Dependable boot and fail safe software for the DTUsat-2

Esben Rugbjerg

Kongens Lyngby 2007
IMM-MASTER-2007-39

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-MASTER: ISSN 0909-3192

Contents

Preface

This report describes the master's thesis project of cand.polyt Esben Rugbjerg. The project has been carried out between September 2006 and April 2007 on Technical University of Denmark at Institute of Informatics and Mathematical Modelling. I would like to thank my counsellor assistant professor Hans Henrik Løvengreen of Informatics and Mathematical Modelling at Technical University of Denmark. The project has been written as a part of the DTUsat-2 project. The DTUsat-2 project's primary goal is to teach students at DTU about the special issues which should be considered when designing a spacecraft. Besides it is also planned to set the built satellite in orbit to solve its scientific mission which is to track birds.

Lyngby d. ______________________________

Contents

List of Figures

Introduction

Background

In the summer 2003 the DTUsat-1 was launched using a reused russian ballistic missile. The DTUsat-1 was a CubeSat satellite which means that it adhere to the CubeSat[HT05] design concept. This concept dictates that the design of the satellite shall conform to a simple set of rules where the most important is that the satellite must be a cube with a side length 10 centimeter.

The concept behind the CubeSat project is to design small satellites with a low weight resulting in keeping the launch expenses low. Another major asset of the design is that the launch system is standardised making it simpler to incorporate the satellites as secondary payload on commercial rockets.

No signals were received from this satellite which means that it for some reason never got into function. The cause of the failure of the DTUsat-1 has never been determined since no information about the state of the satellite could be collected. The satellite is a complex system where a specific combination of several events needs to occur in the correct order if just a single 'beep' shall be sent from it. Because several different subsystems need to work in cooperation to startup the satellite the fault causing the error can be found in any of these[1]. This complicates the error detection and has made it impossible to determine why the satellite never started working.

In the middle of 2005 a new satellite project (the DTUsat-2) was proposed at

[1]No chain is stronger than its weakest link.

DTU. In this project a new cubeSat should be built. In spite of the failure of the first satellite it was decided that the DTUsat-2 should contain a scientific mission as its payload. A payload contest was arranged to get proposals for the payload. The winning project was announced ind November 2005. It suggested that birds (cuckoos) should be tracked on their migratory route by placing GPS transmitters on the back of them and collect the information about their positions using the satellite. Besides that the satellite should also contain a simple camera such that pictures of the earth can be transmitted back from the satellite.

Status

The subsystems of the DTUsat-2 are built as student projects, because the major goal of the satellite project is to educate engineer students in system engineering and aspects of spacecraft design. Both hardware and software are designed and implemented by students. Until now some of the subsystems (the electrical power system (EPS) and the on board computer (OBC)) have been designed and implemented but the radio systems are not yet designed. A lot of the software for the satellite is designed and some is implemented. Parts of the software of the DTUsat-1 has been reused. Some of the software projects are delayed because the complete hardware design is not known yet.

Problem

The most mission critical group of applications developed in the engineering world is that of spacecrafts. These highly developed technical applications contain state of the art solutions in all their aspects. All components of the system are chosen especially for their purpose and most of them are also developed with this specific purpose in mind. This is also the case concerning the software programs running on the computers in, for example a satellite.

When a system is given the predicate *mission critical* it means that if any subsystem fails during operation it is very likely that the mission of the whole system fails. This property results from the fact that it is very difficult to recover the system from the fault e.g. because it is almost impossible to get in physical contact with the system again. This is e.g. the case with satellites. It can also result from a time-wise property for example that some astronomic event only can be observed very rarely.

The software of the DTUsat-1 was divided into the following parts: A *nominal mode* which was recognised by the fact that the operating system (OS) had been booted successfully, and the *fail safe mode* (FS) in which the FS software was executed. Execution of the *boot software* is a part of the FS. This structure of the software has been adopted by the DTUsat-2 project. The nominal mode is quite complex system whereas the boot and FS software is kept as simple and minimalistic as possible.

For this reason the boot and FS software of a satellite is considered mission critical. The task of the program is to startup the satellite and ensure that all subsystems function correctly before the OS of the satellite is started. If for some reason the OS cannot be started for example due to some subsystem not working as expected or the OS itself fails, the FS program should be started to handle this situation. Therefore the boot and FS software is the most mission critical software on the satellite: If the boot software does not function correctly the satellite will never start, and if the FS software does not function correctly the satellite cannot be recovered from any failures caused by some of the other software on the satellite or one of the hardware subsystems. In both situations the satellite and thereby its mission will fail.

The main task in this project is to design and implement the boot and FS software for the DTUsat-2. The program should be as complete as possible meaning that as much of the functionality of the program as possible should be designed and implemented. The goal is a highly dependable, operational program which can be used on the DTUsat-2 when it is completed with as few modifications as possible.

Approach

The boot and FS software needs to be very robust and dependable and it should be developed to an operational level in this project. These three properties have been decisive for the choice of approach in the project.

In order to achieve the robustness and dependability of the program best practice methods of software development have been applied during the project. The choice of the used software tools and tool chain also reflects this. As explained later in chapter 7 the GCC tool chain has been chosen among other reasons because it is open source software which enables us to inspect the source code if some problems demands it. Another and even more important reason is that the GCC tool chain is considered very reliable due to its long development history.

The project has been carried out as a classic software project with an analysis phase, a design phase and an implementation phase, since this is an approved procedure and since I am familiar with this method.

Through the analysis and design phases of this project an operational approach has been chosen. This approach has been based on classical “pen and paper”-pseudo-coding. This working method is in contrast to the more academic method where a formal model of the whole satellite and the program running on it would have been developed. The temporal properties of this model would have been verified using a model checker. The verified model could then constitute the outline for the implementation of the program in the C programming language. The time frame of the project did not allow such a model to be developed. Another reason why this approach was abandoned was that the modelling approach is not believed to lead to the desired product i.e. an operational program. These two reasons entailed that a more operational approach was chosen. It was tried to combine the described approach with the usage of available software tools whenever they were found to simplify specific tasks in the project or raise the robustness or dependability of the final product. It was however not found feasible to use any software tools to handle any larger issues during the software development process. Neither suitable problems nor tools were found.

The software of the DTUsat-1 has been used as inspiration and the FS part has been modified and merged into the DTUsat-2 software. This has been done partly because this part of the program was well designed and tested and due to lack of time in the present project to write the everything from scratch.

The report

The report describes the development of the boot and FS software of the DTUsat-2. The first chapter contains short introduction to dependability in software which is a main property of mission and safety critical software. The second chapter contains a description of the OBC and the test board used during development of the software. The third chapter contains an analysis of the boot situation of the satellite and outline the requirements of the boot and FS software. The fourth chapter describes the development of the memory test used during the boot procedure. The fifth chapter describes the development of other parts of the boot and FS software. The sixth chapter contains a description of methods used while testing and building the software. The seventh chapter describes the attempts to verify the soundness of the protocol used to communicate between the on board computer (OBC) and the radio (COMM) subsystem of the satellite using the model checker Uppaal. Finally a conclusion describing the obtained result and the properties of the developed software are given in chapter eight. After that some appendices are given containing the source code of the program as well as test output etc.

In this report a lot of references to background information which is not necessarily interesting when reading the report as the documentation of a master's project are given.

The reason for this is that the report is written with more than this purpose in mind. Besides being the documentation of a master's thesis it is also intended to be the primary source of information about the boot system and its internals for the students who shall complete the system when the hardware platform of the DTUsat-2 will be finished. Emphasis has therefore been put into collecting all relevant references. In addition to this an index and a glossary are also available. In the index the names of the C functions are listed together with links to the source code and their prototype. The beginning positions of the C functions are also listed in the table of contents. The glossary contains explanations of the abbreviations used during the report together with definitions of some central terms in the project.

Chapter 1

Dependability

Dependability is a central subject in computer technology especially concerning embedded systems. The reason for this is that most applications of embedded systems possess some kind of safety or mission critical properties.

This chapter is divided into two main sections. The first section contains a brief description of the basic of dependability notions and the second section introduces some methods which can be used to raise the dependability of a system. A more thorough survey than the one given in this chapter is presented in [GL02].

1.1 Dependability notions

Dependability of computer systems has been a research subject since the beginning of the development of electronic computers. The reason is that computer scientists always have tried to raise the dependability of the systems such that the results computed by the computer could be trusted and such that the computers could be trusted to work at all. Later when computers were used to control systems where failures could be lethal to humans this research subject became even more important.

1.1.1 The framework of Dependability theory

A good starting point for getting an overview over the notions used in dependability theory is the *dependability tree* which is a simple tree structure showing the relations between the central notions in dependability. It is shown in figure 1.1 which is copied from figure 2.1 of [ALR04].

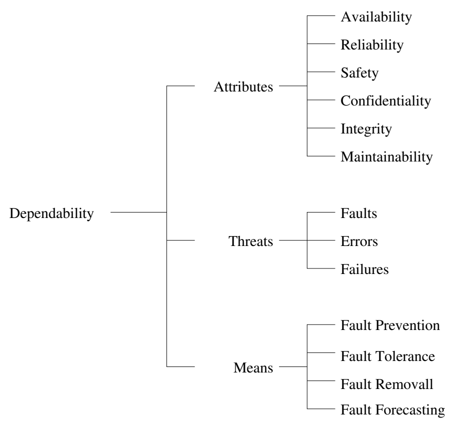


Figure 1.1: The dependability tree. Copied from figure 2.1 of [ALR04].

Figure 1.1 shows that the three central areas of dependability theory are:

Dependability attributes Dependability attributes are the quantities which a system's dependability is measured in. The attributes are not necessarily given a specific value but the quantities constitute a framework which a system's dependability can be assessed in.

Dependability threats The threats are faults, errors and failures.

Dependability means The means are the tools available to raise the dependability of a system.

1.1.2 The Attributes

The dependability attributes are a set of attributes which can be described for a specific system. It does not always make sense to try to determine the value of all attributes for a specific system. For example it is not meaningful to determine the safety [1] of the DTUsat-2 because it will be very far away and no people will ever get in physical contact or even close to it while it is in service.

A central attribute to the DTUsat-2 project is the *availability* which is defined as:

> readiness for correct service

in [ALR04, section 2.3]. The most important attribute is the *reliability*. The reliability is defined as the system's exhibits of:

> continuity of correct service

in [ALR04, section 2.3]. The availability of the DTUsat-2 is raised by ensure the robustness of the EPS and including a battery in the system's design of the satellite etc.

The attributes *confidentiality* and *Integrity* are together with *Availability* closely connected to the IT security area. *confidentiality* is defined as:

> absence of unauthorized disclosure of information

and *integrity* as the:

> absence of improper system alterations

Both quoted from [ALR04, section 2.3].

Maintainability is defined in [ALR04, section 2.3] as the:

> ability to undergo modifications and repairs

Even though it is impossible to get in physical contact with the satellite when it has been launched this attribute is interesting even in the case of a satellite.

[1] defined as "absence of catastrophic consequences on the user(s) and the environment" in [ALR04, section 2.3].

On the DTUsat-2 as little of the software as possible is stored in a ROM, and as much as possible in a FLASH memory. The software stored in the ROM is made capable of writing to the FLASH memory making it possible to erase the entire OS and application software of the satellite and upload a new version. In this sense the satellite has the highest possible maintainability.

1.1.3 The notions of faults, errors and failures

In normal conversations the three terms fault, error and failure is almost interchangeable or at least the two first are. This is not the case in dependability theory where each term has a specific meaning.

In dependability theory a *failure* is the consequences to the surrounding or system caused by some wrong internal state of the system. The wrong internal state is called the *error*. The cause of the error is called a *fault*. This distinction will be illustrated in a small case study.

1.1.3.1 The Mars Climate Orbiter incident

September 23th 1999 Nasa's Mars Climate Orbiter was navigated into orbit around Mars. Unfortunately it was lost during this maneuver. The following investigation showed that the reason for the loss was that the altitude of the satellite was to low. It should have entered an orbit with an altitude of 150 kilometers above the surface of the planet but the actually altitude was only about 60 kilometers which had catastrophic consequences for the satellite [MCO99].

In the following investigation it was revealed that the cause of the low altitude was a fault where one team of engineers used imperial units and another team used the metric system [IHU99].

In this example the **fault** is the usage of the imperial units instead of the metric system. The **error** is the low altitude of the spacecraft and the **failure** the spacecraft's crash into the surface of Mars.

1.1.4 The means available in dependability

As it has been implied indirectly there are different techniques to improve dependability of a system. All these techniques can be classified into one of the following groups (copied from [ALR04, section 2.4]:

fault prevention are means to prevent the occurrence or introduction of faults.

fault tolerance are means to avoid service failures in the presence of faults.

fault removal are means to reduce the number and severity of faults.

fault forecasting are means to estimate the present number, the future incidence, and the likely consequences of faults.

In the boot and FS software on the DTUsat-2 two major **fault tolerance** methods have been implemented. The first is the memory test (described in chapter 4) which tries to prevent the usage of defective memory locations. The second is the use of a WDT which can recover the system if it gets trapped in an infinite loop. In the rest of this chapter some fault prevention and fault tolerance methods will be presented.

1.2 The *KIS(S)* principle

The kis(s) (Keep It Simple (stupid!)) principle is not a real design paradigm but a guiding rule used when choosing between different design options. The principle require the designer to choose the simplest solution which solves the problem, avoiding adding unnecessary features and complexity. The principle can also be applied to writing style of the source code where the programmer is demanded to keep the code as simple and self explained as possible. The principle is also connected to Occam's razor [kis06].

The main design principle which should adhered in all design processes concerning parts of the DTUsat-2 project is the *kis(s)* principle. This principle should be adhered even when other design paradigms are applied.

1.3 Redundancy

Redundancy can be applied to both software and hardware. It can be applied in both a parallel (computational) and a serial (temporal) manner.

The parallel redundancy is normally applied as hardware redundancy where several computers compute the answer to the same problem using different methods or algorithms. When all computers have computed an answer the answers are compared and the answer which occur most often is used. The method is based on the assumption that if an answer occur several times even if it is computed using different methods it must be correct. The method can also be described as a voting process where the computers vote on the solutions to a problem.

The temporal redundancy is normally applied as software redundancy. Empty slots of time are added to the schedule making it possible to rerun processes which fail. Another solution is computing the solution to the problem again using a simpler algorithm.

The size of the DTUsat-2 and the available resources of energy and therefore computing resources does not allow any kind of hardware redundancy to be implemented in the satellite.

In general both software and hardware redundancy adds a lot of complexity to a system if applied. This argument constitute another major reason for avoiding software redundancy on DTUsat-2, since the kis(s) principle should always be adhered. This does not mean that software redundancy in general cannot be applied to a system in combination with the kis(s) principle but that it has not been found feasible in this particularly project.

1.4 Graceful service degradation

Graceful service degradation means to adapt the quality of a service based on the available computing resources and available functional subsystems. Graceful service degradation is closely related to *reconfiguration*. Reconfiguration means to restore a stalled system to some operational state, avoiding use of subsystems which contain permanent faults. Reconfiguration is primarily applied on the hardware level.

Graceful service degradation on software level may be based on using simpler or slower algorithms to compute the solution. In this case a trade off between quality of the produced solution and ability to deliver a solution at all is made.

On the DTUsat-2 graceful service degradation is used as a design principle by adding a fail safe mode in addition to the nominal mode.

1.5 Available tools

Several software tools have been developed to help developing more dependable software. In this section a short summary of some of the available tools will be given. The tools have been chosen for their relevance to the DTUsat-2 boot and FS software.

None of the tools have actually been used during the development of the boot and FS software development due to lack of time.

1.5.1 Stack size calculation tools

Stack usage tools analyse the binary code to estimate the worst case stack size. This is relevant when planing memory usage of the system. The stack usage analysis is a static test meaning that the worst case stack usage is computed for the specific binary file not inferring any kind of equation for the stack usage.

The reason for doing the analysis on the binary is that the compiler optimisation can influence the stack size, hence it is not meaningful to do the calculations on the source code.

The stack size analysis makes it possible to prevent stack overflows and is hence a good fault prevention method. An example of a stack analysing tool is the commercial tool StackAnalyzer (see [stA07]).

Another solution is to do the calculation manually by constructing a test case which gives rise to the largest possible stack usage, and then inspect the memory by a debugging tool when the test program has been run. This method is outlined in [ARM] together with other similar methods. This solution does not lead to a verifiable calculation of stack size but only to an estimation.

1.6 Coding rules

Coding rules are conventions and rules for how the source code should be written. They include rules about which language constructs are allowed, how variables and functions are named etc. Coding rules have two purposes: Prevent the usage of “dangerous” constructions in the source code and raise the uniformity between code written by different developers.

An example of a classic coding rule is that **while()** loops are not allowed, because they contain the ability to go into infinite loops. All loops should be bound to finite upper level of iterations.

Especially in the case of development of mission or safety critical software compliance of coding rules are required. An example of this is the "C and C++ Coding Standards" of the European Space Agency [esa00]. This report contains 113 rules and recommendations on how to write understandable, portable and safe C and C++ code. The rules and recommendations are based on experiences more than theoretical or strictly analytical work. A more thorough survey on design and coding constraints is given in [PP02].

1.7 Source code inspection tools

A group of fail prevention tools do an automated code inspection of the source code. Several different programs exist. In this section short descriptions of two of the tools are given. The tools have been chosen for the following reasons: Security Programming Lint (Splint) implements a lot of tests also implemented in an older and popular tool called "Lint", and it is optionally annotation driven.

The Berkeley Lazy Abstraction Software Verification Tool (BLAST) is a annotation driven model checker for C source code.

1.7.1 Splint

Splint ([spl]) is a source inspection tool. It reads the source files and searches for unused declarations, type inconsistencies, use before definition etc. [Sec, p. 9]. The coding mistakes searched for by Splint is often the same as the coding rules mentioned in section 1.6 try to avoid.

Splint is capable of finding a lot different coding mistakes. Unfortunately it produces a lot of false positives. The most of these false positives can be removed if annotations are added to the source code. These annotations helps Splint to determine which of the potentially faults are actually faults and which are code constructions intended by the programmer.

An example is the **/*@null@*/** which tells Splint that the pointer declared on the next line is intended to contain null pointer in some cases. As it can be seen the annotations are added to the source code as comments. Therefore the program can be compiled directly.

```
/*@null@*/
int * counter;
```

1.7.2 BLAST

The Berkeley Lazy Abstraction Software Verification Tool (BLAST) (see [BLA05] is a model checker for C source code. The model-checker can verify that invariants are not violated during execution of the code. The code is executed in a symbolic fashion by the model checker.

BLAST also contains a scripting language which makes it possible to carry out more advanced tests where more than one invariant needs to be tested at the same time. Using the scripting language also avoids applying changes to the source code.

BLAST contains some advanced features which will not be discussed here.

CHAPTER 2

System description

The brain of the satellite is the OBC. Physically the OBC is a circuit board. This board is equipped with the central CPU chip (described in 2.1.1), the external memory chips (SRAM and FLASH, described in 2.1.1.3), an external watch dog timer (WDT)[1] and some components controlling the energy supplies.

The energy for the subsystems of the satellite is supplied by the electric power system (EPS), which therefore must be considered the heart of the satellite. This board is connected to the solar panels, the batteries, the sun sensor board, the Altitude Control System sensor board (ACS sensor board), the ACS board[2] and all other subsystems. This board contains all central electrical systems and power connections.

2.1 The development board

This section contains a description of the hardware system used during the software development. The system is an Olimex LPC-E2294 development prototype board [oli06] equipped with a Philips LPC2294 ARM CPU chip and several pe-

[1]no information is available on this device yet.
[2]The "ACS sensor board" and the "ACS board" are two diffenrent boards.

ripherals for example a 16x2 character LCD display. Where no other thing is mentioned the information given in this chapter also complies to the OBC of the DTUsat-2. No information will be given about the interface between the programming PC and the development board.

2.1.1 The CPU chip

The CPU chip on the development board which has been used during the development of the software is a Philips LPC2294. Besides the processor it contains all necessary devices to build a simple embedded system. In this section a description of the relevant subsystems on the chip is presented. The CPU chip on the DTUsat-2 is a Philips LPC2292. The only difference between the two chips is that the LPC2294 contains four CAN bus controllers whereas the LPC2292 only contains 2 CAN bus controllers.

2.1.1.1 The processor core

The processor core implements the ARM7TDMI architecture which is a true 32-bits architecture with both 32-bits instruction set and memory space. It is a RISC architecture which does not contain any floating point (FP) instructions. The CPU contains 13 general purpose registers. The ARM7TDMI is a pipelined architecture having three stages (fetch, decode and execute).

Besides the 32-bits instruction set the processor is also capable of executing the 16-bits *thumb* instruction set. This instruction set was developed to offer more compact binary code than the 32-bit code offers.

2.1.1.2 The clock system

The crystal which supplies clock pulses to the CPU chip runs at 14.7456 MHz. This clock is passed to the phase locked loop (PLL) circuit, which is used to increase the clock frequency of the CPU core. This frequency is referenced as the *cclk* in the documentation which gives a more thorough description of the use and configuration of the PLL in [Phi03, page 75]. An example of how to calculate the constants needed to setup the PLL is also given in [Lyn05, section 15]. Besides the *cclk* the chip has a peripheral clock net. This clock is referenced to as the *pclk*. The frequency of this clock is controlled by the *VLSI peripheral* *bus (VPB)*[3] *clock divider* (VPB divider). How to configure the VPB divider is described in page 86 of [Phi03]. the *pclk* is for example used by the WDT. The frequency of the *pclk* also depends on the configuration of the PLL since it is generated by dividing the *cclk*.

2.1.1.3 The memory system

The LPC2294 contains both internal FLASH memory and internal static RAM. Besides that, it is possible to connect external memory devices. On page 48 and 49 in [Phi03] two complete memory maps of the memory in the LPC2294 chip are given. The memory system contains no cache stages, but the bus system includes a memory accelerator module (MAM) which enables execution of sequential code at the speed of the internal clock (*cclck*). This module is described further in section 2.1.1.3. A more complete describtion of the memory layout of the system is given in section 2.3.

The RAM The internal static RAM is 16 KB in size. It starts at address 0x40000000 and ends at 0x40003FFF.

The FLASH memory The internal FLASH of the LPC2294 chip has a storage capacity of 256 KB. It is mapped into the address space from address 0 to address 0x003FFFF. The top 8 KB of the FLASH is reserved for the boot loader program and should not be erased or used for anything else. At the lowest 64 bytes of the FLASH (starting at address 0x00000000) the interrupt vectors should be placed. At address 0x00000000 the reset interrupt vector should be placed and it should contain a jump function to the entry point of the boot program.

The memory accelerator module The memory accelerator module (MAM) speeds up reading of the FLASH memory by prefetching 128 bit in each read operation and latch them for faster answering on instruction fetch requests from the CPU. The FLASH memory is divided into two banks each having a 128-bit latch. This enables a switch behavior where one bank is reading 128 bit of instruction data while the other bank is prefetching the next 128-bit. The MAM is described in [Phi03, page 90 - 93].

[3]Described in [Phi03] on page 18

2.1.1.4 The internal watch dog timer

The internal watch dog timer (WDT) is used as the primary WDT during software development. The WDT can be setup to have a timeout period between $69.4\mu\,seconds$ and $1165.08 seconds$ ($= 19.41 minutes$) if the $pclk$ runs at 14.7456 Mhz.

The WDT contains a mode and status register referenced as **WDMOD** in the documentation. The value of this register is maintained during a reset if the power of the chip is not disconnected or interrupted. From this register it is possible to determine whether the WDT induced the reset or not.

The WDT is described more thoroughly in [Phi03, pp. 256].

2.1.1.5 The real time clock

The real time clock (RTC) is a clock system using normal time units like hours, minutes, month and year. It is possible to read the time from the clock's counter registers. The RTC can also be setup as an alarm by providing interrupt generation based on value matching between its counters and a set of register which contain the desired time for the alarm.

The value of the counters is not maintained during power off. If the WDT reset the system only a delay as long as it takes the system to startup and initialise the RTC will occur.

The RTC is described more thoroughly in [Phi03, pp. 242].

2.1.1.6 The boot loader

Every time the CPU chip is turned on or reset a boot loader program laying in the internal FLASH memory is executed. This program executes before any user program is executed.

If the 'BSL' jumper is set the boot loader calls its "In-System Programming" (ISP) functions. These functions is used to program the internal FLASH i.e. uploading a new user program. This is done through a UART of the chip.

The boot loader program also contains functions which are used to program the internal FLASH through a program running on the CPU. In this way unused memory of the internal FLASH can be utilised by the user program. The mentioned functions are called “In-Application Programming” (IAP) functions and are described in [Phi03, pp. 262].

An example of the usage of the IAP functions is presented in 5.5. Through out the rest of the report the boot loader program is referenced as the “boot loader” or the “internal boot loader”. The program developed in the project is referenced to as the “boot program”.

2.2 The externally connected peripherals

Several external peripherals are connected to the CPU. All subsystems of the satellite can be considered peripherals if the OBC controls them. As all subsystems have not yet been designed an exhaustive description of all peripherals and subsystems is not provided here. Only peripherals and subsystems which are designed, implemented and relevant for the boot or FS software are described.

2.2.1 External memory

The external memory is accessed through the external memory controller (EMC). It provides the possibility to have four banks of 16 MB each. Bank number 0 begins at address 0x80000000. The interface of the EMC is described in details in [Phi03, pp. 56].

It is possible to change the boot behavior of the chip such that the boot loader boots the boot program from address 0 of the first external memory bank. This would be address 0x80000000. See [Phi03, p. 133] for details.

The ability to change the boot address of the system will be exploited on the satellite to boot the boot software stored in a ROM. This ROM is connected as memory bank zero of the EMC, starting at address 0x80000000. The interrupt vectors are placed from the beginning of this and 64 bytes up and starts with the RESET interrupt vector at address 0x80000000.

External RAM On the development board 1 MB of external S-RAM is mounted. This is 10 ns devices of the type: “K6R4016V1D” [sam04] and it is manufactured by Samsung.

The external RAM on the OBC is static RAM as the internal RAM is. It is connected as the external memory bank one through the EMC. The size of the RAM is two MB and it starts at address 0x81000000 and ends at address 0x8101FFFFF. Details on the configuration of the communication between the EMC and the external RAM are found in [Phi03, pp. 56].

The external RAM is primarily used by the OS but is also used as stack area for the boot and FS software if the whole internal RAM area is corrupted.

External FLASH memory The development board is equipped with 4 MB of external FLASH memory. This memory is connected to the bank zero of the EMC. The external FLASH is of the type “Intel Advanced+ Boot Block Flash Memory 28F160C3” as described in [Int05].

The external FLASH memory of the OBC has a storage capacity of 2 MB and is manufactured by Intel. The type of the FLASH is “Intel Advanced+ Boot Block Flash Memory 28F160C3” and it is described in [Int05]. The purpose of the external FLASH is to store data collected by the OBDH. It will be connected as bank one of two of the EMC.

2.2.2 The GPIO interface

The GPIO interface is a port based interface. It is only used for the hold signal send to the COMMpic because as much as possible of the communication between the OBC and the other subsystems should be transmitted over standard bus systems. This is a design goal of the DTUsat-2. The GPIO interface is described in [Phi03, page 134 - 137].

2.2.3 The CAN bus and SPI interface

The CAN bus is used to communicate with the other subsystems on the satellite. Two different CAN bus controllers resulting in two individual channels are available on the LPC2292 chip, where four are available on the LPC2294 chip. The controllers of the CPU chip are described in
[Phi03, page 188 - 210].

The SPI interface is also a hardware interface. It is used to connect the COMM system to the OBC.

2.3 The memory layout

During execution of the boot program and other programs which use the IAP routines to write data to the internal FLASH memory of the CPU chip, areas of the RAM is used by these routines. This should be taken into account in the design of the memory layout. Also the usage of areas by the stacks of the exception routines should be considered when prioritising the allocation of the RAM.

2.3.1 Memory area used by the boot loader of the chip

The memory layout concerning the ISP and IAP routines can be seen in figure 2.1.

The boot loader of the chip uses some of the internal RAM when some of its code is executed. The ISP routines are used to write to the FLASH memory when uploading a program to the system. Therefore these routines are only used when no program is executed on the processor. The IAP routines are used to do operations on the FLASH through a user program. Besides the memory especially reserved to the IAP routines it also use some memory on the normal program stack. This amounts to be at most 128 bytes [Phi03, page 265].

Besides the FLASH writing routines the boot loader also provides a debug interface called *RealMonitor* which is especially suited for real time debugging. This is not used during development in this case due to lack of Linux based debugging applications which supports the RealMonitor protocol. The memory use of the system is shown in figure 2.1 for completeness only.

2.3.2 The memory areas used by exception routines

Several exception types exist in the ARM architecture. A very important type is the interrupt. In this section the terms exception vectors and interrupt vectors are interchangeably.

When handling exceptions the ARM processor changes into other modes only used for this purpose. These modes have their own set of special purpose registers for example the stack pointer (SP) register. This design is chosen to speed up handling of interrupts and exceptions by avoiding time consuming context shifts. To support this design, special areas are reserved for the stacks used by these routines. The area containing these stacks has been placed at the top of the RAM area.

The area in the bottom of the internal RAM is used to store the remapped exception vectors, see [Phi03, page 52]. There is two reasons for this solution: speed, since the RAM responds faster than the FLASH memory, and security: If one of the IAP routines are called by a user program, the internal FLASH enters a busy mode making it impossible to read from it. Therefore if an interrupt occurs while the IAP routine is executed, it will not be possible for the system to read the exception vectors from the FLASH memory. Instead it is possible to read them from the RAM if they are also stored there.

The size of the area used by the exception vectors is 64 bytes. Since the interrupts are disabled during boot this area is not used. When booting the system, the boot loader of the chip sets up the MEMMAP register which controls the remapping, see [Phi03, page 74], such that no remapping is activated.

The area in the top is the actual stack areas used by the interrupt serving routines. If the interrupts are disabled they can be reduced to 4 bytes each. It is necessary to have unique stacks for the *undefined instruction mode*, the *abort mode*, the *fast interrupt* (FIQ) mode, the *interrupt* (IRQ) mode, and the *supervisor mode*. The user mode and the system mode share their stack. The user and system stack is a real stack laying below the other pseudo stacks and grows downwards. As it can be seen five pseudo stacks are needed, taking up four bytes each, using twenty bytes all included. This is described in more details in [Lyn05, section 14]. These twenty bytes are taken from the area available for boot and FS program, The memory layout of the user space can be seen in figure 2.2.

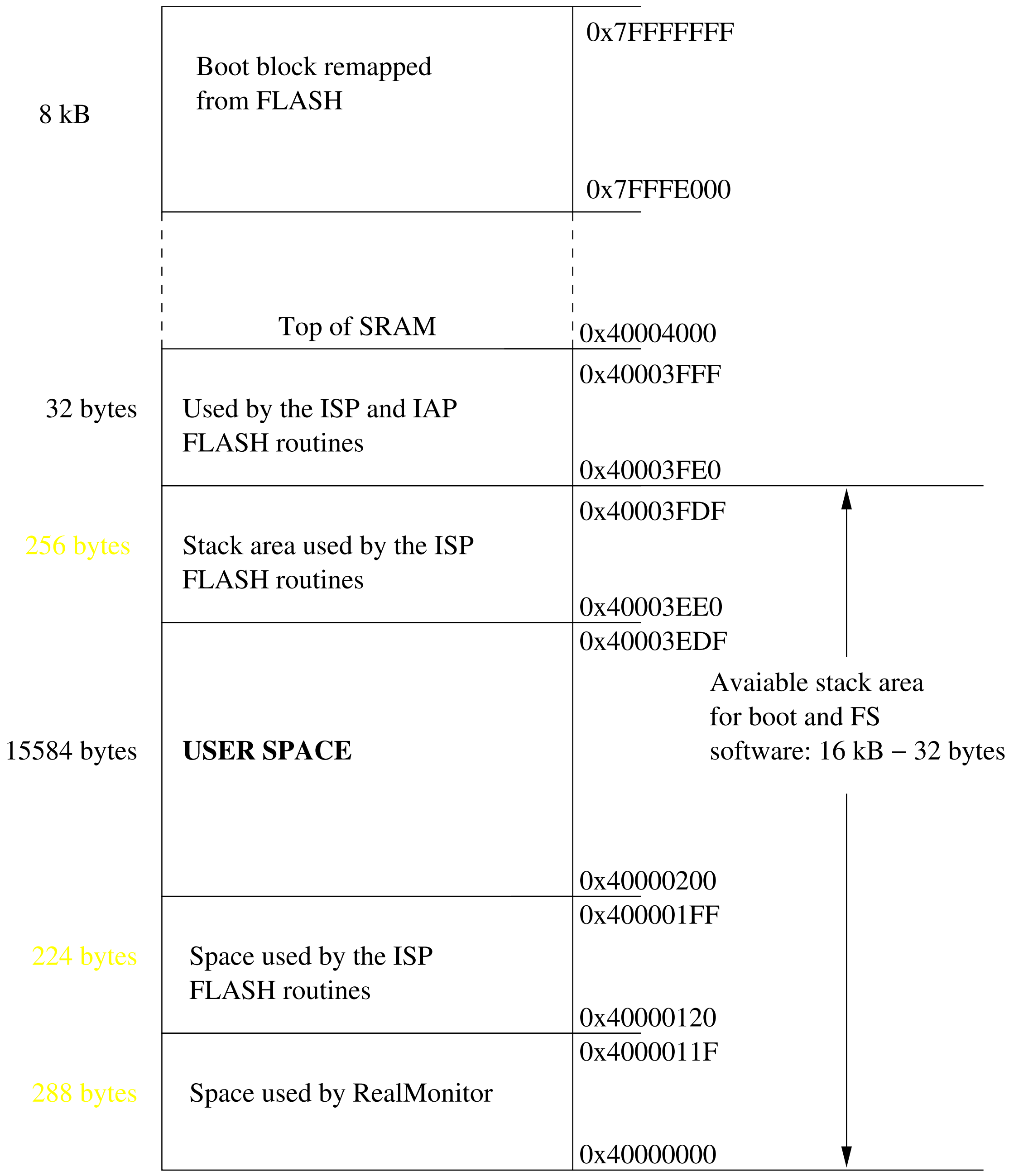

8 kB
Boot block remapped from FLASH
0x7FFFFFFF
0x7FFFE000
Top of SRAM
0x40004000
32 bytes
Used by the ISP and IAP FLASH routines
0x40003FFF
0x40003FE0
256 bytes
Stack area used by the ISP FLASH routines
0x40003FDF
0x40003EE0
15584 bytes
USER SPACE
0x40003EDF
Avaiable stack area for boot and FS software: 16 kB – 32 bytes
0x40000200
224 bytes
Space used by the ISP FLASH routines
0x400001FF
0x40000120
288 bytes
Space used by RealMonitor
0x4000011F
0x40000000

Figure 2.1: The memory layout of the RAM concerning the routines provided by the boot loader. The areas written in yellow is not used during execution of a program and thereby also available to the program together with the user space.

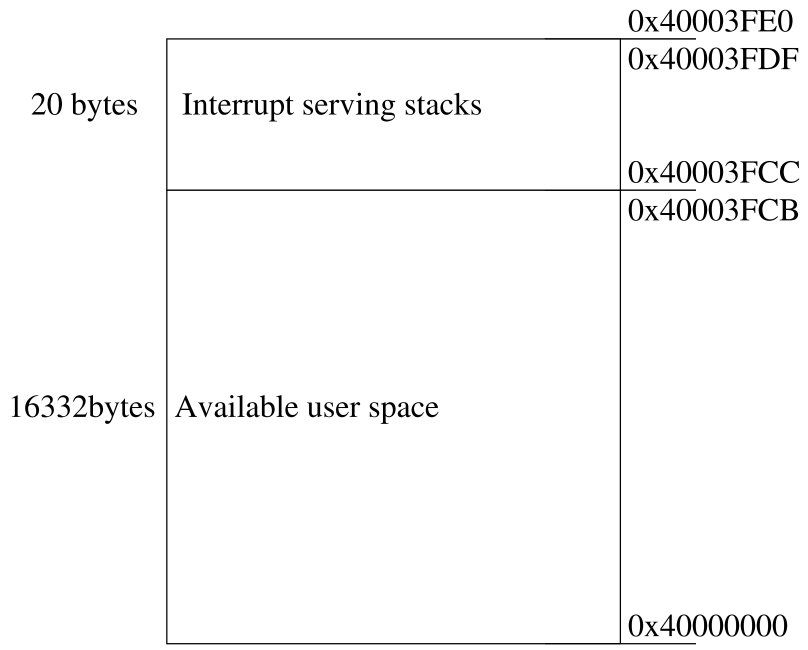


Figure 2.2: The memory layout of the user space just after boot time.

Chapter 3

Requirement analysis of the boot procedure

This chapter contains a requirements analysis of the boot process of the DTUsat-2 regarding requirements of the boot software which carry out and control the boot procedure.

The following description of the launch lacks some data since some informations about the launch and subsystems is not available yet.

3.1 The choice of implementation languages

The choice of implementation language is a central design decision. In this project dependability, robustness and simplicity are central demands to the implemented program. Therefore the implementation language should support these three properties. To ensure these three properties the programmer should have as much control over the program as possible. This is accomplished by using a simple programming language where only the most simple things are taken care of by the compiler.

Another central characteristic of the candidate language is that it should run directly on the CPU since no OS is available yet, why facilities such as memory management not yet is available.

3.1.1 The assembly language

These demands point in the direction of the assembly language. This language demands the programmer of controlling almost everything and give him the most freedom to control the computer. Another very important property is that it runs directly on the CPU.

Unfortunately the assembly language also makes things very complicated as a consequence of the vast quantity of things the programmer needs to control. Therefore only small programs where the special facilities which the language offers to the programmer's disposal are needed, should be programmed in the assembly language.

3.1.2 High level languages

Several high level languages and there derived dialects are available as candidates: C and Pascal etc. C is known to be very robust and it gives a lot of freedom to the programmer. Both languages are supported by open source software and free compilers. Both languages can be compiled to run directly on the CPU without needing any facilities supplied by an OS.

The C language is familiar to the programmer which Pascal is not. This measure is the deciding one since one additional way to make a program dependable is by letting the programmers work in a well known environment.

Both C and Pascal are stack based programming languages meaning that they both need a memory area to store variable values during execution. Therefore they both need that a memory area is prepared for this purpose before a program is called.

3.1.3 Conclusion

The assembly language can run directly on a CPU without needing any stack area but programs written in assembly language normally get very complex when they grow large.

High level languages on the other hand keep programming issues fairly simple even in large programs. Unfortunately they need an initialised system in order to execute.

This leads to a situation where a combination of a low level language as the assembly language to control the initialisation of the system and a high level language as C to implement the advanced tasks in, is the optimal solution. This combination is therefore chosen.

3.2 Before launch

When the final design of the DTUsat-2 is finalised it is built and assembled in Denmark.

As a part of the assembling the final version of the boot and FS software is built.

After the assembling another test procedure of the final satellite is carried out. During this procedure, it is necessary to be able to communicate with FS software to upload data and applications and download test results. Therefore the FS software should contain facilities to support this.

When the satellite has been tested it is sent to the launch location where it is prepared for launch. After that it is placed in the deployment system called a P-POD, see [HT05]. In the P-POD more tests and preparations can be carried out. At this point it should be ensured that the system is setup correctly.

- The satellite should be able to communicate before through a wired connection.
- The satellite should have facilities implemented which makes it possible to communicate with when it is placed in the P-POD.

3.3 After launch

After the satellite has been released from the launch vehicle, the kill switch will be turned on and the EPS should test the voltage level. If the voltage level of the batteries is high enough the EPS should turn on the rest of the sub systems on the satellite inclusive the OBC. If the voltage level not is high enough the EPS will start charging the batteries instead and turn on the rest of the subsystems when the voltage level has reached the correct value.

- The boot software should test the battery level and only start the OS if it exceeds a certain level. If it does not exceeds this level the FS software should be started.

3.3.1 Exception and interrupt handling during the boot process

The ARM7TDMI architecture supports hardware exceptions. This subject is closely related to the notion of running the CPU in a *priviliged mode*. The CPU is able to run in the following modes:

User mode Unprivileged mode which normal user applications should run in.

Fast interrupt mode(FIQ) Mode to handle *fast interrupts*. This mode has a large set of its own registers in the CPU. This enables a faster context shift than normally possible.

Interrupt mode Mode to process normal interrupts.

Supervisor mode (SVC) Also called **software interrupts mode**. This mode is a protected mode and used to handle software interrupts in .

Abort mode Mode to handle memory faults.

Undefined mode Mode to handle undefined instructions.

System mode Privileged mode used by the operating system.

The different type of exceptions can be divided into two groups: A group of control able exceptions i.e. the ones which can be disabled. This is the software interrupts, the IRQ and FIQ interrupts. The second group cannot be disabled and consists of reset, undefined instruction, prefetch abort and data abort.

In general the exceptions should be avoided because they raise unpredictability of the system.

The first group should be disabled as the first thing in the boot procedure. The reason for this is to raise the predictability of the system but even more important it prevents failures caused by the interrupt routines. Since no execution stack is available early in the boot process it is very difficult to implement any kind of exception handling routines. This is caused by the fact that no memory to store the values of the registers is available. Therefore any alteration of any of the registers by an interrupt routine may lead to malfunction and failure of the software after the return from the interrupt routine.

- The interrupt system should be turned off as early as possible in the boot process.

The second group of exceptions consists of the exceptions which is caused by abnormalities in the execution of the code. The only way to prevent these are by designing a well structured program. Even when this is done these exceptions could occur anyway. An 'undefined instruction' -exception could occur because of a bit flip in the memory. If this happens the system should be able handle the exception in a controlled manner.

- The system should able to handle all exceptions in a controlled manner.

3.3.2 Initialisation of the WDT

Another important task which should be carried out as early as possible in the boot process is to setup and start of the WDT. This is necessary to recover the system from hardware failures during the earliest phases of the boot procedure.

- The WDT should be initialised and turned on as early as possible in the boot process.

3.3.3 Location of the C stack

A major task of the boot software is to set up a C stack on which the rest of the boot and FS program can be executed on. Therefore the boot program needs to choose an area of the memory to host the stack. The most simple approach is to place the stack in the same area at every boot i.e. give the stack a static location. This is not the optimal solution since the chosen memory area could get damaged by cosmic radiation and be rendered useless. If this happened the satellite would loose its ability to boot.

Instead the boot program should place the stack in a flawless area of the memory. Before the program can do that, it needs to ensure that the chosen memory area is actually fully functional and contains no defective memory cells. To test this the boot software should carry out a memory test on the memory area used to store the stack before the stack is setup.

All the above tasks needs to be carried out by a program implemented in the assembly language because no C stack is available.

- The boot program should setup a C stack. Before doing that it should ensure that the chosen area is fault free.

3.3.4 Task carried out by the C program

After the C stack has been established the rest of the tasks during the boot process can be carried out by a program implemented in the C programming language.

3.3.4.1 The silence period

According to the CubeSat Design Specification [HT05], the satellite must stay silent i.e. send no beacons or anything else for the first fifteen minutes after it has been released from the launch vehicle, and it is only allowed to send low power beacons for the next fifteen minutes (from fifteen to thirty minutes from launch). After this period it is allowed to activate its high power transmission and primary radios.

When the COMMpic/beacon module gets turned on, it will test the state pins connected to the OBC, to test whether it is allowed to start sending beacon messages or not. At the first boot just after launch these pins should be set in hold state such that the satellite stays silent for the first fifteen minutes after it has been launched. Since it cannot be detected for how long the charging of the batteries lasted it is necessary to keep the satellite silent for a fifteen minutes period to ensure that it has been silent for at least this period.

As the COMM subsystem is not designed yet, the length of its boot period is unknown. It is expected though that it is shorter than the time it takes for the OBC to startup and setup the state pins to carry the 'hold' signal. This will lead to a situation where the COMM starts to send beacons just after it has been powered on and before the fifteen minutes silence period is over. Obviously this should be avoided, and a solution to this problem is to let the beacon module wait for a short period of time every time it is turned on. The size of this period will lay in the range of a few seconds and will depend on the worst case execution time of the memory test described in chapter 4.

- The OBC should send a silence signal to the COMM and keep it for first fifteen minutes after launch.

Since reboots could occur during the silence period it should be recorded between the boots how long the satellite has been silent or how much time is left of the silence period.

- The boot software should record for how long the hold signal has been hold, and save the information in the FLASH memory periodically.

3.3.4.2 The rest of the boot procedure

After the C program is started it should be able to determine whether the satellite has been booted recently. If this is the case it should also be able to determine how many times, and start the FS software if the number of attempts exceeds a certain limit.

Before the boot program is allowed to boot the satellite in nominal mode i.e. start the OS, a complete memory test of all volatile memory should be carried out. Only if no faults are found the boot software should boot the satellite in nominal mode.

In addition to verify the integrity of the volatile memory the boot program should also verify the integrity of the binary image containing the OS and the application software. If faults are encountered the FS software should be started.

- The boot program should be able to determine if the OS has been booted recently and for how many times.
- The boot program should do a complete memory test of all volatile memory except the area used by the C stack of the boot and FS software. If faults are found the satellite should not be booted in nominal mode.
- The boot program should only boot into nominal mode if the integrity of the binary files can be confirmed.

3.4 The system information block

During the boot procedure various pieces of information about the system state of the latest boot and the system set up in general are needed. These pieces of information are used to determine for example whether to startup the OS or not. All information which should be stored between boots are written in the *system information block* (SIB).

The information should be stored between consecutive boot attempts, why it needs to be stored in a non volatile memory. On the DTUsat-2 this is either the internal or external FLASH memory.

3.4.1 The content of the SIB

The content of the SIB can be divided into two groups: information which is updated at every boot and rarely updated information. The first group contains information about the state of the system and the second group contains information about the software on the system.

3.4.1.1 The state of the system

The content of the system state part of the SIB should contain the following values:

Launch bit The launch bit indicate whether the satellite has been silent for the first fifteen minutes after launch as demanded by the CubeSat standard. This information is used to control whether the satellite needs to stay silent or not. It is also used to keep track of how long time the system has actually been silent if a reboot should occur during the fifteen minutes period. The value of the launch bit is set to fifteen before launch and should be decremented every time a period of one minute has elapsed. When the value of the launch bit equals zero, the fifteen minutes have elapsed. If the satellite reboots during the fifteen minutes, this counter ensures that the system can continue were it left and not has to restart on the fifteen minutes period. The name 'launch bit' is misleading because it indicates that the variable is a boolean which is not the case. The source of the name is the launch bit of the DTUsat-1 which were used to indicate whether the silence period had elasped or not. This variable only had two legal values unlike the launch bit of the DTUsat-2 which will have at least fifteen legal values.

Boot counter The boot counter stores the number of allowed boot attempts. It is used to determine whether the OS or the FS software should be booted. If the OS has been attempt booted to many times the FS should be booted instead and the earth station should be contacted.

Checksum The checksum contains the checksum of the SIB exclusive the checksum itself.

Dynamic change of the boot attempt limit To allow the operators at the ground station to change the boot attempt limit the boot counter should be a *decrementing* counter instead of an *incrementing*. If it is decrementing it is initialised to a value larger than zero, and when a boot has been attempted it is decremented.

When it reaches zero it indicates that no more attempts to boot the OS are allowed and that the FS mode software should be booted.

If the operators on Earth want to change the boot counter they can just upload a new version of the SIB containing a different value of the boot counter.

3.4.1.2 The state of the software

The second part of the system information block should contain information about the location of the OS and application software and its checksum.

OS beginning address The beginning address of the binary image containing the OS and the application software.

OS end address The end address of the binary image.

OS checksum The checksum of the binary image containing the OS and the applications.

OS pointer Pointer which points to the address where the execution of the OS should begin.

3.4.2 The default configuration of the SIB

If the system fails to read the most recent SIB for example due to memory faults it should have a default version to fall back on. The default version is only used if errors occur. Because of this the default version of the SIB should contain the values which leads execution into the FS software. The variable which controls this is the boot counter which therefore should be set to zero.

The default configuration of the SIB should also be easy identifiable. This is because the boot software needs to be able to identify that it is not the most recent SIB which is used. The boundaries of the OS stored in the default version of the SIB should never be used to anything since the boot counter is set to zero, indicating that the FS program should be started instead of the OS. Therefore these fields could be used to identify the default SIB from.

As a consequence of this, a valid version of the SIB containing the correct boundaries, checksum of the OS and pointer to the starting position of the OS should be stored in the first SIB position in the FLASH memory before launch. An important property of this SIB is that it should have the launch bit set to 15 to indicate that the system should be silent for the first fifteen minutes.

If the boot program for some reason is unable to read the SIB placed in the FLASH before launch it will not set the hold signal to the COMM beacon module. This results in the beacon module starting to send beacons immediately after the satellite has left the launch vehicle. This is as mentioned in section 3.3.4.1 an illegal action and should be avoided. The probability that this situation should occur is extremely unlikely. The solution which protects against the occurrence of this situation demands that the default version of the SIB has the value fifteen on its launch bit. This could complicate the maintenance and usage of the satellite:

If for some reason the EPS is only able to deliver power for a period shorter than fifteen minutes and the most recent SIB is damaged, the satellite will enter a state where it is unreachable. This situation is considered worse than the extremely unlikely situation that the satellite should start sending beacons before the fifteen minutes period has elapsed just after launch.

3.5 The fail safe mode

The fail safe mode software is used to do maintenance on the OBC and *on board data handling* (OBDH) (the OS and the application software) which cannot be carried out while the OS is running or if the OS fails to startup. The last task is the most important since it enables the operators on Earth to get in contact with the satellite if the OS has crashed and cannot restart on its own.

To resolve these tasks the FS software needs to be able to perform the following operations:

Download data The FS software needs to be able to download any data placed anywhere in the memory of the satellite. This capability should be used to download status information, and data which could be used to debugging and to trace errors.

Upload data The FS software needs to be able to upload any data to any location in the memory of the satellite (except for the area containing the boot and FS software). This capability should be used to replace the OS and OBDH. It could also be used to upload special programs which should be used for debugging purposes or carrying out special tasks on the satellite. Finally the capability could be used to place new system information after replacement of the OS.

Execute programs The FS software should be able to start execution of code for any memory address in the satellite. This facility should be used to start any uploaded programs.

Collect status information The FS sofware should be able to collect and return every kind of status information generated on the satellite. This facility should be used collect status information which could be used during diagnostics of faults in the satellite.

Of course a lot of scenarios of failure situations should be considered while analysing the satellite. First of all it should be determined which minimum pre-conditions are necessary in order to execute the FS software and get in contact with Earth:

1. The EPS needs to be in working condition such that power is supplied to the other subsystems of the satellite (at least the OBC and COMM).
2. The OBC needs to be in such a condition that the CPU chip and the ROM chip containing the boot and FS software, can work and communicate,

such that the boot program is able to start and run. This also demands the internal FLASH of the CPU chip to work since it contains the boot loader which startup the chip.

3. Finally the radio needs to be in such a shape that commands and data can be sent between the ground station (GS) and the satellite.

As it is seen neither the external RAM nor the external FLASH memory need to be in working condition. Other subsystems of the satellite could also be defective in some way without necessarily influencing the execution of the FS software. The satellite could be in a condition where the systems needed to start up the FS software are in working condition but where the lack of other subsystems makes it impossible to run the satellite in nominal mode. This is of course not a desired situation but as long as any contact to the satellite can be established the satellite is usable to some extent.

3.5.1 The DTUsat-1 FS software

Due to lack of time it has not been possible to develop a new version of the FS software. On the other hand the FS software of the DTUsat-1 was well functioning and software to communicate with it has been developed and implemented. Therefore it was decided that as much of this code base as possible should be reused.

To get the DTUsat-2 boot software and the DTUsat-1 FS software to work together the latter needed to be modified. The major differences between the two are about every aspect concerning the SIBs and its handling.

Another major area where the software needs to be updated are the FLASH drivers. The primary reason for this is that neither the external nor internal FLASH are the same as used on the DTUsat-1. Since these drivers are heavily hardware specific a complete reimplementation is necessary.

The SIB system demands some kind of storage to store runtime information about it. Therefore a new data structure to store these data should be developed. This structure does not necessarily need to be stored between boots because it can be regenerated during the boot process.

Chapter 4

The memory test

4.1 Memory test - analysis

Due to the fact that the satellite will be placed in a harsh environment in terms of electromagnetic and cosmic radiation there is a high probability that this radiation will affect the electric systems of the satellite.

The radiation could induce transient faults as bit flips changing the content of a memory cell or a register.

It could also damage the ICs permanently by damaging either the wires or the transistors. This results in a situation where for example memory cells would carry either a zero or one no matter what is written into it (known as stuck at zero/ones errors).

Also the impact of the forces on the satellite during launch could induce faults, for example by damaging the tracks on the circuit board.

All these kinds of errors have the potential of leading to failure of the system and thereby reset, but where transient bit flips will be removed when the memory cell is rewritten after a reboot, the permanent damage of wires (inside or between ICs) or transistors will lead to recurrence of failure unless the defective memory cell is identified and use of it is avoided by including some kind of fault tolerance.

4.1.1 Categorising the type of fault

Two kinds of faults may occur: transient and permanent. The first may be handled in a simple way by rewriting the memory cell, but the latter may be fatal to the system. Since the two kinds of faults exhibit these different characteristics the memory test should be able to identify which kind of fault it detects so that the fault is handled correctly and a transient fault is not perceived as permanent triggering a radical actions.

4.1.2 Requirements by the memory test program

When the system starts up, the boot software starts using the RAM as soon as C code is executed since this is stack based. A stack placed in a memory area containing flawed memory cells will therefore lead to failure already in the start up phase since a part of the boot program is written in C. Therefore a memory test should be carried out before the C stack is setup and the result of the test should influence the placement of the stack.

The primary requirement of the memory test is therefore that it should be able to identify defective memory cells or other errors in the memory system. Based on this information the system should be able to identify a continuous memory area without faults, large enough to contain the C stack.

During the execution of the C based boot software a complete memory test should be carried out. This test should verify that the complete memory is functioning correctly since this is a necessary condition in order to startup the OS.

The test implemented in C is based on the algorithm in [Bar99]. It should test the external memory since the stacks of the OS should be placed here. If the internal RAM contains such a big amount of flawed memory cells that no area large enough to host the C stack used by the boot software is found, and it therefore has to be placed in the external RAM, the memory test should not be carried out by the boot program. Instead it should start the FS program. After that a memory test controlled by the FS can be carried out.

In the rest of this section only the test executed before setting up the C stack is considered.

4.1.3 Faults in memory systems

As mentioned in section 4.1 the memory test should detect any kind of faults in the memory system. In this case the memory system refers to volatile memory of the system i.e. the RAM. This requirement is reduced a little since only software based fault detection is available.

An article written by Michael Barr [Bar00] treats the subject from a practical point of view, identifying the kinds of faults needed to be detected and gives guidelines on how to design and implement a test suite which fulfils the requirements to detect these fault types.

Michael Barr describes *electrical wiring problems* as the most common group of faults in memory systems. He also mentions problems in the chips, that is permanent damages of either wires or transistors, but categorise them as rare which might be correct in an protected environment as the surface of the Earth. But in space these kinds of errors may be common. Fortunately the test suite he suggests is told also to be capable of detecting errors in the data storage functionality of ICs, which is also a basic requirement of the memory test. Also shorts or opens of the address wires which lead to situations where two addresses refers to the same memory cell are detected.

Jack G. Ganssle also treats the subject of memory testing in two articles ([Gan95] and [Gan97])[1]. The most of the highlights in the first article are copied by Michael Barr in [Bar00], except for one central key point: Since the memory test should test the RAM, naturally it cannot utilise the RAM during its execution. Therefore the memory test cannot be implemented in C since C uses a stack which is normally placed in the RAM.

In the latter ([Gan97]), one of the key points is that the test should run as fast as possible and toggle as many address lines and data lines of the memory chip at the same time as possible [Gan95], since this will reveal weaknesses caused by the electrical characteristics of the chip. This is not necessary in our case for two reasons:

1. The clock frequency on the external memory is low compared to the speed normally used on memory busses in embedded systems. Therefore it is unlikely that the test would reveal any faults, simply because the chips are designed to run of a much higher pace.
2. If any faults were found, no solutions to the problem could be applied since the only solution is to replace the chips.

[1]The latter contains replications of some of the first's paragraphs

A test of the electrical properties of the memory system should of course be carried out before launch of the satellite.

The sources presented in this section seems to be the only ones treating the subject of software based memory test. This conviction is based on an extensive search for articles and homepages treating this subject on both the Google search engine and several article databases. The subject does not seem to represent an active research field and in general the algorithms presented in the articles are based on common sense and long time experience. It should also be stressed that the field of hardware based memory tests and embedded selftests seems to be an active research field, but this field has not been investigated further due to lack of time and because it was deemed less important for the project and the found information would probably be difficult to utilise.

4.1.4 Discussion of the approach used in DTUsat-1

The approach used in DTUsat-1 is to do a simple memory test during the boot process to identify a memory area to host the C stack. This memory test uses the algorithm presented in figure 4.1 based on the assembly code in the file `/failsafe/init.S` as found in appendix E.1.1. The control function can be seen in figure 4.1. This function tests the two memory areas RAM0 and RAM1.

```
Input: base addresses of the RAM areas, lengths of the RAM areas, test
       patterns: 1: 0x00000000 and 2: 0xFFFFFFFF, size of the needed space
       for the stack
Output: beginning address of the descending C stack
1 load inputs ;
2 ;
3 setup parameters for RAM0 for Write();
4 call Write() ;
5 ;
6 setup parameters for RAM1 for Write();
7 call Write() ;
```

Figure 4.1: The memory test algorithm used in DTUsat-1 as described in the assembler code in `init.S`.

The real functionality of the test is hidden in the `Write()` function showed in figure 4.2.

There are at least two issues which should be addressed in this function. The first is the choice of pattern used for the tests. These two patterns have the advantage that they generate a lot of changes on the data lines of the bus as suggested by Ganssle in [Gan95].

Unfortunately the way this test is performed makes it vulnerable to capacitive effects on the wires as described by Barr in [Bar00]. A solution could be to read each memory cell twice to try to unload it and if it shows the correct result the second time it is read, then it could be expected as working correctly. Another argument for this is that the running speed of the satellite system is so low that a stress test of the bus is not necessary. A general argument against doing any kind of stress test during boot is that it would not make sense to do when the satellite is launched since the bus cannot be changed in any way after launch. The only information needed after launch is whether the memory cells works correct or not. Of course a stress test will reveal this but it will be sensitive to the extreme condition which is induced as a part of the stress test. This increase the probability of the test producing false positives.

The second issue concerning the implemented memory test is its structure which leads the function into an infinite loop if a permanent error in the memory is detected. The fault in the code which leads to the erroneous infinite loop and thereby failure of the system is the lack of an increment in the memory address if an error is detected. The correct place to do it would be in the `Fail` function in the assembly code. This should be done just after the update of the 'highest valid address'.

The memory test is run before any kind of branching in the init sequence. If a fault in this function results in an infinite loop the WDT will not have any effect on it even if it times out. The reason is that every time the system reboots it will be caught in the loop again. The mechanism is illustrated in figure 4.3. A model verifying this behavior has been implemented in UPPAAL. A description of it can be found in section 8.3.

4.1.5 Conclusion

It can be concluded that the memory test should be capable of detecting faults present in and outside the ICs. It should be capable of identifying a correct functioning continuous memory area which can be used for the C stack used by the boot software. The test should not perform a stress test but simply ensure that the used part of the memory works correctly. Finally it should be implemented in such a way that it does not use the RAM during its execution since the correctness of this is not established yet.

```
Input: base ADDR of memory area, highest valid ADDR of memory area,
       test patterns: 1: 0x00000000 and 2: 0xFFFFFFFF, size of the needed
       space for the stack
repeat
    write pattern 2 to this address ;
    if content of memory address is different from pattern 1 then
        change highest valid address to this address ;
        Write() ;
        break ;
    end
    write pattern 1 to this address ;
    if content of memory address is different from pattern 1 then
        change highest valid address to this address ;
        Write() ;
        break ;
    end
    if (highest valid address - this address) = stack size then
        setup stack ;
        break ;
    end
    jump to next address ;
until this address < base address ;
```

Figure 4.2: Pseudo code of the 'Write' function

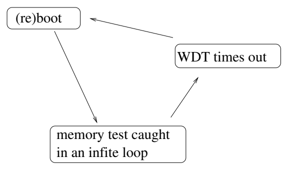


Figure 4.3: The mechanism showing why the WDT cannot recover the system from an infinite loop caused by a software fault or permanent fault in the memory test function.

4.2 Memory test - design

The memory test is divided into two tests: A test for identifying a memory area large enough to contain the C stack, and a complete memory test which is able to test the entire memory and detecting the kind of fault found. This division is similar to the design used in DTUsat-1.

4.2.1 Stack area test

The *stack area* test should be able to identify an area of sufficient size to contain the C stack. The test should be executed very early in the boot process and should be implemented in the assembly language. For this reason and for robustness it should be as simple as possible.

The algorithm is a modified version of the algorithm described in [Bar99, section: Device test on page pp. 71]. This particular algorithm is chosen because the test should detect defective storage locations which is a kind of device errors. In figure 4.4 and 4.5 a modified version of the algorithm is given. The modifications extend the algorithm by wrapping the original one in a house keeping system. This system keeps track of when a flawless memory area which is large enough to host the boot and FS software stack is found.

It also extends the algorithm such that it can distinguish between transient and permanent faults. This facility however infer with the algorithm's ability to detect faults where two memory locations point at the same memory cell. The reason for this is that the algorithm rewrite the memory location if a fault is found. This is done to test if the fault was transient (which will disappear when rewritten) or permanent.

Since transient faults caused by the radiation in space will occur much more frequently than shorts or opens after launch, it has been considered most important to be able to distinguish between transient and permanent faults.

Of course a test which *can* detect shorts or opens should be carried out before launch.

Since the algorithm is so long it is divided into two parts where the first part tests the memory using the one pattern and the second part tests the memory using the inverted pattern and eventually initialising the C stack.

In the algorithm presented in figure 4.4 and 4.5 the following input and output values are used:

base address is the lowest valid address of the memory area.

top address is the highest valid address of the memory area meaning the highest valid memory address pointing at a full 32-bit word. This address is normally equal to the length of the area minus one full 32-bit word.

top address of stack is the highest valid address of the stack area similar to the top address of the memory area.

4.2.1.1 Placing the stack

As it can be seen the pseudo code (figure 4.4 and 4.5) the algorithm traverses through the memory area. The memory is divided into parts each the size of the area needed to host the stack. Every time a permanent damaged memory location is localised the top address is aligned to the beginning of the word containing the flawed byte and decremented by one. After that the test is restarted at this position.

This algorithm ensures that the area used for the stack is placed in the highest located area fulfilling the demands of size and flawlessness, if such an area exists at all.

4.2.1.2 Fault type identification

The mechanism for identifying the type of fault located is quite simple: When meeting a fault the program will try to rewrite the test pattern to the memory location. If the fault is transient it is removed by rewriting the memory location. If the fault is permanent it cannot be removed by rewriting the memory address.

```
Input: base address, top address, stack size
Output: top address of stack area
start at top address + 1 ;
for (current address > base address) AND (top address - current address) < stack size do
    decrement current address;
    change pattern ;
    apply pattern to current address ;
end
reset current address to top address and pattern to initial state ;
for (current address ≥ base address) AND (top address - current address) < stack size do
    change pattern ;
    read byte value ;
    if read value ≠ pattern then
        write pattern to current address ;
        if address is the highest valid address then
            write inverted pattern to lowest address in potential stack area ;
        end
        else
            write inverted pattern to the address above ;
        end
        read the value of the of the current address ;
        if read value ≠ pattern then
            set top address = current address - 1;
            reset pattern to initial state ;
            restart test ;
        end
    end
    decrement byte counter ;
end
```

Figure 4.4: First part of the memory test algorithm used during the init phase. This is a short version meant for obtaining an overview over its functionality.

```
reset pattern to initial state ;
start at top address ;
for $(current\ address \geq base\ address)\ AND\ (top\ address - current\ address) < stack\ size$ do
    change pattern ;
    write pattern to current address ;
    decrement current address;
end
reset pattern to initial state;
start at top address ;
for $(current\ address \geq base\ address)\ AND\ (top\ address - current\ address) < stack\ size$ do
    change pattern ;
    read the value of the current address ;
    if $read\ value \neq pattern$ then
        write pattern to current address ;
        if address is the highest valid address then
            write inverted pattern to lowest address in potential stack area ;
        end
        else
            write inverted pattern to the address above ;
        end
        read the pattern from current address ;
        if $read\ value \neq pattern$ then
            set top address = current address - 1;
            restart test ;
        end
    end
    decrement byte counter ;
end
if $(top\ address - current\ address + 1) = stack\ size$ then
    Start C based boot program ;
else
    start register based FS software ;
end
```

Figure 4.5: Second part of the memory test algorithm used during the init phase. This is a short version meant for obtaining an overview over its functionality.

This mechanism is not completely reliable: It is theoretically possible that a transient fault induced by a bit flip could be perceived as a permanent fault. This situation would occur if another bit flip is induced between the rewriting of the memory location and the second reading instruction.

This situation is extremely unlikely to occur however, since only a few instruction lays in between the rewriting and reading meaning that the instruction would be separated by only a few milliseconds. On the other hand the algorithm would never misinterpret a permanent fault as a transient which is the most important property in this case.

4.2.2 The memory test done by the C based boot program

When the boot program is started it should carry out a more complete test of the whole RAM area on the system. The test functions used in this process is the functions described by Michael Barr in [Bar99, pages 66 - 73]. The tests described here include a data bus test, an address bus test and a device test, which test the same proporties of the memory chips as the test carried out prior to C stack initialisation does.

The functions should be extended to be able to log information about the found faults. It should also be ensured that they do not enter the area containing the stack of the boot program. Therefore information about the localisation of the stack should be passed to the program when it is called.

4.3 Memory test - implementation

This section describes the implementation issues of the memory test which is executed before the initialisation of the C stack. Therefore it is of course implemented in the assembly language. The implementation is based on the pseudo code shown in figure C.2 and C.4. As it can be seen when comparing this algorithm to the one presented in figure 4.4 and 4.5 the compound expressions used in the major **for** loops have been changed to a single expression and the second half of the expression has been moved inside the **for** loop to simplify the implementation. The input and output of the function presented in figure C.2 and C.4 also follows the definitions given in section 4.2.1.

The algorithm shown in figure C.2 and C.4 also reflects that the test should be carried out on byte level of the memory as suggested in [Bar00]. This also involves that the parameters given to the algorithm should be expressed in byte units.

Finally the top address of the stack is aligned such that it is changed to the correct 32-bit word address.

4.4 Memory test - software test

The assembly function doing the memory test works directly on the RAM. This makes it difficult to test since no output data can be collected during execution except from doing a memory dump. No input data are used either which complicates fault injection.

The optimal solution is to test the function on two portions of RAM: one which was known to be fault free, and a second which was known to be defective. Unfortunately it is not possible to do this since the flat sat[2] is not ready yet and since it would be impossible to change the RAM circuits on it.

4.4.1 Fault injection

Since it is impossible to test the program on defective hardware the tests demand the possibility to do fault injection in order to test the function in general and especially the parts of the function which implements some sort of fault tolerance.

A way to do fault injection in our case could be by changing the code directing all load and store operations through a function which potentially could inject faults. This demand either a simulator containing this functionality or modification of the code.

A modification of the program would probably introduce more faults which would lead to more debugging and unreliable test results. Therefore this solution has been abandoned. Instead a simulator is used. This solution is described in section 7.3.2.2.

[2]The prototype of the satellite

4.4.2 Test strategy

The following structural test should be carried out:

- Inspect the counters when they are reset to be ensured that they are given the correct value. This includes the test pattern.
- Ensure that the counters are incremented correctly. This includes the test pattern.
- Inspect the counters and constants which define the **for**-loops ensuring that they breaks at the correct values.

The following functional tests should be carried out:

1. Test that the function works correctly and terminates correctly when the RAM is fault free.
2. Test that the function works correctly when different kinds of faults are introduced in the RAM.

4.4.3 The general test setup

The GNU Debugger was used as test environment because it can execute command scripts. This makes it possible to automate the test. This system was also used during debugging and a general description of the use of GDB can be found in chapter 7.

A memory setup which differs from the one in the satellite is used in the tests. This is done to give the parameters different values such that they, are easier to recognise through the test. This should not change the tests ability to produce correct answers since it only involves the change of specific memory addresses, the size of the memory areas and the size of the stack. The values of the parameters describing the memory setup passed to the function are listed in table D.2.

4.4.4 Structural test

The inspection of the values of the registers is done on specific places in the code e.g. when the program passes a label in the source code. Therefore a lot of new labels which is only used during test have been introduced in the code. The names of these labels all start with “wp” for *watch point*.

The test cases can be seen in table D.3. An overview of the purpose of the different groups of tests are given in table D.1.

4.4.5 Results of the tests

During the test three faults were discovered. Two of them were cut and paste faults where changes had not been carried out to adapt the code. These faults occurred because the code developed to test the memory using the normal pattern was copied and adapted to test the memory using the inverted pattern.

The third fault found concerns the special case where a fault is found in the first byte in the memory which is subject to the test. In this case the inverted pattern would be written outside of the memory at the byte address just above the flawed byte. The solution is to write the inverted pattern in the bottom of the tested area.

Chapter 5

Software modules

This chapter contains design and implementation descriptions of various software modules and drivers which have been developed to solve different tasks during the boot process or as helper functions for the FS software.

5.1 The SIB - design

5.1.1 Location of the SIB

Two locations are candidates to host the SIB: The internal and the external FLASH memory. Several measures should be evaluated when the storage location of the SIB is decided: robustness to resist faults caused by cosmic radiation, the available storage and the access method.

5.1.1.1 Cosmic radiation

Measurements of the two candidates' qualities concerning resistance to cosmic radiation are partly unknown, since no experiments have been carried out on the external FLASH chip. Therefore this measure cannot be evaluated. If any measurements had been available this measure would be the primary one when deciding on the location of the SIB. Measurements have been carried out on the internal FLASH showing that it is quite robust to cosmic radiation[1]. Another issue concerning the cosmic radiation is the connection between the components. The external FLASH is of course connected through tracks on the circuit board. This in itself causes a weakness since these rails could get damaged from.

5.1.1.2 The available storage capacity

In itself the size is not really an issue in this case since the SIB is expected to take up only a small number of bytes. Unfortunately the procedure for writing to the FLASH complicates the situation. It demands that a whole flash block must be erased periodically due to the design of FLASH memories. This issue is explained in section 5.4. The internal FLASH consist of blocks of eight kilo bytes and the external FLASH consist of blocks of 64 kilo bytes. Therefore it will be most efficient to store the SIB in the internal FLASH. In the internal FLASH, it will use eight kilo bytes of the available FLASH memory compared to the 64 kilo bytes used in the external FLASH. These considerations of course presume that the FLASH blocks could not be used for anything else at the same time it is used for the SIB. This is the most realistic assumption since the block would be erased at every boot attempt if only room for one SIB is allocated in the block.

5.1.1.3 The access method

The measure where the two candidates differ most is the access method. The internal FLASH is placed in the CPU chip using some functions supplied by the manufacturer of the chip (Philips). This means that the delete and write functions which are essential in the use of the FLASH relies on some closed code which cannot be inspected for faults or changed if that should be relevant.

The external FLASH was not delivered containing any drivers why they need to be implemented as a part of the boot software. This gives full access to control and inspect the code. It could also be argued that implementing the FLASH driver would lead to a higher complexity of the code and thereby give a higher risk that fault occurred in the code. Here it should be mentioned that

[1]It should last at least for 18 month with normal amount of cosmic radiation according to the test carried out by the OBC team of the DTUsat-2

the FLASH drivers for the external FLASH needs to be implemented in the FS in any case since it should be able to write to it.

It can be concluded that placing the SIB in the external FLASH gives the best knowledge and thereby assurance of a correctly functioning FLASH driver.

5.1.1.4 Conclusion

It must in general be concluded that in spite of the weakness coming from the drivers of the internal FLASH, the external FLASH is the most insecure device. Two arguments support this: The external FLASH has not been radiation tested and it is connected through weaker connections than the internal FLASH, which is the primary candidate to store the SIB.

An even better but substantially more complex solution would be to let the system search for the best place to store the SIB based on different measures having both the internal and external FLASH as candidates.

5.1.1.5 Location in the FLASH

An important issue concerning the location of the SIB is where in the FLASH it should be placed. A whole eight kilo byte block is available in the internal FLASH. Two principles are available to decide its location: static location and dynamic location.

Static location If static location is used as principle for the location of the SIB the memory address of its location is hard coded into the program.

This solution has the big advantage that it makes it simple to find it since the beginning address is known.

It also has the big disadvantage that the FLASH need to be flashed every time the new SIB is saved. Since it is only possible to flash the FLASH a finite number of times, it will last for a shorter period of time.

Dynamic location If dynamic location is used as principle for the location of the SIB there is no static address for its location. It can be placed at a number of different locations throughout the FLASH block. These locations lay after each other. A magic number then identify the beginning of the SIB.

To find the most recent version of the SIB the memory area is searched backwards from the highest address. When the first instance of the magic number is detected the beginning of the most recent SIB is found.

When the Block is filled up it is flashed and filled up from the lowest address again.

Conclusion The disadvantage of flashing the FLASH block at every boot giving it a shorter function period is a serious issue since the SIB provide vital information during the boot process. Therefore it should be ensured that the FLASH storing the SIBs stay functional as long as possible. Therefore dynamic location is chosen as principle for the location of the SIB.

5.1.2 Communication between the OS and the boot software

In the version of the software used on DTUsat-1 the communication between the OS and the boot software was handled by having a variable in the SIB which was used to give commands from the OS to the boot software. The OS changed the value of the variable and the different values were interpreted as different commands.

In this version of the software this system is abandoned since it is unnecessary. The messaging can be handled by manipulating the other variables in the SIB, and still pass the same commands to the boot software or the OS can carry out the jobs itself.

5.1.2.1 Reseting boot counter

On the DTUsat-1 the boot counter was reset by the boot program after receiving a signal from the OS. On the DTUsat-2 the OS will have functions at its disposal to do it.

5.1.2.2 Forcing the FS to boot

To signal the boot software to boot the FS instead of the OS it should set the boot counter to zero and reboot the system.

5.1.3 Analysis of usage of the SIB

The SIB is used and manipulated by both the boot and FS software and the OS. Therefore a function library should be made available containing the function related to the SIB manipulation. This section will contain an analysis of the demands of the modules which should be able to read and manipulate the SIBs.

5.1.3.1 Common needs of the modules

This section presents the common functional needs of the boot software and the application programs which should handle the SIBs.

First of all the modules need to be able to find the most recent valid version of the SIB or at least a valid default version. When a valid version is found the modules are able to read and manipulate the values of the SIB.

Because the SIBs are stored in a FLASH memory, the number of writing operations should be kept at a minimum. This is because each bit in the FLASH only can be changed from 'one' to 'zero' once before it is locked until after the next 'flash' operation resets the block. Concerning the number of write operations, the most general solution is to maintain a working copy of the SIB stored in the RAM and then write this back to the FLASH when the session is over, if any changes have been applied to the content of it. Unfortunately this solution has a small fault tolerance since it cannot be ensured that the possibly changed SIB is stored in the FLASH if the WDT resets the system.

When a new version of the SIB is written to the FLASH it should also be ensured that as few values are written from the RAM as possible and they should instead be copied from the last written instance of the SIB if it is valid. If it has been rendered invalid, a new SIB should be uploaded from the Earth, since only the newest written SIB could be assured to carry the correct values. The reason for this is that the FLASH is considered more robust against bit flips induced by cosmic radiation than the RAM.

The only solution to this problem is to write the SIB back to the FLASH instantly whenever a change has been applied to any of the values in it. This solution actually comply very well since it normally will be the case that only one value needs to be changed i.e. the boot counter. The other values are only changed in rare cases for example when the OS is replaced.

5.1.4 Checksum

The validity of the data stored in the SIB should be ensured by calculating a checksum from them, and compare it to the one stored in the SIB.

The checksum used in this project is the CRC32 as described in [Bar00, page 75]. The implementation is also taken from [Bar00, pages 77-79].

As suggested by Barr the algorithm is changed such that the table containing the remainders of the byte divisions are stored in the ROM as a part of the CRC calculation program.

There is two reasons for this approach. The first is that it speeds up the calculation of the checksum. The second is that the ROM is considered more resistant against faults induced by the radiation in space than the RAM. Therefore fewer faults in the data in the remainder table will occur when it is stored in the ROM.

5.1.5 Memory efficiency

All values in the SIB are implemented as 32-bit integers. This is obviously not the most efficient implementation regarding memory usage.

An example is the boot counter which will never need to carry a value above 100 and probably not above 25. Therefore only 7 bits are needed to represent it.

The launch bit waste even more space than the boot counter since its greatest value should 15 and therefore it only needs 4 bits but take up 32 bits.

The argument for using 32-bit integers in all fields in the structure is that the data words of the ARM architecture used here is 32-bit long. Therefore the structure will fill in a whole number of data words, avoiding padding of the structure by the compiler. The overhead generated by using 32-bit values is at most: 28 bits from the launch bit and 25 bits from the boot counter summing up to 53 bits per structure.

If an eight kilo bytes FLASH block is used to store the SIBs in, there is room for 256 instances of the SIB in the block:

$$\frac{8 \cdot 1024 bytes}{8 \cdot 4 bytes} = 256$$

If the most memory efficient version of the SIB is used there is room for 322 instants of the SIB in the block:

$$\frac{8 \cdot 1024 bytes}{(8 \cdot 4) bytes - (53/8) bytes} = 322$$

The division used here is integer division because the byte is the smallest memory unit available in the system.

These calculations demonstrate that 12 % of the room used by an instance of the SIB could be saved, because only one 32-bit data word can be saved. This is because the data words need to be aligned on 32-bit boundaries:

$$\frac{32 - 28}{32} \cdot 100 = 12\%$$

Giving room for 25 % more instances of the SIB:

$$\frac{322 - 256}{256} \cdot 100 = 25\%$$

This does not give any practical impact on the reliability of the system: The number of instances of SIBs which the FLASH block contains, only influence the length of the life of the FLASH block. This is because it only changes the number of times the block need to be flashed during the service of the satellite.

If the FLASH is guarantied to survive to be flashed 1000 times during service this will result in 256.000 possible reboots during the service of the satellite, even if the large version of the SIB structure is used, and only the boot counter is changed at each boot. This should be more than enough. The number of guarantied flashes during the life of the FLASH is at least 10000 according to the documentation [Phi03, page 262].

5.1.6 The general design approach

The general approach has been to encapsulate the SIB, only allowing access indirectly through function calls. This approach has been chosen to raise the robustness of the code by preventing corruption of the SIB by illegal values in the individual fields.

This approach has also influenced the design of the access from the functions to the valid version of the SIB by simply making a pointer which always points at the valid instance of the SIB.

5.2 The SIB - implementation

The implementation language of the function used to read and manipulate the SIB is C. This issues the question of how to represent the SIB. The first choice for forming compound data types in C is a structure declaration.

The structure for the SIB is declared as follows (see appendix F.6 on page 155 from line 30.)

```
30 struct SIB { /**  \label{lst:app:sysInfoh:SIB} */
31     int  magicNum;      /*Magic number: 0xFEEDBEEF */
32     int launchBit;    /*initial 15. Decreased by 1 each minute.
           0 the 15 minutes is over */
33     int bootCounter; /*Count the number of boot attempts =
          MAX_BOOTS - boot attempts */
34     unsigned long  eCosBeg;       /*32bit beginning address of
           area containing eCos */
35     unsigned long  eCosEnd;       /*32bit end address of area
           containing eCos */
36     unsigned long  eCosCheck;     /*32bit CRC checksum of the
           area containing eCos */
37     unsigned long  eCosP;         /*pointer to execution
           beginning of eCos */
38     unsigned long checksum;       /*32bit CRC checksum of the SIB
            minus the checksum itself*/
39 };
```

5.2.1 Memory structure and location of the SIBs

The two following models for describing the location of the SIBs in the memory have been considered:

1. Describing the location directly in the memory by calculating the values of the pointers to them manually.

2. Describing the location as an index in an array of SIBs. This solution leaves the calculations of the pointers to the compiler.

The second solution was chosen because it seems more robust and gives an easier and understandable code. The robustness comes from the fact that the style of the code saves the base address of the area through the code and index from this relatively. This invites the programmer not to change the value of the pointer and thereby protect against errors comming from 'out of area'-faults.

The concept of the model is to cast the pointer to the beginning of the area containing the SIBs as a pointer to a SIB:

```
/* Array declaration to cast the memory area as an array of SIBs*/
struct SIB * sibs = (struct SIB *) BEG_ADDRESS;
```

Because a pointer can be handled as an array [KR88, page 99], the area is perceived as such using an index. The pointer is called `sibs` and the index variable `index` (see appendix F.9 from line 28):

```
26 /*Function to find the location of the most recent SIB
27  * Sets the 'theSIB' to point at it, and the value of
        idxOFtheSib.  */
28 int findSIB(struct SPS *sps) { /**\label{lst:app:sysInfoc:
       findSib}
29 \index{findSIB!\textit{source code}}
30 \addcontentsline{toc}{subsection}{findSIB()}*/
31     int index = 0;
32     for(index = sps->arrayLength; index > 0; ){
33         if(sps->sibs[--index].magicNum == MAGIC_NUM){
34         sps->idxOfTheSib = index;
35         sps->theSib = &(sps->sibs[index]);
36         index = 0;
37         }
38     }
39     if(sps->idxOfTheSib >= 0)
40         return 0;
41     else
42         return 1;
43 }
```

As it can be seen from the source code above the pointer `theSib` *always* points to the found valid SIB. If no valid SIB is found `theSib` points to the default configuration of the SIB which boots the FS.

The index of the valid SIB is also stored in the variable `idxOfTheSib`. This value should only be used when the index of the next instance of a SIB is to be calculated when it needs to be written to the FLASH.

5.3 The SIB Parameter Structure

A group of variables containing information about the location of the most recent SIB and other central system information is used in a lot of different functions. Therefore a lot of functions have the same arguments. This situation should be simplified. Normally this would be done by making these variables global. Global variables are not allowed on the system because they are allocated statically by the linker. If the area used for the statical variables is damaged it could block the system, preventing it from starting the OS or failsafe mode. The group of central variables are instead gathered in the *SIB Parameter Structure* (SPS). A reference to a central instance of the SPS is then passed to the functions which use these variables. In this way the number of arguments of the functions is reduced, making the code more readable and simpler in general. Below is the declaration of the structure given:

```
struct SPS {
   struct SIB * sibs; /*Pointer to array of SIBs in FLASH. */
   struct SIB /*@null@*/ * theSib; /*Pointer to the latest
        valid SIB or the default SIB. */
   int idxOfTheSib; /*Index of theSib in sibs or RAM*/
   int arrayLength; /*Number of SIBs in array 'sibs'. */
   struct SIB * defaultSib; /*Pointer to default SIB. */
   struct SIB tempSib; /*Structure containing temporary values
         of SIB */
};

/*Function to find the location of the most recent SIB.
```

5.4 FLASH driver - analysis

The nonvolatile memory system of the satellite consist of two different FLASH memories: the internal FLASH of the CPU chip (256 KB) and some external FLASH chips (2 MB in total). Both are memory mapped systems.

The purpose of the internal FLASH is to store the OS and the System Information Blocks.

The purpose of the external FLASH is to store data collected by the applications running on the satellite in nominal mode.

One central property of FLASH memory is that it can be read as normal RAM by accessing the individual memory addresses. When one wants to write data to it a special sequence of actions need to be carried out called the *erase and write cycle*. The erase and write cycle can only be carried out a certain number of times before the FLASH chip stops to work.

Another important property is that the FLASH memory is divided into blocks. When an area in the block is written the first time since an erase operation has been carried out on the block, any bit pattern can be written to it. When written second or third time etc. only memory cells containing a '1' can be changed to a '0' but not vice versa. It needs to be 'flashed' i.e. erased (actually setting all memory cells to contain '1') before a new pattern can be written.

Throughout the report the terms 'block' and 'sector' are used interchangeably. In the documentation [Phi03] the term 'sector' is used exclusively.

Due to lack of time in the project only a driver for the internal FLASH will be developed and implemented. Therefore the rest of this sections describing the FLASH driver will only refer to the internal FLASH.

5.4.1 Minimum size of written area

According to the User Manual of the LPC2294 [Phi03, Table 216, p. 282] at least 512 bytes need to be written at a time. This data unit is referred to as *line*. This does not give rise to any concerns when writing programs to the FLASH since 512 is a small number compared to the number of bytes occupied by the whole program.

However when writing SIBs to the FLASH, a lot of memory will be wasted if it is necessary to start at 512-boundaries every time a new entry needs to be saved.

A solution to this problem could be to look at the pointer pointing to the most recent SIB and then calculate the beginning and end of the next SIB which should written. From this information it is possible to fill the area behind the latest SIB with zeros and the area in front of it up to the next 512-boundary with ones, as if they were flashed.

A problem concerning this solution is that the area containing the ones in front of the SIB could be altered by cosmic radiation to carry zeroes instead of ones. This renders the area containing the illegal zeros useless until it flashed the next time. The solution to this problem is to check that the area contain nothing but ones before it is used to store anything in.

A better solution to the line-problem would be to rewrite the FLASH writing routines using the information found in [Jay06]. This source describes how to reverse engineer the In Application Programming (IAP) routines to rewrite them to for example only write 16 bytes at a time.

Due to lack of time it is not possible to use this solution. The FLASH writing system should be designed in a layered fashion such that the lower layer later could be replaced by a version which utilise the possibilities outlined in [Jay06].

In the rest of the project it is therefore assumed that the it is possible to use the proposed solution i.e. writing 512 bytes data unit padding it with zeros and ones.

5.4.2 Accessing the FLASH memory

During read operations the FLASH memory is accessed as any other memory device through a mapping of its memory area into the global memory address space.

When doing any other operations like write operations, it is necessary to operate the FLASH driver circuit in the FLASH by writing and reading some command and status registers. At the internal FLASH of the LPC2294 chip, this task is normally done indirectly by calling some low level functions which is a part of the boot loader of the chip. Therefore the FLASH driver routines should utilise these functions and do the housekeeping which is necessary to simplify the usage of the low level functions.

5.4.2.1 Available low level functions

The LPC2294 chip contains a boot loader which also makes some low level functions, to maintain the internal FLASH memory, available. The following functions are available:

Prepare sector(s) for write operation Unlock the sector before it is erased or written to.

Copy RAM to flash The write function which copy an area of the RAM to the FLASH described by beginning address and size.

Erase sectors Erase one or more sectors. After erase the sector contain nothing but ones.

Blank check sector(s) Test if the sector is empty (i.e. all ones) before a write operation.

Read Boot code version Reads the version of the boot loader of the chip.

Compare Compare the content of two memory areas.

A complete reference for these functions can be found in [Phi03, p. 279 - 284]. The LPC2292 has been reported to contain faults which leads to errors where the above described low level functions never return. This result in a hanging system leading to reboot by the WDT. In Errata sheet of the 5th of August 2005 for [Phi03] concerning the LPC2292 chip, it is reported that the fault is removed such that newer chips should not produce this error. The fault can also be removed by updating the firmware to version 1.63 or later.

5.4.3 The block structure of the internal FLASH

As mentioned above the FLASH memory is divided into smaller blocks.

In the LPC2294 chip there is 256 KB of internal FLASH memory which is divided into 18 blocks. Unfortunately not all blocks have the same size: Block 0 - 7 and 10 - 17 contain 8 KB of memory each. Block 8 and 9 contain 64 KB of memory each.

This pattern complicates the design of the functions handling the memory, since simple multiplications of the size of block cannot be used everywhere.

5.4.4 Necessary functionality

The FLASH driver is used in two major use cases: to write the SIBs to their dedicated area and to replace the OS if this should be necessary.

Since the SIBs are stored in a dedicated data structure special functions should be designed to handle write operations of these.

The prepare and erase operations should be hidden to the user in the general usage of the functions. Also the blank test made before each write should be hidden by the functions.

At the same time it should be possible to prepare and erase FLASH blocks by direct commands through the FS software.

5.5 FLASH driver - design

The FLASH driver should be divided into two separate parts: one which handles the writing of the area containing the OS and one which handle the writing of the SIBs. The reason for this division is that different procedures are used because the FLASH blocks will be erased before the writing of the OS which is not the case when writing the SIBs.

5.5.1 The SIB writing system

The SIB writing system takes care of writing the SIB to a location in the FLASH block allocated to store these.

5.5.1.1 Preparing the 512 bytes array

As described in section 5.4.1 before the SIB can be written to the FLASH memory it should be placed in an array which is 512 bytes long and be padded with zeroes in front of it and ones behind. This is illustrated in figure 5.1

The function which do the padding job is described pseudo code in function `prepDataArraySIB` shown in figure 5.2.

The area which is used to store the SIB should be tested for any content before it is used and after the write operation it should be verified that the correct data were stored. A flowchart showing the procedure for writing a SIB can be found in figure 5.3.

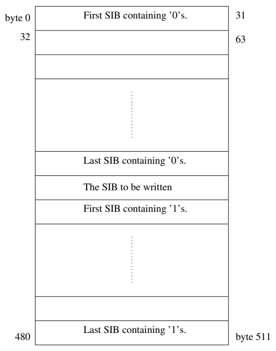

byte 0
First SIB containing '0's.
31
32
63
Last SIB containing '0's.
The SIB to be written
First SIB containing '1's.
480
Last SIB containing '1's.
byte 511

Figure 5.1: Figure ill. the contents of the 512 bytes data assembled before being written to the FLASH.

5.5.1.2 Test of Flash before writing operation

The FLASH memory is blank tested using a simple comparison, since it is known that a blanked FLASH area contains nothing but ones. Therefore the bytes in that area can simply be compared to the value 0xFF.

```
Input: data[], SIB[], beginning of SIB
for $0 \le index < 512$ do
    if $index <$ beginning of SIB then
        data[index] = 0x00 ;
    else if $index ==$ beginning of SIB then
        foreach byte in SIB do
            data[index] = SIB[SIB byte index] ;
            increment index ;
        end
        data[index] = 0xff ;
    else
        data[index] = 0xff ;
    end
end
```

Figure 5.2: Pseudo code of **prepDataArraySIB**().

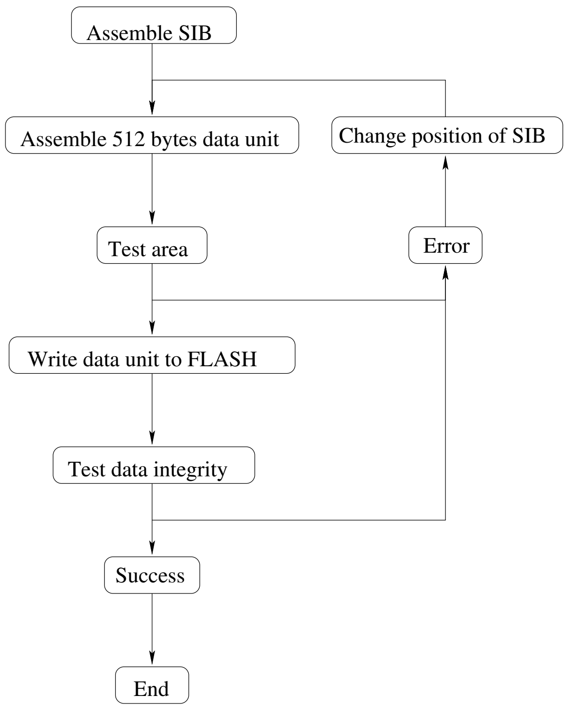


Figure 5.3: Flowchart showing the procedure for writing a SIB to the FLASH memory.

5.6 FLASH driver - Implementation

The functions which are implemented to do the FLASH operations are divided into two groups:

1. A group of high level function where several tasks are carried out by the same function.
2. A group of functions which only carry out a single function. Each of these function calls one of the functions made available by the boot loader of the chip. All the names of these functions begin with the three capitalized letters "IAP".

The first group of the functions are thought as the primary group used by any programmer and the second group should only be used when new functions are added to the first group.

5.6.1 Error information

All functions return a return value. The IAP data compare function returns both a return value and the offset to the first fault if any is discovered. Therefore all functions using the compare function have an extra return value which is called `errorInfo`. This value should be a pointer to a 32 bit integer. The offset of the first fault found is written to the variable which the `errorInfo` points at.

5.7 Real Time Clock - analysis

The LPC2294 chip is equipped with a Real Time Clock (RTC). This is a clock circuit which not only is capable of introducing an interrupt at a specific interval but also keeps track of time. Therefore it is possible to read the time from registers and setup an alarm which gives rise to an interrupt.

When the chip is powered off the values of the registers are not maintained, why it is not possible to maintain a correct time over a cold reset. If the chip is only reset by the reset pin (warm reset) the value of the time counters are kept.

As mentioned above the clock is capable of initiating interrupts. The usage of interrupts should be avoided in order to keep the predictability of the behavior of the system as high as possible. Therefore a method to keep track of the pace of time without using interrupts should be developed.

The primary task of the RTC during execution of the boot program is to keep track of time during the execution just after launch where the satellite needs to be silent for a fifteen minutes period. Since the satellite will start with a cold boot when it is released from the launch vehicle (LV), no real time values are kept. Therefore only a basic functionality which detects changes in the values of the counter registers of the RTC is needed.

For this reason the boot program is also allowed to alter the values of the counter registers if this is necessary. This is because no real time has been set when the fifteen minutes period should be measured.

When the fifteen minutes period has elapsed the RTC is no longer used for anything by the boot or FS program.

5.8 RTC - design

In order to fulfil the demands outlined in section 5.7, a simple initialisation function which ensures the disabling of the interrupt should be designed.

The initialisation procedure should reset the seconds counter such that it is ensured that a full minute or second has elapsed every time the minutes counter or seconds counter change. These considerations lead to the procedure outlined here:

1. Disable interrupts by writing 0 to the Counter Increment Interrupt Register.
2. Ensure that clock is disabled by writing 0 to Clock Control Register.
3. Reset seconds counter by writing 0 to its register.
4. Record minutes counter's value.
5. Enable clock by writing 1 to Clock Control Register.

5.8.1 Measuring time

The method to detect time elapsing is to detect changes in the counter registers. Therefore the value of the register should be recorded and after that the CPU should poll the register and compare its value to the recorded value. When they are not equal any longer the time period has elapsed. This leads to the following routine for detecting the elapse of one minute:

```
Reset seconds counter to 0 ;
record minutes value in variable 'minValue' ;
start clock ;
record minutes value in variable 'temp';
while $temp == minValue$ do
    update 'temp' ;
end
stop clock ;
```

5.9 RTC - implementation

5.9.1 Intialisation of the hardware

The RTC contains a circuit called the *prescaler*. The purpose of this circuit is to adapt the RTC to the clock frequency of the system. It should emit 32.768 impulses every second, which the counter circuit of the RTC use as input.

The RTC needs two values to be set during initialisation: The first is the number of clock ticks per impulse (it is referenced as *PREINT* in the documentation). Unfortunately this value could consist of a integer part and a fraction part. In this case, the fraction part is cut away. Instead a fraction of the impulse periods prolonged by a single clock tick. The number of impulse periods which should be extended is the second value (it is referenced as *PREFRAC* in the documentation). How to calculate these values are explained in [Phi03, page 253 - 254].

The values used on the Olimex test board which has a 14.7456 Mhz crystal are:

$$\begin{array}{lcccc} PREINT & = & (14.745.600/32.768) - 1 & = & 449 \\ PREFRAC & = & 14.745.600 - ((449+1) \cdot 32.768) & = & 0 \end{array}$$

Chapter 6

Implementation details of the boot procedure

In this chapter an overview of the implemented boot procedure will be given. The presentation is based on the annotated source code of the boot program.

The boot procedure can be divided into two phases: The operations carried out before the C stack initialisation and the operations carried out afterwards. The code carrying out the first phase is pure assembly code and the second phase is implemented in normal C code.

6.1 The first phase

The first phase is a sequence of operations which are performed by the code of the reset routine i.e. the routine which is called when a reset exception is caught. This routine is also carried after a cold boot.

1. Ensure that the interrupts are disabled. This operation is carried out as a part of the HW-reset. As an extra precaution the operation is repeated by the boot program.

2. Initialise the WDT.
3. Setup EMC.
4. Setup PLL.
5. Run memory test to find an area for the stack.
6. Setup the stack.

6.1.1 Implementation

The first phase is implemented in the `init.S` (listed in appendix F.1), `memTest.S` (listed in appendix F.2) and `cStack.S` (listed in appendix F.3) files.

6.1.1.1 Global variables

Normally it is necessary to copy the initialised variables placed in the .data section and the uninitialised variables placed in the .bss section to the RAM to make write operations to the variables possible. In the boot software of the DTUsat-2 no static variables are used. Instead all variables are allocated on the stack. This concept is used to avoid this copying of variables because they are placed on static locations by the linker. Therefore a flawed memory address in the area containing the static variables placed by the linker could obstruct the execution of the boot program.

6.1.1.2 Constants in the assembly code

Through out the assembly code constant's values are needed for example as specific addresses of registers. The easiest way to use a constant is using the *mov* instruction to place the value in a register.

Unfortunately this solution will not always comply. The reason is that the encoding method used to place the constant in the binary version of the instruction only support a limited range of values. The encoding utilise a rotation of an 8-bit value in a 32-bit data word, which cause the following restrictions on the value:

- It is not possible to encode values which contain 'ones' in bit 8 to bit 15.

- It is not possible to encode values where the distance (measured in bit positions) between the two outer bits is larger than 8.

A more complete description of the issue can be found in [Fur00, page 119].

The solution to the problem is to define the problematic values as constants stored in the memory and load their values into the registers using an *ldr* instruction. This solution has two disadvantages:

- Space of the memory is used to store the values.
- If slow memory is used, this solution will run slower than the *mov* solution which can run in a single clock cycle.

None of these disadvantages lead to significant problems on the DTUsat-2, since speed is not a concern in general, and the EMC is able to handle up to 16 MB of memory in each bank.

6.1.1.3 The exception vectors

The first part of the code contains the exception vectors. Since interrupts are disabled and therefore no interrupt routines are necessary in the boot program, they all contain a jump instruction to the same procedure. The exception vectors also jump to the same label as the interrupt vectiors. The symbols they are jumping to has been given different names emphasizing that these could be individual functions.

```
    ldr     PC, SWI_Addr
    ldr     PC, PAbt_Addr
    ldr     PC, DAbt_Addr
    nop                             /* Reserved Vector (holds Philips ISP checksum) */
    ldr     PC, [PC,#-0xFF0]        /* see page 71 of "Insiders Guide to the Philips ARM7-Based Microcontrollers" by Trevor Martin  */
    ldr     PC, FIQ_Addr


Reset_Addr:     .word   Reset_Handler           /* defined below */
Undef_Addr:     .word   UNDEF_Routine           /* defined below */
```

```
SWI_Addr:        .word    SWI_Routine             /* defined
    below  */
```

If for some reason an exception arises in the system during the boot program the exception vectors will jump to the exception handling routine which changes the mode to the system mode and start at the beginning of the boot program again.

6.1.1.4 Disabling of the interrupts

When the boot program is executed the first task is to disable the interrupts.

```


    ldr    r0, =_stack_end
    msr    CPSR_c, #MODE_UND|I_BIT|F_BIT      /* Undefined
        Instruction Mode  */
    mov    sp, r0
    sub    r0, r0, #UND_STACK_SIZE
    msr    CPSR_c, #MODE_ABT|I_BIT|F_BIT      /* Abort Mode */
    mov    sp, r0
    sub    r0, r0, #ABT_STACK_SIZE
    msr    CPSR_c, #MODE_FIQ|I_BIT|F_BIT      /* FIQ Mode */
    mov    sp, r0
    sub    r0, r0, #FIQ_STACK_SIZE
    msr    CPSR_c, #MODE_IRQ|I_BIT|F_BIT      /* IRQ Mode */
    mov    sp, r0
    sub    r0, r0, #IRQ_STACK_SIZE
    msr    CPSR_c, #MODE_SVC|I_BIT|F_BIT      /* Supervisor Mode
        */
    mov    sp, r0
@ User mode is not entered because we cannot return to a
    privileged
@ mode from user mode.
@       sub    r0, r0, #SVC_STACK_SIZE
@       msr    CPSR_c, #MODE_SYS|I_BIT|F_BIT      /* User Mode */
```

To disable the interrupts in all possible modes (minus user mode)[1] of the CPU, the mode is changed and the CSPR of that mode is altered to disable the interrupts.

At the same time the stack pointer of the mode is initialised to a defined value

[1]The user mode is not entered because it is impossible to get back into any privileged mode from the user mode unless a software interrupt is issued.

giving 4 bytes to each stack. The stacks are placed in the top of the user space as shown in figure 2.2. The area used for these stacks is untested for faults because they are not considered used for anything.

6.1.1.5 Enabling the WDT

The second task of the boot program is to enable the WDT. This is done according to the procedure described in [Phi03, p.256 - 258].

```
    mov r0, #WDMOD @Load register with address of WDMOD.
    mov r1, #0x03 @Prepare enabling pattern.
    str r1, [r0] @Enable WD.
    mov r0, #WDFEED @Load register with address of WDFEED.
    mov r1, #0xAA @Prepare first feed pattern.
    str r1, [r0] @Start feed sequence of WD.
    mov r1, #0x55 @Prepare second feed pattern.
    str r1, [r0] @End feed sequence of WD.

pll: .long _pll
vpbdiv: .long _vpbdiv
bc: .long _bc
```

The most important thing to notice is that the timer constant is set as large as possible (0xFFFFFFFF), and that the WDT needs to be fed to start. The last thing is done by sending 0xAA and 0x55 to its feed register.

6.1.1.6 The configuration of the PLL

The PLL is a circuit used to control the clock speed of the system. It is described in section 2.1.1.2. The presented configuration assumes that the crystal on the system runs at 14.7456 Mhz as the one on the Olimex development board. This input frequency is multiplied by four.

The PLL has a feed procedure as the WDT. Therefore the same feed sequence is send to the PLL after an alteration of one of its control registers.

It should be noticed that this block of code contains a loop which determines when the PLL has locked to its new frequency.

```

    /*configuration of the PLL */
    ldr r4, pll @set base address for constants
    ldr r0, [r4, #pll_cfg] @Load the address of PLLCFG into the register.
    mov r1, #0x23 @Prepare value of configuration register 0x23 = 0b100011.
    mov r2, #0xAA @Prepare feed 1 value.
    mov r3, #0x55 @Prepare feed 2 value.
    str r1, [r0] @Store value in configuration register.
    ldr r0, [r4, #pll_feed] @load register the address of the feed register into the register.
    str r2, [r0] @write first part of feed sequence.
    str r3, [r0] @Write second part of feed sequence.
    /* enabling of the PLL */
    ldr r0, [r4, #pll_con] @Load register with the address of PLLCON.
    mov r1, #0x1 @load register with enabling value.
    str r1, [r0] @store enabling value register.
    ldr r0, [r4, #pll_feed] @load register the address of the feed register into the register.
    str r2, [r0] @write first part of feed sequence.
    str r3, [r0] @Write second part of feed sequence.
    ldr r0, [r4, #pll_stat] @Load the address of PLLSTAT into the register.
plllock:
    ldr r1, [r0] @get value from PLLSTAT.
    ands r1, r1, #(1<<10) @And value of register with a one on the 10th place.
```

When it is confirmed that the PLL has locked to the requested frequency the VPB divider is configured to output a fourth of its input frequency i.e. the frequency which is supplied to the PLL.

```

    /* configuration of the VPB divider */
    ldr r0, vpbdiv @Load the address of the VPBDIV into the register.
    mov r1, #0x00 @The pclk is set to one fourth of the cclk.
```

6.1.1.7 The configuration of the EMC

The EMC is the interface between the external memory devices and the CPU. The configuration described here is a configuration developed to work on the Olimex development board. The EMC is fed with the *pclk*, which runs at 14.7456 Mhz. This gives a clock period of the *pclk* of $67.8 \cdot 10^{-9} sec$.

The external FLASH	
Parameter	Value
IDCY	4
WST1	2
RBLE	1
WST2	0
BUSERR	-
WPERR	-
WP	0
BM	0
MW	01
AT	00

(a)

The External RAM	
Parameter	Value
IDCY	0
WST1	0
RBLE	1
WST2	0
BUSERR	-
WPERR	-
WP	0
BM	0
MW	10
AT	00

(b)

Table 6.1: Configuration values for the external FLASH and the static RAM connected through the EMC.

The FLASH memory The external FLASH memory is connected to the bank zero of the EMC, why the relevant configuration register is *BCFG0*. The device is a 70 ns device which means that the read cycle time is 70 ns (ref. [Int05, page 25]). The table 6.1(a) shows the configuration *BCFG0* register. This gives the following binary string:
0001 0000 0000 0000 0000 0100 0100 0100 which can be expressed as 0x10000444.

```

    /* Configuration of bank 0 - the external FLASH */
    ldr r4, bc @load base address of bank configuration.
    ldr r0, [r4, #b0] @Load the address of the BCFG0 into the
         register.
    ldr r1, [r4, #conf_b0] @Writing value to control register.
```

The external RAM The external RAM is connected to bank one of the EMC, why the relevant configuration register is *BCFG1*. As a whole writing circle of this RAM last only 10 ns (see [sam04, page 2]) the timing of in general should not give rise to any concerns. A thorough inspection of the timing diagram of the devices ([sam04]) has not indicated any issues either. The values of the parameters as given in [Phi03, table 3.8, page 58] are listed in table 6.1(b). This gives the following binary string:
0010 0000 0000 0000 0000 0100 0000 0000 which can be expressed as 0x20000400.

```

    /* Configuration of bank 1 - the external static RAM */
    ldr r0, [r4, #b1] @Load the address of the BCFG0 into the
        register.
    ldr r1, [r4, #conf_b1] @Writing value to control register.
```

6.1.1.8 The memory test

The third task is to execute the memory test to find a place for the C stack. Therefore the program branches to the memory test:

```

    /*Continue to memory test */
    b memoryTest
```

The memory test has been described in chapter 4.

6.1.1.9 Setting up the C stack

When a usable area of RAM is found by the memory test the C stack is setup. The code performing this operation is found in `cStack.c`.

```
cStack:
/*Input:                       */
/* r0: base address of RAM area */
/* r1: highest valid address of */
/* stack area.                 */
/*Output:                      */
/* r0: Stack pointer           */
/* r1: Stack Limit             */


and r1, r1, #0xFFFFFFFC @Align Address.
mov sp, r1 @move address for stack pointer to correct
    register.
sub sl, sp, #STACK_SIZE @calculate stack limit (sl) and places
            @ in correct register
mov r0, sp @Copy stack pointer to r0.
mov r1, sl @Copy stack limit to r1.

b boot @ Branch to C code.
```

This function also calls the boot C function. The stack pointer and the end of the stack area are passed as arguments to the boot function.

6.2 The second phase

The second phase consist of many small steps. In this section only the central steps are reported. The reader is asked to investigate the source code if more details are needed.

1. Initialise central structures.
2. Find a SIB and test integrity of it.
3. If silence period is still lasting, enter silence mode.
4. Check boot counter to determine whether to start the OS or not.
5. Write SIB to record boot attempt and control validity of written data.
6. Perform complete memory test.
7. Test integrity of OS before starting it.

6.2.1 The implementation

The code of the main boot program is placed in `boot.c`. As described in section 5.3 central system information is gathered in the SPS. A central instance of this structure is allocated on the stack as the first thing done by the boot function:

```

    //Used to collect information about FLASH write errors.
    int errorInfo;

    //Variables used by the SIB system. SIB Parameter Structure
    struct SPS sps= {
        (struct SIB *) BEG_ADDRESS, /*Beginning address of the
            array 'sibs'.*/
        0x0 , /*Pointer to the latest valid SIB or the default
            SIB. */
        -1, /*Index of theSib in the array 'sibs'. */
        MAX_NUM_OF_SIBs, /*number of SIBs in array 'sibs'. */
        0x0, /*Pointer to default SIB. */
        DEFAULT_SIB /* temporary version of the SIB used during
            alteration of values in the SIB */
    };
    struct SPS *theSps = &sps;
```

The most important information in the SPS is a pointer to the valid SIB. It is called `theSib`. The pointer to the central instance of the SPS is called `theSps`.

Together with this a few status variables are also put on the stack. The extensive 'call by reference' usage of these central variables reduce the stack usage to a minimum.

As described in section 5.1 the system information which should be saved during reboots is stored in SIBs. Therefore the task solved by the C program is to find the most recent SIB and test its integrity.

```

    if (findSIB(theSps) != 0)
        /*writeLog("Couldn't find any sib. Use default")*/;

    if (testTheSib(theSps->theSib) != 0) {
        /*TODO: writeLog("sib contains invalid data")*/;
        failsafe(theSps->theSib);
```

If no valid SIB is found in the FLASH, the default configuration is used.

6.2.1.1 The silence period

The completion of the silence period just after launch is controlled by the `launchSilence` function which is called from the main function

```

    //If first start
    if (theSps->theSib->launchBit > 0) {
        //TODO: Set hold Flag to COMMpic
        result = launchSilence(errorInfo,theSps->idxOfTheSib);
        //TODO: Remove hold flag from COMMpic
```

6.2.1.2 Boot counter test

The boot counter is inspected to determine whether the OS should be started or the FS should be started instead.

```

    //Test boot counter
    if (theSps->theSib->bootCounter == 0)
```

6.2.1.3 Recording of boot attempt

At each boot attempt a new SIB should be written to record the boot attempt. This storage operation is carried out as early as possible in the boot process to ensure that the information about the attempt is stored in case of an uncontrolled reboot.

```

    //Decrement boot counter
    result = decretBootC(theSps,(int *)&errorInfo);
    if (result != 0)
        failsafe(theSps->theSib);

    if (findSIB(theSps) != 0)
        failsafe(theSps->theSib);

    if (testTheSib(theSps->theSib) != 0)
```

6.2.1.4 The complete memory test

If the C stack not is placed in the external RAM a complete memory test is carried out.

```

    //Test RAM completely for memory errors.
    if (stackLimit < RAM1_BASE) {
        result = memTestC((datum *) RAM0_BASE, (stackLimit -
            RAM0_BASE));
        if (result != 0)
            return result;
        result = memTestC((datum *) RAM1_BASE, RAM1_LENGTH);
        if (result != 0)
            return result;
    }
    else
```

If this memory test finds any flawed memory locations the FS software is started instead of starting the OS i.e. bring the satellite in nominal mode.

6.2.1.5 Testing validity of the OS image

The final operation which is to be carried out before the OS is started is to test its integrity. This is done by calculating its checksum:

```

    //Check checksum of OS before start
    if(crcCompute((unsigned char *)theSps->theSib->eCosBeg, (
        theSps->theSib->eCosEnd - theSps->theSib->eCosBeg)) ==
        theSps->theSib->eCosCheck );
    //start nominal mode.
    else
```

If the test is passed the OS is started. The procedure for starting the OS has not been clarified yet. The SIB contains a field where a pointer to an entry point could be stored. The OS should reinitialise all memory usage as the stack allocated for the boot and FS software is to small.

CHAPTER 7

Compilation and debugging

This chapter presents the programs and techniques used for building, debugging and testing the boot and FS software for the DTUsat-2. Basically all programs used are parts of the GNU GCC tool chain. The chapter will primarily describe and discuss the techniques used for debugging and testing. The reason for this is that the building process is automated by the GNU Make tool and therefore should not be subject to any problems.

7.1 Building and compilation

The tool chain used in the build process is a GNU GCC based tool chain based on GCC version 4.1.1. This tool chain is chosen because it is used to build the OS used on the DTUsat-2. To ensure compatibility between the binary files used on the satellite all programs will be built using the same tool chain.

7.1.1 Compilation of assembly modules

Some of the functions used early in the boot process are implemented in the assembly language and compiled using the GNU Assembler (GAS). To be able to use the facilities of preprocessing i.e. primary text substitution, the assembly files should be run through the GNU C preprocessor (CPP) before being assembled by GAS. Unfortunately GAS does not call CPP during its working cycle. Therefore it should be called manually before the file is processed by GAS.

This is done by a rule in the Makefile and the extension of the output file is 's', where the input has the extension 'S'. This also means that changes should be made to the file having a capital 'S' as extension and not a lower case 's'.

After that the GAS is called to process the output file.

7.1.2 Pitfalls during the building process

This section contains descriptions of some of the problems encountered during the build process, and the solution found to avoid them. The section is thought of as a way to hand over experiences from developer to developer and an inspiration when other problems should be solved in connection with the building process.

7.1.2.1 Handling floating point

The ARM7TDMI processor does not contain any floating point unit in its core. Therefore all FP operations needs to be handled by software functions. This is normally done by including a standard function library for example the `libgcc.a`-library which contains the necessary functions.

If a program does not contain any usage of FP data types it does not need the FP library function in order to compile.

For compatibility reasons the binary ARM format have the possibility of containing both software based and hardware based FP handling, but not both in the same binary. This is controlled by the "-msoft-float"- and "-mhard-float"-flags of the compiler.

The principle for handling FP influence the call conventions of the binary why all functions must use the same FP handling scheme. More precisely it influence the *Application Binary Interface* (ABI).

Because the program contains objects assembled from assembly source code and objects compiled from normal C-code both the assembler and compiler need to get the correct flags. Unfortunately the “msoft-float”- and “mhard-float”-flags are unknown to the assembler. Instead the “-mfloat-abi=soft” construction should be used which is known by both the GAS and the GCC.

7.1.2.2 Handling binaries containing both thumb and 32bit code

In order to make binaries which contain both 32-bit code and thumb code or just contains calls to functions compiled into thumb code, the compiler needs a certain flag: “-mthumb-interwork”. The IAP functions are compiled as thumb code why any binary containing calls to the IAP functions should be compiled with the flag in order to work correctly.

7.2 The linker scripts

The DTUsat-2 has an unique memory configuration. This configuration consists of different ROM, FLASH and RAM areas where some are remapped.

The memory configuration also differs between the Olimex test board and the flat-sat, and between the flat-sat and the flight configuration.

On the Olimex test board the memory configuration is straight forward: All software related to the boot system (interrupt vectors, boot and FS software) is placed in the internal FLASH memory from address zero. No remapping of the interrupt vectors are done, because they are unused by the boot and FS software.

On the flat-sat the 64 bytes containing the interrupt vectors and their checksum are placed as the 64 first bytes in bank zero of the external memory interface. This is because bank zero in the flight configuration will be connected to a ROM containing the boot software and the FS software. To simulate this a ROM only containing the interrupt vectors and their checksum is installed on the flat-sat. The reset vector contains a branch instruction which branch to the beginning of the second block of the internal FLASH. This block should then contain the boot and FS software[1]. In the flight configuration all software related to the boot system is placed in a ROM connected through bank zero on the external memory interface.

Therefore three tailored linker scripts are necessary: one for the configuration

[1]The first block in the internal FLASH is allocated for the SIBs

of the Olimex, one for the flat sat and one for the flight configuration.

Only the linker script for the Olimex development board is implemented. In this script no texttt.section directives is given since they induced an error during the linking process which could not be solved in this project.

7.3 Debugging and Test

The debugging and tests were carried out using the GDB version 6.5.0 together with the Tcl/Tk based GUI *Insight*. Through these programs the program which was subject to the test or debugging was executed on the ARM simulator which is a part of GDB.

The *Insight* program is a graphical front end for GDB. The GUI gives the user a better overview of the registers, memory and variable values during execution and ease the placement of break points. This is primarily important during debugging.

7.3.1 The assembly language implementation of the memory test

The memory test routine were called through a simple test harness also implemented in the assembly language. The source code can be inspected in listings F.18. The test harness is generic in the sense that it branches to a label called *main* which is defined as global. If it is linked with any object file containing a label called *main* it will branch to this during execution.

7.3.2 Using GDB command scripts

The GDB has a script system which works almost as the shell script concept. It is possible to write scripts using the same commands which are available in the command line interface. Besides that it has some extra commands which makes it possible to write scripts containing conditionals and loops. See [gdb06, sec. 20] for the extra commands.

The scripts are used to automate tests and make them reproducible. Two examples of used scripts will be given here to illustrate the used commands and how they are combined.

7.3.2.1 Testing the value of registers

The first script which will be described is a simple test of the value of a range of registers. It is used to test the initialisation part of the memory test function implemented in assembly language:

```
define test1
#Test if values are correct initialised in the beginning
#of the function.
   echo *** Test 1 *** \n
   #Stop program if running.
   kill
   #Set temporary breakpoint.
   tbreak TestMemory
   #Start program.
   r
   echo Test 1:
   #Test Base address of RAM0.
   reg0 0
   echo Test 1:
   #Test highest valid address of RAM0.
   reg1 0x1FFF
   echo Test 1:
   #Test pointer to current byte address.
   reg2 0x1FFF
   echo Test 1:
   #Test Stack size
   reg8 0x400
end
```

The lines starting with a hash mark (#) are comments. The functions called 'reg0' to 'reg8' are defined earlier in the source file (see appendix F.19 line 151 - 173) which show the ability to define and call functions just as this function (test1) is defined.

The operating principle of this is first to define a break point, run the program until the break point and test the values of the registers to ensure that they are correct. The break point defined in this function is actually a temporary break point which means that it is removed automatically when the program pass it the first time.

7.3.2.2 Fault injection during execution

The next script is more advanced and illustrates the ability to do fault injection. This script is used to test the memory test functions fault tolerance by handling a transient fault in the memory.

```
define test24
echo *** Test 24 *** \n
   kill
   tbreak WriNormIni
   break wp1Test24
   r
   c
   c 13
   set $r11 = 0x25
   c
   c 10
   echo Test 24:
   memByte 0xf2 0x1ff2
   echo Test 24:
   memByte 0xf1 0x1ff1
   echo Test 24:
   memByte 0xf0 0x1ff0
   echo Test 24:
   memByte 0xef 0x1fef
   echo Test 24:
   memByte 0xee 0x1fee
   echo Test 24:
   memByte 0xed 0x1fed
   echo Test 24: \n
   x /40xb 0x1fe0
   clear wp1Test24
end
```

The operating principle of this script is to define two break points. The first is temporary and only used as initialisation of the script. The second is used during the fault injection where the first is passed twelve times and the thirteenths time the program is stopped. Then the value of register 11 is changed which is the fault injection and the break point is passed ten times again. After that the values of a row of memory addresses are tested to see that the program has continued correctly. Finally a dump of 40 memory addresses starting at 0x1fe0 is done and the break point is removed. The memory dump is only done as a service to the operator who then can inspect the memory area where the fault was injected.

As it can be seen in both the mentioned examples absolute values are used in the scripts. A better solution would have been to assign the basic values as base address of the memory areas to some variables and then calculate the needed values from these during the execution of the test. This would have made it much easier to adapt the test to another environment for example if the memory boundaries are changed. As the scripts are implemented right now it is necessary to do some search and replace in the scripts before they can test systems with a different memory setup.

Chapter 8

The timed models

8.1 Introduction

This chapter contains descriptions and analysis of two formal models. The first is a model of the basic communication between the OBC and the beacon module of the radio subsystem (COMM).

The second model is a model of the functionality of the memory test carried out during the boot of the DTUsat-1. This second model does not model any concurrency or parallel issues. Instead it models the functionality and behavior of a sequential assembler routine which carry out a simple memory test.

Both models are built in the formal modelling language Uppaal. The two main arguments for using formal models are:

1. A formal model makes it possible to reason precisely about the properties of a system and it makes it possible to prove certain properties of the system.
2. A formal model makes it possible to concentrate on the central aspects of the functionality and behavior of the system putting less important issues in the background.

8.1.1 The Uppaal Modelling language

An Uppaal model consist of a network of connected **processes**. Each process is a **timed automaton** described as a state diagram consisting of **places** and **edges** (see examples in appendix H). The places represent the individual states and the edges represent the transitions between the states. An edge can be **guarded** by a logical expression which needs to be fulfilled for the edge to be **fired** i.e. used as path from one state to another. The edges can also contain **actions** which should be carried out when the edge is fired. The places can contain **invariants** which should always be fulfilled when an automaton is in that particular state. The processes are connected directly through **channels**, which is a primitive invented to synchronise two processes. The processes are also connected indirectly through reading and writing of **global variables**. The Uppaal modelling language is further described in [BDL04] and on its homepage: [UA].

8.2 Communication between OBC and COMM

The boot and FS software is kept as simple as possible. This means that any means which could add unpredictability or complexity are avoided. Besides the interrupt system which raise the unpredictability of the system, any concurrency are also avoided. Therefore both the boot and FS software is programmed in a purely sequential fashion.

There is one area where concurrency and the unpredictability it gives cannot be avoided: the communication between the different subsystems. The majority of this communication is handled by standard bus systems as the CAN or SPI bus which ensures high predictability even in this asynchronous context. These buses will not be considered further.

Besides that it is also necessary to establish some simple direct communication between the OBC and beacon module of the COMM subsystem. The first model which is described in this chapter analyse this issue.

The COMM subsystem has not been described earlier, why a short description is presented here. The COMM subsystem has not been designed or implemented yet but following information can be considered predefined why a potential design should comply with it.

The COMM subsystem is essentially a radio. It consists of a transmitter, a receiver and some control logic. It is both capable of sending data provided by the OBC and generate its own data packets. The last kind of packets are some simple beacon packets which only contain basic information about the state of the satellite.

The COMM module works by sending beacon packets periodically if it does not get any other commands from the OBC. The possible other commands are a silence command which asks the COMM to stop sending beacon packets and a send command which asks the subsystem to send some data provided by the OBC. Finally it is also capable of receiving data packets from the ground station.

The COMM subsystem is an autonomic subsystem, which means that it contains its own micro controller. This concept entails that the only provision which needs to be satisfied for the COMM subsystem to operate is that it is supplied with power. The operation of the module is thus not dependent of the OBC being in an operational condition or directly controlled by it. It will therefore operate in parallel to the OBC. This also entails that the communication between the OBC and the COMM is based on a communication protocol with some timing constraints being observed by both systems.

8.2.1 Communication protocols

Communication protocols often relies on a specific timing which is a part of the protocol description. If the sender or receiver violates the timing constraints described in the protocol the consequence is failure of the communication. In worse cases the receiver, the sender or both are led into an illegal state causing a deadlock in either one or both of the modules. Therefore the time-wise properties should always be verified when a new protocol is designed.

A formal model of the communication between the sender and the receiver should be built. This model should then be used to simulate and verify the temporal properties of the communication using a model checker.

The communication protocol used between the OBC and the COMM is designed for this specific communication set up only and should therefore be verified to ensure that it is flawless.

8.2.2 The modelled system

The OBC and the COMM are connected through two different connections: A SPI bus between the OBC and the COMM which is used for data transmission and command signaling, and a direct link using four pins of the GPIO port. The last connection is used during the boot phase and in nominal mode.

The connection through the GPIO is connected to the *beacon module* of the COMM subsystem. The beacon module sends status beacons from the satellite announcing the state and condition of the satellite. It is able to send two different kinds of beacons: One type contains the temperature of the satellite only. This type of message only requires the EPS, COMM and beacon module to be in operational condition. The second type contains more information. This information is collected by the OBC and therefore rely on the OBC being in operational condition.

When the satellite is launched it needs to be silent for the first fifteen minutes after launch. It is also necessary that the OBC can turn off the beacon module while it use the radio. This is signaled through the connection made through the GPIO.

The connection through the GPIO consist of four wires. Three of them are signaling the mode of the satellite. The three wires carry the same value. They are triplicated for redundancy. The fourth wire is a serial data connection transferring status information from the OBC to the beacon module. A schematic figure of the connection configuration is presented in figure 8.1.

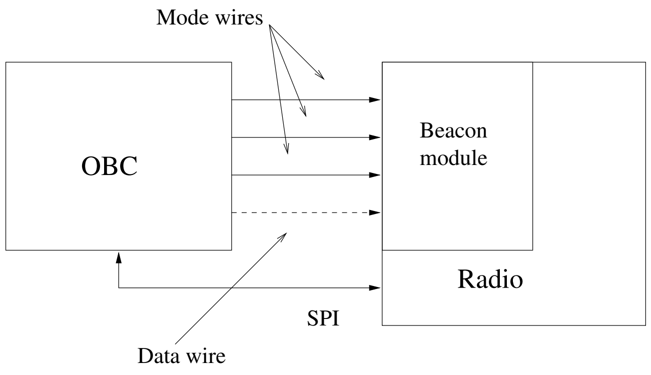


Figure 8.1: Schematic illustration of the wire connection between the OBC and the beacon module of the COMM subsystem. All unnecessary details are discarded.

The connection can be in three different modes: silence mode, FS mode and nominal mode. The individual wires can be in three different states: low (carrying a zero (0)), high (carrying a one (1)) and undefined (carrying an undefined floating value (X), optionally a well defined data signal). In table 8.1 the individual configurations of the wires are given for the three modes.

Wires/Mode	Silence	FS mode	Nominal
Wire 0	0	0	1
Wire 1	0	0	1
Wire 2	0	0	1
Data wire	1	0	X

Table 8.1: Table showing value configurations of the wires in the OBC-beacon-module connection. “X” means undefined or serial data.

8.2.3 The model

The model is built from nine processes each representing a part of the system. The nine processes are made from six process templates. They are interconnected as shown in figure 8.2.

The individual processes are shown in appendix H.1. Below is a description of some of the processes and the issues encountered during the implementation of them.

8.2.4 The wire

As part of the system the wires are modelled too. The easiest way to model the wire is as a global variable. This solution is however too simple for our purpose. The reason for this is that this model should contain the possibility for fault injection on the wire.

Therefore the wire is modelled as a simple process containing two locations. In order to ensure that the signal is propagated instantly from the OBC to the COMM it is necessary to synchronise the edges through urgent channels. Channels do not have any equivalents in the real world and the model therefore uses abstract objects that cannot be implemented in the real world. On the other hand a simple wire can be considered urgent except from loading time deriving from capacitive properties of the wires which should not be considered in this model.

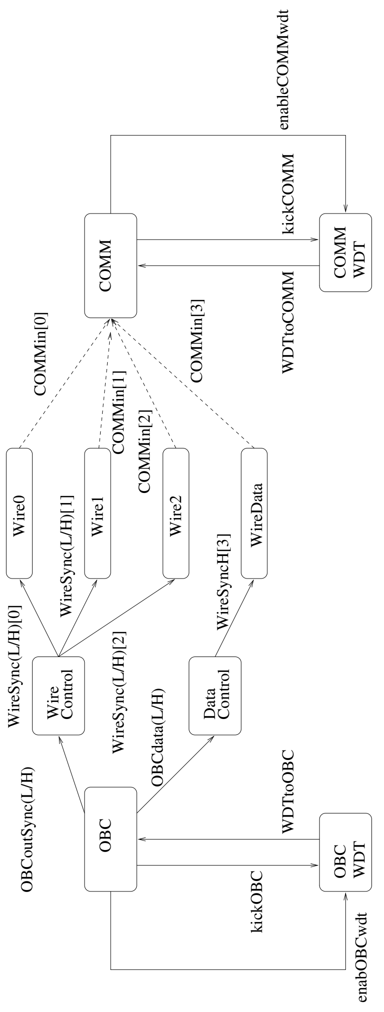


Figure 8.2: Figure showing the interconnections between the processes in the OBC-COMM model. Solid lines represent channel connections and dashed lines represent connections through global variables. Only the important interconnections are showed.

To solve this issue the wires are synchronised using urgent channels when the OBC changes its state output and the wires are updating an boolean value in COMM reflecting the state of the wire.

No fault injection is actually implemented in this model but the ability to do it later is kept open.

8.2.5 The OBC

The model of the OBC is simple and illustrates that the boot procedure is a simple sequence with only a minimum of branching.

8.2.6 Modeling a reset of one of the subsystems

The Uppaal modelling language lacks primitives which support simple modelling of reset operation which will bring the system back to initial state instantly no matter which location and state it may be in.

This problem also applies to the ability to model exception handling capabilities of the modelled system.

The only way to simulate this is by make a new location e.g. called 'reset' and connect it to *all* locations. This is a quite cumbersome way to do it which will make a big model even more unclear. If a fault injection mechanism also should be able to reset the system at arbitrary times this add another 'reset' net besides the 'real' reset net. To reduce the number of places and edges in the model the reset tree is extended to do any kind of fault injection resulting in a reset of the system. This is simply done by implementing a fault injection process which also synchronise the OBC over the 'reset' channel.

8.2.7 Setting up hold mode

The COMM needs to have an initial delay to wait for the OBC to setup the control lines between the two. If they are not setup correctly before they are read by the COMM they would be in an undefined state which could be interpreted as safe mode. This would initiate transmission of safe mode beacons. This is an illegal operation in the first 900 seconds and therefore be avoided.

Another solution could be to let the COMM setup the wires itself as a part of its initiation. After that the OBC could change it when ready. This would solve the timing problem making the system tolerant to changes in the timing of the OBC's initiation phase. This solution was modelled successfully. It is not known at the moment whether this solution can be implemented or not. The problem comes from the fact that the COMM needs to write to its own input registers which may be impossible due to the way the hardware is designed.

Another major problem concerning this solution is that it possesses a potential race condition. This results in a situation where the input of the COMM could be put into an undefined state if both the OBC and the COMM try to write to the input at the same time. This last concern should result in that this solution is abandoned.

8.2.8 Open issues

In the present version the model is unable to reason precisely about the communication between the OBC and the COMM. The reason for this is lack of information. All timing values are still unavailable and need to measured on the flat sat and incorporated in the model before the model gets useful. It has not been possible to gather any of the relevant values because the hardware is not ready yet.

A way to simplify the model is to parameterise all timing values by having a global clock speed as a constant and then let the dependent variable values be the product of a multiplication between this global clock and an individual parameter. This will also ease adaption of the model to changes in the hardware.

8.2.9 Conclusion

A model of the communication set up between OBC and the COMM beacon module has been designed and implemented. Some parameters and values still needs to be adjusted.

During development of the model it was found out that the time it takes to send a beacon or data packet of course should be shorter than the timeout of the WDT. This result has been found using the simulator on early versions of the model.

The model has been verified not to deadlock in its present version.

8.3 Modelling the memory test of DTUsat-1

The purpose of the model of the memory test used in the initialisation phase of the DTUsat-1 is to verify whether this part of the software is able to go into an infinite loop or not. The following block of assembly code should be modelled:

```
Write:    /* r0=base, r1=higest valid address,
     * r2=pointer to actual address,
     * r3=tmp, r8=0x00000000,
     * r9= 0xFFFFFFFF, r10=STACK_SIZE */

    str r9, [r2] /* save HIGH in mem */
    ldr r3, [r2] /* read pattern back again */
    cmp r9, r3   /* test */
    bne Fail
    str r8, [r2] /* save LOW in mem */
    ldr r3, [r2] /* read pattern back again */
    cmp r8, r3   /* test */
    bne Fail
    sub r3, r1, r2 /* space from pointer to top */
    cmp r10, r3    /* compare with needed stack space */
    ble CRTSetup   /* setup stack if enough space */
    sub r2, r2, #4 /* sub 4 from address */
    cmp r2, r0     /* compare pointer to base */
    movlt pc, r14  /* return if actual address was below base
        addr. */
    b Write
Fail: mov r1, r2
    b Write
```

As it can be seen the software which should be verified is already implemented. This entails big constraints on the model since it should be so close to the actual implementation that it can reason about its behavior.

Therefore a 'precautionary principle' has been applied which simply demands the model to be as close to the actual implementation as possible. The implementation of this principle has been done by letting each instruction of assembly code be represented in the model by one or two edges. In figure 8.3 the connections between the individual edges and assembler instructions are presented.

Another reason for precision in the model is that only a very small but very important modification needs to be carried out in order to remove the fault from the implementation. To be able to demonstrate how little change it is the model needs to be quite detailed.

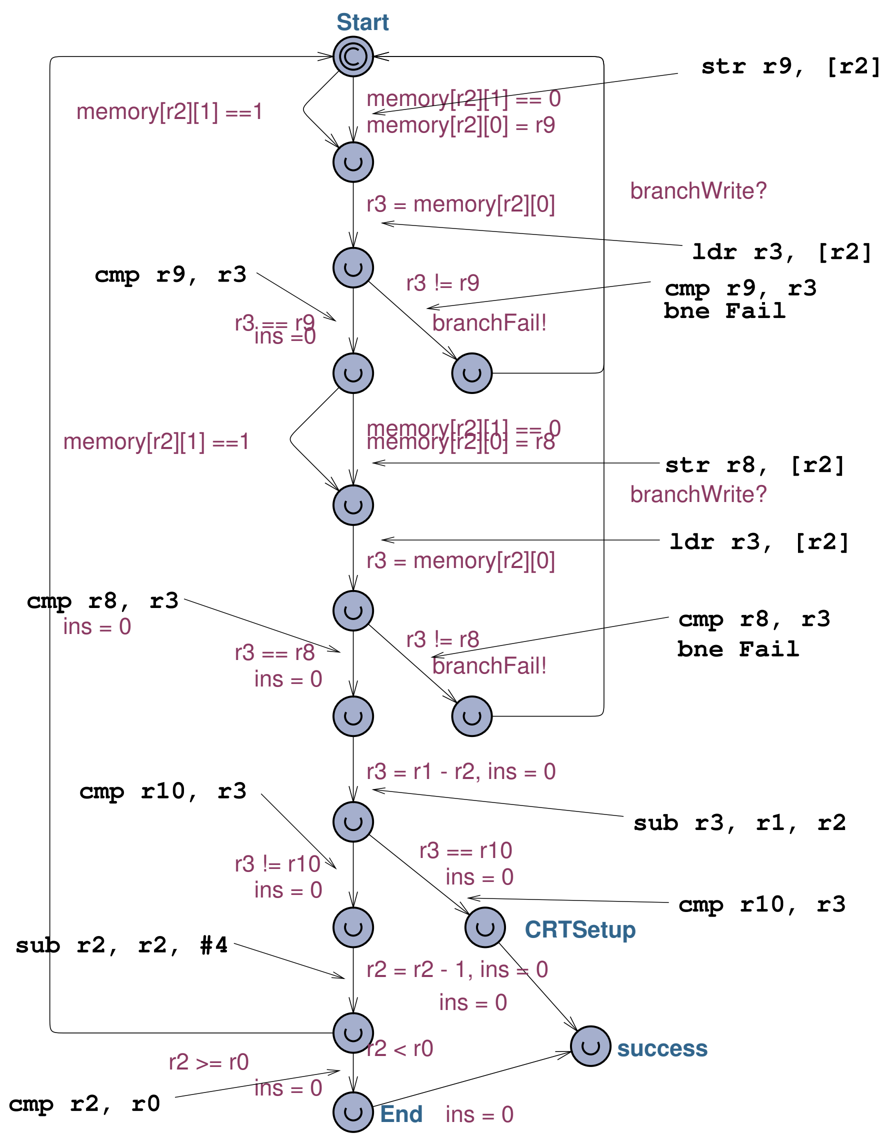


Figure 8.3: Figure showing connection between edges and assembler instructions in model of memory test on DTUsat-1.

8.3.1 The temporal issues in the model

The assembly code which should be modelled is executed on a single processor in a linear fashion. It does not use any kind of clocks either. This removes all concurrency and temporal issues of the model since no race conditions could occur. Therefore it could be argued that it is unnecessary to model the function using UPPAAL. On the other hand the UPPAAL tool gives a good view of what is happening during the execution. It could also be argued that not using clocks in any of the processes converts UPPAAL to a tool verifying systems of interconnected state machines.

It is possible to use UPPAAL in this way if special precautionary measures are used: Because UPPAAL is developed to simulate and verify models which consist of timed automatons, time will always be taken into consideration in the verification of a model. This means that if a model gets into a stable state and no invariants in any of the active places contains time constraints which force the model to the next state, it can stay in that stable state forever. The solution to this is to make almost all places **urgent** or **committed**.

If a state of the system contains any urgent place, the time is not allowed to pass before the urgent place is left. This forces the model to continue to the next state all the time because all states of the system contains urgent places.

If a state contains a committed place, time is not allowed to pass and the verifier should always fire an edge which goes out from a committed place when continuing to the next state. This rule also forces the model to continue all the time and it forces the model to choose an edge from a committed place if it can choose between an edge comming from a committed place and one coming from an urgent place. If a state contains more than one committed place the verifier is allowed to choose any of the legal edges from any of the committed places.

Another precautionary measure which should be used, is the ability to limit the number of edges that can be fired in one automaton before an edge of another automaton should be fired. It is necessary to consider this issue because the fault injection automaton for instance is not directly connected to any other automatons through channels and it is not connected indirectly through any global variables either. This gives the verifier the ability to let this automaton circle through its states forever without firing any of the edges of the other automatons in the model. The solution to this problem is to connect the considered automaton to the other automatons through a global variable. This variable is used as a counter which is incremented by all edges in one automaton and reset by all edges in another automaton. The incrementing edges should also be guarded with a limit of the value of the counter. In this way the incrementing automaton is allowed only to fire the number of edges the limit of the counter gives before the other automaton is allowed to run and thereby resetting the counter.

8.3.2 Fault injection

The fault injection used in the model is quite simple even though it is able to model both transient and permanent faults. The fault injection is done by extending the model with a fault injection process which is shown in figure 8.4. This process first injects a fault by changing the value of the memory cell being examined by the memory test function. After that it decides whether the fault should be marked as permanent or not. The result is written in an extra field in the array modelling the memory. The two decisions are of course made randomly.

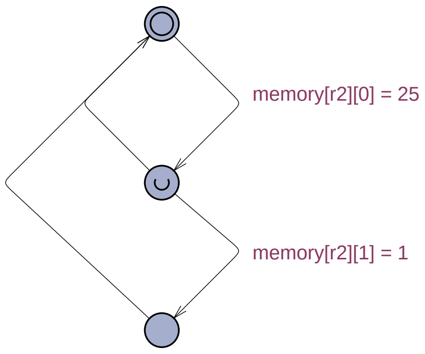


Figure 8.4: The process handling fault injection in the memory test model.

8.3.3 The test case

The fault in the assembly code entails that the test loop will get caught in an infinite loop if a permanent fault is found. Therefore a permanent fault is induced in the memory array at the second location in the array. This is done to show that the code runs correctly until the fault is met.

Query	Without permanent fault	With permanent fault
$A\Box$ not deadlock	property is not satisfied	property is satisfied
$A\Box$ Write.success	property is not satisfied	property is not satisfied
Write.Start $\rightarrow$ Write.success	property is not satisfied	property is not satisfied
$A\Diamond$ Write.success	property is not satisfied	property is not satified
$E\Diamond$ Write.success	property is satisfied	property is not satified

Table 8.2: Queries and result of the tests of the model of the OBC $\leftrightarrow$ COMM interaction.

8.3.4 The results

In table 8.2 the queries together with the results of the tests are presented.

If the memory test should work satisfactorily the second query "$A\Box$ Write.success" should be satisfied since this query requires that the process will always end in the "success" place. This will happen if the C stack is initialised correctly or the whole memory area is read without finding any proper areas for the stack. The verifier is able to find a counter example where the model is trapped in an infinite loop.

Only the weakest query is fulfilled ($E\Diamond$ Write.success). This is because the fault injection is able to induce transient faults such that the model is kept in the upper loop in an infinite loop, see figure 8.5. It is most unlikely that this should happen. The reason is that it requires a lot of transient faults to occur on the same memory address in long period of time. On the other hand it shows that the model contains a potential risk to go into an infinite loop.

As it can be seen not even the weakest property is satisfied if a permanent fault is present in the memory, (see table 8.2). This means that the test will get caught in an infinite loop if it detects the fault type it is designed to find. The model is trapped in an infinite loop in the upper loop of the 'Write'-process as illustrated in figure 8.5.

The solution to the fault is quite simple: The address counter needs to be decremented in the top of the routine. If a *sub* instruction was placed in the top of the routine, it would work correctly, but since it does not decrement the address counter if a permanent fault is encountered the routine is trapped in an infinite loop.

The experience with this assembly routine resulted in that all code which was copied from the code base of the DTUsat-1 project was inspected closely and

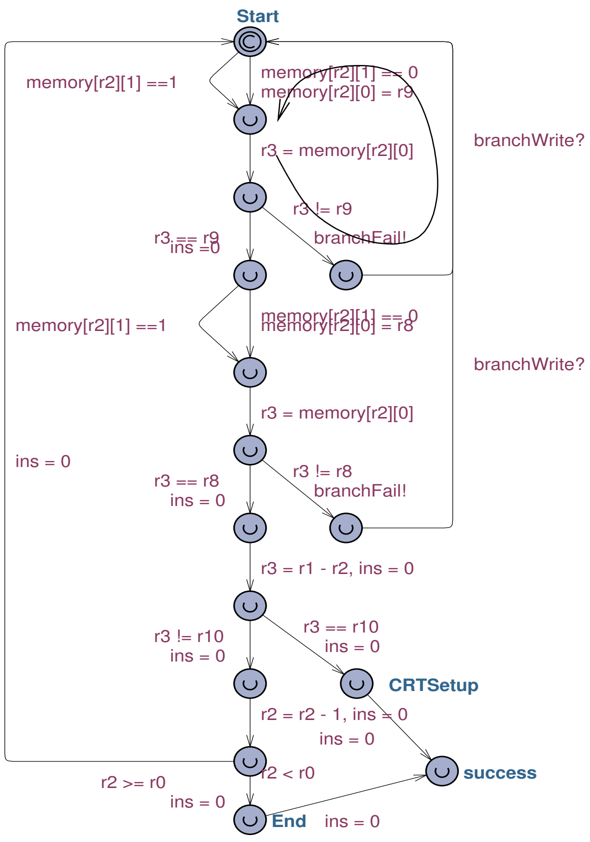


Figure 8.5: Ill. showing the infinite loop in the 'Write'-process.

only if it was totally clear and simple to explain what the code did it was actually used in the DTUsat-2 project.

Also during the implementation of the memory test implemented in assembly language in the DTUsat-2 a lot of attention and emphasis was put into ensuring that no infinite loops were possible in this code. The approach used on DTUsat-1 where the top of the potential stack area is moved down every time a permanent fault is encountered is copied in the DTUsat-2 but it is ensured that the top is actually moved down *below* the flawed memory address.

Conclusion

Summary

This report has presented the work of my master's project. An overview of dependability theory and some of the available means and tools to raise the overall dependability of system was given in chapter 1.

An analysis of the boot procedure, resulting in some requirements to the software has been performed. This analysis revealed that the memory test of the system before setting up a C stack needed special care why a dedicated analysis of this was done.

On the basis of these analysis a boot program was designed and implemented. The boot program is divided in two parts: A part implemented in assembly which handle the tasks which needs to be carried out before a C stack can be established: interrupt disabling, WDT configuration and start, and some more hardware and memory configurations.

The other part of the software is implemented in C. This part handles the silence period, the system information handling and boot attempt control, ensuring that the change from FS to nominal mode is only performed if the satellite is in a state where it is able to run the OS fault free.

These two parts of the boot program constitutes an implementation of the boot software which complies to the demands outlined in chapter 3.

The memory test implemented in assembly code has two main features: If any fault is found the type is identified and if any fault free memory area large enough to host the C stack exists on the system it is found. The memory test has been tested systematically using an automated test utilising the GDB script language. This memory test implements a fault tolerance technique which ensures that faults in the RAM are handled in a controlled way.

The FS software of the DTUsat-1 has been updated and modified in order to work together with the new version of the boot program for DTUsat-2.

Two formal models have been designed and implemented in the Uppaal modelling language. The first models the communication between the OBC and the COMM subsystems. This model has only been designed and implemented.

The second model models a part of the assembly code of the boot program used on the DTUsat-1. This model verifies that the code possesses the ability to get caught in an infinite loop if a fault in the memory is detected.

8.4 Main contributions

The following main contributions have been contributed to the DTUsat-2 project:

- Extended memory test.
- Redesign of the boot procedure and the system information block
- Adaption of the FS software from the DTUsat-1 to the DTUsat-2

8.5 Future work

The boot and FS software has not been tested systematically and they both lacks some functionality primary due to lack of information about the hardware. The code which has been developed in this project is considered operational when it has been tested.

The FS software has nor been tested functionally together with the communication application 'FSTerm'.

A procedure for start of the OS has not been developed why it needs to developed and implemented. The supplied fields in the system information block should ease this task however.

In the present program the WDT is only fed/kicked during the silence period. The worst case execution time of the memory tests should be measured and the calls of the WDT kicking function should be added at relevant places in the code.

The linker scripts for the flat sat and the flight configuration has not been implemented yet, why this task also needs to be carried out.

In the section describing the FLASH driver for the internal FLASH some design preconditions are described. These preconditions should be tested on the FLASH and the drivers corrected according to the results.

8.6 Final conclusion

Index

Appendix A

Schematic of the satellite

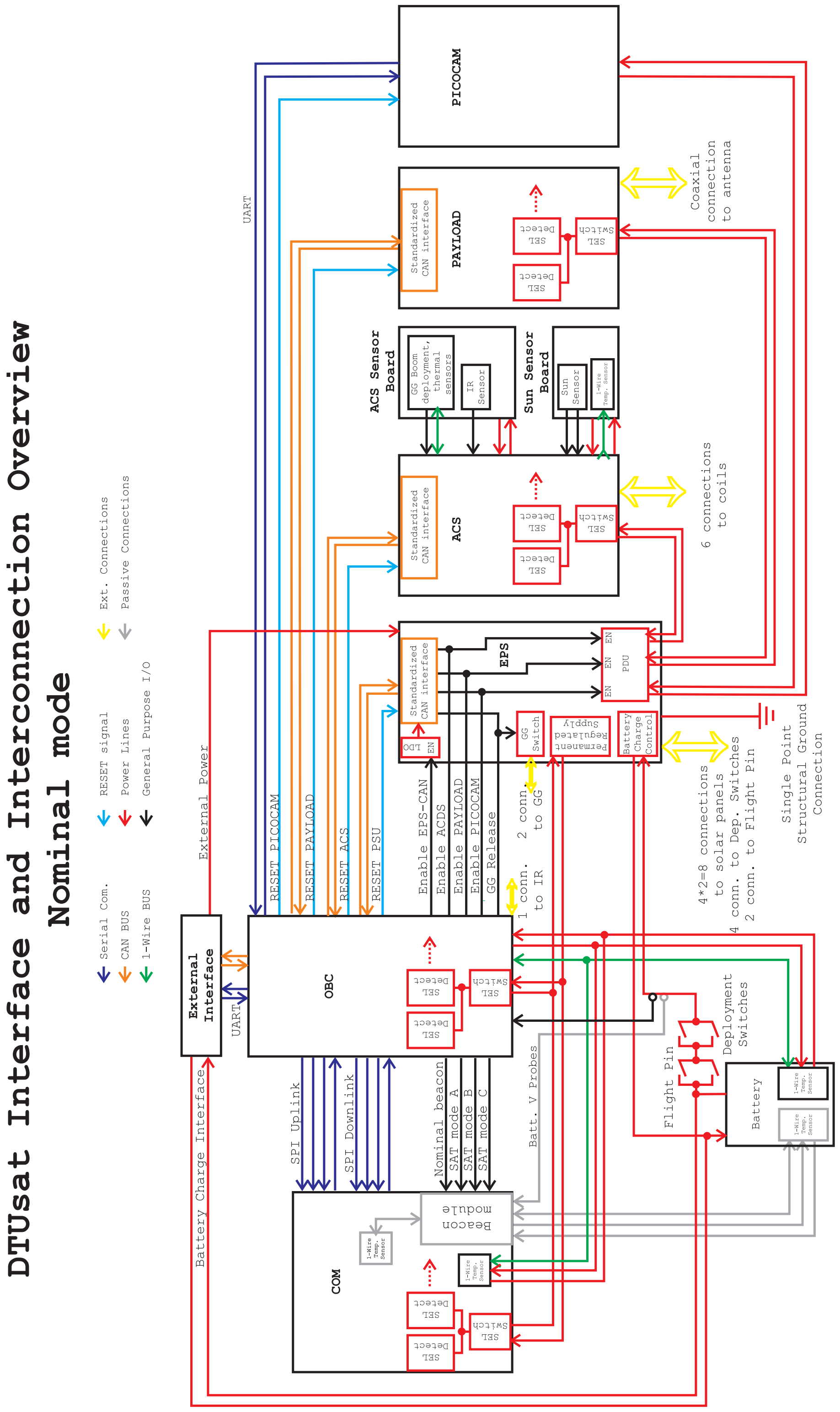


Figure A.1: The schematic system layout of DTUsat-2. Created by Jonas

APPENDIX B

Solutions used in report generation

As the reader may have noticed this report is generated using LaTeX. This appendix describes how different issues have been solved during report generation using various LaTeX-packages.

This appendix has been written to inspire future developers to document and list their code in and simple, easy readable way which only demands a minimum of work.

B.1 Source listing

All source listing in the report and appendices have been done using the package Listings version 1.3c. This package has been chosen for its many supported languages, simple addition of code to handle new languages, many features and nice output.

Before any listings can be included in the document the following line should be added to the preamble of the document:

```
\usepackage{listings}
```

As it can be seen no parameters are given during initialisation of the package but the configuration is carried out as a part of the individual listings commands.

To include a file containing source code a listings command should be given at the place where one wants the listing placed in the document:

```
\lstinputlisting[numbers=left,numberstyle=\tiny,{language=C},
label=lst:app:sysInfoh,caption={Header file for the System Information
Block (SIB) functions.}]{../../Source/DTUsat2/Lib/sysInfo.h}
```

This line tells which file to include, which language the file is written in a various other things which influence the look of the listing.

B.1.1 Listing of assembly code

The ARM assembler language was not supported by the Listings package why a definition of the keywords and comment identifiers etc. needed to be added to the system. This definition is added to the preamble of the document:

```
\lstdefinelanguage[arm7tdmi]{Assembler}{
morekeywords={add,and,b,beq,bgt,bl,bne,cmp,cmpeq,eor,ldmia,ldr,ldrb,
mov,moveq,movlt,movs,mrs,msr,mvn,nop,stmfd,stmia,str,strb,sub,subs},
morekeywords=[2]{.arm,.align,.end,.global,.long,.section,.text,.word},
alsoletter=.,
sensitive=false,
morecomment=[l]@,
morecomment=[s]{/*}{*/}
}[keywords,comments]
```

As it can be seen the language and the dialect is given on the first line after that two groups of key words are defined. Only the instructions used in source code are listed here in this version. To make the package use different typography of different key words they are defined in two different groups.

B.1.2 Adding entries to the table of contents and the index

A need feature of the Listings package is the possibility to include LaTeXcode in the source files and let the LaTeXengine interpret it as such. This feature for example allows the author to include entries pointing to the individual functions declarations into the table of content. To use this feature a
lstset{escapeinside={/**}{*/}} declaration needs to be included in the document above the place where the source code is included. This declaration define a set of LaTeXcode identifiers in the source code, which should surround the LaTeXcode in the source file. As it can be seen here the identifiers are defined based on the comment identifiers of the C-language. This is done so to make the C-compiler ignore the LaTeXcodes when the source file is compiled.

In the source file normal LaTeXcode can now be included:

```
/**\index{findSIB!\textit{source code}}*/
int findSIB(struct SPS *sps) { /** \label{lst:app:sysInfoc:findSib}
\addcontentsline{toc}{section}{findSIB()}*/
int index = 0;
```

B.1.3 Referencing lines in source code

As it can be seen in the source listing in section B.1.2 a label declaration is also included in the LaTeX code. This is done to make possible to make references to the individual code lines of the source code. If a
ref{} is made inside the document it will return the line number of the that listing.

B.1.4 Presenting source blocks in the text

The Listings package is also able to make listings inside the text. This simply done by specify the first and last line number and file name:

```
\lstinputlisting[numbers=left,numberstyle=\tiny,
{language=[arm7tdmi]Assembler},label=lst:intVec,firstline=43,lastline=54]
{../../Source/DTUsat2/Boot/init.S}
```

Appendix C

Pseudo code of the memory test implemented in Assembly

```
Input: base address, top address, stack size
Output: top address of stack area
initialise pattern -1 ;
initialise byte counter to top address ;
for (top address - current address) $<$ stack size do
    if current address $\geq$ base address then
        increment value of pattern ;
        filter out 24 highest bits of pattern ;
        write pattern to byte address ;
        decrement byte counter ;
    else
        start register based FS ;
    end
end
reset pattern to -1 ;
reset byte counter to top address ;
```

Figure C.1: To be continued in figure C.2.

```
for (top address - current address) $<$ stack size do
    increment value of pattern ;
    filter out 24 highest bits of pattern ;
    read byte value stored at address of byte counter ;
    if read value $\neq$ pattern then
        write pattern to byte address ;
        increment byte counter by one;
        invert pattern ;
        filter out the 24 highest bits ;
        if byte address == top of RAM then
            subtract stack size from byte address ;
            write pattern to byte address ;
            add stack size to byte address ;
            invert pattern ;
            filter out the 24 highest bits ;
            decrement byte by one ;
            read byte value stored at address of byte counter ;
        end
        else
            write pattern to byte address ;
            invert pattern ;
            filter out the 24 highest bits ;
            decrement byte counter by one ;
            read byte value stored at address of byte counter ;
        end
        if read value $\neq$ pattern then
            align present address ;
            top address = current address - 1 ;
            reset pattern to initial state ;
            restart test ;
        end
    end
    decrement byte counter ;
end
```

Figure C.2: First part of the memory test algorithm used during the init phase.

```
reset pattern to stored (pattern - 1) ;
reset byte counter to top address ;
for (top address - current address) < stack size do
    increment value of pattern ;
    inverse pattern ;
    filter out 24 highest bits in pattern ;
    write pattern to byte address ;
    decrement byte counter ;
end
reset pattern ;
reset byte counter to top address ;
```

Figure C.3: To be continued in figure C.4.

```
for (top address - current address) < stack size do
    increment value of pattern ;
    inverse pattern ;
    filter out 24 highest bits of pattern ;
    read byte value stored at current address ;
    if read value $\neq$ pattern then
        write pattern to byte address ;
        decrement byte counter by one;
        invert pattern ;
        filter out the 24 highest bits ;
        if byte address == top of RAM then
            subtract stack size from byte address ;
            write pattern to byte address ;
            add stack size to byte address ;
            invert pattern ;
            filter out the 24 highest bits ;
            decrement byte by one ;
            read byte value stored at address of byte counter ;
        end
        else
            write pattern to byte address ;
            invert pattern ;
            filter out the 24 highest bits ;
            decrement byte counter by one ;
            read byte value stored at address of byte counter ;
        end
        if read value $\neq$ pattern then
            align current address ;
            top address = current address - 1 ;
            restart test;
        end
    end
    if (top address - present address) = stacksize then
        start C part of boot process;
    end
    decrement byte counter ;
end
start register based FS software ;
```

Figure C.4: Second part of the memory test algorithm used during the init phase.

Appendix D
Test cases and results from the memory test function

D.1 The structural test

Test no.	Description
1	Test initialisation of address values and stack size during initialisation of the function.
2	Test initialisation of pattern and byte counter during first initialisation.
3 - 5	Test correct reset of values before start of writing and compare values in memory.
6 - 9	Test that counters are incremented or decremented in loops.
10	Test that function returns if base address is passed.
11 - 16	Test that sub-functions branch to next sub-function when stack size is reached.
17	Test that correct patterns are written to byte addresses, when normal pattern is written.
18	Test that pattern is correctly repeated every 256 bytes.
19	Test that correct patterns are written to byte addresses, when inverted patterns are written.
20	Test that inverted pattern is correctly repeated every 256 bytes.
21	Injecting a permanent fault in the memory and test that the test is restarted at next word address. It is found during test of normal patterns.
22	Injecting a permanent fault in the memory and test that the test is restarted at next word address. It is found during test of inverted patterns.
23 - 24	Test that if a transient fault is identified the function will continue without doing anything.

Table D.1: Description of the groups of tests carried out.

Parameter name	register	Test number	
		1	2
RAM0_BASE	r4	0x0	0x0
RAM0_LENGTH	r5	0x00002000	0x00002000
RAM1_BASE	r6	0x00003000	0x00003000
RAM1_LENGTH	r7	0x00004000	0x00004000
STACK_SIZE	r8	0x400	0x400

Table D.2: Values passed to the memory test function during test.

No.	Place identifier in code	Exp. state	Found state	Result
1	label: `TestMemory` (1st pass)	r0: 0x0 r1: 0x1FFF r2: 0x1FFF r8: 0x400	0x0 0x2000 0x2000 0x400	OK OK OK OK
2	label: `WriNormFor` (1st pass)	r2: 0x2000 r3: 0xfffffff	0x2000 0xfffffff	OK OK
3	label: `TestNormFor` (1st pass)	r2: 0x2000 r3: 0xfffffff	0x2000 0xfffffff	OK OK
4	label: `WriInvFor` (1st pass)	r2: 0x2000 r3: 0xfffffff	0x2000 0xfffffff	OK OK
5	label: `TestInvFor` (1st pass)	r2: 0x2000 r3: 0x0	0x2000 0x0	OK OK
6	label: `WriNormFor` (2nd pass)	r2: 0x1fff r3: 0x0	0x1fff 0x0	OK OK
7	label: `TestNormFor` (3nd pass)	r2: 0x1ffe r3: 0x1	0x1ffe 0x1	OK OK
8	label: `WriInvFor` (4nd pass)	r2: 0x1ffd r3: 0x2	0x1ffd 0x2	OK OK
9	label `TestInvFor` (5th pass)	r2: 0x1ffc r3: 0xfc	0x1ffc 0xfc	OK OK
10	break point: memTest.S, line 36	r0: 0x0 r2: 0xfffffff r3: 0xfffffff	r0: 0x0 r2: 0xfffffff r3: 0xfffffff	OK OK OK
11	label: `TestNormPat`	r2: 0x1fff r3: 0x0	r2: 0x1fff r3: 0x0	OK OK
12	label: `WriInvIni`	r2: 0x1fff r3: 0x0	r2: 0x1fff r3: 0x0	OK OK
13	label: `WriInvIni`	r2: 0x1fff r3: 0x0	r2: 0x1fff r3: 0x0	OK OK
14	label: `TestInvPat`	r2: 0x1fff r3: 0x0	r2: 0x1fff r3: 0x0	OK OK
15	label: `SetupCstack`	r2: 0x1fff r3: 0xff	r2: 0x1fff r3: 0xff	OK OK
16	label: `SetupCstack`	r2: 0x1fff r3: 0xff	r2: 0x1fff r3: 0xff	OK OK
17	label: `TestNormPat`	incrementing byte pattern starting at 0x1fff	See app. G.1	OK

Table D.3: List of tests carried out to inspect reset of values of counters and filter.

No.	Place identifier in code	Exp. state	Found state	Result
18	label: `TestNormPat`	incrementing byte pattern restarting at 0x1eff	See app. G.1	OK
19	label: `TestInvPat`	decrementing byte pattern starting at 0x1fff	See app. G.1	OK
20	label: `TestInvPat`	decrementing byte pattern restarting at 0x1eff	See app. G.1	OK
21	label: `WriNormFor`	decrementing byte pattern restarting 0x1fef	See app. G.1	OK
22	label: `WriNormFor`	decrementing byte pattern restarting 0x1fef	See app. G.1	OK
23	label: `wp1Test23`	decrementing byte pattern starting at 0x1fff	See app. G.1	OK
24	label: `wp1Test24`	decrementing byte pattern starting at 0x1fff	See app. G.1	OK

Table D.4: Second part of table showing tests and results from them.

Appendix E

DTUsat-1 related material

E.1 Source code

E.1.1 init.S

```
#include "init.inc"

# OBC initialization
# Boot-group

.align 4
.global __reset
.global micro_delay

.section ".vectors","ax"

__reset:
   b   reset          /* reset */
undefvec:
   ldr   pc,.undefvec         /* Undef */
swivec:
   ldr   pc,.swivec       /* SW */
```

```
pabtvec:
   ldr   pc,.pabtvec      /* P abt */
dabtvec:
   ldr   pc,.dabtvec      /* D abt */
rsvdvec:
   ldr   pc,.rsvdvec      /* reserved */
irqvec:
   ldr   pc,.irqvec       /* irq */
fiqvec:
   ldr   pc,.fiqvec       /* fiq */

.reset:
   .word 0
.undefvec:
   .word _undefvec
.swivec:
   .word _swivec
.pabtvec:
   .word _pabtvec
.dabtvec:
   .word _dabtvec
.rsvdvec:
   .word _rsvdvec
.irqvec:
   .word _irqvec
.fiqvec:
   .word _fiqvec
__end_vectors:


.section .text

_undefvec:
_swivec:
_pabtvec:
_dabtvec:
_rsvdvec:
_irqvec:
_fiqvec:
   /* IRQ and FIQ should probably be disabled here to
      prevent recursion.. */
   stmfd sp!,{r0-r12,r14}
   mrs   r0,cpsr
   stmfd sp!,{r0}
```

```
    mov    r1, sp
    ldr    r0, =(15*4)
    bl exception_handler

exception_loop:            /* Wait for the WD to reboot the
    system */
    b   exception_loop

reset:
    /* Disable IRQ and FIQ (they should be disabled
       by a H/W reset, but better safe than sorry). */

    mrs    r0, CPSR
    ldr    r1, =CPSR_IRQ_DISABLE | CPSR_FIQ_DISABLE
    mvn    r1, r1
    and    r0, r0, r1
    msr    CPSR, r0

    /* Setup memory */

    ldr    r10, PtEBITable   /* get the address of the chip
        select register image */

    movs   r0, pc, LSR #20        /* pc > 0x100000 */

    moveq r10, r10, LSL #12 /* Mask the 12 highest bits of
        the address */
    moveq r10, r10, LSR #12

    /* Copy Chip Select Register Image to Memory
       Controller and command remap */
    ldmia r10!, {r0-r9,r11-r12}    /* load the complete
       image and the EBI base */
    stmia r11!, {r0-r9}       /* store the complete image
       with the remap command */

    mov    pc, r12          /* jump and break the pipeline */


PtEBITable:
    .long EBITable     /* Table for EBI initialization */

EBITable:
    .long 0x01002122      /* 0x01000000 ---*/
    .long 0x03002122      /* 0x03000000 ----*/
    .long 0x02003121      /* 0x02000000, 16MB, 0 h */
    .long 0x30000000      /* unused */
    .long 0x40000000      /* unused */
    .long 0x50000000      /* unused */
    .long 0x60000000      /* unused */
    .long 0x70000000      /* unused */
    .long 0x00000001      /* REMAP command */
    .long 0x00000000      /* 4 memory regions, standard
        read */
    .long EBI_BASE    /* EBI Base address */
    .long PostRemap


PtMEMTable:
    .long MEMTable

MEMTable:
    .long RAM0_BASE           /* r4 */
    .long RAM0_LENGTH-4       /* r5 */
    .long RAM1_BASE           /* r6 */
    .long RAM1_LENGTH-4       /* r7 */
    .long 0x00000000      /* r8 */
    .long 0xFFFFFFFF      /* r9 */
    .long STACK_SIZE      /* r10 */

PostRemap:
    /* Start wacthdog */
#ifndef DISABLE_WATCH_DOG
    ldr    r10, PtWDTable /* get the address of the
        watchdog image */
    ldmia r10, {r0-r4}    /* Load data to registers */
    str r1, [r0, #0]  /* Disable WD */
    str r2, [r0, #4]  /* Setup Clock Mode Register */
    str r3, [r0, #8]  /* Restart timer */
    str r4, [r0, #0]  /* Enable watchdog */
#endif


DoCopyVectors:
    mov    r0,#0
    ldr    r1,=__reset
    ldr    r2,=__end_vectors
```

```
_vector_copy:
   ldr    r3,[r1],#4
   str    r3,[r0],#4
   cmp    r1,r2
   bne    _vector_copy

   /* Do simple mem test to find space for stack */

DoMemTest:
   /* Load memory variables into registers */
   ldr r11, PtMEMTable ;load address of predifined values
   ldmia r11, {r4-r10} ;load predifined values into
       registers

   mov r0, r4           ;move addr. of start of RAM0 to r0
   add r1, r4, r5       ;r1 = RAM0_BASE + RAM0_LENGTH-4
       find end of RAM0
   mov r2, r1           ;move addr. of end of RAM0 to r2
   bl Write /* Test ram0 */

   mov r0, r6           ;move addr. of start of RAM1 to r0
   add r1, r6, r7       ;r1 = RAM1_BASE + RAM1_LENGTH-4
       find end of RAM1
   mov r2, r1           ;move addr. of end of RAM1 to r2
   bl Write /* Test ram1 */

   b DoPANIC /* No memory available */


Write:    /* r0=base, r1=higest valid address,
    * r2=pointer to actual address,
    * r3=tmp, r8=0x00000000,
    * r9= 0xFFFFFFFF, r10=STACK_SIZE */

   str r9, [r2] /* save HIGH in mem */
   ldr r3, [r2] /* read pattern back again */
   cmp r9, r3   /* test */
   bne Fail
   str r8, [r2] /* save LOW in mem */
   ldr r3, [r2] /* read pattern back again */
   cmp r8, r3   /* test */
   bne Fail
   sub r3, r1, r2 /* space from pointer to top */
   cmp r10, r3    /* compare with needed stack space */
```

```
    ble CRTSetup    /* setup stack if enough space */
    sub r2, r2, #4 /* sub 4 from address */
    cmp r2, r0      /* compare pointer to base */
    movlt pc, r14   /* return if actual address was below
        base addr. */
    b Write
Fail: mov r1, r2
    b Write


PtWDTable:
    .long WDTable           /* Table for WD initialization */


WDTable:
    .long   WD_BASE                 /* WD Base address */
    .long   WD_OMR(WD_OMR_DISABLE)          /* Disable WD */
    .long   WD_CMR(0xf,WD_CMR_MCK1024)          /* Clock mode
        */
    .long   WD_CR_RESET                 /* Reset WD */
    .long   WD_OMR(WD_OMR_WDEN | WD_OMR_RSTEN |
        WD_OMR_EXTEN)
                        /* Enable WD, int.+ext. reset */

/* The WD is set up as follows:
    Clock mode resets to 0xffff, clock division by 1024.
    This gives a total scaledown of 64K*1024=64M.
    @16.000Mhz this gives a time-out periode of 4.19 s
*/

CRTSetup:
    /* If we want to use exception stacks,
        we should probably set them here.
    */

    mov     r0,#(CPSR_IRQ_DISABLE|CPSR_FIQ_DISABLE|
        CPSR_IRQ_MODE) ;prepare options and setup correct
        mode
    msr     cpsr,r0 ; move them to cpsr using special
        instruction
    mov     sp,r1 ; update SP with the found one

    mov     r0,#(CPSR_IRQ_DISABLE|CPSR_FIQ_DISABLE|
        CPSR_FIQ_MODE) prepare options and setup correct
    mode
msr   cpsr,r0  ; move them to cpsr using special
    instruction
mov   sp,r1; update SP with the found one

mov   r0,#(CPSR_IRQ_DISABLE|CPSR_FIQ_DISABLE|
    CPSR_UNDEF_MODE)  ;prepare options and setup correct
     mode
msr   cpsr,r0  ; move them to cpsr using special
    instruction
mov   sp,r1; update SP with the found one


mov   r0,#(CPSR_IRQ_DISABLE|CPSR_FIQ_DISABLE|
    CPSR_ABORT_MODE)  ;prepare options and setup correct
     mode
msr   cpsr,r0  ; move them to cpsr using special
    instruction
mov   sp,r1; update SP with the found one


/* The normal mode of operation (for eCos/application)
     is supervisor. */
mov   r0,#(CPSR_IRQ_DISABLE|CPSR_FIQ_DISABLE|
    CPSR_SUPERVISOR_MODE)  ;prepare options and setup
    correct mode
msr   cpsr,r0  ; move them to cpsr using special
    instruction

/* According to the eCos ARM-HAL, some library
routines will cause a "restore from SPSR". */
msr   spsr,r0    ; move them to spsr using special
    instruction

/* setup stackpointers and registers */
mov r3, r1
mov sp, r3      ;setup SP
sub sl, sp, #STACK_SIZE  ;sl (stack limit) calculated
    and stored in correct register r10
mov r2, #STACK_SIZE
mov fp, #0 ;reset argument pointer to 0
mov r7, #0 ;reset another register

mov r0, #0 ;reset another register
```

```
    mov r1, #0 ;reset another register

    bl boot

DoPANIC:
    /* PANIC - no memory available
     * Blame it on the hardware guys
     */

    /* Maybe we should try to run in the int. ram anyway..
       */

Loop: b Loop


micro_delay:
    mov   r0, r0, lsr #1
    sub   r0, r0, #2
micro_delay_loop:
    subs  r0, r0, #1
    bgt   micro_delay_loop
    mov   pc, lr
```

Appendix F

Source files of the DTUsat-2 implementation

F.1 init.S

Listing F.1: Assembly language source code of the init procedure.

```
#define WDMOD 0xE0000000
#define WDTC 0xE0000004
#define WDFEED 0xE0000008
#define WDTV 0xE000000C
#define PLLCON 0xE01FC080
#define PLLCFG 0xE01FC084
#define PLLSTAT 0xE01FC088
#define PLLFEED 0xE01FC08C
#define VPBDIV 0xE01FC100
#define BCFG0 0xFFE00000
#define BCFG1 0xFFE00004
#define BCFG2 0xFFE00008
#define BCFG3 0xFFE0000C
#define CPSR_IRQ_DISABLE 0x80
#define CPSR_FIQ_DISABLE 0x40
#include "boot.h"
**********************************
/* Init file for the DTUsat-2 project */
/* by Esben Rugbjerg                          */
/* Some parts are copied from crt.s           */
*********************************

/* Stack Sizes */
.set  UND_STACK_SIZE, 0x00000004 /* stack for "undefined
    instruction" interrupts is 4 bytes  */
.set  ABT_STACK_SIZE, 0x00000004 /* stack for "abort"
    interrupts is 4 bytes                  */
.set  FIQ_STACK_SIZE, 0x00000004 /* stack for "FIQ" interrupts
      is 4 bytes       */
.set  IRQ_STACK_SIZE, 0X00000004 /* stack for "IRQ" normal
    interrupts is 4 bytes    */
.set  SVC_STACK_SIZE, 0x00000004 /* stack for "SVC" supervisor
     mode is 4 bytes      */

/* Standard definitions of Mode bits and Interrupt (I & F)
    flags in PSRs (program status registers) */
.set  MODE_USR, 0x10            /* Normal User Mode
          */
.set  MODE_FIQ, 0x11            /* FIQ Processing Fast
    Interrupts Mode       */
.set  MODE_IRQ, 0x12            /* IRQ Processing Standard
    Interrupts Mode      */
.set  MODE_SVC, 0x13            /* Supervisor Processing
    Software Interrupts Mode  */
.set  MODE_ABT, 0x17            /* Abort Processing memory
    Faults Mode          */
.set  MODE_UND, 0x1B            /* Undefined Processing
    Undefined Instructions Mode   */
.set  MODE_SYS, 0x1F            /* System Running Priviledged
     Operating System Tasks  Mode  */

.set  I_BIT, 0x80               /* when I bit is set, IRQ is
    disabled (program status registers) */
.set  F_BIT, 0x40               /* when F bit is set, FIQ is
    disabled (program status registers) */

.align 4
.text
.arm
_pll:
.long PLLCON
.long PLLCFG
.long PLLSTAT
.long PLLFEED
.set pll_con, 0x0
```

```
.set pll_cfg , 0x4
.set pll_stat , 0x8
.set pll_feed , 0xC
_vpbdiv:
.long VPBDIV
_bc:
.long BCFG0
.long BCFG1
.long BCFG2
.long BCFG3
.long 0x10000444 @configuration value of EMC bank 0
.long 0x20000400 @configuration value of EMC bank 1
.set b0, 0x0
.set b1, 0x4
.set b2, 0x8
.set b3, 0xC
.set conf_b0 , 0x10
.set conf_b1 , 0x14

.global Reset_Handler
.global _startup
.global loop



@ Exception Vectors
.section .vectors , "x"  /*unique section making it possible to
      place the exception vectors correctly.*/
.align 4
_startup: /*Entry point for code*/
   ldr      PC, Reset_Addr
   ldr      PC, Undef_Addr
   ldr      PC, SWI_Addr
   ldr      PC, PAbt_Addr
   ldr      PC, DAbt_Addr
   nop                              /* Reserved Vector (holds
         Philips ISP checksum) */
   ldr      PC, [PC,#-0xFF0]        /* see page 71 of "Insiders
          Guide to the Philips ARM7-Based Microcontrollers" by
         Trevor Martin  */
   ldr      PC, FIQ_Addr


Reset_Addr:       .word    Reset_Handler             /* defined
     below  */
Undef_Addr:       .word    UNDEF_Routine             /* defined
     below  */
```

```
SWI_Addr:       .word   SWI_Routine         /* defined
    below  */
PAbt_Addr:      .word   UNDEF_Routine       /* defined
    below */
DAbt_Addr:      .word   UNDEF_Routine       /* defined
    below */
IRQ_Addr:       .word   IRQ_Routine         /* defined
    below */
FIQ_Addr:       .word   FIQ_Routine         /* defined
    below */
                .word   0                   /* rounds the
                    vectors and ISR addresses to 64 bytes
                    total  */

.section .startup , "x"
.func   _startup /*only used when code is assembled with
    debugging turned on.*/
UNDEF_Routine:
SWI_Routine:
IRQ_Routine:
FIQ_Routine:
@All interrupts branches to reset handler. The reason is that
    if
@for some reason an interrupt occurs during the execution of
    the
@boot program it returns to the beginning and disable the
    interrupts.

@Set mode to supervisor mode to ensure that we are not in any
    interrupt mode
   ldr    r0, =_stack_end
   sub    r0, r0, #SVC_STACK_SIZE
   msr    CPSR_c, #MODE_SVC|I_BIT|F_BIT      /* supervisor mode
      */
   mov    sp, r0

@ Reset Handler
Reset_Handler:

/* Setup a stack for each mode - note that this only sets up a
     usable
 * stack for User mode. Also each mode is setup with
     interrupts initially disabled. */


   ldr    r0, =_stack_end
```

```
    msr   CPSR_c, #MODE_UND|I_BIT|F_BIT    /* Undefined Instruction Mode */
    mov   sp, r0
    sub   r0, r0, #UND_STACK_SIZE
    msr   CPSR_c, #MODE_ABT|I_BIT|F_BIT    /* Abort Mode */
    mov   sp, r0
    sub   r0, r0, #ABT_STACK_SIZE
    msr   CPSR_c, #MODE_FIQ|I_BIT|F_BIT    /* FIQ Mode */
    mov   sp, r0
    sub   r0, r0, #FIQ_STACK_SIZE
    msr   CPSR_c, #MODE_IRQ|I_BIT|F_BIT    /* IRQ Mode */
    mov   sp, r0
    sub   r0, r0, #IRQ_STACK_SIZE
    msr   CPSR_c, #MODE_SVC|I_BIT|F_BIT    /* Supervisor Mode */
    mov   sp, r0
@ User mode is not entered because we cannot return to a privileged
@ mode from user mode.
@     sub   r0, r0, #SVC_STACK_SIZE
@     msr   CPSR_c, #MODE_SYS|I_BIT|F_BIT    /* User Mode */
@     mov   sp, r0


    /* Enable watchdog. See lpc2200 user manual pp. 256 */
    mov r0, #WDTC @Load register with address of WDTC.
    mov r1, #0xFFFFFFFF @Prepare timer constant.
    str r1, [r0] @Store timer constant in WDTC.
    mov r0, #WDMOD @Load register with address of WDMOD.
    mov r1, #0x03 @Prepare enabling pattern.
    str r1, [r0] @Enable WD.
    mov r0, #WDFEED @Load register with address of WDFEED.
    mov r1, #0xAA @Prepare first feed pattern.
    str r1, [r0] @Start feed sequence of WD.
    mov r1, #0x55 @Prepare second feed pattern.
    str r1, [r0] @End feed sequence of WD.

pll: .long _pll
vpbdiv: .long _vpbdiv
bc: .long _bc


    /*configuration of the PLL */
    ldr r4, pll @set base address for constants
    ldr r0, [r4, #pll_cfg] @Load the address of PLLCFG into the register.
    mov r1, #0x23 @Prepare value of configuration register 0x23
        = 0b100011.
    mov r2, #0xAA @Prepare feed 1 value.
    mov r3, #0x55 @Prepare feed 2 value.
    str r1, [r0] @Store value in configuration register.
    ldr r0, [r4, #pll_feed] @load register the address of the
        feed register into the register.
    str r2, [r0] @write first part of feed sequence.
    str r3, [r0] @Write second part of feed sequence.
    /* enabling of the PLL */
    ldr r0, [r4, #pll_con] @Load register with the address of
        PLLCON.
    mov r1, #0x1 @load register with enabling value.
    str r1, [r0] @store enabling value register.
    ldr r0, [r4, #pll_feed] @load register the address of the
        feed register into the register.
    str r2, [r0] @write first part of feed sequence.
    str r3, [r0] @Write second part of feed sequence.
    ldr r0, [r4, #pll_stat] @Load the address of PLLSTAT into
        the register.
plllock:
    ldr r1, [r0] @get value from PLLSTAT.
    ands r1, r1, #(1<<10) @And value of register with a one on
        the 10th place.
    beq plllock @return to beginning of the loop if PLL has not
        locked yet.

    /* configuration of the VPB divider */
    ldr r0, vpbdiv @Load the address of the VPBDIV into the
        register.
    mov r1, #0x00 @The pclk is set to one fourth of the cclk.
    str r1, [r0] @The value is stored in VPBDIV.

    /* Configuration of the EMC */

    /* Configuration of bank 0 - the external FLASH */
    ldr r4, bc @load base address of bank configuration.
    ldr r0, [r4, #b0] @Load the address of the BCFG0 into the
        register.
    ldr r1, [r4, #conf_b0] @Writing value to control register.
    str r1, [r0] @store control value in control register

    /* Configuration of bank 1 - the external static RAM */
    ldr r0, [r4, #b1] @Load the address of the BCFG0 into the
        register.
    ldr r1, [r4, #conf_b1] @Writing value to control register.
    str r1, [r0] @store control value in control register

    /* Continue to memory test */
    b memoryTest


.endfunc

loop:
    b loop

.end
```

F.2 memTest.S

Listing F.2: Assembly language source code of the memory test done before initialisation of the C stack.

```
#include "boot.h"

.global memoryTest
.global cStack

MEMTable:
    .long RAM0_BASE          /* r4 */
    .long RAM0_LENGTH-1      /* r5 */
    .long RAM1_BASE          /* r6 */
    .long RAM1_LENGTH-1      /* r7 */
    .long STACK_SIZE      /* r8 */
    .long ONES            /* r9 */

PtMEMTable:
    .long MEMTable


memoryTest:

DoMemTest:  /* Modified version of function from DTU-1 */
    /* Load memory variables into registers */
    ldr r11, PtMEMTable @Load address of predefined values.
    ldmia r11, {r4-r9} @Load predefined values into registers.

    mov r0, r4      @Move addr. of start of RAM0 to r0.
    add r1, r4, r5 @r1 = RAM0_BASE + RAM0_LENGTH-1 find end of
        RAM0.
    mov r2, r1      @Move addr. of end of RAM0 to r2.
    mov r12, r1     @Move addr. of end of RAM0 to r12.
    add r12, r12, #1 @First address above RAM0.
    bl TestMemory /* Test ram0 */

wp1Test10:
   mov r0, r6       @move addr. of start of RAM1 to r0
   add r1, r0, r7 @r1 = RAM1_BASE + RAM1_LENGTH-1 find end of
        RAM1
   mov r2, r1       @move addr. of end of RAM1 to r2
   mov r12, r1      @move addr. of end of RAM1 to r12
   add r12, r12, #1 @first address above RAM1
   bl TestMemory /* Test ram1 */

   b RegBased/* No memory available */

TestMemory:
   @r0 : base address
   @r1 : highest valid address
   @r2 : pointer to current (byte) address
   @r3 : pattern value
   @r8 : stacksize
   @r9 : constant used during inversion of bits.
   @r10 : temporary value of address counter
   @r11 : value loaded from the memory which should be
        evaluated
   @r12 : end address

WriNormIni:
   mov r3, #0       @Initialise pattern.
   sub r3, r3, #1
   mov r2, r1       @Initialise byte counter to top address.
   add r2, r2, #1 @Increment byte counter by one, to
                    @to compensate for decrementing it
                    @in the beginning of the next loop.

WriNormFor:
   @Current address points to the highest byte tested.
   sub r2, r2, #1 @Decrement byte counter by one to point to
        next
                    @untested address.
   cmp r2, r0       @Compare current address to base address.
   movlt pc, r14    @return if current address is below base
        address.
   add r3, r3, #1 @Increment filter by one.
   strb r3, [r2]    @Store pattern.
   sub r10, r1, r2 @Tested area(r10) = top address(r1) -
        current address(r2).
   add r10, r10, #1 @Compensate for the fact that the top
        address
```

```
                    @also store data why it should be included
wp1Test11:          @in the length.
    cmp r10, r8     @Tested area(r10) compared to stacksize(r8).
    beq TestNormPat@Branch to test if stacksize is reached.
    b WriNormFor    @Branch back to start of loop if tested area
                    @is too small to host stack.

TestNormPat:
    mov r3, #0      @Initialise pattern.
    sub r3, r3, #1
    mov r2, r1      @Initialise byte counter to top address.
    add r2, r2, #1 @Increment byte counter by one, to
                    @to compensate for decrementing it
                    @in the beginning of the next loop.

TestNormFor:
    sub r2, r2, #1 @Decrement byte counter by one to point to
         next
                    @unverified address.
    sub r10, r1, r2 @Tested area(r10) = top address(r1) -
         current address(r2).
    add r10, r10, #1 @Compensate for the fact that the top
         address
                    @also store data why it should be included
wp1Test12:          @in the length.
wp1Test13:
    add r3, r3, #1 @Increment value of pattern.
    and r3, r3, #0x000000FF @Filter out 24 highest bits.
    ldrb r11, [r2]  @Load byte value stored at current address.
wp1Test21:
wp1Test23:
    cmp r11, r3      @Compare stored value to expected pattern.
    cmpeq r10, r8    @Compare tested area(r10) to stacksize(r8)
         if above
wp2Test13:           @comparison reported equality.
    beq WriInvIni   @Branch to test using inverted patterns if
         stacksize
                    @is reached.
    cmp r11, r3      @Compare stored value to expected pattern
         again to
                    @ensure correct CSPR state.
    beq TestNormFor@If test was passed, branch back to test
         next byte
    strb r3, [r2]   @Store pattern at byte address.
    add r2, r2, #1 @Increment byte counter by one.
    eor r3, r3, r9 @Invert pattern.
    and r3, r3, #0x000000FF @Filter out 24 highest bits.
    cmp r2, r12     @compare byte address to end of RAM
    subeq r2, r2, r8 @decrement byte counter by stacksize
    strb r3, [r2]   @Store pattern at byte address.
    addeq r2, r2, r8 @increment byte counter by stacksize
    eor r3, r3, r9 @Invert pattern.
    and r3, r3, #0x000000FF @Filter out 24 highest bits.
    sub r2, r2, #1 @Decrement byte counter by one.
    ldrb r11, [r2]  @Load byte value stored at current address.
wp2Test21:
    cmp r11, r3      @Compare stored value to expected pattern.
    cmpeq r10, r8    @Compare tested area(r10) to stacksize(r8)
        if above
                     @comparison reported equality.
    beq WriInvIni   @Branch to test using inverted patterns if
        stacksize
                     @is reached.
    cmp r11, r3     @Compare stored value to expected pattern
        again to
                     @ensure correct CSPR state.
    beq TestNormFor@If test was passed, branch back to test
        next byte
    and r2, r2, #0xFFFFFFFC @Align address by filtering out the
        two
                               @lowest bits.
    sub r2, r2, #1 @Decrement current address by one to find
        new top address
    mov r1, r2      @Set new top address as flawed address minus
        one.
    b WriNormIni    @Restart test from new top address.

WriInvIni:
    mov r3, #0      @Initialise pattern.
    sub r3, r3, #1
    mov r2, r1      @Initialise byte counter to top address.
    add r2, r2, #1 @Increment byte counter by one, to
                    @to compensate for decrementing it
                    @in the beginning of the next loop.

WriInvFor:
    @Current address points to the highest byte tested.
    sub r2, r2, #1 @Decrement byte counter by one to point to
        next
                    @untested address.
    add r3, r3, #1 @Increment filter by one.
    eor r3, r3, r9 @Invert pattern.
    strb r3, [r2]  @Store pattern.
    eor r3, r3, r9 @Invert pattern again to ensure that it will
```

```
                    @be incremented correct.
   sub r10, r1, r2  @Tested area(r10) = top address(r1) -
       current address(r2).
   add r10, r10, #1 @Compensate for the fact that the top
       address
                    @also store data why it should be included
wp1Test14:          @in the length.
   cmp r10, r8      @Tested area(r10) compared to stacksize(r8).
   beq TestInvPat   @Branch to test if stacksize is reached.
   b WriInvFor      @Branch back to start of loop if tested area
                    @is too small to host stack.

TestInvPat:
   mov r3, #0       @Initialise pattern.
   sub r3, r3, #1
   eor r3, r3, r9   @Invert pattern as part of
                    @initialisation.
   mov r2, r1       @Initialise byte counter to top address.
   add r2, r2, #1   @Increment byte counter by one, to
                    @to compensate for decrementing it
                    @in the beginning of the next loop.

TestInvFor:
   sub r2, r2, #1   @Decrement byte counter by one to point to
       next
                    @unverified address.
   sub r10, r1, r2  @Tested area(r10) = top address(r1) -
       current address(r2).
   add r10, r10, #1 @Compensate for the fact that the top
       address
wp1Test15:          @also store data why it should be included
wp1Test16:          @in the length.
   eor r3, r3, r9   @Invert pattern again to ensure
                    @that it will be incremented correct.
   add r3, r3, #1   @Increment value of pattern.
   eor r3, r3, r9   @Invert pattern.
   and r3, r3, #0x000000FF @Filter out 24 highest bits.
   ldrb r11, [r2]   @Load byte value stored at current address.
wp1Test22:
wp1Test24:
   cmp r11, r3      @Compare stored value to expected pattern.
   cmpeq r10, r8    @Compare tested area(r10) to stacksize(r8)
       if above
wp2Test16:          @comparison reported equality.
   beq SetupCstack@Branch to C stack initialisation if
       stacksize
                    @is reached.
```

```
    cmp r11, r3     @Compare stored value to expected pattern
        again to
                    @ensure correct CSPR state.
    beq TestInvFor @If test was passed, branch back to test
        next byte
    strb r3, [r2]  @Store pattern at byte address.
    add r2, r2, #1 @Increment byte counter by one.
    eor r3, r3, r9 @Invert pattern.
    and r3, r3, #0x000000FF @Filter out 24 highest bits.
    cmp r2, r12     @compare byte address to end of RAM
    subeq r2, r2, r8 @decrement byte counter by stacksize
    strb r3, [r2]  @Store pattern at byte address.
    addeq r2, r2, r8 @increment byte counter by stacksize
    eor r3, r3, r9 @Invert pattern.
    and r3, r3, #0x000000FF @Filter out 24 highest bits.
    sub r2, r2, #1 @Decrement byte counter by one.
    ldrb r11, [r2] @Load byte value stored at current address.
wp2Test22:
    cmp r11, r3     @Compare stored value to expected pattern.
    cmpeq r10, r8   @Compare tested area(r10) to stacksize(r8)
        if above
                    @comparison reported equality.
    beq SetupCstack@Branch to C stack initialisation if
        stacksize
                    @is reached.
    cmp r11, r3      @Compare stored value to expected pattern
        again to
                    @ensure correct CSPR state.
    beq TestInvFor@If test was passed, branch back to test next
         byte
    and r2, r2, #0xFFFFFFFC @Align address by filtering out the
         two
                                    @lowest bits.
    sub r2, r2, #1 @Decrement current address by one to find
        new top address
    mov r1, r2      @Set new top address as flawed address minus
         one.
    b WriNormIni    @Restart test from new top address.


SetupCstack:
    b cStack  @Branch to C stack initialisation.

RegBased:
    b RegBased      @Infinite loop to use for test.
```

F.3 cStack.S

Listing F.3: Assembly language source code of the initialisation of the C stack.

```
****************************
/* C stack initialisation in    */
/* boot of DTUsat-2             */
/* by Esben Rugbjerg            */
****************************
#include "boot.h"
#define STACK_SIZE 0x1000

.global boot
.global cStack

cStack:
/*Input:                        */
/* r0: base address of RAM area */
/* r1: highest valid address of */
/* stack area.                  */
/*Output:                       */
/* r0: Stack pointer            */
/* r1: Stack Limit              */


and r1, r1, #0xFFFFFFFC @Align Address.
mov sp, r1 @move address for stack pointer to correct
    register.
sub sl, sp, #STACK_SIZE @calculate stack limit (sl) and places
            @ in correct register
mov r0, sp @Copy stack pointer to r0.
mov r1, sl @Copy stack limit to r1.

b boot @ Branch to C code.
```

F.4 boot.h

Listing F.4: Header file of the boot functions.

```
/*
 *
  ==============================================================================

 *
 *        Filename:  boot.h
 *
 *     Description:  Header file for the boot program.
 *
 *         Version:  1.0
 *         Created:  19/02/07 15:16:50 CET
 *        Revision:  none
 *        Compiler:  gcc
 *
 *          Author:  Esben Rugbjerg (),
 *         Company:  Denmark's Technical University
 *
 * =====================================================================================
 */

#define RAM0_BASE                   0x40000000
#define RAM0_LENGTH                 0x3FE0
#define RAM1_BASE                   0x00003000
#define RAM1_LENGTH                 0x00004000
#define ONES                        0xFFFFFFFF

/* Memory required by c-stack */
#define STACK_SIZE                  0x1000

#define IOSET   0xE0028004
#define IODIR   0xE0028008
#define IOCLR   0xE002800C
#define LCD_LIGHT 0x00000400
```

F.5 boot.c

Listing F.5: C source code of the boot function.

```
/*
 * =====================================================================================
 *
 *        Filename:  boot.c
 *
 *     Description:  Functions to perform high level part of
 boot process on DTUsat-2.
 *
 *         Version:  1.0
 *         Created:  25/01/07 10:51:10 CET
 *        Revision:  none
 *        Compiler:  gcc
 *
 *          Author:  Esben Rugbjerg (),
 *         Company:  Denmark's Technical University
 *
 *   ========================================================================
 */

#include "sysInfo.h"
#include "intFlash.h"
#include "rtc.h"
#include "cMemTest.h"
#include "boot.h"
#include "../Failsafe/failsafe.h"
#include "intWDT.h"



boot(int stackPointer, int stackLimit) {
   int result = 0;

   //Used to collect information about FLASH write errors.
   int errorInfo;

   //Variables used by the SIB system. SIB Parameter Structure
   struct SPS sps= {
      (struct SIB *) BEG_ADDRESS, /*Beginning address of the
           array 'sibs'.*/
      0x0 , /*Pointer to the latest valid SIB or the default
           SIB. */
      -1, /*Index of theSib in the array 'sibs'. */
      MAX_NUM_OF_SIBs, /*number of SIBs in array 'sibs'. */
      0x0, /*Pointer to default SIB. */
      DEFAULT_SIB /* temporary version of the SIB used during
           alteration of values in the SIB */
   };
   struct SPS *theSps = &sps;
   result = initDefaultSib(theSps);

   (*((volatile unsigned long *) IODIR )) |= LCD_LIGHT;
   (*((volatile unsigned long *) IOSET )) |= LCD_LIGHT; //
        light on

   if (findSIB(theSps) != 0)
      /*writeLog("Couldn't find any sib. Use default")*/;

   if (testTheSib(theSps->theSib) != 0) {
        /*TODO: writeLog("sib contains invalid data")*/;
        failsafe(theSps->theSib);
    }

    //If first start
    if (theSps->theSib->launchBit > 0) {
        //TODO: Set hold Flag to COMMpic
        result = launchSilence(errorInfo,theSps->idxOfTheSib);
        //TODO: Remove hold flag from COMMpic
    }

    //Ensure that voltage level is high enough
    //to start satellite in nominal mode.
    //if ( TODO: testPower() != 00) {
    // failsafe(theSps->theSib);
    //}

    //Test boot counter
    if (theSps->theSib->bootCounter == 0)
        failsafe(theSps->theSib);

    //Decrement boot counter
    result = decretBootC(theSps,(int *)&errorInfo);
    if (result != 0)
        failsafe(theSps->theSib);

    if (findSIB(theSps) != 0)
        failsafe(theSps->theSib);

    if (testTheSib(theSps->theSib) != 0)
        failsafe(theSps->theSib);

    //Test RAM completely for memory errors.
    if (stackLimit < RAM1_BASE) {
        result = memTestC((datum *) RAM0_BASE, (stackLimit -
            RAM0_BASE));
        if (result != 0)
            return result;
        result = memTestC((datum *) RAM1_BASE, RAM1_LENGTH);
        if (result != 0)
            return result;
    }
    else
        failsafe(theSps->theSib);

    //Check checksum of OS before start
    if(crcCompute((unsigned char *)theSps->theSib->eCosBeg, (
        theSps->theSib->eCosEnd - theSps->theSib->eCosBeg)) ==
        theSps->theSib->eCosCheck );
    //start nominal mode.
    else
    failsafe(theSps->theSib);
while(1);

}

int launchSilence(struct SPS * theSps, int * errorInfo) {
    int i = 0;
    int result = 0;
    int sibBeg = -1;
    unsigned char data[INT_MIN_WR_SIZE];

    /* if it impossible to initialise a temporary version of
        the SIB => start FS */
    if(initTempSIB(&(theSps->tempSib), theSps->theSib) != 0);
        //start FS
        failsafe(theSps->theSib);

    /*startup RTC. If it fails => start FS */
    if (simpleInitRtc() != 0)
    //start FS
    failsafe(theSps->theSib);

    /*Loop which counts the minutes */
    for (i = (theSps->theSib->launchBit)-1; i >= 0; i--) {
        if (waitMinRtc() !=0)
            //Start FS
            failsafe(theSps->theSib);
        else
            theSps->tempSib.launchBit = i;

        kickWDT();
        prepDataArraySIB(data, &theSps->tempSib, (theSps->
            idxOfTheSib +1)*sizeof(struct SIB));

        //Find start position of SIB.
        if(theSps->idxOfTheSib != -1){ /*if SIB is not the
            default. */
        sibBeg = (theSps->idxOfTheSib + 1) * sizeof(struct SIB);
        //If the SIB overflows the size of the sector:
        //Place it in the beginning of the sector.
        if(sibBeg > ((MAX_NUM_OF_SIBs - 1)*sizeof(struct SIB)))
            sibBeg = 0;
        }
        else //If SIB is the default, start at the beginning
        //of the sector.
        sibBeg = 0;

        result = prepDataArraySIB(data,&theSps->tempSib,(sibBeg
            % INT_MIN_WR_SIZE));
        if (result != 0)
            return result;

        result = writeDataArrayToFLASH(\
                (unsigned char *) ((INT_FLASH_BEG + sibBeg)-(
                    sibBeg % INT_MIN_WR_SIZE)),\
                data, INT_MIN_WR_SIZE, errorInfo);
        if(result != 0)
            return result;
        //TODO: Kick WD
    }
return 0;
}
```

F.6 sysInfo.h

Listing F.6: Header file for the System Information Block (SIB) functions.

```
/*
 *
   ======================================================================

 *
 *        Filename:  sysInfo.h
 *
 *     Description:  Header file for the data structures and
    functions associated with the
 *                   System Information Block (SIB).
 *
 *         Version:  1.0
 *         Created:  25/01/07 17:35:18 CET
 *        Revision:  none
 *        Compiler:  gcc
 *
 *          Author:  Esben Rugbjerg (),
 *         Company:  DTU
 *
 *
   ======================================================================
 */
#ifndef __sysInfo_h
#define __sysInfo_h

#define MAGIC_NUM 0xFEEDBEEF
#define MAX_BOOTS 0x5
#define BEG_ADDRESS 0x00030000
#define END_ADDRESS 0x00031FFF
#define MAX_NUM_OF_SIBs ((END_ADDRESS - BEG_ADDRESS)+1)/sizeof
    (struct SIB)
#define DEFAULT_SIB {MAGIC_NUM,0x00000000,0x00000000,0
    x00000000,0x00000000,0x00000000,0x00000000,0x4B50E5B1}

/* The System Information Block structure  */
struct SIB {
   int  magicNum;     /*Magic number: 0xFEEDBEEF */
   int launchBit;    /* initial 15. Decreased by 1 each minute.
        0 the 15 minutes is over */
   int bootCounter; /*Count the number of boot attempts =
        MAX_BOOTS - boot attempts */
   unsigned long  eCosBeg;     /*32bit beginning address of
        area containing eCos */
   unsigned long  eCosEnd;     /*32bit end address of area
        containing eCos */
   unsigned long  eCosCheck;   /*32bit CRC checksum of the
        area containing eCos */
   unsigned long  eCosP;       /*pointer to execution
        beginning of eCos */
   unsigned long checksum;     /*32bit CRC checksum of the SIB
         minus the checksum itself*/
};

/*The structure used to handle information about the SIB
    system in the boot and: */
/*FS software: SIB Parameter Structure (SPS*/
struct SPS {
   struct SIB * sibs; /*Pointer to array of SIBs in FLASH. */
   struct SIB /*@null@*/ * theSib; /*Pointer to the latest
        valid SIB or the default SIB. */
   int idxOfTheSib; /*Index of theSib in sibs or RAM*/
   int arrayLength; /*Number of SIBs in array 'sibs'. */
   struct SIB * defaultSib; /*Pointer to default SIB. */
   struct SIB tempSib; /*Structure containing temporary values
         of SIB */
};

/*Function to find the location of the most recent SIB.
```

```
 * Sets the 'theSIB' to point at it, and the value of
      idxOFtheSib.  */
int findSIB(struct SPS *sps);

/*Function to decrement bootCounter  and save new SIB. Returns
      0 for success. */
int decretBootC(struct SPS *sps, int * errorInfo);

/*Function to test the validity of theSib by checking the
      checksum */
int testTheSib(struct SIB *theSib);


//Function to prepare the 512 bytes data array that should be
      written to
//FLASH.
//Returns 0 on success and DATA_ARRAY_OVERFLOW (22) on failure
      .
//Failure would be that the SIB continues past the 512th byte.
int prepDataArraySIB(unsigned char data[],struct SIB *sib,
      unsigned int begSIB);

//Function to initialise temporary SIB with correct values.
int initTempSIB(struct SIB * TempSib,struct SIB * theSib);

//Function which return the pointer of the default SIB
int initDefaultSib(struct SPS * sps);

//Store sib given by argument on the next empty place in the
      array.
//Handles every thing about the FLASH.
int storeSib(struct SIB *sib, struct SPS *sps, int * errorInfo
      );

#endif //__sysInfo_h
```

F.7 sysInfo.c

Listing F.7: C source code of the System Information Block (SIB) handling functions.

```
/*
 *
      ==================================================================


 *
 *         Filename:  sysInfo.c
 *
 *     Description:  Functions to read and manipulate the
      System Information Block used
 *                   in the DTUsat-2 boot and FS software.
 *
 *         Version:  1.0
 *         Created:  26/01/07 15:50:59 CET
 *        Revision:  none
 *        Compiler:  gcc
 *
 *          Author:  Esben Rugbjerg (),
 *         Company:  Denmark's Technical University
 *
 *
      ==========================================================================
 */
#include "sysInfo.h"
#include "intFlash.h"
#include "crc.h"

/* The default configuration of the SIB */
struct SIB defaultSib = DEFAULT_SIB;

/*Function to find the location of the most recent SIB
 * Sets the 'theSIB' to point at it, and the value of
     idxOFtheSib.  */
int findSIB(struct SPS *sps) {
   int index = 0;
   for(index = sps->arrayLength; index > 0; ){
      if(sps->sibs[--index].magicNum == MAGIC_NUM){
      sps->idxOfTheSib = index;
      sps->theSib = &(sps->sibs[index]);
      index = 0;
      }
   }
   if(sps->idxOfTheSib >= 0)
      return 0;
   else
      return 1;
}

/*Function to test the validity of theSib by checking the
     checksum */
int testTheSib(struct SIB *theSib){

   //Compute checksum of theSib
```

```
    unsigned int candidate = crcCompute((unsigned char *)
        theSib, sizeof(struct SIB)-4);

    //Compare computed checksum to stored checksum
    if(candidate == theSib->checksum)
        return 0;
    else
        return 1;
}

/*Function to decrement bootCounter  and save new SIB. Returns
     0 for success. */
int decretBootC(struct SPS *sps, int * errorInfo) {

    unsigned char data[INT_MIN_WR_SIZE];
    int result = 0;
    unsigned int sibBeg = -1;

    result = initTempSIB( (struct SIB *) &(sps->tempSib), (
        struct SIB *) &(sps->theSib) );
    if(result != 0)
        //Start FS
        return result;

    // forbered data []

    //Find start position of SIB.
    if(sps->idxOfTheSib != -1){ /*if SIB is not the default. */
        sibBeg = (unsigned int) ((sps->idxOfTheSib + 1) * sizeof
            (struct SIB));
        //If the SIB overflows the size of the sector:
        //Place it in the beginning of the sector.
        if(sibBeg > (unsigned int)((MAX_NUM_OF_SIBs - 1)*sizeof(
            struct SIB)))
            sibBeg = 0;
    }
    else //If SIB is the default, start at the beginning
        //of the sector.
        sibBeg = 0;

    result = prepDataArraySIB(data,&(sps->tempSib),(sibBeg %
        INT_MIN_WR_SIZE));
    if (result != 0)
        return result;

    result = writeDataArrayToFLASH(\
```

```c
            (unsigned char *) ((INT_FLASH_BEG + sibBeg)-(sibBeg %
                INT_MIN_WR_SIZE)),\
            data, INT_MIN_WR_SIZE, errorInfo);
    if(result != 0)
        return result;

    return 0;
}

//Function to prepare the 512 bytes data array that should be
    written to
//FLASH.
//Returns 0 on success and 1 on failure.
//Failure would be that the SIB continues past the 512th byte.
int prepDataArraySIB(unsigned char data[],struct SIB *sib,
    unsigned int begSIB) {

    int i = 0;
    int j = 0;
    for(i = 0; i < INT_MIN_WR_SIZE; i++) {
        if( i < begSIB) {
            data[i] = 0x00;
        }
        else if(i == begSIB) {
            for(j = 0; j < sizeof(struct SIB);j++, i++) {
                if(i == INT_MIN_WR_SIZE)
                    return DATA_ARRAY_OVERFLOW;
                data[i]  = ((unsigned char *) sib)[j];
            }
            if(i == INT_MIN_WR_SIZE)
                return DATA_ARRAY_OVERFLOW;
            data[i] = 0xFF;
        }
        else {
          data[i] = 0xFF;
        }
    }
    return 0;
}

//Function to initialise temporary SIB with correct values.
    Checksum should
//be recalculated when a field is changed.

int initTempSIB(struct SIB *tempSib, struct SIB * theSib) {
    tempSib->magicNum = theSib->magicNum;
    tempSib->launchBit = theSib->launchBit;
```

```

   if(theSib->bootCounter == 0)
      tempSib->bootCounter = 0;
   else
      tempSib->bootCounter = (theSib->bootCounter) - 1;

   tempSib->eCosBeg = theSib->eCosBeg;
   tempSib->eCosEnd = theSib->eCosEnd;
   tempSib->eCosCheck = theSib->eCosCheck;
   tempSib->eCosP = theSib->eCosP;
   tempSib->checksum = crcCompute((unsigned char *) &tempSib,
       sizeof(struct SIB)-4);
}

int initDefaultSib(struct SPS *sps) {
   sps->theSib = (struct SIB *) &defaultSib;
return 0;
}

//Store sib given by argument on the next empty place in the
    array.
//Handles every thing about the FLASH.
int storeSib(struct SIB *sib, struct SPS *sps, int * errorInfo
    ) {

   unsigned char data[INT_MIN_WR_SIZE];
   int result = 0;
   unsigned int sibBeg = -1;

   if(sps->idxOfTheSib != -1){ /*if SIB is not the default. */
      sibBeg = (unsigned int) ((sps->idxOfTheSib + 1) * sizeof
          (struct SIB));
      //If the SIB overflows the size of the sector:
      //Place it in the beginning of the sector.
      if(sibBeg > (unsigned int)((MAX_NUM_OF_SIBs - 1)*sizeof(
          struct SIB)))
         sibBeg = 0;
   }
   else //If SIB is the default, start at the beginning
      //of the sector.
      sibBeg = 0;

   result = prepDataArraySIB(data,sib,(sibBeg %
       INT_MIN_WR_SIZE));
   if (result != 0)
      return result;

    result = writeDataArrayToFLASH(\
            (unsigned char *) ((INT_FLASH_BEG + sibBeg)-(sibBeg %
                INT_MIN_WR_SIZE)),\
            data, INT_MIN_WR_SIZE, errorInfo);
    if(result != 0) {
        int zeros[] = {0,0,0,0,0,0,0,0};
        result = prepDataArraySIB(data,(struct SIB *) zeros,(
            sibBeg % INT_MIN_WR_SIZE));
        if (result != 0)
            return result;
        result = writeDataArrayToFLASH(\
            (unsigned char *) ((INT_FLASH_BEG + sibBeg)-(sibBeg %
                INT_MIN_WR_SIZE)),\
            data, INT_MIN_WR_SIZE, errorInfo);
    }

    return 0;

}
```

F.8 intFlash.h

Listing F.8: C source code of the FLASH handling functions for the internal FLASH.

```
/*
 * =====================================================================

 *
 *        Filename:  intFlash.h
 *
 *     Description:  Header file for the FLASH driver of the
       internal FLASH of the
 *                   CPU chip on the DTUsat-2
 *
 *         Version:  1.0
 *         Created:  08/02/07 17:34:47 CET
 *        Revision:  none
 *        Compiler:  gcc
 *
 *          Author:  Esben Rugbjerg (),
 *         Company:  Denmark's Technical University
 *
 *
 =====================================================================
```

```c
 */


//Clock frequency of system clock in KHz
#define SYS_CLOCK_FREQ 10000

#define INT_FLASH_BEG 0x20000000
#define INT_FLASH_END 0x2003FFFF
#define INT_FLASH_SIZE ((INT_FLASH_END - INT_FLASH_BEG) +1)
#define INT_SECTOR_SIZE_NORM 0x2000
#define INT_SECTOR_SIZE_LARGE 0x10000
#define INT_L_SECTOR_BEG 8
#define INT_L_SECTOR_END 9
#define INT_NUM_OF_SECTORS 18
#define INT_TOTAL_ROOM_IN_FLASH (16 * INT_SECTOR_SIZE_NORM) + (2 * INT_SECTOR_SIZE_LARGE)
#define INT_MIN_WR_SIZE 0x200 //512 bytes
#define IAP_LOCATION 0x7ffffff1

//Command numbers of IAP functions
#define IAP_PREP_SECTORS 50
#define IAP_WR_RAM_TO_FLASH 51
#define IAP_ERASE_SECTORS 52
#define IAP_BLANK_CHECK_SECTORS 53
#define IAP_READ_PART_ID 54
#define IAP_READ_BOOT_CODE_VERSION 55
#define IAP_COMPARE 56

//Error codes from IAP function
#define CMD_SUCCESS 0
#define INVALID_COMMAND 1
#define SRC_ADDR_ERROR 2
#define DST_ADDR_ERROR 3
#define SRC_ADDR_NOT_MAPPED 4
#define DST_ADDR_NOT_MAPPED 5
#define COUNT_ERROR 6
#define INVALID_SECTOR 7
#define SECTOR_NOT_BLANK 8
#define SECTOR_NOT_PREPARED_FOR_WRITE_OPERATION 9
#define COMPARE_ERROR 10
#define BUSY 11

//Error codes returned by software functions.
#define INVALID_SECTOR_NUMBER 20
#define AREA_NOT_BLANK 21
#define DATA_ARRAY_OVERFLOW 22
#define SECTOR_OVERFLOW 23
```

```
#define FLASH_OVERFLOW 24

typedef void (*IAP) (unsigned long [], unsigned long []);

//Function to test if an area of the FLASH is blank i.e.
    contains nothing
//but ones.
//Returns 0 for success and 1 for failure.
int BlankTestArea(unsigned char *start, int numOfChars);


//Function to prepare a 512 bytes data array that should be
    written to
//FLASH.
//Returns 0 on success and DATA_ARRAY_OVERFLOW (22) on failure
    .
int prepDataArray(unsigned char data[], unsigned char input[],
     int begPos, int numOfBytes);

//Function to write data to FLASH. Takes care of preparation
    of the sector
//Returns 0 on success and some int different from zero in
    case of failure.
int writeDataArrayToFLASH(unsigned char * dest, unsigned char
    * data, unsigned long numOfBytes, int * ErrorInfo);

//Function to write sectors in the FLASH. This function takes
    care
//of erasing and writing the sector.
//Returns 0 on success and other values on errors. These
    values are
//defined above.
int writeSector(unsigned char padArray[], int begByte, int
    endByte, int secNum, int * errorInfo);

//Function to write binary image to the internal FLASH. This
    function
//takes of all necessary steps including erasing the sectors
    needed.
int writeImageFromRAM(unsigned char data[], int numOfBytes,
    int startSec, int * errorInfo);

//Function to erase individual sectors.
int eraseSector(int secNum);

//Wrapper function for the IAP 'Prepare sector(s) for write
    operation'-function.
```

```
int IAPprepSectors(int startSec, int endSec);

//Wrapper function for the IAP 'Copy RAM to FLASH'-function.
int IAPcopyRAMtoFLASH(unsigned char * dest, unsigned char *
    source, int numOfBytes, int clckFrq);

//Wrapper function for the IAP 'Erase sector(s)'-function.
int IAPeraseSectors(int startSec, int endSec, int clckFrq);

//Wrapper function for the IAP 'Blanck check sector(s)'-
    function.
int IAPblankChkSectors(int startSec, int endSec);

//Wrapper function for the IAP 'Read Part ID'-function.
int IAPreadPartID(int * partID);

//Wrapper function for the IAP 'Read boot code version'-
    function.
int IAPrdBootCodeVer(int * version);

//Wrapper function for the IAP 'Compare'-function.
int IAPcompare(int * dest, int * src, int numOfBytes, int *
    errorLocation);
```

F.9 intFlash.c

Listing F.9: C source code of the FLASH handling functions for the internal FLASH.

```
/*
 * ==========================================================================

 *
 *       Filename:  intFlash.c
 *
 *    Description:  Driver for the internal FLASH of the CPU
       chip LPC2294. Contains
 *                  special functions to handle System
       Information Blocks of the
 *                  DTUsat-2.
 *
 *        Version:  1.0
 *        Created:  08/02/07 17:32:38 CET
 *       Revision:  none
 *       Compiler:  gcc
 *
 *          Author:  Esben Rugbjerg (),
 *         Company:  Denmark's Technical University
 *
 *    ==================================================================================
 */

#include "intFlash.h"

  //Declarations used by the IAP functions.
  unsigned long command[5]; //Array containing commands to
      the boot code
  unsigned long result[3];  //Array containing status and the
       result codes from the boot code
  IAP iapEntry =(IAP) IAP_LOCATION; //Ini of the pointer to
      the boot code function


//Function to test if an area of the FLASH is blank i.e.
    contains nothing
//but ones.
//Returns 0 for success and 21 for failure.
int BlankTestArea(unsigned char *start, int numOfChars) {
  int i = 0;
  for(i = 0; i <= numOfChars;i++) {
    if(*(start+i) != 0xFF)
      //Return 1 if anything ones is found.
      return AREA_NOT_BLANK;
  }
return 0;
}



//Function to prepare a 512 bytes data array that should be
    written to
//FLASH.
//Returns 0 on success and DATA_ARRAY_OVERFLOW (22) on failure
    .
int prepDataArray(unsigned char data[], unsigned char input[],
     int begPos, int numOfBytes) {
  int i = 0;
  int j = 0;
  for(i = 0; i < INT_MIN_WR_SIZE; i++) {
    if( i < begPos) {
      data[i] = 0x00;
```

```c
        }
        else if(i == begPos) {
            for(j = 0; j <= numOfBytes;j++, i++) {
                if(i == INT_MIN_WR_SIZE)
                    return DATA_ARRAY_OVERFLOW;
                data[i]  = input[j];
            }
            if(i == INT_MIN_WR_SIZE)
                return DATA_ARRAY_OVERFLOW;
            data[i] = 0xFF;
        }
        else {
          data[i] = 0xFF;
        }
    }
    return 0;
}

//Function to write data to FLASH. Takes care of preparation
     of the sector
//Returns 0 on success and some int different from zero in
     failure.
int writeDataArrayToFLASH(unsigned char * dest, unsigned char
     * data, unsigned long numOfBytes, int *errorInfo) {


    int sectorNum = 0;
    int sectorSize = 0;
    int result = 0;

    sectorNum = (((int) &dest) - INT_FLASH_BEG) /
         INT_SECTOR_SIZE_NORM ;

    //Test whether sector number is valid or not
    if(sectorNum > (INT_NUM_OF_SECTORS -1))
        return INVALID_SECTOR_NUMBER;

    //Set sector number to the correct value.
    if (sectorNum < INT_L_SECTOR_BEG);
    else if ( (sectorNum >= INT_L_SECTOR_BEG) && \
            (sectorNum < (INT_L_SECTOR_BEG + (
                INT_SECTOR_SIZE_LARGE/INT_SECTOR_SIZE_NORM))))
        sectorNum = INT_L_SECTOR_BEG;
    else if ( (sectorNum >= (INT_L_SECTOR_BEG + (
         INT_SECTOR_SIZE_LARGE/INT_SECTOR_SIZE_NORM)))\
            && (sectorNum < (INT_L_SECTOR_BEG + (
                INT_SECTOR_SIZE_LARGE/INT_SECTOR_SIZE_NORM)*2)))
        sectorNum = INT_L_SECTOR_END;
    else
        sectorNum = sectorNum - (2*(INT_SECTOR_SIZE_LARGE/
            INT_SECTOR_SIZE_NORM)+2);

    //Is the sector 8 KB or 64 KB ?
    if ( (sectorNum == INT_L_SECTOR_BEG) || (sectorNum ==
        INT_L_SECTOR_END))
        sectorSize = INT_SECTOR_SIZE_LARGE;
    else
        sectorSize = INT_SECTOR_SIZE_NORM;

    //Call IAP function through wrapper to prepare sector.
    result = IAPprepSectors(sectorNum, sectorNum);

    if(result != 0)
        return result;

    result = IAPcopyRAMtoFLASH(dest,data,numOfBytes,
        SYS_CLOCK_FREQ);

    if(result != 0)
        return result;

    return IAPcompare((int *) dest,(int *) data,numOfBytes,
        errorInfo);
}

//Function to write sectors in the FLASH. This function takes
    care
//of erasing and writing the sector.
//Returns 0 on success and other values on errors. These
    values are
//defined above.
int writeSector(unsigned char data[], int begByte, int endByte
    , int secNum, int *errorInfo) {

    unsigned char padArray[INT_MIN_WR_SIZE];
    int endSize = 0; //size of data chunk which needs to be
        padded to fit 512 bytes
    int dataIndex = 0; // index in array in RAM
    int padIndex = 0; //index in 512 byte array where the end
        chunk of data are padded to fit 512 bytes
    int chunks = 0; //number of 512 bytes data chunks in
        selected part of RAM.
    int wrSize = 0; //variable containing number of bytes which
        should be written.
    int result = 0; //return value from low level functions.
    int secSize = 0; //size of sector
    int secBeg = 0; //Beginning address of sector.

    //Test blankness of sector.
    result = IAPblankChkSectors(secNum,secNum);
    //If not blank it is erased.
    if(result != 0) {
       //Prep sector
       result = IAPprepSectors(secNum,secNum);
       if(result != 0)
          return result;
       //Erase sector
       result = IAPeraseSectors(secNum,secNum,SYS_CLOCK_FREQ);
       if(result != 0)
          return result;
       //Test blankness of sector
       result = IAPblankChkSectors(secNum,secNum);
       if(result != 0)
          return result;
    }

    //Is the sector 8 KB or 64 KB ?
    if ( (secNum == INT_L_SECTOR_BEG) || (secNum ==
        INT_L_SECTOR_END))
       secSize = INT_SECTOR_SIZE_LARGE;
    else
       secSize = INT_SECTOR_SIZE_NORM;

    // check if data is longer than sector.
    if ( (endByte - begByte) > secSize)
       return SECTOR_OVERFLOW;

    if (secNum <= INT_L_SECTOR_BEG)
       secBeg = INT_FLASH_BEG + (secNum * INT_SECTOR_SIZE_NORM)
           ;
    else if (secNum == INT_L_SECTOR_END)
       secBeg = INT_FLASH_BEG + (8 * INT_SECTOR_SIZE_NORM) +
           INT_SECTOR_SIZE_LARGE;
    else
       secBeg = INT_FLASH_BEG + ((secNum -2) *
           INT_SECTOR_SIZE_NORM) + 2 * INT_SECTOR_SIZE_LARGE;

    // Pad data with ones until a 512 byte limit is met.
    endSize = (endByte - begByte) % INT_MIN_WR_SIZE;
    if (endSize > 0) {
        for(dataIndex = (endByte - endSize), padIndex = 0;
            dataIndex <= endByte; dataIndex++, padIndex++)
            padArray[padIndex] = data[dataIndex];

        for(;padIndex < INT_MIN_WR_SIZE;padIndex++)
            padArray[padIndex] = 0xFF;
    }

    // Write data to sector
    chunks = (endByte - begByte) / INT_MIN_WR_SIZE;
    wrSize = chunks * INT_MIN_WR_SIZE;

    result = writeDataArrayToFLASH((unsigned char *) secBeg,(
        unsigned char *) &(data[begByte]), wrSize, errorInfo);

    if (result != 0)
        return result;
    if(endSize > 0)
    return writeDataArrayToFLASH((unsigned char *) (secBeg +
        wrSize), padArray, INT_MIN_WR_SIZE, errorInfo);

    return 0;
}

//Function to write binary image to the internal FLASH. This
    function
//takes of all necessary steps including erasing the sectors
    needed.
int writeImageFromRAM(unsigned char data[], int numOfBytes,
    int startSec, int * errorInfo) {

    int roomInFLASH = 0; //Room in FLASH from beginning of
        startSec to the end.
    int secNum = 0; //Section number of present section.
    int result = 0; //Storage of the return value of called
        functions.
    int begByte = 0; //Beginning position in data array for
        present write operation.
    int endByte = 0; //End position in data array for present
        write operation.
    int secSize = 0; //Size of section.

    //Calculate available room in FLASH
    if (startSec <= INT_L_SECTOR_BEG)
        roomInFLASH = INT_TOTAL_ROOM_IN_FLASH - (startSec *
            INT_SECTOR_SIZE_NORM);
    else if (startSec == INT_L_SECTOR_END)
```

```c
        roomInFLASH = INT_TOTAL_ROOM_IN_FLASH - ((
            INT_L_SECTOR_BEG * INT_SECTOR_SIZE_NORM) +
            INT_L_SECTOR_BEG);
    else if (startSec > INT_L_SECTOR_END)
        roomInFLASH = INT_TOTAL_ROOM_IN_FLASH - ((2 *
            INT_SECTOR_SIZE_LARGE) + ((startSec - 2) *
            INT_SECTOR_SIZE_NORM));

    //Return error if there is to little room for data
    if(numOfBytes > roomInFLASH)
        return FLASH_OVERFLOW;

    //Loop over bytes which needs to be written.
    for (secNum == startSec; endByte < numOfBytes ;secNum++ ) {

        //Is the sector 8 KB or 64 KB ?
        if ( (secNum == INT_L_SECTOR_BEG) || (secNum ==
            INT_L_SECTOR_END))
            secSize = INT_SECTOR_SIZE_LARGE;
        else
            secSize = INT_SECTOR_SIZE_NORM;

        //Calc end position candidate in data array for this
            sector.
        endByte = begByte + secSize - 1;

        //Correct endByte value if over flow occurred.
        if(endByte > (numOfBytes - 1))
            endByte = numOfBytes -1;

        //Call function to write sector
        result = writeSector(data, begByte, endByte, secNum,
            errorInfo);

        if(result != 0)
            return result;

        //Update begByte for next run.
        begByte = endByte + 1;


    }
    return 0;
}

//Function to erase individual sectors.
int eraseSector(int secNum) {

    int result = 0;

    result = IAPprepSectors(secNum,secNum);
    if(result != 0)
        return result;

    return IAPeraseSectors(secNum,secNum,SYS_CLOCK_FREQ);
}


//Wrapper function for the IAP 'Prepare sector(s) for write
    operation'-function.
int IAPprepSectors(int startSec, int endSec) {

    //Set up values in command array.
    command[0] = IAP_PREP_SECTORS;
    command[1] = startSec;
    command[2] = endSec;

    //Call function.
    iapEntry (command, result);

    //Return result.
    return result[0];
}

//Wrapper function for the IAP 'Copy RAM to FLASH'-function.
int IAPcopyRAMtoFLASH(unsigned char * dest, unsigned char *
    source, int numOfBytes, int clckFrq) {

    //Set values of command to IAP for the writing
    //operation
    command[0] = IAP_WR_RAM_TO_FLASH;
    command[1] = (unsigned long) dest;
    command[2] = (unsigned long) source;
    command[3] = numOfBytes;
    command[4] = clckFrq;

    iapEntry (command, result);

    return result[0];
}

//Wrapper function for the IAP 'Erase sector(s)'-function.
int IAPeraseSectors(int startSec, int endSec, int clckFrq) {

    //Set values in the command array.
    command[0] = IAP_ERASE_SECTORS;
    command[1] = startSec;
    command[2] = endSec;
    command[3] = clckFrq;

    iapEntry (command, result);

    return result[0];
}

//Wrapper function for the IAP 'Blanck check sector(s)'-
     function.
int IAPblankChkSectors(int startSec, int endSec) {

    //Set values in the command array.
    command[0] = IAP_BLANK_CHECK_SECTORS;
    command[1] = startSec;
    command[2] = endSec;

    iapEntry (command, result);

    return result[0];
}

//Wrapper function for the IAP 'Read Part ID'-function.
int IAPreadPartID(int * partID) {

    command[0] = IAP_READ_PART_ID;

    iapEntry (command,result);

    *partID = result[1];

    return result[0];
}

//Wrapper function for the IAP 'Read boot code version'-
     function.
int IAPrdBootCodeVer(int *version) {

    command[0] = IAP_READ_BOOT_CODE_VERSION;

    iapEntry (command,result);

    *version = result[1];

    return result[0];
}


//Wrapper function for the IAP 'Compare'-function.
int IAPcompare(int * dest, int * src, int numOfBytes, int *
     errorLocation) {

    command[0] = IAP_COMPARE;
    command[1] = (unsigned long) dest;
    command[2] = (unsigned long) src;
    command[3] = numOfBytes;

    *errorLocation = result[1];

    return result[0];
}
```

F.10 rtc.h

Listing F.10: C source code of the handling functions for real time clock.

```
/*
 *
      ==========================================================================


 *
 *        Filename:  rtc.h
 *
 *     Description:  Driver function for the Real Time Clock of
       the LPC2294 chip. Only the
 *                   most important functions are implemented.
 *
 *         Version:  1.0
 *         Created:  16/02/07 20:04:54 CET
 *        Revision:  none
 *        Compiler:  gcc
 *
 *          Author:  Esben Rugbjerg (),
 *         Company:  Denmark's Technical University
 *
 *
      ==========================================================================


 */

//Values needed to set up the prescaler.
```

```
//These are for the Olimex board using a 14.7456 Mhz crystal.
#define RTC_PREINT_VAL 0x1C1 // 449
#define RTC_PREFRAC_VAL 0

#define RTC_ILR 0xE0024000
#define RTC_CTC 0xE0024004
#define RTC_CCR 0xE0024008
#define RTC_CIIR 0xE002400C
#define RTC_AMR 0xE0024010
#define RTC_CTIME0 0xE0024014
#define RTC_CTIME1 0xE0024018
#define RTC_CTIME2 0xE002401C
#define RTC_SEC 0xE0024020
#define RTC_MIN 0xE0024024
#define RTC_HOUR 0xE0024028
#define RTC_DOM 0xE002402C
#define RTC_DOW 0xE0024030
#define RTC_DOY 0xE0024034
#define RTC_MONTH 0xE0024038
#define RTC_YEAR 0xE002403C
#define RTC_ALSEC 0xE0024060
#define RTC_ALMIN 0xE0024064
#define RTC_ALHOUR 0xE0024068
#define RTC_ALDOM 0xE002406C
#define RTC_ALDOW 0xE0024070
#define RTC_ALDOY 0xE0024074
#define RTC_ALMON 0xE0024078
#define RTC_ALYEAR 0xE002407C
#define RTC_PREINT 0xE0024080
#define RTC_PREFRAC 0xE0024084

//Function to initialize the Real Time Clock just to run.
//No correction of the time is done. Second are reset though.
int simpleInitRtc(void);

//Function returning when one minute has elapsed after it was
    called.
int waitMinRtc(void);
```

F.11 rtc.c

Listing F.11: C source code of the handling functions for real time clock.

```
/*
 * ==========================================================================
 *
 *         Filename:  rtc.c
 *
 *      Description:  Functions to control the Real Time Clock
    of the LPC2294 chip. Only
 *                    a minimum of funtions are implemented.
 *
 *          Version:  1.0
 *          Created:  17/02/07 17:04:24 CET
 *         Revision:  none
 *         Compiler:  gcc
 *
 *           Author:  Esben Rugbjerg (),
 *          Company:  Denmark's Technical University
 *
 *
    ==========================================================================
 */
#include "rtc.h"

//Function to initialize the Real Time Clock just to run.
//No correction of the time is done. Second are reset though.
int simpleInitRtc(void) {
   unsigned char * tempPtr;
   unsigned int * tempInt;

   //Ensure that the RTC is stopped.
   tempPtr = (unsigned char *) RTC_CCR;
   *tempPtr = (unsigned char) 0x02;

   //Disable interrupts
   tempPtr = (unsigned char *) RTC_AMR;
   *tempPtr = 0xFF;
   tempPtr = (unsigned char *) RTC_CIIR;
   *tempPtr = 0x00;

   //Reset seconds counter to zero.
   tempPtr = (unsigned char *) RTC_SEC;
   *tempPtr = 0x00;

   //Setup prescaler.
   tempInt = (unsigned int *) RTC_PREINT;
   *tempInt = RTC_PREINT_VAL;

   tempInt = (unsigned int *) RTC_PREFRAC;
   *tempInt = RTC_PREFRAC_VAL;
```

```

    //Start RTC.
    //(unsigned char *) CCR = 0x0;
}

//Function returning when one minute has elapsed after it was
     called.
int waitMinRtc(void) {
    //Value of minute counter when polling begins.
    unsigned char minValue = 0x00;
    //Value of minute counter while polling.
    unsigned char temp = 0x00;
    //Pointer used to point on the memory locations of the
         relevant
    //registers in the RTC.
    unsigned char *tempPtr ;

    //Reset seconds counter to zero.
    tempPtr =(unsigned char *) RTC_SEC;
    *tempPtr = 0x00;

    //record minutes value
    tempPtr = (unsigned char *)RTC_MIN;
    minValue = (*tempPtr) << 2;

    //Start RTC.
    tempPtr = (unsigned char *) RTC_CCR;
    *tempPtr = 0x00;


    tempPtr = (unsigned char *) RTC_MIN;
    temp = (*tempPtr) << 2;

    while(temp == minValue)
    temp = *tempPtr;

    //Stop the RTC.
    tempPtr = (unsigned char *) RTC_CCR;
    *tempPtr = 0x02;

return 0;
}
```

F.12 intWDT.h

Listing F.12: C source code of the handling functions for real time clock.

```
/*
 * =====================================================================================
 *
 *       Filename:  intWDT.h
 *
 *    Description:  Header file for the watch dog timer functions of the internal WDT of
 *                  the LPC2292 chip
 *
 *        Version:  1.0
 *        Created:  09/04/07 22:15:40 CEST
 *       Revision:  none
 *       Compiler:  gcc
 *
 *         Author:  Esben Rugbjerg (),
 *        Company:  Technical university of Denmark
 *
 * =====================================================================================
 */
#define INT_WDT_WDMOD 0xE0000000
#define INT_WDT_WDTC 0xE0000004
#define INT_WDT_WDFEED 0xE0000008
#define INT_WDT_WDTV 0xE000000C


/*Function to kick the WDT */
void kickWDT(void);
```

F.13 intWDT.c

Listing F.13: C source code of the handling functions for real time clock.

```
/*
 * =====================================================================================
 *
 *       Filename:  intWDT.c
 *
 *    Description:  Function for managing the internal watch dog timer of the LPC2292 chip
 *
 *        Version:  1.0
 *        Created:  09/04/07 22:21:58 CEST
 *       Revision:  none
 *       Compiler:  gcc
 *
 *         Author:  Esben Rugbjergf (),
 *        Company:  Technical University of Denmark.
 *
 * =====================================================================================
 */

#include "intWDT.h"


void kickWDT(void){
    unsigned char * tempPtr;
    unsigned int * tempInt;

    //Ensure that the RTC is stopped.
    tempPtr = (unsigned char *) INT_WDT_WDFEED;
    *tempPtr = (unsigned char) 0xAA;

    *tempPtr = (unsigned char) 0x55;
}
```

F.14 crc.h

Listing F.14: Header file for the CRC32 calculation functions.

```
/*
 * =====================================================================================
 *
 *       Filename:  crc.h
 *
 *    Description:  Header file for the CRC calculation
      functions. The functions are all
 *                  copied from "Programming Embedded Systems"
       by Michael Barr
 *                  (1999, O'Reilly). So are the definitions
      and prototypes in this file.
 *
 *        Version:  1.0
 *          Created:  30/01/07 14:33:16 CET
 *         Revision:  none
 *         Compiler:  gcc
 *
 *           Author:  Esben Rugbjerg (),
 *          Company:  Denmarks Technical University
 *
 * =====================================================================
 */


// CRC32 #define POLYNOMIAL 0x04C11DB7
#define POLYNOMIAL 0x04C11DB7
// CRC32 #define INITIAL_REMAINDER 0xFFFFFFFF
#define INITIAL_REMAINDER 0xFFFFFFFF
// CRC32 #define FINAL_XOR_VALUE 0xFFFFFFFF
#define FINAL_XOR_VALUE 0xFFFFFFFF

typedef unsigned long width;

#define WIDTH (8 * sizeof(width))
#define TOPBIT (1 << (WIDTH - 1))

#ifdef __GEN_CRC_TABLE
void crcInit(void);

/* Function to print the remainder table such that it can be
    included
 * in the source and compiled into the object file. */
int printRemainderTable(void);
#endif

width crcCompute(unsigned char *, unsigned int);
```

F.15 crc.c

Listing F.15: C source code of the CRC32 calculation functions.

```
/*
 * =====================================================================
 *
 *         Filename:  crc.c
 *
 *    Description:   C code file for the CRC calculation functions. The functions are all
 *                   copied from "Programming Embedded Systems" by Michael Barr
 *                   (1999, O'Reilly).
 *
 *        Version:  1.0
 *        Created:  30/01/07 14:49:16 CET
 *       Revision:  none
 *       Compiler:  gcc
 *
 *         Author:   (),
 *        Company:
 *
 * =====================================================================================
 */

#include "crc.h"

#ifdef __GEN_CRC_TABLE
#include <stdio.h>

width crcTable[256];
#else
width crcTable[256] = {0x0, 0x4c11db7, 0x9823b6e, 0xd4326d9, 0x130476dc, 0x17c56b6b, 0x1a864db2,
                0x1e475005, 0x2608edb8, 0x22c9f00f, 0x2f8ad6d6, 0x2b4bcb61, 0x350c9b64, 0x31cd86d3,
                0x3c8ea00a, 0x384fbdbd, 0x4c11db70, 0x48d0c6c7, 0x4593e01e, 0x4152fda9, 0x5f15adac,
                0x5bd4b01b, 0x569796c2, 0x52568b75, 0x6a1936c8, 0x6ed82b7f, 0x639b0da6, 0x675a1011,
                0x791d4014, 0x7ddc5da3, 0x709f7b7a, 0x745e66cd, 0x9823b6e0, 0x9ce2ab57, 0x91a18d8e,
                0x95609039, 0x8b27c03c, 0x8fe6dd8b, 0x82a5fb52, 0x8664e6e5, 0xbe2b5b58, 0xbaea46ef,
                0xb7a96036, 0xb3687d81, 0xad2f2d84, 0xa9ee3033, 0xa4ad16ea, 0xa06c0b5d, 0xd4326d90,
                0xd0f37027, 0xddb056fe, 0xd9714b49, 0xc7361b4c, 0xc3f706fb, 0xceb42022, 0xca753d95,
                0xf23a8028, 0xf6fb9d9f, 0xfbb8bb46, 0xff79a6f1, 0xe13ef6f4, 0xe5ffeb43, 0xe8bccd9a,
                0xec7dd02d, 0x34867077, 0x30476dc0, 0x3d044b19, 0x39c556ae, 0x278206ab, 0x23431b1c,
        0x2e003dc5 , 0x2ac12072 , 0x128e9dcf , 0x164f8078 ,
            0x1b0ca6a1 , 0x1fcdbb16 , 0x18aeb13 ,
        0x54bf6a4 , 0x808d07d , 0xcc9cdca , 0x7897ab07 , 0
           x7c56b6b0 , 0x71159069 , 0x75d48dde ,
        0x6b93dddb , 0x6f52c06c , 0x6211e6b5 , 0x66d0fb02 ,
            0x5e9f46bf , 0x5a5e5b08 , 0x571d7dd1 ,
        0x53dc6066 , 0x4d9b3063 , 0x495a2dd4 , 0x44190b0d ,
            0x40d816ba , 0xaca5c697 , 0xa864db20 ,
        0xa527fdf9 , 0xa1e6e04e , 0xbfa1b04b , 0xbb60adfc ,
            0xb6238b25 , 0xb2e29692 , 0x8aad2b2f ,
        0x8e6c3698 , 0x832f1041 , 0x87ee0df6 , 0x99a95df3 ,
            0x9d684044 , 0x902b669d , 0x94ea7b2a ,
        0xe0b41de7 , 0xe4750050 , 0xe9362689 , 0xedf73b3e ,
            0xf3b06b3b , 0xf771768c , 0xfa325055 ,
        0xfef34de2 , 0xc6bcf05f , 0xc27dede8 , 0xcf3ecb31 ,
            0xcbffd686 , 0xd5b88683 , 0xd1799b34 ,
        0xdc3abded , 0xd8fba05a , 0x690ce0ee , 0x6dcdfd59 ,
            0x608edb80 , 0x644fc637 , 0x7a089632 ,
        0x7ec98b85 , 0x738aad5c , 0x774bb0eb , 0x4f040d56 ,
            0x4bc510e1 , 0x46863638 , 0x42472b8f ,
        0x5c007b8a , 0x58c1663d , 0x558240e4 , 0x51435d53 ,
            0x251d3b9e , 0x21dc2629 , 0x2c9f00f0 ,
        0x285e1d47 , 0x36194d42 , 0x32d850f5 , 0x3f9b762c ,
            0x3b5a6b9b , 0x315d626 , 0x7d4cb91 ,
        0xa97ed48 , 0xe56f0ff , 0x1011a0fa , 0x14d0bd4d , 0
           x19939b94 , 0x1d528623 , 0xf12f560e ,
        0xf5ee4bb9 , 0xf8ad6d60 , 0xfc6c70d7 , 0xe22b20d2 ,
            0xe6ea3d65 , 0xeba91bbc , 0xef68060b ,
        0xd727bbb6 , 0xd3e6a601 , 0xdea580d8 , 0xda649d6f ,
            0xc423cd6a , 0xc0e2d0dd , 0xcda1f604 ,
        0xc960ebb3 , 0xbd3e8d7e , 0xb9ff90c9 , 0xb4bcb610 ,
            0xb07daba7 , 0xae3afba2 , 0xaafbe615 ,
        0xa7b8c0cc , 0xa379dd7b , 0x9b3660c6 , 0x9ff77d71 ,
            0x92b45ba8 , 0x9675461f , 0x8832161a ,
        0x8cf30bad , 0x81b02d74 , 0x857130c3 , 0x5d8a9099 ,
            0x594b8d2e , 0x5408abf7 , 0x50c9b640 ,
        0x4e8ee645 , 0x4a4ffbf2 , 0x470cdd2b , 0x43cdc09c ,
            0x7b827d21 , 0x7f436096 , 0x7200464f ,
        0x76c15bf8 , 0x68860bfd , 0x6c47164a , 0x61043093 ,
            0x65c52d24 , 0x119b4be9 , 0x155a565e ,
        0x18197087 , 0x1cd86d30 , 0x29f3d35 , 0x65e2082 , 0
           xb1d065b , 0xfdc1bec , 0x3793a651 ,
        0x3352bbe6 , 0x3e119d3f , 0x3ad08088 , 0x2497d08d ,
            0x2056cd3a , 0x2d15ebe3 , 0x29d4f654 ,
        0xc5a92679 , 0xc1683bce , 0xcc2b1d17 , 0xc8ea00a0 ,
            0xd6ad50a5 , 0xd26c4d12 , 0xdf2f6bcb ,
                0xdbee767c, 0xe3a1cbc1, 0xe760d676, 0xea23f0af,
                    0xeee2ed18, 0xf0a5bd1d, 0xf464a0aa,
                0xf9278673, 0xfde69bc4, 0x89b8fd09, 0x8d79e0be,
                    0x803ac667, 0x84fbdbd0, 0x9abc8bd5,
                0x9e7d9662, 0x933eb0bb, 0x97ffad0c, 0xafb010b1,
                    0xab710d06, 0xa6322bdf, 0xa2f33668,
                0xbcb4666d, 0xb8757bda, 0xb5365d03, 0xb1f740b4
                  };
#endif

#ifdef __GEN_CRC_TABLE
void crcInit(void) {

   width remainder;
   width dividend;
   int bit;

   /*
    * Perform binary long division, a bit at a time.
    */
   for(dividend = 0; dividend < 256; dividend++) {

      /*
       * Initialize the remainder.
       */
      remainder = dividend << (WIDTH - 8);

      /*
       * Shift and XOR with th polynomial.
       */
      for (bit = 0; bit < 8; bit++) {
         /*
          * Try to divide the current data bit.
          */
         if (remainder & TOPBIT) {
            remainder = (remainder << 1) ^ POLYNOMIAL;
         }
         else {
            remainder = remainder << 1;
         }
      }

      /*
       * Save the result in the table.
       */
      crcTable[dividend] = remainder;
   }
```

```
} /* crcInit() */
#endif

width crcCompute(unsigned char * message, unsigned int nBytes)
    {
   unsigned int offset;
   unsigned char byte;
   width remainder = INITIAL_REMAINDER;

   /*
    * Divide the message by the polynomial, a byte at a time.
    */
   for(offset = 0; offset < nBytes; offset++) {
      byte = (remainder >> (WIDTH - 8)) ^ message[offset];
      remainder = crcTable[byte] ^ (remainder << 8);
   }

   /*
    * The final remainder is the CRC result.
    */
   return (remainder ^ FINAL_XOR_VALUE);
} /* crcCompute() */

#ifdef _GEN_CRC_TABLE
/* Function to print the remainder table such that it can be
     included
 * in the source and compiled into the object file. */
int printRemainderTable(void){
   printf("crcTable[256] = {0x%x", crcTable[0]);

   int i;
   for(i = 1; i < 256; i++)
      printf(", 0x%x",crcTable[i]);

   printf("}");
}
#endif
```

F.16 cMemTest.h

Listing F.16: Header file for the memory test functions.

```
/*
 *
     ================================================================
 *
 *        Filename:  cMemTest.h
 *
 *     Description:  Header file for memory test function
       implemented in C. The functions
 *                   are copied from Programming Embedded
       Systems in C and C++ by
 *                   Michael Barr. The source code is given on
       page 66 to 73. The functions
 *                   are changed such that they return integers
       .
 *
 *         Version:  1.0
 *         Created:  01/02/07 13:38:43 CET
 *        Revision:  none
 *        Compiler:  gcc
 *
 *          Author:  Esben Rugbjerg (),
 *         Company:  Denmark's Technical University.
 *
 *
       =====================================================================================

 */

typedef unsigned char datum; /* Set the data bus width to 8
     bits */

//Test data bus by using a 'walking one' on a single address.
//Returns 0 on success and the 'or'ed' result of 0xDEAD0000
     and the pattern which failed
//if an error is detected.
int memTestDataBus(volatile datum * );

//Return 0 on success and the address of the defective memory
     location
//if an error is detected.
datum * memTestAddressBus(volatile datum *, unsigned long );

//Return 0 on success and the address of the defective memory
     location
//if an error is detected.
datum * memTestDevice(volatile datum *, unsigned long );

//Wrapper function which collects all memory test functions.
     The
//number of bytes tested should be more than 16.
int memTestC(datum * begAddr, unsigned long numOfBytes);
```

F.17 cMemTest.c

Listing F.17: C source code of the memory test functions.

```
/*
 * ==========================================================================
 *
 *       Filename:  cMemTest.c
 *
 *    Description:  Source code for memory test function implemented in C. The functions
 *                  are copied from Programming Embedded Systems in C and C++ by
 *                  Michael Barr. The source code is given on page 66 to 73.
 *
 *        Version:  1.0
 *        Created:  01/02/07 13:44:57 CET
 *       Revision:  none
 *       Compiler:  gcc
 *
 *         Author:   (),
 *        Company:
 *
 * ==========================================================================
 */
#include "stdio.h"
#include "cMemTest.h"

int memTestDataBus(volatile datum * address) {

   datum pattern;

   /*
    * Perform a walking 1's test at the given address.
    */
   for (pattern = 1; pattern != 0; pattern <<=1) {
      /*
       * Write the test pattern.
       */
      *address = pattern;

      /*
       * Read it back (immediatly is okay for this test)
```

```
        */
        if (*address != pattern)
            return (int) (pattern || 0xDEAD0000 );
    }
    return 0;
}

datum * memTestAddressBus(volatile datum * baseAddress,
     unsigned long nBytes) {
    unsigned long addressMask = (nBytes -1);
    unsigned long offset;
    unsigned long testOffset;

    datum pattern = (datum) 0xAAAAAAAA;
    datum antipattern = (datum) 0x55555555;

    /*
     * Write the default pattern at each of the power-of-two
          offsets..
     */
    for (offset = sizeof(datum); (offset & addressMask) != 0;
         offset <<=1) {
        baseAddress[offset] = pattern;
    }
    /*
     * Check for address bits stuck high.
     */
    testOffset = 0;
    baseAddress[testOffset] = antipattern;

    for (offset = sizeof(datum); (offset & addressMask) != 0;
         offset <<= 1) {
        if (baseAddress[offset] != pattern)
            return ((datum *) &baseAddress[offset]);
    }

    baseAddress[testOffset] = pattern;

    /*
     * Check for address bits stuck low or shorted.
     */
    for (testOffset = sizeof(datum); (testOffset & addressMask)
          != 0; testOffset <<= 1) {
        baseAddress[testOffset] = antipattern;

        for (offset = sizeof(datum); (offset & addressMask) !=
             0; offset <<= 1){
```

```
            if((baseAddress[offset] != pattern) && (offset != testOffset))
                return ((datum *) &baseAddress[testOffset]);
        }
        baseAddress[testOffset] = pattern;
    }
    return NULL;
} /* memTestAddressBus() */

datum * memTestDevice(volatile datum * baseAddress, unsigned long nBytes) {
    unsigned long offset;
    unsigned long nWords = nBytes / sizeof(datum);

    datum pattern;
    datum antipattern;

    /*
     * fill memory with a known pattern.
     */
    for (pattern = 1, offset = 0; offset < nWords; pattern++, offset++) {
        baseAddress[offset] = pattern;
    }

    /*
     * Check each location and invert it for the second pass.
     */
    for (pattern = 1, offset = 0; offset < nWords; pattern++, offset++) {
        if (baseAddress[offset] != pattern)
            return ((datum *) &baseAddress[offset]);

        antipattern = ~pattern;
        baseAddress[offset] = antipattern;
    }

    /*
     * Check each location for the inverted pattern and zero it.
     */
    for (pattern = 1, offset = 0; offset < nWords; pattern++, offset++) {
        antipattern = ~pattern;
        if (baseAddress[offset] != antipattern)
            return ((datum *) &baseAddress[offset]);

        baseAddress[offset] = 0;
    }

    return (NULL);
} /* memTestDevice() */

//Wrapper function which collects all memory test functions.
    The
//number of bytes tested should be more than 16.
int memTestC(datum * begAddr, unsigned long numOfBytes){
    int result = 0;

    result = memTestDataBus(begAddr + 0xF);
    if(result != 0)
        return result;

    result = (int) memTestAddressBus(begAddr, numOfBytes);
    if (result != 0)
        return result;

    return (int) memTestDevice(begAddr, numOfBytes);
}
```

F.18 testBench.S

Listing F.18: Assembly language source code of the test harness used to test the memory test function implemented in assembler. The test harness is designed to be generic and could be used with any assembly program having a label called 'main' defined at program start and having it defined as 'global'.

```
/*
    ------------------------------------------------------------------------

*Exception vect ors
*------------------------------------------------------------------------

*/

.text
.arm
.global memoryTest

start :
/* Vectors (8 total ) */
b reset /* reset */
```

```
b loop /* undefined instruction */
b loop /* software interrupt */
b loop /* prefetch abort */
b loop /* data abort */
nop /* reserved for the bootloader checksum */
b loop /* IRQ */
b loop /* FIQ */

/*section which is automatically called as the first */
reset :

b memoryTest



.end
```

F.19 test01.gdb

```
echo This script is designed to test the memory test function \n
echo implemented in assembler. It should test the structural issues \n
echo of the function, as correct counter increment and such. \n

set logging file assMemTest.log
set logging on

echo set file. \n
file ../Boot/testBench.elf

echo Connect to target.  \n
target sim

echo load file \n
load ../Boot/testBench.elf

tbreak TestMemory
r

#Functions to test register value and print result.
define reg0
if $r0 == $arg0
printf "register r0 == 0x%x " , $r0
echo Test passed.\n
else
printf "register r0 == 0x%x ", $r0
```

```
echo Test failed.\n
end
end
define reg1
if $r1 == $arg0
printf "register r1 == 0x%x ", $r1
echo Test passed.\n
else
printf "register r1 == 0x%x ", $r1
echo Test failed.\n
end
end
define reg2
if $r2 == $arg0
printf "register r2 == 0x%x ", $r2
echo Test passed.\n
else
printf "register r2 == 0x%x ", $r2
echo Test failed.\n
end
end
define reg3
if $r3 == $arg0
printf "register r3 == 0x%x ", $r3
echo Test passed.\n
else
printf "register r3 == 0x%x ", $r3
echo Test failed.\n
end
end
define reg4
if $r4 == $arg0
printf "register r4 == 0x%x ", $r4
echo Test passed.\n
else
printf "register r4 == 0x%x ", $r4
echo Test failed.\n
end
end
define reg5
if $r5 == $arg0
printf "register r5 == 0x%x ", $r5
echo Test passed.\n
else
printf "register r5 == 0x%x ", $r5
echo Test failed.\n
end
```

```
end
define reg6
if $r6 == $arg0
printf "register r6 == 0x%x ", $r6
echo Test passed.\n
else
printf "register r6 == 0x%x ", $r6
echo Test failed.\n
end
end
define reg7
if $r7 == $arg0
printf "register r7 == 0x%x ", $r7
echo Test passed.\n
else
printf "register r7 == 0x%x ", $r7
echo Test failed.\n
end
end
define reg8
if $r8 == $arg0
printf "register r8 == 0x%x ", $r8
echo Test passed.\n
else
printf "register r8 == 0x%x ", $r8
echo Test failed.\n
end
end
define reg9
if $r9 == $arg0
printf "register r9 == 0x%x ", $r9
echo Test passed.\n
else
printf "register r9 == 0x%x ", $r9
echo Test failed.\n
end
end
define reg10
if $r10 == $arg0
printf "register r10 == 0x%x ", $r10
echo Test passed.\n
else
printf "register r10 == 0x%x ", $r10
echo Test failed.\n
end
end
define reg11
```

```
if $r11 == $arg0
printf "register r11 == 0x%x ", $r11
echo Test passed.\n
else
printf "register r11 == 0x%x ", $r11
echo Test failed.\n
end
end
define reg12
if $r12 == $arg0
printf "register r12 == 0x%x ", $r12
echo Test passed.\n
else
printf "register r12 == 0x%x ", $r12
echo Test failed.\n
end
end
define memByte
#Function to test value of memory location referenced as a byte.
#Location is type casted as char pointer and then the value
#is taken out by 'dereferencing' it by using the '*'-operator
#memByte has the syntax as the ARM 'ldr' instruction: <value> <address>
if *((char*) $arg1) == $arg0
printf "Memory addr. 0x%x == 0x%x ",  ((char *) $arg1), *((char *) $arg1)
echo Test passed. \n
else
printf "Memory addr. 0x%x == 0x%x ", ((char *) $arg1), *((char *) $arg1)
echo Test Failed. \n
end
end
define test1
#Test if values are correct initialised in the beginning
#of the function.
   echo *** Test 1 *** \n
   #Stop program if running.
kill
   #Set temporary breakpoint.
tbreak TestMemory
   #Start program.
r
   echo Test 1:
   #Test Base address of RAM0.
reg0 0
echo Test 1:
   #Test highest valid address of RAM0.
   reg1 0x1FFF
echo Test 1:
   #Test pointer to current byte address.
reg2 0x1FFF
echo Test 1:
   #Test Stack size
   reg8 0x400
end
define test2
echo *** Test 2 *** \n
kill
tbreak WriNormFor
   r
   echo Test 2:
   #Test that byte counter was incremented
   #previous to entering loop (write norm. loop).
   reg2 0x2000
echo Test 2:
   #Test correct initialisation of pattern.
   reg3 0xFFFFFFFF
end
define test3
echo *** Test 3 *** \n
   kill
tbreak TestNormFor
   r
echo Test 3:
   #Test that byte counter was incremented
   #previous to entering loop (test norm. loop).
   reg2 0x2000
echo Test 3:
   #Test correct initialisation of pattern.
   reg3 0xFFFFFFFF
end
define test4
echo *** Test 4 *** \n
kill
   tbreak WriInvFor
   r
echo Test 4:
   #Test that byte counter was incremented
   #previous to entering loop (write inv. loop).
   reg2 0x2000
echo Test 4:
   #Test correct initialisation of pattern.
   reg3 0xFFFFFFFF
end
define test5
echo *** Test 5 *** \n
```

```
kill
   tbreak TestInvFor
   r
echo Test 5:
   #Test that byte counter was incremented
   #previous to entering loop (test norm. loop).
reg2 0x2000
echo Test 5:
   #Test correct initialisation of (inverted) pattern.
   reg3 0x0
end
define test6
echo *** Test 6 *** \n
   kill
   tbreak TestMemory
   break WriNormFor
   r
   #Continue until 'WriNormFor' breakpoint.
   c
   #Continue and pass 'WriNormFor' breakpoint
   #once and stop at it when passing it again.
   c 1
echo Test 6:
   #Test that byte counter is decremented correctly.
   reg2 0x1fff
echo Test 6:
   #Test that pattern is incremented correctly.
   reg3 0x0
   #Remove breakpoint from system.
   clear WriNormFor
end
define test7
echo *** Test 7 *** \n
   kill
break TestNormFor
   r
   c 2
echo Test 7:
#Test that byte counter is decremented correctly.
   reg2 0x1ffe
echo Test 7:
#Test that pattern is incremented correctly.
   reg3 0x1
   clear TestNormFor
end
define test8
echo *** Test 8 *** \n
```

```
break WriInvFor
   r
   c 3
   echo Test 8:
#Test that byte counter is decremented correctly.
   reg2 0x1ffd
echo Test 8:
#Test that pattern is incremented correctly.
   reg3 0x2
clear WriInvFor
end
define test9
echo *** Test 9 *** \n
break TestInvFor
r
c 4
echo Test 9:
   #Test that byte counter is decremented correctly.
reg2 0x1ffc
echo Test 9:
#Test that pattern is incremented correctly.
   reg3 0xfc
clear TestInvFor
end
define test10
   #Test that the function returns if base address
   #is passed.
echo *** Test 10 *** \n
kill
tbreak TestMemory
break WriNormFor
   #Breakpoint placed where function returns to.
break wp1Test10
r
c
   #set byte counter to base address.
set $r2 = $r0
c
echo Test 10:
   #Test value of base address.
   reg0 0x0
echo Test 10:
   #Test value of byte counter.
   reg2 0xffffffff
echo Test 10:
   #Test value of pattern.
   reg3 0xffffffff
```

```
clear WriNormFor
clear wp1Test10
end
#Test 11 - 16 test that subfunctions call each other
#when stacksize is reached.
define test11
echo *** Test 11 *** \n
kill
   tbreak TestMemory
   #BP where tested length is changed.
break wp1Test11
break TestNormPat
r
c
   #Set length of tested area to stacksize.
set $r10 = $r8
c
echo Test 11:
   #Test byte counter.
   reg2 0x1fff
echo Test 11:
   #Test pattern.
   reg3 0x0
clear wp1Test11
clear TestNormPat
end
define test12
echo *** Test 12 *** \n
kill
   tbreak TestMemory
   #BP where tested length is changed.
break wp1Test12
break WriInvIni
r
c
#Set length of tested area to stacksize.
set $r10 = $r8
c
echo Test 12:
   #Test byte counter.
reg2 0x1fff
echo Test 12:
#Test pattern.
   reg3 0x0
clear wp1Test12
clear WriInvIni
end
```

```
define test13
echo *** Test 13 *** \n
kill
   tbreak TestMemory
   #BP where false values are introduced
break wp1Test13
   #BP where length of tested area is set
   #to stacksize and value from memory is
   #kept false.
break wp2Test13
break WriInvIni
r
c
set $r10 = 0x25
set $r11 = 0x25
c
set $r10 = $r8
set $r11 = 0x25
c
echo Test 13:
   reg2 0x1fff
echo Test 13:
   reg3 0x0
clear wp1Test13
clear WriInvIni
clear wp2Test13
end
define test14
echo *** Test 14 *** \n
kill
   tbreak TestMemory
break wp1Test14
break TestInvPat
r
c
set $r10 = $r8
c
echo Test 14:
   reg2 0x1fff
echo Test 14:
   reg3 0x0
clear wp1Test14
clear TestInvPat
end
define test15
echo *** Test 15 *** \n
kill
```

```
   tbreak TestMemory
break wp1Test15
break SetupCstack
r
c
set $r11 = $r3
set $r10 = $r8
c
echo Test 15:
   reg2 0x1fff
echo Test 15:
   reg3 0xff
clear wp1Test15
clear SetupCstack
end
define test16
echo *** Test 16 *** \n
kill
   tbreak TestMemory
break wp1Test16
break wp2Test16
break SetupCstack
r
c
set $r10 = 0x25
c
set $r10 = $r8
set $r11 = 0x25
c
echo Test 16:
   reg2 0x1fff
echo Test 16:
   reg3 0xff
clear wp1Test16
clear SetupCstack
clear wp2Test16
end
define test17
   #Test that correct patterns are written to
   #correct addresses.
echo *** Test 17 *** \n
kill
   tbreak TestMemory
   tbreak TestNormPat
r
c
   #Test that the function doesn't write above
   #its address limit and the pattern below.
echo Test 17:
memByte 0x0 0x2000
echo Test 17:
   memByte 0x0 0x1fff
echo Test 17:
memByte 0x1 0x1ffe
echo Test 17:
memByte 0x2 0x1ffd
   #Print first 24 bytes to inspect patterns.
echo Test 17: \n
   x /24xb 0x1ff0
end
define test18
   #Test that the pattern is repeated every 256 bytes.
   #Test three bytes before and three bytes after the
   #border between the two repetitions.
   echo *** 18 *** \n
kill
tbreak TestMemory
tbreak TestNormPat
r
c
echo Test 18:
memByte 0xfd 0x1f02
echo Test 18:
memByte 0xfe 0x1f01
echo Test 18:
memByte 0xff 0x1f00
echo Test 18:
memByte 0x0 0x1eff
echo Test 18:
memByte 0x01 0x1efe
   echo Test 18:
memByte 0x02 0x1efd
echo Test 18: \n
   x /40 0x1ee8
end
define test19
   #Test That correct patterns are written to
   #correct addresses when inverted patterns are written.
echo *** Test 19 *** \n
kill
   tbreak TestMemory
   tbreak TestInvPat
r
c
```

```
#Test that the function doesn't write above
   #its address limit and the pattern below.
echo Test 19:
memByte 0x0 0x2000
echo Test 19:
   memByte 0xff 0x1fff
echo Test 19:
memByte 0xfe 0x1ffe
echo Test 19:
memByte 0xfd 0x1ffd
   #Print first 24 bytes to inspect patterns.
echo Test 19: \n
   x /24xb 0x1ff0
end
define test20
   #Test that the pattern is repeated every 256 bytes
   #when inverted patterns are written.
   #Test three bytes before and three bytes after the
   #border between the two repetitions.
echo *** Test 20 *** \n
kill
tbreak TestMemory
tbreak TestInvPat
r
c
echo Test 20:
memByte 0x02 0x1f02
echo Test 20:
memByte 0x01 0x1f01
echo Test 20:
memByte 0x00 0x1f00
echo Test 20:
memByte 0xff 0x1eff
echo Test 20:
memByte 0xfe 0x1efe
   echo Test 20:
memByte 0xfd 0x1efd
echo Test 20: \n
   x /40 0x1ee8
end
define test21
#Test that memory test is restarted at next word
#address when a permanent fault is identified during
#test of normal pattern.
echo *** Test 21 *** \n
kill
tbreak WriNormIni
```

```
break wp1Test21
r
c
c 13
   set $r11 = 0x25
tbreak wp2Test21
c
set $r11 = 0x25
break WriNormFor
c
c 10
echo Test 21:
memByte 0x0d 0x1ff2
echo Test 21:
memByte 0x0e 0x1ff1
echo Test 21:
memByte 0x0f 0x1ff0
echo Test 21:
memByte 0x00 0x1fef
echo Test 21:
memByte 0x01 0x1fee
echo Test 21:
memByte 0x02 0x1fed
echo Test 21: \n
   x /40xb 0x1fe0
clear wp1Test21
clear WriNormFor
end
define test22
#Test that memory test is restarted at next word
#address when a permanent fault is identified during
#test of inverted pattern.
echo *** Test 22 *** \n
kill
tbreak WriNormIni
break wp1Test22
r
c
c 13
   set $r11 = 0x25
tbreak wp2Test22
c
set $r11 = 0x25
break WriNormFor
c
c 10
echo Test 22:
```

```
memByte 0xf2 0x1ff2
echo Test 22:
memByte 0xf1 0x1ff1
echo Test 22:
memByte 0xf0 0x1ff0
echo Test 22:
memByte 0x00 0x1fef
echo Test 22:
memByte 0x01 0x1fee
echo Test 22:
memByte 0x02 0x1fed
echo Test 22: \n
   x /40xb 0x1fe0
clear wp1Test22
clear WriNormFor
end
define test23
#Test that memory test is continued when a
#transient fault is identified during
#test of normal pattern.
echo *** Test 23 *** \n
kill
tbreak WriNormIni
break wp1Test23
r
c
c 13
   set $r11 = 0x25
c
c 10
echo Test 23:
memByte 0x0d 0x1ff2
echo Test 23:
memByte 0x0e 0x1ff1
echo Test 23:
memByte 0x0f 0x1ff0
echo Test 23:
memByte 0x10 0x1fef
echo Test 23:
memByte 0x11 0x1fee
echo Test 23:
memByte 0x12 0x1fed
echo Test 23: \n
   x /40xb 0x1fe0
clear wp1Test21
end
define test24
```

```
#Test that memory test is continued when a
#transient fault is identified during
#test of inverted pattern.
echo *** Test 24 *** \n
kill
tbreak WriNormIni
break wp1Test24
r
c
c 13
   set $r11 = 0x25
c
c 10
echo Test 24:
memByte 0xf2 0x1ff2
echo Test 24:
memByte 0xf1 0x1ff1
echo Test 24:
memByte 0xf0 0x1ff0
echo Test 24:
memByte 0xef 0x1fef
echo Test 24:
memByte 0xee 0x1fee
echo Test 24:
memByte 0xed 0x1fed
echo Test 24: \n
   x /40xb 0x1fe0
clear wp1Test24
end
echo Set breakpoints and run tests. \n
test1
test2
test3
test4
test5
test6
test7
test8
test9
test10
test11
test12
test13
test14
test15
test16
test17
```

```
test18
test19
test20
test21
test22
test23
test24
```

Appendix G

Test output

G.1 Output from test of memory test function

Appendix H

The timed models

H.1 OBC vs. COMM

H.1.1 Global declarations

Listing H.1: Global declaratoins of the OBC vs. COMM model.

```
  // Place global declarations here.

//state of individual ports on the modules
bool OBCout[4];
bool COMMin[4];

// used to simulate low power problems
//i.e. only if true enough power is available to the system
bool power = true;
// signals when WDT is enabled by OBC
bool enabOBCwdt = false;
// signals when WDT is enabled by COMM
bool enabCOMMwdt = false;

// channel to send OBC into fault-mode
chan OBCfault;

```

```
// channel to synchronise wires high when set high by OBC
chan OBCoutSyncH;

// channel to synchronise wires low when set low by OBC
chan OBCoutSyncL;

// channel to synchronise data wire low when set low by OBC
chan OBCdataL;

//channel to synchronise date wire high when set high by OBC
chan OBCdataH;


chan WireSyncH[4];
chan WireSyncL[4];



// channel to reset OBC-WDT's counter to start value i.e.
     kicking it.
chan kickOBC;

// channel to reset COMM-WDT's counter to start value i.e.
     kicking it
chan kickCOMM;

// channel to signal reset to OBC when WDT times out
chan WDTtoOBC;

// channel to signal reset to COMM-PIC when WDT times out
chan WDTtoCOMM;

// channel to synchronise OBC-WDT when OBC enables it
chan enableOBCwdt;

// channel to synchronise COMM-WDT when COMM-PIC enables it
chan enableCOMMwdt;

// period of OBC WDT
int OBCwdtPeriod = 600;

// period of COMM WDT
int COMMwdtPeriod = 145;

//time needed to send safe beacon by COMM PIC
const int SafeBeaconPeriod = 140;

```

```
//initial hold period for the system, is 900 sec (15 min)
//const int holdPeriod = (15 * 60 * 1000);
const int holdPeriod = (15 * 60);
```

H.1.2 System declarations

Listing H.2: System declaratoins of the OBC vs. COMM model.

```
  // Place template instantiations here.
OBC = OBCtem(WDTtoOBC, OBCfault);
ComPic = ComPicTem(WDTtoCOMM);
wire0 = wireTem(0);
wire1 = wireTem(1);
wire2 = wireTem(2);
wireData = wireTem(3);
//EPS = EPStem();
WDoBC = WDTtem(OBCwdtPeriod, kickOBC, WDTtoOBC, enableOBCwdt,
    enabOBCwdt);
WDcomm = WDTtem(COMMwdtPeriod, kickCOMM, WDTtoCOMM,
    enableCOMMwdt, enabCOMMwdt);
WireCntrl = WireCntrlTem();
DataCntrl = DataCntrlTem();
//WDTobserver = WDTobsTem();


// List one or more processes to be composed into a system.
system OBC, ComPic, wire0, wire1, wire2, wireData, WDoBC,
    WDcomm, WireCntrl, DataCntrl;
```

H.1.3 Processes

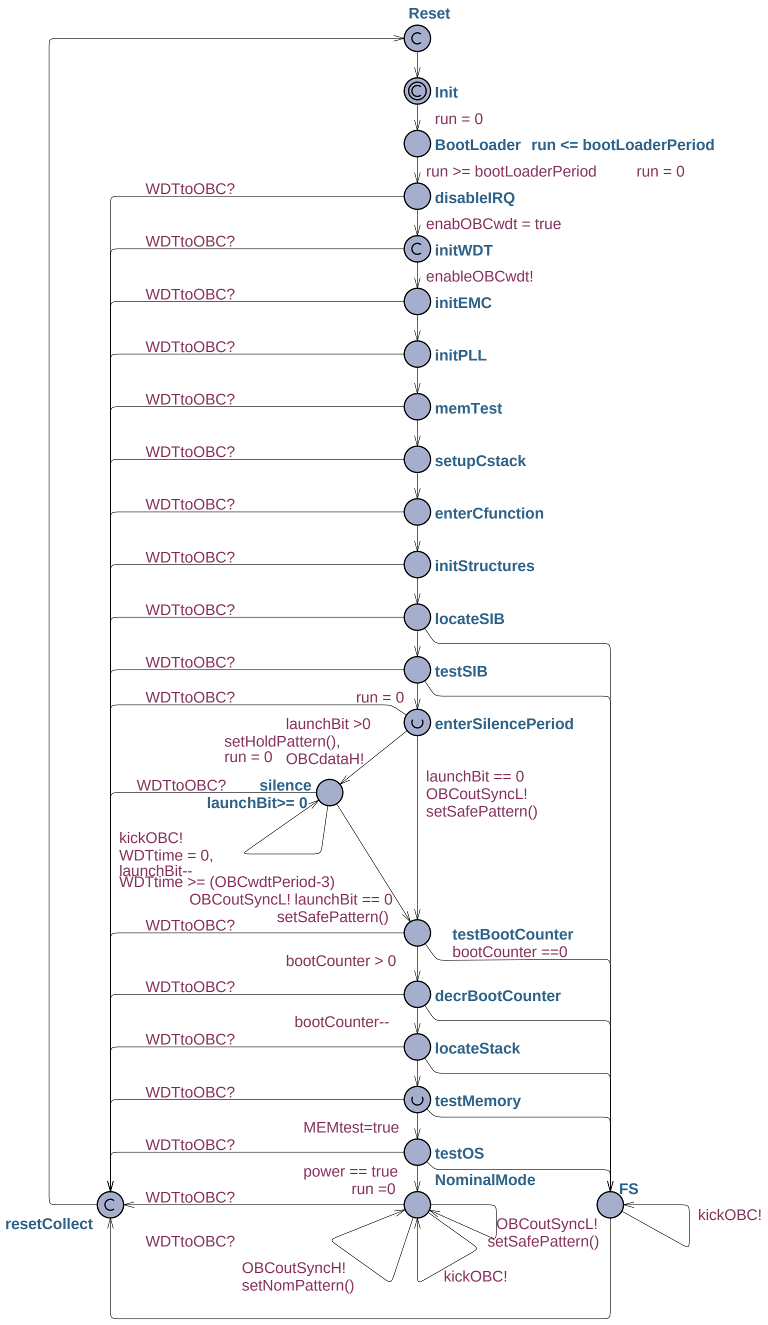


Figure H.1: The OBC process.

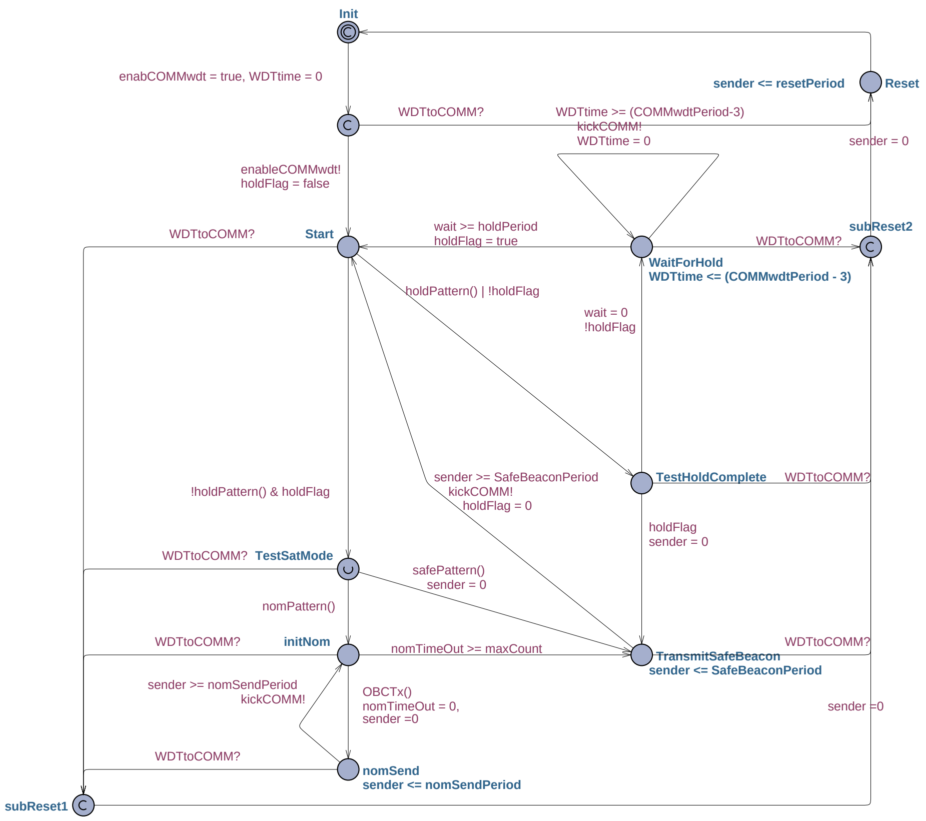
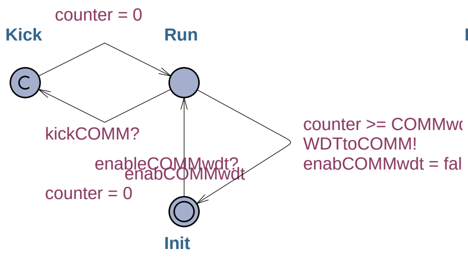
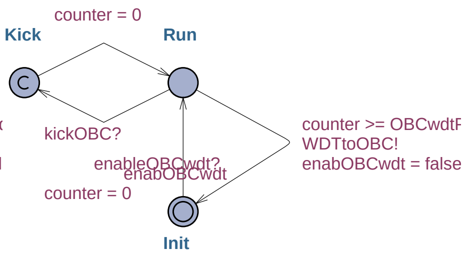


Figure H.2: The COMM process.

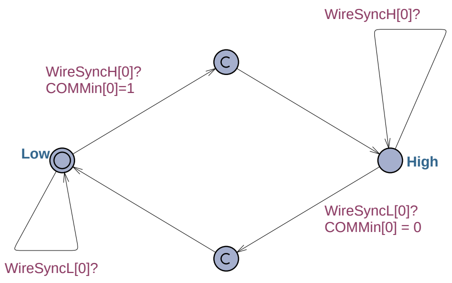


(a) The COMM WDT.

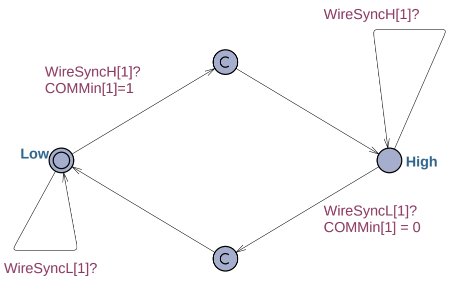


(b) The OBC WDT.

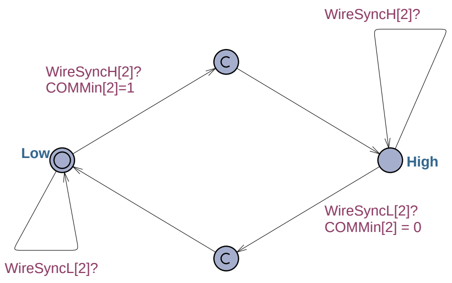
Figure H.3: The WDTs.



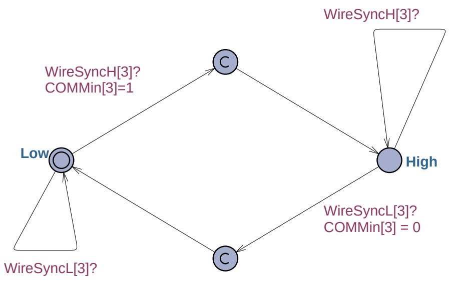
(a) The wire 0 process.



(b) The wire 1 process.



(c) The wire 2 process.



(d) The data wire process.

Figure H.4: The wire processes.

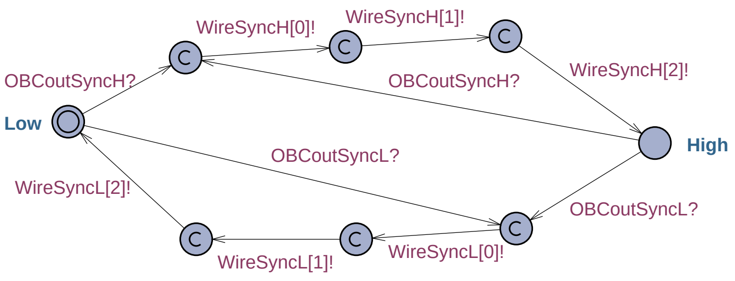


Figure H.5: The wire control process.

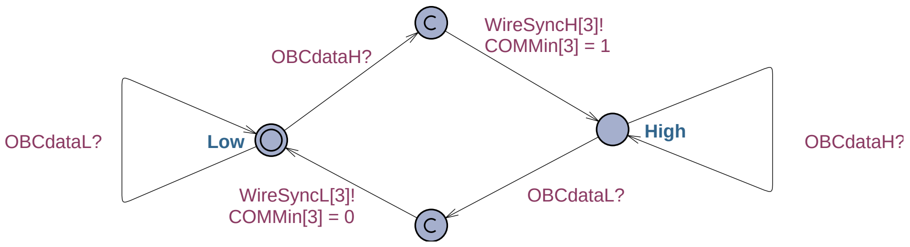


Figure H.6: The data control process.

H.2 Memory test of DTUsat-1

H.2.1 Global declarations

Listing H.3: Global declaratoins of the memory test model.

```
   // Place global declarations here.
int r0 = 1; //base
int r1 = 7; //highest valid address
int r2 = 7; //pointer to actual address
int r3 = 25; //tmp

const int r8 = 11; //pattern 1
const int r9 = 33; //pattern 2
const int r10 = 3; //stack size

//memory array
//first field contains the data and tells whether the cell has
      been damaged permantly or not:
//0 = not permanently damaged (but could have a transient
     fault) and 1 = damaged permanently
int memory[8][2] =
     {{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0},{0,0}};

chan branchWrite; // branch to start of Write
chan branchFail; //branch to start of Fail
```

H.2.2 System declarations

```
   // Place template instantiations here.
Write = WriteTem();
Fail = FailTem();
FaultInjection = FaultInjectionTem();

// List one or more processes to be composed into a system.
system Write, Fail, FaultInjection;
```

H.2.3 Processes

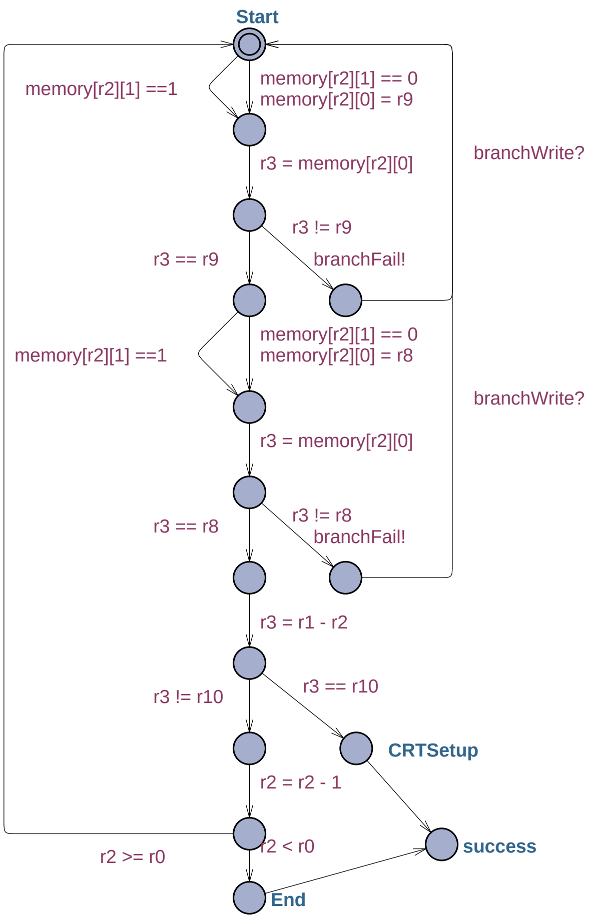


Figure H.7: The 'write'-function process

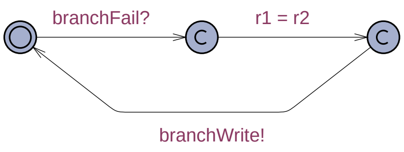


Figure H.8: The failure control process.

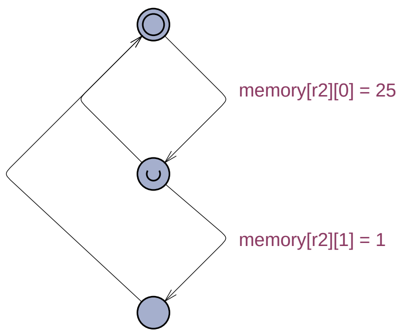


Figure H.9: The fault injection process.

Bibliography

[ALR04] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats: A taxonomy. In R Jacquart, editor, *Proceedings of the Building the Information Society*, pages 91–120. IFIP 18th World Computer Congress, Kluwer Academic Publishers, August 2004.

[ARM] Technical support faqs: Estimating stack size requirements. http://www.arm.com/support/faqdev/1444.html. Last visited 22th of March 2007.

[Bar99] Michael Barr. *Programming Embedded Systems*. O'Reilly and Associates, first edition, 1999.

[Bar00] Michael Barr. Software-based memory testing - if ever there was a piece of embedded software ripe for reuse it is the memory test. this article shows how to test for the most common memory problems with a set of three efficient, portable, public-domain memory test functions. *Embedded Systems Programming*, 13(7):28–40, 2000. http://embedded.com/2000/0007/0007feat1.htm, last visited December 13th 2006. The article has also been published as the section 'Memory Testing' of [Bar99, chp. 6].

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. On the web: http://www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf., 2004.

[BLA05] Blast: Berkeley lazy abstraction software verification tool. http://mtc.epfl.ch/software-tools/blast/. Last visited 22th of March 2007, 2005.

[esa00] C and c++ coding standards. Technical report, European Space Agency, 8-10, rue Mario-Nikis, 75738 PARIS CEDEX, France, 2000.

[Fur00] Steve Furber. *ARM System-On-Chip Architecture*. Addison-Wesley, 1. edition, 2000.

[Gan95] Jack G. Ganssle. Thanks for the memories. http://www.ganssle.com/articles/aramrom.htm, last visited December 13th 2006. Also published in Embedded Systems Programming, August 1995, 1995.

[Gan97] Jack G. Ganssle. Ram tests. http://www.ganssle.com/articles/ramtest.htm, last visited December 13th 2006. Also published in Embedded Systems Programming, October 1997, 1997.

[gdb06] Debugging with gdb. http://sourceware.org/gdb/current/onlinedocs/gdb.html#SEC_Top, last visited March 15th 2007., 2006.

[GL02] Georges Gonthier and Jean-Jacques Lévy. Software robustness engineering - robustness methodology survey. Technical report, INRIA Rocquencourt, 2002.

[HT05] Amy Hutputtanasin and Armen Toorian. CubeSat design specification (cds). Design specification 9, Aerospace Engineering Department, California Polytechnic State University, Aerospace Engineering Department, California Polytechnic State University,San Luis Obispo, CA 93401, 2005. http://cubesat.atl.calpoly.edu/media/Documents/Developers/CDS%20R9.pdf, last visited March 12th 2007.

[IHU99] Douglas Isbell, Mary Hardin, and Joan Underwood. Mars climate orbiter team finds likely cause of loss. News bulletin, september 1999. http://mars.jpl.nasa.gov/msp98/news/mco990930.html. Last visited 22th of March 2007.

[Int05] Intel(R), http://www.intel.com. *Intel(R) Advanced+ Boot Block Flash Memory (C3) -datasheet*, 2005.

[Jay06] Jayasooriah. Lpc2000 boot loader internals – a tutorial introduction. http://water.cse.unsw.edu.au/esdk/lpc2/boot-loader.html, last visited March 15th 2007., 2006.

[kis06] Kiss principle. http://en.wikipedia.org/wiki/KISS_principle. Last visited the 18th of March 2007., 2006.

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[Lyn05] James P. Lynch. Arm cross development with eclipse. as PDF on the net, 2005.

[MCO99] Nasa's mars climate orbiter believed to be lost. News bulletin, September 1999. http://mars.jpl.nasa.gov/msp98/news/mco990923.html. Last visited 22th of March 2007.

[oli06] Product description of the lpc-e2294. http://www.olimex.com/dev/lpc-e2294.html. Last visited 16th of March 2007., 2006.

[Phi03] Philips Semiconductors. *LPC2119/2129/2194/2292/2294 USER MANUAL*, 2003.

[PP02] Jan Storbank Pedersen and Steen Ulrik Palm. Software robustness engineering, design and coding constraints. Technical report, ESTEC, INRIA, TERMA, 2002.

[sam04] 256kx16 bit high speed static ram(3.3v operating). operated at commercial and industrial temperature ranges. datasheet 4, SAMSUNG Electronics CO., LTD, 2004. The data sheet of the static RAM used on the Olimex development board. Type: K6R4016V1D-TC10.

[Sec] Secure Programming Group, University of Virginia, Department of Computer Science, http://www.splint.org/downloads/manual.pdf. *Splint Manual, Version 3.1.1-1, 5 June 2003.*

[spl] Secure programming lint. http://www.splint.org/. Last visited 22th of March 2007.

[stA07] Stackanalyzer — stack usage analysis. http://www.absint.com/stackanalyzer/. Last visited 22th of March 2007, 2007.

[UA] UPP and AAL. The uppaal model checker. http://www.uppaal.com. Developed by Department of Information Technology at Uppsala University (UPP) and the Department of Computer Science at Aalborg University (AAL).