slave device has an unique address. A data transfer can be a write operation or a read operation. A write operation is a data transfer from the master to a slave device and a read operation is a data transfer from a slave device to the master device. All data transfers are started with a start condition and stopped with a stop condition. Furthermore slave devices are allowed to stretch the I^2C clock in order to slow down a data transfer.

3 I²C Bus Analysis

This chapter is devoted to a failure analysis and performance analysis of the Delfi-n3Xt I^2C bus. In Section 2.4 the basics of communication over the I^2C bus were already described. In this chapter, an extensive analysis of I^2C bus failures with their possible causes and impacts is given in Section 3.1. The I^2C bus performance analysis of the engineering model hardware is discussed in Section 3.2. Section 3.3 summarizes this chapter.

3.1 I^2C Bus Failure Analysis

During I^2C communication between a master and a slave device, several failures may occur. These failures can lead to unpredictably behavior or bus lock-ups. In space, these failures may be introduced by external factors like electromagnetic radiation (e.g. direct solar radiation, Albedo radiation and thermal radiation emitted by earth) and particle radiation [27]. The different kinds of radiation may introduce bit flips or distortions on the SDA and SCL lines. The failures that may occur are analyzed according to the data transfer procedure from the beginning to the end (i.e. from the start condition to the stop condition). In the end of this section the failures are summarized in a table in which the causes, impacts and resolve steps of the failures are described.

3.1.1 Start Condition Failures

Each data transfer on the I^2C bus starts with a start condition. The start condition is defined as a high to low transition on the SDA line while the SCL line is high, as shown in Figure 3.1. Here it is assumed that the SDA line and SCL line are sampled at times t_1 and t_2.

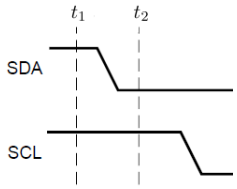
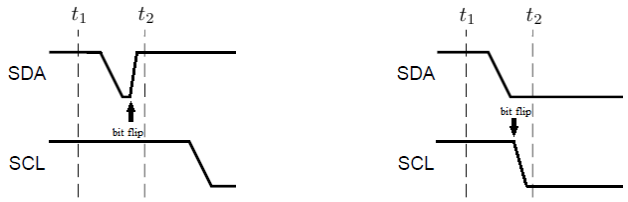


Figure 3.1: *Start condition with SDA high at time t_1 and SDA low at time t_2*

When slave devices read a high to low transition on the SDA line as shown in Figure 3.1 (i.e. SDA is high at time t_1 and SDA is low at time t_2 while SCL is high), the slaves connected to the bus will continue reading the 7 address bits and the R/$\overline{W}$ bit

from the SDA line. Once a slave read the 8 bits (address and R/$\overline{W}$ bit), the slave will compare its own address with the address that it read from the bus. If the slave its own address is the same as the address that it read from the bus, the slave will respond with an ACK bit. If the slave its own address is different than the address that is read from the bus, the slave does nothing.



(a) Bit flip on SDA (b) Bit flip on SCL

Figure 3.2: *Bit flips in a start condition on the I^2C lines between times t_1 and t_2*

Now suppose a bit flip occurs somewhere between time t_1 and time t_2 such that the SDA line is also high at time t_2, as illustrated in Figure 3.2a. If a bit flip occurs at this time on the SDA line, the slave devices read SDA high at time t_1 and SDA high at time t_2 while SCL is high. Because of the bit flip the original start condition has now been transformed into something which is not a start condition anymore. A similar situation happens when a bit flip occurs on the SCL line between time t_1 and time t_2 as illustrated in Figure 3.2b. Now the SCL line is not high anymore on time t_2, which means that it is not a start condition anymore.

The direct consequence of bit flips as illustrated in Figure 3.2a and Figure 3.2b is that the slave devices will not read address bits and the R/$\overline{W}$ bit from the bus, since they did not read a start condition. The master expects an ACK bit if the slave device with the requested address is present. But even if the slave device is present it wont respond with an ACK bit since it missed the start condition. The master device concludes that the requested slave device is not present on the bus, while it could be the case that it actually is present.

3.1.2 Slave Request Failures

The next thing that can go wrong is the request for a slave device from the master. Here it is assumed that the start condition is read and interpreted correctly by the slave devices. As already explained earlier, the 7 address bits and the R/$\overline{W}$ bit are send directly after the start condition. The slave devices read the address bits and the R/$\overline{W}$ bit and compare the address with their own address. If there is a match, the requested slave device will respond with an ACK bit. An error may be introduced by a bit flip in one or more bits of the 7 address bits. A bit flip in the address may lead to two situations:

- The address that is put on the bus is turned into an address which is not present on the bus.
- The address is turned into an address used by another slave device.

Suppose the master wants to perform a read or write operation from or to a slave device with address 0x69 in hexadecimal (1101001 in binary). Now suppose a bit flip occurs in the fifth clock pulse, i.e. the requested slave device address turns from 0x69 into 0x6D (from 1101001 into 1101101 in binary). When there is no slave device present with address 0x6D (1101101) the master will not read an ACK. The master will read a NACK (No Acknowledge) so that it knows that the requested slave device is not present. However, the master does not know that a bit flip occurred and that the requested slave device actually is present and connected to the bus.

In the other case, the requested slave address can be transformed into a slave address of another slave device. Suppose one slave device has the address 0x69 (1101001 in binary) and another slave device has the address 0x6D (1101101 in binary). Now when the master wants to perform a read or write operation from or to the slave device with address 0x69 (1101001) and the bit flip transforms the requested slave address into the address 0x6D (1101101), the master unintentionally performs a request for the wrong slave device. This should really be avoided, otherwise there is a possibility that the wrong data is being fetched from the wrong slave device.

To reduce the probability that a bit flip transforms an address into an address that is in use by another slave device, a minimal hamming distance of 2 between allocated slave addresses can be used. The hamming distance is defined as the number of different symbols between two strings. In the case of I^2C addresses, the hamming distance is the number of different bits between two addresses. The higher the hamming distance between allocated slave addresses means the lower the probability that a requested slave address will be transformed into another slave address that is present on the bus. In the CDHS Software ICD [18], most of the I^2C addresses of the subsystems in the Delfi-n3Xt satellite are already defined such that the hamming distance between each allocated address equals 2.

3.1.3 Read/Write Bit Failures

After the start condition and the requested slave address, the R/$\overline{W}$ bit is placed on the bus by the master. A logical value of 1 (SDA line high) means that the master requests a read operation and a logical value of 0 (SDA line low) represents a write operation, as shown earlier in this document in Figure 2.12 and Figure 2.11 respectively. For the R/$\overline{W}$ bit, a bit flip on the SDA line may result in the two following situations:

- The master wants to read (R/$\overline{W} = 1$) but the requested slave reads (R/$\overline{W} = 0$).
- The master wants to write (R/$\overline{W} = 0$) but the requested slave reads (R/$\overline{W} = 1$).

In both situations it is assumed that there were no bit flips in the start condition and the slave request. The first situation is the case when the master device wants to read from the requested slave device. In this case the master does not pull low the SDA bus line during the clock pulse corresponding to the R/$\overline{W}$ bit (i.e. it does nothing with the SDA line). Herewith the master indicates that it wants to perform a read operation, so the master will be in master-receiver mode. Now suppose that a bit flip occurs and that the SDA line is pulled low by some external event. The requested slave device

understands that the master wants to write to the slave device (because it reads R/$\overline{W}$ = 0 instead of R/$\overline{W}$ = 1) so the slave will wait for bytes to be received from the master (i.e. the slave will be in slave-receiver mode). However, the master still wants to perform a read operation, so the master is waiting for bytes to receive from the requested slave device. Now both sides are waiting for each other because both sides are in receiver mode, which obviously should result in a time-out.

A more interesting failure becomes visible when a bit flip occurs if the master device wants to write to the requested slave device. The master device pulls low the SDA bus line during the clock cycle that corresponds to the R/$\overline{W}$ bit to indicate that it wants to write to the requested slave device (i.e. R/$\overline{W}$ = 0). The master will be in master-transmitter mode since it wants to write data on the bus. Now suppose that a bit flip occurs and the requested slave device reads R/$\overline{W}$ = 1 instead of R/$\overline{W}$ = 0. The slave understands that the master devices wants to read data from the slave device. The slave device will go into slave-transmitter mode, resulting in both sides to become a transmitter. Now both the master and the slave device will try to write data on the bus and both sides will wait for the ACK bit of the other side in order to proceed the data transfer. In this situation the I^2C bus its SDA line becomes in an unpredictable state. However, both sides are waiting for an acknowledgement from the other side. Hence on both sides a timeout occurs after they wrote the first data byte on the bus.

3.1.4 Slave Acknowledgement Failures

Suppose that the master would like to initiate a data transfer and that the start condition, slave device address and R/$\overline{W}$ bit are free of errors (bit flips). The next step is the acknowledgement which has to be performed by the slave device (if a slave with the requested address exists on the bus, of course). If there is no slave device present on the bus with its address equal to the address of the requested slave device, the SDA line will stay high during the clock cycle that corresponds to the acknowledgement (this indicates a NACK). If the slave device with the requested address is present, the slave device will pull the SDA line low to let the master know that it is present. For the slave acknowledgement bit, two failures may be introduced on the bus because of a bit flip on the SDA line:

- The slave device is not present, but the master reads an ACK.
- The slave device is present and responds with an ACK, but the master reads a NACK.

The first failure case is when the master reads an ACK response bit from the slave device while the slave device is actually not present on the bus. This obviously can only happen when a bit flip occurs on the SDA line exactly during the clock pulse (on the SCL line) that corresponds to the slave acknowledgement bit. The bit flip causes the SDA line to be pulled low, with the consequence that the master receives an acknowledge from the slave. The master now continues the data transfer. It issues clock cycles and writes 8 bits on the bus when it would like to write data to the slave, or it just waits for the 8 bits to be received from the slave device if it wants to read data from the slave. If

the master wants to write it will write the 8 bits on the bus, but then it wont receive an acknowledgement from the slave at the ninth clock pulse, which causes the master to stop the transfer. If the master wants to read it will read all the bytes it wants, but all bytes will have a value of 0xFF in hexadecimal since the SDA line will always be high during clock pulses corresponding to the data.

The second failure case is the other way around: the slave device is present and connected to the bus and the slave device responds with an ACK, but the master device reads a NACK and understands that the slave does not exist. The master will not attempt to read data from or write data to the bus and will stop the data transfer.

3.1.5 Data Byte Transfer Failures

To illustrate possible failures during the transfer of data bytes, it is again assumed that during the previous bus activities (start condition, slave address, R/$\overline{W}$ bit and ACK bit) no errors (bit flips) occurred. A data transfer is a series (1 or more) of one-byte data transfers where each data byte is followed by an ACK bit. Bit flips in the data bytes are at this point unimportant since they do not bring the I^2C bus in a failure state and will only result in bit errors in the data bytes. Bit flips in the data bytes should be handled by a data consistency check procedure and/or a data error correction procedure in order to ensure correct data reception. The important part in the data transfer that may introduce failures is the acknowledgement bit. There are two different modes in which a failure may occur:

- Master-transmitter, slave-receiver mode: the slave has to acknowledge.
- Master-receiver, slave-transmitter mode: the master has to acknowledge.

In the master-transmitter, slave-receiver mode (i.e. when the master device is sending data to the slave device), the slave device has to acknowledge each data byte that it received from the master device. For a transfer from the master to the slave, two failures may occur:

- The slave device acknowledges, but the master device reads a NACK.
- The slave device does not acknowledge, but the master device reads an ACK.

When a slave device acknowledges but the master reads a NACK, the master device will abort the transfer by sending out a stop condition such that the bus becomes free. It may be possible that the slave device does not acknowledge, for example when the slave device misses clock pulses or the slave device is not operable anymore because of some kind of hardware failure. In this case the master reads a NACK, but when a bit flip occurs the master will read an ACK and will continue with sending the next data byte. This does not harm the health of the I^2C bus, since at the end of the next data byte transfer the master will read the actual NACK (when a bit flip does not occur) after which the master will eventually issue a stop condition that will free the bus.

For the master-receiver, slave-transmitter mode, the master device receives data from the slave device so the master device has to acknowledge each data byte that it received

from the slave device except for the last data byte [24]. The master reads n bytes from the slave device where $n \geq 1$. In this mode there may happen two failures because of a bit flip on the SDA line:

- For one or more of the first $n-1$ bytes, the master device acknowledges but the slave device reads a NACK.
- The master device does not acknowledge for the n_{th} byte, but the slave device reads an ACK.

The first failure should not lead to a problem on the bus. If the slave device reads a NACK after it wrote a data byte on the bus, the slave device will stop with writing the next data bytes on the bus and becomes in an idle state. The master however, will still issue 9 more clock pulses for the next data byte and the ACK bit. The slave device is in idle mode now so it wont write any data bytes on the bus. This means that the SDA line will remain high, and thus the master reads for all the next bytes a value of 0xFF (11111111 in binary).

The second failure should not lead to a problem on the bus either. The master did not acknowledge for the last byte, but the slave device reads that it acknowledged. This is actually not a problem because the master device will continue with sending a stop condition anyway (since this was the last byte the master sent). The slave device will recognize the stop condition and the bus will be released such that it can be used for a new data transfer.

3.1.6 Stop Condition Failures

Stop condition failures are similar as the start condition failures. As mentioned earlier, each data transfer on the I^2C bus stops with a stop condition. The stop condition is defined as a low to high transition on the SDA line while the SCL line is high, as shown in Figure 3.3. Here it is assumed that the SDA line and SCL line are sampled at times t_1 and t_2.

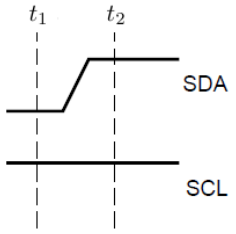
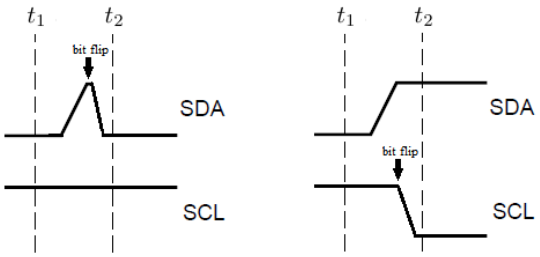


Figure 3.3: *Stop condition with SDA low at time t_1 and SDA high at time t_2*

When the slave device the master was reading from or writing to reads a low to high transition on the SDA line as shown in Figure 3.3 (i.e. SDA is low at time t_1 and SDA is high at time t_2 while SCL is high), the slave device understands that the transaction is about to end. After the stop condition the slave will completely release the bus lines such that the bus becomes free again for a new transaction between the master and any slave device.



(a) Bit flip on SDA (b) Bit flip on SCL

Figure 3.4: *Bit flips in a stop condition on the I^2C lines between times t_1 and t_2*

Now suppose a bit flip occurs somewhere between time t_1 and time t_2 such that the SDA line is also low at time t_2, as illustrated in Figure 3.4a. If a bit flip occurs at this time on the SDA line, the slave devices read SDA low at time t_1 and SDA low at time t_2 while SCL is high. Because of the bit flip the original stop condition has now been transformed into something which is not a stop condition anymore. A similar situation happens when a bit flip occurs on the SCL line between time t_1 and time t_2 as illustrated in Figure 3.4b. Now the SCL line is not high anymore on time t_2, which means that it is not a stop condition anymore.

The direct consequence of bit flips as illustrated in Figure 3.4a and Figure 3.4b is that the slave device will not release the I^2C bus lines. When a slave is in slave-transmitter mode it can still write data on the bus when the master issues clock pulses on the clock line, while it is not supposed to do that since the master device issued a stop condition. This may introduce earlier described failure cases and data inconsistencies.

3.1.7 Slave Device Missing Clock Pulses

Besides failures on the SDA line during the transfer of data, failures on the SCL line may occur as well. When a bit flip on the SCL line occurs in the middle of a data transfer, the slave device might miss one or more clock pulses. I^2C controllers that are implemented in hardware will then get stuck somewhere in their state machine, meaning that the I^2C controller of the slave device is waiting for more clock pulses from the master in order to finish the data transfer.

When this happens in master-transmitter, slave-receiver mode, the slave device still wants to read bits from the bus but the master already expects an ACK from the slave device. This might happen if the master already send out 9 clock pulses (8 clock pulses for the data byte and 1 clock pulse for the ACK) but the slave received less than 9 clock pulses (because of e.g. a bit flip on the SCL line). At the ninth clock pulse the master will read a NACK because the slave does not pull the SDA line low at this clock pulse since it still wants to read bits from the bus. This results in an abortion of the data transfer, followed by a stop condition from the master.

In the case of the master-receiver, slave-transmitter mode, the slave device will write bits on the bus that are out of sync. Suppose the master already send out 3 clock pulses but the slave device missed the third clock pulse. This means that the first 2 bits are

received correctly, but the third bit might or might not be correct. The point is that the hardware state machine of the slave device is still waiting for a clock pulse from the master in order to send out the third bit. When the master sends out the fourth clock pulse and the slave device receives that clock pulse, the slave device will send out the third bit over the data line while the master expects the fourth bit. At this point the data transfers runs out of sync, and at the ninth clock pulse the master expects an ACK (i.e. the slave device pulls the SDA line low) but the slave device will send out bit number 8 of the data byte (assuming that the slave device missed exactly one clock pulse). Depending on the value of the last data bit, the following two events may happen:

- If the last bit of the data byte equals 0, the master will read an ACK at the ninth clock pulse.
- If the last bit of the data byte equals 1, the master will read a NACK at the ninth clock pulse.

In the first case the master will continue with sending clock pulses over the SCL line. If this failure occurs on the very last byte to be transferred, the slave device might pull low the SDA line indefinitely. In the second case the master will abort the data transfer and send out a stop condition such that the bus becomes free.

3.1.8 Data Line Pulled Low Indefinitely

A serious failure occurs when one of the I²C devices connected to the I²C bus pulls low the SDA line indefinitely. When this happens, further communication over the I²C bus is not possible. Unfortunately, this failure can happen anywhere in the system at any time. An I²C device can pull low the SDA line indefinitely when one of the following events occur:

- An electrical short between the ground and the SDA line
- A lock-up in the I²C controller its hardware state machine while it pulls the SDA line low
- A latch-up in the I²C controller electronics

An electrical short between the ground and the SDA line can be caused by an internal hardware failure or a soldering failure that connects the SDA line with the ground. When the SDA line is connected to the ground constantly, the SDA line will be pulled low constantly. Clearly, this results in a situation where communication is not possible anymore since the representation of a logical 1 corresponds to a high SDA line (hence a logical 1 can not be represented anymore in this situation). The SDA line can not become in the high state anymore since it is pulled low constantly. A failure like this can be solved if the I²C device that is responsible for pulling the SDA line low constantly is isolated from the I²C bus. The failure may also be solved by replacing the component that introduces the electrical short. This is of course not possible once the satellite is in space, so the only option for solving this problem is to isolate the faulty component.

If the SDA line is pulled low constantly by a lock-up in the I²C controller its hardware state machine or by a latch-up in the I²C controller electronics, the SDA line will be pulled low until the I²C device responsible for pulling the SDA line low is power cycled. The lock-up in the I²C controller hardware state machine can be caused by a bug in the hardware I²C controller or when the I²C device missed a clock pulse. This problem may be solved if the master device issues at least 9 clock pulses over the SCL line [24]. The slave device will receive the clock pulses such that it can continue its hardware state machine. After those 9 clock pulses, the state machine becomes in the idle state. In the idle state the slave device releases the SDA line such that the SDA line will be pulled up by the pull-up resistor, meaning that the health of the I²C bus is recovered.

A latch-up in the I²C controller electronics is an unwanted conductance that can be caused by a charged particle. The charged particle heats up the electronics, which may create a conducting plasma [13]. If one of these two events occurs and the power of the responsible I²C device is cycled, the I²C device releases the SDA line so that the line becomes in a high state. Once the line is in a high state, communication can be continued.

3.1.9 Clock Line Pulled Low Indefinitely

Another serious failure occurs when one of the I²C devices connected to the I²C bus pulls low the SCL line indefinitely. Like with pulling low the SDA line continuously, I²C communication is also not possible when the SCL line is pulled low continuously. Unfortunately, also this failure can happen anywhere in the system at any time. An I²C device can pull low the SCL line indefinitely when one of the following events occur:

- An electrical short between the ground and the SCL line
- Indefinite clock stretching
- A latch-up in the I²C controller electronics

The cause and impact of an electrical short between the ground and the SDA line is described above. For the SCL line the same impact holds: the SCL line is pulled low continuously, meaning that communication over the I²C lines is not possible anymore since the SCL line cannot become in the high state anymore.

The impact of a latch-up in the I²C controller electronics is already described above; the latch-up can pull the SCL line low for an indefinite period of time. A slave device can pull low the clock line indefinitely if it is stretching the clock and gets trapped in a software routine such that it cannot release the clock line. This problem can be solved by implementing time-out detection that will reset the I²C controller when a time-out occurs. Power cycling the I²C device that is pulling the SCL line low will also solve this problem [24].

3.1.10 I²C Failures Summary

All the I²C failures that are described in the sections above are summarized and listed in an overview in this section. The overview is shown in Table 3.1. The causes, impacts and possible resolve steps are listed as well.

Failure	Cause	Impact	Resolve step
Start condition failure	Bit flip on the SDA or SCL line	Slave devices may or may not recognize the start condition. When a start condition is not recognized the slave device does not ACK on a data transfer request	At this point the master cannot detect whether or not a slave device missed the start condition. The master must put the slave address on the bus to proceed
Slave request failure	Bit flip on the SDA line	A slave device does not acknowledge when it missed the start condition. The master may read a NACK while the slave is present, or another slave device that is present on the bus acknowledges. The latter may result in undesired operations and data flow	When the master receives a NACK the master must stop the data transfer by issuing a stop condition. Precautions for this failure can be taken by using 2 or more different bits between I^2C addresses such that the probability for this error to occur is decreased
R/$\overline{W}$ bit failure	Bit flip on the SDA line	Master and slave devices are waiting for each other for an ACK. Software at both master and slave side may enter an infinite loop if the software does not check for NACK	On both sides the software must stop the data transfer procedure when a NACK is read. To achieve this, the master must send out a stop condition to free the bus
Slave acknowledge failure	Bit flip on the SDA line	The master reads a NACK while the slave device is present and stops the data transfer, or the master reads an ACK while the slave device is not present and reads all ones (data bytes with value 0xFF) in case of a read operation	This does not affect the health of the I^2C bus so no resolve step needs to be taken
Data byte transfer failure	Bit flip on the SDA line	Bit errors occur when a bit flip happens on one or more of the first 8 clock pulses of a data byte transfer. When a bit flip occurs on the ninth clock pulse (ACK bit) a sequence of ones (data bytes with value 0xFF) will be read by the master during a read operation	This does not affect the health of the I^2C bus so no resolve step needs to be taken

Continued on next page

Table 3.1 – continued from previous page

Failure	Cause	Impact	Resolve step
Stop condition failure	Bit flip on the SDA or SCL line	If the slave is in slave-transmitter mode and misses the stop condition it still may write bytes on the bus since it not releases the bus lines	This is solved by the next I^2C operation, after the next stop condition has been issued by the master and the slave did not miss that stop condition
Slave misses clock pulses	Bit flip on the SCL line	The slave and the master device get out of sync and the slave device might pull the SDA line low indefinitely	Let the master send at least 9 clock pulses over the SCL line such that the slave I^2C controller its hardware state machine gets out its trapped state
Data line pulled low indefinitely	An electrical short between the ground and the SDA line, a lock-up in the I^2C controller its hardware state machine or a latch-up in the I^2C controller electronics	The SDA line is pulled low indefinitely such that data transfers over the I^2C lines are not possible anymore	Isolate the I^2C device in case of an electrical short. In case of a latch-up, a power cycle is sufficient. For the I^2C controller lock-up, the master device must issue 9 clock pulses such that the I^2C hardware state machine of the slave device that pulls the SDA line low becomes in the idle state and releases the SDA line
Clock line pulled low indefinitely	An electrical short between the ground and the SCL line, clock stretching by the slave device or a latch-up in the I^2C controller electronics	The SCL line is pulled low indefinitely such that data transfers over the I^2C lines are not possible anymore	Isolate the I^2C device in case of an electrical short. In case of a latch-up, a power cycle is sufficient. For clock stretching, a time-out must be implemented at the slave side such that it resets its I^2C controller and thus releases the SCL line

Table 3.1: *I^2C failures with their causes, impacts and resolve steps*

3.2 I^2C Bus Performance Analysis

Using engineering model test boards, an I^2C bus performance analysis has been performed. During the performance analysis, the stability and reliability of the I^2C bus was extensively tested. In Section 3.2.1 it is explained how the bit error rate is defined and in Section 3.2.2 the test setup is described. The procedure to measure the performance of the I^2C bus is given in Section 3.2.3 and Section 3.2.4 describes the verification of the performance procedure. Finally, in Section 3.2.5, the results are presented.

3.2.1 Bit Error Rate

A good measurement for I²C bus performance is the bit error rate (BER). The BER is defined as the number of incorrectly transferred bits divided by the total amount of transffered bits during the performance test. An incorrectly transferred bit may be caused by a bit flip on the SDA line or a malfunction in one of the I²C devices.

$$BER = \frac{B_{incorrect}}{B_{total}} \tag{3.1}$$

When one keeps track of the amount of transferred bits and the amount of incorrectly transffered bits, the BER can be computed using Equation 3.1. Any bit that is different than it is supposed to be is considered as an incorrectly transferred bit and will consequently increase the BER. Ideally the number of incorrectly transferred bits equals 0, which results in a BER of at most ${B_{total}}^{-1}$. A lower BER means higher performance and reliability metrics, which is of course desirable.

3.2.2 Test Setup

To successfully perform the BER measurement test, various kinds of hardware and software are needed. Since communication between subsystems is realized over the I²C bus, all the hardware that is used has to be I²C compatible. The following components are needed to perform the BER measurement:

- Exactly one microcontroller that acts as the I²C master device
- One or more microcontrollers that act as a I²C slave device
- One I²C sniffer, which is in our case implemented in FPGA technology
- Cables to connect the I²C devices
- Serial cable (RS232) to connect the I²C sniffer device to a computer

In this setup, 10 microcontrollers are used that act as slave device that simulate subsystems. The I²C master device communicates with the I²C slave devices by initiating data transfers. Data is transferred from the master device to the slave devices and the other way around such that I²C write operations and I²C read operations are both executed during the performance test. The I²C sniffer shows all the transferred data (including I²C addresses) in real time on the display of the computer. The I²C sniffer is not a master device nor a slave device. It only listens on the I²C bus to sniff all the data that is transferred over the bus, so it is invisible to other devices. For this test there are two kinds of software needed:

- Software for the microcontrollers, including I²C service layer software and test application layer software that uses the I²C service layer software to measure the BER
- Software for the I²C sniffer (i.e. I²C Monitor software)

The microcontroller software that is needed must be able to initialize a microcontroller as an I^2C master or I^2C slave device. Furthermore this software must include procedures to send and receive data over the I^2C bus. With these basic procedures one is able to design test software on the application layer level to perform BER measurements. With the I^2C monitoring software it is possible to read the number of transferred bits and the number of incorrectly transferred bits if the test software is designed such that these numbers are transferred over the bus. When these numbers are transferred over the bus the I^2C monitor software will display these numbers on the screen of the computer. The user can in turn compute the BER using Equation 3.1.

3.2.3 Test Procedure

The test procedure needed to measure the BER consists of a couple of steps. In order to measure the BER, communication is needed between the master device and the 10 slave devices that simulate subsystems. The microcontroller that is the I^2C master must keep track of the number of incorrectly transferred bits and total amount of transferred bits per slave device. With this approach it can be determined which slave device is introducing errors on the I^2C bus. The test consists of the following steps, which are executed iteratively till the end of the test:

- The master device sends a request to a slave device for data acquisition, it also send the contents of the 'number of transferred bits' and the 'number of incorrect bits' variables. This is an I^2C write operation that tests the reliability from master to slave.
- The requested slave device responds to the request by putting a block of known, fixed, data on the bus that is defined a priori. Besides that, the slave will send back the content of the 'number of transferred bits' and the 'number of incorrect bits' variables back to the master.
- The master collects the data from the slave and checks the data for correctness. The master is able to do this since it knowns what data it should receive (i.e. the correct data).
- The master updates the 'number of transferred bits' variable for the specific slave.
- The master updates the 'number of incorrect bits' variable for the specific slave if one or more bits are incorrect. The master can determine the amount of incorrect bits by applying an XOR operation to the expected (correct) data and the received data. The number of incorrect bits will be equal to the amount of ones in the XOR result and the 'number of incorrect bits' variable will be updated accordingly.

The number of transferred bits and the number of incorrect bits must be send to a slave device in order to be able to sniff it using the I^2C sniffer. The I^2C monitor software will display these values on the screen such that the user can use them to compute the BER. The test data that will be send from a slave device to the master device is known, so that the master device can check the received data from the slave for correctness.

Based on this check the number of incorrect bits can be maintained per slave device, after which the BER can be computed individually for each slave device.

3.2.4 Software Verification

Before the BER test software is being used it is important to test and verify the functionality of the software. This can be accomplished by running the software with the procedure described above and a switch. By placing a switch between the SDA line and the ground (see Figure 3.5), bit errors can be introduced on the SDA line intentionally by pressing the switch such that the SDA line is shorted to the ground.

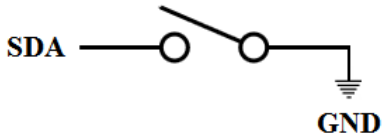


Figure 3.5: *Switch to pull low the SDA line intentionally*

During the BER measurement software test, the switch must be tipped very shortly such that the SDA line is pulled low for a small amount of time. The same mechanism can be used to introduce errors on the SCL line. In order to fully verify the functionality of the BER measurement software, the following actions must be performed:

- Run the BER measurement procedure as discussed in the previous section.
- Tip the switch shortly while the BER measurement software is running such that bit errors are introduced.
- Watch the content of the sniffed data that is send from the slave to the master using the I^2C sniffer.
- Check the number of occurred bit errors that was counted by the BER measurement software.
- Verify whether or not the number of counted bit errors is correct. This can be done by counting the number of bit errors in the displayed sniffed data by hand. This can be performed since the transferred data is known a priori. The number of counted bit errors by the BER measurement software that is being displayed in the sniffed data by the I^2C monitor program must be equal to the number of bit errors that were counted by the test user that is verifying the BER measurement software. If the numbers are equal, it is verified for this particular test run that the BER measurement software is working properly.

The verification procedure described above should be performed several times with different slaves and different chunks of data. By following the above described procedure several times, it has been verified that the BER measurement software works correctly. This means that the developed BER measurement software can be used reliably in order to produce some real performance results.

3.2.5 Performance Results

In a preliminary phase of I^2C service layer software development, several performance tests were already performed in order to check the performance and verify the correctness of the hardware and software designs of the MSP430F1611 and the DSSB microcontrollers in terms of I^2C transfers. Several short duration tests (less than 4 hours) have been performed with the following setups:

- Transfers of short data packages to and from only 1 slave device
- Transfers of long data packages to and from only 1 slave device
- Transfers of short data packages to and from multiple slave devices
- Transfers of long data packages to and from multiple slave devices

Short data packages are packages of 10 bytes or less. The test with short data packages measures how well the I^2C hardware and software behave with a lot of I^2C overhead data (i.e. stop conditions, slave address request and stop conditions). The test with longer data packages measures how well the I^2C hardware and service layer software are able to handle incoming and outgoing I^2C data transfers. Transfers of longer data packages may increase the probability of clock stretching and hence may influence the performance of the I^2C bus. In the tests with multiple slave devices, 6 slave devices were used during the test. All the four tests described above resulted in a bit error rate of at most 10^{-9} because 1 billion bytes were transferred and no bit errors were detected.

Later on during the software development of the OBC, more hardware became available. The new hardware included more engineering model test boards and cables that could be used to easily connect multiple test boards to one another. With this setup, and a more completed OBC with a significant amount of service layer software and application layer software finished, a more representative performance test could be performed. The more representative performance test setup was as follows:

- Long duration test (around 16 hours)
- Long packages and short packages mixed
- Multiple slave devices (12 slave devices)

The above described test was performed 10 times in order to determine the bit error rate with the more representative setup reliably. For each test run, a total of 5 billion bits were transferred from the OBC to the 12 different microcontrollers (Texas Instruments MSP430F1611 and Atmel AVR88PA) and vice versa. In 2 out of the 10 tests, bit errors were detected. One test showed 1 bit error and the other test showed 2 bit errors. During the other 8 tets, no bit errors were detected so it seems that bit errors occur accidentally. Since during one test 2 bit errors were measured, the bit error can be considered to be at most $2 \cdot (5 \cdot 10^9)^{-1} = 4 \cdot 10^{-9}$.

3.3 Summary

This Chapter presented a failure analysis and performance analysis on data transfers that take place over the I²C bus.

For the I²C bus failure analysis, failures that are most likely to occur were extensively analyzed. The failures that were examined are failures that may occur during a data transfer and failures that may occur at any time.

Failures that may occur during a data transfer are failures in the start condition, slave address request, read/write bit assertion, slave device acknowledgement, data byte transfers and stop condition. The failure case in which a slave misses a clock pulse also belongs to this category. Failures like these are mostly caused by bit flips on the SDA or SCL lines and do not much harm. They can end up in a failure of the current data transfer, but they can not harm the health of the I²C bus permanently.

The I²C bus health can be put into real danger by failures that may occur at any time. These failures pull low the SDA and/or the SCL line for a longer period of time. They are usually caused by an electrical short between the lines and the ground, a lock-up in the I²C controller hardware state machine, indefinite clock stretching by a poorly implemented or configured I²C slave device, or a latch-up in the I²C controller electronics. In the case of an electrical short, the slave device causing the failure must be isolated from the I²C bus in order to recover the I²C bus health. With the other causes, the failure can usually be solved by power cycling the slave device that is responsible for the failure.

The I²C performance analysis was performed by measuring the bit error rate for transfers between the master and the slaves. The software used for this analysis counts the number of incorrectly transferred bits and the total number of transferred bits. Using these two numbers, the bit error rate can be computed by dividing the amount of incorrectly transferred bits by the total number of transferred bits. All the 10 tests were long duration tests of around 16 hours in which 5 billion bits were transferred. The bit error rate measurement software has been verified and the tests resulted in a bit error rate of at most $4 \cdot 10^{-9}$ (2 bit errors out of 5 billion transferred bits).

On-Board Computer Software Design

4

This chapter describes the baselined OBC software design of the Delfi-n3Xt Nanosatellite. In Section 4.1 the requirements of the OBC software are given. Section 4.2 gives a detailed overview of the OBC software architecture. The design of the OBC service layer software is described in Section 4.3. Finally, in Section 4.4, the design of the application layer software of the OBC is given.

4.1 OBC Software Requirements

In this section the requirements for the OBC software are described. The requirements are adopted from the CDHS Top Level Design document [9] and the Delfi-n3Xt Requirements and Configuration Item List [19] and are split up into five different categories; the Subsystem Communication Requirements 4.1.1, Fault-Tolerant Software Requirements 4.1.2, Telecommanding Requirements 4.1.3, Data Acquisition Requirements 4.1.4 and Monitoring Requirements 4.1.5. All the requirements are listed including their requirement code, inherited parent requirements codes and rationale.

4.1.1 Subsystem Communication Requirements

This section states the requirements that are related to the communication between the OBC and the various subsystems present in the Delfi-n3Xt Nanosatellite.

4.1.1.1 The I^2C bus protocol must be used for subsystem communications

Requirement code	*SAT.2.6-C.01* (constraint)
Parent #1	*MIS-F.02* (Delfi-C^3 also used non I^2C-compliant devices)
Parent #2	*SAT-C.05* (the interfaces must be I^2C-compliant)
Parent #3	*SAT.2-C.01* (a uniform bus standard is more reliable)

Rationale
During the CDHS top level system design it is decided that the I^2C protocol is used for communications between the OBC and the other subsystems. No adaptations to the I^2C bus protocol are allowed. This means that the I^2C design guidelines must be strictly followed.

4.1.1.2 The I²C bus must operate in standard mode

Requirement code	*SAT.2.6.1-C.01* (constraint)
Parent #1	*SAT.2.6-C.01* (no adaptations to the I²C bus are allowed)

Rationale
The I²C bus protocol in standard mode operates at a data rate of 100kbit/s. Higher data rates may result in a higher amount of bit errors and more power consumption. The I²C bus must be as reliable as possible; therefore the I²C bus protocol will be implemented in standard mode. It is investigated that a data rate of 100kbit/s is sufficient for full data acquisition and telemetry [9].

4.1.1.3 Switch subsystems on, off or in different modes

Requirement code	*SAT.2.6.2.1-F.01* (functional requirement)
Parent #1	*SAT-F.03* (the OBC implements the switching of subsystems)
Parent #2	*SAT.2.6-F.02* (switching is required for testing and verifying)
Parent #3	*SAT.2-C.01* (the OBC switches off malfunctioning subsystems)

Rationale
Subsystems have different modes and each subsystem can be turned on and off (except for the CDHS where always at least one OBC is turned on). The subsystems can be switched on, off or into different modes by communicating with the subsystem its controllers over the I²C bus. Subsystems must be switched on or off or in different modes according to the operational mode of the satellite.

4.1.1.4 A failing I²C operation must time-out after 30ms

Requirement code	*SAT.2.6.2.1-P.02* (performance requirement)
Parent #1	*SAT.2-F.01* (a suitable time-out period facilitates payloads)
Parent #2	*SAT.2.6-F.05* (a time-out is required for correct data flow)

Rationale
During the CDHS top level system design it is calculated that an I²C operation must time-out after 30ms [9]. The largest possible I²C data transfer will consist of 1 address byte and 255 data bytes. The address byte and the 255 data bytes all must be acknowledged by 1 acknowledge bit, so a total of $256 \times 9 = 2304$ bits will be transferred during the largest possible I²C data transfer. At a bus speed of 100kbit/s such a data transfer should be finished within 23.04ms when no clock stretching occurs. In order to be safe and take minimal clock stretching into account, an I²C operation time-out period of 30ms has been defined.

4.1.1.5 The bit error rate of the I^2C bus must be at most 10^{-6}

Requirement code	*SAT.2.6.1.3-P.01* (performance requirement)
Parent #1	*SAT.2-C.01* (a low bit error rate ensures reliability)
Parent #2	*SAT.2.3-P.02* (related to the bit error rate of the COMMS)

Rationale
In the top level design it is defined that a data bus with a bit error rate (BER) of 10^{-6} or less can be considered as reliable enough for Delfi-n3Xt. The BER is defined as the ratio of the number of erroneous bits and the total number of bits transferred. The I^2C communication software must be optimized such that the combination of hardware and software results in a bit error rate of at most 10^{-6}.

4.1.2 Fault-Tolerant Software Requirements

This section states the requirements that are related to the fault-tolerant implementation of the OBC software.

4.1.2.1 The OBC software must be fault-tolerant

Requirement code	*SAT.2.6.2.1.3-F.03* (functional requirement)
Parent #1	*SAT.2.6.2-F.03* (acquire housekeeping data)
Parent #2	*SAT.2.6.2-F.04* (acquire payload data)
Parent #3	*SAT.2-C.01* (loss of the OBC is unacceptable)

Rationale
The OBC software must be fault tolerant in the sense that an OBC software fault must not lead to undefined states. Software that is trapped in an undefined state may cause loss of the satellite. This is unacceptable since the OBC is the central control unit of the satellite. Loss of the OBC means loss of the entire satellite.

4.1.2.2 The OBC software must be redundant

Requirement code	*SAT.2.6.2-F.09* (functional requirement)
Parent #1	*SAT.2-C.01* (loss of the OBC is unacceptable)

Rationale
Single-Point-of-Failures (SPoFs) in the OBC must be avoided, since loss of the OBC means loss of the satellite. Therefore the OBC must be fully redundant, resulting in a primary and a secondary OBC. Exactly one of the OBCs must be the I^2C master device, since a design with only one I^2C master device is easier to implement and less error prone compared to a design with multiple master devices. Initially, the primary OBC takes full control over the I^2C bus lines. When the primary OBC fails in its operations, the secondary OBC must take over.

4.1.2.3 Delay between POD ejection and activation of the radios

Requirement code	*SAT.2.6.2-C.02* (constraint)
Parent #1	*SAT-C.06* (the delay is launcher dependent)
Parent #2	*SAT-F.02* (the delay is launcher dependent)

Rationale
The launcher that will be used requires a delay between the POD ejection and the activation of the transmitters and receivers of the satellite. This delay is launcher dependent and for the launcher that will be used to launch Delfi-n3Xt a delay of 10 minutes must be implemented [15].

4.1.2.4 Delay between POD ejection and deployment

Requirement code	*SAT.2.6.2-C.03* (constraint)
Parent #1	*SAT-C.06* (the delay is launcher dependent)
Parent #2	*SAT-F.02* (the delay is launcher dependent)

Rationale
The launcher that will be used requires a delay between the POD ejection and the deployment of the antennas and the solar panels of the satellite. This delay is launcher dependent and for the launcher that will be used to launch Delfi-n3Xt a delay of 10 minutes must be implemented [15].

4.1.3 Telecommanding Requirements

Requirements that are related to telecommanding are described in this section.

4.1.3.1 Acquire, interpret and execute incoming telecommands

Requirement code	*SAT.2.6.2-I.01* (interface requirement)
Parent #1	*SAT-F.01* (the ground station transmits telecommands)
Parent #2	*SAT-F.01* (the PTRX and/or ITRX receive telecommands)
Parent #3	*SAT-F.01* (the OBC must process the telecommands)

Rationale
Telecommands received from the ground station need to be interpreted and executed by the OBC. A telecommand can be send from a ground station to the PTRX and/or the ITRX. The PTRX and/or ITRX receive the telecommand and store the INFO field of the AX.25 frame [22] in a buffer such that the OBC can fetch the data from the PTRX and/or ITRX. The OBC software has to interpret and execute the telecommand.

4.1.3.2 Acknowledge the last received telecommand

Requirement code	*SAT.2.6.2-F.01* (functional requirement)
Parent #1	*SAT-F.01* (acknowledgement of reception is required)

Rationale
Various faults (e.g. bit flips) can occur during the transfer of a telecommand from a ground station to the satellite [2]. This is the reason why the last received telecommand needs to be acknowledged back to the ground station. By doing this the engineers can check whether or not the desired telecommand is received properly.

4.1.3.3 Acknowledge the last executed telecommand

Requirement code	*SAT.2.6.2-F.02* (functional requirement)
Parent #1	*SAT-F.01* (acknowledgement of reception is required)

Rationale
It is important to know which command is executed, since the execution of a telecommand can change the behavior of the satellite. Therefore the last executed telecommand is also acknowledged back to the ground station. With this feature engineers can detect faults and take action if needed.

4.1.3.4 Parameters are configurable by telecommands

Requirement code	*SAT.2.6.2.1.3-I.01* (interface requirement)
Parent #1	*SAT-F.01* (communication functionality with ground system)
Parent #2	*SAT.2-C.01* (erroneous software may be reparable)
Parent #3	*SAT.2.6.2-F.07* (the decoder may have parameter alterations)

Rationale
Various parameters are stored in the memory of the OBC and other subsystems. The configurable parameters may be stored in the OBC flash memory, the OBC random access memory (RAM) or in the flash memory or the RAM of the controllers of a subsystem. Reconfiguration of these parameters may influence the functional behavior of the satellite.

4.1.4 Data Acquisition Requirements

In this section the requirements for the data acquisition are described.

4.1.4.1 Collect all housekeeping data from other subsystems

Requirement code	*SAT.2.6.2-F.03* (functional requirement)
Parent #1	*SAT.2-F.02* (the OBC must acquire housekeeping data)
Parent #2	*SAT.2.6-F.02* (housekeeping data is required for verification)
Parent #3	*SAT.2.6-F.05* (data flow must be initiated by the OBC)

Rationale
The OBC is the central control unit of the satellite, and therefore one of the tasks of the OBC is to collect all housekeeping data from the other subsystems. All these housekeeping data, together with the payload data and the time tag must be put together in one large data frame. This frame can then be forwarded to the PTRX or ITRX and the STX such that it can be send to the ground station.

4.1.4.2 Collect all payload data from other subsystems

Requirement code	*SAT.2.6.2-F.04* (functional requirement)
Parent #1	*SAT.2-F.02* (the OBC must acquire payload data)
Parent #2	*SAT.2.6-F.02* (payload data is required for verification)
Parent #3	*SAT.2.6-F.05* (data flow must be initiated by the OBC)

Rationale
The OBC is the central control unit of the satellite, and therefore one of the tasks of the OBC is to collect all payload data from the other subsystems. All these payload data, together with the housekeeping data and the time tag must be put together in one large data frame. This frame can then be forwarded to the PTRX or ITRX and the STX such that it can be send to the ground station.

4.1.4.3 Data acquisition must be initiated each 2 seconds

Requirement code	*SAT.2.6.2.1.3-P.01* (performance requirement)
Parent #1	*SAT.2-C.01* (the data bus must adhere to reliability standards)

Rationale
The OBC has to acquire new payload data and housekeeping data from the subsystems every two seconds. It is decided that it is sufficient to collect data from the subsystems (both payload data and housekeeping data) with a frequency of 0.5Hz [9]. Therefore the control loop must be designed such that data acquisition is initiated each 2 seconds. This has to be accomplished with an accuracy of at least 1ms.

4.1.4.4 Payload and housekeeping data must be initiated with dummy data

Requirement code	*SAT.2.6.2.1.3-F.01* (functional requirement)
Parent #1	*SAT.2.6-F.05* (testing the register is part of enabling data flow)
Parent #2	*SAT.2.6.2-F.04* (acquired data is stored into the data register)
Parent #3	*SAT.2.6.2.1-P.01* (the capacity has to be tested)

Rationale
Distinguishable dummy data should be present in all payload data and housekeeping data registers that the OBC holds. When a subsystem does not react in a certain period of time, the payload data or housekeeping data register in the memory of the OBC corresponding to that particular subsystem will contain the dummy data. Whether or not a subsystem works properly can be deduced from the received payload or housekeeping data at the ground station. In the case that the subsystem is not reachable, the payload data or housekeeping data registers will contain the dummy data. In the case that the subsystem works properly the payload data or housekeeping data registers will contain other data than the dummy data.

4.1.4.5 The OBC must determine which radio is used for data transmission

Requirement code	*SAT.2.6.2.1.3-F.02* (functional requirement)
Parent #1	*SAT-F.01* (the OBC determines the communication link to use)
Parent #2	*SAT.2.3-F.02* (the OBC supplies data to the PTRX or ITRX)

Rationale
Only one of the transmitters (PTRX or ITRX) must be used to transmit telemetry data to the ground station. Depending on the operational status of the PTRX and ITRX, the OBC has to make the decision to which transmitter the data will be send. The telemetry data consists of unique time tag data, payload data and housekeeping data.

4.1.4.6 Telemetry data must be send to the PTRX or ITRX and the STX

Requirement code	*SAT.2.6.2-F.05* (functional requirement)
Parent #1	*SAT-F.01* (this step is required in the communication link)
Parent #2	*SAT.2.3-F.02* (PTRX, ITRX and STX receive telemetry data)
Parent #3	*SAT.2.6.2.1-P.01* (the capacity has to be tested)

Rationale
The OBC must send all telemetry data to the PTRX or the ITRX (not both at the same time) and it should always send the telemetry data to the STX. This is needed such that the telemetry data is received at the ground station.

4.1.4.7 A unique time tag must be added to the telemetry data

Requirement code	*SAT.2.6.2-F.06* (functional requirement)
Parent #1	*MIS-F.02* (improvement with respect to Delfi-C^3)

Rationale
Unique timestamps can be very useful when data is being analyzed at the ground station. Each telemetry data frame must hold a unique time tag such that different frames can be distinguished from one another.

4.1.5 Monitoring Requirements

In this section the requirements for on-ground monitoring of the satellite are described.

4.1.5.1 The OBC must be monitored through a test interface

Requirement code	*SAT.2.6-F.04* (functional requirement)
Parent #1	*SAT-C.05* (the test interface is a required interface)
Parent #2	*SAT-F.03* (switching must be verified using a test interface)
Parent #3	*SAT.2-C.01* (testing ensures that the system is reliable)

Rationale
On-ground testing is a very important task in the development of a satellite. Since the OBC must be able to be monitored through a test interface, test software needs to be developed. The test software for the OBC must monitor the behavior and data flow of the satellite during integration and verification of the satellite.

4.2 OBC Software Architecture

The architectural overview of the OBC software is discussed in this section. The OBC software is subdivided in three different software layers. These software layers, together with the hardware interfacing, are discussed in Section 4.2.1. Furthermore the OBC software is split up into different modules. Each software layer contains several modules. The module overview of the OBC software is shown and described in Section 4.2.2

4.2.1 OBC Software Layers

The OBC software is subdivided in three different software layers. These layers are named the application layer, the subsystem layer and the service layer. The service layer software is the lower level software. It interfaces with the hardware and consists of software modules that implement the driving of the MSP430 hardware peripherals. The application layer software and subsystem layer software are the higher level software. These layers make use of the service layer software to implement the correct data flows in order to form the complete system. The subsystem layer is actually a sublayer within the application layer, as shown in the high level architectural overview of the OBC

software in Figure 4.1 In this figure it can be clearly seen that the subsystem layer is a sublayer within the application layer and that these layers interact with the service layer. Furthermore it can be seen that the service layer interfaces with the MSP430 hardware layer. The *external data in* and *external data out* arrows represent the incoming and outgoing data flows from and to the other subsystems of the satellite. Incoming I^2C data flows from the bottom to the top (i.e. from the MSP430 hardware layer to the service layer, and then from the service layer to the application layer and subsystem layer). The outgoing I^2C data flows from the top to the bottom (i.e. from the application layer and the subsystem layer to the service layer, and eventually to the MSP430 hardware layer where the data is written on the bus).

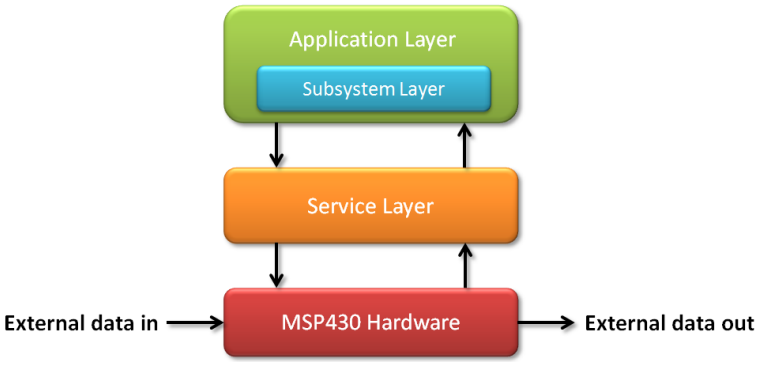


Figure 4.1: *High level architectural overview of the OBC*

The application layer software and service layer software can be further subdivided into software modules. The MSP430 hardware layer can be further subdivided into the central processing unit (CPU) and the peripherals that are listed in Section 2.3.1.

4.2.2 OBC Module Overview

As already mentioned earlier, the service layer software implements the driving of the MSP430 hardware peripherals. In Section 2.3.1 the hardware peripherals of the MSP430 microcontrollers that are used for the OBC are listed. Not all the peripherals of the microcontroller need to be used. Server layer software modules must be implemented for the hardware peripherals that are being used by the Delfi-n3Xt OBC application layer software. The following service layer software modules are implemented:

- Clock source module
- Programmable interval timers module
- Flash memory controller module
- Analog to digital converter module
- I^2C controller module
- Watchdog timer module

These software modules that are part of the service layer software are represented graphically in Figure 4.2. The service layer software modules are mapped to the corresponding hardware peripherals of the MSP430 microcontroller.

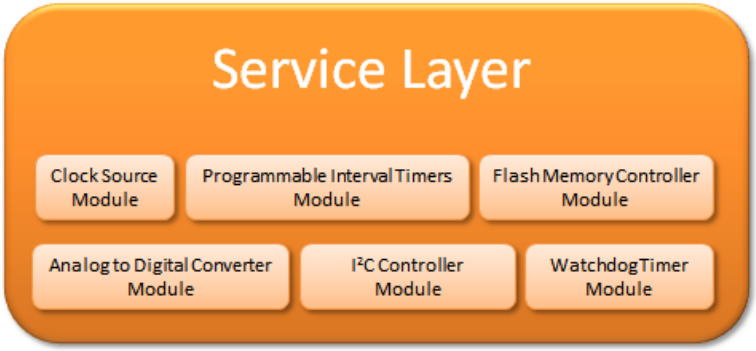


Figure 4.2: *Module overview of the OBC service layer software*

From Figure 4.1 it can be seen that the subsystem layer is actually a sublayer within the application layer. This sublayer is introduced in order to improve the modularity of the OBC software. The subsystem layer contains software modules that are responsible for the commanding of the subsystems of the satellite. In this layer a software module is present for each satellite subsystem. The software modules contain commanding functionalities for the corresponding subsystem. The following software modules are present within the subsystems layers:

- OBC module
- ADCS module
- DAB module
- DSSB module
- EPS module
- ITRX module
- PTRX module
- STX module
- SDM module
- $T^3\mu PS$ module
- TCS module

In general, these software modules contain functions that command the corresponding subsystem and acquire housekeeping data or payload data from the subsystem. The DSSB software module is a special one. The DSSB is actually not a subsystem. The

DSSB software module is present because it contains functionalities related to the DSSB microcontrollers, which are identical (except for the I^2C address) and present at all subsystems. The subsystems modules are not described in detail in this thesis.

Besides the subsystem layer, the application layer contains software modules that are responsible for executing tasks that correspond to the current operational mode of the satellite. In total there are five different operational mode modules:

- Boot mode module
- Delay mode module
- Deployment mode module
- Main mode module
- I^2C recovery mode module

The operational modes are all described in the OBC Application Layer Software Design section (Section 4.4). The OBC application layer software modules (including the subsystem layer and its software modules) are graphically represented in Figure 4.3.

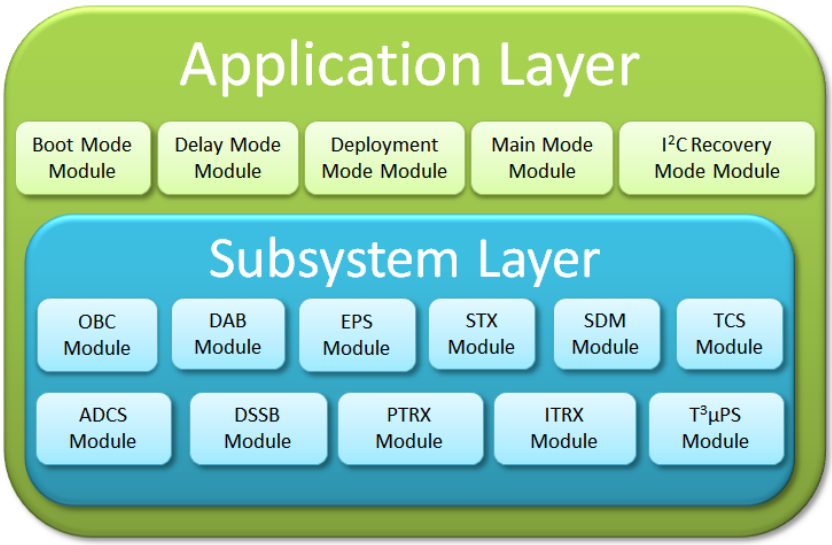


Figure 4.3: *Module overview of the OBC application layer software*

4.3 OBC Service Layer Software Design

This section gives the detailed design of the OBC service layer software modules. First the design of the clock source module is given in Section 4.3.1. The programmable interval timer module design is described in Section 4.3.2. Section 4.3.3 gives the design of the flash memory controller module. The analog to digital converter module is discussed in Section 4.3.4. Finally, the design of the I^2C controller module is given in Section 4.3.5 and the design of the watchdog timer is discribed in Section 4.3.6.

4.3.1 Clock Source Module

The input clock signal for the MSP430F1611 CPU and its peripherals can be selected using the clock source module. This Section describes the MSP430F1611 clock system and the clock source software module design.

4.3.1.1 MSP430 Clock System

Three clock sources can be used to drive the MSP430F1611 microcontroller CPU and peripherals [28]. These three clock sources are:

- **LFXT1CLK**
  This clock source is an external oscillator that can be either a low-frequency 32768 Hz watch crystal or a standard high-frequency crystal or resonator in the 450 kHz to 8 MHz range.
- **XT2CLK**
  This clock source is an external high-frequency oscillator that can be used with standard crystals, resonators, or external clock sources in the 450 kHz to 8 MHz range.
- **DCOCLK**
  This clock source is the on-chip internal digitally controlled oscillator (DCO) with RC-type characteristics which can be configured by software or external resistors.

The MSP430F1611 basic clock module can be configured such that the microcontroller operates with or without any external crystals or resonators. The OBC hardware has an external low-frequency 32768 Hz watch crystal as the LFXT1CLK clock source and an external high-frequency 8 MHz crystal as the XT2CLK clock source. The DCO is not used since this clock source has RC-type characteristics. Oscillators with RC-type characteristics are known to be less accurate under certain circumstances. Their frequencies vary with temperature, voltage and from device to device [1]. These kind of oscillators are certainly not appropriate for the OBC of Delfi-n3Xt, since the calculated OBC temperature range varies from $-15.6\,^{\circ}\mathrm{C}$ to $+22.22\,^{\circ}\mathrm{C}$ [17] and accurate timing is a requirement.

The three clock sources shown above can be used to drive the three clock signals that are available from the basic clock module. These three clock signals are:

- **ACLK**
  This clock signal is called the auxiliary clock. The auxiliary clock is the buffered LFXT1CLK clock source divided by 1, 2, 4 or 8. The clock source divider can be selected by software and the clock signal is software selectable for individual peripherals.
- **MCLK**
  This clock signal is called the master clock. The master clock is software selectable as LFXT1CLK, XT2CLK or DCOCLK. Furthermore the clock source divider can be selected by software and the clock source divider can be 1, 2, 4 or 8. The master clock is always used by the CPU of the microcontroller.

- **SMCLK**
  This clock signal is known as the sub-main clock. The sub-main clock is, like the master clock, software selectable as LFXT1CLK, XT2CLK or DCOCLK. The sub-main clock divider can be selected by software as 1, 2 ,4 or 8. Furthermore, the sub-main clock is software selectable for individual peripherals.

The ACLK clock signal is automatically sourced by the LFXT1CLK clock source. For Delfi-n3Xt, the MCLK and SMCLK signals must be sourced by the high-frequency 8 MHz external crystal since the CPU of the microcontroller and some of the peripherals must run at high enough frequencies.

4.3.1.2 Clock Module Design

The aim is to design the clock module software such that the LFXT1CLK clock source is configured in low-frequency mode and the MCLK and SMCLK clock signals are configured such that they use the high-frequency external 8 MHz crystal that sources the XT2CLK. This is shown graphically in the activity flow diagram in Figure 4.4.

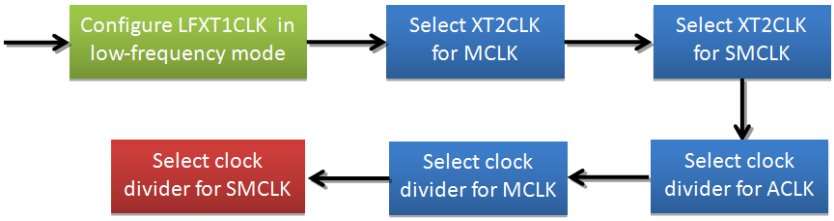


Figure 4.4: *Activity flow for configuring the MSP430 clock system of the OBC*

The actions shown in the activity flow diagram above can be executed by configuring the MSP430F1611 clock module registers.

4.3.2 Programmable Interval Timers

Programmable Interval Timers (PITs), or just timers in short, can be used to generate intervals for e.g. the execution of periodic functions or the handling of time-outs. This can be achieved by setting a flag in the Interrupt Service Routine (ISR) of the timer after a certain amount of time. The application layer software can poll for this flag. In this section the basics about timers is described. It is also discussed how the interval time of the timer can be calculated given a desired threshold value, and how the threshold value can be calculated given a certain desired time interval.

4.3.2.1 Timer Basics

Timers are usually 8-bit or 16-bit. Therefore timers can count up to a maximum 8-bit value of 2^8-1 = 255 = 0xFF in the case of a 8-bit counter or a maximum 16-bit value of 2^{16}-1 = 65535 = 0xFFFF in case of a 16-bit timer. A timer capture/compare interrupt is generated when the internal timer tick counter of the PIT reaches the configured threshold value. This interrupt is generated by the timer peripheral and the interrupt

can be handled in the ISR of the timer. The MSP430F1611 microcontroller on the OBC of the Delfi-n3Xt Nanosatellite has two 16-bit timers; timer A and timer B.

There are various modes in which a timer can operate. However, we only consider the *up mode* and *up/down mode* here since these are the only interesting modes for implementation. In the *up mode* the timer just counts up until the desired threshold value is reached. This threshold value is stored in a register (named *TACCR0* for timer A, and *TBCCR0* for timer B). The timer just actively compares the amount of elapsed timer ticks with this configured threshold value. Once the value is reached, the timer will generate a *timer capture/compare interrupt* which can be handled by the timer ISR. The operation of timer A in *up mode* is shown in Figure 4.5.

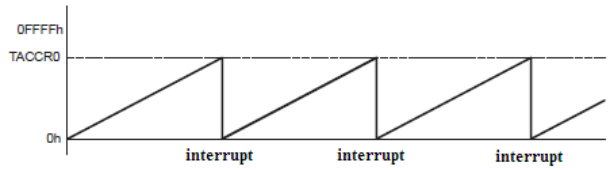


Figure 4.5: *16-bit timer A operating in up mode and generating interrupts*

In the *up/down mode* the timer counts up until the configured threshold value stored in the *TACCR0* register is reached, and then counts back to 0. The timer will generate an interrupt once the timer tick counter is back at 0. In this timer mode the desired interval is exactly two times longer compared to the interval generated in the *up mode*. The operation of timer A in *up/down mode* is shown in Figure 4.6.

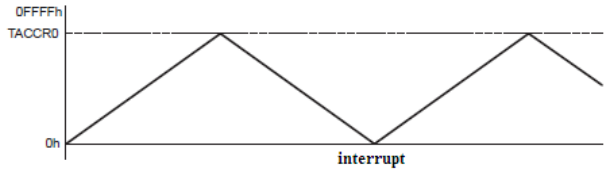


Figure 4.6: *16-bit timer A operating in up/down mode and generating interrupts*

From this figure it is clear that the interrupt in *up/down mode* occurs twice as late as the interrupt that occurs in *up mode*.

Normally, timers can be configured to use a clock divider in order to achieve longer timer interval periods. On the MSP430F1611, the clock divider can be configured such that the clock input is divided by 1, 2, 4 or 8. With a clock divider of 1 the timer will count up or down each clock cycle of the input clock. With a clock divider of 2, 4 and 8 the timer will count up or down each 2, 4 or 8 clock cycles of the input clock, respectively.

4.3.2.2 Timer Intervals

As already discussed in Section 4.3.1.1, the two clocks connected to the MSP430 microcontroller of the OBC are a 32768 Hz low frequency crystal and a 8 MHz high frequency crystal. The low frequency crystal will be used for the watchdog timer as described in Section 4.3.6 and the high frequency crystal is used for the timers. A frequency of 8 MHz

results in a clock period of 8000000^{-1} s, or 125 ns. Now let t_{th} be the threshold value stored in the *TACCR0* register, f_{clk} the clock frequency in Hz and clk_{div} the configured clock divider. Then, for the *up mode*, the timer interval $t_{i_{up}}$ in seconds becomes

$$t_{i_{up}} = \frac{t_{th} \cdot clk_{div}}{f_{clk}} \tag{4.1}$$

and for the *up/down mode* the timer interval $t_{i_{up/down}}$ is simply the double of the timer interval as computed in Equation 4.1 and thus becomes

$$t_{i_{up/down}} = 2 \cdot t_{i_{up}} = 2 \cdot \frac{t_{th} \cdot clk_{div}}{f_{clk}} \tag{4.2}$$

With the high frequency 8 MHz crystal, the longest possible interval can be achieved with the timer in *up/down mode*, the threshold value set to 2^{16}-1 and the clock divider set to 8 which is the largest possible clock divider. Putting these values in Equation 4.2 gives the longest possible interval and this interval $t_{i_{max}}$ will become

$$t_{i_{max}} = 2 \cdot \frac{(2^{16} - 1) \cdot 8}{8000000} \approx 131ms \tag{4.3}$$

Now, by rearranging Equation 4.2, the threshold value t_{th} can be computed for a given timer interval $t_{i_{up/down}}$. Solving Equation 4.2 for t_{th} gives

$$t_{th} = \frac{t_{i_{up/down}} \cdot f_{clk}}{clk_{div} \cdot 2} \tag{4.4}$$

With Equation 4.4, the threshold value for any timer interval in the range $[0, t_{i_{max}}]$ can be computed when the timer is configured to operate in *up/down mode*.

4.3.2.3 Timer Module Design

A timer can be intialized and started by setting the timer threshold value, selecting the clock source for the timer, select the clock source divider, setting the timer mode (e.g. *up mode* or *up/down mode*) and finally enabling the interrupt of the timer. This is shown in the activity flow diagram in Figure 4.7.

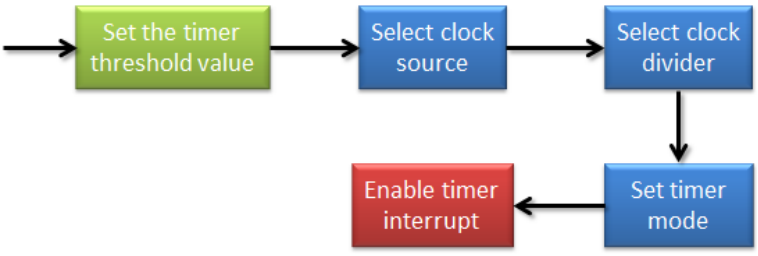


Figure 4.7: *Activity flow for the configuration of a timer*

Stopping the timer is simple and can be done by disabling the timer interrupt. Disabling the timer interrupt disables the complete timer peripheral such that it does not consumes any power.

4.3.3 Flash Memory Controller

The flash memory controller peripheral of the MSP430F1611 can be used to store OBC parameters in the flash memory of the microcontroller. The microcontroller has an on-chip flash memory of 48 kB. Flash memory is a non-volatile memory type, i.e. it keeps the contents of its memory when the system is unpowered. This can be useful when OBC parameters must be loaded after a reboot of the OBC. In this section the flash memory segmentation of the MSP430F1611 is given (see Section 4.3.3.1), including the design of the functionalities needed for the flash memory controller service layer software module (Section 4.3.3.2).

4.3.3.1 Flash Memory Segmentation

The MSP430F1611 has 48KB + 256B of flash memory and 10KB of Random Access Memory (RAM) [1]. The 48KB chunk of flash memory is called the main memory and the additional 256B of flash memory is called the information memory. The 48 KB main flash memory starts at address 0xFFFF and stops at address 0x4000. The 256 bytes of information flash memory starts at address 0x10FF and stops at address 0x1000. Furthermore, the MSP430 flash memory is memory mapped.

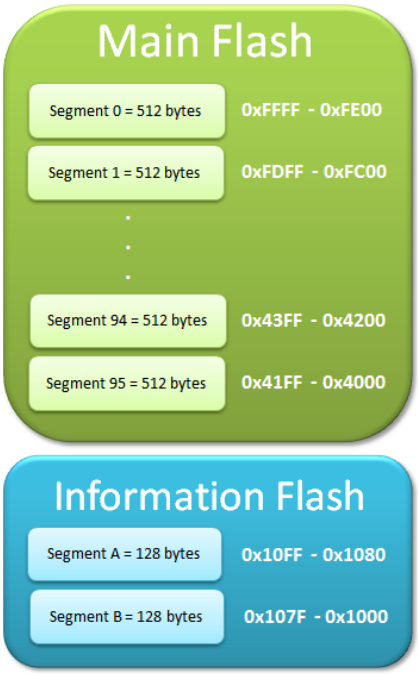


Figure 4.8: *Segmentation of the MSP430F1611 flash memory*

As shown in Figure 4.8, the flash memory of the MSP430F1611 is divided into segments. The main flash memory is 48 KB and the segments in the main flash memory are 512 bytes each. Hence the main flash memory contains 96 segments and the segments are called segment 0 till segment 95. The information flash memory consists of two segments that are 128 bytes each. The two segments are called segment A and B.

4.3.3.2 Flash Memory Operations

On Random Access Memory (RAM) two operations can be applied. These are the write and the read operation. The read operation simply reads content from the RAM and the write operation can change bit values in the RAM from a logical 0 to a logical 1 or vice versa. For flash memory, the read operation is exactly the same as that for RAM. However, the write operation for flash memory can only change bit values in the flash memory from 1 to 0. When a transition of a bit value from 0 to 1 is needed, an erase operation is needed. So for flash memory there is a total of three different operations:

- **Read operation**
  With this operation data can be read from the flash memory. Since the flash memory is memory mapped, data bytes can be read directly from the address values on which the flash memory is mapped.
- **Write operation**
  The operation for writing bytes to the flash memory changes bit values in the flash memory from a logical 1 to a logical 0. This is also known as a flash write cycle.
- **Erase operation**
  With this operation bit values in the flash memory can be changed from a logical 0 to a logical 1. The transition from a logical 0 to a logical 1 in the flash memory is known as an erase cycle. Usually an erase cycle is performed to larger chunks of flash memory. For the MSP430F1611, segments 0 to 95 may be erased in one step or each segment may be individually erased [28].

It must be noted that flash erase cycles eventually affect the health of the flash memory. A limited amount of erase cycles can be performed. For the MSP430 the amount of erase cycles that can be performed lies between 10^4 and 10^5 [1].

4.3.3.3 Flash Memory Module Design

With the design of the flash memory software module, care must be taken with the boundaries of the flash memory. The read operation activity flow is shown in Figure 4.9.

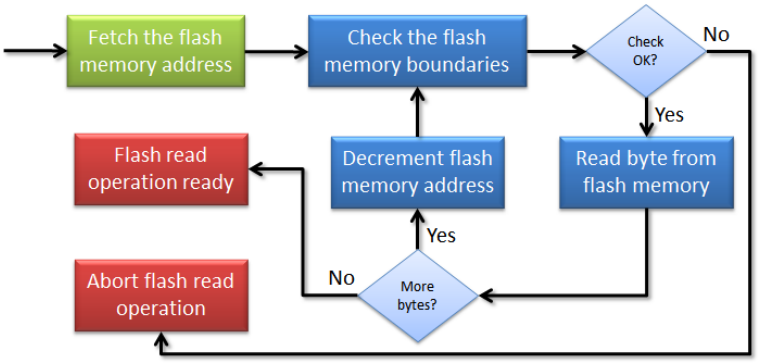


Figure 4.9: *Activity flow for reading data from flash memory*

A flash memory read operation starts with the fetching of the flash memory address. Then the flash memory boundaries must be checked before data can be read from the flash memory. When the flash memory address does not lie within the flash memory boundaries, the read operation must be aborted. When more bytes needs to be read, the flash memory address must be decremented and the boundaries must be checked again before the byte is read from the flash memory. When all bytes are read, the read operation is finished.

The flash memory write operation is analogue to the flash memory read operation. The only difference is that a byte must be written to the flash memory. The activity flow for the write operation is graphically represented in a diagram in Figure 4.10.

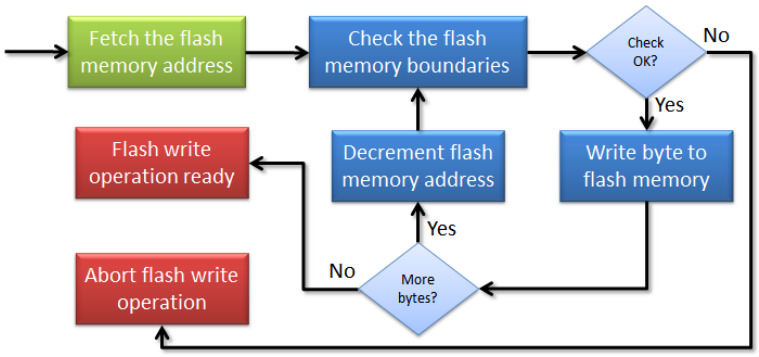


Figure 4.10: *Activity flow for writing data to the flash memory*

The erase flash memory operation must consist of two options. These options are to erase an individual segment or to erase the complete main flash memory. When an individual segment is going to be erased, the segment that corresponds to the given flash memory address will be erased. When the complete main flash memory is going to be erased, any given flash memory address within the main flash memory boundaries suffices. This description of the erase operation results in the activity flow diagram shown in Figure 4.11.

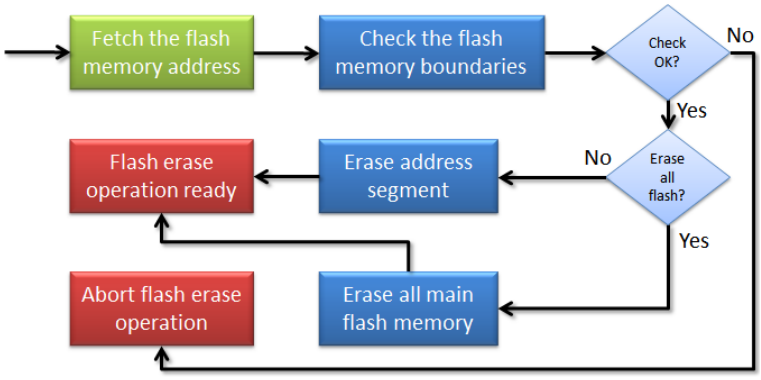


Figure 4.11: *Activity flow for erasing (part of) the flash memory*

4.3.4 Analog to Digital Converter

The analog to digital converter (ADC) of the MSP430F1611 can for example be used to read out sensor values. The ADC converts an analog signal into a digital value. For the OBC, the ADC must be used to read out the temperature of the OBC. This section shows the design of the ADC with the functionalities that are needed to read out the OBC temperature. First it is described why calibration of the temperature sensor is needed. Finally, the functionalities that are needed to read out the OBC temperature are listed and a graphical representation of the activity flow is given. The temperature sensor transfer function is not considered here, since conversion from the digitized calibrated sensor value to the temperature in degrees Celsius is part of the ground segment data processing. Still, the temperature transfer function is mentioned because it is needed to determine the reference voltage needed for the conversion.

4.3.4.1 Temperature Sensor Transfer Function

The temperature sensor transfer function gives the relationship between the measured sensor output voltage and the temperature in Celsius. The temperature sensor transfer function for the on-chip temperature sensor of the MSP430F1611 is defined as follows where the measured output voltage of the sensor is a function of the ambient temperature:

$$V_{out} = 0.00355 \cdot T + 0.986 \tag{4.5}$$

By rearranging Equation 4.5, the ambient temperature T becomes a function of the measured output voltage V_{out} of the sensor:

$$T = \frac{V_{out} - 0.986}{0.00355} \tag{4.6}$$

The relationship between V_{out} and the temperature T is linear, as can be seen in Figure 4.12 [28].

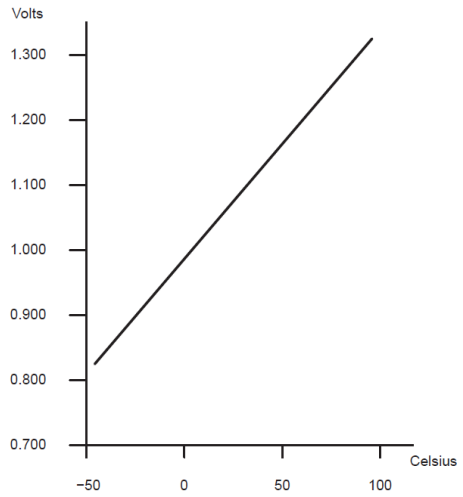


Figure 4.12: *Temperature sensor transfer function for the on-chip temperature sensor*

4.3.4.2 Temperature Sensor Calibration

In order to get accurate temperature readings, the on-chip temperature sensor of the MSP430F1611 must be calibrated. The MSP430F1611 on-chip temperature sensor can have an offset as much as $\pm 20\,^{\circ}\mathrm{C}$ [1]. By substituting a known value for T in Equation 4.5 and checking the measured voltage V_{out}, one is able to compute the desired offset value for the particular MSP430F1611 chip that is being calibrated.

In order to reliably compute the offset value, an already calibrated temperature meter must be used as a reference. It is important that this reference temperature meter has already been calibrated as accurate as possible and does not deviate to much from the real ambient temperature that is being measured. During the calibration process it is important that the reference temperature meter is held close to the MSP430F1611 microcontroller chip such that they sense more or less the same ambient temperature.

The computation of the temperature sensor offset is done in three steps:

- **First step:** Measure the output voltage of the temperature sensor at a known ambient temperature T_1 using Equation 4.5.
- **Second step:** Compute the temperature T_2 using Equation 4.6 and the temperature sensor output voltage that was measured in the previous step.
- **Third step:** Subtract T_2 from T_1 and take the absolute value. The result is the temperature sensor offset in degrees Celsius.

So the computed offset temperature of the temperature sensor becomes $T_{offset} = |T_2 - T_1|$ and is expressed in degrees Celsius.

4.3.4.3 ADC Module Design

On the MSP430F1611 there is a 10-bit ADC and a 12-bit ADC. To read out the OBC temperature, the 10-bit ADC will be used. The reason for this is that the 10-bit ADC is faster than the 12-bit ADC and it consumes less power. Besides the advantages in performance, the 10-bit ADC meets the requirements for the resolution of the OBC temperature which is defined to be 8 bits. The activity flow diagram for reading out the temperature of the OBC using the ADC is shown in Figure 4.13.

In the activity flow diagram shown in the figure, the needed functionalities can be seen. The use of the 10-bit ADC is obvious, since this functionality has already been explained above. Furthermore, the ADC has multiple channels that can be used for conversion and a selectable voltage reference. The ADC channel must be chosen such that the ADC does conversions on the channel on which the temperature sensor is connected. The reference voltage must be higher than the maximum voltage that is given by the temperature sensor. This maximum output voltages becomes clear in Figure 4.12 that gives the temperature sensor transfer function. From this figure it is clear that the maximum output voltage of the temperature sensor will not exceed 1.4V, so the reference voltage much be chosen such that it is equal to 1.4V or higher. The ADC module must also contain procedures that enable the ADC interrupt mechanism and start the

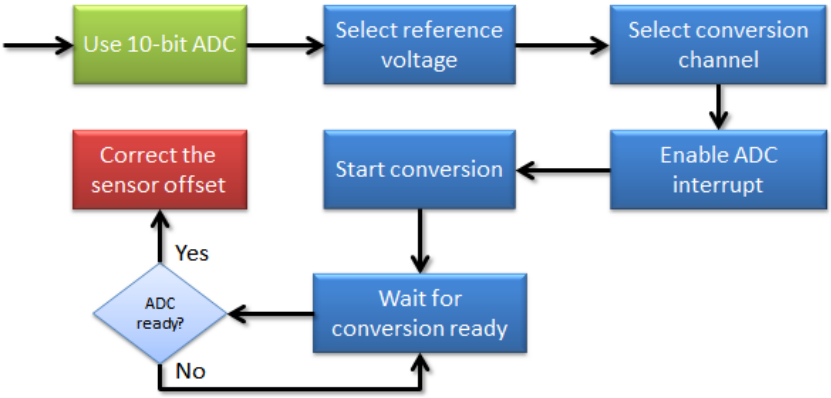


Figure 4.13: *Activity flow for the ADC to read out temperature*

conversion. After the conversion has been started, the conversion result will usually be available within a few microseconds. Finally, the conversion result must be corrected by the digital sensor offset value that has been calculated during the calibration process of the temperature sensor.

4.3.5 I²C Controller

The design of the I²C master device and I²C slave device service layer software is described in this section. First the design of the master is given. This is followed by the design of the slave.

4.3.5.1 I²C Master Module Design

The I²C master service layer software module must consist of a couple of elementary functionalities that are needed for reliable I²C communication. In this section the functionalities needed for the master device module are discussed. The master device module must consists the following functionalities:

- Initialize the I²C peripheral in master mode
- Read data from the bus and handle time-outs
- Write data on the bus and handle time-outs
- Report I²C bus health status

During the initialization procedure of the master device, the I²C hardware peripheral must be configured such that it operates in master mode and generates clock pulses at a frequency of 100 kHz on the SCL line during a data transfer. For the latter the I²C hardware peripheral must select a clock source. The activity flow for initializing the I²C peripheral in master mode is shown in Figure 4.14

The procedure for reading data from the bus must first initialize and start a timer that generates an interrupt after 30 ms. Once an interrupt occurs, a flag that is being polled

Figure 4.14: *Activity flow for initializing the I^2C peripheral in master mode*

by the read procedure must be set by the ISR of the timer. This design ensures that the read procedure does not get stuck in the polling loop forever. After the initialization of the time-out timer, the read procedure must wait until the bus is free to use. If the bus does not become free within 30 ms, the read operation must be aborted and the I²C bus health status must be reported as *'bus lines held low'*. In the case of a free bus the read procedure must continue with putting the I²C peripheral in master-receiver mode. The next step is to send out a start bit, followed by the desired 7-bit slave device address from which the master wants to read data and a bit with the logical value 1 that indicates that the I²C operation is an I²C read operation.

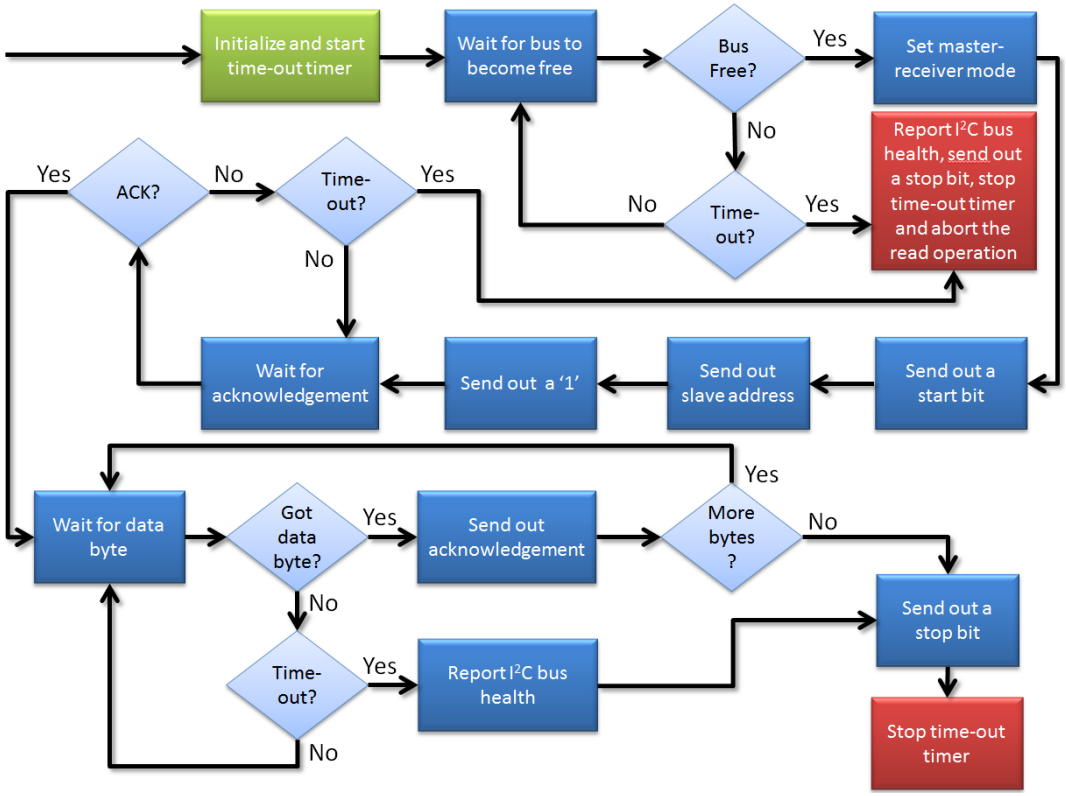


Figure 4.15: *Activity flow for reading data from the I^2C bus in master-receiver mode*

At this point the read procedure must wait for an acknowledgement from the requested slave device. If the acknowledgement from the slave device is not received within 30 ms, the master device must abort the read operation and it must report the I²C bus health status *'no acknowledgement received'*. In the case the master device did receive the acknowledgement from the requested slave device, the read operation can continue

with reading data bytes from the bus. For each data byte the master device must wait until the slave device is ready to put the data byte on the bus. The master device must acknowledge the reception of each data byte, except for the last data byte. When a time-out occurs during the reading of the data bytes, the master device must send out a stop bit, report the I^{2}C bus health status *'time-out'* and abort the read operation. In the case that no time-out occured the master must send out a stop bit in order to free the bus. At this point the read operation is finished. Finally, the master must stop the time-out timer. The activity flow for reading data from the I^{2}C bus in master-receiver mode is shown in Figure 4.15.

The wait actions shown in the activity flow diagram continuously poll a flag that will be set when the 30 ms time-out period has been elapsed. During intialization of the time-out timer, the flag that is polled must be cleared.

The procedure for writing data on the bus is analogue to the procedure for reading data from the bus. The small differences are that the I^{2}C peripheral must be put in master-transmitter mode and that the master must send out a bit with a logical value of 0 after writing the 7-bit slave address on the bus to indicate that a write operation is initiated. Besides that it must of course write data bytes on the bus and wait for acknowledgements from the slave device, instead of reading data bytes from the bus and produce acknowledgements. The activity flow for writing data on the I^{2}C bus in master-transmitter mode is shown in Figure 4.16.

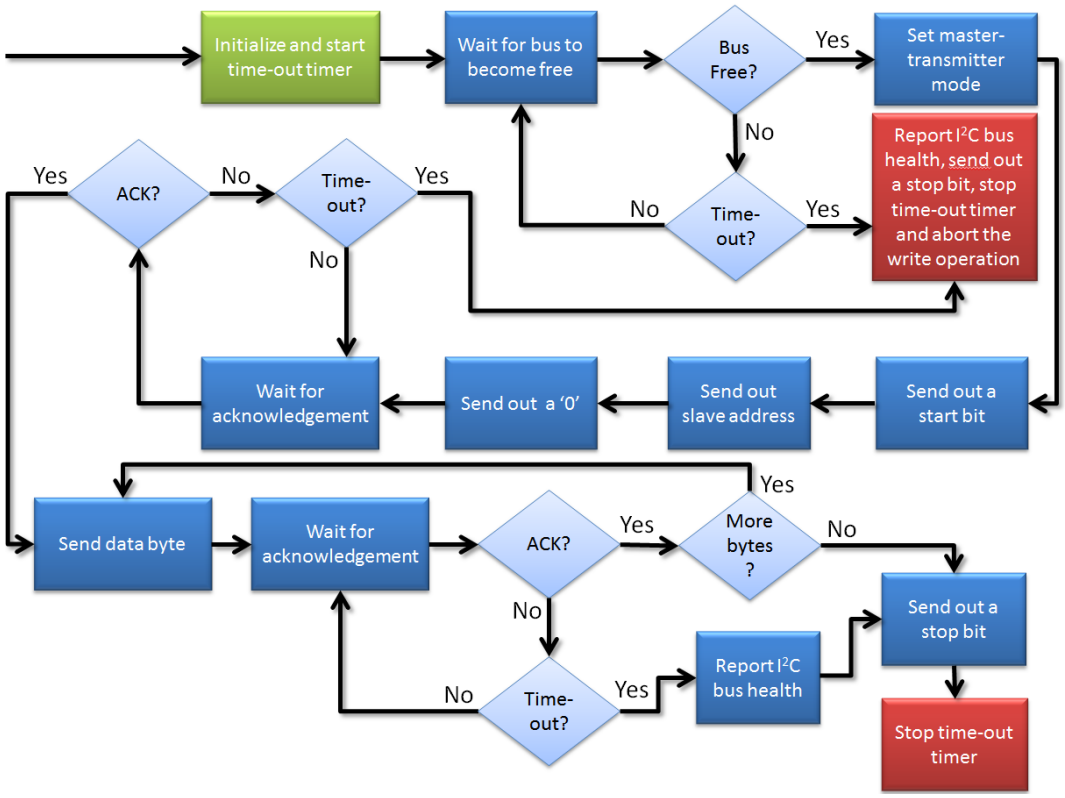


Figure 4.16: *Activity flow for writing data on the I^{2}C bus in master-transmitter mode*

4.3.5.2 I²C Slave Module Design

The I²C slave service layer software module must consist, just like the I²C master service layer software module, of a couple of elementary functionalities that are needed for I²C communication with the master device. Since the slave device is contacted by the master device, the use of interrupts suits best for the read and write operations. With an interrupt-driven approach, no time-out handling is needed. In this section the functionalities needed for the slave device module are discussed. The slave device module must consist of the following functionalities:

- Initialize the I²C peripheral in slave mode
- Read data from the bus using interrupts
- Write data on the bus using interrupts

Initialization of the I²C peripheral for a slave device consists of configuring the I²C hardware peripheral in slave mode, setting the slave address of the slave device, selecting the clock source for the I²C hardware peripheral and enabling the I²C peripheral interrupts. This is translated in an activity flow diagram that is shown in Figure 4.17

Figure 4.17: *Activity flow for the initialization procedure for an I²C slave device*

Reading data from the bus in slave mode starts when a start interrupt is generated by the I²C hardware peripheral. This interrupt is generated when the I²C hardware peripheral detects a start condition followed by the slave address of the slave device and the R/$\overline{W}$ bit. The detection of events on the I²C bus is completely handled by the I²C hardware. The only thing the software has to do is execute tasks that are needed such that data bytes can be received without any problems. A typical example of such a task is clearing the receive buffer that is used for data byte reception.

After this procedure the I²C hardware peripheral will receive one or more data bytes. When a single data byte is received, the I²C hardware peripheral will generate an interrupt. The software must handle this interrupt and store the received data byte in the right place in the receive buffer. Each received data byte must be acknowledged by the slave and reading data from the bus stops when all bytes have been received. The ending of the data transfer is indicated by a stop condition that is generated by the master device. When the I²C hardware detects a stop condition on the I²C bus it will generate a stop interrupt that can be used by the software to execute tasks that handle the ending of a data transfer.

Writing data on the I²C bus in slave mode is analogue to reading data from the I²C bus in slave mode. The difference here is that the data bytes must be read from a transmit buffer and that the data bytes must be written on the bus. Also the slave device has only to acknowledge on the start interrupt. When this interrupt occured the slave

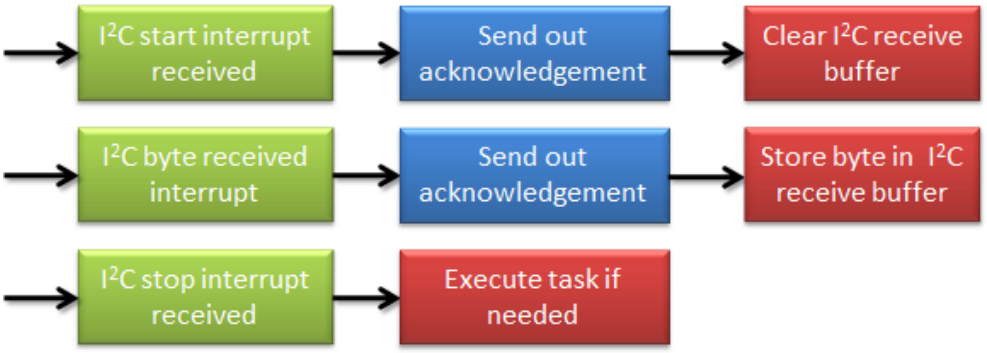


Figure 4.18: *Activity flow for handling I^2C interrupts for reading data in slave mode*

device can prepare the data content in the I^2C transmit buffer. The acknowledgements of the data bytes are done by the master device since the master device is receiving the bytes. A data byte can be transmitted when an acknowledgement of the master is received and the transmitter is ready. When this occured, the I^2C hardware will generate an interrupt that indicates that a byte can be safely written on the bus by the slave device. Again, on a stop condition interrupt, the software may execute tasks that handle the ending of a data transfer. The activity flow diagram for writing data on the bus in slave mode is shown in Figure 4.19.

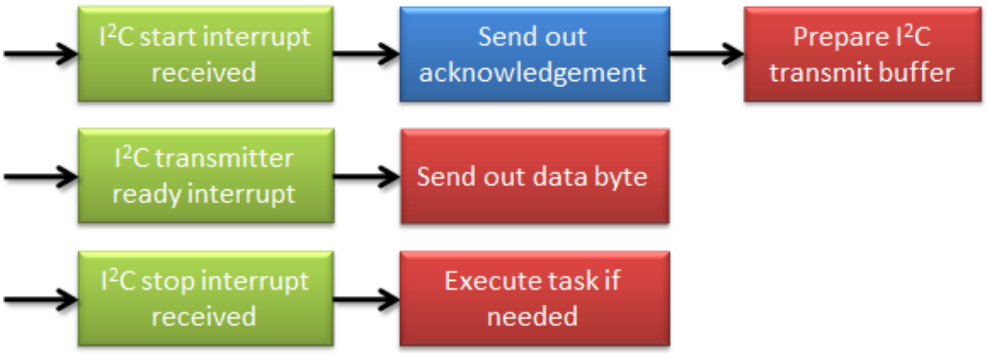


Figure 4.19: *Activity flow for handling I^2C interrupts for writing data in slave mode*

4.3.6 Watchdog Timer

The watchdog timer is an important peripheral. It is in essence a timer, but it can be configured to operate in one of the two following modes; the watchdog mode and the interval timer mode. A requirement for the watchdog timer that must be used in the OBC software is that it must be able to be configured for an expiration time in the order of seconds. First, the differences between the modes are discussed in Section 4.3.6.1. This is followed by a description about the clock source design options (Section 4.3.6.2) and the watchdog timer module design (Section 4.3.6.3).

4.3.6.1 Watchdog Modes

As already briefly mentioned, the watchdog timer can be configured for either the watchdog mode or the interval timer mode. The difference between the two modes is the action that will be performed when the watchdog timer expires (i.e. when it reached its configured interval). The differences between the two modes are the following:

- **Watchdog mode:** in this mode a Power-Up Clear (PUC) will be generated when the watchdog timer expires.
- **Interval timer mode:** in this mode an interrupt will be generated when the watchdog timer expires, which works the same as an ordinary timer in *up mode*.

For the OBC software the interval timer mode is not of interest, since the ordinary programmable interval timers are more appropriate to generate the desired interval interrupts or delays. The watchdog mode will be used in order to let the microcontroller reset itself (generate a PUC) when the watchdog timer counter value is not cleared within the specified expiration time.

4.3.6.2 Watchdog Clock Signal Design Options

For the watchdog timer to function, a clock signal must be selected. This clock signal can either be the sub-main clock (SMCLK) or the auxiliary clock (ACLK). In Section 4.3.1 it has already been explained that the low-frequency 32768 Hz crystal must be used for the watchdog timer. Furthermore it has been explained that the low-frequency 32768 Hz drives the ACLK signal and that the ACLK signal is divided by 8 such that the signal becomes a 4096 Hz signal. This is needed in order to get a watchdog timer expiration time of 8 seconds, as will be shown later on in this section. The 8 MHz crystal that drives the SMCLK signal is not suitable for the watchdog timer in the OBC software. This results in a too low expiration time in the order of a few milliseconds while an expiration time in the order of seconds is needed [19].

An important limitation of the watchdog peripheral is that the watchdog expiration interval can only be configured using 4 different clock signal dividers:

- Clock signal divider of 32768
- Clock signal divider of 8192
- Clock signal divider of 512
- Clock signal divider of 64

There is no timer threshold register that can be configured such that the watchdog timer expires in any desired amount of clock ticks. The best thing one can do is selecting an appropriate clock signal and configure the clock source divider properly such that the expiration time interval is as close as possible to the desired expiration time interval. This

truly limits the flexibility in designing applications that need a wide range of watchdog expiration time intervals.

In order to get an expiration interval in the order of seconds, the ACLK clock signal must be used as clock input signal for the watchdog peripheral. Using the ACLK, various clock signals can be used as clock signal input depending on the clock divider of the ACLK and the clock divider of the watchdog timer peripheral. All the possible options are listed in Table 4.1.

f_{ACLK}	DIV_{ACLK}	f_{input}	$DIV_{watchdog}$	Expiration time
32768 Hz	1	32768 Hz	32768	1 Hz = 1 s
32768 Hz	1	32768 Hz	8192	4 Hz = 250 ms
32768 Hz	1	32768 Hz	512	64 Hz = 15.625 ms
32768 Hz	1	32768 Hz	64	512 Hz = 1.95 ms
32768 Hz	2	16384 Hz	32768	0.5 Hz = 2 s
32768 Hz	2	16384 Hz	8192	2 Hz = 0.5 s
32768 Hz	2	16384 Hz	512	32 Hz = 31.25 ms
32768 Hz	2	16384 Hz	64	256 Hz = 3.9 ms
32768 Hz	4	8192 Hz	32768	0.25 Hz = 4 s
32768 Hz	4	8192 Hz	8192	1 Hz = 1 s
32768 Hz	4	8192 Hz	512	16 Hz = 62.5 ms
32768 Hz	4	8192 Hz	64	128 Hz = 7.8125 ms
32768 Hz	8	4096 Hz	32768	0.125 Hz = 8 s
32768 Hz	8	4096 Hz	8192	0.5 Hz = 2 s
32768 Hz	8	4096 Hz	512	8 Hz = 125 ms
32768 Hz	8	4096 Hz	64	64 Hz = 15.625 ms

Table 4.1: Selectable watchdog expiration times using the 32768 Hz clock source

In this table f_{ACLK} is the frequency of the crystal that drives the ACLK signal, DIV_{ACLK} the clock divider used for the ACLK clock signal, f_{input} the input frequency for the watchdog peripheral which equals f_{ACLK} divided by DIV_{ACLK} and $DIV_{watchdog}$ is the clock divider used for the watchdog peripheral input frequency. The expiration time is calculated by dividing f_{input} by $DIV_{watchdog}$.

4.3.6.3 Watchdog Module Design

The functionalities for the watchdog timer include configuration of the watchdog timer and clearing the watchdog timer counter in order to prevent a PUC. Clearing the watchdog timer is evident. Configuration of the watchdog timer consists of two steps:

- Select the input clock signal
- Select the appropriate clock signal divider

The order in which these two steps are executed to configure the watchdog timer peripheral is not important.

4.4 OBC Application Layer Software Design

In this section the design of the application layer software for the Delfi-n3Xt Nanosatellite is described. The application layer software makes use of the service layer software that is described in the previous section of this chapter. The satellite can be in 6 different states, called operational modes, or just modes for short. These are the boot mode, delay mode, deployment mode, main mode, test mode and I^2C bus recovery mode. The OBC can only be in one of these 6 mode at a time.

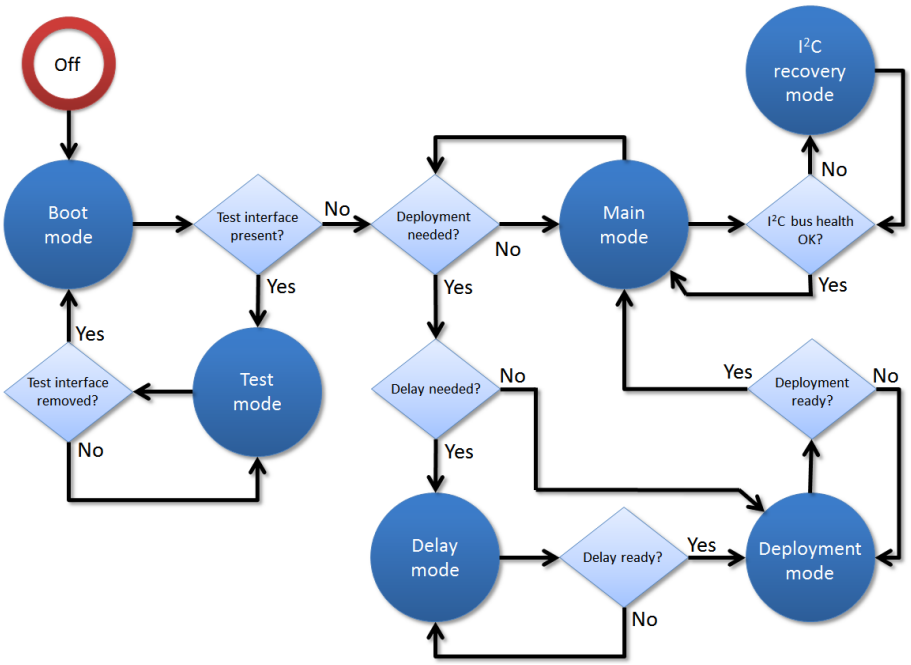


Figure 4.20: *Activity flow for the transitions between operational modes*

In Figure 4.20, the possible transitions between operational modes are shown. Once the OBC is powered, the primary OBC microcontroller will start running which will result in the execution of the boot sequence. During the boot sequence the satellite its operational mode is the boot mode. The boot mode is described in detail in Section 4.4.1.

After completion of the boot mode, the satellite will become in the test mode, delay mode, deployment mode or main mode. The test mode will be entered if the test interface is present on the I^2C bus. A transition from the test mode back to the boot mode may take place when the test interface is removed from the I^2C bus. The test mode is not yet designed and implemented and therefore it is not discussed in this thesis.

When the test interface is not present or if it is removed, the delay mode, deployment mode or main mode will be entered. The delay mode is described in Section 4.4.2 and will be entered when deployment is needed after a certain amount of time (e.g. when the satellite is launched and deployed for the first time). The deployment mode is described in Section 4.4.3 and will be entered if no deployment has been attempted before or if the OBC is commanded (by a telecommand) to make a transition from the main mode to the deployment mode.

The main mode will be entered immediately after the boot mode when there is no test interface present on the I^2C bus and deployment is not needed. The main mode will also be entered when the deployment mode is ready. Furthermore, the main mode will switch to I^2C bus recovery mode if the I^2C bus got stuck. A detailed description of the main mode is given in Section 4.4.4.

When the I^2C bus got stuck, the I^2C bus recovery mode will be entered. This mode will continuously check the status of the I^2C bus and will make a transition back to the main mode when the I^2C bus is recovered and considered as healthy. A detailed design description about this mode is given in Section 4.4.5.

4.4.1 Boot Mode

As already mentioned earlier, every OBC boot starts in the boot mode. The boot mode initializes all the variables and peripherals that are needed for the OBC to function properly. Besides that it updates the boot counter, configures the other subsystems of the satellite and checks whether or not a test interface is present on the I^2C bus. The updating of the boot counter is further described in Section 4.4.1.1 and the configuration of the subsystems at boot time in Section 4.4.1.2. The activity flow of the boot sequence execution is shown in Figure 4.21.

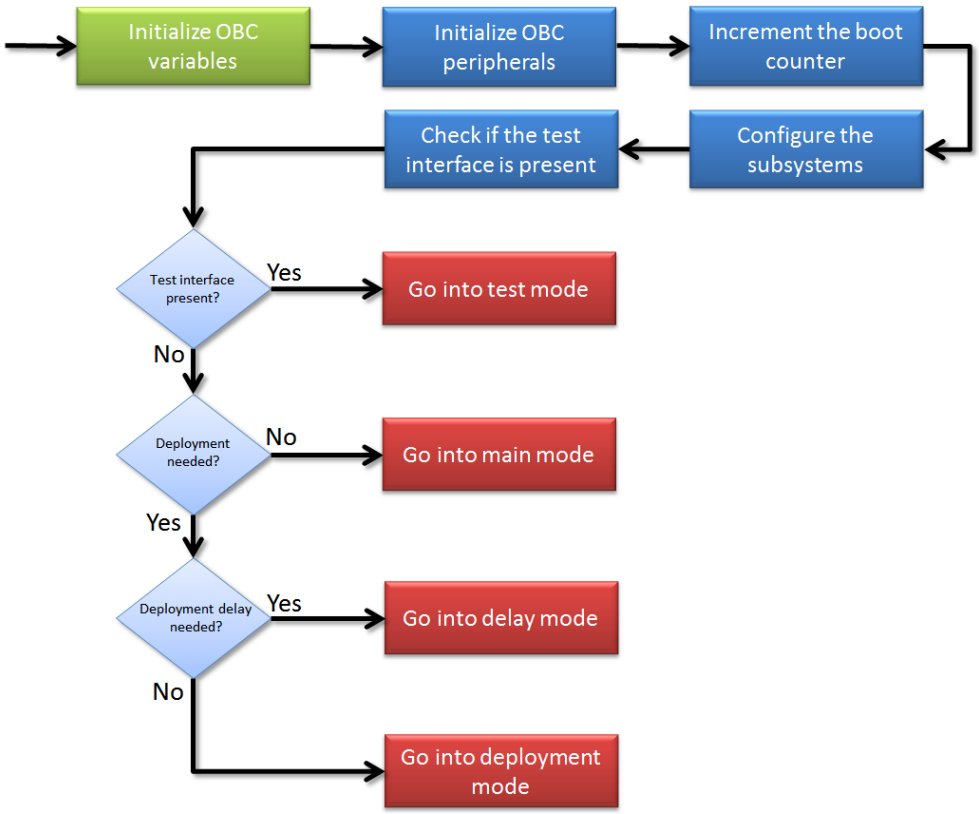


Figure 4.21: *Activity flow for the execution of the boot mode*

Initialization of the OBC peripherals yields initialization and configuration of the clock source module (Section 4.3.1), programmable interval timers (Section 4.3.2), flash memory controller (Section 4.3.3), analog to digital converter (Section 4.3.4), I^2C controller (Section 4.3.5) and watchdog timer (Section 4.3.6). The check for the presence of the test interface can easily be realised by contacting the test interface I^2C address over the I^2C bus. If the OBC receives an acknowledgement on a request to the test interface device it means that the test interface is present, in the other case the test interface can be considered as not present.

After this execution, decisions will be made about the next mode to which the OBC must switch. The delay mode will be entered only if deployment is needed and if the delay for deployment is needed. If deployment is needed but the deployment delay is not needed, the OBC will switch to deployment mode immediately. Deployment is needed when one or more deployment devices have not been attempted to deploy yet. Whether or not deployment has already been attempted for a deployment device, is stored as an 16-bit vector in the flash memory. This vector contains one bit for each resistor that can be burned in order to deploy an antenna or solar panel, and it is updated accordingly. The sequence of deploying the deployment devices is further described in Section 4.4.3. If deployment is not needed the OBC will switch immediately to the main mode.

4.4.1.1 Boot Counter Update

The boot counter is implemented in such a way that it saves flash erase cycles when the new incremented value must be written to flash memory. As already mentioned in Section 4.3.3, the flash memory of the MSP430F1611 can be erased somewhere between 10^4 and 10^5 times before it becomes unpredictable and unstable. The boot counter is a 2-byte value so it can contain $2^{16} = 65536$ different values and it overflows back to 0 when the boot counter has reached the value 65535 and is incremented by 1. In order to save flash erase cycles, the 2-byte boot counter is encoded in a 33-byte boot counter such that a flash erase cycle only needs to be performed after 256 boots. The 2-byte boot counter is encoded into a 33-byte boot counter as follows:

- The first 32 bytes (256 bits) are used are used for counting up to 256 boots.
- The last byte (8 bits) is used to hold the amount of multiples that the OBC booted 256 times

With this approach, the 33-byte encoded boot counter can hold up to $256 \times 256 = 65536$ different values, which is the same as the regular 2-byte boot counter. Initially all the 264 bits are set to 1, which is the default state of flash memory content after erasing. This represents a boot counter value of 0. So the 33-byte flash memory content of the encoded boot counter will initially look as follows (Figure 4.22).

When the boot counter is incremented by 1, the first 1 that is found in the first 256 bits of the encoded boot counter will be set to 0. For a bit transition from 1 to 0 there is no flash erase cycle needed, hence the first 256 boots there is no flash erase cycle needed when the encoded boot counter is updated in the flash memory. After the first boot the encoded boot counter content will look as follows (Figure 4.23).

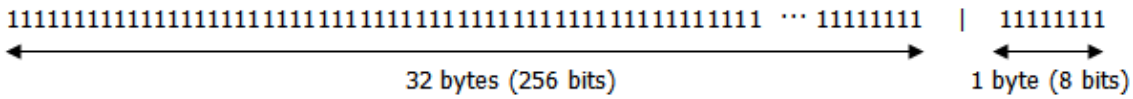


Figure 4.22: *The content of the encoded boot counter that represents a value of 0*

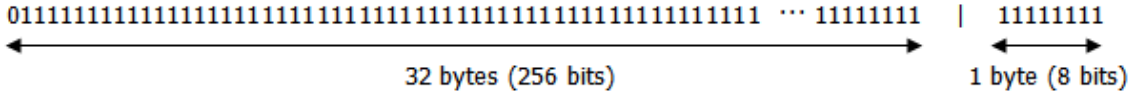


Figure 4.23: *The content of the encoded boot counter that represents a value of 1*

This can be repeated 256 times such that after 256 times all the bits of the first 256 bits are set to 0. At this point the last byte (byte 33) must be incremented by 1 as well since the boot counter value is now a multiple of 256. However, the last byte is complemented since its inital value consists of all ones. In order to increment the represented value of this byte by one, we can simple decrement the byte value by 1 since it is complemented. The change of this last byte needs a flash erase cycle in order to store it properly in flash memory. Because of this reason, this is the moment where the first 256 bits should all be reset to 1 (which is automatically done by the flash erase cycle). When this is done, the encoded boot counter that represents the value of 256 will look as shown in Figure 4.24.

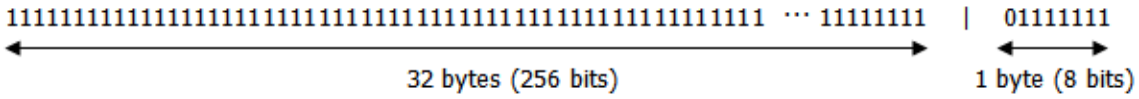


Figure 4.24: *The content of the encoded boot counter that represents a value of 256*

At this point all the first 256 bits are set to 1 again, and the process of scanning the first 256 bits until a 1 is detected can start over again. After another 256 boots the first 256 bits are set to 0 again, and the last byte (byte 33) will be decremented by 1 again after which a flash erase cycle is necessary. This whole process can be repeated in total 256 times (65536 boots) until an overflow occurs which brings the value of the boot counter back to 0.

4.4.1.2 Configuring Subsystems

The procedure that configures the subsystems and their DSSB power controllers at boot time ensures that the EPS, CDHS, PTRX, STX, MechS and SDM subsystems are turned on (i.e. they are powered). All the other subsystems should remain off and thus must be unpowered.

The EPS and CDHS subsystems are 'standard on'. This means that when power becomes available on the satellite bus, the CDHS (including the OBC and all the DSSB microcontrollers) and EPS subsystems are automatically powered. The other subsystems are 'standard off', so they must be turned on by the OBC explicitly. Turning on or off

a subsystems is done by the DSSB power controllers belonging to the subsystems. The DSSB power controllers are configured by the OBC over the I^2C bus.

Once the subsystems are turned on by the OBC after commanding the appropriate DSSB power controllers, the subsystems can be configured to function in a desired sub-mode. This configuration is also performed over the I^2C bus, but this time the OBC directly commands the microcontroller of the subsystem and not the DSSB power controller of the subsystem. So for the EPS and CDHS nothing has to be done, since they are automatically powered and functioning correctly. The next action is to put the PTRX in RX only mode such that it will not send any data to earth at boot time. Configuration of the PTRX should be done using the following sequence:

- Turn on the PTRX subsystem by commanding the DSSB power controller belonging to the PTRX
- Command the PTRX subsystem to switch to RX only mode (meaning the transmitter is turned off)

However, the PTRX is not capable of shutting down its transmitter [30] such that the mode becomes RX only. Because of this reason, no commanding can be performed by the OBC to configure the PTRX for RX only mode. In order to let the PTRX behave as a receiver only, the OBC must hold a variable that holds the current sub-mode of the PTRX. Fortunately this was already taken into account in the CDHS top level design [9], so the OBC can just stop sending data frames to the PTRX when the RX only sub-mode for the PTRX is defined in the sub-modes vector.

For the MechS and SDM, nothing more has to be done. They just need to be turned on and they do not need to be put into a sub-mode. The STX however must be configured such that it is in the TX off mode. This means that the STX subsystem is powered, but it is not allowed to transmit any data to the earth. This can be solved in the same way as for the PTRX. However, the STX does have the ability to turn off its transmitter. This can be performed by writing a single byte to the STX I^2C address.

4.4.2 Delay Mode

When the satellite is just launched and being ejected from the POD, the OBC will start for the very first time in space. For this very first boot, and after the execution of the usual boot mode as explained in Section 4.4.1, the satellite will go into the delay mode. The delay mode is needed in order to make sure that the deployment of the solar panels and antennas does not happen too early. Early deployment while the satellite is still in vicinity of the POD may cause damage to one or more of the solar panels and/or antennas [20]. The delay mode must ensure that the solar panels and antennas can be deployed safely.

While the idea is simple, the delay mode can introduce a failure case that occurs when the OBC continuously reboots itself within the amount of time the delay threshold value is set to. If this is the case, the OBC will never reach the deployment mode. In order to take this failure case into account, a delay counter will be held in flash memory. This delay counter must be updated in flash memory every minute until the delay threshold

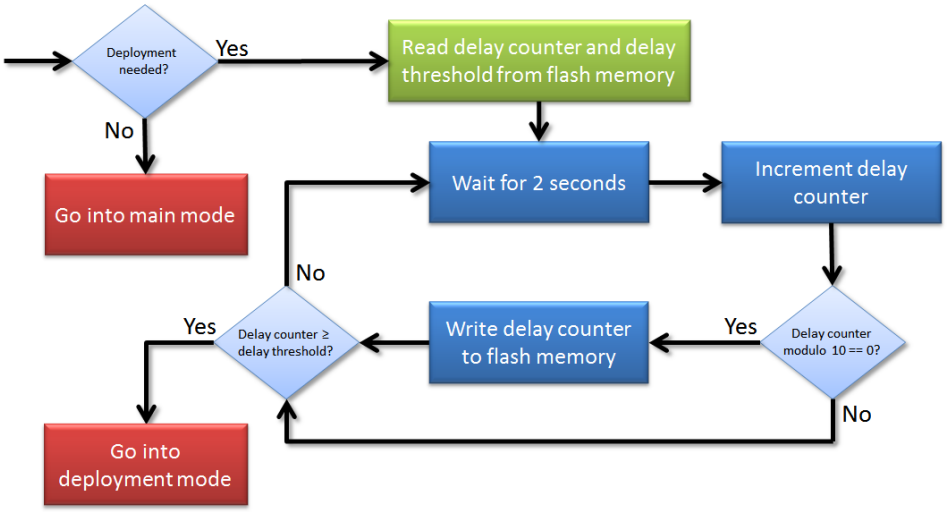


Figure 4.25: *Activity flow for the delay that may be needed before deployment*

value is reached. The sequence for the delay mode is given in the form of an activity flow in Figure 4.25.

4.4.3 Deployment Mode

After the boot mode or the delay mode (depending on whether or not a delay between boot mode and deployment mode is necessary), the OBC may enter the deployment mode if deployment of one or more of the solar panels and/or antennas is needed. The deployment mode consists of a sequence that deploys the deployment devices one by one. This sequence is further described in Section 4.4.3.1. When the deployment mode is entered, the deployment sequence can be started immediately since it is determined in the previous mode that deployment is needed.

4.4.3.1 Deployment Sequence

The deployment sequence consists of two separate sequences: the primary deployment sequence and the secondary deployment sequence. The primary deployment sequence is responsible for commanding the MechS subsystem to deploy the 4 solar panels and 4 antennas by burning the primary resistors. First, the 4 solar panels will be deployed using the burning of the primary resistors. The activity flow for this is shown in Figure 4.26.

When a deployment device must be deployed, and the command to deploy the device is sent to the MechS subsystem, the OBC must wait 16 seconds until it goes on with the next deployment device. This is because burning a wire using the resistors may take up to 15 seconds worst case, and it is not allowed to burn more than one wire at the same time because of the high power consumption that is needed to burn a wire. After deploying the 4 solar panels using the primary resistors, the 4 antennas can be deployed using their primary resistors using the same mechanism as shown in the activity flow for the solar panels (Figure 4.26).

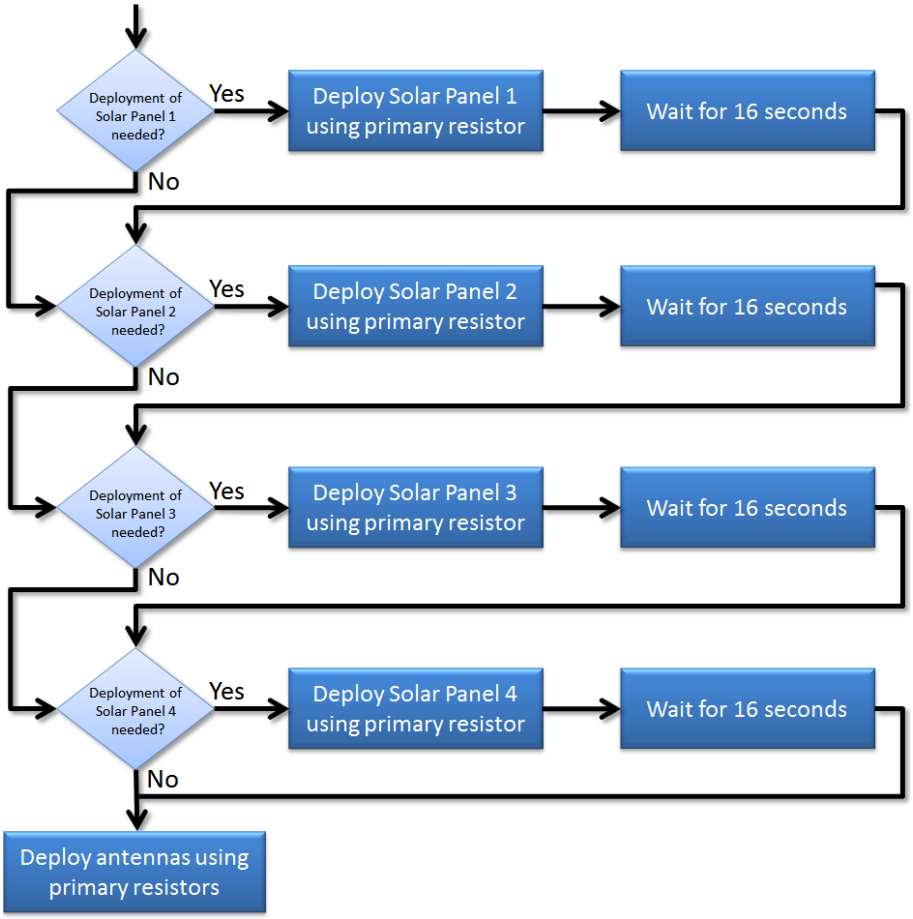


Figure 4.26: *Activity flow for deploying the solar panels using the primary resistors*

After deploying the solar panels and antennas using the primary resistors, the solar panels and antennas should be deployed again by burning the secondary resistors. This is done in order to be sure that they are deployed if one or more of the primary resistors fail to burn the wires. This redundancy makes sure that there are no Single-point-of Failures (SPoFs) in the deployment mechanism.

The sequence that burns the secondary resistors to deploy devices that need to be deployed is a bit different compared to the sequence that burns the primary resistors. The activity flow for deploying the solar panels using the secondary resistors is shown in Figure 4.27. After burning the wire using the secondary resistor, the 'deployment attempted' vector in flash memory must be updated accordingly. This vector is an 8-bit vector because there are in total 8 deployment devices (4 solar panels and 4 antennas). Each bit in the vector corresponds to a deployment device. The bit that corresponds to a deployment device must be set to '1' immediately after the secondary resistor is used to burn the wire of that deployment device. This same vector is used to determine whether or not deployment of a solar panel or antenna is needed. When the corresponding bit of a deployment device is set to '1', deployment is not needed, otherwise deployment

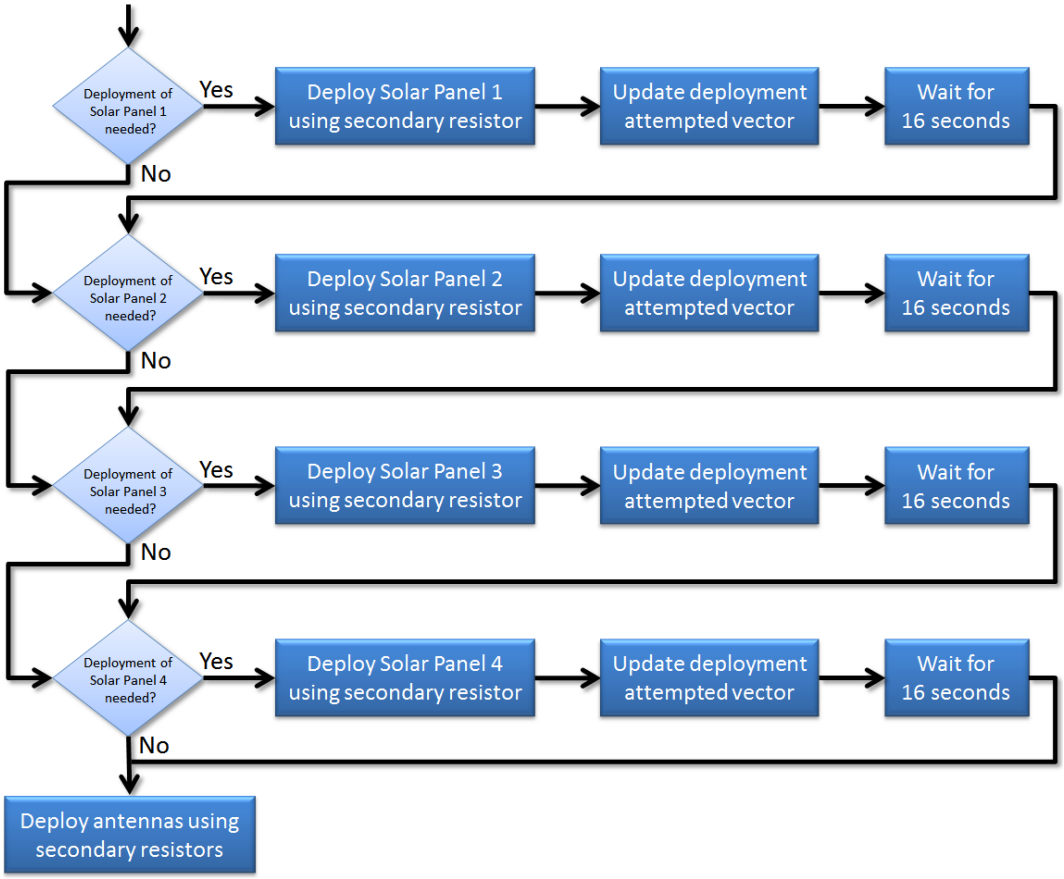


Figure 4.27: *Activity flow for deploying the solar panels using the secondary resistors*

is needed. The vector can be modified by a telecommand, this is further described in Section 4.4.4.4.

The very last part of the deployment mode is shown in an activity flow in Figure 4.28. It is similar to the deployment of the solar panels shown in the activity flow in Figure 4.27. However, in this very last part of the deployment mode, the antennas are being deploying using the secondary resistors (if needed). After this the deployment mode is ready and the OBC must switch to the main mode.

4.4.4 Main Mode

Most of the time, the satellite will be in the main mode. This mode consists of the so called 'main loop', which main responsibilities are acquiring data from other subsystems and the execution of telecommands that are uploaded from a ground station at Earth to the satellite. Besides that it checks the state of all the subsystems present in the satellite. A more detailed description about the main loop is given in Section 4.4.4.1. The section about the main loop is followed by a section about the production of OBC telemetry data (Section 4.4.4.2), data acquisition (Section 4.4.4.3) and the execution of received telecommands (Section 4.4.4.4) from Earth.

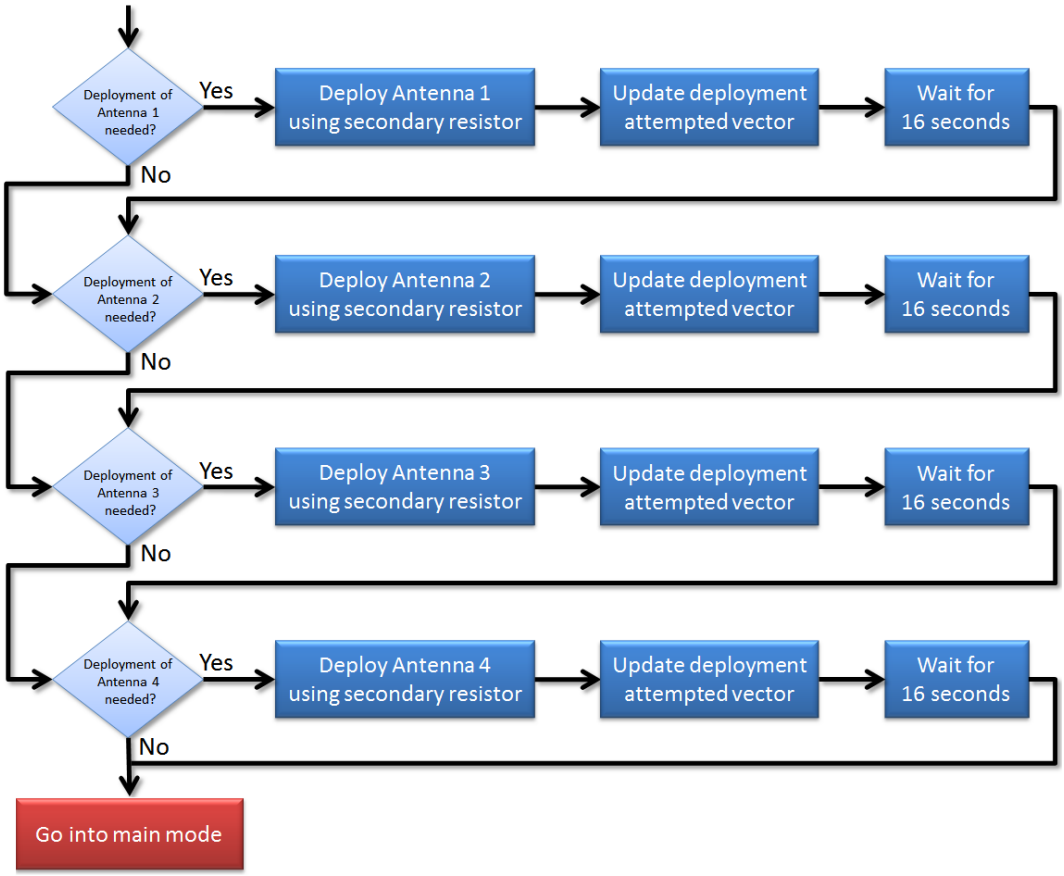


Figure 4.28: *The last part of the deployment mode*

4.4.4.1 Main Loop

The main loop is a procedure that is being executed periodically at a frequency of 0.5Hz (i.e. it is being executed once every two seconds). The main loop consists of a couple of blocks that are executed in a timed sequence that is shown in the activity flow in Figure 4.29.

Basically, the main loop can be subdivided into two large parts: blocks that are being executed in the first second of the main loop, and blocks that are being executed in the second second. During the very beginning of the first second, a timer is being initialized such that every 100ms a timer interrupt is generated as described in Section 4.3.2.2. By having a counter that is incremented every time a timer interrupt occurs and by letting it count up to 10, a timer interval of 1 second is realised.

After initialization of the timer, the OBC will send out a byte with value 0xFF over the I^2C bus to the general call address. This address is like a broadcast and will be received by every subsystem that is connected to the I^2C bus [24]. This standard in the I^2C protocol is exploited and used to synchronize all the subsystem. Subsystems will start their measurements after receiving this command. The block that checks for the initial conditions of subsystems is not included in this thesis. Descriptions about the

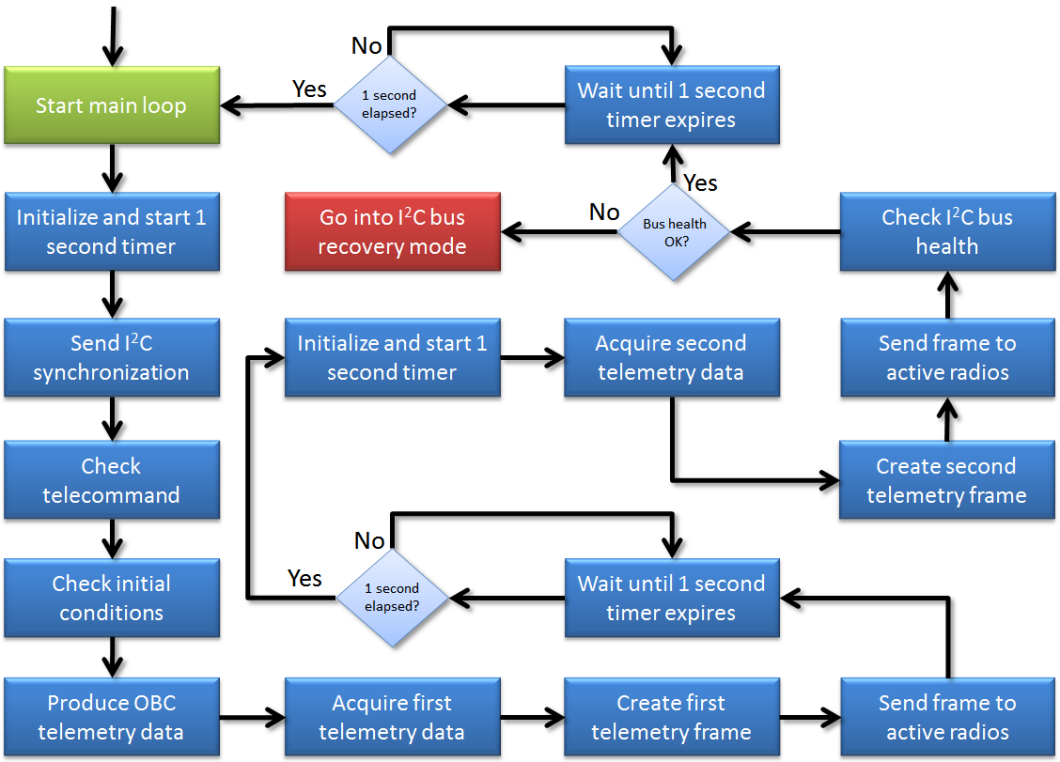


Figure 4.29: *Activity flow of the main loop*

production of OBC telemetry data, the data acquisition and telecommand execution are given in Section 4.4.4.2, Section 4.4.4.3 and Section 4.4.4.4 respectively.

When all the blocks in the first second are executed, the main loop will wait until the second is elapsed. During the second second, the OBC will acquire telemetry data from the other subsystems and forwards the acquired data to the active radio such that it can be received at the groundstation(s) on Earth. When this is completed, the main loop will again wait until the second second is elapsed. After the full two seconds the main loop will start over again, unless a telecommand has been received that lets the OBC switch to deployment mode or delay mode.

4.4.4.2 OBC Telemetry Data

The two telemetry frames that are produced by the OBC during the main loop consists of measurement (housekeeping) data from the subsystems, payload data from the payloads, housekeeping data from all the DSSB microcontrollers and housekeeping data of the OBC. The OBC inserts two types of data in the telemetry frames:

- 10-bytes telemetry frame tag data
- 14-bytes OBC housekeeping data

The 10-bytes telemetry frame tag data is inserted in both telemetry frames, whereas the 14-bytes OBC housekeeping data is inserted into the second telemetry frame only.

The 10-byte frame tag data consists of the following data in the given order [18]:

- 32-bit elapsed time counter
- 16-bit boot counter
- 31-bit frame counter
- 1-bit frame type identifier

The 32-bit elapsed time counter can be acquired by calling a function that is provided by the elapsed time counter service layer software. The elapsed time counter service layer software is not part of this thesis since it is designed and implemented by someone else. However, it is part of the OBC and therefore the data must be present in the telemetry frame tag data. The boot counter is already discussed in Section 4.4.1.1.

After the creation of each frame, a frame counter must be incremented such that, after receiving telemetry data on Earth, engineers know how many frames are already sent since the last boot. By having a 32-bit frame counter variable in the OBC, the 31-bit frame counter and 1-bit frame type identifier can be combined. Using the 1-bit frame type identifier, the first and second telemetry frames can be distinguished from one another. The 1-bit value will be set to 0 in case of the first telemetry frame and to 1 in case of the second telemetry frame.

In the second telemetry frame, the 14-bytes OBC housekeeping data will consist of the following data in the given order:

- 8-bit stuck bus counter
- 8-bit short circuit counter
- 8-bit last short circuit perpetrator
- 11-bit subsystem mode settings vector
- 5-bit allignment bits
- 8-bit OBC temperature
- 32-bit last executed telecommand (first 4 bytes)
- 1-bit last received telecommand receiver ID
- 31-bit communication status vector

The 8-bit stuck bus counter will count how many times the I^2C bus got stuck (i.e. how many times the SDA and/or SCL line were being held low for a longer period of time). The 8-bit short circuit counter simply counts how many short circuits have been occurred using the DSSB housekeeping data of all the subsystems. The I^2C address of the subsystem that was responsible for the last short circuit will be stored in the 8-bit last short circuit perpetrator variable.

The 11-bit subsystem mode settings vector indicates whether the subsystems (primary OBC, secondary OBC, primary ADCS, secondary ADCS, primary DAB, secondary DAB, PTRX, ITRX, STX, T$^3\mu$PS and SDM) should be on or off. The 5 alignment bits are used to fill up the remaining 5 bits of the two bytes that are needed by the 11-bit subsystem mode settings vector. This makes it easier to concatenate the rest of the data to the telemetry frame; without the 5 allignment bits, shift operations would be needed in order to fill the remaining bits. Shifting makes the code of the OBC more complex and error prone. Besides that, a lot of work must be done when a change is made somewhere in the telemetry frame without the 5 allignment bits.

From the ADC service layer software (see Section 4.3.4), the OBC temperature can be requested. The result of the 10-bit ADC is a 10-bit temperature sensor value. It is decided that a resolution of 8-bit for the OBC temperature is sufficient [9], so the 2 least significant bits can simply be dropped from the 10-bit temperature sensor value in order to obtain an 8-bit OBC temperature sensor value. This makes it easy to add the temperature sensor value to the telemetry frame, since it is now exactly 1 byte.

The last telecommand that is executed by the OBC must be acknowledged in the telemetry frame such that engineers at Earth know that the telecommand they sent to the satellite is successfully received and executed. For this it is sufficient to acknowledge only the first 4 bytes of the last executed telecommand. Besides the first 4 bytes of the last executed telecommand, an additional bit will be added that acknowledges the receiver ID (PTRX or ITRX) of the last received telecommand. Both should be stored in a variable by the OBC, such that they can be used once the OBC housekeeping data is inserted into the second telemetry frame.

Finally, the OBC must hold a 31-bit communication status vector that must be send with the other OBC housekeeping data to the Earth. This vector will hold a bit for every DSSB and subsystem microcontroller present in the satellite (including the body temperature sensors) that gives a status about whether or not communication is possible with that particular device. A bit with a value of 0 means that no communication is possible. A bit with a value of 1 indicates that communication is possible with the particular device. The information about whether or not communication is possible can be extracted from the I^{2}C service layer software (Section 4.3.5).

After producing the telemetry frame tag data and the OBC housekeeping data, the data can be inserted into the proper places in the first and second telemetry frames by the OBC. More about this is discussed in the next section.

4.4.4.3 Data Acquisition

During the data acquisition, all the housekeeping data and payload data is being fetched from all the other subsystems, including the DSSB microcontrollers of all the subsystems and the body temperature sensors. In total there are two data acquisition chains. The first one fetches data from the subsystems that belong to the first telemetry frame and the second one fetches data from the subsystems, DSSB microcontrollers and body temperature sensors that belong to the second telemetry frame. The OBC actually does not care about the data content. The only thing the OBC does is fetch the data, put it

in the first or second telemetry frame and forward the frame to the active transmitter. If a subsystem is turned off or not reachable, the OBC will fill the space of the subsystem in the telemetry frame with zeros. It can be distinguished whether or not the zeros are real data or not by inspecting the 31-bit communication status vector as explained in Section 4.4.4.2. This relaxes the requirement on dummy data in the telemetry frames (see Section 4.1.4).

The first data acquisition chain will fetch in total 194 bytes of data and is executed during the first second of the main loop. This is data from the following subsystems and it is being fetched in the following given order:

- 12 bytes from the PTRX
- 2 bytes from the primary DAB or secondary DAB
- 10 bytes from the first battery subsystem
- 10 bytes from the second battery subsystem
- 35 bytes from the SDM
- 95 bytes from the $T^3\mu$PS
- 30 bytes from the primary EPS or secondary EPS

Whether the OBC fetches housekeeping data from the primary or secondary DAB subsystems depends on which DAB subsystem is powered and used for data acquisition (only one DAB subsystem is allowed to be turned on). The same holds for the primary and secondary EPS subsystems. The first battery subsystem contains housekeeping data about the first and second batteries, whereas the second battery subsystem will contain housekeeping data about the third and fourth batteries. The reason why the $T^3\mu$PS and EPS subsystems are in the end of the first data acquisition chain is because, compared to the other subsystems, they need some more time in order to have their measurements ready. The first telemetry frame will contain a total of 204 bytes (the 10 telemetry frame tag bytes produced by the OBC plus the 194 bytes fetched from the other subsystems). They are packed in the first telemetry frame in the following order:

- 10 bytes telemetry frame tag data
- 12 bytes PTRX data
- 2 bytes primary DAB or secondary DAB data
- 30 bytes primary EPS or secondary EPS data
- 10 bytes first battery subsystem data
- 10 bytes second battery subsystem data
- 95 bytes $T^3\mu$PS data
- 35 bytes SDM data

The second data acquisition chain will fetch in total 177 bytes of data and is executed during the second second of the main loop. The data is fetched from subsystems, DSSB microcontrollers and body temperature sensors. The data is being fetched in the following order:

- 12 bytes from the ITRX
- 2 bytes from the STX
- 1 byte per body temperature sensor (4 in total)
- 3 bytes per DSSB microcontroller (11 in total)
- 20 bytes from the primary ADCS or secondary ADCS
- 102 bytes additional data from the primary ADCS
- 4 bytes from the ADCS magnetorquer

There is a total of 4 body temperature sensor and for each temperature sensor 1 byte of temperature sensor data must be fetched, so for the temperature sensors this results in a total of 4 bytes. For the DSSB microcontrollers a total of 33 bytes will be fetched, 3 bytes per DSSB microcontroller while there is a total of 11 DSSB microcontrollers. Whether the OBC fetches housekeeping data from the primary ADCS or secondary ADCS depends on which ADCS is operational at that moment. When the primary ADCS is operational, an additional 102 bytes of ADCS data will be fetched from the primary ADCS. If it is not operational the space will be filled with zeros. The second telemetry frame will contain a total of 201 bytes (the 10 telemetry frame tag bytes produced by the OBC and the 14 OBC housekeeping data bytes plus the 177 bytes fetched from the other subsystems). They are packed in the second telemetry frame in the following order:

- 10 bytes telemetry frame tag data
- 12 bytes ITRX data
- 2 bytes STX data
- 4 bytes body temperature sensor data
- 20 bytes primary ADCS or secondary ADCS data
- 102 bytes additional primary ADCS data
- 4 bytes ADCS magnetorquer data
- 33 bytes DSSB data
- 14 bytes primary OBC or secondary OBC data

When sending a telemetry frame to the active transmitter fails, the telemetry frame will be discarded and the OBC will not try to send it a second time to the transmitter. This is in order to assure that the main loop will not stall and will always continue with its execution at a frequency of 0.5 Hz.

4.4.4.4 Telecommand Execution

When the PTRX and ITRX are both powered, they both are able to receive telecommands that are uplinked from a ground station at Earth. Depending on the uplink frequency, the PTRX or ITRX will receive the telecommand (the radios work on different frequencies). Since the OBC does not know beforehand to which radio a telecommand is being uplinked, the OBC must check both radios for the presence of a telecommand. There are 5 different types of telecommands:

- Update nonvolatile parameter
- Update volatile parameter
- I^2C pass through
- OBC reset
- Dummy telecommand

The 5 different telecommand types can be distinguished from one another by interpreting the first byte of the telecommand.

A value of 0x01 in hexadecimal means that content in the flash memory of the OBC must be updated (update nonvolatile parameter). The 'update nonvolatile parameter' telecommand will always be followed by 2 bytes that specify the flash memory base address and an additional n bytes that will form the new content of the flash memory for the specified base address.

A value of 0x02 in hexadecimal means that the content of an OBC variable in RAM memory must be updated (update volatile parameter). Like the 'update nonvolatile parameter' telecommand, the 'update volatile parameter' telecommand will always be followed by 2 bytes that specify the flash memory base address and an additional n bytes that will form the new content of the OBC variable. The specified flash memory base address will be mapped to the proper volatile OBC variable, and the OBC will update that variable accordingly.

The telecommand is a 'I^2C pass through' telecommand if the first byte of the telecommand has a value of 0x04 in hexadecimal. The 'I^2C pass through' telecommand can be used to send a telecommand from Earth immediately to any powered I^2C device that is present on the bus. The second byte of the telecommand specifies the 7-bit I^2C address of the device to which data must be send (the most significant bit of that byte will always have a value of 0, the least significant 7-bits specify the I^2C address). The next n bytes will contain the data that must be forwarded from the OBC to the specified I^2C device.

A first telecommand byte with a value of 0x40 in hexadecimale will result in a reset of the OBC. Just the first byte is not sufficient to reset the OBC. In order to be sure that the telecommand is realy an 'OBC reset' telecommand, a safety content byte is followed. The safety content byte must have a value of 0xAA in hexaxecimal. If this is the case, the OBC will reboot itself immediately.

Finally, a so called 'Dummy telecommand' exists which is a telecommand with a first byte value of 0x80. The OBC just does nothing after reception of this telecommand. The only thing the OBC does is store the telecommand in the 'last executed telecommand'

variable which is done for any valid received and executed telecommand. The purpose of the dummy telecommand is purely to test the uplink to the satellite. In the next telemetry frame after execution of the telecommand, the engineers at Earth can check whether or not the uplinked telecommand is realy received by one of the radios and executed by the OBC.

An activity flow of the procedure for checking the radios for a telecommand and executing the telecommand is given in Figure 4.30.

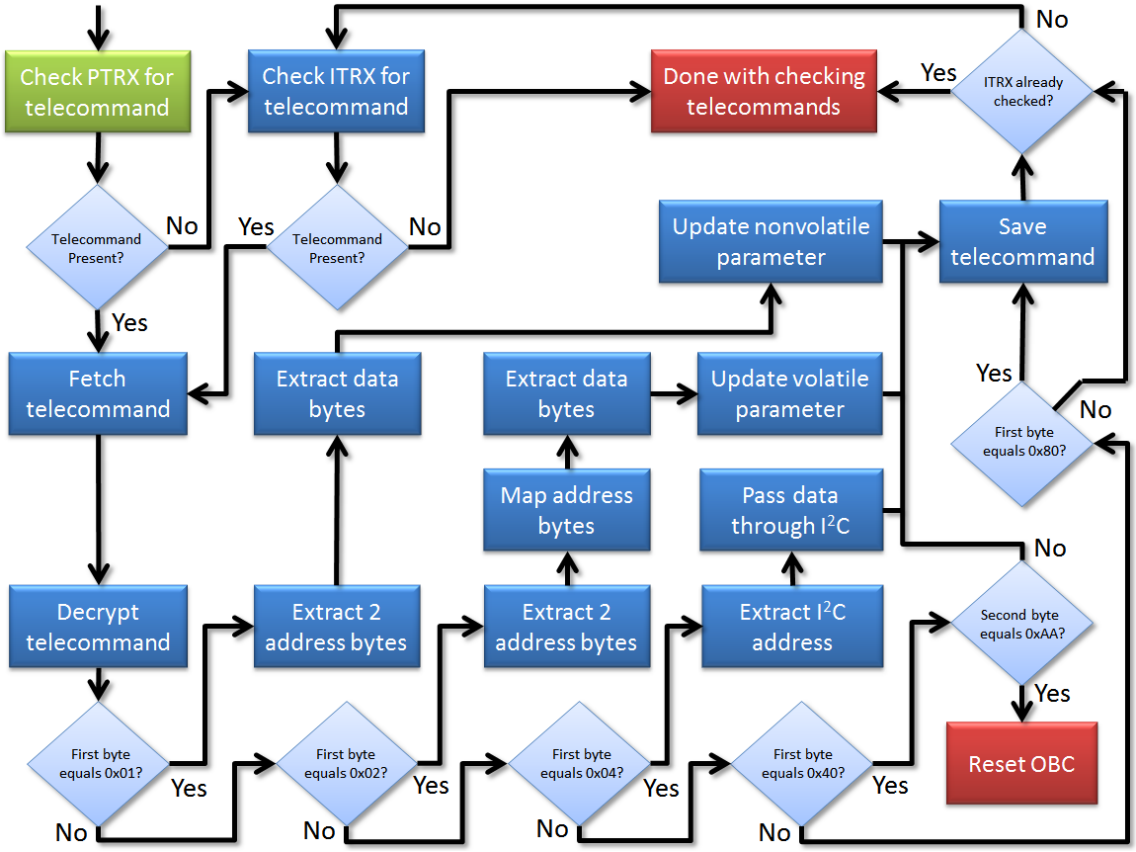


Figure 4.30: *Activity flow for checking and executing a telecommand*

It is worth to mention that the decryption of telecommand data is part of this thesis. For security reasons, telecommands that are being uplinked from a ground station to the satellite are encrypted. By encrypting telecommands and keeping secret the encryption details (the algorithm and the encryption/decryption keys), other users than the ones who know the encryption details cannot command the satellite by uplinking telecommands.

Though the decryption mechanism is part of this thesis, details about the encryption algorithm will not be discussed in this thesis since this thesis will be published publicly.

4.4.5 I²C Bus Recovery Mode

As already quickly discussed, a switch from the main mode to the I²C bus recovery mode will be made when the I²C bus got stuck. The I²C bus gets stuck when one of the I²C lines (SDA or SCL) are being pulled low for a longer period of time. If the I²C bus got stuck for 4 seconds (i.e. 2 times the execution of the main loop), the main mode will switch to the I²C bus recovery mode.

Once entered, the I²C bus recovery mode will continuously send out I²C synchronization commands using the I²C general call. The OBC will stay into the I²C bus recovery mode until the I²C bus is considered as recovered and healthy, meaning that none of the I²C lines are being pulled low. Once the health of the I²C bus is finally recovered, the OBC can switch back to the main mode. The activity flow is shown in Figure 4.31.

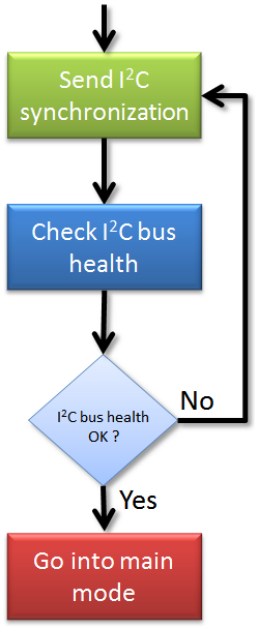


Figure 4.31: *Activity flow for the I²C bus recovery mode*

A stuck I²C bus can be caused by any device that is connected to the I²C bus SDA and SCL lines. Most of the times, a reset of the device that causes the stuck I²C bus can solve the problem and can fully recover the I²C bus such that it is returned into a healthy state. However, devices can not know whether they are pulling low one of the I²C lines, or that another device is responsible for that. Therefore, like the OBC, all the other subsystem microcontrollers and DSSB microcontrollers have an I²C recovery mechanism as well. For these microcontrollers, a timer is implemented that counts the amount of elapsed time since reception of the last I²C synchronization. A predefined sequence is specified in which is stated how a microcontroller is being reset and at what time after reception of the last I²C synchronization [18].

In the recovery sequence, the subsystem microcontrollers will first reset themselves after a certain period of time since reception of the last I^2C synchronization command. When this does not recover the bus, the DSSB microcontrollers of all the subsystems will perform a full subsystem reset in a certain sequence. If this still does not result in a recovered bus, the DSSB microcontrollers will start to reset themselves in the same defined sequence as before. In case of still no success in recovery, the primary and secondary OBC will start to reset themselves, including their DSSB microcontrollers. As one of the latest steps, the EPS subsystem will perform a full power cycle of the complete satellite and in the end perform a reset of the microcontrollers on the EPS subsystem itself. At this point the I^2C bus should be recovered. If not, a serious problem which can not be resolved is responsible for the stuck bus. Usually, this means end of the mission since the satellite will not be operable anymore. For completeness, the I^2C recovery timings of resetting the I^2C devices after the last reception of the I^2C synchronization are given in Table 4.2.

4.5 Summary

In this chapter, the software requirements for the Delfi-n3Xt CDHS are presented and a detailed description about the Delfi-n3Xt OBC software architecture is given. Besides that, a detailed design description about the Delfi-n3Xt OBC software is presented, which consists of the service layer software and the application layer software.

The requirements are split up into five different categories about subsystem communication, fault-tolerant software, telecommanding, data acquisition and monitoring. A requirement can be either a constraint, functional requirement, performance requirement or an interface requirement.

The architecture of the Delfi-n3Xt OBC software is subdivided in two layers: the service layer and the application layer. The service layer software is the software that interfaces with the MSP430F1611 microcontroller hardware. It consists of modules that drive the MSP430F1611 peripherals like the programmable interval timers, flash memory controller, analog to digital converter, I^2C controller and watchdog timer. The service layer software acts as a bridge between the MSP430F1611 hardware peripherals and the application layer software.

Speaking in terms of software architecture, the application layer software lays on top of the service layer software. This application layer software consists of a sublayer that implements subsystem specific functionalities and the implementation of the different operational modes of the satellite. There is a total of 6 different operational modes: the boot mode, delay mode, deployment mode, main mode, I^2C bus recovery mode and the test mode. All the modes are discussed in detail in this chapter, except for the test mode. The test mode is not part of this thesis.

Recovery type	I^2C device	Time since last I^2C sync
Subsystem microcontroller reset by internal watchdog	SDM	4200 ms
	$T^3\mu PS$	4250 ms
	primary DAB	4300 ms
	secondary DAB	4350 ms
	ADCS magnetorquers	4400 ms
	STX	4450 ms
	ITRX ITC	4500 ms
	ITRX IMC	4550 ms
	PTRX ITC	4600 ms
	PTRX IMC	4650 ms
	secondary ADCS	4700 ms
	primary ADCS	4750 ms
	primary battery	4800 ms
	secondary battery	4850 ms
Subsystem switch off by DSSB timer	SDM	5000 ms
	$T^3\mu PS$	5200 ms
	primary DAB	5400 ms
	secondary DAB	5600 ms
	STX	5800 ms
	ITRX	6000 ms
	PTRX	6200 ms
	primary ADCS	6400 ms
	secondary ADCS	6600 ms
DSSB microcontroller reset by internal watchdog	SDM DSSB	8000 ms
	$T^3\mu PS$ DSSB	8050 ms
	primary DAB DSSB	8100 ms
	secondary DAB DSSB	8150 ms
	STX DSSB	8200 ms
	ITRX DSSB	8250 ms
	PTRX DSSB	8300 ms
	primary ADCS DSSB	8350 ms
	secondary ADCS DSSB	8400 ms
OBC reset sequence	secondary OBC microcontroller	8450 ms
	secondary OBC subsystem	8500 ms
	secondary OBC DSSB microcontroller	8550 ms
	primary OBC microcontroller	8600 ms
	primary OBC subsystem	8650 ms
	primary OBC DSSB microcontroller	8700 ms
Satellite power cycle reset	by primary EPS	10000 ms
	by secondary EPS	15000 ms
EPS microcontroller reset	primary EPS microcontroller	16000 ms
	secondary EPS microcontroller	16000 ms

Table 4.2: I^2C recovery reset sequence and timings

On-Board Computer Software Implementation

5

In this chapter, the implementation of the baselined Delfi-n3Xt OBC software design that was discussed in Chapter 4 is presented. The implementation of the OBC software is done in the programming language C and the software is executed on a Texas Instruments MSP430F1611 microcontroller. The implementation details about the service layer software are given in Section 5.1 whereas details about the application layer software implementation are given in Section 5.2. This chapter is summarized in Section 5.3.

5.1 Service Layer Software

This section describes the implementation of the Delfi-n3Xt OBC service layer software. The implementation shows how the peripherals of the MSP430F1611 microcontroller can be driven such that they fulfil the design requirements that are specified in the previous chapter. Section 5.1.1 describes the implementation that configures the clock source peripheral of the microcontroller and Section 5.1.2 shows how the programmable interval timers can be configured and used. The implementation that configures and uses the flash memory controller is given in Section 5.1.3, and that for the analog to digital converter in Section 5.1.4. The implementation of the I^2C controller driver is discussed in Section 5.1.5 and finally, in Section 5.1.6, the watchdog driver implementation is provided.

5.1.1 Clock Source Module

The Delfi-n3Xt OBC clock source module consists of a function that configures the MSP430F1611 clock system according to the OBC hardware design explained in Section 2.3 and the activity flow shown in Figure 4.4. The task of the clock configuration function is to properly set up the contents of the BCSCTL1 (Figure 5.1) and BCSCTL2 (Figure 5.2) registers. Both registers are 8-bits wide.

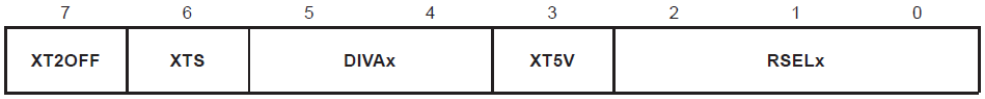


Figure 5.1: *The BCSCTL1 register that is used to configure the clock system*

The BCSCTL1 register consists of 5 fields: XT2OFF, XTS, DIVAx, XT5V and RSELx. Only the first three fields are of relevance and the latter two are unused so they do not need to be configured (they will be just left to a value of 0).

A logical value of 0 for the XT2OFF bit means that the crystal on the second clock source input of the MSP430F1611 is always on. In Section 2.3 it was already explained that the second clock source input is a clock signal driven by an external 8MHz crystal and that it must drive the MSP430F1611 CPU and most of its peripherals. Therefore the value of the XT2OFF bit must be set to 0.

With the XTS bit it can be specified whether or not the first clock source input is a low frequency clock signal originating from a 32kHz external crystal or a high frequency clock signal originating from an external crystal with a frequency between 450kHz and 8MHz [28]. A value of 0 for this XTS bit means that the first clock source input is a 32kHz external crystal. This is what is needed for the implementation of the clock source module, since a 32kHz crystal is connected to the first clock source input of the MSP430F1611 of the OBC. Hence, the XTS bit must be set to 0 as well.

The DIVAx field specifies the clock divider for the auxiliary clock (see Section 4.3.1.1). The clock divider can be set to 1 (no clock division), 2, 4 or 8 and they can be set by loading the bit values 00, 01, 10 or 11 respectively in the DIVAx field of the BCSCTL1 register. The watchdog timer peripheral for the Delfi-n3Xt OBC requires an auxiliary clock divider of 8, as discussed in Section 5.1.6, so the 2-bits in DIVAx field must be set to a logical 1 such that the MSP430F1611 auxiliary clock will provide a 4096 Hz clock signal. The final configuration for the BCSCTL1 register will be set to a value of 00110000 in binary or 0x30 in hexadecimal.

The BCSCTL2 register consists of 5 fields as well: SELMx, DIVMx, SELs, DIVSx and DCOR. The DCOR field is not of relevance. The first two fields (SELMx and DIVMx) select the clock source and the clock divider for the main clock. The other two fields (SELS and DIVSx) select the clock source and divider for the sub-main clock.

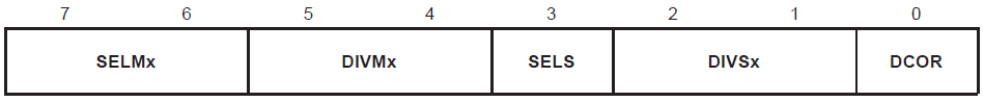


Figure 5.2: *The BCSCTL2 register that is used to configure the clock system*

For the main clock, the clock source can be configured such that the main clock is driven by the internal DCO clock source, the first external clock source or the second external clock source. The main clock must run on the 8 MHz external crystal so the clock source must be set to the second external clock source. This is achieved by setting both bits in the SELMx field to a logical 1. There should be no clock divider for the main clock so both bits in the DIVMx field must be set to a logical 0.

The sub-main clock can be configured such that it is driven by the internal DCO clock source (the SELS bit must be set to 0 in this case) or one of the external clock sources (SELS set to 1). For the OBC, the sub-main clock must be driven by the external 8 MHz crystal so the SELS bit must be set to 1 and no clock divider must be used (so the two bits in the DIVSx field must be set to 0). By setting the SELS bit to 1 the MSP430F1611 will first try to use the second clock source (the 8 MHz crystal) as clock input, and if it is not present it will try to use the first clock source. Since the second clock source is present, the sub-main clock will be driven by the 8 MHz crystal. The content of the BCSCTL2 register thus must be configured to 11001000 in binary or 0xC8.

5.1.2 Programmable Interval Timers

For the OBC of Delfi-n3Xt, two MSP430F1611 PITs are used: timer A and timer B. Timer A is used to handle I^2C time-outs and timer B is used for timing of the main loop. The MSP430F1611 TACCR, TACCTL and TACTL registers can be used to configure and use timer A (see Section 5.1.2.1) and the TBCCR, TBCCTL and TBCTL registers can be used for timer B (see Section 5.1.2.2). Furthermore, interrupt routines are entered when a timer expires. More about the interrupt routines for timer A and B is discussed in Section 5.1.2.3.

5.1.2.1 Timer A

The three MSP430F1611 registers used to configured Timer A are TACCR, TACCTL and TACTL. The TACCR register is a 16-bit wide register that holds the timer threshold value. When the amount of counted timer ticks equals the value in the TACCR register, an interrupt will be generated by the timer A peripheral. The other two registers (TACCTL and TACTL) are 16-bit wide as well and are shown in Figure 5.3 and Figure 5.4 respectively. Since timer A has to handle I^2C time-outs, the threshold value of the timer must be set to 30ms. From Section 4.3.2 it can be derived that a timer interval of 30ms can be achieved by using the timer in 'up mode', use a clock divider of 8 and set the timer its threshold value (i.e. the content of the TACCR register) to 30000 (the timer interval in microseconds). The TACCR register can simply be configured by loading the decimal value of 30000 into the register: TACCR = 30000.

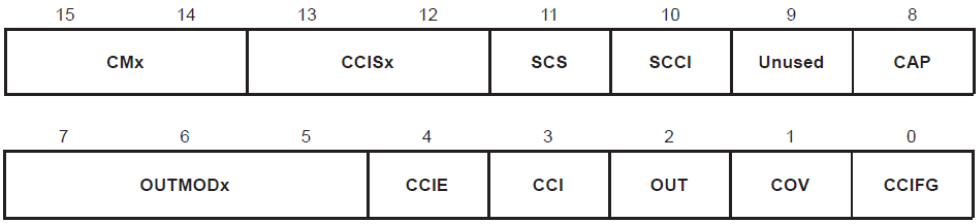


Figure 5.3: *The TACCTL register that is used to set the timer threshold value*

Actually, for the TACCTL register, the only configuration that must be performed is setting the CCIE bit high to a logical value of 1. The other fields in the register should not be altered. Setting the CCIE bit high enables the capture/compare interrupt of timer A such that the timer A peripheral is able to generate an interrupt when the amount of elapsed timer ticks is equal to the threshold value stored in the TACCR register. Setting the CCIE bit field high and leaving the other fields of the register untouched can be performed by a simple bitwise OR operation on the CCIE bit field: TACCTL |= 0x0010.

The only thing that must be done in order to have the timer working properly is selecting the clock source for the timer, the clock divider for the input clock signal and configuring the timer in 'up mode'. Having the timer in 'up mode' is sufficient since no timer interval of 65ms or higher is required. Configuring the clock source, clock divider and timer mode is all done using the TACTL register (see Figure 5.4).

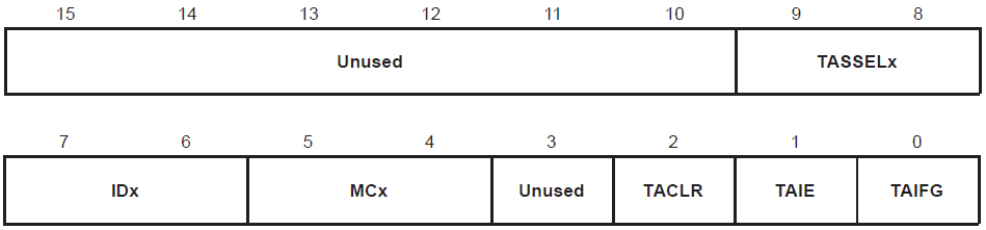


Figure 5.4: *The TACTL register that is used to configure timer A*

The clock source for the timer A peripheral can be selected with the 2-bit wide TASSELx field in the register. In the design section of the PIT module (Section 4.3.2) it was already explained that the clock source must be configured to the 8 MHz external crystal clock source. The main clock and the sub-main clock are driven by this 8 MHz crystal. For the timer A clock source selection there are four possible selections [28]. The selection for SMCLK (the sub-main clock) is appropriate. To do so, the two bits in the TASSELx field must be set to 1 and 0 (bits 9 and 8 in the TACTL register respectively). The timer A clock source divider can be set through the 2-bit wide IDx field in the TACTL register. With that field, four possible input clock dividers can be configured: 1 (no clock division), 2, 4 and 8. For the timer A configuration an input clock divider of 8 is needed. An input clock divider of 8 corresponds to setting both bits high in the IDx field (bits 7 and 6 in the TACTL register).

Finally, the timer mode must be set such that the timer is configured to work in 'up mode'. Setting the timer mode is done through the 2-bit wide MCx field of the register. Again, four modes are possible: stop mode, up mode, continuous mode and up/down mode. The 'up mode' corresponds to the bit values of 0 and 1 in the IDx field (bits 5 and 4 in the TACTL register respectively). Furthermore, it is worth to mention that during configuration of the timer peripheral, the TACLR bit in the TACTL register must be set high such that the internal timer tick counter of the timer peripheral is cleared and thus reset to 0.

5.1.2.2 Timer B

The registers belonging to the timer B peripheral (TBCCR, TBCCTL and TBCTL) work in the same way as the registers for timer A as discussed in the previous section. The only difference in the configuration of the timer B peripheral compared to the configuration of the timer A peripheral is the content of the threshold register and the timer mode. The reason for a different timer mode is because a timer interval of 100ms is desirable to time the main loop that is part of the OBC application layer software. Since a timer interval of 100ms is greater than 65ms, the timer B peripheral must be configured to operate in 'up/down mode', since this provides a twice as long timer interval compared with the 'up mode' with a maximum of about 131ms.

The timer B peripheral will generate an interrupt every 100ms if the timer threshold value is set to 50000 and the timer mode is in 'up/down mode' (as computed in Section 4.3.2.2). In order to do this, the two bits in the MCx field in the TBCTL register must be set to 1 and 0 (bit 5 and 4 in the TBCTL register respectively).

5.1.2.3 Interrupt routines

When a timer peripheral generates an interrupt, the corresponding interrupt routine will be entered. The interrupt routine that belongs to the timer A peripheral sets a flag and clears the CCIE bit in the TACCTL register such that the timer is disabled and will not continue with counting and generating interrupts. The flag that is being set in this interrupt routine is used by the I^2C service layer software in order to detect whether or not a time-out on the I^2C bus has occurred.

The interrupt routine that corresponds to the timer B peripheral is more complex compared to the one of timer A. It does not need to disable the timer B peripheral because the timer must run continuously. The complexity lies in the different flags that must be set in the interrupt routine. The main loop polls for the different flags such that exact timing can be achieved in order to fetch the housekeeping and payload data from the various subsystems at the right time.

5.1.3 Flash Memory Controller

The flash memory controller of the MSP430F1611 is driven by three 16-bit registers: FCTL1, FCTL2 and FCTL3. The higher byte of all the three registers is a security byte that always must have a value of 0xA5 when data is written to the register. When data different than 0xA5XX is written to one of the 16-bit registers, the flash memory controller generates a PUC that immediately resets the microcontroller [28]. From Section 4.3.3 it is already known that the flash module must consist of 4 service routines: configuring and initializing the flash memory timing generator (Section 5.1.3.1), reading from the flash memory (Section 5.1.3.2), writing to the flash memory (Section 5.1.3.3) and erasing a segment in the flash memory (Section 5.1.3.4).

5.1.3.1 Initializing the flash timing generator

Setting op the flash memory controller consists of initializing the flash memory timing generator. The flash memory timing generator controls the data rate on which data is read from and written to flash memory. It must operate in the frequency range from 257kHz to 476kHz [1]. It is up to the software engineer to select a proper input clock source and flash controller clock divider such that the desired operating frequency of the flash memory timing generator is within the specified frequency range. Configuration of the flash timer generator is done through the FCTL2 register that is shown in Figure 5.5.

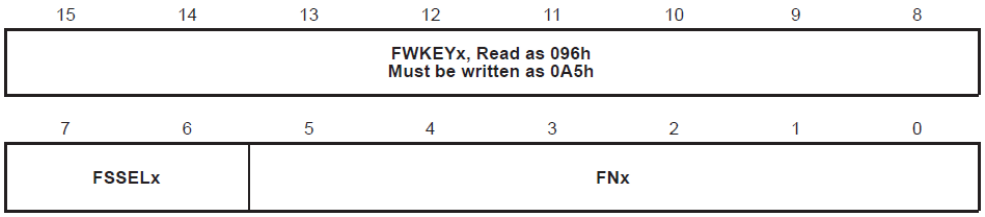


Figure 5.5: *The FCTL2 register that is used to initialize the flash timing generator*

As already mentioned, the higher byte of the 16-bt FCTL2 register must be written with the value 0xA5. The lower byte of the register contains 2 fields: the 2-bits wide FSSELx field and the 6-bits wide FNx field. The 2-bits wide FSSELx field is used to select the clock source input. For the OBC this must be an 8MHz input so the main clock (MCLK) or sub-main clock (SMCLK) must be used (see Section 4.3.1). The following binary values for the FSSELx field will select the following clock source inputs:

- 00 ACLK
- 01 MCLK
- 10 SMCLK
- 11 SMCLK

Since MCLK or SMCLK must be used, the FSSELx field must be set to 01, 10 or 11. The implementation for the OBC uses MCLK so the 2-bit wide FSSELx field contains the corresponding bit configurations for MCLK as given above.

The 6-bit wide FNx field configures the flash memory controller clock divider. It must be properly set such that the 8MHz MCLK source input is divided to a frequency that is in the range from 257kHz to 476kHz. The divisor value will be FNx + 1 [28], so a clock division factor between 1 and 64 can be configured. For the OBC, bits 4, 1 and 0 are set high. This means that the FNx field is loaded with the value 19, resulting in a clock divider value of 20. With the 8MHz MCLK input this will result in a flash timing generator frequency of 400kHz, which is safely within the specified frequency range. The above explained configuration results in a FCTL2 register content value of 0xA553 in hexadecimal.

5.1.3.2 Reading from flash memory

The most simple flash memory operation on the MSP430F1611 microcontroller is reading from flash memory. Since the flash memory controller is memory mapped, no registers of the flash memory controller peripheral needs to be configured in order to read data from the flash memory. By letting a pointer point to the flash memory base address where the data is stored, the content of the flash memory can easily be copied to a volatile memory space in the microcontroller its RAM using *memcpy*.

5.1.3.3 Writing to flash memory

The FCTL1 and FCTL3 registers are used to write data to flash memory and to erase a segment of the flash memory (Section 5.1.3.4). Before writing data to the flash memory, global interrupts must be disabled since the flash memory controller must not be interrupt while it is writing to the flash memory [28]. Without global interrupts disabled, the flash memory controller may not write data correctly to the flash memory because it may get interrupted by the CPU when a higher priority peripheral interrupts the processor. After the writing process, global interrupts can safely be re-enabled such that other peripherals are again allowed to interrupt the CPU. The actual writing to the flash

memory is controlled with the FCTL1 (see Figure 5.6) and FCTL3 (see Figure 5.7) registers. In order to write to flash memory, the WRT bit of the FCTL1 register is used and the LOCK bit of the FCTL3 register.

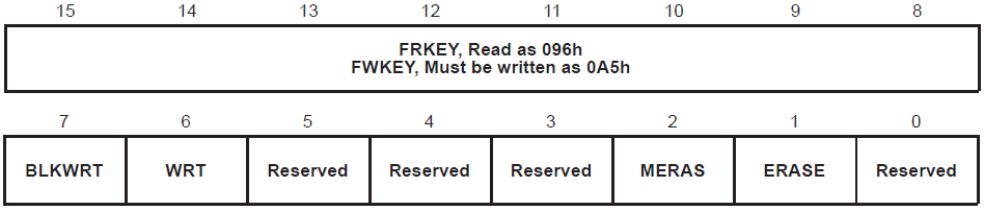


Figure 5.6: *The FCTL1 register that is used to control the flash memory controller*

In order to write data to the flash memory, the WRT bit must first set high in the FCTL1 register. After setting this bit high, data can be written to the content of a flash memory address. This can be realized by a pointer that points to the flash memory address destination the byte must be written to. The flash memory controller will further handle the actual writing to the flash memory. The FCTL1 register can be configured for a flash memory write operation by loading it with the 16-bit value 0xA540.

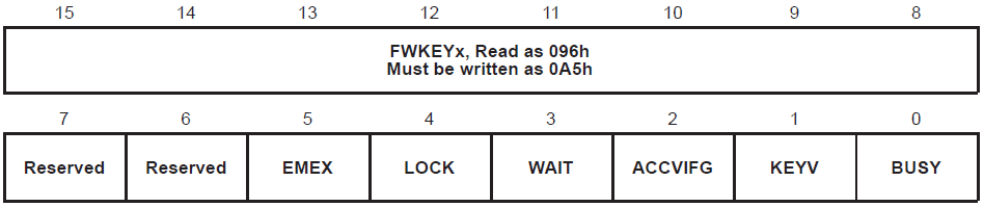


Figure 5.7: *The FCTL3 register that is used to control the flash memory controller*

The LOCK bit in the FCTL3 register is used to lock the flash memory controller until it is finished with writing the data to the flash memory. This ensures that no other flash memory write or erase operation interferes with the current write operation. The LOCK bit is automatically cleared by the flash memory controller when the data is written to the flash memory. In order to lock the flash memory controller, the FCTL3 register must be loaded with the value 0xA510. The final step in finishing the flash memory write operation is to re-enable global interrupts such that the CPU can get interrupted by other peripherals again like for instance one of the timers.

5.1.3.4 Erasing a segment

A flash memory segment can be erased by setting the ERASE bit field in the FCTL1 register high. After setting the ERASE bit high, an arbitrary byte within the desired flash segment that must be erased has to be written with the value 0. Like writing to flash memory, the LOCK bit in the FCTL3 register must be set to 1 in order to start the erase process. The global interrupts must be disabled as well before a flash memory segment is being erased.

5.1.4 Analog to Digital Converter

The ADC module that uses the 10-bit analog to digital converter to read out the OBC temperature sensor can be configured through the 16-bits wide ADC10CTL0 and ADC10CTL1 registers. Besides these two control registers, there is a 16-bit register named ADC10MEM that holds the 10-bit conversion result. In Section 5.1.4.1 it is described how the ADC peripheral can be properly configured in order to read out the MSP430F1611 microcontroller chip temperature. Section 5.1.4.2 shows how the analog to digital conversion can be started to obtain the temperature sensor read-out.

5.1.4.1 Configuring the A/D converter

The design of the ADC module in Section 4.3.4 already describes that in order to read out the microcontroller chip its temperature, the reference voltage and conversion channel of the ADC peripheral must be properly configured. The ADC10CTL0 register shown in Figure 5.8 is used to set the reference voltage. In order to set up the reference voltage for chip temperature read-out, the SREFx, REFON and REF2_5V fields in the register must be loaded with correct configuration values.

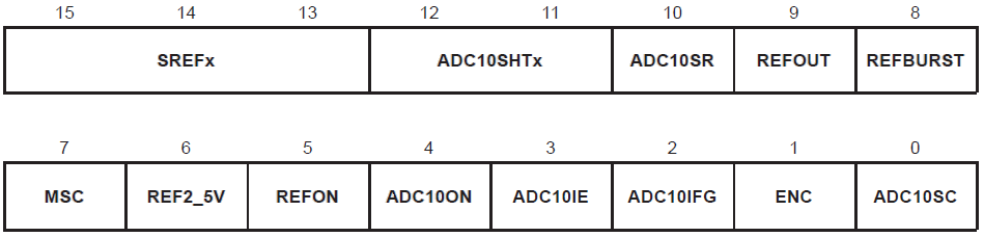


Figure 5.8: *The ADC10CTL0 register that is used to configure the A/D converter*

From the temperature sensor transfer function shown in Figure 4.12, it can be seen that the minimum output voltage of the temperature sensor is around 0.825V and the maximum output voltage around 1.35V. In order to increase the precision of the read-out, an appropriate reference voltage must be configured. For the MSP430F1611, a voltage reference generator can be used that can generate a reference voltage of 1.5V or 2.5V. This can be set up by the REF2_5V field in the ADC10CTL0 register (Figure 5.8). Setting the REF2_5V bit low lets the voltage reference generator generate a reference voltage of 1.5V, which is the appropriate reference voltage for the temperature sensor read-out. Furthermore, in order to use the reference generator, the REFON bit must be set to 1. The reference voltage must be selected by the 3-bit SREFx field which can be one of the following 8 options:

- 000 $V_{R+} = V_{CC}$ and $V_{R-} = V_{SS}$
- 001 $V_{R+} = V_{REF+}$ and $V_{R-} = V_{SS}$
- 010 $V_{R+} = Ve_{REF+}$ and $V_{R-} = V_{SS}$
- 011 $V_{R+} = Ve_{REF+}$ and $V_{R-} = V_{SS}$

- 100 $V_{R+} = V_{CC}$ and $V_{R-} = V_{REF-}/Ve_{REF-}$
- 101 $V_{R+} = V_{REF+}$ and $V_{R-} = V_{REF-}/Ve_{REF-}$
- 110 $V_{R+} = Ve_{REF+}$ and $V_{R-} = V_{REF-}/Ve_{REF-}$
- 111 $V_{R+} = Ve_{REF+}$ and $V_{R-} = V_{REF-}/Ve_{REF-}$

In order to use the voltage reference generated by the voltage reference generator, V_{R+} must be set to V_{REF+}. Besides that, V_{R-} must be set to V_{SS} which is in the case of the OBC connected to ground and it is equal to 0V. This means that a measured voltage of V_{REF+}, which is in this case set to 1.5V, will represent a digital conversion value of $2^{10} - 1 = 1023$ and a measured voltage of 0V represents a digital conversion value of 0. So the 3-bit SREFx field must be loaded with the binary value 001. As a final setting for the ADC10CTL0 register, the 10-bit ADC peripheral must be (interrupt) enabled so the ADC10ON, ADCIE and the ENC bits in the register must be set high. This results in an ADC10CTL0 register value of 0x203A.

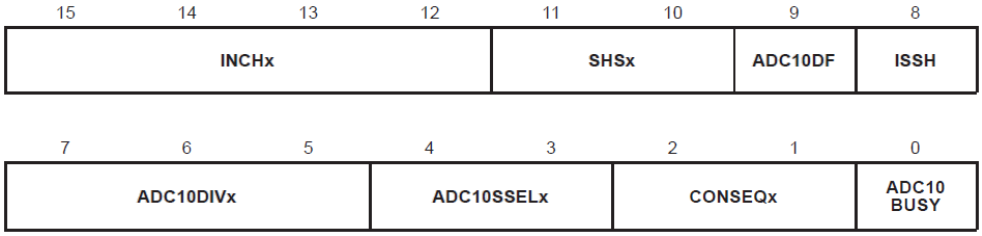


Figure 5.9: *The ADC10CTL1 register that is used to configure the A/D converter*

The ADC10CTL1 register shown in Figure 5.9 is used to select the input channel that should be used by the A/D converter to perform the A/D conversion on. This input channel selection is done by configuring the 4-bits wide INCHx field of the register. A value of 1010 in binary selects the input channel to which the temperature sensor is connected so the content of this register must be set to 0xA000. Configuring the A/D converter should be done only once during boot time of the OBC.

5.1.4.2 Temperature sensor read-out

Now that the ADC peripheral is correctly configured, it can be used to perform conversions in order to measure the ambient temperature of the MSP430F1611 microcontroller chip. To start a conversion, the only thing that needs to be set is the ADC10SC bit in the ADC10CTL0 register. After setting this bit high, the A/D converter will immediately perform an analog to digital conversion on input channel 10 to which the internal temperature sensor is connected. The conversion should not take longer than a few microseconds. Since the ADC10IE bit is set high during the configuration of the A/D converter, the A/D converter will generate an interrupt when the conversion is done and the converted value is ready. The converted value is stored in the 16-bits wide ADC10MEM register that is shown in Figure 5.10.

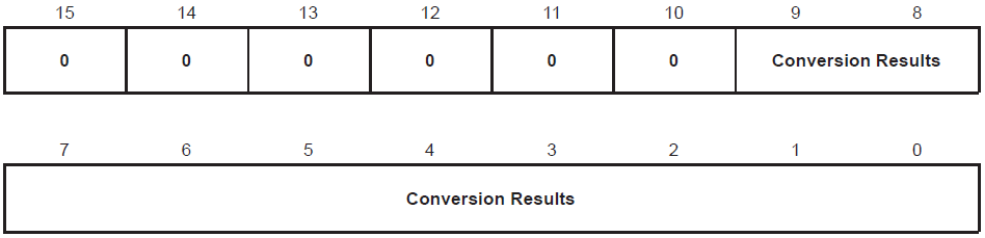


Figure 5.10: *The ADC10MEM register that is used to store the A/D conversion result*

As can be seen from the figure, the conversion result is stored in bits 9 to 0 and bits 15 to 10 are not used and have a fixed value of 0. Bit 9 is the most significant bit of the conversion result and bit 0 the least significant bit. The conversion result is stored in a local 16-bit wide variable and is corrected by the offset value that was determined during the calibration process. Application layer software can in turn fetch the local variable from the ADC server layer software module in which the offset corrected conversion result is stored.

5.1.5 I^{2}C Controller

Because of the OBC redundancy requirement, the primary and secondary OBC can be in an active or an inactive state. The OBC that is in the active state will control the I^{2}C bus and is by definition the I^{2}C master. The inactive OBC will be an I^{2}C slave that only listens on the I^{2}C bus in order to check whether or not the active OBC that is I^{2}C master is still alive and still sending out synchronization commands. When the inactive OBC does not receive synchronization commands anymore for a certain amount of time, the inactive OBC must become active and take over control of the I^{2}C bus. The take-over machanisms that handle OBC redundancy at the application layer level are not part of this thesis. Only the I^{2}C master and I^{2}C slave service layer software for the OBC is part of this thesis and some of the implementation details are given in this Section. Section 5.1.5.1 gives implementation details about the I^{2}C master driver and in Section 5.1.5.2 the implementation details about the I^{2}C slave driver are discussed.

5.1.5.1 I^{2}C Master

Basically, the I^{2}C master service layer software consists of three functional blocks; configuration of the I^{2}C peripheral in master mode, initiating a write operation and write data on the I^{2}C bus and initiating a read operation and read data from the I^{2}C bus. In this section, a general description is given about the I^{2}C master module implementation, including the used registers and how they are configured.

During configuration of the I^{2}C peripheral in master mode, the U0CTL, I2CPSC, I2CSCLH, I2CSCLL and I2CTCTL registers must be configured properly. The 8-bits wide U0CTL register is used to configure the Universal Serial Asynchronous Receiver/Transmitter (USART) into I^{2}C mode. The register consists of 8 fields that are 1 bit wide each. The register, together with its fields, is shown in Figure 5.11.

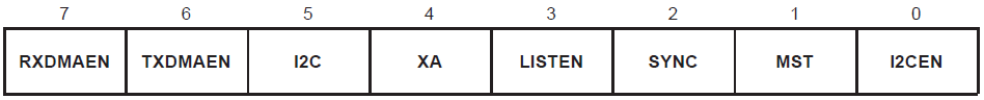


Figure 5.11: *The U0CTL register that configures the MSP430F1611 USART peripheral*

In order the configure the USART into I^2C master mode, the I2C, SYNC, MST and I2CEN bits must be set high. The I2C and I2CEN bits are obvious; they configure the USART for I^2C mode. The MST bit enables master mode so the USART knows it has to generate the I^2C clock pulses on the I^2C clock line during an I^2C transfer. The SYNC bit must be set such that the USART goes into synchronous mode. This is needed since the I^2C protocol is synchronous. Consequently, the U0CTL register must be loaded with the value 0x27.

The I2CTCTL register is used to initiate data transfers and select the clock source input that drives the I^2C peripheral. To configure the I^2C peripheral, the clock source should be selected by setting the 2-bits wide I2CSSELx field in the I2CTCTL register (see Figure 5.12).

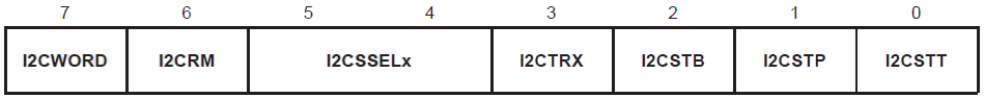


Figure 5.12: *The I2CTCTL register that configures and drives the I^2C peripheral*

The clock source input must be at least 10 times higher than the I^2C bus frequency [24] which operates at 100kHz, so the clock source input frequency must be at least 1MHz. The only two options to select are the ACLK and SMCLK clock sources, which have a frequency of 4096kHz and 8MHz respectively. Therefore the SMCLK must be selected. The SMCLK is selected by writing a binary value of 10 or 11 into the I2CSSELx field in the I2CTCTL register.

The last thing that is needed to properly implement the I^2C master service layer software is configuring the I2CPSC, I2CSCLH and I2CSCLL registers. These registers configure the I^2C bus frequency and the clock high and clock low periods. The three registers all consist of one big 8-bit field as shown in Figure 5.13.

In order to have an I^2C bus frequency of 100 kHz, the SCL high period and SCL low period must both be set to 5 μs. The period of the clock input signal is 125 ns and the SCL high period will be equal to the following [28]:

$$T_{high} = (I2CSCLH + 2) \cdot \frac{1}{f_{input}} \tag{5.1}$$

Without a clock input divider (meaning the I2CPSC register must be loaded with value 0), f_{input} will be 8 MHz and Equation 5.1 reduces to

$$T_{high} = (I2CSCLH + 2) \cdot 0.000000125s \tag{5.2}$$

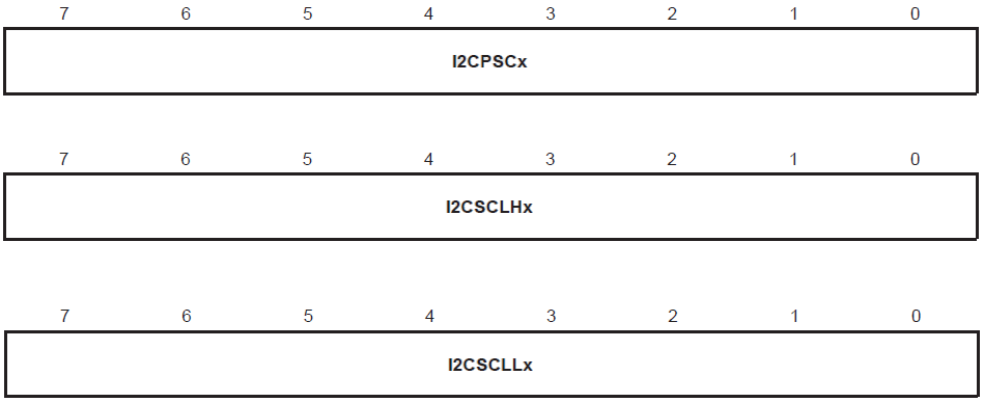


Figure 5.13: *The I2CPSC, I2CSCLH and I2CSCLL registers that configure the bus speed*

Which can be rewritten in order to solve for I2CSCLH:

$$I2CSCLH = \frac{T_{high}}{0.000000125s} - 2 \tag{5.3}$$

Since T_{high} must be 5 μs, the value that must be loaded into I2CSCLH equals 38 in decimal (0x26 in hexadecimal). Configuring the clock low time works the same as configuring the clock high time. The clock high time and clock low time must both equal to 5 μs so the value that is just computed for I2CSCLH can be loaded in the I2CSCLL register as well. So the final register configurations for configuring the bus frequency to 100 kHz become

I2CPSC = 0x00;
I2CSCLL = 0x26;
I2CSCLH = 0x26;

A write or read operation can be initiated by the master by configuring the I2CSA, I2CNDAT and I2CTCTL registers. The I2CSA register is a 16-bits wide register that holds the 7-bit I^2C address of the slave with which the master likes to communicate. Furthermore, the I2CNDAT register is 8-bits wide and contains the number of bytes that must be received or transmitted. The I2CNDAT register is used by the automatic data byte counting feature of the MSP430F1611 microcontroller [28]. By loading the number of bytes that must be received or transmitted, the I^2C hardware peripheral will further take care of acknowledging and setting the start and stop conditions at the right time. One big disadvantage of this is that the I2CNDAT register is only 8-bits wide so the transactions are limited to 255 bytes. On the Delfi-n3Xt satellite, no data transfers larger than 255 bytes take place, so the automatic data byte counting feature can be used, which is convenient and makes the implementation of the service layer software for the I^2C module less complex.

To configure the I^2C peripheral for a write or read operation, the I2CTRX bit in the I2CTCTL register (see Figure 5.12) must be set properly. Setting this bit to 0 represents a read operation and a 1 represents a write operation. Furthermore, to actually initiate the transfer, the I2CSTT and I2CSTP bits of the I2CTCTL register must be set high.

An example of a read transaction of 56 bytes from I^2C slave address 0x48 is initiated as follows:

```
I2CSA = 0x48;
I2CNDAT = 56;
I2CTCTL &= I2CTRX;
I2CTCTL |= I2CSTT | I2CSTP;
```

In order to read a byte from the I^2C bus or write a byte onto it, the OBC must wait until the bus is ready for it. The I2CIFG register can be used to poll flags that indicate whether or not a new data byte can be written to or read from the I^2C bus. The I2CIFG register is shown in Figure 5.14.

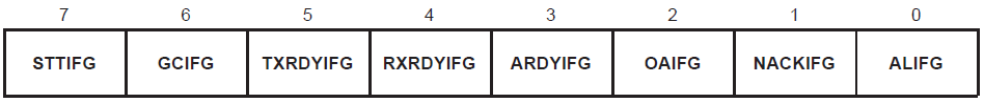


Figure 5.14: *The I2CIFG register that holds the interrupt status flags*

During a read operation, the RXRDYIFG bit in the I2CIFG register must be polled. This bit is set to 1 by the I^2C hardware peripheral when a data byte is just received. The data byte is stored in the I2CDRB register and the RXRDYIFG bit is cleared automatically when the received data bytes is read from the I2CDRB register. By polling the RXRDYIFG flag, the service layer software waits until a data byte is received by the I^2C hardware peripheral. The polling mechanism is demonstrated using the following code snippet:

```
Timer_i2c_timeout_start();

while((I2CIFG & RXRDYIFG) == 0) {

    if(timer_i2c_timeout) {
        I2C_bus_health_status = I2C_BUS_ERROR_TIMEOUT;
        break;

    }

}

data[i] = I2CDRB;
```

For a write operation, the same mechanisms applies. The only difference is that the TXRDYIFG flag is polled and that data is written to the I2CDRB register.

To be sure that the polling will not completely stall the OBC software, a timer is used that will set the *timer_i2c_timeout* flag high after a period of 30 ms after it is started. The health status of the I^2C bus is updated accordingly when a time-out has occured.

5.1.5.2 I²C Slave

Compared to the I²C master service layer software implementation, the implementation for the I²C slave is much less complex. As already mentioned in the I²C peripheral design section (see Section 4.3.5), I²C transfers from and to the slave device are handled using interrupts.

Configuration of the I²C peripheral in slave mode is almost the same as that for the I²C master. There are a few minor differences and they are described in this section. First of all, to let the MSP430F1611 I²C peripheral operate in slave mode, the MST bit in the U0CTL register must be 0. Second, the I²C slave implementation does not need to configure the I²C bus speed including the prescaler and the clock high and low period. This is something that is done by the master since the master always controls the I²C bus. Besides these two differences, two other registers must be configured in order to enable interrupts and set the I²C slave address. The first one discussed is the 16-bits wide I2COA register and it is shown in Figure 5.15.

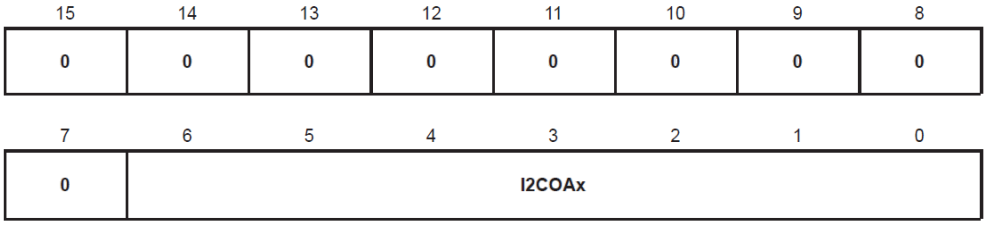


Figure 5.15: *The I2COA register that holds the 7-bit slave address*

The I2COA register is 16-bits wide because the MSP430F1611 supports 10-bits I²C addresses. For Delfi-n3Xt, 10-bit I²C addresses are not needed because an address space of 7-bits wide is enough to address all the I²C devices that are present on the Delfi-n3Xt I²C bus. Because the I²C peripheral is in 7-bits address mode, bits 15 to 7 of the register can not be configured and they all have a fixed value of 0. The 7-bit slave address needs to be loaded in the 7 least significant bits of the register.

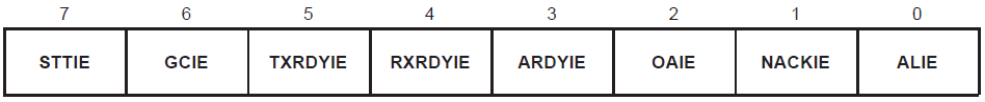


Figure 5.16: *The I2CIE register that enables or disables I²C interrupts*

The second register that is used is named I2CIE and using that 8-bits wide register the different I²C interrupts can be enabled or disabled. The MSP430F1611 I²C interrupt vector contains 8 different I²C interrupts. The I2CIE register is visualized in Figure 5.16. The 8 interrupt enable/disable bit fields represent the following:

- STTIE Start condition detected interrupt
- GCIE General call address interrupt
- TXRDYIE Transmitter ready interrupt

- RXRDYIE Receiver ready interrupt
- ARDYIE Register access read interrupt
- OAIE Own address detected interrupt
- NACKIE No acknowledgement received interrupt
- ALIE Arbitration lost interrupt

For the Delfi-n3Xt OBC in slave mode, the STTIE, GCIE, TXRDYIE and RXRDYIE interrupts are used so those corresponding bit fields must be set to 1 and the others to 0. This is done by loading the 8-bit value 0xF0 in the I2CIE register. The GCIE interrupt is used to detect the I^2C synchronization command that is send by the OBC that controls the I^2C bus, so it can be used to determine whether or not the I^2C master is still working properly. The STTIE interrupt will interrupt the CPU when a transaction to the slave is initiated by the master and it is used to notice the slave device for an upcoming transaction. The TXRDYIE interrupt happens when the transmitter is ready to send a byte to the master and the RXRDYIE interrupt will happen when a data byte is received from the master. A received byte must be fetched from the I2CDRB register. A byte that must be transmitted has to be loaded into the I2CDRB register.

5.1.6 Watchdog

The watchdog timer is used to reset the OBC immediately (through a telecommand) and to reset the OBC after 8 seconds if it is trapped in an undefined state. The functionality for resetting the OBC directly is needed to reset the OBC by a telecommand (see Section 4.4.4.4). The watchdog peripheral is configured through one 16-bit register that is given in Figure 5.17.

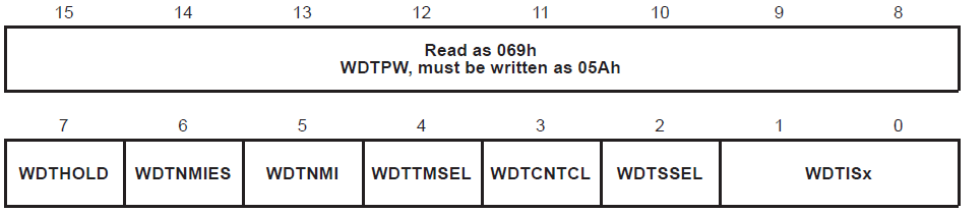


Figure 5.17: *The WDTCTL register that is used to configure the watchdog timer*

Like the 16-bit flash memory registers, there is a security byte present in the watchdog peripheral register as well (the higher byte of the 16-bit register). When the register is read, the higher byte will always have a value of 0x69. Special care must be taken while writing to the watchdog peripheral register. When a value is written to the register, the upper byte must have a value of 0x5A, so one should always write a value of 0x5AXX to the 16-bit register. When a different upper byte value than 0x5A is written to the register, the watchdog peripheral will reset the OBC its microcontroller. This property of the MSP430F1611 microcontroller is exploited in order to fulfil the requirement for a direct OBC reset through a telecommand.

In order to make full use of the watchdog peripheral, functions for the following functionalities were created:

- Configure the watchdog peripheral
- Clear the internal watchdog counter
- Reset the OBC

Section 5.1.6.1 shows the implementation for the configuration of the 16-bit watchdog peripheral register. How the internal counter of the watchdog peripheral can be cleared is shown in Section 5.1.6.2. In Section 5.1.6.3 it is shown how the OBC can be reset by writing a specific value to the 16-bit watchdog peripheral register.

5.1.6.1 Watchdog peripheral configuration

Configuration of the watchdog peripheral is done by configuring the WDTSSEL (bits 2) and WDTISx (bits 1 and 0) bits of the WDTCTL register. As already explained in Section 4.3.6, the watchdog peripheral must have ACLK as clock source input and it must use an input clock divider of 32768 in order to achieve an expiration time of 8 seconds.

From the MSP430F1611 datasheet [28], it is clear that the WDTSSEL bit must be high in order the select ACLK as clock source input. The two WDTISx bits select the watchdog timer interval and they must both be set to 0 in order to select an input clock divider of 32768. This translates into the following two C statements that configure the watchdog peripheral:

- WDTCTL |= 0x5A04;
- WDTCTL &= 0x5AFC;

The first statement sets the WDTSSEL bit high such that the watchdog peripheral will use ACLK as clock source input. The second statement sets the WDTISx bits to 0 in order to configure the desired input clock divider.

5.1.6.2 Clearing the internal watchdog peripheral counter

The internal counter of the watchdog peripheral can easily be cleared by setting the WDTCNTCL bit (bit 3 of the lower byte of the register) of the watchdog peripheral register high. While the WDTCNTCL bit is set high, it must not be forgotten to write the proper security value of 0x5A to the higher byte of the register at the same time. Furthermore, the other bits in the lower byte of the register should not be affected since they configure the watchdog peripheral. Clearing the internal watchdog peripheral counter can be accomplished by an OR operation on the WDTCTL register as follows: WDTCTL |= 0x5A08. This operation assures that the security byte is set to the right value and that the WDTCNTCL bit is set high while the other fields in the register remain untouched.

5.1.6.3 Reset the OBC

An OBC reset is as simple as violating the security policy of the WDTCTL register. The reset can be performed by writing any value that is not equal to 0x5A to the higher byte of the WDTCTL register value. In the OBC software the content of the WDTCTL register is just set to 0, after which an OBC reset will be performed immediately.

5.2 Application Layer Software

In this section, only the most important implementations of the application layer software are discussed. This section about the application layer software implementation describes how the various modes make use of the before described service layer software to fulfil various requirements. Section 5.2.1 gives the implementation description about the boot mode. The delay mode that may be executed before solar panel and antenna deployment is described in Section 5.2.2 and the deployment mode is described in Section 5.2.3. Most of the time, the OBC will be in the main mode. The implementation of the main mode is discussed in Section 5.2.4. When the I^2C bus got stuck, the I^2C recovery mode is executed. The implementation details of the I^2C recovery mode are given in Section 5.2.5.

5.2.1 Boot Mode

When the OBC becomes active for the first time because power becomes available or because of an OBC reset, the boot mode is entered first. In Section 5.2.1.1 is shown with which service layer function calls the hardware peripherals are initialized and configured. The volatile variables will be initialized to their hard-coded default values. The volatile variables initialization is not further described in detail here. The implementation on incrementing the encoded boot counter is given in Section 5.2.1.2. Furthermore, in Section 5.2.1.3, it is shown how subsystems are configured during boot time. Finally, in Section 5.2.1.4, an implementation description on making a transition from the boot mode to another mode is given.

5.2.1.1 Hardware Peripheral Configurations

In the application layer software, configuring the hardware peripherals is not a difficult task because it is already provided by the service layer software described in the previous Section. The required service layer modules are the clock source module, watchdog module, ETC module, ADC module, flash memory module, I^2C module, timers module and interrupts module. The ETC module provides routines that drive the elapsed time counter. It is designed and implemented by another engineer so the detailed description about it is not part of this thesis. Furthermore, global interrupts must be enabled since some of the hardware peripherals (the timers and the ADC) are interrupt driven.

The following C code snippet shows the necessary service layer function calls in the proper order:

```
Clocks_configure();
Watchdog_configure();
ETC_configure();
ADC_configure();
Flash_configure();
I2C_configure_master();
Timer_main_loop_configure();
Timer_i2c_timeout_configure();
Interrupts_enable();
```

Obviously, the order of execution is important. Configuration of the clock system and watchdog timer stabilize the system and prevent the OBC from rebooting continuously so these service layer function calls must be executed first. The *Timer_main_loop_configure()* service layer function call configures Timer B and the *Timer_i2c_timeout_configure()* function call configures Timer A.

5.2.1.2 Boot Counter

During the boot mode, the 33-byte encoded boot counter must be read from flash memory, incremented by 1 and written back to flash memory. In this section it is described how the encoded boot counter is implemented. The sequence starts with reading the boot counter from flash memory and decode it into a 2-byte wide unsigned integer. The following C code snippet is responsible for doing this:

```
unsigned char i, j;
unsigned int obc_boot_counter = 0;
unsigned char obc_encoded_boot_counter[33];

flash_read(BOOT_COUNTER_ADDRESS, 33, obc_encoded_boot_counter);


for(i = 0; i < 32; i++) {
    for(j = 0; j < 8; j++) {
        obc_boot_counter += ((obc_encoded_boot_counter[i]&(1<<j))==0);
    }
}
obc_boot_counter += ((~obc_encoded_boot_counter[32])<<8);
```

First the 33-byte encoded boot counter is read from flash memory by calling the flash memory service layer function named *flash_read()* that reads data from flash memory. After doing this, all the bits of the first 32 bytes of the encoded boot counter are scanned in order to count how many bits are set to 0. The number of multiples of 256 are stored in byte 33 of the encoded boot counter and as already explained during the design in Section 4.4.1.1, the content of the byte is inverted and must be multiplied by 256.

In order to increment and write the boot counter back to flash memory, it was easiest to increment the encoded boot counter directly by changing the very first bit that is set to 1 in the first 32 bytes of the encoded boot counter to 0. When all the 256 bits in the first 32 bytes of the encoded boot counter are already set to 0, the flash segment in which the encoded boot counter is stored must be erased and byte 33 must be decremented by 1 (since the content of that byte is inverted). After doing this, all the first 32 bytes must be set to 0xFF (all the bits set to 1) and the 33 byte encoded boot counter must be written to flash memory.

```
for(i = 0; i < 32; i += 1) {

    for(j = 8; j > 0; j -= 1) {
        if(obc_encoded_boot_counter[i]&(1<<(j-1))) {
            obc_encoded_boot_counter[i] &= ~(1<<(j-1));
            flash_write(BOOT_COUNTER_ADDRESS,33,obc_encoded_boot_counter);
            return;
        }
    }

}
```

The code snippet given above scans the first 32 bytes of the encoded boot counter for the first bit that is set to 1. When a bit with a logical value of 1 is found, the bit is set to 0 and the incremented encoded boot counter is written to flash memory. When no bit with logical value 1 is found in the first 32 bytes of the encoded boot counter, the code snippet shown below is executed.

```
for(i = 0; i < 32; i += 1) {

    obc_encoded_boot_counter[i] = 0xFF;

}

obc_encoded_boot_counter[32] -= 1;
flash_erase(BOOT_COUNTER_ADDRESS);
flash_write(BOOT_COUNTER_ADDRESS, 33, obc_encoded_boot_counter);
```

When no bit with a logical value of 1 is found in the first 32 bytes of the encoded boot counter, byte 33 of the encoded boot counter (index 32 in the encoded boot counter array) is decremented by 1. Besides that, all the bits in te first 32 bytes must be set to 1. The incremented encoded boot counter can not be successfully written to flash memory without a flash erase cycle (since one or more bits are changed from 0 to 1). Therefore, before the encoded boot counter is written to flash memory, the flash memory segment in which the encoded boot counter value is stored must be erased first by the *flash_erase()* function call to the flash peripheral service layer module.

5.2.1.3 Subsystem Configuration

In Section 4.4.1.2 it is already explained that the EPS, CDHS, PTRX, STX, MechS and SDM subsystems are turned on at boot time. The MechS consists of a redundant deployment board (DAB1 and DAB2) and the primary deployment board (DAB1) must be turned on during boot time. At boot time, the STX is not allowed to transmit data. Furthermore the PTRX must be configured such that it will operate at a data rate of 2400 bits/s and sends out flags through the transmitter continuously [30]. This is performed by the following application layer function calls:

```
EPS1_on();
DAB1_on();
SDM_on();
STX_on();
PTRX_on();
STX_tx_off();

PTRX_set_to_callsign();
PTRX_set_from_callsign();
PTRX_set_transmitter_bitrate(PTRX_BITRATE_2400);
PTRX_set_transmitter_idle_state(PTRX_TRANSMITTER_IDLE_ON);
PTRX_set_transmitter_output_mode(PTRX_TRANSMITTER_NOMINAL);
```

The application layer functions shown in the code snippet above will intern use the I^2C service layer software to write data to the DSSB microcontrollers of the EPS1, DAB1, SDM, STX, PTRX and STX to turn on those subsystems. The configuration of the PTRX is done by sending data bytes over the I^2C bus to the PTRX its ITC and IMC microcontrollers [30].

5.2.1.4 Mode Switching

After executing the actual boot mode blocks, the OBC must switch from the boot mode to one of the other modes. It can switch to the test mode, delay mode, deployment mode or main mode. Since it was decided that no test mode had to be implemented because of time constraints, the test mode will actually never get entered since it will never be present on the I^2C bus. The following C code snippet shows the switching logic that decides to which mode must be switched.

```
if(OBC_test_interface_present()) {

    OBC_set_operational_mode(OPERATIONAL_MODE_TEST);

}
```

```
else if(OBC_deployment_delay_needed()) {
    OBC_set_operational_mode(OPERATIONAL_MODE_DELAY);
}
else if(OBC_deployment_needed()) {
    OBC_set_operational_mode(OPERATIONAL_MODE_DEPLOYMENT);
}
else {
    OBC_set_operational_mode(OPERATIONAL_MODE_MAIN);
}
```

As already explained, the test mode will never get entered so the *OBC_test_interface_present()* function is just a stub that always returns *false*. The *OBC_deployment_delay_needed()* function determines whether or not a delay for deployment is needed. It does this by reading the deployment delay counter from flash memory using the flash memory read data service layer function call described in Section 5.1.3.2. The delay mode will be entered when the deployment delay counter is larger than 0, so the C code snippet that performs this check is as easy as the following:

```
return (deployment_delay_counter > 0);
```

For entering the deployment mode, something similar takes place. The 16-bit vector in flash memory that holds one bit for each deployment device that indicated whether or not deployment has been attempted for the device can be used to check whether or not deployment is needed. When deployment is not needed, all the deployment device are already attempted to deploy so the 16-bit vector contains all ones. Hence, the following C code snippet implements the *OBC_deployment_needed()* function and determines whether or not deployment is needed:

```
return (deployment_attempted_vector != 0xFF);
```

Finally, when none of the above transitions is needed, the main mode will be entered. As can be seen, mode switching is done by setting a global variable that holds the current operational mode of the satellite. Depending on the content of this global variable, the content of the modes is executed or not.

5.2.2 Delay Mode

Initially, before Delfi-n3Xt is being launched, the delay counter in flash memory will be set to 600 seconds (10 minutes). When Delfi-n3Xt is just launched and coming out of the POD, the delay counter is read from flash memory and it is being used to create a delay before the satellite its operational mode is switched to the deployment mode. For

counting and decrementing the delay counter, the main loop can be used such that the delay counter can be decremented by 2 every iteration of the main loop.

To be sure that the OBC will not get stuck for a long period of time in the delay mode because of OBC resets that may occur within 10 minutes, every minute the delay counter is written to flash memory. The modulo operator is used to check whether or not a minute has been elapsed:

```
if((delay_counter % 60) == 0) {

    flash_erase(DELAY_COUNTER_ADDRESS);
    flash_write(DELAY_COUNTER_ADDRESS, 2, delay_counter);

}
```

Suppose the OBC is being reset after 2 minutes and 30 seconds in delay mode. The delay counter in flash memory will then have a value of 480. The delay mode will again be entered after the boot mode, but the delay mode will now last for 8 minutes. When the OBC continuously resets within one minute after boot, the delay counter can be set to 0 by a telecommand after a while to ensure that the delay mode will not be entered anymore and the OBC can go on with the deployment mode.

```
if(delay_counter == 0) {

    OBC_set_operational_mode(OPERATIONAL_MODE_DEPLOYMENT);

}
```

Once the delay counter reaches the value of 0, a transition will be made from the delay mode to the deployment mode.

5.2.3 Deployment Mode

In the deployment mode, the OBC will send the appropriate commands to the DAB1 and DAB2 subsystems such that they will deploy the antennas and/or solar panels by burning the primary and/or secondary resistors. As explained in Section 4.4.3.1, there are 4 antennas and 4 solar panels that may be deployed and each one of them must be deployed for 16 seconds. Only one deployment device is allowed to be deployed at a time so the OBC must take this into account while commanding the DAB1 and DAB2 to deploy an antenna or solar panel.

Since the main loop has an interval of 2 seconds and a deployment device must have a deployment time of 16 seconds, the main loop can be used as a timing reference in order to command the DAB1 and DAB2 to deploy an antenna or solar panel. After 8 main loop executions, the OBC can go on with commanding the DAB1 or DAB2 to deploy the next deployment device.

The OBC holds a 16-bit vector that holds information about whether or not deployment has been attempted for a device using the primary and secondary resistor.

The burning of the primary resistors are done by the DAB1 and the burning of the secondary resistors by the DAB2. The vector has the following outline:

b_{15}	deploy antenna Y- with secondary resistor
b_{14}	deploy antenna Y+ with secondary resistor
b_{13}	deploy antenna X- with secondary resistor
b_{12}	deploy antenna X+ with secondary resistor
b_{11}	deploy solar panel Y- with secondary resistor
b_{10}	deploy solar panel Y+ with secondary resistor
b_9	deploy solar panel X- with secondary resistor
b_8	deploy solar panel X+ with secondary resistor
b_7	deploy antenna Y- with primary resistor
b_6	deploy antenna Y+ with primary resistor
b_5	deploy antenna X- with primary resistor
b_4	deploy antenna X+ with primary resistor
b_3	deploy solar panel Y- with primary resistor
b_2	deploy solar panel Y+ with primary resistor
b_1	deploy solar panel X- with primary resistor
b_0	deploy solar panel X+ with primary resistor

where b_{15} to b_0 represents bit 15 to bit 0 of the 16-bit vector. The deployment mode will walk through the whole vector from bit 0 to bit 15, so the deployment mode will first check whether or not deployment has already been attempted before for the solar panel on the X+ side of the satellite using the primary resistor. A high bit in the 16-bit vector means that deployment has already been attempted for the corresponding deployment device, and a low bit means that deployment was not attempted yet. Below, a code snippet is shown that checks whether or not a deployment device should be deployed.

```
if(OBC_deployment_current_device_needed()) {

    OBC_start_deploy_current_device();

}
```

The *OBC_deployment_current_device_needed()* function does a check on the 16-bit vector with a mask that corresponds to the current device that must be checked for deployment. For instance, the mask for deployment of solar panel X+ using the primary resistor is set to 0x01, solar panel X- with primary resistor to 0x02, solar panel Y+ with primary resistor to 0x04 and so on until the mask for deployment of antenna Y- with the secondary resistor which has a value of 0x80. The check can now be performed by a simple bitwise AND operation between the 16-bit vector and the mask of the deployment device. If the outcome of the check equals 0 it means that no deployment is attempted before, so the OBC will send the appropriate command to the DAB1 or DAB2 such that the deployment device will get deployed by burning the primary or secondary resistor (the *OBC_start_deploy_current_device()* function handles this). After sending the command to DAB1 or DAB2, the corresponding bit in the 16-bits wide vector will be set high and the vector is written to flash memory in order to save the settings permanently.

5.2.4 Main Mode

In this Section, various parts of the main mode are briefly discussed in terms of implementation. In Section 5.2.4.1, the implementation of the main loop is discussed. The implementation of OBC specific data that is put in the telemetry frame is discussed in Section 5.2.4.2 and Section 5.2.4.3 describes the implementation of the data acquisition. Finally, in Section 5.2.4.4, the implementation details of the telecommand execution is given.

5.2.4.1 Main Loop

Basically, the main loop is the mechanism that provides timing and execution of the functional blocks shown in Figure 4.29. It is an iterative function that is always executed until a mode switch is performed from the main mode to a different mode. The code snippet given below implements the content of the main loop. Mode switching may be caused by telecommands (switch to delay mode or directly to deployment mode) or failures on the I^2C bus (switch to the I^2C recovery mode).

```
OBC_wait_for_main_loop_execution();
OBC_send_i2c_sync_command();
OBC_clear_watchdog_timer();
OBC_check_telecommand();
OBC_check_initial_conditions();
OBC_clear_telemetry_data();
OBC_produce_frame_tag_data();
OBC_acquire_first_telemetry_frame();
OBC_send_first_telemetry_frame();
OBC_acquire_second_telemetry_frame();
OBC_send_second_telemetry_frame();
OBC_check_bus_status();
```

The function names given in the code snippet already describe which functional blocks they implement. Normally, the code will always be executed within 2 seconds and will wait a significant amount of time until the 2 seconds have been elapsed. Another wait condition can be found in the *OBC_acquire_second_telemetry_frame()* function, since the telemetry data that belongs to the second telemetry frame must be fetched in the second second of the main loop.

The *OBC_clear_watchdog_timer()* is an important function since it clears the internal counter of the watchdog timer. When this is not done within 8 seconds since the previous watchdog timer clear, the OBC will reset itself. If the OBC got stuck somewhere else in the code, the *OBC_clear_watchdog_timer()* function will not get executed which results in an OBC reset that probably solves the problem. Furthermore, the *OBC_check_telecommand()* function checks whether or not a telecommand is present in one of the radios and fetches and executes the telecommand if it is present. More about the implementation of this function is given in Section 5.2.4.4. Another important detail is the clearing of all the telemetry data that was acquired during the previous loop

execution. When a time-out occurs on the I^2C bus during data acquisition for a specific subsystem, no telemetry data is overwritten so the telemetry data of the previous loop remains in the telemetry data vector of that specific subsystem. To be sure no telemtry data of the previous loop is present in the subsystem telemetry data vectors, all the vectors are cleared such that they are filled with all zeros. Finally, the *OBC_check_bus_status()* function checks whether or not a failure on the bus happend such that communication is not possible anymore (i.e. the SCL line or SDA line is pulled low). If this is the case the function will let the OBC switch to the I^2C recovery mode.

5.2.4.2 OBC Telemetry Data

In Section 4.4.4.2 it is shown how the OBC telemetry data is obtained. As already given in the design section on the OBC telemetry data, the OBC telemetry data consists of the OBC frame tag data and the OBC housekeeping data. The 10 bytes wide OBC frame tag data vector is produced by shifting bytes into it, as shown in the code snippet below.

```
unsigned long ETC_time = read_ETC_time();

/* OBC tag for the first telemetry frame */
obc_tag_frame1[0] = (ETC_time >> 24);
obc_tag_frame1[1] = (ETC_time >> 16);
obc_tag_frame1[2] = (ETC_time >> 8);
obc_tag_frame1[3] = (ETC_time >> 0);
obc_tag_frame1[4] = (obc_boot_counter >> 8);
obc_tag_frame1[5] = (obc_boot_counter >> 0);
obc_tag_frame1[6] = (obc_frame_counter >> 24);
obc_tag_frame1[7] = (obc_frame_counter >> 16);
obc_tag_frame1[8] = (obc_frame_counter >> 8);
obc_tag_frame1[9] = (obc_frame_counter >> 0) & 0xFE;
obc_tag_frame1[9] |= OBC_TM_FRAME1_ID;

/* OBC tag for the second telemetry frame */
obc_tag_frame2[0] = (ETC_time >> 24);
obc_tag_frame2[1] = (ETC_time >> 16);
obc_tag_frame2[2] = (ETC_time >> 8);
obc_tag_frame2[3] = (ETC_time >> 0);
obc_tag_frame2[4] = (obc_boot_counter >> 8);
obc_tag_frame2[5] = (obc_boot_counter >> 0);
obc_tag_frame2[6] = (obc_frame_counter >> 24);
obc_tag_frame2[7] = (obc_frame_counter >> 16);
obc_tag_frame2[8] = (obc_frame_counter >> 8);
obc_tag_frame2[9] = (obc_frame_counter >> 0) & 0xFE;
obc_tag_frame2[9] |= OBC_TM_FRAME2_ID;

obc_frame_counter += 2;
```

First of all, the 32-bits wide elapsed time counter is requested by calling the *read_ETC_time()* function of the ETC module service layer software. The first 4 bytes of the OBC frame tag data vector are used to store the elapsed time counter. This is done by shifting the elapsed time counter bytes into the vector. The first byte in the vector must contain the most significant byte of the elapsed time counter and the fourth byte in the vector the least significant byte. The most significant byte is obtained by shifting out the first 3 bytes of the elapsed time counter value such that the most significant byte is alligned on the least significant byte position of the elapsed time counter variable (ETC_time). The remaining bytes are shifted as well such that they are properly alligned on the least significant byte position and can be easily inserted into the vector. Something similar happens with the OBC boot counter and frame counter contents, except that the OBC boot counter is only 2 bytes wide.

Besides the OBC frame tag data, the OBC will have to produce housekeeping data. The housekeeping data contains OBC specific data, whereas the OBC frame tag data contains more general data that applies to the health of the complete satellite. The following part describes how the OBC housekeeping data is produced.

The 8-bit stuck bus counter is incremented when the OBC gets no acknowledgement after sending out and I^2C synchronization command. This incrementing is performed when a transition is made to the I^2C recovery mode. The 8-bit short circuit counter and 8-bit last short circuit perpetrator are actually still open action items for implementation. At the moment of writing it was not clear yet how these features can be implemented.

The idea of the 11-bit subsystem mode settings vector was that it holds a bit for each subsystem such that the vector provides information about whether or not a subsystem should be powered on. Later on during the project it was decided that this is shifted to the initial condition checks, which is not part of this thesis.

The OBC temperature is simply obtained by performing a function call to the ADC service layer module that starts the A/D conversion and returns the 8 most significant bits of the read-out. The content that represents the first 4 bytes of the last executed telecommand is evident and is easily obtained by extracting the first 4 bytes from the volatile variable that holds the last executed telecommand.

An interesting variable is the 31-bit communication status vector that indicates whether or not communication is possible with a specific I^2C device (this may be a microcontroller are a specific device like a temperature sensor with an I^2C interface). Every I^2C device on the bus corresponds with a bit position in the vector. The bit position in the vector can be set to either 1 or 0, depending whether communication with the I^2C device is possible or not.

Finally, the 1 bit that provides information about with which receiver (PTRX or ITRX) the last telecommand was received is produced during the fetching of telecommands. The PTRX will always be checked first, followed by the ITRX. If a telecommand is present on the PTRX, the ITRX will not be checked anymore for a telecommand. If a telecommand is not present on the PTRX, the ITRX will be checked for the presence of a pending telecommand. Therefore the PTRX always has higher priority. When both receivers have a telecommand pending, the telecommand present on the PTRX will be fetched and executed during the current main loop execution and the telecom-

mand pending on the ITRX will be ignored (maybe in the next main loop execution the pending telecommand on the ITRX will be fetched and executed if the PTRX has no telecommand pending during the execution of that main loop).

5.2.4.3 Data Acquisition

The data acquisition is actually nothing more than payload data and housekeeping data collection by the OBC. After the I^2C synchronization command is sent, the OBC will wait a while until it will perform I^2C read operations with all the subsystems. On the read request, the requested subsystem will send the payload data or housekeeping data bytes to the OBC. The amount of bytes is known a priori so the OBC knows how many bytes it must read from the various subsystems.

In Appendix A, the directory listing of the OBC source code is given. On the application layer software level, source files are present for every subsystem in the *subsystems* directory. The source files for the subsystems consist of wrapper functions that make use of function calls to the I^2C service layer software in order to let the OBC communicate with the other subsystems. After reception of the data from a subsystem, the data is shifted into one large telemetry vector that is later on send to Earth through the active transmitter.

5.2.4.4 Telecommand Execution

Regarding telecommanding, first the PTRX is checked for the presence of a telecommand and if no telecommand is present the ITRX will be checked. Whether or not a telecommand is present can be checked by sending a specific command byte over the I^2C bus to the radio. The radio will receive and interpret the byte, after which it sends back the number of telecommands in the buffer. If the returned number is greater than 0, the OBC can fetch the telecommand and send a command to the radio to remove the telecommand that was just fetched in order to free up its buffer. Below, a code snippet is shown that fetches a telecommand from the PTRX or ITRX.

```
unsigned char n;
unsigned char telecommand[MAX_TELECOMMAND_LENGTH];
unsigned char telecommand_decrypted[MAX_TELECOMMAND_LENGTH];

if(PTRX_get_number_of_telecommands() > 0) {

    n = PTRX_get_telecommand(telecommand);
    PTRX_remove_telecommand();

    OBC_decrypt_telecommand(telecommand,telecommand_decrypted,n,0);
    OBC_process_telecommand(telecommand_decrypted,n);

    obc_last_receiver_id = ID_PTRX;

}
```

```
else if(ITRX_get_number_of_telecommands() > 0) {

    n = ITRX_get_telecommand(telecommand);
    ITRX_remove_telecommand();

    OBC_decrypt_telecommand(telecommand,telecommand_decrypted,n,0);
    OBC_process_telecommand(telecommand_decrypted,n);

    obc_last_receiver_id = ID_ITRX;

}
```

Once a telecommand is fetched, the telecommand must be decrypted since it is being uploaded in an encrypted form from the ground station to the satellite. The result from the *OBC_decrypt_telecommand()* function is the decrytped telecommand that is ready to be processed. The processing is done by the *OBC_process_telecommand()* function. During the processing of the telecommand, the telecommand is actually being executed. The 5 different types of telecommands (see Figure 4.30) can be distinguished from one another by inspecting the first byte of the telecommand. Depending on the value of the first byte, the telecommand is executed in a certain way. The value of the first byte can efficiently checked with the *switch()* mechanism.

5.2.5 I^2C Recovery Mode

A loop structure is used to implement the I^2C recovery mode, since the I^2C recovery mode has to send I^2C synchronization commands continuously until the I^2C bus becomes healthy (i.e. none of the I^2C lines are in a low state). Inside the continuous loop, the OBC will execute an I^2C service layer procedure that sends out the I^2C synchronization command and it will check the I^2C bus health status flag in order to check whether or not a transition back to the main mode is necessary. The C code snippet from the OBC source code looks as follows:

```
do {

    OBC_send_i2c_sync_command();

} while(I2C_bus_health_status != I2C_BUS_HEALTH_OK);


OBC_set_operational_mode(OPERATIONAL_MODE_MAIN);
```

Once the bus status becomes healthy because of a free bus due to the I^2C recovery procedure given in Table 4.2, the do-while loop will be skipped and the operational mode of the OBC is set to main mode such that a transition takes place from I^2C recovery mode to main mode.

5.3 Summary

The implementation of the OBC service layer software and application layer software was discussed in this chapter. For the service layer software, implementation details were given on the level of MSP430F1611 register configurations and for the application layer software a more detailed description was given about logical decision making on the higher software level.

The first section of the chapter gives descriptions about the service layer software implementation. This includes descriptions about the clock source module, programmable interval timers, flash memory controller, analog to digital converter, I^2C controller and the watchdog timer. The Sections show how the appropriate registers must be configured in order to let the MSP430F1611 microcontroller chip execute actions that are needed to fulfil several requirements. The clock source module implementation shows how the internal clock structure of the MSP430F1611 is configured such that the external 32768Hz and 8MHz crystals are used properly. The programmable interval timers are implemented in such a way that they produce delays that are used by application layer software as a timing reference for exact timing. The implementation that drives the flash memory controller describes how the registers of that peripheral need to be configured in order to configure the flash timing generator and initiate a flash read, write or erase operation. The section shows as well that driving the I^2C peripheral involves a lot of registers. Compared to the I^2C module, the A/D converter module and the watchdog timer module are much less complicated.

In the second section, the application layer software is discussed. It is about the implementation of the various modes in which Delfi-n3Xt may operate. In the section about the boot mode, it is explained how the hardware peripherals are configured by service layer function calls, how the encoded boot counter is implemented, how subsystems are configured at boot time and how the mode switching logic is implemented at the end of the boot mode. The implementation details of the delay mode and deployment mode are given as well in this section. The sections about these 2 modes show how a delay is introduced before the deployment mode, and how the OBC decides whether or not an antenna or solar panel must be deployed by burning the primary or secondary resistor on the DAB. The section about the main mode implementation gives the implementation details about the main loop, OBC telemetry data creation, data acquisition procedure and telecommand execution. Finally, the section gives implementation details about the execution of the I^2C recovery mode.

6 Conclusions

In this thesis, the design and implementation of the higher and lower level software for the Delfi-n3Xt nanosatellite OBC are described and discussed. The software development process for the Delfi-n3Xt OBC started with studying and analyzing the already existing Delfi-n3Xt requirements and configuration items list [19]. Besides that, new requirements were added to the existing list. During the design stage of the software development process, the architecture of the OBC software and the individual software blocks that implement the requirements were clearly defined. All the defined software blocks from the design stage were implemented and unit tested during the implementation stage and test stage respectively. The implementation stage iterated numerous times over the design stage, which resulted in a fine-tuned and optimized software design for this specific application. In the final stage of the OBC software development process, integration tests with the other Delfi-n3Xt subsystems were performed. This final stage verified the defined data flows and activity flows for the OBC and assessed the performance of the overall satellite in terms of power consumptions and data transfers over the I^2C bus. This final stage is not described in detail in this thesis. A brief description about it is given in Section 6.2.

The remaining part of this chapter presents a summary of this thesis in Section 6.1, a compiled list of my personal contributions to the Delfi-n3Xt OBC software design and implementation and the Delfi-n3Xt project in general (Section 6.2). Finally, in Section 6.3, a compiled list of future work that can be done in order to finalize and improve the Delfi-n3Xt OBC software is presented.

6.1 Summary

Chapter 2 describes all the background information that is needed to understand the core chapters of this thesis. It starts with a description about the Delfi-n3Xt mission, including its objectives and advancements compared to the nanosatellite its predecessor: the Delfi-C^3. Besides a general description about the Delfi-n3Xt mission, a brief introduction about the Delfi-n3Xt payloads and subsystems is given. There are two payloads present: the micro propulsion payload and the tranceiver payload. Besides these payloads, there are numerous subsystems that are needed for proper functioning of the satellite. These subsystems include the communications subsystem, attitude determination and control subsystem, electrical power subsystem, structures, mechanisms and thermal control subsystems and the command and data handling subsystem. The Delfi-n3Xt OBC is part of the latter mentioned subsystem. Furthermore, the chapter gives a detailed description about I^2C bus communication basics.

An I^2C bus failure analysis and performance analysis is given in Chapter 3. The I^2C bus failure analysis describes the most likely failure scenarios that may occur on the I^2C bus and may influence the I^2C bus health. The failure analysis includes failures that may occur during an I^2C start condition, slave request, read/write bit assertion, slave acknowledgement, data byte transfer and stop condition. Besides these specific failures, there are failures that may occur at any time during I^2C communication, such as a slave device that misses clock pulses and a device that pulls low the I^2C clock line or data line for a significant amount of time. The I^2C bus performance analysis shows results about the bit error rate that is measured in a representative setup for the Delfi-n3Xt nanosatellite. Furthermore, a description is given on how the bit error rate measurement is assessed and verified. The measured bit error rate turned out to be at most $4 \cdot 10^{-9}$, which comfortably meets the requirement of a bit error rate of at most 10^{-6}.

In Chapter 4, a detailed design description about the Delfi-n3Xt OBC software is given. This includes an analysis of the OBC software requirements, a detailed description about the software architecture of the OBC, and the design of service layer software and application layer software. The OBC software requirements are listed and rationalized in this chapter and they are split up into 5 different categories: subsystem communication requirements, fault-tolerant software requirements, telecommanding requirements, data acquisition requirements and monitoring requirements. Furthermore, the description about the OBC software architecture gives an overview of the different software layers that together form the complete set of OBC flight software. Besides this, it is shown how the different software layers are split up into individual modules. The OBC service layer software consists of the clock source module, programmable interval timers module, flash memory controller module, analog to digital converter module, I^2C controller module and the watchdog timer module. The application layer of the Delfi-n3Xt OBC software implements the 5 different modes of the OBC: the boot mode, delay mode, deployment mode, main mode and I^2C recovery mode. The design of these modes, including the possible transitions between the different modes, is given in this chapter as well.

The final chapter of this thesis gives details about the OBC software implementation (see Chapter 5). Like the chapter on the OBC software design, the chapter about the OBC software implementation is split up into a part about the service layer software and a part about the application layer software. The part about the service layer software describes how the hardware peripherals of the MSP430F1611 microcontroller are driven by configuring the registers of the microcontroller. This is a lower level implementation that is very close to the hardware. The implementation of the application layer software is a higher level implementation that makes use of service layer software function calls. The part about the application layer software gives a brief description about its implementation. The detailed C implementation of the complete OBC software can be examined by obtaining the OBC source code. The OBC source code is part of this thesis as well.

6.2 Contributions

In this section, my personal main contributions to the Delfi-n3Xt project and this thesis are described and discussed. Besides these main contributions, many side activities like general project meetings and subsystem specific meetings took place that contributed to the project as well. During these side activities, interesting discussions with other engineers from different disciplines (e.g. electrical engineering, aerospace engineering) resulted in a continuous and steep learning curve.

The following list describes my personal main contributions to the project:

- **OBC software requirements analysis.**
  The already existing list of OBC software requirements was extensively analyzed. This was needed in order to determine how long it would take to implement certain requirements such that an appropriate work package for this thesis could be defined. However, the main reason for analyzing the OBC software requirements was to derive a proper software design from the requirements. Analysis of the OBC software requirements was also performed in order to discover missing requirements that should be present in the requirements list. The result of this work package is a comprehensive list of requirements including their rationales. Without this list of requirments, no successful software design could have been established.

- **I^2C bus failure analysis.**
  The investigation of failures that may occur on the I^2C bus was also a major contribution to this thesis. This small research topic gave me the opportunity to get familiar with the I^2C bus protocol and learn more about the influences of the space environment on satellites and in particular on their data and power buses. The result of this failure analysis is an overview of possible failure cases, including their causes, impacts and resolve steps. Some of the failures do not harm much and can not be or do not need to be resolved. Other failures may have a larger impact and must be resolved in order to ensure that the satellite is fault-tolerant. With the results of this analysis in mind, an OBC software design could be made that is able to handle the high impact failure cases.

- **I^2C bus performance analysis.**
  Another major contribution of this thesis is the I^2C bus performance analysis that was performed on Delfi-n3Xt engineering model test boards. At the time of the performance evaluation, no real flight hardware was available yet. The performance analysis proved that the bit error rate on the I^2C bus is lower than the specified bit error rate value of 10^{-6}. Actually, with a representative performance test setup, the measured bit error rate turned out to be 2 bit errors out of the 5 billion bits that were transferred. In order to be sure that the bit error rate measurement software was working correctly, a test setup was created with a switch on the SDA or SCL line that can be used to short one of the I^2C bus lines to the ground. By shorting one of the I^2C bus lines to the ground, bit errors could be introduced intentionally such that the bit error rate measurement software could be verified.

- **OBC service layer software development**.
  The major part of this thesis was spent on the software development process of the OBC service layer software. This included software design, implementation and unit testing. The design stage of the OBC service layer software development consumed most of the time. During the design phase of the service layer software, all the needed peripherals of the MSP430F1611 microcontroller had to be studied extensively. In order to do this, the appropriate sections of the MSP430F1611 datasheet were carefully read many times. This was needed for the service layer implementation, since knowledge about how the registers of the microcontroller had to be configured was required. Unit testing of the individual service layer modules proved the functionalities of the individual modules.

- **OBC application layer software development**.
  The software development process for the OBC application layer software consisted of a design, implementation and unit test phase as well. In this thesis, a major part of the application software is covered. The covered parts are the implementation of the most important modes of the satellite and the implementation of the interfaces with other subsystems. Some parts of the application layer software are not covered in this thesis. These missing parts are designed and implemented by other engineers or they still need to be implemented. The work packages that are still open are listed in the next section.

- **Integration testing.**
  The final major contribution is on integration testing. Integration testing consisted of running different test cases that verify the functionality of the interfacing between the OBC and the other subsystems. At the moment of finalizing this thesis, not all subsystems were ready and available yet. Integration testing has been performed with the antenna and solar panel deployment subsystem, the S-band transmitter, the SDM subsystem and the Delfi standard system bus microcontrollers.

6.3 Future Work

Unfortunately, during this thesis, there was not enough time to implement the complete set of OBC software that is actually needed to successfully finalize the Delfi-n3Xt OBC software. Besides that, some improvements can be made for future missions. These future work packages are presented and described in this section.

- **Implementation of the test mode.**
  Due to time constraints, it is decided that the test mode will not be implemented for the Delfi-n3Xt nanosatellite. However, it may be implemented in Delfi-n3Xt its successor. The test mode is especially useful when the complete satellite is assembled and ready for launch and some final tests have to be performed. The final tests can then be performed by an external test device (that is not part of the satellite itself) through the test interface that connects the test device with the satellite. Using the test interface the OBC can be commanded to perform tests.

- **Implementation of the subsystem initialization checks.**
  Probably the most important open item for the Delfi-n3Xt OBC software is the subsystem initialization work package. The actual state of the subsystems on board of the satellite must be consistent with the state in which they should be. For example, when a subsystem is actually off while it is supposed to be turned on, the OBC must decide whether or not the subsystem should be turned on. The details about the initial condition check procedure are not further described in detail since it is not part of this thesis.

- **Implementation of CRC for non-volatile variables.**
  In order to provide a certain level of fault-tolerance in the storage of non-volatile parameters, a CRC algorithm should be implemented such that the content of the flash memory can be checked for errors. The CRC functionality should be part of the flash memory service layer module and it should notify whether or not a non-volatile parameter could be read from the flash memory correctly (i.e. the non-volatile parameter does not contain consistency errors). In the case of no consistency errors, the non-volatile parameter can be loaded and used by the application layer software. When there is a consistency error, the application layer software should load a default value which is hard-coded in the OBC software and known to be safe.

- **Improvements in the I^2C recovery procedure.**
  In order to improve and complete the I^2C recovery procedure, the DSSB microcontrollers should be equipped with external crystals such that they have the capability of exact timing. During the hardware design of the DSSB circuitry, a microcontroller was selected that can only run on its internal RC-type oscillator. Exact timing is necessary to implement detection of the I^2C slave device that is responsible for pulling low one of the I^2C lines for a longer period of time. For exact timing in space, an external crystal is needed mainly because of the occurrence of large temperature differences in the space environment. With the given defined reset procedure in Table 4.2 and exact timing capabilities, the OBC can reliably determine which microcontroller held one of the bus lines low. The microcontroller its I^2C address can then be send to Earth (part of the telemetry data) such that engineers can hold statistics and take action through telecommands if it happens more often. This improvement is definately something for a future mission, since it needs a modification in the hardware design of the DSSB circuitry it can not be applied to Delfi-n3Xt anymore.

Bibliography

[1] *MSP430x15x, MSP430x16x Microcontroller Notes*, 2006.

[2] L.S. Boersma, *Verification and Testing of the Delfi-n3Xt Communications Subsystem*, Master's thesis, Delft University of Technology, department of Space Systems Engineering, 3 2012, p. 158.

[3] J. Bouwmeester, *Delfi Space Homepage: http://www.delfispace.nl.*

[4] J. Bouwmeester, S. de Jong, and G.T. Aalbers, *Improved Command and Data Handling System for the Delfi-n3Xt Nanosatellite*, 59th International Astronautical Congress, Glasgow, Scotland, UK (2008), 9.

[5] N.E. Cornejo, *Fault Detection for the Delfi Nanosatellite Programme*, Master's thesis, Delft University of Technology, department of Computer Engineering, 7 2009, p. 116.

[6] N.E. Cornejo, J. Bouwmeester, and G.N. Gaydadjiev, *Implementation of a Reliable Data Bus for the Delfi Nanosatellite Programme*, 7^{th} IAA Symposium on Small Satellite for Earth Observation (2009), 9.

[7] N.E. Cornejo, L.S.Boersma, J. Bouwmeester, and P.M.C. Beckers, *CDHS Delfi Standard System Bus*, Tech. report, Delf University of Technology, department of Space Systems Engineering.

[8] Dallas Semiconductors, *Parallel-Interface Elapsed Time Counter.*

[9] T.E. de Groot, *Command and Data Handling Subsystem of Delfi-n3Xt*, Master's thesis, Delft University of Technology, department of Space Systems Engineering, 2 2011, p. 34.

[10] W.J. Ubbels et. al (ed.), *The Delfi-C3 Student Nanosatellite - an Educational Test-Bed for New Space Technology*, Delft University of Technology, department of Space Systems Engineering, 2006.

[11] S.Y. Go, J. Bouwmeester, and G.F. Brouwer, *Optimized Three-Unit Cubesat Structure for Delfi-n3Xt*, 59^{th} International Astronautical Congress (2008), 5.

[12] R.J. Hamann and S. de Jong, *Trade-off Procedure for Payload Selection in University Small Satellite Projects*, Tech. report, Delft University of Technology, department of Space Systems Engineering.

[13] T. Hamoen, *Delfi-n3Xt Basic Electronics and EMC Guidelines*, Tech. report, Delf University of Technology, department of Space Systems Engineering.

[14] T. Hoevenaars, E. Dekens, and W. Edeling, *ADCS Reaction Wheel System Design*, Tech. report, Delft University of Technology, department of Space Systems Engineering.

[15] J. Bouwmeester, *Delfi-n3Xt ISILaunch Interface Control Document*, Tech. report, Delft University of Technology, department of Space Systems Engineering, 2011.

[16] J. Bouwmeester,, *Delfi-n3Xt Power Budget*, Tech. report, Delf University of Technology, department of Space Systems Engineering, 2012.

[17] J. Bouwmeester, *Delfi-n3Xt Thermal Budget*, Tech. report, Delf University of Technology, department of Space Systems Engineering, 2012.

[18] J.Bouwmeester, *CDHS Software Interface Control Document*, Tech. report, Delf University of Technology, department of Space Systems Engineering, 2012.

[19] J.Bouwmeester, *Delfi-n3Xt Requirements and Configuration Items List*, Tech. report, Delft University of Technology, department of Space Systems Engineering, 2012.

[20] G.W. Lebbink, *ISIPOD Interface Specification*, Tech. report, ISIS - Innovative Solutions in Space B.V., 2010.

[21] C. Mller, L. Perez Lebbink, B. Zandbergen, G. Brouwer, R.Amini, D. Kajon, and B. Sanders, *Implementation of the $T^3\mu PS$ in the Delfi-n3Xt satellite*, 7^{th} IAA symposium on Small Satellites for Earth Observation (2009), 8.

[22] D.E. Nielsen, J. Taylor, and W.A. Beech, *AX.25 Link Access Protocol for Amateur Packet Radio*, Tech. report, Tucson Amateur Packet Radio Corporation.

[23] A. Noroozi, *OBC Microcontroller Selection*, Tech. report, 10 2008.

[24] NXP Semiconductors, *I^2C-bus specification and user manual*.

[25] J. Reijneveld, *Design of the Attitude Determination and Control Algorithms for the Delfi-n3Xt*, Master's thesis, Delft University of Technology, department of Space Systems Engineering, 1 2012, p. 190.

[26] F. Stelwagen, *A Global Electrical Power Supply for the μSatellite Delfi-n3Xt*, Tech. report, Systematic Design B.V.

[27] G. Swinerd, P. Fortescue, and J. Stark, *Spacecraft Systems Engineering*, Wiley, 2003.

[28] Texas Instruments, *MSP430x1xx Family User's Guide*, 2006.

[29] R. van den Eikhoff, *Design of a Universal Antenna Deployment System*, Master's thesis, Delft University of Technology, department of Space Systems Engineering, 10 2006, p. 84.

[30] W. Weggelaar, *TRXUV UHF-VHF Tranceiver User Manual*, ISIS - Innovative Solutions in Space B.V.

[31] B. Wolters, *Attitude Determination and Control Subsystem Magnetorquer Design*, Tech. report, 8 2009.

OBC Source Code Directory Listing

A

OBC

- main.c
- bool.h

service_layer

- adc.c
- adc.h
- clocks.c
- clocks.h
- etc.c
- etc.h
- flash.c
- flash.h
- i2c.c
- i2c.h
- interrupts.c
- interrupts.h
- timers.c
- timers.h
- watchdog.c
- watchdog.h

application_layer

- i2c_addresses.h
- flash_addresses.h

modes

- modes.h
- main_mode.c
- main_mode.h
- boot_mode.c
- boot_mode.h
- delay_mode.c
- delay_mode.h
- deployment_mode.c
- deployment_mode.h

subsystems
- obc.c
- obc.h
- dab.c
- dab.h
- eps.c
- eps.h
- stx.c
- stx.h
- sdm.c
- sdm.h
- tcs.c
- tcs.h
- adcs.c
- adcs.h
- dssb.c
- dssb.h
- ptrx.c
- ptrx.h
- itrx.c
- itrx.h
- t3ups.c
- t3ups.h

Curriculum Vitae

Alexander Franciscus Cornelis (Sander) van den Berg was born in Noordwijkerhout, The Netherlands on August 13th of 1985.
He received his Bachelor of Science degree in Computer Science and Engineering at Delft University of Technology in 2009. His Bachelor's graduation project was on designing and implementing an application that had to reduce the complexity of adding individual software modules to a project workspace. The application resulted in a system that was less error prone and significantly faster than the original procedure.

His growing interest in embedded systems, hardware designs and electronics motivated him to do a Master of Science degree in Computer Engineering at Delft University of Technology. He specialized himself in the field of (Real-Time) Embedded Systems. During his studies he gained knowledge and experiences in the field of software engineering and software architectures, parallel computing, computer architectures, embedded system architectures and fault-tolerant software implementations. Furthermore, he broadened his knowledge in the field of earth and planetary observation technology and space systems engineering. Sander has strong affinity with basically anything that has to do with space, electronics and computer hardware and software.

During his Master's thesis Sander worked on the On-Board Computer of the Delfi-n3Xt Nanosatellite. He was responsible for a major part of the design and implementation of the fault-tolerant On-Board Computer software. On this project he worked together with engineers from other disciplines. His contributions to the project fulfill most of the requirements of the On-Board Computer software.

Parallel to his Master's thesis work, Sander applied for a job as an (embedded) software engineer at ISIS - Innovative Solutions in Space. At the moment of writing he already works for 6 months for this company. At ISIS, Sander worked on the development of (embedded) ground station software for satellite tracking. Besides that, he is gaining even more experiences in the development of OBC flight software.

Besides his work and studies, Sander likes sports (fitness and running), travelling, spending time with his family, exploring other countries and other cultures, reading, programming, cars, astronomy, and his embedded systems hobby projects.