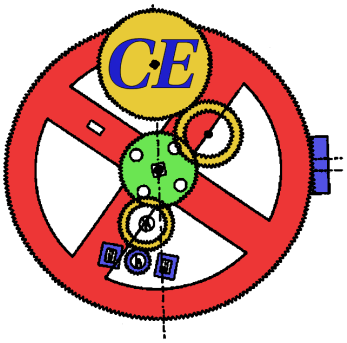


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2009

MSc THESIS

Fault Detection for the Delfi Nanosatellite Programme

Napoleón E. Cornejo B.

CE-MS-2009-12

Abstract

Delfi-C^3 was the first dutch student nanosatellite launched into space on April 28, 2008. After more than a year in operation, it is regarded as a successful mission. The experience, however, revealed noticeable problems with the Command & Data Handling Subsystem (CDHS) and particularly with the data bus, which apparently experiences random "hiccups" and halts. Therefore, this thesis will explore two applications of fault detection methods within the Delfi satellites, focused first on the CDHS and then on dynamic sensor systems. The first focus intends to address the problems in the CDHS, by proposing mechanisms that would increase the reliability of the systems on-board, particularly in their capability to autonomously detect errors. In that sense, this work begins with a study on the design of the Delfi-C^3 CDHS, with measurements on bit-error rates and software debugs that show particular flaws. We present, then, an overview of the next version of the satellite, **Delfi-n3Xt**, with the current architectural design of the CDHS and the data bus, which includes the proposed improvements over the previous design and the rationale behind them. We also describe fault torelant mechanisms for software and communications, more extensively error detection codes that may improve the reliability of the bus when properly implemented. As the capability of autonomous fault detection can be applied to other systems besides the CDHS, we investigate a model based mechanism, inspired on a recent variant of the popular Kalman Filter, the Unscented Kalman Filter, which is fundamented on probabilistic estimation techniques. We apply this method to the navigation sensors of Delfi-n3Xt and provide results that show it is possible to detect errors with this scheme.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Fault Detection for the Delfi Nanosatellite Programme

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Napoleón E. Cornejo B.
born in San Salvador, El Salvador

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

Fault Detection for the Delfi Nanosatellite Programme

by Napoleón E. Cornejo B.

Preface

Delfi-C^3 was the first dutch student nanosatellite launched into space on April 28, 2008. After more than a year in operation, it is regarded as a successful mission. The experience, however, revealed noticeable problems with the Command & Data Handling Subsystem (CDHS) and particularly with the data bus, which apparently experiences random "hiccups" and halts. Therefore, this thesis will explore two applications of fault detection methods within the Delfi satellites, focused first on the CDHS and then on dynamic sensor systems. The first focus intends to address the problems in the CDHS, by proposing mechanisms that would increase the reliability of the systems on-board, particularly in their capability to autonomously detect errors. In that sense, this work begins with a study on the design of the Delfi-C^3 CDHS, with measurements on bit-error rates and software debugs that show particular flaws. We present, then, an overview of the next version of the satellite, **Delfi-n3Xt**, with the current architectural design of the CDHS and the data bus, which includes the proposed improvements over the previous design and the rationale behind them. We also describe fault torelant mechanisms for software and communications, more extensively error detection codes that may improve the reliability of the bus when properly implemented. As the capability of autonomous fault detection can be applied to other systems besides the CDHS, we investigate a model based mechanism, inspired on a recent variant of the popular Kalman Filter, the Unscented Kalman Filter, which is fundamented on probabilistic estimation techniques. We apply this method to the navigation sensors of Delfi-n3Xt and provide results that show it is possible to detect errors with this scheme.

Portions of this work have been published in the *Proceedigns of the 7th IAA Symposium on Small Satellites for Earth Observation*, in Berlin, Germany 2009 (IAA-B7-0908P) and will also be submitted to the *2010 IEEE Aerospace Conference* to be held in Big Sky, Montana, U.S.A.

Laboratory	:	Computer Engineering
Codenumber	:	CE-MS-2009-12
Committee Members	:	
Advisor:		dr.ir. Georgi Gaydadjiev, CE, TU Delft
Chairperson:		dr.ir. Koen Bertels, CE, TU Delft
Member:		dr.ir. Chris Verhoeven, ERL, TU Delft
Member:		dr.ir. Stephan Wong, CE, TU Delft
Member:		ir. Rouzbeh Amini, SSE, TU Delft

Contents

List of Figures

List of Tables

Acknowledgements

The document contained here is the result of several months of work on the Delfi-n3Xt project and I am truly fortunate for having been part of this great team. However, it is only fair to express gratitude to those who in one way or another made it possible.

I would like to thank Georgi Gaydadjiev for his help; his advice was crucial in helping me decide which directions to take and what ideas to explore. Special thanks also to Rouzbeh Amini for his patience and guidance during this process, as he was always available to help me deal with the details of attitude kinematics, quaternions and control theory. To the members and staff of the Delfi-n3Xt team, who besides the hours of discussions about the satellite, always made time for having coffee, playing office ball or the ocassional party/gathering.

To my father, Napoleón Alfredo, who now rests in peace, I can never thank him enough for his care. He always taught me the value of hard work and academic preparation. To my mother Josefina, and my sisters, Iris and Camila; their support from afar pushed me forward. Special thanks also to Rut de Molina, who through the distance, was always there for advice and support in the ups and downs of these two years.

Most importantly, however, if there is one person I owe it to being here is my Lord, Jesus Christ, who has always been there as a shining light to look upon in hope. Now I can only reaffirm the following words - *For you are my hope, oh sovereign Lord, my confidence since my youth.* (Psalm 71:5)

Napoleón E. Cornejo B.
Delft, The Netherlands
July 2, 2009

1 Introduction

Space is regarded as the ultimate frontier for man. Since the beginnings of humanity, men have gazed at the skies in wonder and awe. In a first step to conquer these limits, the Wright brothers in the early 1900's successfully built and tested the first airplane. On 20 July 1969, Neil Armstrong from the United States of America put a foot on the moon. In less than one generation's lifetime, humankind had astonishly moved forward in their endless and insatiable effort to conquer the skies above.

As technology evolved during the 20th century, space became less and less the sole territory of governments and companies starting launching probes and satellites for commercial purposes. Such is the case that as of this day, several hundred operational satellites are currently in orbit around the Earth. Purposes range from the educational to the military.

More recently, universities and higher education institutions have taken into their efforts the development and launch of small satellites, called pico, nano or microsatellites (depending on their size). This serves mostly for educational purposes and engineering experience, along with experimentation with on board payloads. Moreover, as these satellites employ commercial off the shelf components (COTS), their development budget ranges from $65,000 to $80,000, well within the affordable cost range of universities and schools.

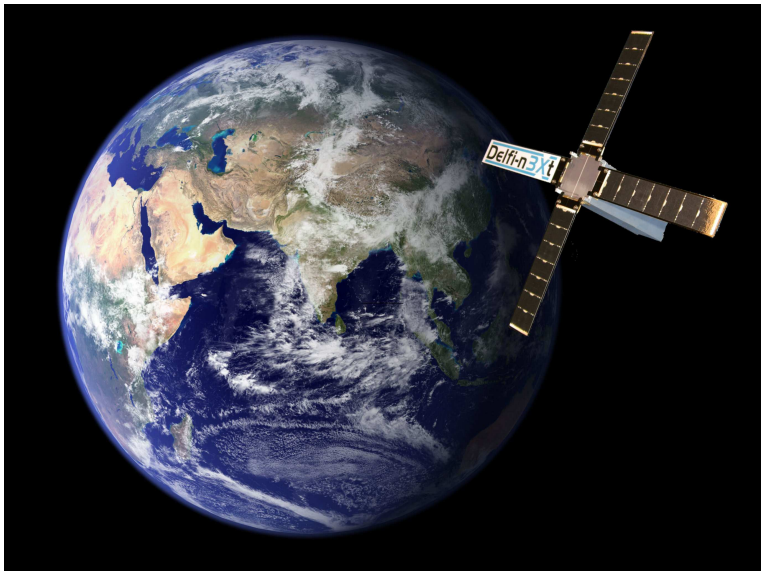


Figure 1.1: Delfi-n3Xt Nanosatellite.

The Chair of Space Systems Engineering at the Faculty of Aerospace Engineering of

the Delft University of Technology has engaged in the development of student nanosatellites as part of an initiative to give students hand on experience into the design and development of such systems. The first major achievement of this programme was the development and launch of the Delfi-C^3 on April 28, 2008 from Sri Harikota, India. As of this moment, Delfi-C^3 has more than a year in orbit and has successfully transmitted more than 500,000 telemetry frames (72.14 MB of data). This ambitious programme aims to launch a satellite every 2.5 years.

1.1 The Delfi-n3Xt Nanosatellite

As a successor to Delfi-C^3, Delfi-n3Xt, illustrated in figures 1.1 and 1.2, will be 3.3 Kg three-unit cubesat with advanced three axis stabilization, battery and high speed data link. Defined on 21 February 2008, it is currently being designed by 80 students and will focus on the qualification of microtechnologies from the Dutch space industry.

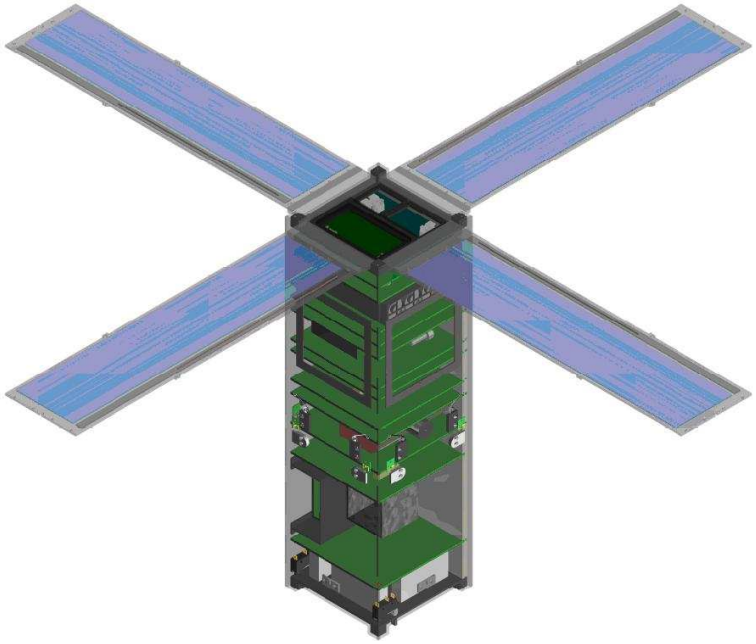


Figure 1.2: Delfi-n3Xt Nanosatellite.

1.1.1 Mission Statement

The mission of Delfi-n3Xt can best be described by the following goals:

- Pre-qualification of a micro-propulsion system from TNO, TU Delft and UTwente
- Pre-qualification of a Multi-functional Particle Spectrometer (MPS) from Cosine Research BV
- Scientific Radiation Experiment of Si-solar cells from DIMES

- Qualification of a high-efficiency communications platform from ISIS BV
- Proof-of-concept for a radiation risk-free implementation of commercial solid-state data storage devices, provided by NLR

Moreover, the satellite will provide the following advancements over its predecessor:

- Implementation of 3-axis active attitude control
- Providing high data-rate (> 9.6kbps) communication links
- A single-point-failure-free EPS with energy storage

1.1.2 Payloads

The five payloads on board Delfi-n3Xt were selected on the basis of their feasibility, educational value, services provided, level of innovation and design impacts. After an assessment procedure carried out by the systems engineers, the instruments listed below have got a flight opportunity. The current position of each within the structure of the satellite is displayed in figure 1.3.

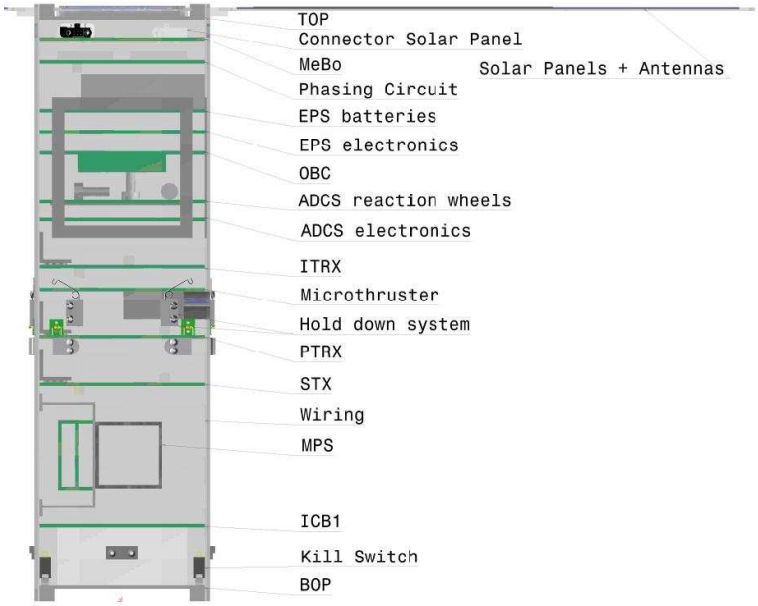


Figure 1.3: Delfi-n3Xt Block breakdown.

- Cool Gas Micropropulsion System - TNO, TU Delft, UTwente

  Designed to supply thrust for positional and orbit corrections, it contains compact storage for solid state propellants and a highly integrated feeding and thruster system based on MEMS technologies. Currently, this unique storage and release technology works with nitrogen, oxygen and hydrogen. The model is shown in figure 1.4.

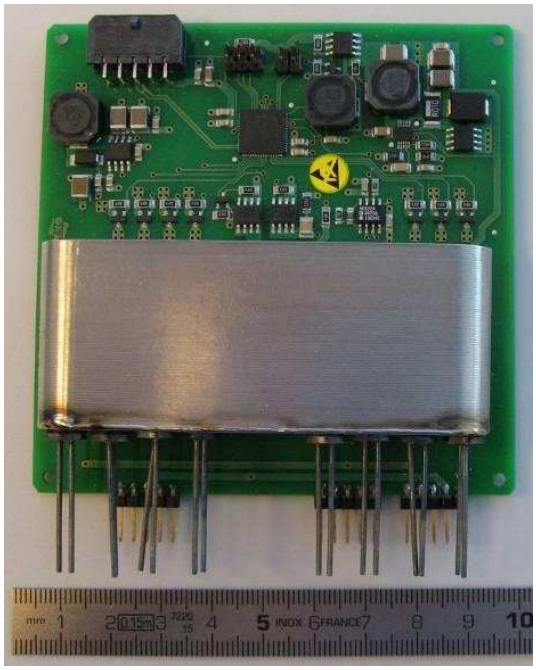


Figure 1.4: Micropropulsion Payload.

- Hydrogenated Amorphous Silicon Solar Cells - DIMES

  Hydrogenated Amorphous silicon allows the fabrication of low cost solar cells, lightweight and radiation hard. In space, these cells tend to degrade due to the effects of sustained illumination and radiation impacts. Measurements of voltage-current relationships on these cells will provide information on the resilience of this technology and predict the end-of-life for future uses.

- Multifunctional Particle Spectrometer - Cosine Research BV

  Referred to as MPS, it is a new type of radiation spectrometer that is able to detect protons, electrons, ions and gamma rays. The device is spectrally sensitive over large energy ranges and also able to measure the angle of incidence within 10 degrees. The tracker is based on a Va32Ta ASIA and the readout is done with a Xilinx FPGA. A prototype is illustrated in figure 1.5.

- Space Flash Memory - NLR

  COTS memory is sensitive to radiation, so different measures need to be taken in order to protect it from damage. It contains electronics to compensate for Single Event Upset (SEU) with redundant data and Single Event LatchUp (SEL). Additionally, it will keep measurements of these events.

- Efficient Nanosatellite Transceiver Module - ISIS BV

  Based on the the transceiver on board the Delfi-C^3, this one has a higher efficiency power amplifier and a more modular design. For uplink, the ITRX uses UHF at

Figure 1.5: MPS Prototype.

a maximum of 1200 bps and VHF at a maximum of 9600 bps for downlink. The transmitter power, which will be at least 400mW may be changed via commands. As a contingency, this payload may be used as a backup command receiver.

1.1.3 Subsystems

- Electrical Power Subsystem (EPS)

  The EPS consist out of four solar arrays directed to the sun and a maximum power point tracker to get the most power out of the solar cells. About 10 Watts of power is expected on the primary power bus of the satellite. Furthermore, for energy storage, four li-ion batteries will have enough capacity for full eclipse operations.

- Command and Data Handling Subsystem (CDHS)

  Performs two major functions. It receives, validates, decodes, and distributes commands to other spacecraft systems and gathers, processes, and formats spacecraft housekeeping and mission data for downlink or use by the On-Board Computer (OBC). In Delfi n3Xt, power switching and deployment operations, and OBC also belong to the CDHS.

- Communication Subsystem (COMMS)

  COMMS will carry no less than three radios: the primary transceiver PTRX, the payload ISIS transceiver and the high-speed transmitter STX. The power of the transmitters is less than a standard Christmas light bulb, but it is enough to transfer around 100 Mbit of data per day, which corresponds to 6000 typed paper sheets. The high-speed transmitter works at 250 kbps. The ground station is located in Delft, but any standard radio amateur equipment can listen to the satellite beacon when it passes overhead.

- Attitude Determination and Control Subsystem (ADCS)

  The Attitude Determination and Control Subsystem will apply active three-axis control. The sensors chosen are sun sensors, magneto meters and optionally gyro

sensors. The actuators are reaction wheels and magnetorquers. The ADCS has to perform sun pointing, measuring, thrusting and tracking.

- Structural Subsystem (STS)

  The Structural Subsystem of Delfi-n3Xt makes use of a customized structure. Some key elements in this structure are the use of a rod system and detachable sides, and the use of a symmetrical PCB layout for dimensions and holes. The objective is reduction in time needed for assembly, integration and testing, and improving the handling capabilities.

- Thermal Control Subsystem (TCS)

  The solar panels will experience large thermal cycles, therefore the need arises for extensive thermal testing of the panels. For the body of the satellite, it is expected that the cycles are not as extreme. This is due to the relatively constant power consumption within the satellite as the implementation of batteries enables a large functionality during eclipse. And as most of the time the body will be in the shadow of the solar panels (the satellite will point itself towards the sun during most of the time), the solar radiation will not be the most most important factor that influences the thermal behavior of the body. It creates possibilities to use passive cooling to prevent overheating of the PCBs.

1.2 Thesis Motivation

The launch of Delfi-C^3 on April 28, 2008 and its subsequent operational life is regarded as a success and provided valuable data for the payload partners and experience to the student designers. However, despite the success, the mission has experienced important problems in the Command & Data Handling Subsystem (CDHS), causing "hiccups" and delays that eventually take a toll on data collection and processing.

Therefore, the motivation of this thesis springs from the problems experienced on Delfi-C^3, encompassing a robust design and implementation for the CDHS and the exploration of fault tolerance methods that could add to the reliability of the system, with a strong focus on the data bus. For low level fault tolerance, we discuss techniques for the CDHS focused on software methods and an analysis of applicable error detection codes.

Fault detection methods can also be applied on a higher level, and therefore this thesis also decides to explore model based fault detection, in an effort to include techniques on a system level which could be implemented on future nanosatellites. In this particular case we decide to study the application of this method to the ADCS of Delfi-n3Xt.

1.3 Thesis Organization

The organization of this thesis is structured in the following manner. Chapter 2 is dedicated to an exploration of the design of the Command and Data Handling Subsystem on-board the Delfi-n3Xt, preceded by an overview of the design and problems of Delfi-C^3. Based on that analysis, the concepts and enhancements of Delfi-n3Xt are explained.

Chapter 3 discusses techniques and concepts for fault tolerance specifically for the CDHS, focused mostly on internal bus communications and error detection options. Chapter 4 explores fault tolerance concepts on a system level, with applications to attitude determination sensors. An innovative technique based on a variant of the Kalman filter is applied to the ADCS system of Delfi-n3Xt. Finally, chapter 5 summarizes the work and provides guidelines for future research and engineering efforts.

Delfi Programme Command and Data Handling Subsystem

2

This chapter contains an overview of the most important aspects of the Command and Data Handling Subsystem (CDHS) for the Delfi-n3Xt mission, although many of the concepts are being conceived in such a way that they can be reused in subsequent missions, so Delfi-n3Xt and Delfi programme may be used interchangeably. The first section briefly describes the CDHS of the Delfi-C^3 and outlines some of its problems. The following sections will provide an insight into the On-Board computer system and the data bus. The data bus is of major concern since it is the component that caused the highest number of glitches in the CDHS of Delfi-C^3. Therefore, special focus is given here to its architecture and implementation.

2.1 Delfi-C^3 CDHS

The CDHS for the Delfi-C^3 was based on an I^{2}C data bus running at 15.9 kHz and the FM430 Flight Module as an OBC, running at 1 MHz, which is based on the MSP430 family of processors from Texas Instruments. The architecture of the FM430 is shown in figure 2.1. The major drawback of this was the PC104 form factor of the board, which required a support PCB to fit into the satellite.

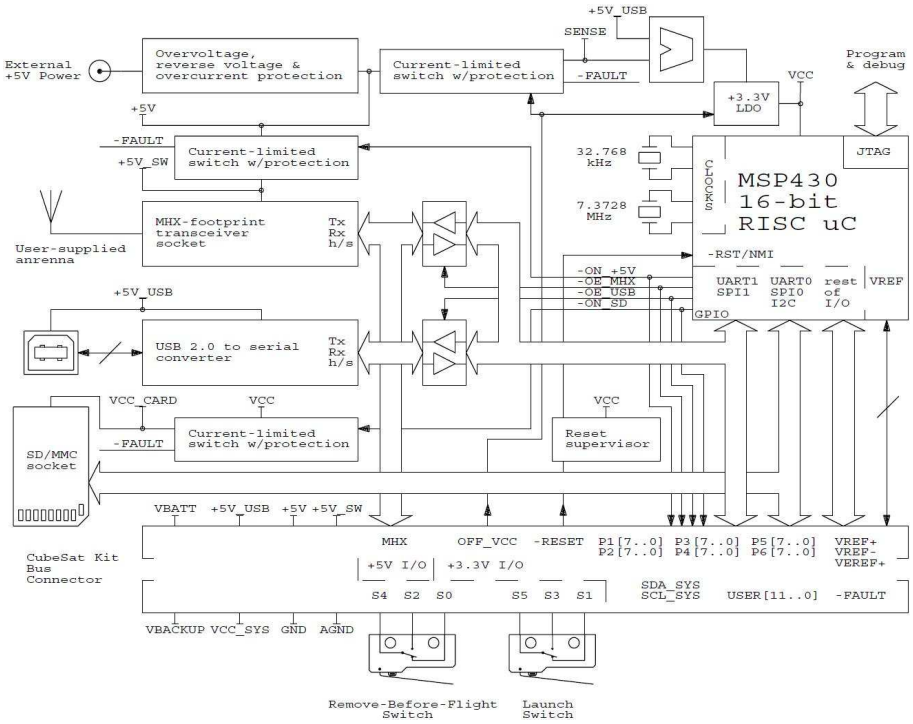


Figure 2.1: FM430 Flight Module Architecture.

Subsystems worked with PIC microntrollers running at 1 MHz as well. Power switching for each subsystem was also controlled by a separate PIC processor running at 31 kHz, shown in figure 2.2 as the EPS microcontroller. As the next section explains, these slow running nodes created issues in performance.

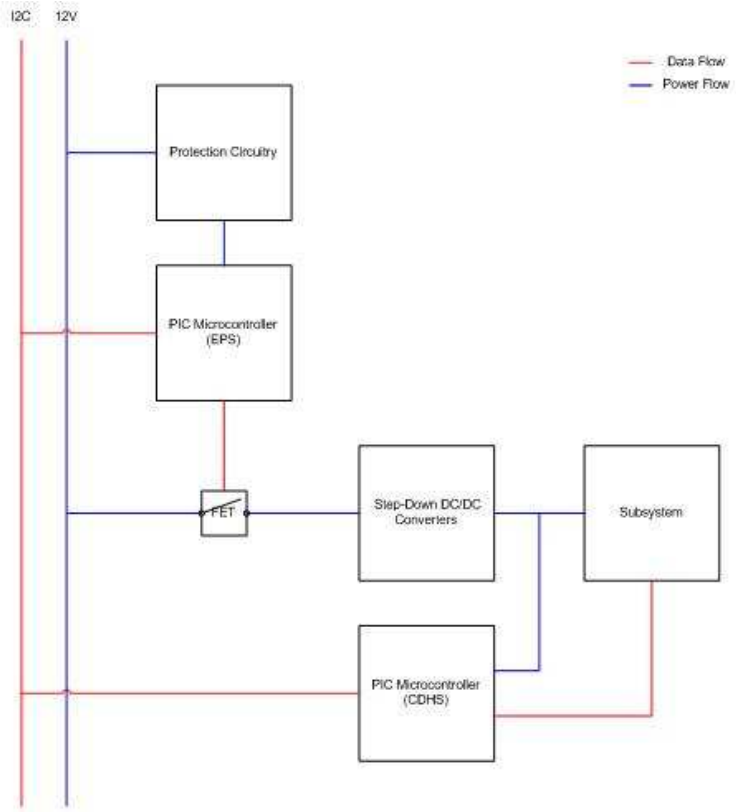


Figure 2.2: Delfi-C^3 Subsystem Architecture.

2.1.1 Issues

Although the design can handle most of the tasks required by the CDHS, it does experience errors and lost data frames from time to time. Determining the exact cause of the problem has proved to be a daunting task due to the large amount of activity, variables and factors that have to be taken into account. The time at which these errors occur also appears to be random. The approach taken here to shed a light on the problem is based on bit error rate analysis from testing a similar setup with the Delfi-C^3 service layer code. The issue is that varying speed of the components in the bus has a noticeable impact on the amount of communication errors (bit-error and timeouts).

2.1.1.1 Test Setup and Results of BER Measurements

The purpose of this measurement was to confirm the idea that the bus speed at which Delfi-C^3 runs is one of the most important reasons behind the glitches in communications

between the subsystems. Therefore, the test was performed with varying bus speeds under the assumption that at some moment, there would be a marked rise in the amount of transmission errors. To measure the bit error rate of the Delfi-C^3 data bus, we mounted a I^{2}C bus with three nodes: an MSP430 OBC and two PIC boards to simulate subsystems (see figure 2.3).

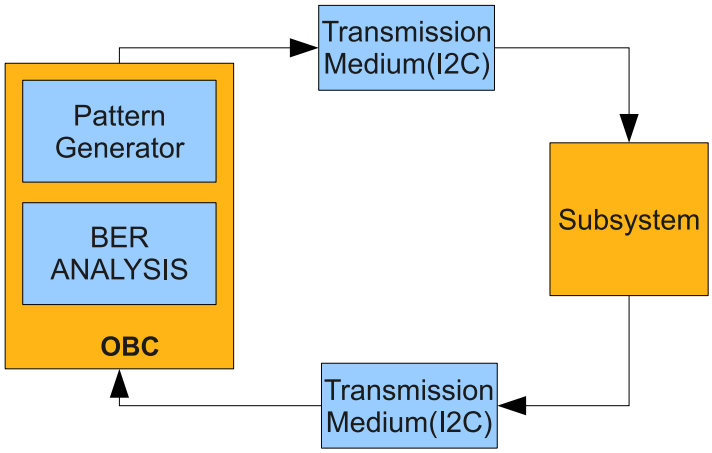


Figure 2.3: Delfi-C^3 Setup for BER Measurement.

Under the assumptions of a Gaussian error distribution, the required time for the test is given by the following formula:

$$t = \frac{ln(1-c)}{br} \tag{2.1}$$

where t is given in seconds, c is the confidence level (0-1), b is the upper bound for BER and r is the bit rate, or bus speed in this case. In our test, since we vary the speed, each test will require a different amount of time. However, we chose $c = 0.95$ as our confidence level and $b = 10^{-6}$ as our upper bound bit error rate, as for a CAN bus the acceptable BER can range from 10^{-4} to 10^{-6}[8],[28]. For example, a test for a 15kHz bus would take approximately 189 seconds. Each test is repeated five times and the result is the average of those tests.

The satellite bus usually has a timeout defined, under which the OBC assumes a communication failure. In Delfi-C^3 this timeout is set to approximately 400 milliseconds. To conduct the BER measurements, two different tests were performed; one with the timeout and another without it. This is to eliminate the effect of this timing requirement in the error rates.

Without time constraints, the results are shown in graph 2.4. As expected, the slow running slaves to perform worse, yielding surprise. It is worth mentioning that the communication software contains within a parity scheme to help detect errors. This error

detection scheme, however, does contain flaws that limit the effectiveness of communications. More in depth discussion of its implementation will be given in sections 3.2.3.1 and 3.2.4.

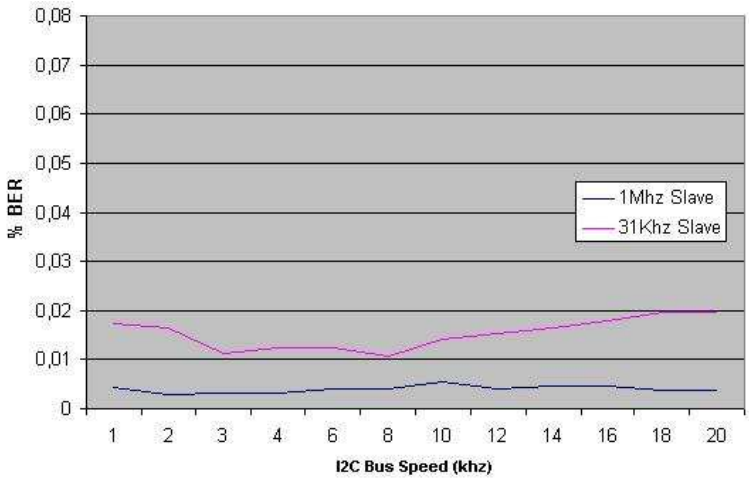


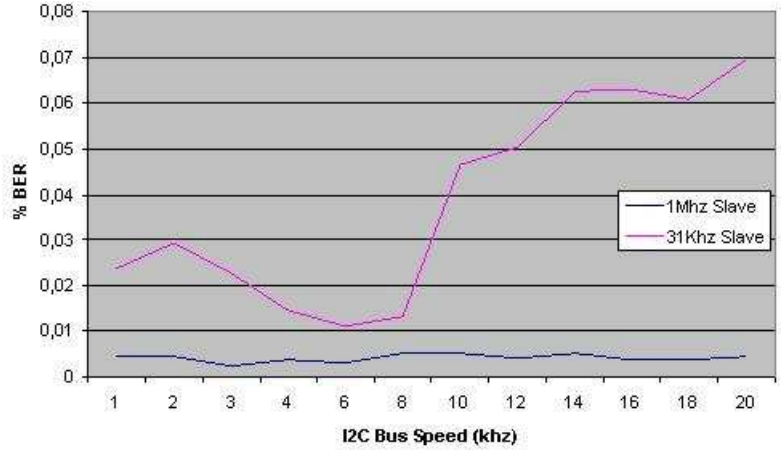
Figure 2.4: BER Measurement with no timeout constraints.

Figure 2.5 shows the measurements taken with the time constraint (400 ms). There is a visible change in the amount of errors for the slower running slaves; a visible peak after 8 kHz, while the 1 MHz node keeps an acceptable performance throughout the test. As expected, too, the amount of timeouts is greatly dominated by the slow running slaves. This could mostly be attributed to the parity calculations. For very low speeds (1 kHz), the OBC will observe more timeouts, which can be explained by the fact that such a low speed will more likely consume more time than the 400 ms timeout limit.

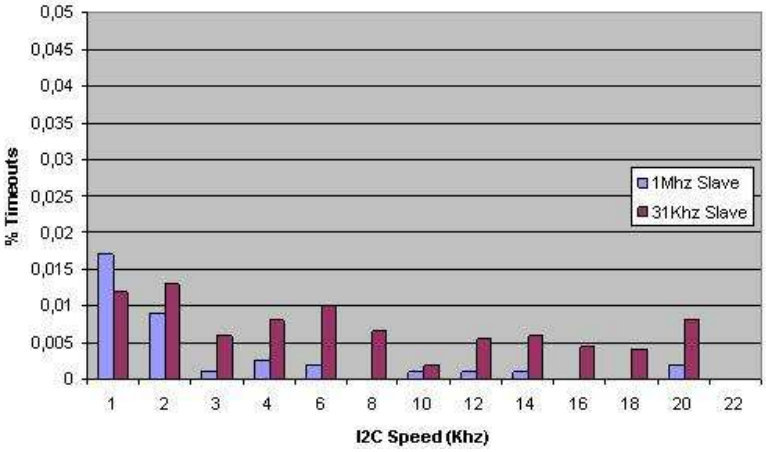
I^2C specifications dictate that all nodes in the bus should run at least 10 times faster than the bus speed[18]. In the design of the satellite, the bus is running at approximately 15.9 kHz, which is completely out of compliance. The ultimate reason behind this was a flaw in the power subsystem design in which no more power could be provided to the rest of the subsystems and therefore these components (PIC microcontrollers) had to be slowed to 31 kHz to work under such tight power constraints. Figure 2.2 shows the setup.

2.1.1.2 Additional Software Issues

During these BER experiments, there were additional observations that are worth mentioning. One of them is the fact that during the runs, the I^2C operation would halt from time to time, for a random number of seconds, and then wake up again. While debugging the program, the error was isolated in the USART0 interrupt request, specifically in the case where the I^2C bus receives more data than expected and therefore remains in constant interrupt. The timeout that should handle this condition is based on tickers from the MSP430s Timer A. However, according to Texas Instruments (TI) documentation on the MSP430, USART0 has a higher interrupt priority than Timer A, and therefore could block the timer counter, perhaps inhibiting proper timeout handling.



(a) BER Measurements with Timeout



(b) Timeouts

Figure 2.5: BER Measurements with time constraints.

An attempt to correct this issue was to implement the timer counter based on Timer B, which has a higher priority than USART0. Although the halting was relieved to some extent, it did not correct the problem entirely, implying a deeper problem. Given the fact that START and STOP signals on the bus are software controlled in the Delfi-C^3 implementation, special care needs to be taken to handle all possible conditions. Specifically, the following section of code taken from the USART0 interrupt routine is in charge of handling bus overflow:

```
/* Receive ready */
else if((I2CIFG & 0x10) == 0x10)
{
   /* [other code ...] */

   if(receive_count > receive_count_goal ||
                i2c_status == I2C_STATUS_IDLE)
```

```
    {
        I2CIFG &= ~0x10;          // clear flag
        return;                   // return immediatel
    }
```

The attempt here is to clear the interrupt flag and return immediately, allowing the rest of the system to run. However, it appears that once in this section, it will tend to remain here until the timer is given a chance to react and handle the timeout condition. This is a possible explanation to the random hiccups on the data bus. The correction in this case, is not only to clear the interrupt flag, but also, according to TI documents, the data buffer must be read to clear it and stop further interrupts. This correction eliminated the problem.

```
if(receive_count > receive_count_goal
        || i2c_status == I2C_STATUS_IDLE)
{
        char t = I2CDRB;                    // read data buffer
        I2CIFG &= ~0x10;                    //  clear flag
        return;                             // return immediately
}
```

2.2 Requirements for Delfi-n3Xt CDHS

This section contains the requirements specified in [5] that are directly related to the CDHS. Most of them deal with the OBC and functionality, which will be developed in the application level of the software. In this thesis, we are mostly interested in low level software for communications and data bus. However, since all of these requirements are interrelated, they are included here for completeness. The numbering corresponds to the numbering used in [5].

- SAT.2.6.REQ.C.000: The CDHS shall operate in the temperature envelope of Delfi-n3Xt
- SAT.2.6.2.1.REQ.F.000: The OBC shall acquire all incoming telecommands presented by PTRX and ITRX
- SAT.2.6.2.1.REQ.F.001: The OBC shall acknowledge the last received telecommand
- SAT.2.6.2.1.REQ.F.002: The OBC shall acknowledge the execution of the last received telecommand
- SAT.2.6.2.1.REQ.F.003: The OBC shall store telecommands for later execution and/or distribution upon acknowledgement by ground operators
- SAT.2.6.2.1.REQ.F.004: The OBC shall distribute processed commands to the relevant subsystems and payloads if applicable
- SAT.2.6.2.1.REQ.F.005: The OBC shall acquire all housekeeping data presented by the subsystems

- SAT.2.6.2.1.REQ.F.006: The OBC shall acquire all payload data presented by the payloads
- SAT.2.6.2.1.REQ.F.007: The OBC shall send all relative housekeeping and payload data to the PTRX or ITRX for down link to ground
- SAT.2.6.2.1.2.REQ.F.008: The OBC shall determine the use of PTRX or ITRX for transmitting
- SAT.2.6.2.REQ.F.000: The OBC shall buffer payload data that cannot be down linked directly for later down link
- SAT.2.6.2.1.REQ.F.008: The OBC should transmit all produced payload data to ground using STX
- SAT.2.6.2.1.REQ.F.009: The OBC shall tag all payload data packages with an RTC timestamp of the time that payload data is acquired from the payload
- SAT.2.6.2.1.REQ.F.010: The OBC shall tag all telemetry frames with an RTC timestamp of the time that the telemetry frame is assembled
- SAT.2.6.2.REQ.F.001: The OBC shall keep track of time
- SAT.2.6.2.REQ.F.002: The OBC shall command and perform tasks triggered by a schedule events or telecommands
- SAT.2.6.REQ.F.000: The CDHS shall control and monitor the boot sequence of the satellite
- SAT.2.6.REQ.F.002: The CDHS shall verify and test the functionality of subsystems and payloads
- SAT.2.6.2.1.REQ.F.011: The CDHS shall use dummy data for requested data that is not received from subsystems or payloads
- SAT.2.6.2.REQ.F.007: The CDHS shall be able to be monitored through a test interface
- SAT.2.6.1.REQ.F.000: The I2C data bus shall be accessible externally for monitoring
- SAT.2.3.1.REQ.F.001: PTRX shall send direct telecommands over the data bus as master
- SAT.2.3.1.REQ.F.002: PTRX shall validate direct telecommands
- SAT.2.6.1.REQ.P.001: No microcontroller may hang-up the bus by a hardware or software failure

Additional requirements that depend on the bus protocol are listed in section 2.4.4.

2.3 On-Board Computer (OBC)

The Delfi-n3Xt OBC will be the central part of the CDHS, and thus, of pivotal importance to correct design of the satellite. The OBC consists of more than the central processing unit, but also considers components such as memories, clocks, oscillators, etc. Some important considerations for the CPU include low power, large temperature range of operation and proper interfaces for the necessary peripherals. At the moment, the selected CPU for the CDHS is the MSP430F1612 from Texas Instruments because of its extremely low power consumption (1 microamp at 2.2 V in idle mode), low cost, wide temperature range of operation and the quality of the tools available for development.

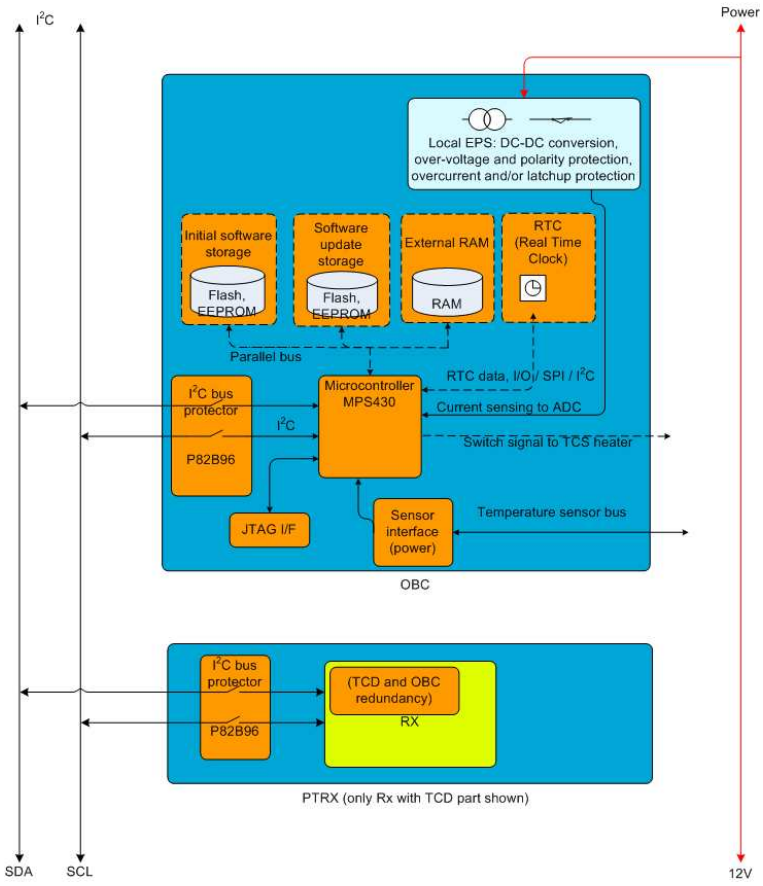


Figure 2.6: Current OBC Architecture.

Additional to the microcontroller, the following peripherals need to be present to design a complete system:

- Memories (FLASH and RAM)

  The nanosatellite will have only a limited number of passes over the ground station in Delft, and whenever it does, the communication window is of about 10 to 12 minutes. Therefore, the CDHS is required to store most of the data collected from the payloads during its operation for later retrieval.

- Bus Controllers

  To communicate with the rest of the subsystems, a proper data bus has to be implemented. The following section illustrates several possible bus protocols for embedded systems and explains the considerations for choosing I^2C.

- Real Time Clock

  The Delfi-C^3 mission tagged every data frame with a counter. However, when the computer reboots, the counter is not always correctly restored. Hiccups in the bus also affect the correct order in which packets arrived and thus, their tag. For the following missions, each frame will be tagged with the time. For this purpose, a separate real-time clock that precisely keeps track of time will be included in the design. At the moment, the component most likely to be selected is the the DS1388 chip, which also interfaces via I^2C.

- Power Supply

  Although most microcontrollers carry on mechanisms to maintain proper power supplies, it is necessary to take additional precautions. The EPS will provide 12V, but most components work at 5V and the MSP430's at 3.3V, so DC-DC conversion is necessary. Moreover, outer space radiation can cause unexpected low-impedance paths that may destroy the microcontroller. A latch-up protection current limiter circuit can provide this protection.

- Oscillator

  The internal oscillator in the MSP430F1612 has RC characteristics and is very unstable and dependent on temperature. Crystal-based oscillators are therefore more appropriate. Most likely a 7.3728 MHz crystal oscillator will be included as the external clock source for the CPU. The RTC as well, requires a 32.768 kHz RTC.

- Debugging Interface

  For proper development and debugging, it is important to have access to the internal networking of components, most importantly the microcontroller. The MSP430F1612 comes with a JTAG interface both for programming and debugging.

In the case of an OBC failure, it is expected that one of the subsystems will be able to take over operations. At this time, the PTRX is seen as candidate to incorporate some of the functions of the OBC, but in degraded mode. At least, the PTRX will contain a telecommand decoder (TCD) in such a way that it transmits commands received from Earth directly to the bus, and thus, provides some level of manual control.

2.4 Embedded Data Buses

Nanosatellites developed by the DELFI programme are in essence a collection of embedded systems controlled by a central On-Board Computer (OBC). The PCB's are stacked

on top of each other, so a proper and flexible data bus, along with a power bus, should enable proper operation of the whole system.

The selection of a proper data bus technology for Delfi-n3Xt has to comply with the requirements of the CDHS as described before. Traditional data buses used in space applications, such as those considered in ESA and NASA literature are high-end buses for long-haul communications between completely different and independent systems. Besides requiring additional components or a relatively large area, these buses may also consume more power than what usually can be delivered by power-tight embedded systems. Examples of these architectures are SpaceWire, FlexRay, SAFEBus, Ethernet and MIL-STD-1533. A nanosatellite such as Delfi-n3Xt, requires a compact and embedded data bus for inter-communication. Examples of serial embedded buses of this nature are SPI [4], I^2C [18],[21] and CAN Bus [27]. Additional to these, wireless standards, such as Bluetooth and Zigbee, are also used in embedded applications and are readily available in COTS components. In the following subsections we give a brief overview of those buses evaluated to be included as part of the Delfi-n3Xt design.

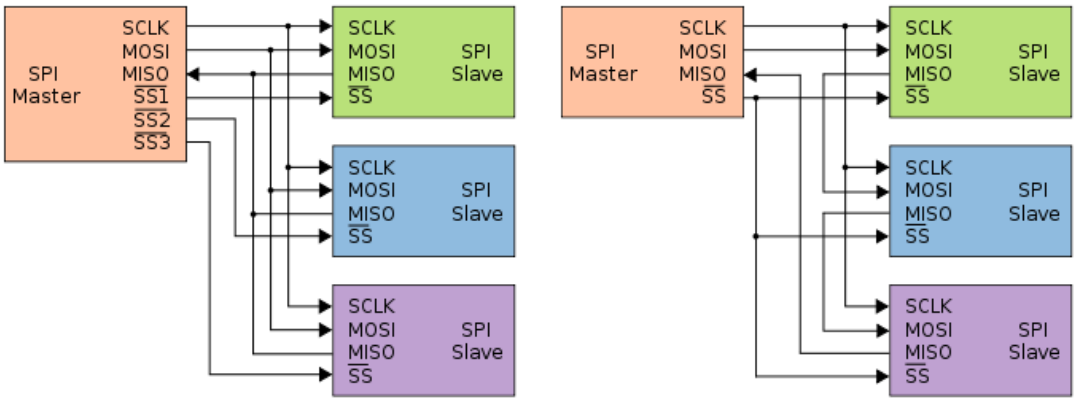
2.4.1 Serial Peripheral Interface Bus (SPI)

Figure 2.7: SPI Data Bus.

The Serial Peripheral Bus [4], shown in figure 2.7 is a de-facto standard developed by Motorola that is able to communicate in full duplex and based on master/slave(s) configuration. It is designed to work at relatively high speeds of about 180 MHz or more. The bus has four lines:

- Serial Clock (SLCK)
- Master Output, Slave Input (MOSI/SIMO)
- Master Input, Slave Output (MISO/SOMI)
- Slave Select (SS)

Data transmission within this bus is initiated by the master, first configuring its clock using a frequency that can be handled by the slave. Then, the master has to pull the SS line low of the device with which it wants to communicate and waits for certain period before it starts issuing clock cycles. As clock cycles are transmitted, the master writes into the MOSI line for the slave to read and the slave on the MISO line, for the master to read.



(a) One-to-one (b) Daisy Chain

Figure 2.8: SPI Topologies.

The topologies usually handled by the SPI are one-to-one and daisy-chains, both illustrated in figure 2.8. The one-to-one is self explanatory and works as described above. In the daisy chain arrangement, each device will transmit on the second session of pulses what it received in the first session. Each outputs the data on MISO line and receives on MOSI. JTAG and SGPIO use daisy chain configurations for their applications.

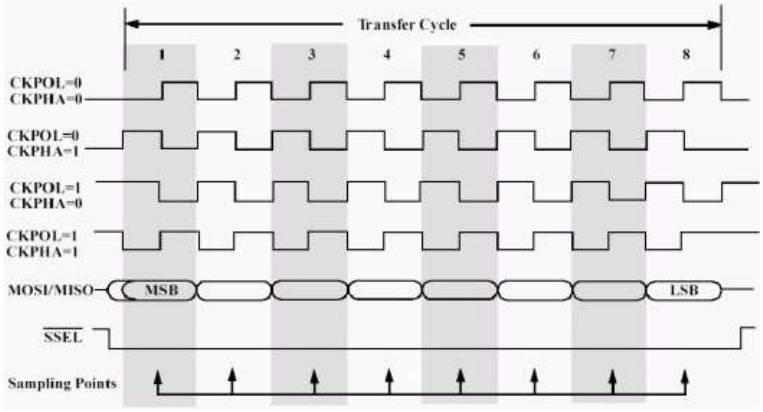


Figure 2.9: SPI Timing Diagram.

One unique aspect of SPI is its ability to configure the clock polarity. Data may be valid either on a low or high state of the clock pulse, depending on the CKPOL and CKPHA options. CKPOL=0 indicates that the base value of the clock is 0, and data is read in the rising edge of the clock (CPHA=0) or in the falling edge (CKPHA=1). Inversely, CKPOL=1 gives a base value of 1 to the clock pulses and CKPHA=0 indicates data will be read on a falling edge and rising edge when CKPHA=1. Figure 2.9 shows the timing diagram of an SPI bus for these options.

One important aspect of SPI is the fact that since it is designed to work at relatively high speeds (to hundreds of MHz), wires have to be short to avoid high reactance of these. This makes it quite suitable to use within the PCB, but not within separated subsystems. Lower speeds may be used for this purpose, at the disadvantage of having to deal with 4 lines. Other protocols are better suited for this.

2.4.2 I^2C Bus

The I^2C bus [18], [21] is a multi master serial bus based on two open drain wires, one for data (SDA) and one for clock (SCL), both pulled up via resistors. The basic architecture is shown in figure 2.10.

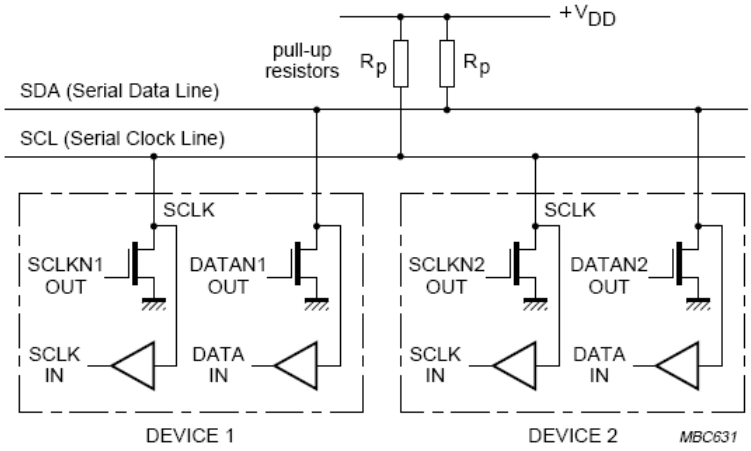


Figure 2.10: I^2C Bus, adapted from [18]

Data on the SDA line is only valid when SCL is low and its frequency is controlled by the master. It is common to use a speeds of 10 Kbit/s (low-speed mode) or 100 Kbit/s (standard mode), but recent revisions allow for speeds of 400 Kbit/s, 1 Mbit/s and 3.4 Mbit/s. Each device is identified by its unique 7-bit address (16 addresses are reserved) although recent revisions allow for extensions of 10-bit addresses that need to follow a slightly different protocol sequence. This basic communication sequence will always be initiated and finished by the master via START and STOP conditions and will also indicate which party will be transmitting data via an R/W bit. The START condition happens when the SDA line is pulled low while the SCL line is high and the STOP condition occurs when the SCL line is pulled high right before the SDA. The timing diagram for an I^2C transmission is illustrated in figure 2.11.

I^2C also allows for the presence of more than one master in the bus. Arbitration for this involves synchronization of clock pulses and data. For the clock line, the bus specification dictates that in case of differing pulses, the longest LOW period and the shortest HIGH period will be conserved by SCL (figure 2.12a). The arbitration procedure is eased by the fact that both SDA and SCL are pulled up by default, and will therefore stay low when pulled down by a device. Every time a master pulls down a line it must therefore check that the line actually goes to low. If this is not the case, that master has

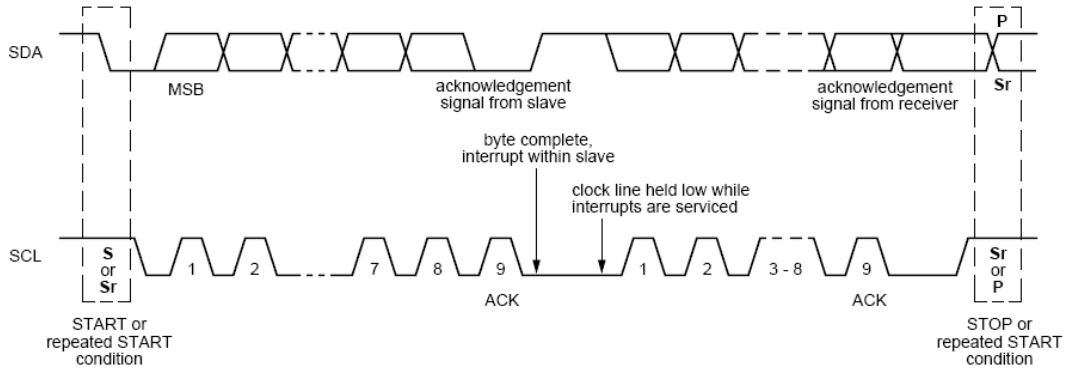
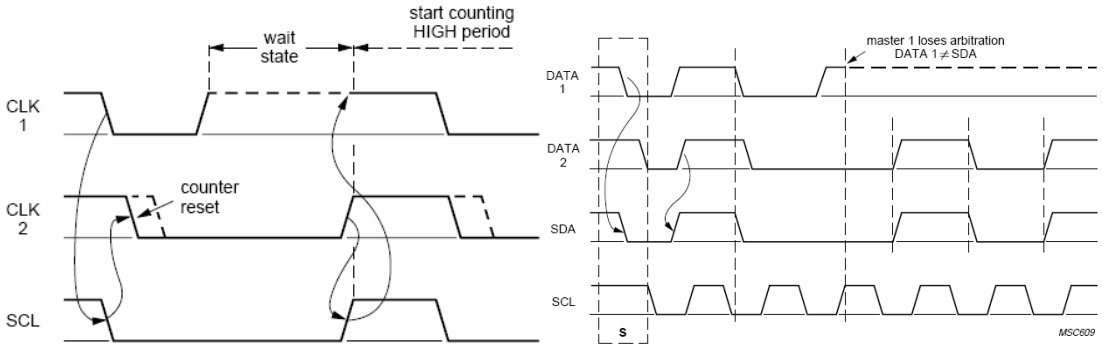


Figure 2.11: I^{2}C Timing Diagram, adapted from [18].

to back off because there must be another device using the bus at the moment (figure 2.12b). The mechanism is simple but effective, and avoids data corruption because before the difference was detected and the device steps down, all previous bits in SDA had to be equal.



(a) Clock Synchronization (b) Data Arbitration

Figure 2.12: I^{2}C Multi-master Arbitration, adapted from [18]

2.4.3 Controller Area Network (CAN) Bus

Originally developed for motor vehicles, the CAN Bus [27] is a multi-master, broadcast serial bus developed by Bosch GmBH. The bus, however, is not full duplex, so each node can only send or receive data at any given time. Since it is a broadcast bus, each node will sense the data, which is encoded in non-return-to-zero (NRZ) format. The bus is physically composed by a twisted pair of wires, terminator resistors at both ends and nodes that contain a host processor, a CAN controller for protocol handling and a CAN transceiver (figure 2.13).

This bus can operate at speeds that range from 1 Kbit/s to 1 Mbit/s, but has no clock line, so instead it uses low voltage differential signals in which every bit lasts for a pre-programmed amount of time quanta, then can be divided into four different

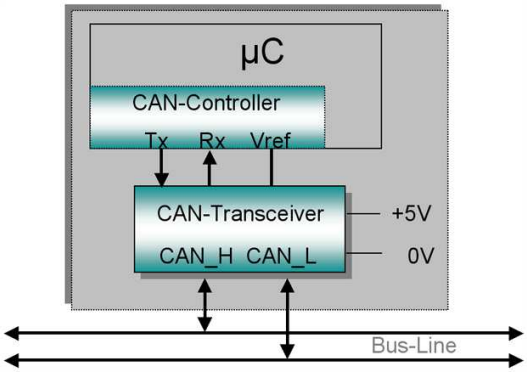


Figure 2.13: CAN Bus Node.

phases as illustrated by figure 2.14. The sync phase lasts for one quantum of time and helps synchronize after the level changes of the bus lines. The propagation segment compensates for physical delay times in the CAN network. Phase1 and Phase2 serve for adjustable delays based on network or node conditions.

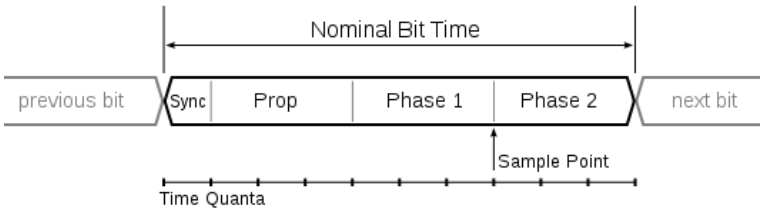


Figure 2.14: CAN Bit Timing.

The standard ISO 11898-1 (2003) describes the aspects of the data link layer, Logic Link Control and Media Access Control. Also, defines four data frame types:

- Data: for normal data transmission
- Remote: a request for transmission of a specific identifier
- Error: transmitted by nodes upon detection of an error
- Overload: serves to add delays between data and/or remote frames

Figure 2.15 shows the structure of the communication data frames transfered in the network. Start-of-frame (SOF) obviously denotes the beginning of a frame, followed by an arbitration block (discussed later), control field, data, checksum, acknowledgement and end-of-frame. The control field mostly serves to indicate the length of data transmitted in the frame. The checksum field contains a CRC of the data based on the $X^{15}+X^{14}+X^{10}+X^{8}+X^{7}+X^{4}+X^{3}+1$ polynomial. Finally, the ACK is two bit field that acknowledges reception.

In the protocol specification of CAN, the arbitration field contains the 11-bit identifier of the transmitting node. Since all messages are broadcasted, there is no need to specify

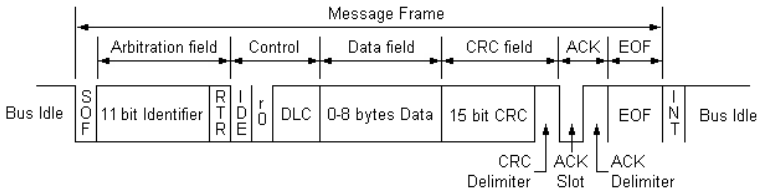


Figure 2.15: CAN Data Frame Format.

a destination. Whenever a node writes this field to the bus, it will monitor the state of the lines, verifying that they are at the level in which the transmitter expects them to be. If while sending a recessive bit (1), the line has a dominant value (0), the node has to stand down. This means, that the identifier of each node assigns them a priority over other nodes, and should be taken into consideration early in the design. This is more commonly known as Carrier Sense Multiple Access/Bitwise arbitration (CSMA/BA).

2.4.4 Bus Considerations and Evaluations

The criteria to chose a proper data bus for the DELFI nanosatellite programme includes considerations of complexity (both in hardware and software), power consumption, reliability and experience/heritage. The comparisons are summarized in table 2.4.4. Each of the criteria have been assigned a weight according to their importance and impact in the design.

	weight	I^2C	SPI	CAN
Power Usage	4	low(1)	low(1)	high(-1)
Complexity	3	low(1)	low(1)	high(-1)
Speed	2	medium(0)	high(1)	high(1)
Reliability	3	medium(0)	low(-1)	high(1)
Heritage	1	yes(1)	no(0)	no(0)
Tradeoff		8	6	-2

Table 2.1: Bus Type Comparisons/Tradeoff.

The Serial Peripheral Bus achieves high data rates with relative simplicity, but as mentioned in the description, works fine only in very short distances, better suited for PCB area. It defines no higher level protocol and no flow control (low reliability). Unlike CAN and I^2C, however, SPI can achieve full duplex communication, increasing it's communication rate. Therefore, SPI is better suited for applications where the node(s) will communicate with streaming data, but comes low in terms of reliability.

The CANBus is mostly used in the automotive industry and can come in three variations: High Speed, Fault Tolerant and Single Wire. Some of the advantages of this bus are greater protection for EMC issues, hardware based error detection and tolerance to failures that avoid global bus hanging. However, the two most notorious disadvantages are its relatively high power consumption and added hardware/software complexity. Very

preliminary estimates showed that power consumption for decentralized EPS nodes alone would consume around 600mW with a PIC18F2480 and around 1000mW in total if implemented with Microchip's MCP25025 I/O port. In terms of software, the PIC18F2480 needs 60 registers for the CAN module, while only 6 for the I^2C port. Moreover, the number of COTS components with I^2C support is higher than CAN and, although not as resilient, can be complemented with software and hardware features that will be detailed in the following sections.

The heritage and experience gained from Delfi-C^3, along with the available tools, provide a higher degree of confidence on the suitability of the I^2C data bus for Delfi-n3Xt. Power usage and complexity fall within the acceptable ranges for the design. Moreover, the SPI or CAN Bus are not useful for interfacing with local EPS switches; SPI because it would require too many lines and CAN because the added complexity for each node. I/O port expanders for I^2C are, on the other hand, simple and common. The weaknesses in reliability of I^2C, such as error detection and hardware resilience can be greatly improved with proper hardware and software implementations. These implementations will be described in the following sections.

Upon decision for an I^2C implementation, additional requirements are established:

- SAT.2.6.2.REQ.P.001: If the I^2C standard in standard mode (sm) must be used for communication trough Delfi-n3Xt, no adaptations to the protocol are allowed.
- SAT.2.6.2.REQ.F.008: The microcontroller(s) attached to the bus must be connected via a data bus protector.
- SAT.2.6.1.1.REQ.F.000: A bus protector must isolate the system from the data bus if the SDA low (logic 1) state exceed the time-out time for all systems.
- SAT.2.6.1.1.REQ.F.001: A bus protector must isolate the system from the data bus if the SCL low (logic 1) state exceed the time-out time for master systems.
- SAT.2.6.1.1.REQ.P.001: Bus protection circuits must be located as close as possible to the data bus connector.
- SAT.2.6.2.REQ.P.000: The microcontroller attached to the data bus must support 100 Kbit/s, standard mode I^2C. This means an internal clock speed of at least 1MHz.

2.5 Hardware Implementation

Although the I^2C protocol provides mechanisms for transfer, synchronization and acknowledgment as described in the previous section, it may be complemented with additional hardware mechanisms that ensure reliability for space flight. Moreover, the architecture has to consider a single-point-failure-free design as this is one of the main design goals of the project.

The actual implementation of the system data bus has many architecture possibilities, ranging from having a single bus through all subsystems to separate buses for different

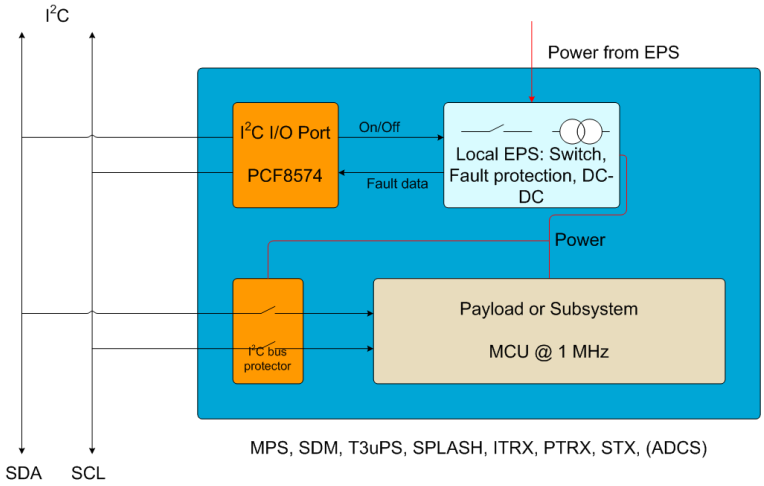


Figure 2.16: Delfi-n3Xt Data Bus Hardware Architecture.

purposes. Overall, a single bus provides a simple implementation, while bus separation can provide more reliability and failure resistance. With all tradeoffs considered, the Delfi-n3Xt will be designed with a single bus, similar to the Delfi-C^3 implementation but complemented with several enhancements. With all payloads and subsystems taken into account, the bus is expected to have around 20 nodes, including the OBC as the bus master. In the Delfi-C^3, the communication hiccups are heavily influenced by slow PIC microcontrollers attached to the local EPS switch on each node. In this next version of the satellite, the implementation will substitute these PIC microcontrollers with I/O ports (PCF8574) controlled directly via I^{2}C commands, which consume around 13 times less power (60uW-600uW compared to 8mW for the PIC) and can handle speeds of 100Kbit/s. An illustration of the architecture of a bus node is shown in figure 2.16.

2.5.1 Local EPS Switching via I/O Expander

As illustrated in figure 2.16, the two I^{2}C lines interface with the subsystem at the I/O Port PC8574 and at the I^{2}C bus protector. The I/O Port is an bus port extender, essentially a switch, that has a distinctive bus address and can be commanded to turn any of its outputs into high or low. This feature is used to serve as power switch for the subsystem, as it directly connects to the Electrical Power Subsystem (EPS). The connection between the bus and the actual microcontroller node is done via an I^{2}C bus protector. The architecture of this component is illustrated in 2.18.

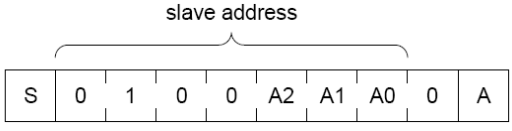


Figure 2.17: PC8574 Address Format, adapted from [24].

Each I/O port is also a slave in the I^2C bus and can be addressed independently at any time. Therefore, this component becomes convenient in case the need comes to reset a subsystem (hangs, errors, rollbacks, etc) or to power up at boot time. The address of this component not only identifies it, but also specifies the port and direction of data. Addresses in I^2C consist of 7 bytes in standard mode. Pins A0, A1 and A2 of the component are hardwired and specify the last 3 bits of the address. The first four bits address serve to communicate to any of the 8 ports. The format is shown in figure 2.17.

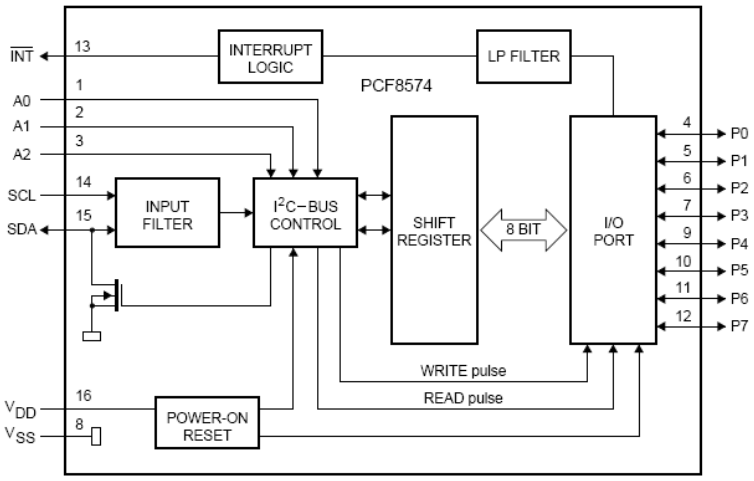
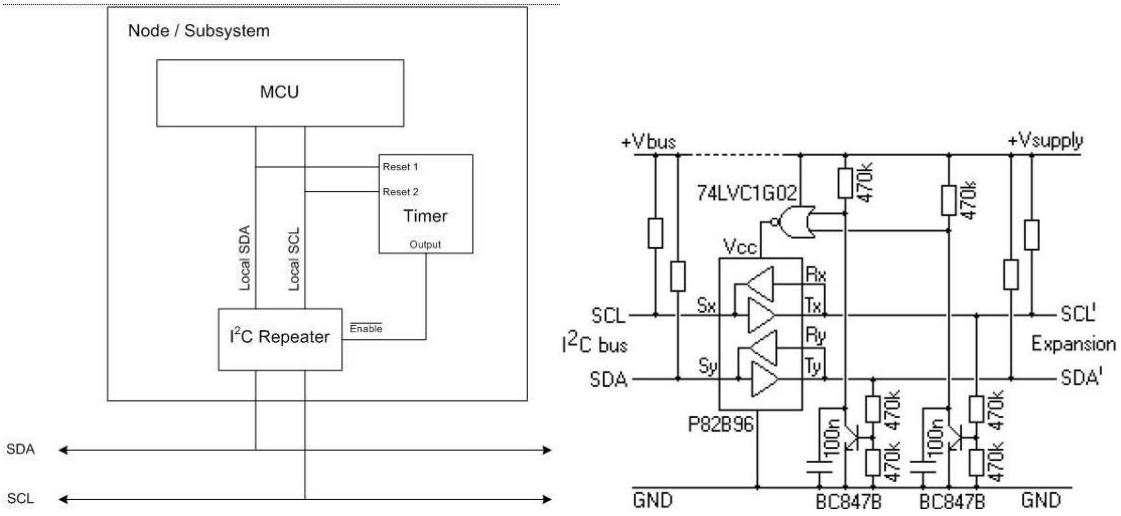


Figure 2.18: Architecture of Bus Expander for Local EPS, adapted from [24].

2.5.2 I^2C Protection Circuit

This bus protection is another innovation in the design and will be added to 12 nodes on the bus, mostly the microcontrollers in each subsystem. This bus protector is conceptually a timer (Fig. 2.19a) that monitors both the SDA and SCL lines. The timer is based on a simple RC circuit which can be tweaked to conformity by choosing the appropriate values of capacitance and resistance. The I^2C I/O ports are not interfaced with this protection under the assumption that it is an industry-proven chip that will cause no hangs. When any of these lines is turned low for a long period of time, the protector will disconnect the node from the central bus, avoiding a global hanging of the system. The circuit will keep the node disconnected from the bus as long as any of the lines (SCL or SDA) remain low. Therefore, the only ways that a node can reconnect is by waiting until comes out of the hanging condition or wait for the OBC to reboot the node, which is achieved by shutting down the electrical power via the PC8574 port.

Physical implementation of the bus is currently under discussion; however some considerations can be made on the type of wiring and connectors that would be ideal for the design. Due to the high flexibility, reliability and low volume, the preferred wiring would be Flex PCBs, but this is considerably more expensive than Nomex or Teflon wiring and has to be manufactured by a company. Some cost savings can be achieved



(a) Concept (b) Circuit

Figure 2.19: I^2C Protection Circuit.

by combining the EPS and data buses in the same wiring. A 10 pin wire allows for enough redundancy (4 I^2C lines, 6 for EPS) and with a proper pin assignment crosstalk can be reduced. Preliminary calculations show that capacitance for this setup would be approximately 315pF, which is acceptable under the I^2C specifications and a resistance of around 0.44 ohms which also seems reasonable for correct operation of EPS. Since the experience with Harwin connectors in Delfi-C^3 was good, this would also appear to be a good design choice for Delfi-n3Xt.

2.6 Software Implementation

The software implementation for the proper operation of the Delfi data bus deals with programming the proper interfaces and adding the capabilities required to make the I^2C design more robust. This short section serves to document the API developed for the service layer of the CDHS.

- int n_initI2C(unsigned char my_address, unsigned char mode)

  Initializes I^2C communications with a specified address and mode. Address is the 7-bit address of this node. Mode can be I2C_MASTER or I2C_SLAVE.

- int n_I2Crecv(unsigned char slave_address, volatile char *data, unsigned char length, unsigned int timeout)

  Used by master device to request information from a slave. The slave_address denotes the node to which information is requested. data is a buffer to contain the transmission, length is the expected size in bytes of the received information and timeout is the amount in milliseconds the master will wait before signaling a timeout at the upper level software layer.

- int n_I2C_slave_packet(volatile char *data, unsigned char length)

  Function called by a slave when a packet is ready to be delivered to the master. It will be transmitted in the next request for data.

- int n_I2Csend(unsigned char slave_address, volatile char *data, unsigned char length, unsigned int timeout)

  Function called by a master to write data into a slave. Again, slave_address is the slave to receive the bytes in data buffer, length is the size in bytes and timeout is the amount of milliseconds before a timeout error.

- void i2c_recv_callback(volatile char *data, int len)

  To be implemented by each slave in the upper software layer, will receive all incoming data from the bus. data contains the received bytes and len, the number of bytes.

- void d_initTimers(void)

  Initializes timer services with TIMER_A in continuous mode, meaning it will cause an interrupt upon reaching 0xffff and start again.

- unsigned int d_startTimer(unsigned int id)

  Starts the timer with the specified id. Will cause an interrupt on each count to 0xffff.

- unsigned int d_getTimer(unsigned int id)

  Obtains the current count from the timer with the specified id.

- void wait(int millis)

  Generic waiting routine.

- void init_sw_interrupts(void)

  Initializes software interrupt services.

- void register_sw_int_callback(unsigned char id, sw_int_callback callback)

  Registers a callback routine to be called upon a software interrupt identified by id.

- void unregister_sw_int_callback(unsigned char cause)

  Unregisters a callback from the software interrupt identified by id.

- void call_sw_interrupt(unsigned char id)

  Cause a software iterrupt identified by id.

2.7 Summary

This chapter presented a general overview of the Command and Data Handling Subsystem on-board Delfi-n3Xt. From the analysis and the lessons learned from Delfi-C^3, more focus was placed on a proper design of the communications data bus. High end data buses on board regular satellite systems like MIL-STD-1553, SAFEBus or Ethernet were not considered here as they are unsuitable for small nanosatellites. SPI, CAN and I^2C, were considered suitable and therefore evaluated in this work. The I^2C bus shows good performance and falls within the power consumption and speed requirements of the current mission.

The robustness that the I^2C bus may be lacking can be compensated with proper hardware and software design. In terms of hardware, proper shielding, redundancy and cabling will help increase its reliability. Additionally, circuit protection for any subsystem hanging the bus will prevent errors of this nature caused by hanged processors. Finally, proper software design, testing and documentation can guarantee the requirements of speed, reliability and data integrity.

3 Fault Tolerance for Delfi CDHS

The previous chapter discussed the design and inner workings of the Command & Data Handling Subsystem with strong focus on the data bus. In this chapter we will focus more on fault tolerance for the CDHS and data bus. We first start with a discussion of software level fault tolerance and gradually move into a deeper discussion on one of these techniques: Error Detection Codes.

It is important to keep in mind that not all methods and schemes for fault detection and recovery are applicable to the Delfi-n3Xt. Embedded systems are characterized by having very tight constraints in terms of timing and resource usage. Whereas software for traditional computer systems can have the luxury of multicore processors, redundant protected memories and multiple interfaces for input, a nanosatellite like Delfi-n3Xt has severe constraints in terms of power, hardware and processing capability that has to be taken into account when deciding for an approach in this matter.

3.1 Software Level Fault Tolerance

As in all mechatronic systems, software will permeate almost every aspect of Delfi-n3Xt, from ground station to the embedded subsystems and payloads. Therefore, proper procedures and methods should be taken into consideration to provide the desired level of fault tolerance for the nanosatellite. This section provides an overview of the concepts and notions of software fault tolerance from literature survey, in the hope that the methods can be evaluated and those applicable can be implemented.

Because software is not a tangible component within the system, it does not degrade in the same way as hardware does. Errors cause by incorrect bit storage, transmission or processing, are actually faults of the hardware components in charge of these tasks. The only software faults are those that can be introduced during the coding process, and as such, it is important to design a software development methodology that reduces the risk of producing them or create filters that can detect them as early as possible. A proper development cycle takes care of the first two ways to deal with software faults: prevention and removal. For those unexpected faults that occur while the system is running, fault tolerance and input workarounds can help maintain sanity. Our focus here will be in fault tolerance techniques.

3.1.1 Fault Detection Methods

The most direct way to add robustness to software is to add extra modules to a single program or piece of code intended to contain a fault in case it happens. This is known as single-version software fault tolerance. Multi-version fault tolerance considers more than one version of a component with the idea that different designs/algorithms/tools

but built for the same purpose, would not fail in the same manner. Therefore, when one of them fails, another version is unlikely to fail at the same point. This, however, comes at the price of larger code and resource consumption. Among the most popular methods of software fault detection methods are:

- Time-outs

  When a certain process or task has to complete within a certain deadline, this amount of time can serve as an error check. Communication tasks, such as for the Delfi-n3Xt data bus, has to have deadlines to avoid deadlocks and hangs. Processing tasks may also be subject to timeouts, where if a process has not finished its computation within a deadline, this may signify a failure.

- Heartbeats

  A heartbeat is a regular and constant signal that is monitored to ensure that a certain system or part of it is "alive" or operational. The Delfi-n3Xt will contain a mechanism, by which a subsystem (possibly the PTRX) will monitor the central flight computer. If no telecommands requests or packets are received within a certain amount of time, it will assume OBC failure. This will trigger a set of events by which the PTRX will work as a degraded OBC, but will keep the satellite alive. Another example is a watchdog timer, which monitors the OBC and will reboot it in case of a hang.

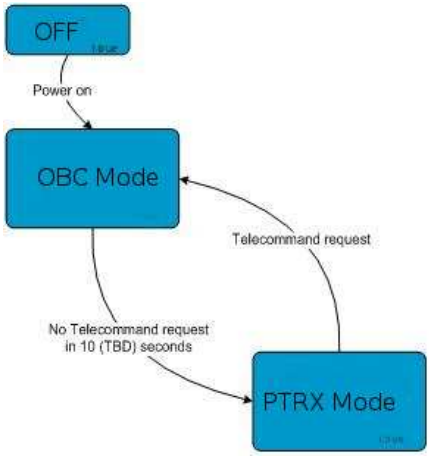


Figure 3.1: Delfi-n3Xt OBC Heartbeat - OBC backup.

Figure 3.1 illustrates this principle with the Delfi-n3Xt OBC. If no requests are detected in 10 seconds (approximately), an OBC backup mode will come to action, most probably from the PTRX node.

- Resource Consumption Delimiters

  It is common than when a software task fails or hangs, the consumption of certain resources will either increase or decrease. Buffer overflows or memory leaks will

create an increasingly amount of requests for memory, which can be detected by a component monitoring memory consumption. The same thing happens when a process suffers a deadlock condition and completely clogs the processor. A crash, on the other hand, may be detected by a sudden decrease in processor usage. Modern operating systems contain provisions of this type.

- Checkpoints

  At certain steps of a process, there are usually certain criteria than can be used to asses the sanity of the current state. At these steps, a specialized monitor can take the responsibility of making sure values and parameters comply with certain criteria. In communications, for example, certain nodes may implement hop counters to check the amount of hops a certain packet has gone through on a network. If the number is unreasonable, measures need to be taken.

- Data integrity

  Complex data structures can be corrupted through a running process, but can be verified for integrity from time to time. Usually, these structures contain redundant or header information that eases the process. For example, linked lists, stacks and/or queues usually contain a counter of elements, which can be verified by running through the list and performing the actual count. One common example is the canary check employed to detect buffer overflows, in which a "canary" value is added to the beginning or the end of buffers in memory. Failure to read this value can be a signal of buffer overflow.

- Error Detection Codes

  To ensure integrity of transmitted data, redundant bits or bytes may be added that serve to asses the correctness of the data packet. The redundant information is generated from the original information segment. When the complete packet is received, the receiver runs the algorithm on the information bytes to generate the rest. If these segments don't match, a transmission error has occurred. Research and theory on these schemes is abundant, and this is discussed in more detail in a later section, along with specific analysis of implementations for the Delfi CDHS.

3.1.2 Fault Management Methods

Measures need to be taken to contain the effects of an error when detected. Recovery mechanisms vary and depend on the graveness of the error. The following are some of the most common patterns:

- Retry

  The simplest of all methods is to repeat the same action once a failure is detected. Of course, this can only be done a limited number of times, since the cause for failure may be permanent and constants retries will hang the system. Usually, this is accompanied by a retry counter, which limits the amount of times the task should be repeated.

- Restart

  A reboot is usually necessary when the system hangs. If the processor, for example, is stuck in one task, no other processes can run. Therefore, an additional component must restart the system and restitute it to startup. This is usually the action performed by a watchdog timer. The drawback of this action is that a complete restart may cause other processes to miss deadlines, or worse, fail completely if they were expecting resources or were performing a delicate task. Therefore, this measure should be implemented with careful analysis.

- Voting

  Voting schemes require the presence of redundant components performing the same task. Upon reception of results from all of these separate components (processes, threads, etc), a voting unit or switch decides which of the outputs should be taken as most viable. The voting unit can be as simple as a majority voter or much more complex, with statistical history checks, statistics or advanced heuristics. The scheme is illustrated in figure 3.2. The price to pay for such a scheme is greater overhead in computation and resources (for each component).

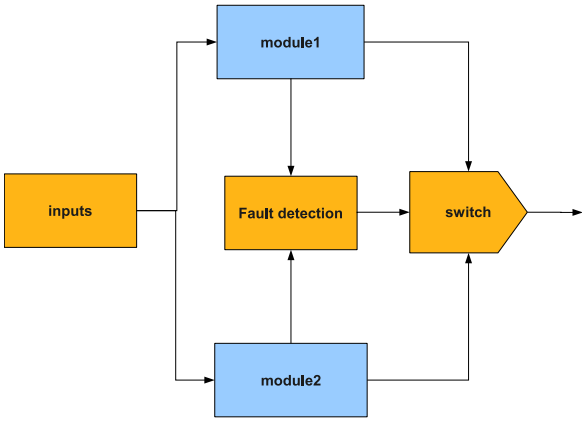


Figure 3.2: Voting Scheme.

- Roll-back & Roll-forward When the system goes into an undesired state, an external component can be set in such a way that in either puts the system back in a previous known state, or to the next safe state. This provides confidence in the recuperation, in the hope that the faulty state will not be encountered again. This mechanism is common in firmware, where states are clearly defined and transition between one and the other is simple.

3.2 Error Detection and Correction Codes

Information theory, the basics of which were developed by Shannon in the mid-1900's, establishes that it is possible to have reliable information transfer over a noisy medium,

provided that the entropy of this medium is lower than it's capacity. Also known, as coding theory, it forms a pivotal part of modern communication systems. Therefore, in this section we start with a very theoretical overview of coding theory and its purpose, and further on we describe techniques and methods than can enhance the Delfi programme's command and data handling subsystem, with emphasis on communications over the data bus.

3.2.1 Noisy Channel Models

The noisy channel model or noisy channel coding theorem is perhaps one of the most fundamental principles of information theory. The theorem states that there is always a way to transmit nearly error-free information over a channel no matter how much contaminated by noise it may be. Formally, it states that for a given channel with capacity C, it is possible to transmit data at a rate R, with $R < C$ with an arbitrarily small degree of error.

According to the theorem, for a probability p_b of bit error, the maximum rate achievable is calculated by:

$$R(p_b) = \frac{C}{1 - H_2(p_b)} \tag{3.1}$$

where H_2 is the binary entropy function, defined as:

$$H_2(x) = x \log \frac{1}{x} + (1 - x) \log \frac{1}{1 - x} \tag{3.2}$$

The channel capacity can be calculated from the physical properties of the medium. For those media bandlimited by Gaussian noise with bandwidth W in hertz, an approximation of its capacity in bits per second is given by:

$$C = W \log_2(1 + \frac{E_s}{N_o}) \tag{3.3}$$

assuming ideal Nyquist sampling, E_s is the average signal (watts) energy and N_o is the noise power present (watts).

The most common type of channels treated in information theory are the binary symmetric channel (BSC) and the binary erasure channels (BEC). They both describe a certain type of behavior caused by the noise that contaminates the medium. The BSC is depicted in figure 3.3 and models a medium in which a transmitted bit can be changed by a probability of p.

The capacity of the BSC is obtained from the equations above and reduces to:

$$C_{BSC} = 1 - H_b(p) \tag{3.4}$$

The other model is the binary erasure channel (BEC) that models a medium by which the noise can simply erase a bit an transmit an *erasure*, symbolized by e.

The capacity of the BEC model is

$$C_{BEC} = 1 - p \tag{3.5}$$

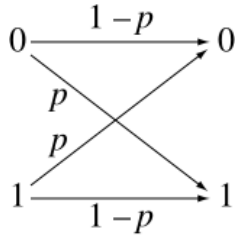


Figure 3.3: Binary Symmetric Channel.

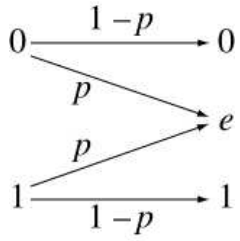


Figure 3.4: Binary Erasure Channel.

3.2.2 Basic Notions of Coding Theory

Binary messages composed of n bits can form at most 2^n different messages or words. If k of those n bits are actual information, then the total number of different information words that can be transmitted is 2^k. Coding schemes are therefore labeled also as (n, k) for this purpose. The objective of any coding method is to evenly spread the 2^k messages as evenly as possible among the 2^n possibilities in such a way that a small disturbance in the bits still leaves the erroneous message near one of the valid codewords, thus, enable correction.

The hamming distance d_{min} is the minimum number of bits that must be changed to convert one codeword into another valid codeword. This distance it was makes a certain coding scheme more or less resilient to disturbances from the channel. If a coding method wishes to detect t errors, it must comply with the following relationship:

$$d_{min} > t \tag{3.6}$$

Sometimes in literature, coding schemes are referred to as (n, k, t). The relationship between these three quantities can be obtained by asking the question on how many extra bits are needed to produce a t-error correcting code. In an n-bit message, the number of different permutations of d bits is given by:

$$\frac{n!}{d!(n-d)!} \tag{3.7}$$

and thus, for a coding scheme that corrects t errors, this relationship can be deduced:

$$\sum_{d=0}^{t} \frac{n!}{d!(n-d)!} \leq 2^{n-k} \tag{3.8}$$

When both sides of the relationship are equal, then the code is called a *perfect code*. Moreover, it is also useful to know the ratio of actual data bits to the total number of bits transmitted, known as the code rate:

$$R = \frac{k}{n} \tag{3.9}$$

and the actual bit overhead, as expected, is the amount of redundant bits in relation to the information bits:

$$B_o = \frac{t}{k} \tag{3.10}$$

3.2.3 Coding Schemes for the Delfi Nanosatellites

Development done for the Delfi nanosatellite programme has to take into consideration the limited resources available on board. For embedded networks, we take into consideration parity bits, some special checksums and cyclic redundancy checks, since their computational requirements fit reasonably within the bounds posed by the systems on board.

3.2.3.1 Parity Bits

One of the simplest error detection methods is the parity bit check. This is an extra bit added to a message that verifies the amount of 1's in a string of bits. If the extra parity bit is 1 when there is an odd amount of 1's in the data (to make it even), it's called even parity, and odd parity when not. For this simple mechanism, $d_{min} = 2$ because one single bit change will render the code invalid, however, two changes will render a valid codeword. An illustration on how parity works for data transmissions is table 3.2.3.1.

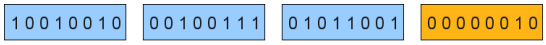
7-bit data	even parity	odd parity
0000001(1)	00000011	00000010
0110101(53)	01101010	01101011
1000001(65)	10000010	10000011
0111000(56)	01110001	01110000

Table 3.1: Parity Scheme Example.

Clearly, simple parity cannot correct errors, only detect them. In memories, where higher dimensional parity checks are employed, the scheme may be able to correct errors. In data transmissions as those considered to the Delfi nanosatellites, this does not apply, however. Parity checks are usually used in serial communications such as RS232, PCI and SCSI buses.

3.2.3.2 Parity Bits Implementation on Delfi-C^3

In Delfi-C^3, there was one parity bit for every data byte. Therefore, the parity bits would be grouped in additional bytes on the packet. For this particular implementation, the even parity check is grouped at the end of the transmission as illustrated in figure 3.5, where the orange blocks represent the parity bytes, on which every bit represents the parity bit for the preceding data bytes, starting from right and shifting left.



(a) 8 bytes (or less) transmitted

(b) More than 8 bytes

Figure 3.5: Parity Error Detection Scheme for Delfi-n3Xt

To determine the actual code rate and bit overhead, it is important to note that the data bus protocol imposes the restriction that data can only be exchanged in the form of bytes. Therefore, when the message length is 1-8 bytes long, there will be an extra byte for parity, for lengths of 9-16, 2 extra bytes, etc. The analysis is depicted in table 3.2.

Data bytes	Lower Bound	Upper Bound
1-8	8/16=0.5	64/72=0.89
9-16	72/88=0.82	128/144=0.89
17-24	136/160=0.85	192/216=0.89

Table 3.2: Code Rate for Parity Scheme.

thus indicating that the worst possible code rate is 50% and at most, the scheme will yield an 89%. The bit overhead can also be analyzed in the same way:

Data bytes	Lower Bound	Upper Bound
1-8	8/8=1	8/64=0.125
9-16	16/72=0.22	16/128=0.125
17-24	24/136=0.176	24/192=0.125

Table 3.3: Bit Overhead for Parity Scheme.

revealing that at best, the implemented parity scheme will yield 12.5% bit overhead, and the worst case will be of 100%.

3.2.3.3 Cyclic Redundancy Checks (CRC)

A cyclic redundancy check (CRC) is a checksum algorithm popular for its error detection capabilities and used in a varied number of applications, such as Ethernet, Controller Area Network, Flexray, the PNG image format and ZIP compression among others. The

CRC coding algorithm adds a short, fixed-length bit sequence to a message block and packs it together to be transmitted. When the receiver gets the packet, it will perform the algorithm on the data bytes and obtain the CRC for it. If the calculated CRC does not match the CRC appended to the message, the receiver assumes there has been a transmission error. Unlike parity bits, CRC's are more effective to handle "burst" errors, that is, errors that occur in a continuous sequence of bits.

The algorithm is based on polynomial division and a complete explanation of the theory behind the method requires an overview of polynomial algebra and group theory. A thorough explanation of the theory goes beyond the scope of this work, but we shall, however, discuss the basic process of calculating a CRC.

First, to each possible n-bit binary message, we associate a polynomial $u(x)$ so that each bit position (p_i) is a coefficient:

$$u(x) = \sum_{i=0}^{n} p_i x^i \tag{3.11}$$

The following table contains examples of this representation 3.2.3.3.

8-bit message	Associated Polynomial
00101011	$x^5 + x^3 + x + 1$
11010000	$x^7 + x^6 + x^4$
00000101	$x^2 + 1$

Table 3.4: Polynomial Encoding Examples.

The purpose of the CRC algorithm is to form a polynomial in such a way that it is a multiple of another polynomial $g(x)$, called the *generator polynomial*. When the receiver obtains a block of data, it will then verify that the block of data is in fact a multiple of $g(x)$, discarding the message if it isn't. Suppose that $u(x)$ is associated with a k-bit message and $v(x)$ with the n-message codeword. The relationship with the generator polynomial $g(x)$ is given by

$$\begin{aligned} v(x) &= u(x)x^{n-k} + s(x) \\ v(x) &= a(x)g(x) \end{aligned} \tag{3.12}$$

where $s(x)$ is the remainder of the division of $u(x)x^{n-k}$ by $g(x)$ and $v(x)$ is a multiple. The message bits are represented by $u(x)x^{n-k}$ and the coefficients of $s(x)$ represent the CRC checksum that is appended.

As an example, consider the message 110101 represented by $u(x) = x^5 + x^4 + x^2 + 1$ and the generator polynomial $g(x) = x^{16} + x^{15} + x^2 + 1$. Then,

$$\begin{aligned} x^{n-k}u(x) &= x^{16}(x^5 + x^4 + x^2 + 1) \\ &= x^{21} + x^{20} + x^{18} + x^{16} \end{aligned}$$

and when dividing $x^{n-k}u(x)$ by $g(x)$, the remainder obtained is $s(x) = x^7 + x^5 + x^4 + x^3 + x^2 + x$. The complete codeword therefore becomes 110101 0111110100000000. It is obvious that both the sender and receiver have to know the generator polynomial

beforehand. Some common polynomials in use in several applications today are depicted in table 3.2.3.3.

Name	Polynomial
CRC-16	$x^{16} + x^{15} + x^2 + 1$
ETHERNET	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$
CAN	$x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$
CCIT-IBM	$x^{16} + x^{12} + x^5 + 1$
CRC-8	$x^8 + x^7 + x^6 + x^4 + x^2 + 1$

Table 3.5: Commonly used CRC Polynomials.

Hardware implementations for the CRC will not be discussed here in detail because the implementation for Delfi-n3Xt is software based. Suffice it to say, however, that a hardware implementation is simple and consists of a shift register with XOR gates placed at the precise positions of the coefficients of $g(x)$. In software, the most direct implementation is the bit-by-bit on the fly calculation. Another option that reduces the computational load is a table-lookup algorithm. With 8-bit CRC's the table is 256 entries long, but for larger sizes this table may be unfeasably large. For longer polynomial sizes, a reduced table algorithm may be used as mentioned in [25], in which only part of the precomputed values are stored and each byte is shifted over the polynomial.

3.2.3.4 CRC Implementation for Delfi-n3Xt

There are several considerations that need to be taken into account when opting for the CRC algorithm. First of all, it is important to consider the length of the generator polynomial. Whereas modern computer system have large word sizes (32 bits or more), most embedded systems, such as those on board Delfi-n3Xt, have short word sizes. Delfi-n3Xt will carry on board the MSP430F1612 microcontrollers from Texas Instruments, which support data types of up to 16-bits. Furthermore, the longer the generator polynomial, the more secure it is in terms of error detection, but will take more time to calculate (both for the sender and receiver). Finally, not all generator polynomials are equally effective on their error detection capabilities. Philip Koopman from Carnegie Mellon University in [26] and [11], along with Castagnoli in [3] have explored several polynomials and studied their effectiveness.

The measure of the effectiveness of a polynomial is it's hamming distance (d_{min}), as explained in previous sections, which is the minimum number of bit errors present to potentially go undetected. For instance, d_{min}=4 means that the code will detect all 3-bit errors, but may be unable to detect 4 or more bits in error. Moreover, as discussed in [11], the effectiveness also depends on the size of the message; a polynomial that works well for certain data sizes, may not necessarily preserve its effectiveness for other sizes.

Figure 3.6 shows the effectiveness of some 8-bit CRC's. The dark line represents the bound of an ideal polynomial at each data size. The y-axis represents the probability of undetected error (P_{ud}).. As can be observed, the the effectiveness varies with the chosen polynomial and data size. [11] cites the case of 0xEA, for example, shows that from

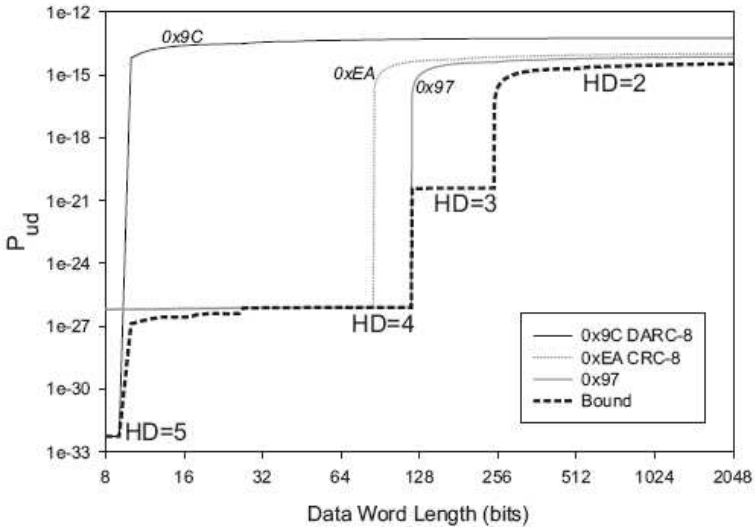


Figure 3.6: Effectiveness of 8-bit CRC's, adapted from [11].

data sizes of 86 to 119, it has an HD=4, but above 119, the effectiveness drops to HD=2 with a gradually increasing probability of undetected errors. Table 3.7 shows the most effective polynomials for given hamming distances.

Max length at HD / *Polynomial*	CRC Size (bits) 3	4	5	6	7	8	9	10	11	12	13	14	15	16
HD=2	2048+ 0x5	2048+ 0x9	2048+ 0x12	2048+ 0x21	2048+ 0x48	2048+ 0xA6	2048+ 0x167	2048+ 0x327	2048+ 0x64D	–	–	–	–	–
HD=3		11 0x9	26 0x12	57 0x21	120 0x48	247 0xA6	502 0x167	1013 0x327	2036 0x64D	2048 0xB75	–	–	–	–
HD=4			10 0x15	25 0x2C	56 0x5B	119 0x97	246 0x14B	501 0x319	1012 0x583	2035 0xC07	2048 0x102A	2048 0x21E8	2048 0x4976	2048 0xBAAD
HD=5						9 0x9C	13 0x185	21 0x2B9	25 0x5D7	53 0x8F8	none	113 0x212D	136 0x6A8D	241 0xAC9A
HD=6							8 0x13C	12 0x28E	22 0x532	27 0xB41	52 0x1909	57 0x372B	114 0x573A	135 0xC86C
HD=7									12 0x571	none	12 0x12A5	13 0x28A9	16 0x5BD5	19 0x968B
HD=8										11 0xA4F	11 0x10B7	11 0x2371	12 0x630B	15 0x8FDB

Figure 3.7: Best polynomials for given CRC sizes and Hamming Distance, adapted from [11].

The two numbers in each cell of table 3.7 represent the polynomial (bottom) and the data size up to which the HD holds (top). For Delfi-n3Xt, which handles data in the data bus in bytes, an 8-bit CRC is ideal. The table shows 0x9C with the highest HD (HD=5) at this size up to data sizes of 9 bits. Given the fact that most of the data transfers within the satellite will be short, this polynomial is the best choice. The data transfers will have the appended CRC byte at the end, as illustrated in figure 3.8. For the Delfi-n3Xt, the implementation for CRC is table-based, as it decreases computation

overhead considerably at the price of a 256 byte table in memory. Memory within the microcontroller, however, is not of primordial importance, as most of the mission data will be stored in external memory.

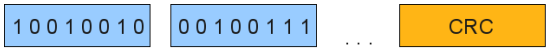


Figure 3.8: CRC implementation for Delfi CDHS.

There is no fixed value for code rate or bit overhead in this implementation, since the message length may vary. Of course, as the message size increases, the less bit overhead and the greater the code rate.

3.2.3.5 Checksums

Although not all hash functions can be considered part of coding theory, they do have error detection capabilities. In this work we consider two checksums algorithms: the Pearson hash and the Fletcher-8.

Algorithm 1 8-bit fletcher checksum

$sum1 \Leftarrow 0$,sum2$\Leftarrow 0$
for i to $message\_length$ **do**
  $sum1 \leftarrow sum1 + message[i]$
  $sum2 \leftarrow sum2 + sum1$
end for
return $sum2$ & 0xFF

Algorithm 2 Pearson checksum

$h[0] \leftarrow 0$
for i to $message\_length$ **do**
  $index \leftarrow h[i-1] \oplus message[i]$
  $h[i] \leftarrow T[index]$
end for
return $h[message\_lenght]$

When compared to CRC's, these hash functions require less computation and are therefore opted by in embedded applications. According to [15], the Fletcher checksum provides slightly better results than the popular Adler algorithm and is easier to compute. This is the reason reason why it is considered as part of this work. Unlike Adler, as well, it is able to detect insertion or deletion of zeroes, reordering and incrementing/decrementing bytes at any end. The pseudocode for the 8-bit fletcher algorithm is shown in listing 1.

The Pearson hash, proposed in [23], is specially designed to be efficient with 8-bit computers, it is easy to implement, and is heavily dependent in all the data bytes of the message. The drawback however, is that it requires a table (T) of 256 pseudorandom

values specifically chosen to be a *perfect* and *minimal*, that is, a contiguous set of integers with no holes and no collisions. The algorithm is illustrated in listing 2.

Again, as with the CRC scheme, code rate will increase with longer messages and bit overhead will tend to decrease.

3.2.4 Tests and Measurements

Implementations of the algorithms described in the previous section were coded and tested along with the Delfi-n3Xt service layer. The algorithms were programmed in the C language with the IAR Workbench Tool provided by Texas Instruments. The test setup consists of a host computer connected via a JTAG interface to the master MSP430F169 microcontroller, which will generate data and transfer it via a 100 Kbit I^2C bus to a slave MSP430F169, who will, in turn, echo the data. Of course, both processors have been programmed with the error detection scheme in test. The JTAG interface eases the debugging and programming of the setup, as well as live monitoring of memory contents and data execution. Figure 3.9 illustrates the setup.

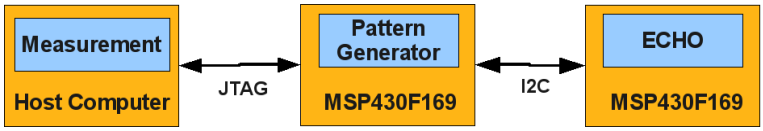


Figure 3.9: Test Environment for Error Detection Codes.

Figure 3.10 shows the performance of the algorithms in terms of execution time overhead in the MSP430.

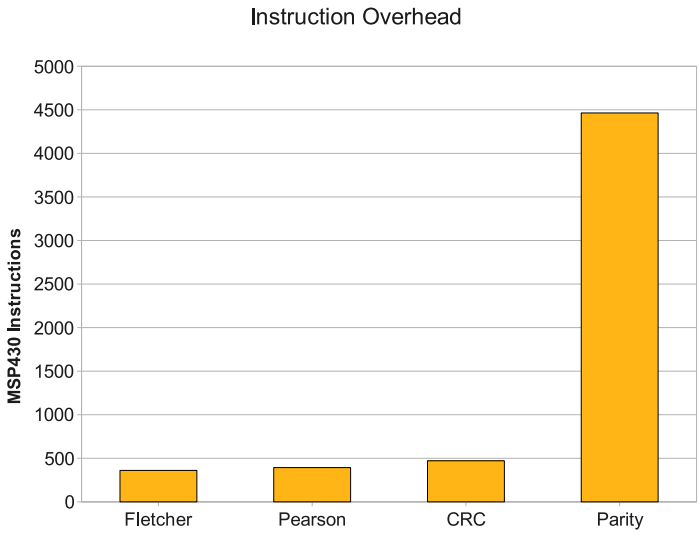


Figure 3.10: EDAC Instructions Overhead for Delfi-n3Xt.

The least overhead was the fletcher checksum (361 instructions), followed closely by

the Pearson hash (393). Both algorithms are quite simple to implement and compute. Cyclic redundancy checks can take up to double the amount of computation overhead as a price for their superior error detection capabilities, but the table-lookup implementation here has an enormous benefit, since the overhead increase when compared to the previous two algorithms is relatively very small (471). And as stated in [15], CRC's have a greater benefits (overhead/error detection tradeoff) than other checksum algorithms for shorter codeword lengths.

Surprisingly, the parity scheme, as implemented for Delfi-C3, has enormous costs. The explanation for this is rather simple. While the other three algorithms add only one more byte to the data packet in each transmission, no matter the number of data bytes, the parity scheme will add a variable number of additional bytes. Thus, at least to know the amount of extra bytes that will be added, divisions have to be performed; divisions being one of the most expensive operations in terms of CPU time. In the implemented code, at least two divisions have to be performed for every transmission. Moreover, byte/bit ordering has to be performed to correctly pack the data frame.

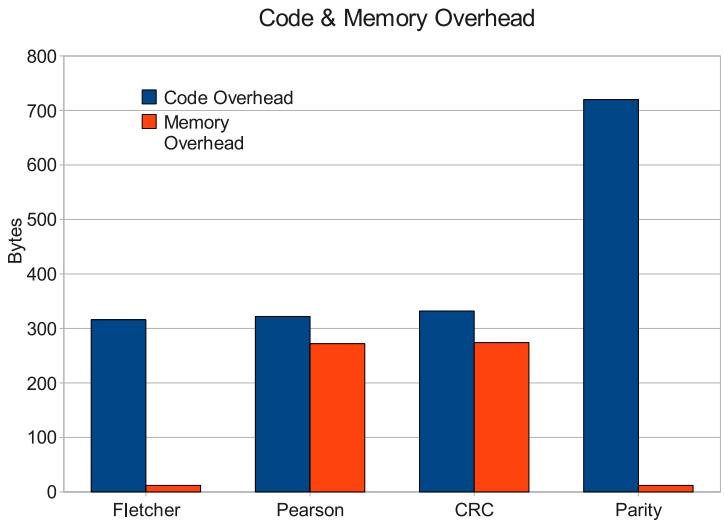


Figure 3.11: EDAC Data/Code Overhead for Delfi-n3Xt.

Figure 3.11 shows the overhead of each EDAC scheme in terms of code and data (memory) size. The results are more or less expected. Fletcher, Pearson and CRC's have about the same code overhead, and such is the result of the way they are implemented. Parity, again, shows high code size, caused mostly by the code to order and shift bytes/bits. Since both Pearson and CRC's work with tables of precomputed values, their data overhead is higher than the rest, but this is compensated by the benefits of in execution time.

3.3 Summary

This chapter presented an overview of the software techniques that can enhance the reliability of the data bus on board the satellites of the Delfi programme. For high level software, techniques to detect errors include heartbeats, resource checkers, timeouts, data integrity checks and error correcting codes. To recover from errors, methods to be used can be retries, restarts, voting schemes and rollbacks.

We also examined some error detection schemes popular for embedded networks, and suited for the type of systems on-board nanosatellites. The analysis shows that the parity scheme as implemented for Delfi-C^3 greatly increased the overhead in terms of execution time and code size. Coupled with the deficiencies discussed in the previous chapter, it is a considerable source of problems for the communications within the data bus. Overall, the CRC, implemented with a precomputed table of values can greatly increase the error detection (up to 5 errors) for data sizes well within the range of Delfi-n3Xt, without considerable additional computation time. Since external memory will be used for higher level mission tasks, internal microcontroller memory usage will not interfere with such data requirements.

4 Model Based Fault Detection

4.1 Introduction and Problem Statement

In the previous chapter, we explored techniques that could be applied in software or hardware to the CDHS for proper fault detection. In this chapter, we decide explore fault detection on a different level. Model based fault detection works with a mathematical model of the system we want to monitor and procures the detection of deviations from expected behaviour via this model.

Therefore, supervision functions are built into modern embedded systems and with the advance of the digital era, software plays an increasingly important role in the implementation of these functions. A very illustrative example of this is the fly-by-wire navigation system included in modern commercial aircraft, where reliability, fault detection and tolerance must be built to ensure the safety of passengers and crew. In satellite navigation systems, such as the Delfi-n3Xt attitude determination and control systems, the techniques to detect a fault can provide the means to maintain proper attitude control even in the presence of such failures in sensors or related components.

In this chapter, we will explore a fault detection method based on a variant of the Kalman filter, which falls into the category of state estimators and state observers of figure 4.3. The specific variant of the filter studied here is a relatively recent idea called the Unscented Kalman Filter (UKF). However, to understand the UKF, we first have to explore the traditional method, discussed in detail in a later section. Finally, an implementation of the UKF is included and applied to the set of sensors on board the Delfi-n3Xt nanosatellite simulation environment to determine the occurrence of faults.

4.1.1 Basic Notions

Before engaging into the topics that concern this section of the thesis, it is appropriate to introduce concepts and definitions related to the matter in discussion. Although there is no standard definitions throughout the literature, the pertinent notions, concepts and ideas are described here as an introduction to the reader.

A *fault* can be defined as the state of a system in which one of its properties deviates considerably from a normal or standard behavior. The qualification of the fault is the difference between the standard and abnormal output[7]. The causes for such an event may range from manufacturing errors, hardware/software errors, wrong operation, poor assembly, etc. The buildup or prolonged presence of a fault may derive into malfunctions or failures.

A *malfunction* is a condition where a system temporarily loses its ability to perform expected functions. This may be caused by the presence of one or more faults. A *failure*, moreover, is a permanent state in which the system fails to perform its tasks as expected.

One example of such a state is the malfunctions and failures on board the Hubble space telescope (HST), which among others, included computer reboots after electrical power outages interrupted the operations of it's data formatter, in charge of sending back observed data to Earth, and the failure of its Advanced Camera for Surveys (ACS) in 2007.

In [22], Parhami suggests the model illustrated in figure 4.1. From an ideal state, a system can become defective and the exposed defects produce the faults of the system. As faults accumulate, it will contaminate the processes and manifest as errors, depending on the fault tolerance built into the system. When an error, or the accumulation of such, hinder the ability to carry out the specified task within certain bounds of acceptability, it is considered to be malfunctioning. This does not necessarily mean a catastrophic loss of operation, but may be a reason to turn the operation into a degraded mode, where the lowering of certain levels of operation is accepted. If the system is unable to operate permanently it is considered to be failed.

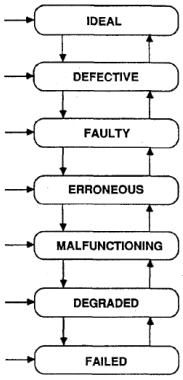
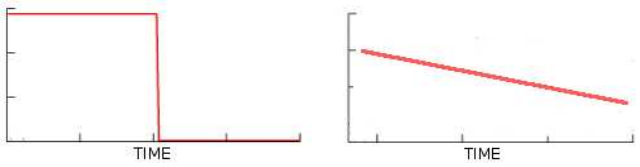


Figure 4.1: System State Model, adapted from [22].

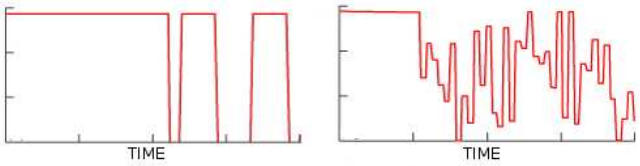
Faults can be classified according to their behaviour, in which case [16] and [6] identify four types of faults. Figure 4.2a shows a *hard failure* or *jump fault*, where the component or sensor under observation has a drastic and permanent jump in its expected behaviour. One example of this can be the occurence of a short or open circuit. A *drift fault* can be visualized in figure 4.2b, which manifests as a gradual degradation of the component, and can occur, for example in the case of thermal or resistance degradation. An *intermitent fault*, in figure 4.2c is understood as sudden jumps between one state and another, alternately functioning and not functioning as expected. A lose connector may cause a fault of this type. Finally, changes in the noise model of a component can also be considered a type of fault, for which the behaviour is illustrated in figure 4.2d, and can be caused, for instance, by a cracked solder joint.

4.1.2 Detection and Diagnosis

Classification of Fault Detection methods can be classified as shown in figure 4.3. The simplest of all schemes are the limit checking and trend checking methods, where thresholds are established for certain parameters and alarms are generated when the threshold



(a) Hard/Jump Fault (b) Drift Fault



(c) Intermitent Fault (d) Change in Noise Model

Figure 4.2: Type of Faults, adapted from [16]

is reached or surpassed. More modern methods can be classified into detection via signal models and process models. Signal models rely on techniques for digital or continuous signal processing, such as spectrum analysis, Fourier transforms or wavelets. Process models try to model and estimate the performance of a process and compare with the actual output. Methods such as neural networks can be included here as they are useful in function approximation, regression analysis and classification. Multi-variant data analysis considers multiple parameters in its estimations, such as Principal Component Analysis (PCA), which identifies the most important gradients (variability) in multiple dimensions from a set of data points.

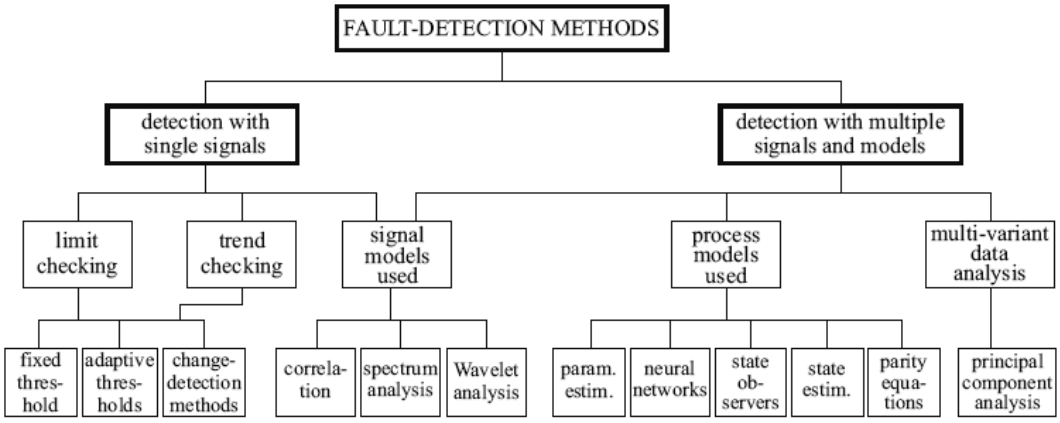


Figure 4.3: Fault Detection Methods, adapted from [7]

More complex than detection is the diagnosis of the fault. The reason for this is the fact that most embedded systems are highly coupled, meaning that the contribution of each component to the output is not linear, and therefore identification of the failing part is nontrivial. Moreover, the behavior may not be time-invariant. Therefore the

techniques employed for isolation or identification involve highly statistical techniques or artificial intelligence methods (neural networks and fuzzy logic). Figure 4.4 shows a classification tree of this methods.

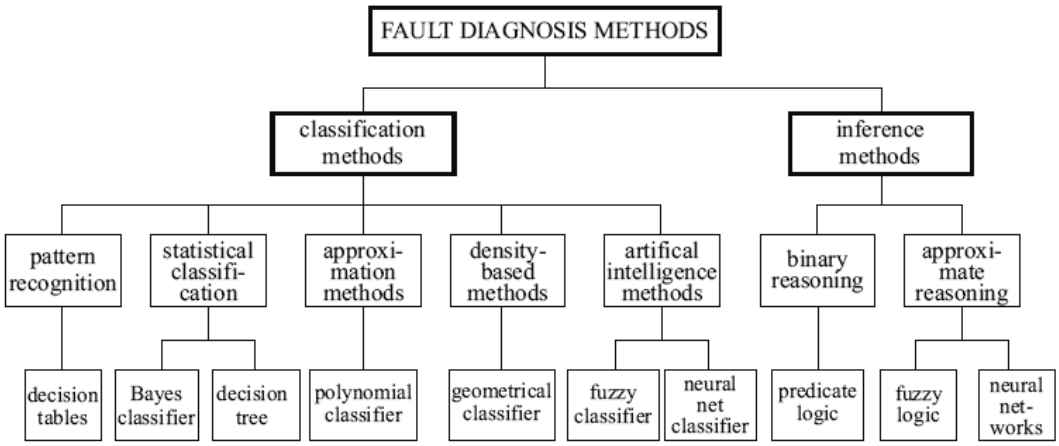


Figure 4.4: Fault Diagnosis Methods, adapted from [7].

Once a fault has been detected and isolated, procedures have to be taken to ensure safety or the system and operators. The measures depend on the urgency and the purpose of the system, but can range from a simple notification to the operator(s) to a complete shutdown of the system. Apart from those two extremes, two important techniques are static (Figure 4.5) and dynamic redundancy (Figure 4.6). Static redundancy describes a system in which several redundant components (sensors, actuators, etc) are included to perform the same task, but the actual output is chose mainly by a voter. This voter can be as simple as an majority voter, use statistical data to chose the most reliable reading or optimize it's inputs to generate a reliable signal[31]. The q-method included in the Delfi-n3Xt ADCS that combines readings from the sun sensors and magnetometer is an example of such a voter[14]. Dynamic redundancy, on the other hand, also includes redundant components, but will use only one of them at a time. Upon the onset or detection of a fault, a switch is activated to output the right readings or signals.

4.2 Related Work

The purpose to this section is to review recent applications of fault detection and isolation techniques and variants of traditional methods. The literature of applications for FDI is extensive, and the works mentioned here are included to provide some background on the recent works being done in the field, but do not represent the whole range of applications and variations available.

As described in the introduction, FDI systems can range from simple threshold checking methods to complex model and signal based systems. A simple voter-based redundancy FDI system is studied in [1] for a steer-by-wire (SBW) system for road-wheel control in a ground vehicle. The steering wheel has attached sensor to it and both front wheels have an road-wheel-angle (RWA) sensor. The mathematical model described in

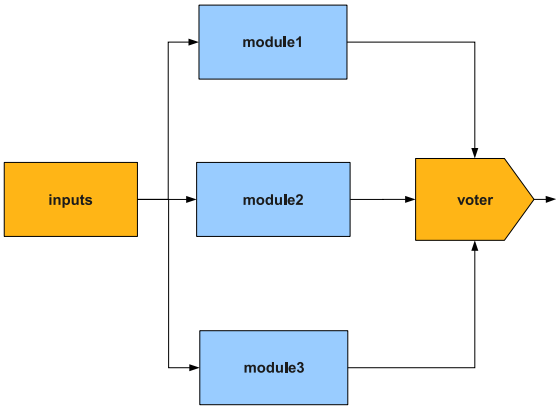


Figure 4.5: Static Redundancy.

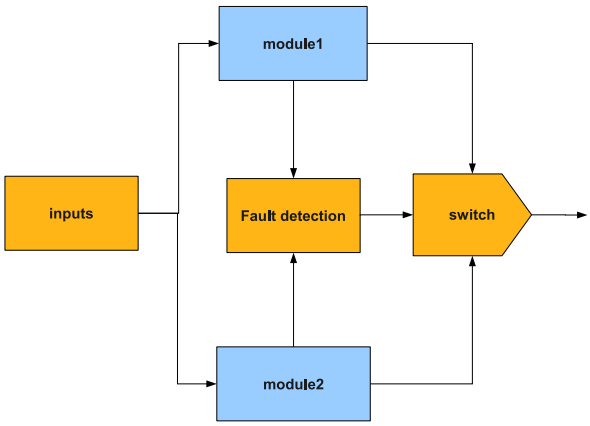


Figure 4.6: Dynamic Redundancy.

[1] shows the relationship between the three signals. The input from these three sensors is taken into account by a majority voter to decide the proper output of the system. A first check is performed directly from the outputs to check that values from the components fall within accepted values (i.e. an angle of 90 degrees for an RWA is invalid). Further on, fault isolation is performed by comparing the differences between each signal with the others. In the presence of a fault, the common signal that causes differences beyond an accepted threshold is marked as erroneous. Moreover, the fault has to persist for a certain amount of (tunable) time for it to be marked. The correct output is assumed to be the average of all incoming signals. The basic Simulink FDI algorithm is shown in figure 4.7.

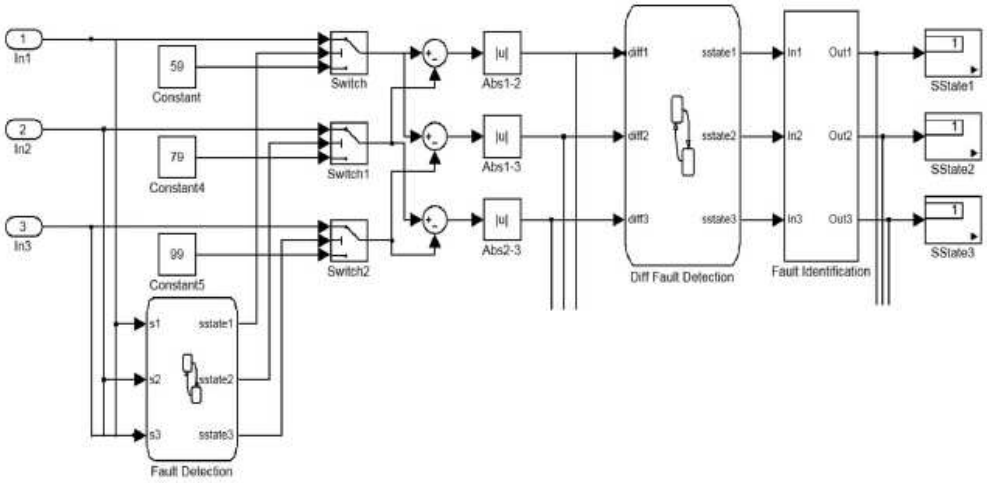


Figure 4.7: FDI Algorithm for Road-Wheel Control Subsystem, adapted from [1].

Simulation results of this scheme show that the algorithm can correctly detect and isolate a persistent stuck-at fault in one sensor. When this happens, the damaged component is shut off and no longer taken into account for nominal operations. The correct output is assumed to be the average of the remaining two sensors. As expected, a fault in two or more sensors cannot be isolated and therefore all signals are considered faulty in that case.

More related to the work presented in this thesis is presented in [32], where an FDI system is implemented for satellite navigation. The implementation, however, considers only gyroscopes and an earth sensor. The strategy for fault detection implemented is highly mathematical and tries to decouple unknown disturbances in navigation with the residue as much as possible using singular value decomposition. The scheme also procures to maximize the norm of the transfer matrix between sensors and residue in order to make it more sensible to sensor disturbances. Moreover, multiple relations of this kind, maximizing the norm between one sensor, while minimizing it for the rest, provide an easy way to isolate erroneous behavior from components. Implementation and simulation in Matlab/Simulink (Fig. 4.8) was carried out with stuck-up faults in one or more of the sensors and actuators.

Realistic noise levels are added to the simulation and disturbances that go beyond the acceptable threshold are flagged as errors if they persist for more than 10 samples. This system, on board the Indian Remote Sensing Spacecraft, is able to track sensor faults, both from gyros and earth sensor, and isolate them correctly, however, actuator faults did not cause enough disturbances to be perceived.

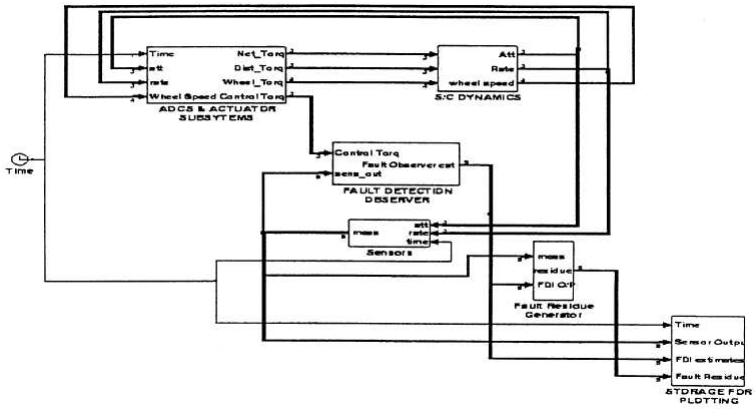


Figure 4.8: FDI Model for Gyro Fault Detection, adapted from [17].

In [17] the application of a Kalman based FDI system for unmanned ground vehicles is studied. The vehicle is equipped with a color camera, a laser range sensor, and electronic compass, a Garmin GPS receiver, wireless Ethernet link, a Crossbow IMU all interfaced to a main board computer consisting of a Pentium IV running at 3.2 Ghz. For fault detection and isolation, the only sensors taken into account are the odometers, the IMU and the GPS receiver. Data from these sensors is filtered through a java/Matlab based

component that performs online residual calculations and signals faults when perceived. Moreover, the module has the ability to switch on and off any component outputting faulty data. When additive faults are injected into the system, results showed that failures were detected and isolated properly. However, when multiple failures occurred, isolation was not always possible.

The LEO satellite ADCS introduced in [19] describes a fault detection strategy based on the extended Kalman filter, but with slight differences. EKF iterations are used for state estimation, faults are not determined via residuals, but by checking the spectral norm of the innovation matrix. The spectral norm is defined as the squared root of the maximum eigenvalue, and the properties of this value allow the determination of abnormal behavior. The authors derive a set of minimum and maximum bounds for the spectral norm, and any breach of these bounds is taken as the onset of a fault. The few simulations presented in this work show a correct behavior of the system in the presence of shift faults (faults where the output of a component is shifted by certain value).

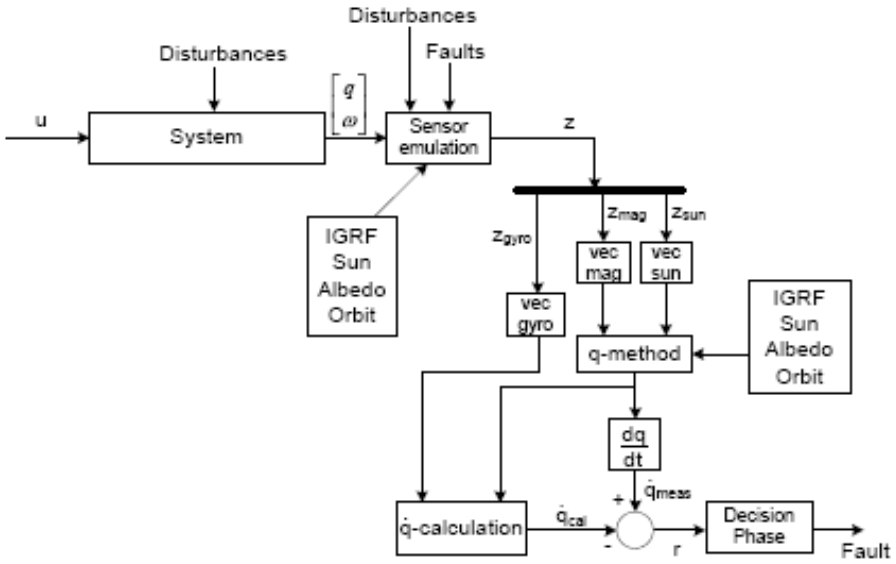


Figure 4.9: AAUSAT FDI Model, adapted from [14].

Finally it is worth mentioning the FDI system proposed for the AAUSAT-II satellite from Aalborg University in Denmark[14], as the model and environment used for the AAUSAT is also the basis for the studies performed in this chapter of the thesis. The system, shown in figure 4.9 will procure detection of faults in gyro measurements. Also included are magnetometers and sun sensors, which are further processed with an optimization block labeled q-method, that calculates the most likely reading from both components.

The decision phase block will compare model estimates with actual readings and signal a fault when a certain threshold value has been surpassed. A stuck-at-0 fault introduced in one of the gyros is correctly indicated by the decision block, but an open-circuit, simulated by introduction of mean-valued noise of 2.5V fails to be detected.

Finally, a short-circuit, treated as a 5V output, is correctly detected when injected.

4.3 Fault Detection via Kalman Filters

As described in the previous section, Kalman filters and its variants play an important role in research of FDI schemes. In this section we introduce the traditional Kalman filter and gradually explain how it can be modified to deal with nonlinear processes. We conclude the section describing the variant that will be implemented for the Delfi-n3Xt simulation environment.

The Kalman filter, or lineal quadratic estimation (LQE), is a recursive mathematical procedure to estimate variables and states in linear processes. In its natural form, it is applicable to linear dynamic systems for state and observation prediction, computer vision, and signal processing. In practice, it provides efficient results and is theoretically appealing because this is the filter that achieves the greatest minimization in the variance of the estimation error. In each recursion, the filter updates its estimation with the mean and covariance of the state, which, although may seem a simple representation of the system, it suffices for most operational activities. Yet with all these benefits, there is one important disadvantage: the standard version of the filter works only for linear processes.

To illustrate how the standard Kalman filter works, consider the following simple dynamic system:

$$\begin{aligned} x_k &= Ax_{k-1} + w_k \\ y_k &= Cx_k + v_k \end{aligned} \tag{4.1}$$

where x_k is the state vector of the process at time k, A is a state transition matrix, y_k is the observation at time k and C is the transition matrix between the state and measurement. Finally, w_k and v_k are Gaussian white noise with known covariance. The two noise models have the following associated covariances:

$$\begin{aligned} Q &= [w_k w_k^T] \\ R &= [v_k v_k^T] \end{aligned}$$

From these basic equations, the filter estimates the new state with the following equation:

$$\hat{x}_k = \hat{x}'_k + K_k(y_k - C\hat{x}'_k) \tag{4.2}$$

K is called the Kalman gain and serves to accommodate the data according to the reliability of the measurements. If the measurement noise is large, K will be small so that the measurement y_k is not given much weight when estimating the next state $\hat{x}_k$. If the measurement noise is small, then K will give more credibility to y_k for the next estimation. This gain is calculated as follows:

$$K_k = P'_k C^T (C P'_k C^T + R)^{-1} \tag{4.3}$$

where P is the current covariance matrix. The update for the covariance matrix is given by 4.4.

$$P_k = (I - K_k C)P'_k \tag{4.4}$$

Finally, the projection to next step is given by

$$\begin{aligned} \hat{x}'_{k+1} &= A\hat{x}_k \\ P_{k+1} &= AP_kA^T + Q \end{aligned} \tag{4.5}$$

The Kalman recursive algorithm is summarized in the following figure 4.10.

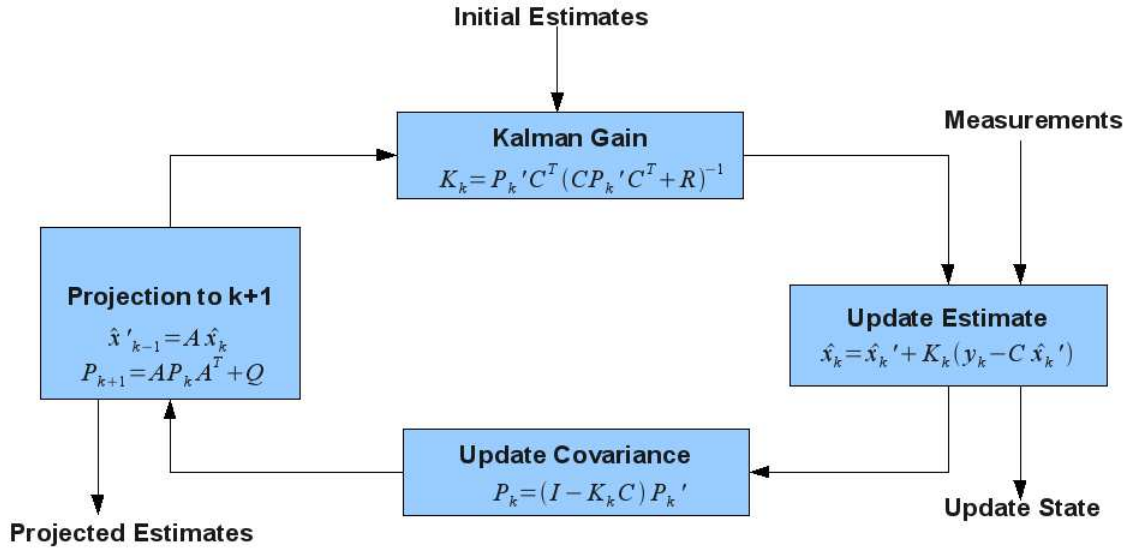


Figure 4.10: The Kalman Filter Algorithm.

An initial covariance matrix P_k has to be given as input to start the calculations. The derivation of the Kalman filter will not be discussed here, but suffice it to say that it comes from minimizing the mean squared error (MSE) of the state-space equations shown before (equation 4.1). Another alternative is to derive the solution as a chi-square merit function, which is a maximum likelihood function. For a complete derivation using both alternatives, see [30].

Complex systems in use today, however, are usually non-linear. Therefore, to implement model based state estimation with LQE, the problem has to be modified. One option is to linearize the problem, the basis for the Extended Kalman Filter (EKF), which is considered the standard filter for non-linear estimation. Linearization theory is vast and we shall not engage into its discussion here, but we shall mention that the idea behind the EKF is to apply a Jacobian linearization about the current mean and covariance. Another, newer idea, is to perform the non-linear transformation to a deterministically selected set of points. The method is the Unscented Transform, basis of the Unscented Kalman Filter (UKF), which we shall explore further in the next section.

4.3.1 The Unscented Transform

The Unscented transform is inspired by the idea that it is simpler to estimate a probability distribution than an arbitrary non-linear function. The basic idea is shown in figure 4.11, where a set of properly chosen points, also called sigma points, are transformed via the nonlinear function into a new set of points. Knowing the covariance and mean of the sigma points and then calculating again for the transformed set, provides a way to estimate how these moments are transformed by the system.

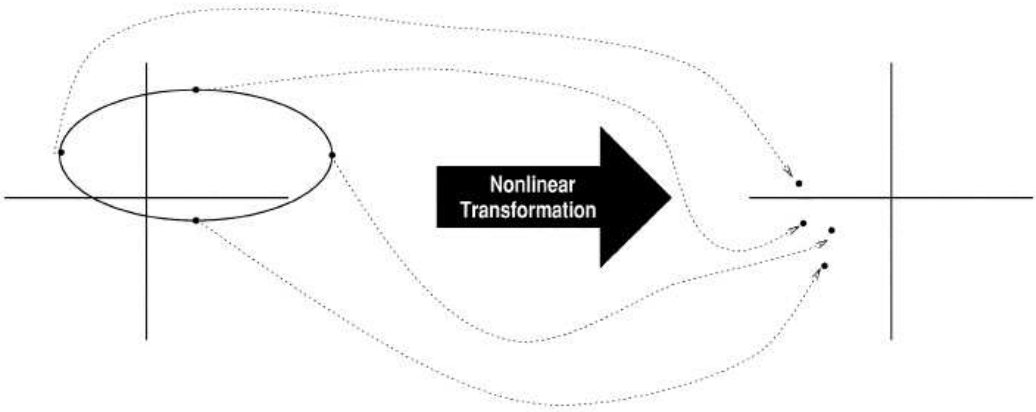


Figure 4.11: The principle of the Unscented transform.

Sigma points are not selected randomly. They have to be deterministically chosen so that they exhibit precisely the properties that we are interested to explore in the non-linear transformation, in this case, their mean and covariance. Moreover, the points can be weighted so that the higher order moments can be fine tuned further. For a set of p sigma points around the state mean $\bar{\mathbf{x}}$, the weights $W^{(i)}$ can be positive or negative, but must obey the following condition:

$$\sum_{i=0}^{p} W^{(i)} = 1 \tag{4.6}$$

A proposed set of sigma points χ^i that exhibit the desired properties of mean $\bar{\mathbf{x}}$ and covariance $\mathbf{P_x}$ is the following:

$$\begin{aligned}
\chi^0 &= \bar{\mathbf{x}} \\
\chi^i &= \bar{\mathbf{x}} + \left(\sqrt{(L+\lambda)\mathbf{P_x}}\right)_i \quad i = 1, ..., L \\
\chi^i &= \bar{\mathbf{x}} - \left(\sqrt{(L+\lambda)\mathbf{P_x}}\right)_{i-L} \quad i = L+1, ..., 2L
\end{aligned}$$

Where L is the dimension of each data point and $\left(\sqrt{(L+\lambda)\mathbf{P_x}}\right)_i$ is the ith column of the square root matrix $(L+\lambda)\mathbf{P_x}$. The corresponding weights can be calculated as follows:

$$W^0 = \lambda/(L+\lambda)$$
$$W^i = 1/2(L+\lambda) \quad i = 1, ..., L$$

with $\lambda = \alpha^2 L - L$ and L being the number of dimensions. These values and constants spring from optimizations obtained by estimating the moments of probability distributions via Taylor expansions. The α parameter serves to provide some degree of control over the spread of the sigma points around $\bar{\mathbf{x}}$. Figure 4.12 shows the effect of the α parameter with a mean centered at (0,0). Points represented by a star were generated with a value of $\alpha = 10^{-2}$, and those shown as a bubble with $\alpha = 10^{-1}$. As you can see, for lower values, the points tend to be closer to the mean. According to [9] a good value for α is in the order of 10^{-3} for most applications dealing with Gaussian distributions. Some authors also distinguish between the weights used for calculating means (W_s^i) and those for calculating covariances (W_c^i), with the purpose of adding additional tweaks to higher order moments.

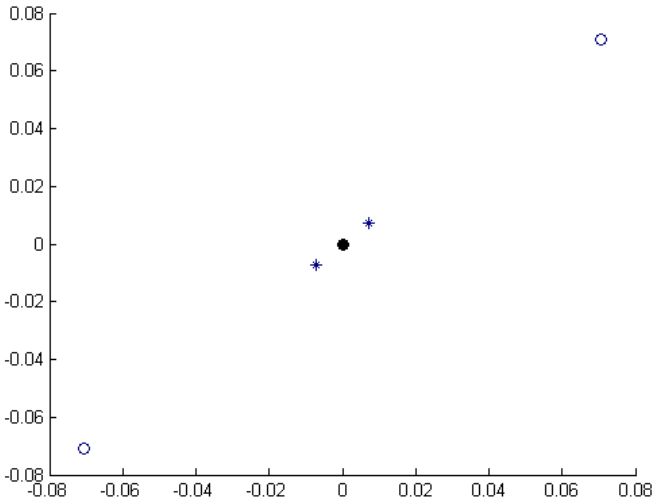


Figure 4.12: Sigma points with different α. Smaller values of α generate points closer to the mean.

Advanced methods to capture and/or minimize errors of higher order moments is described in [13] and [29]. All of them require careful analysis of more Taylor terms used to approximate the probability distribution at hand. For the purposes of the work in this thesis, mean and covariance will suffice.

4.3.2 The Unscented Kalman Filter

When considering a dynamic system governed by the following set of state-space equations:

$$\begin{aligned} x_k &= f(x_{k-1}) + w(k) \\ y_k &= h(x_k) + v(k) \end{aligned} \tag{4.7}$$

where f and h are known non-linear functions with w and v Gaussian white noise with zero mean and covariance matrices $[w(k)w(k)^T] = Q(k)$ and $[v(k)v(k)^T] = R(k)$, the first step of the UKF is to select the set of sigma points and perform the Unscented transform with both f and h. The mean and covariances can then be estimated and used as input to the equations of the standard Kalman filter. The complete algorithm of the UKF can be summarized as follows:

Step 1: Calculate sigma points as described in the previous section.
Step 2: Compute the predicted mean $\hat{\mathbf{x}}_{k|k-1}$ and covariance $\mathbf{P_x}$ with the sigma points $\chi^i_{k-1|k-1}$ and their weights W^i.

$$\chi^i_{k|k-1} = f(\chi^i_{k-1|k-1}) \quad i = 0, 1, ..., 2L$$

$$\hat{\mathbf{x}}_{k|k-1} = \sum_{i=0}^{2L} W^i_s \chi^i_{k|k-1}$$

$$\mathrm{P_x} = \sum_{i=0}^{2L} W^i_c \, [\chi^i_{k|k-1} - \hat{\mathbf{x}}_{k|k-1}][\chi^i_{k|k-1} - \hat{\mathbf{x}}_{k|k-1}]^T + Q(k)$$

Step 3: Compute the predicted observation mean $\hat{\mathbf{y}}_k$ and its convariance $\mathbf{P_y}$ from the transformed sigmas y^i_k.

$$\mathrm{y}^i_k = h(\chi^i_{k|k-1}) \quad i = 0, 1, ..., 2L$$

$$\hat{\mathbf{y}}_k = \sum_{i=0}^{2L} W^i_s y^i_k$$

$$\mathrm{P_y} = \sum_{i=0}^{2L} W^i_c \, [y^i_k - \hat{\mathbf{y}}_k][y^i_k - \hat{\mathbf{y}}_k]^T + R(k)$$

Step 4: Calculate the cross correlation matrix $\mathbf{P_{xy}}$.

$$\mathrm{P_{xy}} = \sum_{i=0}^{2L} W^i_c \, [\chi^i_{k|k-1} - \hat{\mathbf{x}}_{k|k-1}][y^i_k - \hat{\mathbf{y}}_k]^T$$

Step 5: Apply classic Kalman filter prediction equations to obtain the predicted state $\hat{\mathbf{x}}_{k|k}$ and covariance $\mathbf{P}_{k|k}$ for the next step.

$$\mathrm{K}_k = \mathbf{P_{xy}} \mathbf{P_y}^{-1}$$
$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + K_k(\mathbf{y}_k - \hat{\mathbf{y}}_k)$$
$$\mathbf{P}_{k|k} = \mathbf{P}_{k|k-1} - K_k \mathbf{P}^T_{\mathbf{y}}$$

Step 6: Repeat steps 1 to 5 for the next reading/measurement.

4.3.3 UKF Residuals

The problem addressed is the exploration of a scheme to determine faults within the sensor system of the Delfi-n3Xt nanosatellite based on the UKF. By combining the model of the system along with the measurements, it is possible to determine the occurrence of a fault by measuring the change in mean of the measured Gaussian variable. The residuals (ε_k) between the actual observations and those predicted by the UKF serve as a parameter for such an analysis, and such residuals can be calculated as follows:

$$\varepsilon_k = y_k + b_f - \hat{y}_k \tag{4.8}$$

where b_f is the fault. The most straightforward method to detect a fault is to establish a threshold for λ and alert of a possible fault when $\varepsilon_k > \lambda$. A certain window of samples can also be considered to make detection more reliable (avoid detection of short lived spikes):

$$D^m = \frac{1}{\sqrt{m}} \sum_{k=1}^{m} \varepsilon(k) \tag{4.9}$$

Section 4.6 will go into a broader discussion of proper methods to chose values for window sizes and thresholds. It is obvious that a fault has to be considerably larger than the signal noise in order to be detectable. For small faults, this method does not provide a practical solution; however, if the fault cannot be distinguished from normal noise, it probably does not have a distinguishable impact in the system output either.

4.4 Delfi-n3Xt Sensors and Model

4.4.1 Attitude Sensors

To correctly determine the attitude of Delfi-n3Xt, the satellite will carry a set of sensors for correct attitude determination. The collection of sensor devices provides an acceptable level of redundancy that is useful when one fails or outputs erroneous data. A brief description on the manner in which each device works is offered below:

- The Sun sensor measures the angle towards the sun via photodiodes. On Delfi-n3Xt, two sensors will be included in the six side panels of the craft to determine position (12 in total). The amount of light helps determine the angle at which each particular sensor is facing the sun. Special care has to be taken to counteract Earth's albedo. [1]

- Gyros measure the angular velocities in three axes. Several gyros have to be combined in order to obtain complete information on the rotation rate. In this

[1] The albedo of an object is the amount of light it can reflect from the sun. In the case of the Earth, some parts of the surface and the atmosphere reflect from 4% to 26% of sunlight. This causes problems with sun sensors because they may interpret Earth's albedo as the sun. More modern sensors, however, counteract this problem with the fact that the Earth only reflects certain frequencies, and by measuring the frequencies not reflected by it, can correctly identify the sun.

particular case, there are two gyros for each axis, each oriented in the opposite direction (Y+,Y-,X+,X-,Z+,Z-). On Delfi-n3Xt, two integrated gyros with three axis resolution will be included.

- The magnetometer provides information of the alignment with respect to Earth's magnetic field. More than one magnetometer may be included to determine the alignment angle. On board Delfi-n3Xt, as with the gyros, two magnetometers will be included, each with three axis resolution.

  in each axis.

4.4.2 Model

The motion of a satellite can be mathematically modeled by the following state-space equations derived from Newton's laws of motion and Euler's laws of angular momentum. The state of the system is given by the following **differential** equation:

$$\begin{bmatrix} \dot{\omega} \\ \dot{q} \end{bmatrix} = \begin{bmatrix} \underline{I}^{-1}(N_{ext} + N_{ctrl} - \underline{S}(\omega)(\underline{I}\omega + L_{mw})) \\ \frac{1}{2}\underline{\Omega}q \end{bmatrix} \tag{4.10}$$

where $\dot{q}$ is the attitude quaternion and $\dot{w}$ the angular velocities in the three axis. $\underline{I}$ represents a diagonal matrix containing the moments of inertia of the rotating body for its three axis and N_{ext},N_{ctrl} represent external and control forces respectively. $\underline{S}(\omega)$ and $\underline{\Omega}$ are constructions derived to ease the representation of the system and are defined as follows:

$$\underline{\mathrm{S}}(\omega) = \begin{bmatrix} 0 & -\hat{\omega}_3 & \hat{\omega}_2 \\ \hat{\omega}_3 & 0 & -\hat{\omega}_1 \\ -\hat{\omega}_2 & \hat{\omega}_1 & 0 \end{bmatrix} \quad \underline{\Omega} = \begin{bmatrix} 0 & -\hat{\omega}_3 & \hat{\omega}_2 & \hat{\omega}_1 \\ -\hat{\omega}_3 & 0 & \hat{\omega}_1 & \hat{\omega}_2 \\ \hat{\omega}_1 & -\hat{\omega}_1 & 0 & \hat{\omega}_3 \\ -\hat{\omega}_1 & -\hat{\omega}_2 & -\hat{\omega}_3 & 0 \end{bmatrix}$$

The observation equation of this system in particular is the same angular velocities and quaternion components:

$$\begin{bmatrix} \omega \\ q \end{bmatrix} = \begin{bmatrix} \underline{\mathbf{1}}_{3\times3} & \underline{\mathbf{0}}_{3\times4} \\ \underline{\mathbf{0}}_{4\times3} & \underline{\mathbf{1}}_{4\times4} \end{bmatrix} \begin{bmatrix} \omega \\ q \end{bmatrix} \tag{4.11}$$

where $\underline{1}_{m\times n}$ is an $m \times n$ identity matrix and $\underline{0}_{m\times n}$ is an $m \times n$ matrix of zeros. Equations 4.10 and 4.11 are the state-space equations for this dynamic system (equations 4.7). The fact that the state equation is a differential equation poses an additional challenge as it cannot be applied as-is in the UKF, but has to be solved numerically in each step of the recursion.

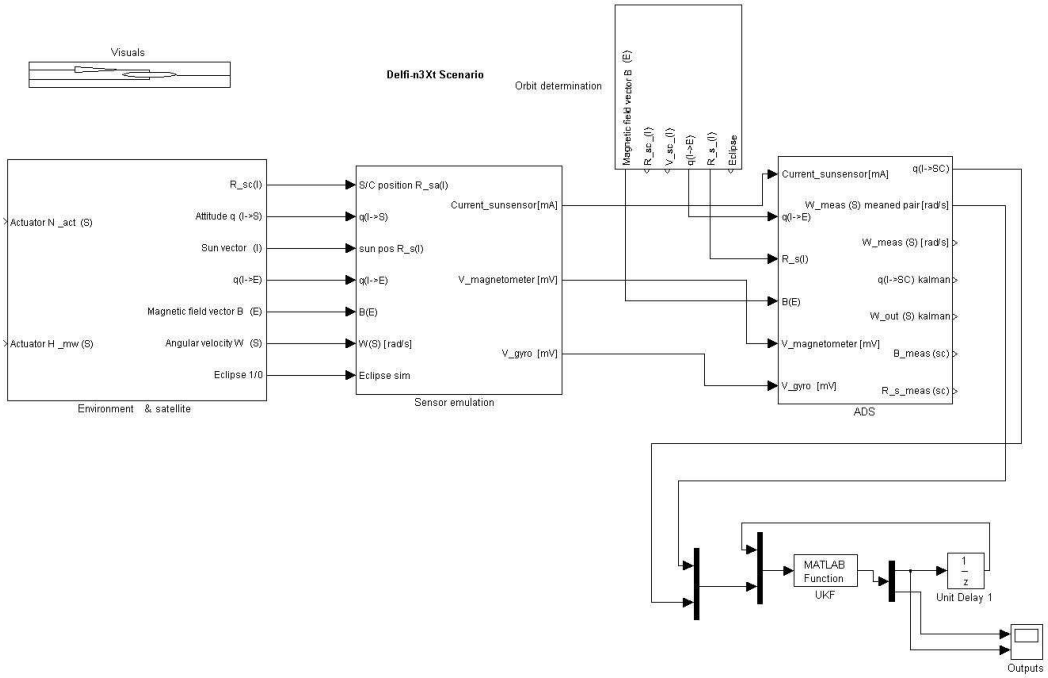


Figure 4.13: The Delfi-n3Xt Simulink model.

4.5 Implementation and Simulation Results

Figure 4.13 shows the Simulink model used to perform simulations of the spacecraft. For simplicity, actuator blocks have been removed. The following blocks are present:

- **Environment & Satellite** provides a simulation of space kinematics and spacecraft properties calculated on properties such as the initial attitude quaternion (spacecraft-center to Earth-inertial), initial angular velocities, inertial moments matrix, mass, magnetic dipole moments and center of mass.
- **Sensor Emulation** will calculate outputs from a sun sensor, three magnetometers (for each axis) and size gyros (two per axis, in opposite directions. Output from the sun sensor comes as millivolts (mV) for magnetometer and gyros.
- **Attitude Determination System** will output angular velocities (rad/s) and an attitude quaternion. The angular velocities are determined by the gyros, while the quaternion is determined via a least squared optimization method (q-method) based on the input from the sun sensor and magnetometers.

The conceptual setup of the implemented method for our experiments is illustrated in figure 4.14. The output from the ADCS unit comes as an input to the implementation of the Unscented Kalman filter. This output contains the current quaternion and

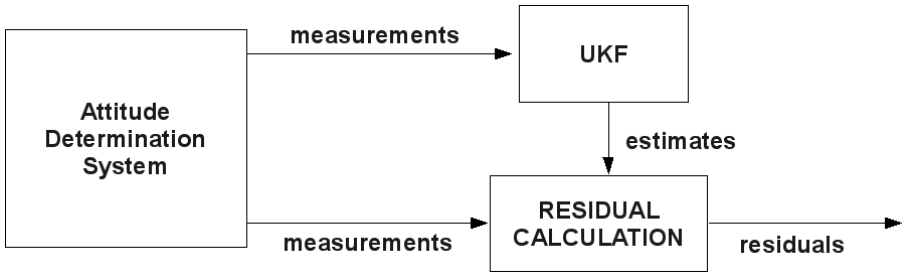


Figure 4.14: Basic Setup of the Fault Detection Scheme.

angular velocities, which after processed by the UKF, are then compared to calculate the corresponding residuals, which we further analize in search of a fault.

4.5.1 Gyro Faults

We start with a simulation of a gyro failure. Sensors are stimulated with normally distributed white noise. Additionally, the model contains switches to add abnormal noise to a gyro and to simulate a stuck-at-0 condition, which may be caused by an open circuit. Normal noise levels all have a mean of 0 and deviations of around 10% of the signal root mean square RMS power (0.05mA for the sun sensor, 0.005mV for the magnetometer and 250mV for the gyro). Kinematic initial conditions for our simulation are as follows:

$$\omega = \begin{bmatrix} 0.3 \\ 0.00 \\ 0.05 \end{bmatrix} \quad q = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad I = \begin{bmatrix} 0.053 \\ 0.053 \\ 0.024 \end{bmatrix} \quad P_{initial} = \begin{bmatrix} 10^{-6} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 10^{-6} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 10^{-6} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 10^{-6} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 10^{-6} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 10^{-6} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 10^{-6} \end{bmatrix}$$

ω, q and I come from the model described in section 4.4.2 and represent the nanosatellite in a constant rotation in X and Z axes. $P_{initial}$ is the initial covariance matrix needed for equation 4.4 with values chosen to indicate a normally working system.

The simulation is run for 300 units of time or steps (each step being a reading from the sensors), and the normal output from gyros is shown in figure 4.15. Since the initial values of angular velocities defined rotation only in two axis, one can clearly see the two gyros static in the axis that has no rotation.

It is highly unlikely that all gyros fail at the same time, more common is a fault in one of the components. Therefore, at t=240, a sudden change in the noise model of one of the gyros is induced to simulate a fault. The new noise model follows a deviation of 1000mV (around 50% of signal RMS power) and keeps the mean to 0. The output from the sensor unit is illustrated in figure 4.16a. The residuals obtained from the UKF are

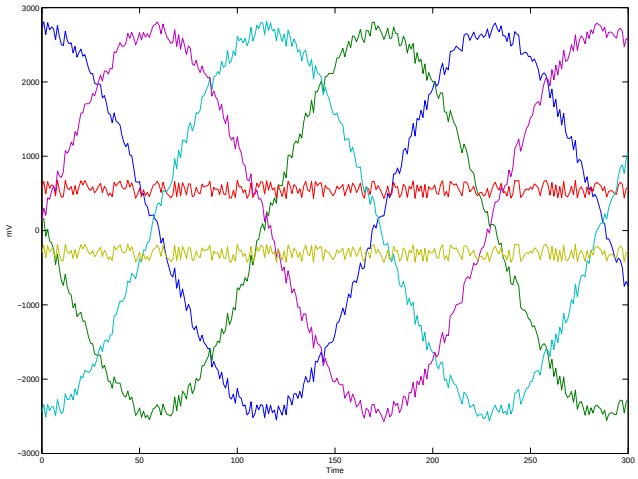


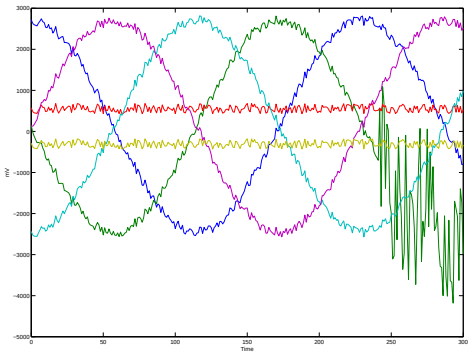
Figure 4.15: Normal Output from Gyros.

shown in figures 4.16b to 4.16d, with error correlation values of $R = 0.1I$ and $Q = 0.01I$, $Q = 0.05$, $Q = 0.08$ respectively. To detect a fault, we used a windowed residual method as described by equation 4.9, with a window size of $m = 6$ and $\lambda = 0.02$.

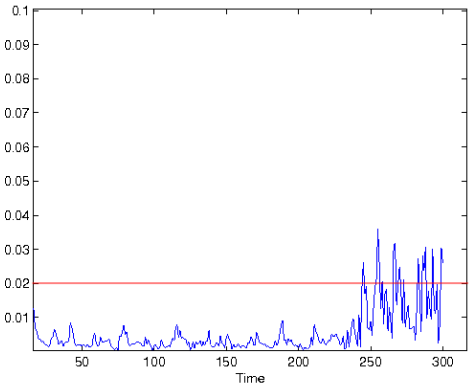
Another possible fault is a stuck-at condition, where a component is assumed to be stuck at a certain value, regardless of the real conditions. This is may be caused by manufacturing defects or open circuits. In this particular simulation, we stimulated a stuck-at-0 behavior. The simulation is again run for 300 time units, and at t=240 one gyro is forced to freeze at a 0 transmission. The fault is shown in figure 4.17a. The residuals obtained from the UKF are shown in figures 4.17b to 4.17d, again with error correlation values of $R = 0.1I$ and $Q = 0.01I$, $Q = 0.05$, $Q = 0.08$ respectively. The same window and λ values are used as in the previous simulation.

In both error cases, the fault can be visualized as the change in the mean of the residuals from t=240 onwards. As expected, a lower value given to the model's error covariance matrix gives more weight to the prediction associated to the model in the UKF, and thus, a more pronounced and visible change. We can see that with smaller covariance values for the model (more trust), the method is quicker to detect faults. This can also be perceived visually by a more pronounced changed in the residuals.

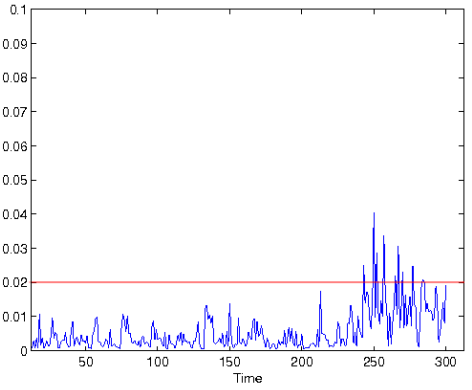
It is important to find a balance between the confidence given to the model and the measurements. Assigning a very small covariance error to the model (more trust) can make it difficult to detect abnormal events or conditions, such as unexpected torques, which will be read by the sensors, but may be considered faults by the UKF residuals method. On the other hand, too much confidence on the measurements can make it difficult to detect erroneous readings. Literature suggests that much of these parameters can be chosen from experience [10], but also a realistic measure of noise and disturbances in the particular environment of each application. A more in depth discussion of a method to decide proper values of window size and threshold is done in section 4.6



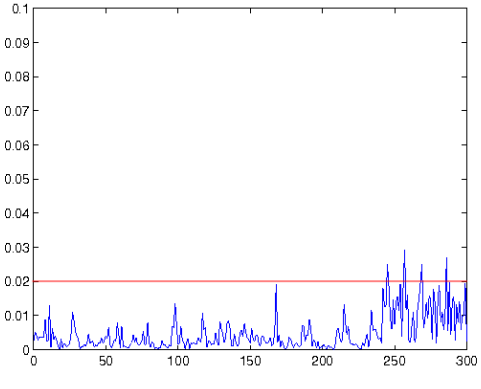
(a) Change in noise model



(b) $Q = 0.01I, R = 0.01I$. Fault detected at t=249.



(c) $Q = 0.05I, R = 0.01I$. Fault detected at t=266.



(d) $Q = 0.08I, R = 0.01I$ Fault undetected.

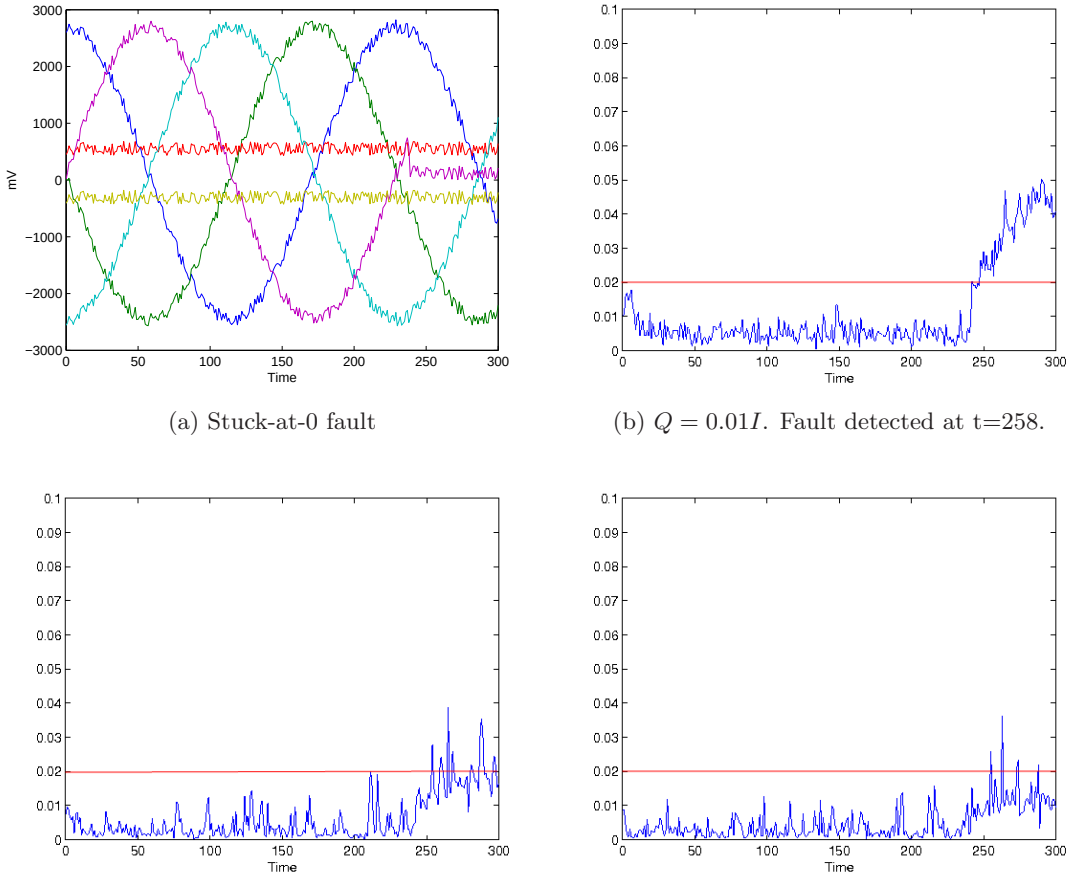
Figure 4.16: UKF Residuals for Gyroscope change in noise model

4.5.2 Faults from Euler Angle Estimations

One approach explored during the development of this work was to take advantage of the redundancy present in the system and use the quaternions calculated from magnetometer and sun sensor to estimate errors in the gyros. Attitude and rotations can be specified in both quaternions or euler angles, and to explore this approach, we must first convert the quaternions to euler angles for further processing. Figure 4.18 shows the residuals obtained from this method.

As the figure shows, the output has noticeable short lived spikes present. The cause of this spike is none other than the presence of infinities that come out of the conversion of quaternions to euler angles. These infinities are also know as *singularities* and they are an inherent problem to attitude determination via Euler angles. Since the conversion involves trigonometric and inverse trigonometric functions applied to quaternion components, they result in singularites at certain values.

Proper values of window length and threshold can be used to compensate these



(a) Stuck-at-0 fault

(b) $Q = 0.01I$. Fault detected at t=258.

(c) $Q = 0.05I$. Fault undetected.

(d) $Q = 0.08I$. Fault undetected.

Figure 4.17: UKF Residuals for Gyroscope stuck-at-0 fault

spikes, but it will not remove the problem completely. For instance, if the spacecraft is rotating at a faster angular velocity, these spikes will be more and more common, and the increases in the mean of residuals in the presence of faults will become indistinguishable. Rotation over the three axes may also increase this phenomenon. Some solutions that may be explored to counteract this problem include determination of angular velocity from the mathematical model and solve the system of equations or constrain the solution to points not close to these singularities.

4.5.3 Magnetometer Faults

In this scenario, we simulate magnetometer failures. Sensors are stimulated with normally distributed white noise. Additionally, the model contains switches to add abnormal noise to a gyro and to simulate a stuck-at-0 condition, which may be caused by an open circuit. Kinematic initial conditions are the same as for the gyro.

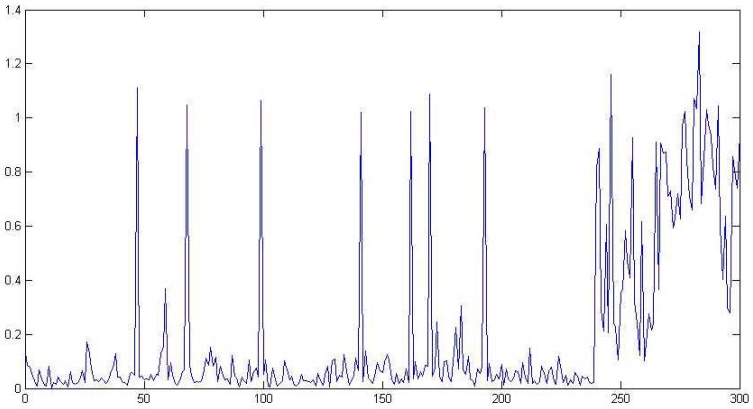


Figure 4.18: Fault Detection from Euler Angle Estimations.

The model shows additive noise for all sensors and switches to simulate a stuck-at-0 fault and addition of abnormal noise (change in noise model). Again the simulations are run for 300 units of time and the fault is introduced at t=240. The normal output from magnetometers is shown in figure 4.19.

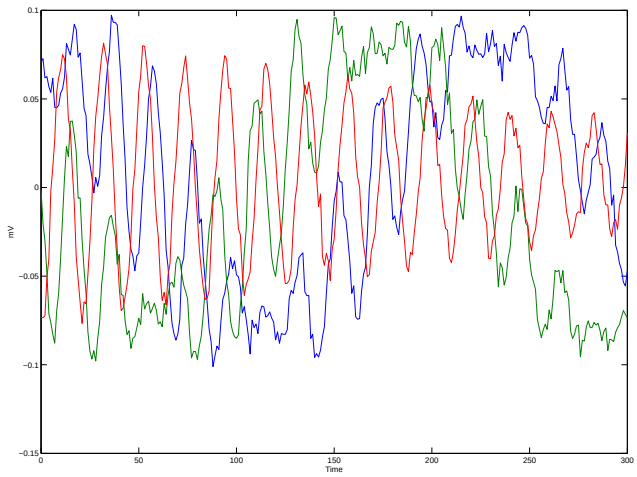
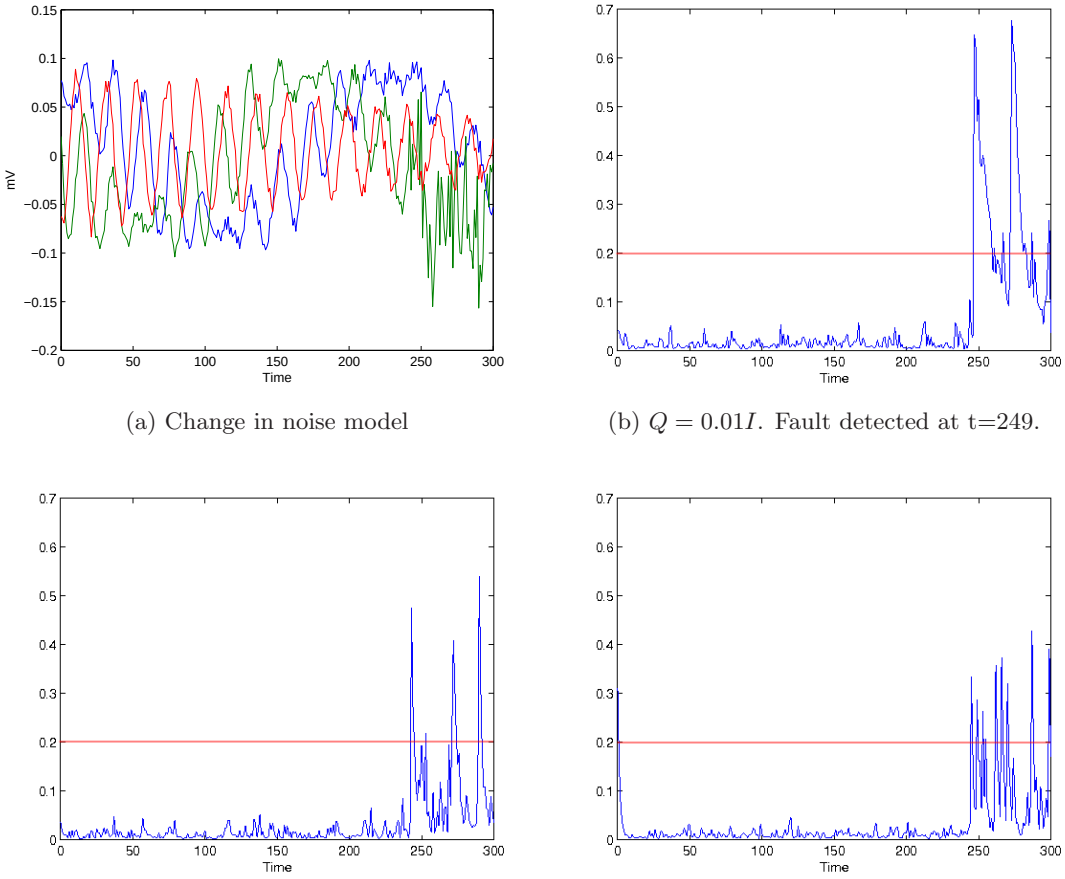


Figure 4.19: Normal Output from Magnetometers.

The first fault to be simulated is a change in noise model. Delfi-n3Xt needs three axis magnetometers, which require either three magnetometers of one axis resolution or one with three axis determination capability. In this case we will assume the former. As with the gyros, it is unlikely that all components fail, so only one is injected with the fault. The new noise model has a deviation of 0.05mV (around 50% RMS power) and

keeps the mean at 0. The output of the magnetometers with the injected disturbance is illustrated in figure 4.20a.

The residuals obtained from the UKF are shown in figure 4.20, with error correlation values of $R = 0.1I$ and $Q = 0.01I$, $Q = 0.05$, $Q = 0.08$ respectively. The graph shows a definite increment in the mean of the residuals, however it is full of short lived spikes with abrupt changes. This spiky output is problematic to deal with because depending on the window size, the method may interpret one fault, several intermittent faults or none at all. In this simulation we use a window size of 6 and $\lambda = 0.2$.



(a) Change in noise model

(b) $Q = 0.01I$. Fault detected at t=249.

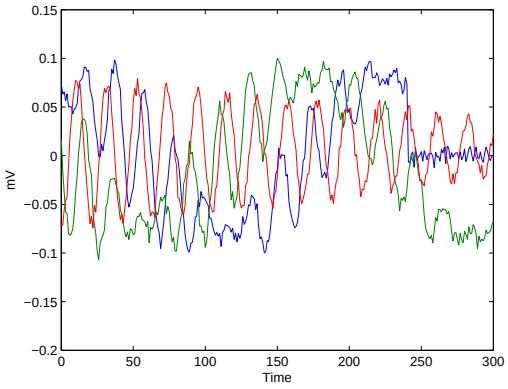
(c) $Q = 0.05I$. Fault detected at t=249.

(d) $Q = 0.08I$. Fault detected at t=268.

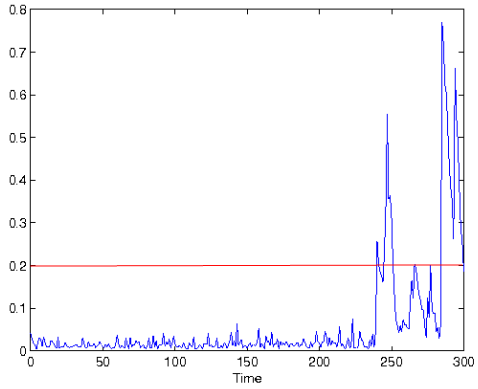
Figure 4.20: Residuals for Magnetometer change in noise model

The stuck-at-0 fault is also simulated at t=240, and the results are illustrated in figure 4.21. Again, the graphs show a spiky output for the residuals. As expected, at a higher error covariance of the model, the less marked the change in residuals.

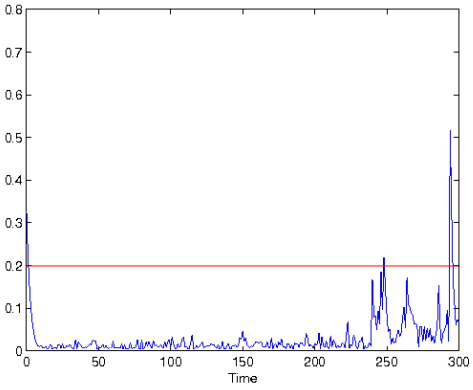
Unlike faults in the gyros, the residuals coming out from the magnetometer faults, although noticeable, come out with abrupt changes. The explanation for this relies in the q-method block visible in figure 4.9. The q-method is a least squared optimization procedure that tries to compute the most likely attitude quaternion from the outputs of



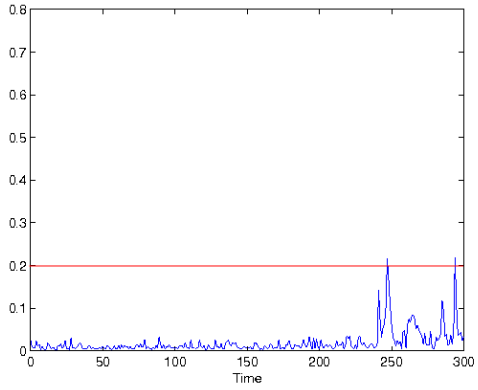
(a) Stuck-at-0 Fault



(b) $Q = 0.01I$. Fault detected at t=246.



(c) $Q = 0.05I$. Fault detected at t=298.



(d) $Q = 0.08I$. Fault undetected.

Figure 4.21: Residuals for Magnetometer stuck-at-0 fault

the sun sensor(s) and magnetometer(s). Therefore, any disturbance to these sensors will not translate to a linear disturbance in the calculation of the quaternion. This, as well, makes it extremely difficult to isolate a fault within these components. At most, as can be seen from the results presented here, the UKF is able to detect the presence of an anomaly. Inclusion of an observer before or within the q-method block may provide a way to add proper fault isolation to these components, since each sensor output would be monitored independently.

The general tendency again is that with more trust placed in the model, the easier and faster the method will detect the presence of an anomaly. Spiky output can, however, cause problems. Depending on the window size, spikes properly separated may not be captured by the window size, and render the fault undetected, as in the result presented in figure 4.21d.

4.6 Parameters for Fault Detection Method

In the simulations and results presented above, detection (or undetection) of faults was decided with given values of window sizes and thresholds. These values, however, can be chosen appropriately with statistically acceptable confidence levels. [10] suggests a method based on the χ^2 distribution.

As the cumulative windowed sum of the residuals is defined by

$$D^m = \frac{1}{\sqrt{m}} \sum_{k=1}^{m} \varepsilon(k) \tag{4.12}$$

where m is the number of samples (window size) and since $\varepsilon(k)$ is assumed to be Gaussian with mean 0 in the absence of faults, then D^m is also Gaussian $N(0, P_{yy})$ under those same conditions. However, in a faulty system, from equation 4.8 it is clear that the distribution of the residual becomes $N(b_f, P_{yy})$ and, thus, for D^m, $N(\sqrt{m}b_f, P_{yy})$.

More useful, yet, is a normalized version of D^m by its variance

$$S^m = (D^m)^2 P_{yy}^{-1} \tag{4.13}$$

and a fault is assumed to occur when $S^m > \lambda$, with λ being a threshold value. S^m is now a χ^2 distributed expression, and therefore λ can be chosen with the desired confidence level. The probability density function with a no fault hypothesis (H_0) is represented by

$$p(S^m/H_0) = \frac{1}{\sqrt{2\pi S^m}} \quad \exp(\frac{-S^m}{2}) \tag{4.14}$$

The probability of having a false alarm (signaling an inexistent fault) is expressed by

$$P_f = \int_{\lambda}^{\infty} p(S^m/H_0)\, dS^m \tag{4.15}$$

In the case we assume the presence of a fault (H_1), the distribution deviates by $N(\sqrt{m}b_f, P_{yy})$, and the probability function becomes:

$$p(S^m/H_1) = \frac{1}{2\sqrt{(2\pi S^m)}} \Big(\exp(\frac{(-\sqrt{S^m} + \sqrt{m/P_{yy}}b_f)^2}{2}) + \exp(\frac{(-\sqrt{S^m} - \sqrt{m/P_{yy}}b_f)^2}{2}) \Big) \tag{4.16}$$

and the false positive rate is therefore:

$$P_m = \int_{0}^{\lambda} p(S^m/H_1)\, dS^m \tag{4.17}$$

The size of a window and λ, therefore, can be decided via equation 4.15 and 4.17. The size of a window has an impact in the performance and should not be too small to promote false detections or too long such that the system cannot detect the error within acceptable time limits.

The method suggested by [10] is, however, based on the assumption that b_f is a constant value and the faults studied were also of that nature; shift faults. In this thesis, however, the faults explored are not all shift faults, so this method would have to be slightly adapted.

For faults involving a change in additive Gaussian noise model, the value of b_f cannot be simply the expected value of the normal distribution, because in most cases, this is 0. A more interesting approach would be to consider b_f as a function of the noise deviation, so that changes in this respect can be used to decide upon values for λ and window size. For faults of stuck-at-0 nature, b_f can be considered a function of the expected value of the model. In the case of continuous signals as those presented here for gyros, the expected value can be the root mean square or quadratic mean.

4.7 Summary

In this chapter we provide an investigation of model based fault detection, particularly well suited for embedded sensor systems. Delfi-n3Xt will carry on board a complete Attitude Determination and Control Subsystem, which makes use of magnmetometers, sun sensors and gyroscopes. Therefore, we take on the task of exploring the applicability of a Kalman based fault detection scheme for a system such as this one.

We begin our discussion with the traditional Kalman Filter algorithm and illustrate its use. Then the Unscented Transform is introduced as a form to map statistical moments into nonlinear functions, which is then taken as the basis of the Unscented Kalman Filter, and this can now be used for error detection in nonlinear dynamic systems. The implementation of this filter is included in the simulation environment of Delfi-n3Xt. Faults are induced in the system to observe the performance of the method and the results show that the method is indeed capable to detecting the fault within acceptable boundaries of time.

Conclusions & Recommendations

5

This chapter provides a summary of the work presented in this thesis, highlighting the conclusions and major points. Recommendations for future work are also included, which are mostly inspired from the experience gained during the development of this project.

5.1 Summary & Conclusions

This thesis was motivated by the problems encountered during the developed and operation of the Delfi-C^3 nanosatellite, launched on April 2008. Failures in the bus and CDHS prompted for an exploration of the design and implementation flaws. Therefore, this thesis encompassed solutions for the CDHS with a special focus on the data bus for internal communications.

In chapter 2, we discussed very briefly about Delfi-C^3's CDHS design and flaws, including hardware and software bugs. This was complemented by Bit-Error measurements with hardware on board the satellite. Furthermore, the CDHS overall architecture for the new Delfi-n3Xt satellite was discussed, including the architecture of the On Board Computer and data bus. The I^{2}C data bus is chosen after a comparison with several other bus architectures. Finally, the implementations in hardware and software are presented, with their specific enhancements to increase the realiability of this primordial component. Hardware will include redundant lines, I/O ports for easier and effective local power control for each subsystem and and I^{2}C circuit protector that guarantees that no single subsystem will hang the bus and stop all communications.

Chapter 3 goes deeper into the discussion of software fault tolerance for the CDHS, including some methods and techniques that should be considered and implemented in the upper software layers. Mainly, however, the chapter focuses in communication reliability, were coding theory is introduced and several coding schemes are presented as options for Delfi-n3Xt and the Delfi Programme in general. The parity scheme, as implemented in Delfi-C^3 contains flaws that make it a poor choice for future mission. CRC's, if properly implemented, however, would provide realibility in error detection with low computational overhead.

Finally, as fault tolerance can also be applied on a higher level and to other systems, this thesis also decides to explore model based fault detection, in an effort to include techniques on a system level which could be implemented on future nanosatellites of the Delfi programme. This can be performed via kalman filtering techniques, which take into account a mathematicla model of the process along with its actual measurements. Traditional kalman filters however, are designed for linear systems, and thus, needs to be modified for nonliear processes, such as those involved in satellite navigation. The Unscented Kalman Filter provides a way to apply the filter to a nonlinea model based a simple probabilistic principle. A UKF was implemented and included in a Delfi-n3Xt

simulation environment with results showing that is possible to detect errors in sensors by proper analysis of residuals. Finally, methods to decide the proper value of initial parameters for the UKF to optimize its effectiveness are discussed.

5.2 Recommendations for Future Work

As of this moment, the Delfi-n3Xt has not gone into the detailed design phase yet, so the architectures presented here must be implemented. Payload partners should be provided with well tested interfaces to guarantee ease of integration. Some other concrete recommendations and guidelines would be the following:

- Proper documentation should be written and available at all times. One of the problems encountered when exploring the design of Delfi-C^3 was the lack of proper documentation, specially for software. Proper documentation, commenting and versioning should be regarded as pivotal in the project.

- Testing of the designs proposed should be carried out as soon as possible. The proposed architectures and solutions presented here need to be teste throughly. At the moment of presenting this thesis, not enough students were available in the project to take on the task of breadboarding and developing prototype boards for the Delfi-n3Xt, and thus thorough tests could not be carried out.

- As the Delfi Programme grows, it is adivisable to develop a standard set of components, of which the data bus architecture could be one. In this sense, a higher level protocol over I^{2}C should be developed, that includes mechanisms for error detection, many-to-many communication capabilities (not only master-slave) and a transparent software layer to be used for the rest of the software in the subsystems.

- Testing Bit-Error Rate on a large bus with 10+ nodes would provide a realistic environment to asses the quality of the enhancements proposed here.

- Explore the performance of the Unscented Kalman Filter for error detection with actuators similar to those proposed for Delfi-n3Xt. This would serve as a realistic benchmark for the failure detection method, and truly provide clues into its practicality.

- Detailed benchmark between the Unscented Kalman Filter and the Extended Kalman Filter, which is the standard used filter in industry. Comparison should contain details of computational resource requirements and performance to fault detection applications.

- Look further into fault estimations via Euler angle predictions. Some ways to avoid the problems of singularities may be to restrict the problem to angles not to close to those that cause problems or to derive the angular velocity from the model equations.

- Explore techniques for failure isolation in the sun sensor and magenetometer. At this point, the method proposed here can only detect the presence of a failure in the ADCS system related to quaternion calculation, but cannot distinguish which components is the cause. This may be achieved by placing the UKF before the q-method block.

Bibliography

[1] Sohel Anwar and Lei Chen, *An analytical redundancy-based fault detection and isolation algorithm for a road-wheel control subsystem in a steer-by-wire system*, vol. 56, September 2007, pp. 2859–2869.

[2] Olivier Bilenne, *Fault detection by desynchronized kalman filtering, introduction to robust estimation*, June 2004.

[3] G. Castagnoli, S. Brauer, and M Herrmann, *Optimization of cyclic redundancy-check codes with 24 and 32 parity bits*, Communications, IEEE Transactions on **41** (1993), no. 6, 883–892.

[4] Freescale Semiconductors, Inc., *M68hc11 reference manual*, 6.1 ed., 2007.

[5] M. Genbrugge, *Delfi-n3xt requirements management tool*, April 2009.

[6] Janos Gertler, *Fault detection and diagnosis in engineering systems*, CRC Press, May 1998.

[7] R. Isermann, *Fault diagnosis systems. an introduction from fault detection to fault tolerance*, Springer, 2006.

[8] Ferreira J, Oliveira A, and Fonseca P, *An experiment to assess bit error rate in can*, 2004, pp. 15–18.

[9] S.J. Julier and J.K. Uhlmann, *Unscented filtering and nonlinear estimation*, Proceedings of the IEEE **92** (2004), no. 3, 401–422.

[10] C. W. Chan K. Xiong and H. Y. Zhang, *Unscented kalman filter for fault detection*, 16th IFAC World Congress (2005).

[11] Philip Koopman and Tridib Chakravarty, *Cyclic redundancy code (crc) polynomial selection for embedded networks*, Dependable Systems and Networks, International Conference on **0** (2004), 145.

[12] E.C. Larson, Jr. Parker, B.E., and B.R. Clark, *Model-based sensor and actuator fault detection and isolation*, vol. 5, 2002, pp. 4215–4219 vol.5.

[13] U. Lerner, *Hybrid bayesian networks for reasoning about complex systems*, 2002.

[14] M. N. Kragelund et al. M. Green, M. O. Halse, *Attitude determination for aausat-ii*, May 2005.

[15] T.C. Maxino and P.J. Koopman, *The effectiveness of checksums for embedded control networks*, Dependable and Secure Computing, IEEE Transactions on **6** (2009), no. 1, 59–72.

[16] Kevin J. Melcher and William A. Maul, *Sensor data qualification system developed and evaluated for assessing the health of ares i upper-stage sensors*, Tech. Report NASA/TM2007-214479, NASA Glenn Research Center, Cleveland, Ohio 441353191, U.S.A., 2007.

[17] A. Monteriu, P. Asthan, K. Valavanis, and S. Longhi, *Model-based sensor fault detection and isolation system for unmanned ground vehicles: Experimental validation (part ii)*, April 2007, pp. 2744–2751.

[18] NXP Semiconductors, *I2c-bus specification*, 2.1 ed., January 2000.

[19] A. Okatan, Ch. Hajiyev, and U. Hajiyeva, *Kalman filter innovation sequence based fault detection in leo satellite attitude determination and control system*, June 2007, pp. 411–416.

[20] Reza Olfati-Saber, *Distributed kalman filtering and sensor fusion in sensor networks*, Dartmouth College, Thayer School of Engineering, Hanover, NH 03755.

[21] D. Paret, *The i2c bus from theory to pratice*, John Wiley Sons, February 1997.

[22] B. Parhami, *Defect, fault, error,..., or failure?*, Reliability, IEEE Transactions on **46** (1997), no. 4, 450–451.

[23] Peter K. Pearson, *Fast hashing of variable-length text strings*, Commun. ACM **33** (1990), no. 6, 677–680.

[24] Philips Semiconductors, *Pcf8575 remote 8-bit i/o expander for i2c-bus*, 2002nov22 ed., November 2002.

[25] Tenkasi V. Ramabadran and Sunil S. Gaitonde, *A tutorial on crc computations*, IEEE Micro **8** (1988), no. 4, 62–75.

[26] J. Ray and P. Koopman, *Efficient high hamming distance crcs for embedded networks*, June 2006, pp. 3–12.

[27] Robert Bosch GmbH, *Can specification*, 2.0 ed., 1991.

[28] J. Rufino, P. Verssimo, G. Arroz, C. Almeida, and L. Rodigues, *Fault-tolerant broacast in can*, 1998, pp. 150–159.

[29] D. Tenne and T. Singh, *The higher order unscented filter*, vol. 3, June 2003, pp. 2441–2446 vol.3.

[30] Neil Thacker, *Tina algorithms guide (beyond geometric vision)*, ch. 12, pp. 127–134, Electronics Systems Group, University of Sheffield, Sheffield, England, 2003.

[31] Wilfredo Torres-Pomales, *Software fault tolerance: A tutorial*, Tech. Report TM-2000-210616, NASA - Langley Research Center, Hampton, Virginia 23681-2199, U.S.A., October 2000.

[32] N. Venkateswaran, M. S. Siva, and P. S. Goel, *Analytical redundancy based fault detection of gyroscopes in spacecraft applications*, Acta Astronautica **50** (2002), no. 9, 535 – 545.

[33] Xianghua Xu, Wanyong Chen, Jian Wan, and Ritai Yu, *Distributed fault diagnosis of wireless sensor networks*, Nov. 2008, pp. 148–151.

Nomenclature

ADCS	Attitude Determination and Control Subsystem
BER	Bit-Error Rate
bps	bits per second
CAN	Controller Area Network
CDHS	Command & Data Handling Subsystem
COMMS	Communications Subsystem
COTS	Commercial Off the Shelf
CRC	Cyclic Redundancy Check/Code
EKF	Extended Kalman Filter
EPS	Eletrical Power Subsystem
Kbps	Kilobits per second
kHz	Kilohertz
mA	Milliamperes
MHz	Megahertz
mV	Millivolts
OBC	On-Board Computer
PCB	Printed Circuit Board
SPI	Serial Peripheral Interface
STS	Structural Subsystem
TCS	Thermal Control Subsystem
UKF	Unscented Kalman Filter

Communications Service Layer Souce Code for Delfi-n3Xt

A

```
1  /*********************************************
2  I2C Driver for MSP430
3
4  Napoleon E. Cornejo
5  ncornejo@gmail.com
6  Delft, NL
7
8  *********************************************/
9  #ifndef __N_I2C__
10 #define __N_I2C__
11
12 //#define USE_EDAC
13
14 #define I2C_BUF_SIZE 150
15
16 #ifdef USE_EDAC
17 #define I2C_EDAC_BUF_SIZE (I2C_BUF_SIZE/8)+1
18
19 //set the edac header, change to any other
20 #include "pearson.h"
21
22 #endif
23
24 //modes for device
25 #define I2C_MODE_MASTER 1
26 #define I2C_MODE_SLAVE 0
27
28 //states
29 #define I2C_STATUS_IDLE 0
30 #define I2C_STATUS_RX 1
31 #define I2C_STATUS_TX 2
32
33 //return conditions
34 #define I2C_OK 0
35 #define I2C_ERR_TIMEOUT -2
36 #define I2C_ERR_XMIT -1
37
38
39 int n_initI2C(unsigned char my_address, unsigned char slave);
40 int n_I2Crecv(unsigned char slave_address, volatile char *data, unsigned char length, u
41 int n_I2Csend(unsigned char slave_address, volatile char *data, unsigned char length, u
42 void getI2CData(char *buf);
43 int n_I2C_slave_packet(volatile char *data, unsigned char length);
44
45 extern void i2c_recv_callback(volatile char *data, int len);
46
```

```
47 #endif

 1 /*********************************************
 2 I2C Driver for MSP430
 3
 4 Napoleon E. Cornejo
 5 ncornejo@gmail.com
 6 Delft, NL
 7
 8 *********************************************/
 9
10 #include <msp430x16x.h>
11 #include "cross_studio_io.h"
12 #include "n_i2c.h"
13 #include "n_timers.h"
14
15
16 static volatile char *xdata =0;
17 static volatile unsigned char xdata_len =0;
18 static volatile unsigned char xfered =1;
19
20 // are we a slave or master?
21 static unsigned char i2c_mode =0;
22
23 // used when we are a slave
24 static volatile char slv_rcvdata[I2C_BUF_SIZE];
25 static volatile char slv_rcvdata_len =0;
26 static volatile char slv_senddata[I2C_BUF_SIZE];
27 static volatile char slv_senddata_len =0;
28 static volatile char xmit_mode = I2C_STATUS_IDLE;
29
30 // for EDAC
31 #ifdef USE_EDAC
32 static volatile char edac[I2C_EDAC_BUF_SIZE];
33 static volatile unsigned char edac_len =0;
34 static volatile unsigned char exfered =0;
35 #endif
36
37
38 int n_initI2C(unsigned char my_address, unsigned char mode){
39
40       // configure pins (Ports)
41       P3DIR = 0x00;
42       P3SEL = 0x0A;
43
44       //stop any activity in I2C mode
45       U0CTL = 0x00;
46       I2CTCTL = 0x00;
47
48       // Interrupts: receive, transmit, access and nack
49       I2CIE = TXRDYIE + RXRDYIE + ARDYIE + OAIE + NACKIE;
50
51       // set local address
52       I2COA = my_address;
53
54      // clock stuff
```

```
55      I2CPSC = 0x00;
56      I2CSCLH = 0x08;     //high state clock duration
57      I2CSCLL = 0x08;     //low state clock duration
58
59      i2c_mode = mode;
60
61      if ( mode == I2C_MODE_SLAVE ) {
62
63            U0CTL = I2C + SYNC;
64            U0CTL &= ~I2CEN;
65            I2CTCTL = I2CSSEL1;
66            I2CTCTL &= ~I2CTRX;
67
68          I2CTCTL |= I2CSTT+I2CSTP;
69            U0CTL |= I2CEN;    // enable I2C functionality
70      }
71
72      I2CIFG =0;
73
74      return 0;
75  }
76
77  // this fucntion only used by master
78  // only master can ask for info
79  int n_I2Crecv(unsigned char slave_address, volatile char *data, unsigned char length, u
80
81      int rtimer =0;
82
83      // Configure USART for I2C mode
84      // I2C mode, 7-bit addrs, I2C mode sync, master mode
85      U0CTL = I2C + SYNC + MST;
86
87      // make sure I2C not yet released
88      U0CTL &= ~I2CEN;
89
90      // configure transmission
91      // byte mode, hw controlled data transfer, ACLK clock, SDA pin in receive mode
92      I2CTCTL = I2CSSEL1;
93      I2CNDAT = length;
94
95      // buffer for reception
96      xdata =data;
97      xdata_len =length;
98      xfered =0;
99
100 #ifdef USE_EDAC
101     length = length + num_edac_bytes(length);
102     I2CNDAT = length;
103 #endif
104
105     // Set adress of destination slave controller
106     I2CSA = slave_address;
107
108     // Enable I2C hardware
109     U0CTL |= I2CEN;
```

```
110     // Send START bit
111     //I2CTCTL |= 0x01;
112     I2CTCTL |= I2CSTT+I2CSTP;
113
114     d_startTimer(I2C_TIMER);
115     while ( rtimer < timeout && xfered < length){ rtimer = d_getTimer(I2C_TIMER);
    }
116
117     // check to see if I2C bus is busy
118     while(I2CDCTL & I2CBB){   }
119
120     if ( rtimer >= timeout )
121         return I2C_ERR_TIMEOUT;
122
123
124     if ( !(xfered == length) ) return I2C_ERR_XMIT;
125     else {
126
127         #ifdef USE_EDAC
128         if ( !checkedac(data, xdata_len, edac) ) return I2C_ERR_XMIT;
129         #endif
130
131         return I2C_OK;
132     }
133 }
134
135 /* Slave prepares next transmission */
136 int n_I2C_slave_packet(volatile char *data, unsigned char length){
137     int i=0;
138     for (; i < length; i++ )
139       slv_senddata[i] = data[i];
140
141     #ifdef USE_EDAC
142     {
143       volatile char *edc = slv_senddata+length;
144       exfered = 0;
145           edac_len = setedac(slv_senddata, length, edc);
146           length = edac_len + length;
147     }
148     #endif
149
150     slv_senddata_len = length;
151     return 0;
152 }
153
154 int n_I2Csend(unsigned char slave_address, volatile char *data, unsigned char length, unsigned i
155
156      int rtimer =0;
157
158      // Configure USART for I2C mode
159     // I2C mode, 7-bit addrs, I2C mode sync
160     if ( i2c_mode == I2C_MODE_SLAVE )
161       U0CTL = I2C + SYNC;
162     else
163       U0CTL = I2C + SYNC + MST;
```

```
164
165     // make sure I2C not yet released
166     U0CTL &= ~I2CEN;
167
168     // configure transmission
169     // byte mode, hw controlled data transfer, SMCLK clock, SDA pin in transmit mode
170     //I2CTCTL = 0x28;
171     I2CTCTL = I2CSSEL0 + I2CSSEL1 + I2CTRX;
172     I2CNDAT = length;
173
174     I2CSA = slave_address;
175     xdata =data;
176     xdata_len =length;
177     xfered =0;
178
179 #ifdef USE_EDAC
180           exfered = 0;
181           edac_len = setedac(data, length, edac);
182           length = edac_len + length;
183           I2CNDAT = length;
184 #endif
185
186     // Enable I2C hardware
187     U0CTL |= I2CEN;
188     // Send START bit
189     //I2CTCTL |= 0x01;
190     I2CTCTL |= I2CSTT + I2CSTP;
191
192     d_startTimer(I2C_TIMER);
193     while ( rtimer < timeout && xfered < length){ rtimer = d_getTimer(I2C_TIMER);
    }
194
195     while( I2CDCTL & I2CBB ){  }
196
197     if ( rtimer >= timeout )
198         return I2C_ERR_TIMEOUT;
199
200     if ( !(xfered == length) )  return I2C_ERR_XMIT;
201     else return I2C_OK;
202 }
203
204 #pragma vector=USART0TX_VECTOR
205 //void I2CISR(void) __interrupt[USART0TX_VECTOR] {
206 __interrupt void I2CISR(void) {
207
208   // activity on the bus, so reset timer
209   d_startTimer(I2C_TIMER);
210
211   //received ready
212   if((I2CIFG & RXRDYIFG)) {
213     char c =0;
214
215     c = I2CDRB;            // always read the register.. otherwise interrupt continues
216     I2CIFG &= ~RXRDYIFG;   // clear the interrupt flag as well
217
```

```
218     if ( i2c_mode == I2C_MODE_SLAVE ) {
219        xmit_mode = I2C_STATUS_RX;
220
221        slv_rcvdata[xfered++] = c;
222
223     } else
224
225       #ifndef USE_EDAC
226
227           xdata[xfered++] = c;
228
229       #else
230
231           if ( xfered < xdata_len )
232                 xdata[xfered++] = c;
233           else {
234                 /* the rest is edac codes */
235                         edac[exfered++] = c;
236                         xfered++;
237           }
238
239       #endif
240   }
241
242   // transmit ready
243   if ( I2CIFG & TXRDYIFG ) {
244     I2CIFG &= ~TXRDYIFG;
245
246    if ( i2c_mode == I2C_MODE_MASTER ){
247
248             #ifndef USE_EDAC
249
250             /* just transfer data */
251          I2CDRB = xdata[xfered++];
252
253          #else
254          /* first transfer all data */
255          if ( xfered < xdata_len )
256                  I2CDRB = xdata[xfered++];
257          else {
258                  /* when we have transfered all the data, transfer the edac codes */
259                  I2CDRB = edac[exfered++];
260                  xfered++;
261          }
262          #endif
263    }
264
265     // slaves ...
266     else  {
267        xmit_mode = I2C_STATUS_TX;
268       //there is still data to be sent
269       if ( xfered < slv_senddata_len )
270         I2CDRB = slv_senddata[xfered++];
271       else return;
272     }
```

```
273     }
274
275     // we received no ACK
276     // need a smart way to report this error to
277     // the upper software layers
278     if ( I2CIFG & NACKIFG ) {
279        I2CIFG &= ~NACKIFG;
280     }
281
282     // access ready.  Means that stuff has finished,
283     // usually STOP condition detected in the bus
284     if ( I2CIFG & ARDYIFG ) {
285        I2CIFG &= ~ARDYIFG;
286
287       if ( i2c_mode == I2C_MODE_SLAVE )
288           // as a slave, we finished recieving from the Master
289           // time to call the i2c callback
290           if ( xmit_mode == I2C_STATUS_RX ) {
291                     #ifdef USE_EDAC
292                           if ( checkedac(slv_rcvdata, xdata_len, edac) )
293                     #endif
294                       i2c_recv_callback(slv_rcvdata, xfered);
295                 #ifdef USE_EDAC
296                    else
297                           P1OUT = P1OUT;
298                 #endif
299           }
300
301
302        xmit_mode = I2C_STATUS_IDLE;
303     }
304
305     // we hear a start conditions (SLAVE)
306     // so set transfer counter to 0, reception or
307     // transmission is about to begin.
308     if ( I2CIFG & STTIFG ) {
309           I2CIFG &= ~STTIFG;
310           xfered =0;
311
312           #ifdef USE_EDAC
313           exfered = 0;
314           #endif
315     }
316
317     // own address..
318     // if we are slaves, pay attention
319     if ( I2CIFG & OAIFG ){
320     }
321  }
```

```
1  /*************************************************
2  Timer Drivers for MSP430
3
4  Napoleon E. Cornejo
5  ncornejo@gmail.com
6  Delft, NL
```

```
7
8  ***********************************************/
9
10 #ifndef __N__TIMERS__
11 #define __N__TIMERS__
12
13
14 #define NUM_TIMERS  10
15 #define I2C_TIMER   0
16 #define WAIT_TIMER  1
17
18 void d_initTimers(void);
19 unsigned int d_startTimer(unsigned int id);
20 unsigned int d_getTimer(unsigned int id);
21 void wait(int millis);
22
23 #endif
```

```
1  /***********************************************
2  Timer Drivers for MSP430
3
4  Napoleon E. Cornejo
5  ncornejo@gmail.com
6  Delft, NL
7
8  ***********************************************/
9
10 #include <msp430x16x.h>
11 #include "n_timers.h"
12
13 volatile static unsigned int global_timer =1;
14 volatile static unsigned int timers[NUM_TIMERS];
15
16 // Uses Timer A
17 void d_initTimers(void){
18
19   // Start Timer_A in continuous mode.
20   TACTL |= 0x0010;
21
22   // count up to...
23   TACCR0 = 0xffff;
24
25   // enable interrupt and output high
26   TACCTL0 = 0x14;
27
28 }
29
30
31 unsigned int d_startTimer(unsigned int id){
32     if ( id >= NUM_TIMERS ) return -1;
33     timers[id] = global_timer;
34     return timers[id];
35 }
36
37 unsigned int d_getTimer(unsigned int id){
38     unsigned int l =0;
```

```
39      if ( id >= NUM_TIMERS ) return -1;
40
41
42      if ( global_timer < timers[id] )
43        l = (0xffff - timers[id]) + global_timer;
44     else
45      l= (global_timer - timers[id]);
46
47      return l;
48  }
49
50  void wait(int millis){
51       d_startTimer(WAIT_TIMER);
52      while ( d_getTimer(WAIT_TIMER) < millis){
53          //wait for timeout..
54      }
55  }
56
57  #pragma vector=TIMERA0_VECTOR
58  //void timera_isr(void) __interrupt[TIMERA0_VECTOR] {
59   __interrupt void timera_isr(void) {
60
61    global_timer++;
62    //P1OUT ^= 0x01;
63
64    TACTL &= ~0x0001;
65  }
```

```
1  /*********************************************
2  Software Interrupts for MSP430
3
4  Napoleon E. Cornejo
5  ncornejo@gmail.com
6  Delft, NL
7
8  *********************************************/
9
10 #ifndef __SW_INTERRUPTS__
11 #define __SW_INTERRUPTS__
12
13 #define SW_INT_I2CRCV 0
14 #define SW_INT_OTHER1 1
15 #define SW_INT_OTHER2 2
16
17 typedef void (*sw_int_callback)(void);
18
19 void init_sw_interrupts(void);
20 void call_sw_interrupt(unsigned char cause);
21
22 void register_sw_int_callback(unsigned char cause, sw_int_callback callback);
23 void unregister_sw_int_callback(unsigned char cause);
24
25
26 #endif
```

```
1  /*********************************************
```

```
2  Software Interrupts for MSP430
3
4  Napoleon E. Cornejo
5  ncornejo@gmail.com
6  Delft, NL
7
8  ************************************************/
9  #include <msp430x16x.h>
10 #include "cross_studio_io.h"
11 #include "swint.h"
12
13 // the interrupt vector
14 static char SW_INT_VECTOR =0;
15 static sw_int_callback SW_INT_CALLS[8] = {0,0,0,0,0,0,0,0};
16
17 // we use the first bit for a software interrupt
18 void init_sw_interrupts(void){
19    SW_INT_VECTOR =0;
20
21    P2DIR &= ~0x01;  // set bit 0 to 0
22    P2SEL &= ~0x01;  // set function to general input function
23    P2IFG &= ~0x01;  // clear interrupts
24
25 }
26
27 //register a software interrupt callback
28 void register_sw_int_callback(unsigned char cause, sw_int_callback callback){
29   SW_INT_CALLS[cause] = callback;
30 }
31
32 void unregister_sw_int_callback(unsigned char cause){
33   SW_INT_CALLS[cause] = 0;
34 }
35
36 void call_sw_interrupt(unsigned char cause){
37     SW_INT_VECTOR |= (0x01 << cause);
38     P2IFG |= 0x01;
39 }
40
41 #pragma vector=PORT2_VECTOR
42 __interrupt void SWINT(void) {
43 //void SWINT(void) __interrupt[PORT2_VECTOR] {
44
45   //software interrupt
46    if ( P2IFG & 0x01 ) {
47
48
49            // we received something from I2C bus
50            if ( SW_INT_VECTOR & (0x01 << SW_INT_I2CRCV) ) {
51                 SW_INT_VECTOR &= ~(0x01 << SW_INT_I2CRCV);
   //clear the flag
52
53                 // process the commands
54                 if ( SW_INT_CALLS[SW_INT_I2CRCV] )
55                      SW_INT_CALLS[SW_INT_I2CRCV]();
```

```
56
57                  return;
58                }
59
60                //check if no other interrupts pending
61                if ( SW_INT_VECTOR == 0 )
62                    P2IFG &= ~0x01;
63      }
64  }
```

Error Detection Codes for Delfi-n3Xt

B

```
1  #ifndef CRC_H_
2  #define CRC_H_
3
4  /* calculates crc */
5  int crc(volatile char *data, int len, volatile char *pbytes);
6  int comparecrc(volatile char *data, int len, volatile char *edac);
7  int checkcrc(volatile char *data, int len);
8
9  /* these have to be set */
10 #define num_edac_bytes(n) 1
11 #define checkedac comparecrc
12 #define setedac crc
13
14 #endif /*CRC_H_*/
```

```
1  /*********************************************
2   Calculates crc
3
4   Napoleon E. Cornejo
5   ncornejo@gmail.com
6   Delft, NL
7
8   *********************************************/
9
10 #include "n_i2c.h"
11 #ifdef USE_EDAC
12
13 /* Table for polynomial 0x9C */
14 unsigned char crcTable[256] = {         0, 156, 164, 56, 212, 72, 112, 236, 52, 168, 144
15                                 12, 224, 124, 68, 216, 104, 244, 204, 80, 188,
16                                 32, 24, 132, 92, 192, 248, 100, 136, 20, 44,
17                                 176, 208, 76, 116, 232, 4, 152, 160, 60, 228,
18                                 120, 64, 220, 48, 172, 148, 8, 184, 36, 28,
19                                 128, 108, 240, 200, 84, 140, 16, 40, 180, 88,
20                                 196, 252, 96, 60, 160, 152, 4, 232, 116, 76,
21                                 208, 8, 148, 172, 48, 220, 64, 120, 228, 84,
22                                 200, 240, 108, 128, 28, 36, 184, 96, 252, 196,
23                                 88, 180, 40, 16, 140, 236, 112, 72, 212, 56,
24                                 164, 156, 0, 216, 68, 124, 224, 12, 144, 168,
25                                 52, 132, 24, 32, 188, 80, 204, 244, 104, 176,
26                                 44, 20, 136, 100, 248, 192, 92, 120, 228, 220,
27                                 64, 172, 48, 8, 148, 76, 208, 232, 116, 152,
28                                 4, 60, 160, 16, 140, 180, 40, 196, 88, 96,
29                                 252, 36, 184, 128, 28, 240, 108, 84, 200, 168,
30                                 52, 12, 144, 124, 224, 216, 68, 156, 0, 56,
31                                 164, 72, 212, 236, 112, 192, 92, 100, 248, 20,
32                                 136, 176, 44, 244, 104, 80, 204, 32, 188, 132,
```

```
33                                  24, 68, 216, 224, 124, 144, 12, 52, 168, 112,
34                                  236, 212, 72, 164, 56, 0, 156, 44, 176, 136,
35                                  20, 248, 100, 92, 192, 24, 132, 188, 32, 204,
36                                  80, 104, 244, 148, 8, 48, 172, 64, 220, 228,
37                                  120, 160, 60, 4, 152, 116, 232, 208, 76, 252,
38                                  96, 88, 196, 40, 180, 140, 16, 200, 84, 108,
39                                  240, 28, 128, 184, 36};
40
41 int crc(volatile char *data, int len, volatile char *pbytes){
42
43         unsigned char tindex =0;
44     unsigned char remainder = 0;
45     int byte =0;
46
47     for (byte = 0; byte < len; ++byte)
48     {
49         tindex = data[byte] ^ (remainder >> ((8 * sizeof(char)) - 8));
50         remainder = crcTable[tindex] ^ (remainder << 8);
51     }
52
53     /* the crc */
54     pbytes[0] = remainder;
55
56    /* crc only occupies 1 byte */
57    return 1;
58 }
59
60
61 int comparecrc(volatile char *data, int len, volatile char *edac){
62
63         char crc1 =0;
64
65         crc(data, len, &crc1);
66
67         if ( crc1 == edac[0] ) return 1;
68         else return 0;
69 }
70
71 #endif
```

```
1 #ifndef FLETCHER_H_
2 #define FLETCHER_H_
3
4 /* calculates crc */
5 int fletcher(volatile char *data, int len, volatile char *pbytes);
6 int checkfletcher(volatile char *data, int len, volatile char *edac);
7
8 /* these have to be set */
9 #define num_edac_bytes(n) 1
10 #define checkedac checkfletcher
11 #define setedac fletcher
12
13 #endif /*FLETCHER_H_*/
```

```
1 /*********************************************
2 Calculates Fletcher's 8-bit Hash
```

```
3
4  Napoleon E. Cornejo
5  ncornejo@gmail.com
6  Delft, NL
7
8  ************************************************/
9
10 #include "fletcher.h"
11
12 int fletcher(volatile char *data, int len, volatile char *pbytes){
13
14     int sum1 =0, sum2 =0, i =0;
15
16     for ( i =0; i < len; i++ ){
17         sum1 += data[i];
18         sum2 += sum1;
19     }
20
21     pbytes[0] = (sum2 & 0xFF);
22
23     return 1;
24 }
25
26
27 int checkfletcher(volatile char *data, int len, volatile char *edac){
28
29         char hash =0;
30
31         fletcher(data, len, &hash);
32
33         if ( hash == edac[0] ) return 1;
34         else return 0;
35
36 }
```

```
1  #ifndef PEARSON_H_
2  #define PEARSON_H_
3
4  /* calculates crc */
5  int pearson(volatile char *data, int len, volatile char *pbytes);
6  int checkpearson(volatile char *data, int len, volatile char *edac);
7
8  /* these have to be set */
9  #define num_edac_bytes(n) 1
10 #define checkedac checkpearson
11 #define setedac pearson
12
13 #endif /*PEARSON_H_*/
```

```
1  /************************************************
2  Calculates Pearson's Hash
3
4  Napoleon E. Cornejo
5  ncornejo@gmail.com
6  Delft, NL
7
```

```
 8 ***********************************************/
 9
10 #include "n_i2c.h"
11 #ifdef USE_EDAC
12
13 char vtable[] =
14 {
15     1, 87, 49, 12, 176, 178, 102, 166, 121, 193, 6, 84, 249, 230, 44, 163,
16     14, 197, 213, 181, 161, 85, 218, 80, 64, 239, 24, 226, 236, 142, 38, 200,
17     110, 177, 104, 103, 141, 253, 255, 50, 77, 101, 81, 18, 45, 96, 31, 222,
18     25, 107, 190, 70, 86, 237, 240, 34, 72, 242, 20, 214, 244, 227, 149, 235,
19     97, 234, 57, 22, 60, 250, 82, 175, 208, 5, 127, 199, 111, 62, 135, 248,
20         174, 169, 211, 58, 66, 154, 106, 195, 245, 171, 17, 187, 182, 179, 0, 243,
21         132, 56, 148, 75, 128, 133, 158, 100, 130, 126, 91, 13, 153, 246, 216, 219,
22     119, 68, 223, 78, 83, 88, 201, 99, 122, 11, 92, 32, 136, 114, 52, 10,
23     138, 30, 48, 183, 156, 35, 61, 26, 143, 74, 251, 94, 129, 162, 63, 152,
24     170, 7, 115, 167, 241, 206, 3, 150, 55, 59, 151, 220, 90, 53, 23, 131,
25     125, 173, 15, 238, 79, 95, 89, 16, 105, 137, 225, 224, 217, 160, 37, 123,
26     118, 73, 2, 157, 46, 116, 9, 145, 134, 228, 207, 212, 202, 215, 69, 229,
27     27, 188, 67, 124, 168, 252, 42, 4, 29, 108, 21, 247, 19, 205, 39, 203,
28         233, 40, 186, 147, 198, 192, 155, 33, 164, 191, 98, 204, 165, 180, 117, 76,
29     140, 36, 210, 172, 41, 54, 159, 8, 185, 232, 113, 196, 231, 47, 146, 120,
30     51, 65, 28, 144, 254, 221, 93, 189, 194, 139, 112, 43, 71, 109, 184, 209,
31 };
32
33
34 int pearson(volatile char *data, int len, volatile char *pbytes){
35         unsigned char h =len;
36         int i =0, index =0;
37
38         for ( i =0; i < len; i++ ){
39                 index = h ^ data[i];
40                 h = vtable[index];
41         }
42         pbytes[0] = h;
43         return 1;
44 }
45
46 int checkpearson(volatile char *data, int len, volatile char *edac){
47
48         char hash =0;
49
50         pearson(data, len, &hash);
51
52         if ( hash == edac[0] ) return 1;
53         else return 0;
54 }
55
56 #endif
```

Unscented Kalman Filter Matlab Source Code

C

```
1  % Sigma Point Selection algorithm
2  % ================================
3  %
4  % Napoleon E. Cornejo (ncornejo@gmail.com)
5  % Technische Universiteit Delft
6  % Delft, The Netherlands
7  % March 11, 2009
8  %
9  % X = complete set of points defined by the function in a certain range
10 % each row in X represents a point of the function
11 % Px - covariance
12
13 function [sigmas weights] = sigmas(X, Px, W0)
14 sX = size(X);
15 npoints = sX(1);
16
17 alpha=1e-2;                                    %default, tunable
18 beta=2;                                        %default, tunable
19 lambda=alpha^2*npoints-npoints;
20 c=npoints+lambda;
21
22 % dimension of random variable X
23 s = size(X);
24 dimX = s(1,1);
25
26 % initialize sigma/weights points array
27 weights = zeros(2*dimX+1, 1);
28
29 %compute the weights
30 weights(1) = lambda/c;
31 for k=1:dimX
32         weights(k+1,:)= 0.5/c;
33         weights(k+dimX+1,:) = 0.5/c;
34 end
35
36 meanX = X';
37 covX = Px;
38
39 %compute the sigma points
40 sigmas = zeros(2*dimX+1, dimX);
41 sigmas(1,:) = meanX;
42 % sqrt covariance matrix x scaling factor
43 c = sqrt(c);
44 sqcovX = c*sqrtm(covX);
45     for k=1:dimX
46         sigmas(k+1,:)= meanX + sqcovX(k,:);
```

```
        sigmas(k+1+dimX,:) = meanX - sqcovX(k,:);
    end
```

```
% Unscented Kalman Filter
% ==========================
%
% Napoleon E. Cornejo (ncornejo@gmail.com)
% Technische Universiteit Delft
% Delft, The Netherlands
% March 2009
%

% X_init current/initial state points
% Px_init current/initial state covariance
% Y_init current/inital observation/measurement points
% sN state noise
% oN observation (measurement) noise

function [X_next Px_next] = ukalmanf(fstate, X_init, Px_init, fobserv, Y_init, sN, oN)

% state noise w(k) and observation noise v(k) covariances
Q=sN*sN';
R=oN*oN';

% calculate sigma points and associated weights
[SG,W] = sigmas(X_init,Px_init, 0);
dimsg = size(SG);
npoints = dimsg(1);
dimX = dimsg(2);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% nonlinear transformation of sigma points %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Z = zeros(dimX, npoints);
meanZ =0;
for k=1:npoints
    % each transformed point
    Z(:,k) = fstate(SG(k,:)');

    % we obtain the weighted mean from all the transformed points
    meanZ = meanZ + W(k,:)*Z(:,k);
end
% we substract the mean from each innovation point and
normZ = Z - meanZ(:,ones(npoints,1));
% calculate the estimated convariance from the transformed points
Pzz = normZ*diag(W)*normZ' + Q;
%%%%%%%% altenative way %%%%%%%%%%
%Pzz =zeros(size(Q));
%for i=1:npoints
%     nZ = Z(:,k) - meanZ;
%     Pzz = Pzz + nZ*nZ';
%end
%Pzz = Pzz + Q;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
54 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
55 % prediected observations from transformed points %
56 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
57 Y =zeros(dimX, npoints);
58 meanY = 0;
59 for k=1:npoints
60     Y(:,k) = fobserv(Z(:,k));
61     % we get the weighted mean from all the innovation points
62     meanY = meanY + W(k,:)*Y(:,k);
63 end
64 % we substract the mean from each innovation point and
65 % calculate covariance matrix
66 normY = Y - meanY(:,ones(npoints,1));
67 Pyy = normY*diag(W)*normY' + R;
68
69 %%%%%%%%% altenative way %%%%%%%%%%%
70 %Pyy =zeros(size(R));
71 %for i=1:npoints
72 %     nY = Y(:,k) - meanY;
73 %     Pyy = Pyy + W(i)*nY*nY';
74 %end
75 %Pyy = Pyy + R;
76 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
77
78 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
79 % classical kalman filter steps %
80 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
81 % cross covariance matrix
82 Pzy = normZ*diag(W)*normY';
83 %Pzy =zeros(size(normZ*normY'));
84 %for i=1:npoints
85 %     Pzy = Pzy + W(i)*normZ*normY';
86 %end
87
88
89 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
90 % Kalman gain
91 K = Pzy*inv(Pyy);
92 % state prediction
93 X_next = meanZ + K*(Y_init - meanY);
94 % covariance prediction
95 Px_next = Pzz - K*Pzy';
```