Boot-strapping a satellitte
A midterm project at DTU

c991305 - Jakob Skriver
c990391 - Morten Proschowsky
c971403 - Morten F. Friisgaard

September 10, 2002
IMM

Preface

Boot-strapping a satellite is a midterm project written at DTU. The work was caryed out at the institute of Informatics and Mathematical Modeling (IMM). The professor associated to this project was Hans Henrik Løvengren (hhl@imm.dtu.dk). At the beginning of this project we were four persons to do a work of each 15 ECTS points on the subject *"Boot-strapping a Satellite.* Later on one decided to leave the project due to personal problems.

Many times during development of this software we have had questions concerning other groups of the DTUsat project. Every time a friendly person has answered our questions. We would therefore like to thank all the members of the DTUsat project for their helpfulness. We would also like to thank the staff at IMM for their help.

Morten F. Friisgaard
c971403@student.dtu.dk

Morten Proschowsky
c990391@student.dtu.dk

Jakob Skriver
c991305@student.dtu.dk

Richard Petersens Plads,
Building 321,
DK-2800 Kongens Lyngby,
DTU September 10, 2002

Contents

Chapter 1

Introduction

This rapport is made as a midterm project at the Technical University of Denmark (DTU). The project is a small part of the CubeSat project at DTU.

1.1 What is CubeSat?

The CubeSat concept is a collaborative effort between Japanese and United States Universities to build very small satellites, launch and operate them in space. This way, students can get hands-on experience in the life-cycle of a space project without the investment and risks associated with a full-scale mission.

The Technical University of Denmark and Aalborg University both have established teams to build the first two Danish student satellites. The project is run by students and if possible all parts of the project will be done by students. Each project-member contributes according to his or her area of interest and expertise. The work is organized in groups, each working on a specific part of the satellite.

1.2 The DTUsat mission

The following eight prioritized goals have been decided by the members of the DTUsat crew[4]:

1. That all groups learn something. This goal is already fulfilled.
2. To finish and document all the different modules, so others, e.g. future DTU satellites, can use them.
3. To make the satellite fly.
4. To receive a beacon signal, telling that the satellite is up and that something works in space.

5. To receive a smart-beacon-signal from the on-board computer that sends more information to earth.
6. To establish two way-communication with the satellite.
7. To obtain three dimensional attitude control
8. To deploy two specific payloads

The two payloads chosen to bring on the DTUsat mission are:

- An active pixel CCD camera
- An electrodynamic tether

To make this project possible it was necessary to do a modular division of the work. Several small work tasks makes it easier for many persons to work on the project at the same time. The DTUsat project was divided into the following tasks:

- Mechanical Design and Construction
- Power
- On-board Computer
- Software
  - Architecture
  - Boot-strap
  - Drivers
  - Ground segment
  - Protocol
- Communication
  - Antenna
  - Ground Station
  - Radio Hardware
- Attitude Control and Determination
- Payloads
  - Camera
  - Tether
- System Engineering

This document will function as the documentation for the boot-strap module witch is a subdivided module of the overall software module.

1.3 The problem: Boot-strapping a Satellite

The problem we were asked to solve at the DTUsat project was to make the boot-strap program.

> A boot-strap program is defined as the first set of instructions executed when the computer is powered on.

The task of the boot-strap program is to start up the computer, load the application software, and start it. Since it is no standard computer used on the satellite, the boot-strap program also differs a lot from normal boot-strap programs.

A boot-strap program on a satellite has to have a sort of failsafe mode, to protect the satellite from going into a deadlocked situation. From this failsafe mode it should be possible to upload new software, report the error status back to earth etc.

1.4 Requirements to the Boot-Strap program

Before the work on the boot-strap code began we were aware of the following requirements for the final boot-strap program. These requirements were:

- The boot-strap program must be without error, because it is impossible to correct errors from the earth when the satellite is in orbit.
- The boot-strap program must be able to set up the On board Computer, and start the application software.
- If something goes wrong during a boot, the boot-strap program should enter a failsafe mode and contact the ground station.
- The boot-strap program must in some case be able to detect and bypass hardware-errors.

1.5 Software to load

In the section on requirements it is mentioned that the boot-strap program must be able to load the "application software". Throughout this document we will use application software as defined:

> Application software = eCos + a number of on board software modules.

eCos is an operating system designed for embedded systems (see 3.3.6). The on board software is designed in different modules to run on top of eCos to benefit of the features implemented in eCos. The on board software is

designed in different modules, each module controls one single part of the satellite, the radio is controlled by a module, the camera is controlled by a module etc. Redefining the term application software has no effect on the boot-strap program.

1.6 The Contents of this document

This document is the documentation on the work we have carried out concerning the boot-strap program of the DTUsat satellite project. It is divided into the following chapters:

1. **Introduction:** A short introduction to the problem that has to be solved.
2. **The Environment:** describes the environment where the program has to be executed.
3. **Software Development:** A description of what theoretical models, and what software tools we have used during development.
4. **Design and Analysis:** A description of how the implementation should be done, and a discussion on why it should be done this way.
5. **Implementation:** A description on how the designs are implemented.
6. **Test:** The documentation of the tests carried out to verify the system.
7. **Conclusion:** The final conclusion on the boot-strap problem.
8. **Bibliography:** A list of the references used in this document.
9. **Appendix:** In the appendix we list all the informations we have found irrelevant to list in the rapport as well as all the source code.

Chapter 2

The Environment

Developing software to an embedded system such as a satellite is very different from normal software development. An embedded computer is designed to do one specific thing very good (i.e. control a satellite), whether as a standard PC is designed to be able to do everything at a acceptable level. The main difference in writing code for an embedded system and a standard PC is no operating system is provided, and therfore no interface to the hardware. This makes it important for the software developer to understand the hardware interface of the embedded system, since it is not possible to have an easy understandable interface with I/O-devices and plug and play drives for everything, at the same time as a very efficient real-time system. A real-time system is a computer system that has timing constraints. In other terms a real-time system is designed to make certain calculations within a deadline. Most embedded systems are real-time systems.

A conclusion to this must be, that an embedded system is a combination of computer hardware and software, designed to perform a specific function [2]. We will start by taking a closer look at the hardware used on the DTUsat.

2.1 On board Computer

The DTUsat project has chosen to use a on board computer (OBC) on the satellite. One of the advantages of having one main computer instead of having many small circuits consisting of micro-controllers and memory chips is that the developing time (and cost) is lower. Also communication between different circuits could be a huge problem. A disadvantage of having one OBC is that all parts rely on the same computer, meaning a fatal error in the OBC leads to a failure in the mission, where if each hardware module had their own intelligent circuit one error could only bring down one hardware module. The decision about what design and witch component to chose was taken by a group under the DTUsat project, they have made a report on the subject.[13]

2.1.1 Processor

A processor is needed to execute the on board programs. The chosen ARM AT91M40807 processor features the RISC instruction set, as well as a 32-bit core. Another feature of the processor is the hardware watch dog, which timeout can be set in the range from 1 ms to about 6 sec. running the processor at 10 MHz. The watch dog reboots the system, if the watch-dog timer is not reset with in timeout.

The processor has 32 programmable I/O-lines, but only 6 of these are dedicated general-purpose I/O lines. The rest of the I/O lines are multiplexed with other functions. The only hardware function we will discuss is the USART. USART means Universal Synchronous/Asynchronous Receiver Transmitter, which is a serial communications device. The USART on the satellite will be used for the connection to the radio. The USART can also be connected to a standard serial port witch allows a simulation of the radio communication. Since we have to make the program the processor should execute as the first thing, we need to know where the processor starts executing code from. Figure 2.1 shows the address scheme at boot-time. The

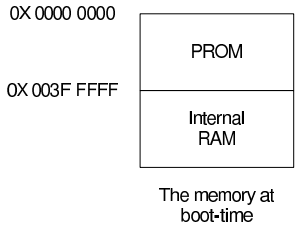

Figure 2.1: The memory map at boot time

processor starts executing code at address `0X 0000 0000`.

2.1.2 PROM

Since a corrupt boot-program would make it impossible ever to boot the on board computer, it is of very high importance that no corrupt data can occur at the place where the boot-program has to be stored. The PROM cannot be erased, therefore it is only possible to program it once, but this also assures the correctness of the data. The PROM (Programmable Read Only Memory) is the place, where the boot-program is stored. There is chosen a PROM of the size 64 Kb which leaves us with almost no requirements for a small footprint, even though we have chosen to make the size of our program as small as possible due to the chance of errors in a bigger program. In the development a flash (see 2.1.4) has been used because it is very inconvenient that the PROM only can be programmed once.

2.1.3 RAM

The OBC needs RAM in order to execute programs. On the DTUsat satellite static RAM (SRAM) is used since it is more simple to do operations on static RAM than on dynamic RAM (DRAM). The chosen ram is manufactured by Samsung and has the type K6T4016U3B.

Some ram have an EDAC (Error Detection And Correction) implemented. An EDAC can detect and correct bitflips in the ram (and in the flash). Since bitflips without doubt can occur in space the idea of implementing an EDAC circuit between the processor and the ram/flash has been considered. After some research the on board computer group reached the conclusion that this was impossible to do without redesigning the entire on board computer. Therefore no EDAC is implemented on the on board computer.

There is one 1 MB RAM module on the OBC, and the processor itself has 128 KB of internal RAM.

2.1.4 Flash

The flash memory on the on board computer is used to store the on board software, as well as collected data. The main difference between flash memory and usual RAM is no power is needed to keep the data in the flash. Compared to RAM, flash are fast to read (approx. 9 micro sec.), whether as it is slow to erase (approx. 760 milisec. for one block). Prior to writing the flash all the bits has to be set to one (erased), this can only be done by erasing one entire block at the time.

Since the satellite is expected to reboot frequently, a storage device is needed. Usual storage devices such as hard drives are not possible on a space mission due to the high power consumption as well as the little tolerance to movements. On the OBC the flash works as a substitute for a hard-drive. The flash is divided into 32 blocks. The first one is called a boot-block which again is subdivided into two blocks of 8 Kb, one block of 16 Kb and one of 32 Kb. The remaining 31 blocks are all 64 Kb big (see figure 2.1.4).

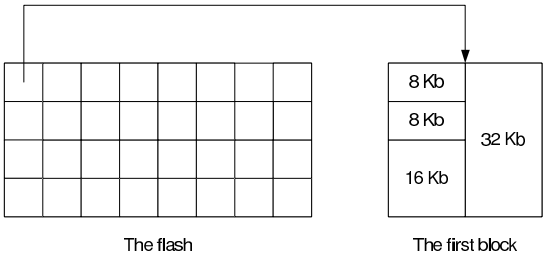

Figure 2.2: The devision of the Flash-RAM

We use a AMD AM29LV170D flash, which contains embedded algorithms

for writing and deleting data.

2.2 Space environment

The space environment is a very rough environment regardless of what the purpose of being there is. For hardware the special features of temperatures from -40^oC to 80^oC and hard radiation can be very hard to overcome. Further more there is a risk of collision with particles. The temperature and the radiation can be taken into account by doing special temperature test and radiation test. If the satellite collides with anything of reasonable size (compared to the satellite) the entire satellite will be destroyed.

2.2.1 Special Requirements for Space certified software

Since it is impossible to change components on the on board computer (see 2.1) on a satellite there are some special requirements the software has to fulfill.

First of all the software should be error free- no bugs at all. And is this possible? No of cause not, just take a look at the early Windows platform, which from time to time just crashes or stalls, and you then have to push the magic reboot button that will fixed all problems and makes the Windows system runs again[1]. In some cases the problem can not be fixed by a reboot, and then you have to install at patch, to make your computer run aging. The only solution to the stalling problem, would be to setup the watch-dog timer. The watch-dog will then reboot the system, if the system should stall. The patch problem is a much bigger problem to solve, and it may not be possible to solve at all. Because of the high radiation in space, which can cause bit-flips in the RAM circuits and therefor corrupt the code, is it necessary to store the boot-code in the PROM but with the loss of the possibility to patch th boot code. Therefor the boot-code should be as simple as possible to avoid bugs and errors in the boot-code.

[1] Could happened for other systems than Windows

Chapter 3

Software Development

Besides having the proper development tools, it is important to have a strategy. Projects involving many persons as the DTUsat project are very hard (if not impossible) to control if the project development does not follow a overall strategy.

3.1 V-model

We have chosen the v-model, which is a very common model in the area of software development for highly critical systems. No decision on what developing model to follow on the entire DTUsat project has been taken, however the development follows the v-model very close. The idea of the

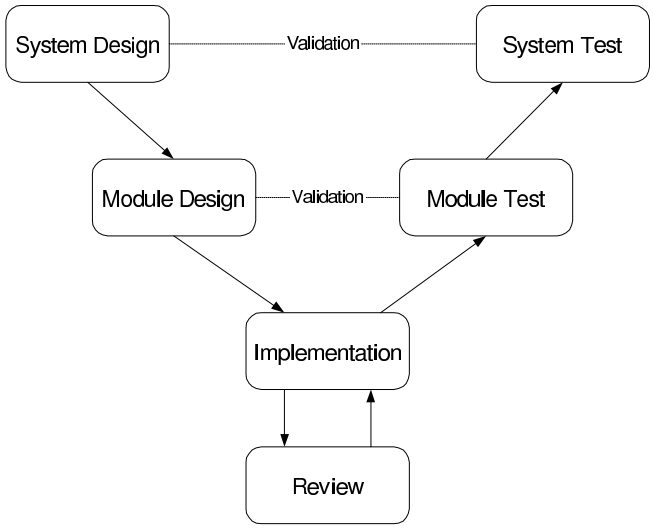

Figure 3.1: A graphical representation of the V-model

v-model is that there are some certain steps the project must go through. To move one step further, the first step must be approved (reviewed) by an authorized person. The main goal of the v-model is to assure a well documented project, where the requirements always can be tracked back to the design. The steps in the v-model are:

1. System Design
   In this step the system design is decided, and the overall requirements are described. When the system design is described it is possible to identify the modules needed.

2. Module Design
   The requirements for each module are identified and described.

3. Implementation
   The implementation must follow the designs chosen by the earlier steps.

4. Module Test
   In the module test the different modules are tested one by one. If the module test fails it should be considered whether the correct module design was chosen.

5. System Test
   The system test tests the entire system. The modules must be able to work together correctly. If the system test fails it should be considered whether the correct system design is chosen.

If one step fails the project should return to the step before and go on from there. The steps number 5 is the documentation for step number one, as well as step number 4 is documentation for step number 2. After each step some documentation must be presented.

3.2 The V-model applied to the DTUsat project

There are many groups involved in the DTUsat project. In general each group is in charge of their own module in the entire DTUsat project. The authorized persons are the different professor associated to the project. These professors must approve the different designs chosen by the groups before they can proceed their work. The documentation consists of the rapports handed in by the students. Most of the documentation is done in one big document describing all the 5 steps. This document should apply to the v-model standard.

3.3 Development environment

To develop software some development tools are needed. The development tools should make it possible for the developer to write, debug and test source code. Before you chose specific development toola, there are some things to be considered:

The Platform - The target platform must be supported by all the development tools in order to be able to develop correct software.

The standard - The development tools must implement the standard you have chosen to follow, otherwise it will be impossible to build the project.

The Documentation - The development tools must be well documented.

The cost - Many development tools are rather expensive. It is important to consider whether it is worth the cost to buy a expensive commercial product.

Having considered these things, we have chosen the GNU[5] software. The GNU development tools supports almost every platform (including the one we are using). All GNU software is very well documented, and as every program released under the GPL[1], the software is free. The GNU development tools is divided into three software packages: binutils, gcc and gdb. The normal method to get GNU software is to download it from one of the mirror sites available at the GNU home-page[2]. After downloading the software it should be compiled with the right options to fit the specific project.Se [6] for a guide on how to setup the development environment.

3.3.1 Binutils: GNU Binary Utilities

The GNU Binutils are a collection of binary tools. The main ones are[5]:

as - the GNU assembler.

ld - the GNU linker.

A full list of the programs included in binutils can be seen in Appendix A.

The GNU assembler compiled source code and make it to object code. This is done i two passes: In pass one the input file is read line by line to create the symbol table and the opcode table[3]. The symbol table has one entry for each symbol. A symbol is either a label or a value that is assigned a symbolic name. The opcode table contains an entry for each

[1] The GPL is the most common license among the free software licenses

[2] http://www.gnu.org

[3] Some assembly programs also produces a pseudo instruction table

opcode in the assembly language, this entry contains the type operands, the instruction length etc. In the second pass the object program is generated. The assembler once more runs the input file through one line at the time reading the opcodes, symbols and checking them with the generated tables and at last generating object code. If a symbol or an opcode is not present in the symbol table or the opcode table, an error message is produced. Finally the assembler generates information used by the linker.[1]

The GNU linker collects the separately compiled and assembled procedures and link them together to a executable binary program. The main task of the linker is to setup the program to run at the correct address space. The linking process is controlled by a linker script, which defines the memory layout of the output file (the binary executable program). The linking is very dependent on where the program should be run, therefore a special linker script for programs running on the on board computer had to be produced.

3.3.2 GCC: GNU Compiler Collection

In order to build code that should run on a ARM7TDMI processor, a cross-compiler is needed. A cross-compiler is a compiler running on one processor (i.e. an i386 processor) producing binary code for another (i.e. ARM7). To setup a cross-compiler the entire compiler should be recompiled (with a working C compiler) with the right options. We have chosen to use the gcc compiler which is the most widely used compiler to compile C code. The task of a compiler is to process a input source-file through 4 stages:

1. Preprocessing: The input file is run through the preprocessor which inserts macros and other definitions from header files.

2. Compilation: The compiler translates the high-level language (i.e. C) into some machine language.

3. Assembling: The assembler translates the compiled source file into object files (see section 3.3.1).

4. Linking: The linker collects the separately compiled and assembled procedures and link them together to a executable binary program (see section 3.3.1).

The gcc compiler uses the binary tools ld and as from the binutils to do low-level manipulation. To build the entire project we have used the make-program. Our Makefile is listed in Appendix C.1.

3.3.3 GDB: GNU Project Debugger

A debugger is a necessary tool for a software developer. The debugger makes it possible to look inside programs, even insight the registers of the CPU.

These features are used to find (and correct) bugs in the programs. Normally the debugged programs run on the same machine as the GDB (native), but the GDB implements a communication protocol that allows the debugged program to run on another machine (remote). This feature is very important developing software to embedded system, because it allows you to test whether the program runs properly on the target system.

3.3.4 The target

The target system described in section 2.1, is the hardware design of the flight model. Since the hardware development was not finished at the time this project began, there where no target system available. The target used for development is an evaluation board (EB40) provided by Atmel[15], as well as one early prototype of the on board computer, manufactured by the OBC-group[13]. To be able to do some low-level debugging the OBC-group has developed a extention board with 8 LEDs and 4 buttons. Connecting this extention board to the parrallel I/O-controller of the CPU makes it possible to do low-level debbuging, which is very useful of testing.

The evaluation board differs slightly from the final target system. The processor is another version (AT91M40800) with only 8Kb. of internal RAM, and the PROM is substituded by a flash, which makes it possible to rewrite the bootpogram if an error is found. What we describe in this rapport is the system how it is going to look on the final flight model.

3.3.5 Redboot

To get the first contact to the target system we used Redboot[17]. Redboot is a boot manager provided by the eCos system. The redboot software is a very useful debugging tool since it implements the gdb-stup protocol, witch allows remote debugging (see section 3.3.3). The first communication to the board was made through redboot, as well as the first program to run on the board was uploaded and started through redboot.

3.3.6 eCos

eCos is the operating system the on board computer should run, when it is working properly. The eCos[18] system is an open source operating system designed for embedded systems[4]. eCos has no influence on this project, except it is the operating system to load, when the boot process has finished.

[4]If Linux©is a open source parallel to Windows©, eCos©is a parallel to Windows CE©

3.4 Working with the evaluation board

This section describes how to use the evaluation board.

3.4.1 Communication with the board

It is possible to communicate with the board using the JTAG debug interface, or one of the two serial ports on the board.

JTAG

JTAG is a hardware debug interface to the CPU, which means it can be used without additional software running on the CPU. This makes it very suitable for doing software uploads to the evaluation board. We have used a Wiggler JTAG interface and OCD Commander software, which unfortunately only runs on Windows©. The On Board Computer group has written a document for setting up and using OCD Commander.[7]

Serial ports

The two serial ports can only be used if some software is running on the CPU. If you want to be able to debug programs on the CPU, you should install Redboot on the evaluation board. A precompiled version of Redboot(redboot.bin) and an installer(program_flash.srec) can be found at the DTUsat Homepage under Software>Boot-strap>files. Run the installer on the board through JTAG, and reboot the board. You should now be able to communicate with the evaluation board using a terminal client like minicom. The serial settings should be 38400 baud, 8 bit, no parity and 1 stop bit.

3.4.2 Debbuging software

Software can be debugged using any debugger which support the remote gdb protocol. We have used the gnu debugger and the graphical front end insight. Set target to Remote/Serial and baud rate to 38400. Remember that the debugger should be an cross-debugger. Se [6] for at guide on how to setup the development environment.

3.4.3 Running and debugging the boot-strap program

The boot-strap program can be compiled in two ways. `make test` will create a program which can be run through JTAG or gdb. `make boot` will create a stand-alone boot-strap program, which is suitable to write into the flash. The final version will have to be stored at address 0x0100000, but until then we use address 0x01040000. This way the evaluation board will always startup in redboot. To run the boot-strap program from address 0x01040000,

connect to redboot using minicom and execute `go 0x01040000`. This can be done by redboot at boot time by creating a boot-script. Type `fconfig` and write the script.

3.4.4 Communication with the boot-strap program

It is possible to communicate with the boot-strap program, by specifying the serial driver in the Makefile. On the host computer you should use our program `packet_sender` or write your own. The `packet_sender` program reads an hex encoded packet from `stdin`, convert it to bytes and send it to the evaluation board. The return packet from the board is converted to hex codes and send to `stdout`.

Chapter 4

Analysis and Design

We will first identify which errors that can happen with the satellite, and find out what to do with them. After that we will analyze what have to be done during a reboot, and how to do it.

4.1 Possible errors

Before we can start to design the boot-strap program we need to know which errors that can occur. We will divide the possible errors into two groups; the ones we want to be able to handle, and the ones we cannot do anything about.

4.1.1 Software errors

It is very likely that there will be error in the application software, since it is a very complex system. These errors could mean that some parts of the satellite does not function correctly. It should therefore be possible to upload new software to the satellite from the ground station.

4.1.2 Memory errors

Because of the radiation in space, it is very likely that we will have bit-flips in the memory. Depending of where the bit-flips occurs we will either get corrupted data, or the application software will crash. Corrupted data may lead to a crash of the application software, but until then we cannot handle it. When the application software crash, the watchdog timer will reset the on board computer. We can then choose to start the application software again, or enter the failsafe mode. We can also get other errors, so we can not access some part of the memory. Therefore should there be no hard coded memory addresses in the boot-strap program.

4.1.3 PROM errors

The boot-strap program is stored in the PROM, and if the PROM fails we cannot do anything.

4.1.4 CPU errors

If the CPU fails we cannot do anything. We might be lucky that only some part of it does not work, but we have decided that this is out of the scope of the boot-strap program.

4.1.5 Power errors

The satellite can run out of power, and the on board computer will then be shut down. When we get the power back, we should be able to restart the application software.

4.1.6 Other errors

Besides that there is a lot of errors that we can not handle. It could be problems with the radio and the mechanical construction of the satellite.

4.2 What should happen at boot?

First we need to initialize the CPU, and find a place in memory to store variables. Then we need to decide whether we should try to load the application software, or if we should enter a failsafe mode and establish communication with the ground station.

The most secure solution is to enter failsafe mode, and first load the application software after confirmation from the ground station. This way we can ensure that the satellite wont get stuck in some infinite reboot loop, but we are also wasting a lot of time waiting on the conformation from ground station. In most cases the satellite will be able to reboot immediately without errors, and therefore we do not want to wait for a confirmation from ground station. But if the satellite can not reboot without error, we should enter a failsafe mode and contact the ground station for further directives.

4.2.1 What should we load?

If we decide to load the application software, we need to know what to load, where to load it to and where to start execution. To keep track of data stored in the flash and where to load it, we have to create a simple file system. We want the file system to be flexible, support redundancy and contain checksums and information about where the data should be loaded to.

4.2.2 What should happen in failsafe mode?

The failsafe mode should be able to receive commands from the ground station, execute them and send the result back. It should be possible to upload and download software, manipulate the memory and flash and control the boot-strap program.

4.3 Our solution

We have decided to divide the boot-strap program into four modules:

- **Low level module** Initialize the CPU and setup a C-stack.
- **Load module.** Keeps track on system variables and loads the application software.
- **Failsafe module.** Establish communication and execute commands send from the ground station.
- **Library module.** Contains functions used by other modules.

We have to decide when to use C-code, and when to used assembly-code. C-code is far the most easy to write (and understand), and since there is no performance to gain by writing the code in assembly[1] we will write our programs in C when ever it is possible. To write in C one must have a stack where you can save your variables. This stack has to be set up in an assembly program. Another reason for using assembly code is the possibility to write programs that does not use any memory, but only registers. This is a very big advantage when you should test the memory. These fact has let us to the conclusion that we want to write the low level module in assembly-code and the rest i C-code. The entire boot-strap program will be executed from the PROM, to avoid memory errors and bit-flips.

4.3.1 Low level module

The low level module will be the first module to be executed. The purpose of the module is to initialize the CPU, and find enough memory to setup a C-stack. The module will do the following:

- Disable interrupts.
- Remap memory and flash so the CPU can access it.
- Enable the watchdog timer.

[1] Even if you are an assembly expert it is very hard (almost impossible) to write code better optimized than the compiler does when it compiles C-code

- Perform a simple memory test to find some working memory.
- Setup a C-stack pointer to the working memory.
- Start execution of the Load module.

We disable interrupts so we do not get interrupted during the boot. If the memory test do not find any working memory we are stucked. It is possible to write some sort of panic mode in assembly without memory, but it is out of the scope of this project. After the C-stack has been setup, we can execute code written i C.

4.3.2 Load module

The Load module is responsible for choosing whether we should load the application software or enter the failsafe mode. When the application software fails it could be caused by temporary errors like bit-flips, rare programming errors and power errors, or it could be permanent errors like permanent memory fails. After temporary errors we can restart the application software immediately without errors, but after permanent errors the application software will keep failing and we need to contact the ground station.

We have chosen to use a counter to count the number of reboots. If the counter exceeds a hard coded limit, the system will enter failsafe mode. If a temporary error happens we will restart the application software, and things should be fine. If it is a permanent error, we will keep restarting the application software until we exceeds the limit. It should be possible to reset the reboot counter, so a number of temporary errors over a long period of time, do not start failsafe mode.

System information

We need to store some variables in the satellite between reboots. We have chosen to store them in the flash, in something we call a system information block. Since it is very important that the system information is correct we can not rely on a hard coded address in the flash. If some part of the flash fails it should be possible to store the system information at another address. The system information contains the following data:

- Boot counter that count how many times we have rebooted.
- The entry point to the application software.
- A checksum of the system information.
- Information to communicate with the application software.
- Information about the data stored in flash.

Communicating with the application software

The application software might want to force the boot-strap program into failsafe mode for doing software uploads, or might want to reset the boot counter if the application software has been stable for some time. Since the application software and the boot-strap program do not run at the same time, the application software needs to store commands somewhere in the satellite. Again we can not rely on hard coded addresses, so we have decided to store commands as a part of the system information. This requires that the application software knows the address of the current system information block.

Flash file system

To know which data to load, we have to implement a simple file system for the flash memory. The file system contains a list of data module descriptors, which describe different chunks of code that should be loaded during boot. These descriptors are linked to the different flash block. The data chunks have to be stored in the beginning of each flash block. We have chosen this solution because it is rather simple, and very flexible. It is possible to load the same data chunk multiple places in memory, and to have a data chunk stored multiple places in the flash for redundancy.

4.3.3 Failsafe module

The main purpose of the failsafe module is to provide a way the ground station can manipulate with the on board computer. The communication with the ground station is going through a communication interface, which is described later. We have chosen to implement a set of commands that is strong enough to control the satellite, but operates on a very low level. The commands designed to our failsafe-program are: [2]

REBOOT

The reboot command reboots the system, and starts the boot-process all over again. When some corrections are done in the failsafe mode the reboot command must be run, to leave the failsafe mode.[3]

RAM_TEST

The ram_test command tests the RAM in the area specified by the packet received by the command handler. If an error occurs the return-packet con-

[2] These are the commands we have chosen to support, more could very well be implemented.

[3] It is possible to start the application software without rebooting by sending a EXE command with the correct start address.

tains the address to the place of failure.

COPY_TO_FLASH

The copy_to_flash command writes data to the flash from an address in the RAM. The return packet tells whether the write was successful or not.

COPY_TO_RAM

The copy_to_ram command is used to copy data from any address to an address in the ram.

GET_STATUS

The get_status command requests the failsafe mode to send the sysinfo block (see section 4.3.2) back to earth, so it can be examined.

SEND_STATUS

The send_status is a command that uploads a new sysinfo block to the satellite and stores it at the correct place in the flash.

EXE

The EXE command starts execution of a program located at a specific address.

GET_CHECK_SUM

The get_check_sum command calculates the checksum on the selected data and returns the checksum to earth.

UPLOAD

The upload command uploads data to the satellite and stores it in the RAM. If the data should be stored permanently the copy_to_flash command must be executed.

DOWNLOAD

The download command sends the selected data back to earth, where it can be examined.

DELETE_FLASH_BLOCK

The delete_flash_block command deletes a specific block in the flash. It is necessary to delete a block before it can be rewritten.

The most reasonable way to start a communication is to send a **GET_STATUS** command to the satellite. The satellite will respond by returning the sysinfo block containing the file system information and the information from the application software. By examine these data it should be possible to determine why the error occurred, or at least correct the error. If changes are made to the flash, the communication with the satellite should end with a **SEND_STATUS** command containing the correct information on the new file system. To exit the failsafe mode a **REBOOT** command is send.

The entire boot-process is shown as a state diagram in figure 4.1.

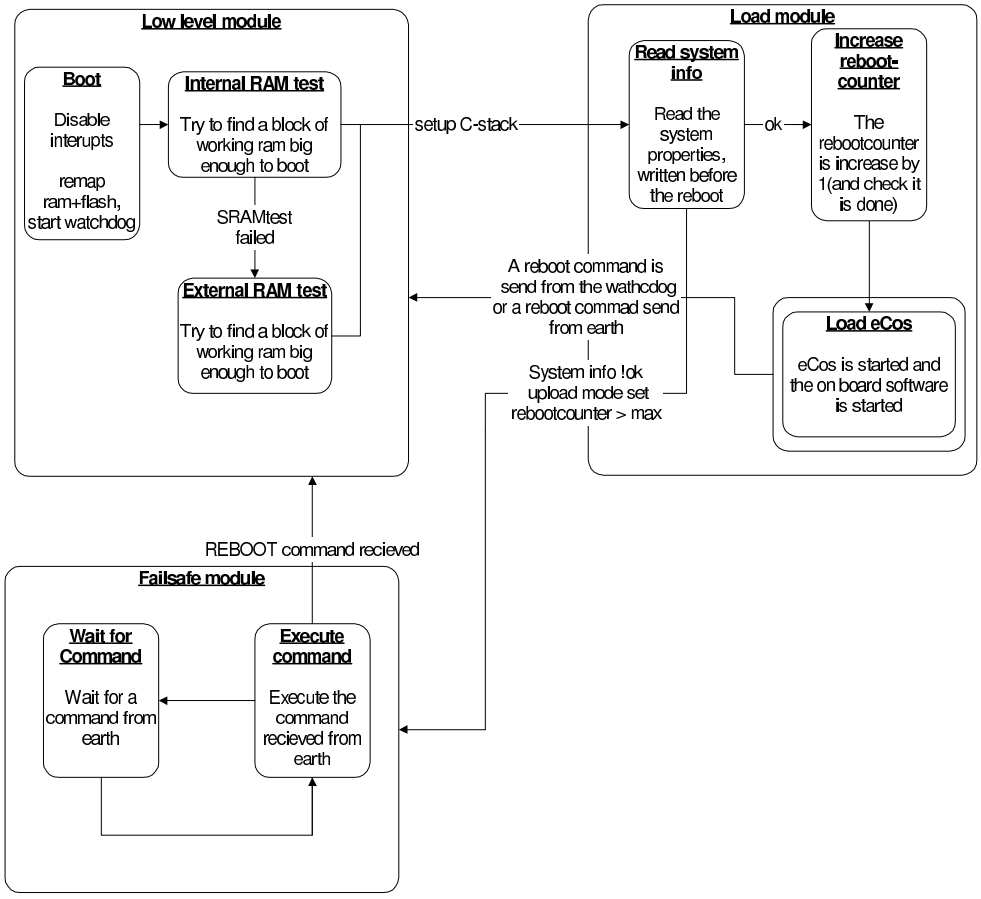

Figure 4.1: The system design

4.3.4 Library module

Some overall functions are needed by the boot-program. These should not be implemented as a module, but as individual functions. This will make the test more easy, since they do not depend on each other.

4.3.5 Flash driver

To be able to write to the flash it is necessary to do a set of low-level commands. To make this more easy we have chosen to implement a flash-driver. The flash-driver should provide functions to write and to delete parts of the flash. To assure the correctness of the data there should also be a check on whether the written data is correct or not.

4.3.6 Checksum computation

A checksum computation function is necessary due to the requirement of detecting corrupt data. We have chosen to implement the CRC checksum algorithm. This algorithm is one of the most well documented checksum calculation algorithms and it is widely used. A `cksum` program is implemented on most UNIX© platforms, this is an implementation of the 32-bit CRC. The default 32-bit crc checksum computing uses a standard polynomium:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

The theory behind the CRC is quite mathematical and beyond the scope of this project, but here is a brief description of what it does:

> The n bits to be evaluated are considered to be the coefficients of a $mod2$ polynomial $M(x)$ of degree $n-1$. These n bits are the bits from the file, with the most significant bit being the most significant bit of the first octet of the file and the last bit being the least significant bit of the last octet, padded with zero bits (if necessary) to achieve an integral number of octets, followed by one or more octets representing the length of the file as a binary value, least significant octet first. The smallest number of octets capable of representing this integer are used. $M(x)$ is multiplied by x^{32} (i.e., shifted left 32 bits) and divided by $G(x)$ using $mod2$ division, producing a remainder $R(x)$ of degree <= 31. The coefficients of $R(x)$ are considered to be a 32-bit sequence. The bit sequence is complemented and the result is the CRC.[11]

The 32-bit crc detects of cause not every possible error[4], but since the 32-bit crc algorithm is the one chosen by the Ethernet protocol we assume it is good enough for our purpose.

4.3.7 Memtest routine

It is very important to know the state of the RAM on the on board computer. If the RAM does not function correctly the software can come up with all types of strange failures. Therefore a memory test program is very good to have, from the developers point of view. If something is failing, it might be a good idea to test whether the memory works correctly. The main issue of a memory test on the satellite is to test whether the memory functions correctly, and if not where the failure is. There is no need knowing the type of the error[5], since there is no way of correcting the error. The memory test should only implement a test on, whether every bit in the tested area is capable of holding both 0 and 1.

4.3.8 Watch dog driver

The watch dog is implemented in the processor (see section 2.1.1) as a timer that triggers a reboot, if the timeout occurs. To avoid that the on board computer reboot, the timer must be reset before the timeout. Starting the watch dog, and reseting the watch dog timer is done by some low-level commands. To have a easy interface to the watch dog there should be implemented a watch dog driver providing a start command, that starts the watch dog, and a reset command for resetting the watch dog timer.

4.3.9 Communication interface

During the different phases of the project, we want to communicate with the satellites in different ways. It is therefore important to have a clear interface between the communication and the rest of the boot-strap program. We have designed such an interface, and implemented a serial driver so we can test the boot-strap program without a radio-link.

[4]To increase the security of the checksum, the polynomium must be bigger, and hence the checksum wider than 32-bit

[5]In ordinary desktop computers it can be very useful to know the type of the memory error

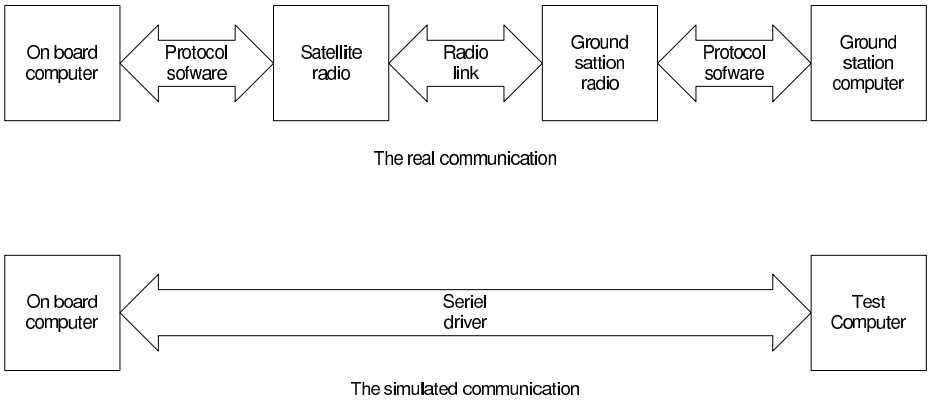

Figure 4.2: The communication interface

Chapter 5

Implementation

This section is the documentation on the third step in the V-model (see section 3.1). We will describe the most interesting parts in the implementation of the different modules. The final source files can be found in the appendix.

5.1 Coding guidelines

Before the implementation could begin we had to chose a programming language. Since the C programming language almost has become a standard to embedded systems and is the most widely supported language we have chosen to write the source-code in C. Compared to new modern languages as C++ and Java the C language is very small, but it provides all the features needed to write an efficient program for an embedded system. The source code is divided into a number of different files, to make the overview more easy. The main goal of this is to maintain a general and easy understandable syntax. All variables should be named with a meaningful name, e.g. `rp` does not make any sense unless you have written the code yourself however `return_packet` is easy to understand for everyone. Another thing that could be done to ease the readability of the code is to type cast the data section of a packet into a struct:

```
cmd_data = (cmd1_struct*) packet.data
```

Where the struct looks like this:

```
/* definition of the cmd1_packet */
typedef struct {
     unsigned char error_code;
     type variable1;
     type variable2;
} __attribute__ ((packed)) cmd1_struct;
```

The `packed` attribute specifies that a variable or structure field should have the smallest possible alignment, one byte for a variable, and one bit for a field, unless you specify a larger value with the `aligned` attribute. Otherwise the compiler will try to optimize the struct by reorganize the struct variables and insert extra bytes so all variables are aligned. When you have a pointer to at struct it you do not have to consider where in the data section the different data are stored, you simply just assign data to the struct:

```
cmd_data->error_code=NO_ERROR;
```

It is also important to consider how the stack should be used by the C-programs. We would like to keep as small a stack size as possible. With a very small piece of memory used for the stack it will almost always be possible to find a big enough piece of working memory to set up a stack, and hence it will always be possible to at least start up in failsafe mode. To reduce the amount of memory used we will throughout the program parse arguments *by reference* in stead of *by value* unless the argument is a fixed size on 4 bytes or below.

In the area of checks on whether addresses and pointers are valid data we have chosen not to implement any checks. This is done due to the fact that the worst thing that can happened, if an address points to an invalid address space, is a reboot. If we introduce even a very small error in a check, the effect could be fatal. We suggest these checks are implemented in the ground software, since software errors can easily be corrected on the ground. In our code we have used little endian. This is due to the fact that most processors are running little endian.

5.2 Library module

The library functions are described in the header file `system.h` (see Appendix C.13). Most of the overall functions are more or less functions collected from different libraries and modified to interface with our target system.

5.2.1 Flash driver

The idea of a device-driver is to hide the hardware layer and provide an easy understandable API (Application Programmers Interface). The function needed is basically `write` and `delete` a block. The flash can be accessed as normal memory therefore no read function is needed. The data sheet concerning the flash[19], describes which byte sequences one should use to program the flash, and the state diagrams to use, when waiting for the embedded algorithmens to finish. The implemented functions have the following prototypes:

```
int flash_write(address flash, const address mem, int len);
int flash_erase_sector(address flash);
```

The integer returned by the functions should be `NO_ERROR` on success or an error code. The implementation of the write function does not start by erasing the sector. This means it returns an error if the data section contains any data. The reason not to implement the erase function prior to the write is that i could be very useful, since it is possible only to write a very little amount of bytes at the time, but you have to delete an entire block every time the delete function is called. The flash driver (`flash_driver.c`) can be found in appendix C.8.

5.2.2 Checksum computation

An implementation of the 32-bit crc is found in [2]. We have chosen to follow this implementation since it is important to have a well described checksum function when several modules should interface to the same code. The only changes compared to the source file listed in [2], is a number of types are changed to general types, we have defined for our project. The source file `crc.c` is listed in Appendix C.6. The function prototype is:

```
checksum crcCompute(address message, unsigned int nBytes);
```

Another function called `crcInit()` is present. This function is used to calculate the remainder of every possible byte compared to the polynomial. Since this does not change we have chosen to make this remainder a constant array (see appendix C.7), therefore the `crcInit()` is never called in the program.

5.2.3 Memtest routine

The memory test program implemented is a modified version of a memory program presented in [2]. The selected data section is verified by writing a data pattern to each address, reading it back comparing it to the first value. Then the data pattern is inverted and the procedure is repeated. The `memtest` prototype is defined:

```
address memtest(int *baseAddress, unsigned long nBytes);
```

The caller of this program must be aware of the fact that the addressed memory is overwritten. No memory protection is implemented, hence a memtest on an address space already in use will result in undefined errors.

5.2.4 Watch dog driver

The watch dog driver can be compared to the flash driver. The main purpose is to hide the hardware layer and provide function calls. The watch dog driver can be set up by writing a specific byte sequence to watch dog. The prototype of the functions are:

```
void wd_start();
void wd_touch();
```

When the watch dog is started it has to be touched frequently if a reboot should not occur. The watch dog driver (watchdog.c) is shown in appendix C.15.

5.3 Low level module

The initial boot code written in assembly is the initializing part of the boot-program. First we turn of all interrupts, to avoid unexpected errors. When the interrupts are turned off the `remap` function is called. This function maps the hardware so the external memory and flash is available for the CPU (see figure 5.1). A simple memory check is now performed. This is done by writing one word and then reading it back. When a big enough RAM section is found the C-stack is set up, and the load module is started. The implementation of the low level module is done in `init.S` (see appendix C.9).

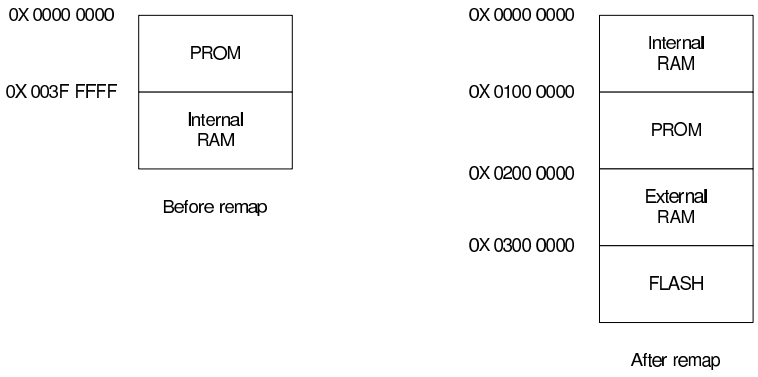

Figure 5.1: The Memory Map before and after the remap command

5.4 Load module

The module can be divided into smaller sections. We need to implement a secure way of load and store system information, a simple file system to keep track of the flash and a function which decides what to do at boot time.

5.4.1 System information

We have decided to store all informations about the state of the boot-strap program in the flash. It is not necessary to store all the information in the

flash, but it gives us a simple interface to the system information. It is very important that the system information is correct. A flash error, so the boot counter is not increased, would cause the satellite to enter an infinite boot loop. Therefore we have to be very careful before we accept the data read from flash. To avoid problems with defect bytes in the flash, we have decided that the system should be able to load and store the system information from multiple places in the first 8 Kb flash block. This is done by adding a magic number to the system information, and loop through the flash until we find a valid magic number. The magic number is a 32-bit number, it is theoretically impossible to get this by accident. The system information is also protected with a checksum, so we can detect errors.

load_sysinfo

The function will find valid system information in the flash, and copy it to the place where the pointer points to. The function loops backwards through the possible system informations blocks, until it finds a valid magic number. Then it calculates the checksum and compares it to the stored value. If everything goes alright, the information is copied to ram, and we perform a test of the flash block before we return.

```
void load_sysinfo(sysinfo_block*)
```

By testing that we can erase the block, and write zeros to it, we ensure that no bits in the flash are stuck somewhere. This is the reason to copy the sysinfo block to ram. The function does not return anything, but set an error code in the system information. If something goes wrong it is `SYSINFO_LOAD_ERROR`, otherwise it is `OK`.

save_sysinfo

The function will save the system information to the flash.

```
void save_sysinfo(sysinfo_block*)
```

One of the variables in the system information tell us where the system information should be stored to. Before we store the system information we compute the checksum. If the system information can not be stored the error code is set to `SYSINFO_WRITE_ERROR`, otherwise it is `OK`.

init_sysinfo

The function should be called if we can not load a valid system information.

```
void init_sysinfo(sysinfo_block*)
```

The function test the flash block to find a place to store a new default system information. If the functions find working space for system information the error code is `SYSINFO_INIT`, otherwise it is `SYSINFO_INIT_WRITE_ERROR`

5.4.2 Communication with the application software

The boot-strap accepts commands from the application software, if they are stored in the `os_comm` variable in the system information. Since the variable can be located many places in the flash, we need to specify a pointer to the system information we uses. This is done by declaring an `sysinfo_block* sysinfo` in the application software, compile and link the application software and set the `sysinfo_os_pointer` variable in system information to the memory address where the `sysinfo_block* sysinfo` is stored. During load of the application software the `sysinfo_block* sysinfo` will be replaced by a pointer to the actual system information block. The boot-strap program accept the following commands:

- `OS_DO_NOTHING` Default command.
- `OS_RESET_BOOT_COUNTER` Reset the boot counter.
- `OS_FORCE_FAILSAFE` Force the boot-strap program to start failsafe mode.

The corresponding values are selected so it is possible to save commands in the flash, without erasing the flash first.

5.4.3 Flash file system

The file system contains of two tables. A data table and a flash block table. The data table contains a list of data modules descriptors with information of the length and checksum of the data, the place the data should be loaded in the memory and a status field that indicates if the data should be loaded during boot. The flash block table links the different flash blocks with the data modules. It contains of a 32 byte array, with one byte to each flash block. This way it is possible to specify which data each flash block contains, and where it should be loaded. During boot all data modules with status set to load, will be loaded from the flash. It is possible to store data redundant in the file system by linking a data module descriptor with two or more flash blocks. You can also have the same flash block loaded multiple places in the memory by linking more data module descriptors with one flash block. The data stored in the flash blocks, must be located at the beginning of each sector. The data is loaded by two nested loops. For each data module descriptor with status set to `MODULE_LOAD`, it will iterate through the flash to find a block with the same identification number and a valid checksum. If we can not find all needed modules we set the error code to `MISSING_MODULE_ERROR` and start failsafe mode

5.4.4 System information struct

There must be a specific definition on how a sysinfo block should look like before implementing the functions. We have implemented a `struct` that describes where the different variables are stored in the sysinfo block. The organization of this struct is shown in figure 5.2.

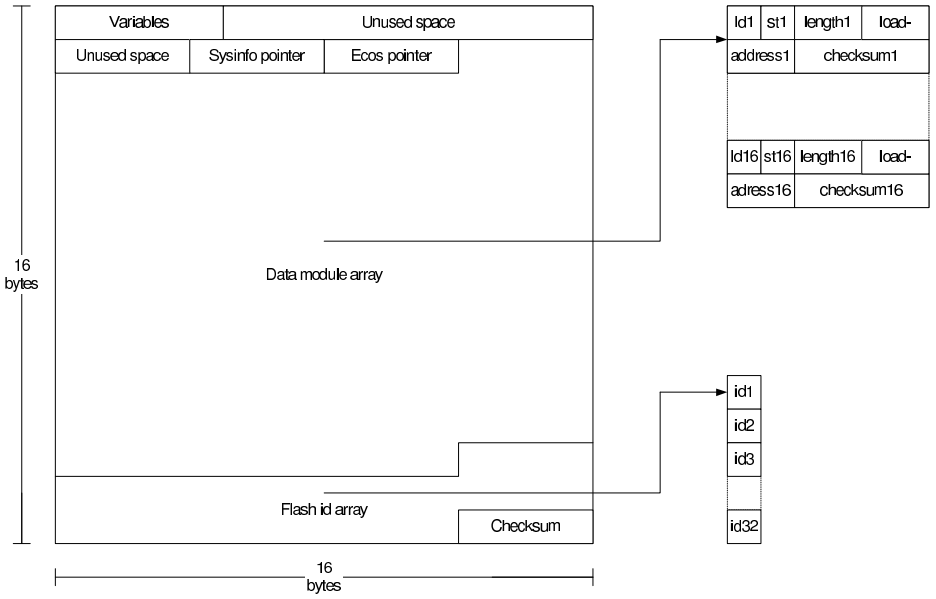

Figure 5.2: The structure of the sysinfo block

5.5 The Failsafe module

The failsafe module is done in a very simple design. Basically the failsafe mode is a command handler, and a set of commands. The command handler get at packet form the communication interface, then the command is executed and a return packet is send back. To leave the failsafe mode a REBOOT command must be send.

5.5.1 The Command Handler

The command handler is the main-loop of the failsafe program. The command handler waits for a packet from ground station containing a commando. The command handler can be described in pseudo code:

```
for(;;){
  get_packet(packet);
```

```
        switch(packet.command){
             case COMMAND1:
                  command1(*arguments);
                  break;
             case COMMAND2:
                  command1(*arguments);
                  break;
             /* The rest of the commands goes here */
             }
        send_packet(return_packet);
   }
```

To avoid errors there must be a predefined packet format that both the ground station and the satellite software agree on. The format chosen is shown in figure 5.3. The data section of the packet varies from command to

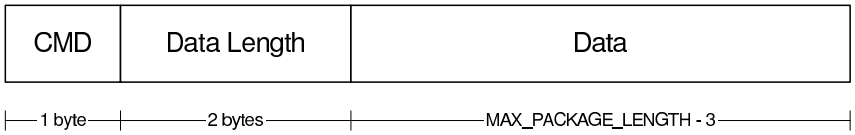

Figure 5.3: The packet format

command. Some commands have an empty data section, some have a data section containing pointer to something, and some again contain actual data. A list of what the data section for the different command contain is listed in Appendix B. The source code for the command handler can be found in Appendix C.4. The `MAX_PACKET_LENGTH` should be decidet by the protocol group.

5.5.2 Commands

We have designed a set of commands that can be executed from ground station, when the satellite is in failsafe mode. These commands[1] all executes and return a packet to earth reporting how the execution of the command went. The commands can be described in pseudo code as following:

```
command(packet_data,return_packet){
  command_variable=(command_type*)packet_data;
  /*The packet data is type cast into the a command variable/*
  local_variable=function_call(command_variable);
  if (no_errors){
```

[1] Except the REBOOT command

```
            return_packet->error_status=error;
            return_packet->data_length=0;}
        else{
            return_packet->error_status=OK;
            return_packet->data=local_variable;
            return_packet->data_length=length_of(local_variable);}
    }
```

The format of the return packet is shown i figure 5.4. The data section

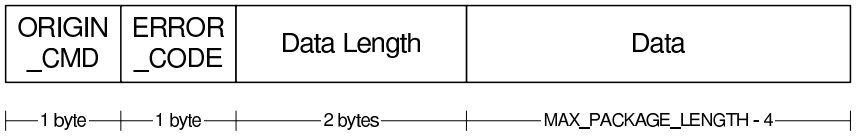

Figure 5.4: The return packet format

of the return packet varies from command to command. A list of what the data section return packet contains, as well as a full list of the commands is listed in Appendix B.

5.6 The Communication interface

The interface consist of three functions with the following prototypes:

`void comm_init();` Initialize communication.

`void comm_get_packet(pckt* packet);` Get packet from ground station and write data to `*packet`.

`void comm_send_packet(rpckt* packet);` Send data in `*packet` to the ground station.

We have made an serial driver which implement the communication interface. The driver sends and receives packets through the serial ports on the evaluation board. To send and receive the packets on the host computer we have created a program called `packet_sender`. The program is based on the example in HOWTO-SERIAL_PROGRAMMING[12].

Chapter 6

Test

This chapter is the documentation of step 4 and 5 in the V-model (see section 3.1). The idea of the test is to verify that the modules are working correct according to the design, and that the overall system is working. We have tested all the modules, and have made a complete test environment for the system test.

6.1 Module test

Tests are carried out on the evaluation board described in section 3.4. It is important to notice that the PROM is substituted by a flash, this makes it possible to rewrite the boot-program if an error is found. On normal desktop computers test programs would produce information to the standard output that should be reviewed by authorized test persons. No standard output is attached to the on board computer, so we are using a test board with lights connected to the parallel I/O-controller. By turning the different lights on and off, it is possible to verify the test is doing what it is supposed to do. In a good test it is important not only to test the case of the expected behavior but also the behavior when unexpected things occur.

6.1.1 Test: Overall functions

The overall functions should be tested one by one. The idea is to turn on a light on the test board every time a function has passed a test. The main function of the overall test program looks as the following:

```
void main() {
     led_init();
     led_blink();/* Blink the lights */
     led_on(LED0 | LED1);/*Turn on light 0 and 1 */
     if (!test_flash_erase_sector())
          led_on(LED2); /* Turn on light 2 */
```

```
        if (!test_flash_write())
             led_on(LED3); /* Turn on light 3 */
        if (!test_crc_computation())
             led_on(LED4); /* Turn on light 4 */
        if (!memtest_test())
             led_on(LED5); /* Turn on light 5 */
        if (!watchdog_test())
             led_on(LED6); /* Turn on light 6 */
        led_on(LED7); /* Turn on light 7 */
}
```

To verify the test all the lights must be turned on (from 0 to 7). When all the lights are turned on the system should wait for approx. 6 seconds and then reboot. When the system reboots the lights are turned of shortly.

The test program behaves as expected. The entire test program for the overall functions is listed in Appendix D.5. In the following the test for each function is described.

Flash driver

To test whether the flash driver works, all the APIs must be tested, but it is also important to test whether the function returns the correct error codes. Therefore the following test sequence is implemented:

`write zeros to flash block;`
OK
`overwrite flash block with a 1;`
This test should fail since the flash must be erased prior to writing
`delete flash block;`
OK
`overwrite flash block with a 1;`
OK

The implementation in C-code of this sequence is implemented in the `test_flash_write()` (see Appendix D.5).

Checksum computation

To test the implementation of the crc algorithm we have chosen the data section: *"A test of crcComputation"*. Stored in a character array which is 24 bytes. Running the `crcCompute` on this data section a checksum is calculated. Prior to the test we have calculated the checksum on this data section running the crc program on a standard desktop computer. This checksum is stored in the variable `known_checksum` and used to compare

with the check sum calculated on the target system. The data is copied to the RAM and to the flash, and the checksum is recalculated to verify that it still returns the correct checksum. At last the data section is changed to *"A test of CrcComputation"* (the c in crcComputation is changed to at capital C). This is done to verify that the checksum changes when the data changes.

Memtest routine

Before testing the memtest program we must assume that the RAM is working correct. Since we do not have the tools to generate a controlled hardware error the only thing we can test is whether the memtest runs correctly and returns without an error. Of cause we can can run a memory test on a piece of memory not existing (address an non existing address) and make sure this returns an error. Th `memtest` routine is first called with a valid address and a number of bytes to test. Afterwards it is called with an invalid address to see it detects the non existing memory.

Watch dog

The watch dog is rather hard to test, because the main goal is to reboot the system. The test we have done is implemented by first initializing the watch dog. Then we continuously reset the timer but we wait while making the period between the resets longer and longer before we at last stop resetting the timer. When the timer is not reset any more the watch dog should reboot the system after approx. 6 seconds.

```
wd_start();
for(f= 0; f< 50000;f=f+1001){
     i=f;
     while (i) i-; /* wait longer and longer */
     wd_touch();/* reset the timer */
     if (f%2==0) /* Blink the light number 6 */
          led_on(LED6);
     else
          led_off(LED6);
}
```

The blinking with light number 6 is done so it is possible to identify whether anything is going on. When the system is rebooted the lights are turned off.

6.1.2 Test: Low level module

The low level module is tested by writing the code to the start address of the flash that simulates the PROM. When the on board computer is rebooted the low level module must be able run the low level memory test, and set up

a C-stack and the load the boot-code (boot.c). The low level module is also tested in the system test (6.2).

6.1.3 Test: Load module

The test program `load_module_test` test the three sysinfo functions and the code that load data from flash to memory.

Sysinfo

`sysinfo_load()` : We test the function by writing three sysinfo blocks to the flash. The first block has an invalid magic number, the second has a invalid checksum and the third is correct. We call the function, and see that the third sysinfo block is returned.
`sysinfo_save()` : We create a sysinfo block in the memory, and call the function. Afterward we compare the sysinfo block in memory, with the one in flash.
`sysinfo_init()` : We call the function, and it returns a sysinfo block. We test the block is OK, by calling sysinfo save.

The Load module

We test the load code by saving a program (`lightshow`) and a sysinfo block in the flash, and then call `boot()`. Depending on what we save in the sysinfo block, we have tested six different cases:

- All data is valid, and the program starts running.
- Some date is invalid, and failsafe will start.
- Some data is invalid, but valid data is stored another place in the flash. The checksum check will then load the valid data and boot up.
- Set bootcount $>$ max_bootcount and failsafe will start.
- Set os_comm to OS_FORCE_FAILSAFE and failsafe will start.
- Set bootcount $>$ max_boot_count and os_comm to OS_RESET_BOOT_COUNTER and the light show will start.

6.1.4 Test: Failsafe module

To test the failsafe module we developed a cmd_test_driver (see D.1) which implements the communication interface.

Command handler

For testing the functionality of the command handler, including the packet handling, and that the commands call the right function, we have hard coded a sequence of packet, and every time the cmd_handler calls the `comm_get_packet`, a hard coded packet is returned. The sequence of packets will include all the different commands. We use the debugger and the lights to the verify the result. The light blinks twice when a command was sent and received successfully.
The sequence have the following packet order

- UPLOAD packet
- RAM_TEST packet
- COPY_TO_FLASH packet has to fail, because the block was not deleted
- DELETE_FLASH_BLOCK packet
- COPY_TO_FLASH
- COPY_TO_RAM
- SEND_STATUS
- GET_STATUS
- GET_CHECK_SUM
- DOWNLOAD
- EXE executes a program that makes a nice blink show.
- REBOOT reboot the test board;

Further more was the function `cmd_func_test_flow()` develop to test the following functions

Up- and download

To test the `upload()` and `download()` functions we try to upload data, and then download the same data again. If the downloaded data corresponds to the uploaded data then both methods works. Of cause there could be a writing error and a similar reading error and the test will still return true, even though the data was stored corrupt. But using the Insight/GDB debugger, gives us the option to check the value of an address, and we will then be able to verify that the data was not stored corrupted.

Get- and sendStatus

To test the `get_status()` functions we try to get the status and checks if i t corresponds to the init status data. To check the `send_status()` we send new values, and get them again. The retrieved should corresponds with the values just send. One should remember to check the address value, using the debugger, to verify that the data was not stored corrupt.

6.2 System Test

The system test is a test of the entire system. The purpose of the system test is to test overall design of the software. It should consist of a number of cases where we send a receive packets to the satellite to simulate real space usage. We have written a communication driver for the boot software, which can communicate through a serial link connected to the USART on the processor. On the host computer we have written a program which can send and receive packets from the serial link, and a shell script which runs automated tests based on test files. The test file should contain a number of packets to send to the on board computer, and the expected returns packet. An example of a test file is locate in appendix D.9. The test can be run by typing `./runtest.sh testfile`.

First of all the on board computer has to be able to boot correctly. This is tested by writing a application to the flash, and writing the correct information to the system information. When booting the boot-program should load the program from flash. To increase the `bootcount` variable in the sysinfo, the on board computer should be rebooted a number of times. When `bootcount > MAX_BOOT_COUNT` the on board computer should enter failsafe mode instead of the application.

6.3 Performance

Since the test setup was not the final flight model of the on board computer, there is no reason to give any precise time estimates, because these values can differ a lot when just a few components are changed. The amount of memory and CPU resources is large according to our needs so there should not be any problems. We have chosen to compile the boot-strap program optimized for size. This generates smaller code, but longer execution times. Depending on the flight model, we might need to do some tradeoff between speed and size by choosing other optimizations methods.

6.4 Verification

No formal verification of the boot-program has been done. All the code has been written without any infinite loops, and without stopping the watch dog at any point. The critical section would be in the communication with the ground station, where the protocol has to wait for a packet.

All the test described were successfully carried out.

Chapter 7

Conclusion

In our solution to the boot-strapping problem of a satellite, we have chosen to approach the problem more practical than theoretically. This is done because of the fact that the product is a part of a real life project, and therefor has to be able to boot the DTUsat.

We have made certain priorities during the development of this software. We would rather have working boot-program than a long theoretic discussion on how to implement it.

The boot program is designed i a way that one easily could add a new command to the cmd_handler. It will then be possible to extend the boot program with missing command, if it is decided.
When the satellite is in orbit, it is obvious that the boot system can not be extended with a missing command. But using the `upload` and `execute`, it would still be possible to make small program, and run then, thus the satellite is in failsafe mode. This makes the satellite very flexible, even in failsafe mode.

7.1 To do

We will shortly describe what we think has to be done before the boot-strap program can be launched into space:

- Implementation of the communication interface. We have made a serial implementation for testing purposes, but the flight model will use the radio link. It is important that interface to the radio link, remember to touch the watch dog, while waiting for a new packet.

- Implementation of a DoPanic mode. The do panic mode, is a state were no error free RAM was found. It may be possible to develop a small failsafe program that runs only in the registers of the CPU. At the moment the DoPanic mode blinks and wait for the watchdog to reboot the system.

- The system test still needs to be made. We have provided all the tools to make the system test.
- It has to be consider the possibility if some hardware modules should be started during failsafe. For instance the detumbling has to be started in order to stabilize the satellite.
- Make on board software module. There should be made a module so we can reset the boot counter and force the boot-strap program to start into failsafe mode.

In general the project fulfill the requirement, listed in section 1.4. Of cause the test is to superficial to ensure that there are no errors at all.

A more systematic test should be done, where all possible combinations of software failures should be checked for unpredictable behavior that will bring the satellite in some bizarre situation.

In general this project has been an interesting experience for all of us, and we hope the satellite is in orbit and functioning in the spring of year 2003.

Bibliography

[1] Andrew S. Tannenbaum: *Structured Computer Organization*, Prentice Hall (1999)

[2] Michael Barr: *Programming Embedded Systems*, O'Reilly (1999)

[3] Tobias Oetiker, Hubert Partl, Irene Hyna and Elisabeth Schlegl. *The Not So Short Introduction to LaTeX,* Version 3.20 (2001)

[4] http://www.dtusat.dtu.dk

[5] http://www.gnu.org

[6] http://sources.redhat.com/ecos/tools/linux-arm-elf.html

[7] http://dtusat.dtu.dk/files/download/471/arm-dev.html

[8] GNUPro© Toolkit, *Getting Started Guide*, 2001 Red Hat, Inc.

[9] GNUPro©Toolkit, *GNUPro Development Tools*, 2001 Red Hat, Inc.

[10] GNUPro©Toolkit, *GNUPro Auxiliary Development Tools*, 2001 Red Hat, Inc.

[11] http://www.tac.eu.org/cgi-bin/man-cgi?cksum+1

[12] http://www.linuxdoc.org

[13] Jonas Sølvhøj, Malte Breiting and Morten Briand Thomsen *Onboard Computer for Pico Satellite*, Ørsted DTU

[14] Brian W. Kernighan and Dennis M. Ritchie: *The C Programming Language*, Prentice Hall (1988)

[15] Data Sheet, AT91X40, 2001

[16] http://www.at91-forum.com

[17] Redboot^{TM} User's Guide, July 2001 Red Hat, Inc.

[18] eCos^{TM} Reference Manual, March 2000 Red Hat, Inc.

[19] Data Sheet, AMD Am29LV017D, November 7, 2000

[20] Steve Furber, *ARM system-on-chip architecture*, Addison-Wesly (2000)

Appendix A

The programs in the GNU binutils

ld - the GNU linker.

as - the GNU assembler.

addr2line - Converts addresses into filenames and line numbers.

ar - A utility for creating, modifying and extracting from archives.

c++filt - Filter to demangle encoded C++ symbols.

gprof - Displays profiling information.

nlmconv - Converts object code into an NLM.

nm - Lists symbols from object files.

objcopy - Copys and translates object files.

objdump - Displays information from object files.

ranlib - Generates an index to the contents of an archive.

readelf - Displays information from any ELF format object file.

size - Lists the section sizes of an object or archive file.

strings - Lists printable strings from files.

strip - Discards symbols.

windres - A compiler for Windows resource files.

Appendix B

Commands in failsafe

B.1 The commands of the failsafe mode

Command Name	Command Code
REBOOT$^{*(+)}$	1
RAM_TEST$^{((+))}$	2
COPY_TO_FLASH^{+}	3
COPY_TO_RAM^{+}	4
GET_STATUS*	5
SEND_STATUS^{+}	6
EXE^{+}	7
GET_CHECK_SUM	8
UPLOAD^{+}	9
DOWNLOAD^{+}	10
DELETE_FLASH_BLOCK^{+}	11

Table B.1: The commands

*The data section is empty for the recieve packet.
$^{+}$The data section is empty for the return packet.
$^{(+)}$No return packet is send.
$^{((+))}$If no failure the data section of the returnpacket is empty.

B.2 The data format of the recieved data

RAM_TEST Data section	
address (4 bytes)	**unsigned int** (4 bytes)
The adress where to start the RAM test	The length of the area to test

COPY_TO_FLASH Data sectio		
address (4 bytes)	**address** (4 bytes)	**unsigned int** (4 bytes)
The address where to copy from	The address where to copy to	The length of the area to copy

COPY_TO_RAM Data sectio		
address (4 bytes)	**address** (4 bytes)	**unsigned int** (4 bytes)
The address where to copy from	The address where to copy to	The length of the area to copy

EXE Data section
address (4 bytes)
The address where to begin the execution

GET_CHECK_SUM Data section	
address (4 bytes)	**unsigned int** (4 bytes)
The adress to the checked data	The length of data to check

DOWNLOAD Data section		
address (4 bytes)	**unsigned short** (2 byte)	**char array** (1 - 252 bytes)
The address where to store the uploaded data	The length of the data in the packet (maximum 252 bytes)	The data

GET_CHECK_SUM Data section	
address (4 bytes)	**unsigned int** (4 byte)
The where the data to download is stored	The length of the data to download (maximum 252 bytes)

DELETE_FLASH_BLOCK Data section
address (4 bytes)
The address of the block that should be deleted.

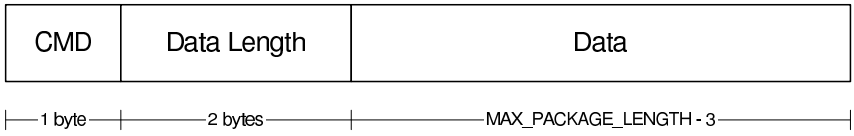

Figure B.1: The package format

B.3 The data format of the send data

RAM_TEST return data section
address (4 bytes)
The address where the RAM-failure is. If no failure is detected, the return data section is empty.

GET_CHECK_SUM return data section
checksum (4 bytes)
The calculated checksum is returned.

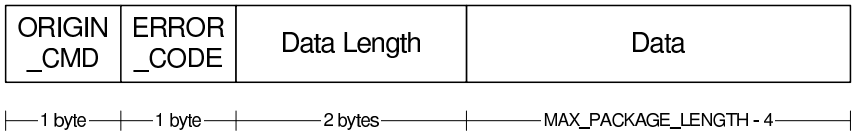

Figure B.2: The return package format

B.4 An ML description of the commands

```
Incoming packet(pckt):

|CMD |DATA_LENGTH |DATA |
-----------------------------------------------
|1 byte |2 bytes |max_packetsize -(1+2) |


pckt::CMD,DATA_LENGTH,(DATA|NULL)

CMD::REBOOT|RAM_TEST|COPY_TO_FLASH|COPY_TO_RAM|GET_STATUS|
SEND_STATUS| EXE|GET_CHECK_SUM|UPLOAD|
DOWNLOAD|DELETE_FLASH_BLOCK

DATA_LENGTH::unsigned short
```

```
DATA::cmd_copy|cmd_upload|cmd_ramtest|cmd_download
|cmd_cksum|cmd_status|cmd_exe|cmd_delflash

REBOOT:: 1
RAM_TEST::2
COPY_TO_FLASH::3
COPY_TO_RAM::4
GET_STATUS::5
SEND_STATUS::6
EXE::7
GET_CHECK_SUM::8
UPLOAD::9
DOWNLOAD::10
DELETE_FLASH_BLOCK::11

cmd_copy::address,address,INT_DATALENGTH
cmd_upload::address,(data)*
cmd_ramtest::address,INT_DATA_LENGTH
cmd_download::address,INT_DATA_LENGTH
cmd_cksum::address,INT_DATA_LENGTH
cmd_status::(data)*
cmd_exe::address
cmd_delflash::address

INT_DATA_LENGTH::unsigned integer
address::unsigned char pointer
data::char

Return packet(rpckt):

|ORIGIN_CMD |ERROR_CODE |DATA_LENGTH |RETURN_DATA |
-------------------------------------------------------------------------
|1 byte |1 byte |2 bytes |max_packetsize-(1+1+2) |


rpckt::ORIGIN_CMD, ERROR_CODE, DATA_LENGTH, RETURN_DATA

ORIGIN_CMD::CMD
ERROR_CODE::NO_ERROR|UNDEFINED_CMD|RAM_ERROR|FLASH_WRITE_ERROR|
    FLASH_DELETE_ERROR

RETURN_DATA::return_ramtest|return_cksum|cmd_status|data

NO_ERROR::0
```

```
UNDEFINED_CMD::1
RAM_ERROR::2
FLASH_WRITE_ERROR::3
FLASH_READ_ERROR::4

return_ramtest::address
return_cksum::checksum
cmd_status::(data)*

checksum::long
```

Appendix C

Source files for boot

All source files are availeble at a public CVS-server http://cvsdtusat.it.dtu.dk.
Informations on how to obtain the source files can be found on the webpage.

C.1 Makefile

```
#Makefile for DTUsat - boot

# make boot creates boot-strap program to be written in the prom
# make test creates boot-strap program which could be executed
# from other programs like redboot.

#select communication interface
COMM = seriel_driver.o
#COMM = radio_driver.o
#COMM = cmd_test_driver.o

DEBUG = -g

PROC = arm
TYPE = elf

LDSCRIPT = ldscript
LDSCRIPT2 = ldscript.test

CC = $(PROC)-$(TYPE)-gcc
AS = $(PROC)-$(TYPE)-as
LD = $(PROC)-$(TYPE)-ld
OC = $(PROC)-$(TYPE)-objcopy

ARMCFLAGS = -mcpu=arm7tdmi -mlittle-endian -mapcs-frame -mno-sched-prolog
```

```
CFLAGS = $(ARMCFLAGS) $(DEBUG) -Wall -ffreestanding -Os -fvolatile
ASFLAGS =
LDFLAGS = -T $(LDSCRIPT) $(DEBUG) -EL
LDFLAGS2 = -T $(LDSCRIPT2) $(DEBUG) -EL

all: boot test

boot: boot_c boot_asm
$(LD) $(LDFLAGS) -o boot.elf boot_asm.o boot_c.o
$(OC) -O binary boot.elf boot.bin

test: boot_c crt.o test.o
$(LD) $(LDFLAGS2) -o test.elf crt.o test.o boot_c.o


load_module_test: boot_c crt.o testing/load_module_test.o
$(LD) $(LDFLAGS2) -o load_module_test.elf crt.o boot_c.o \
testing/load_module_test.o

overall_functions_test: boot_c crt.o testing/overall_functions_test.o
$(LD) $(LDFLAGS2) -o overall_functions_test.elf crt.o boot_c.o \
testing/overall_functions_test.o

boot_c: crc.o flash_driver.o memtest.o boot.o sysinfo.o cmd_handler.o watchdog.o \
data.o system.o $(COMM)
$(LD) $(LDFLAGS2) -i -o boot_c.o crc.o flash_driver.o memtest.o boot.o\
sysinfo.o cmd_handler.o watchdog.o data.o system.o $(COMM)

boot_asm: init.S
$(AS) $(ASFLAGS) -o boot_asm.o init.S

crt.o: crt0.o crt_wrapper.o
$(LD) $(LDFLAGS) -i -o crt.o crt0.o crt_wrapper.o

clean:
rm -f *.o
```

C.2 ldscript

```
/* Default linker script, for normal executables */
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm",
      "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(__reset)

SECTIONS
{
  . = 0x01040000;
  reset.text  :{ *(reset.text) }
  .text       :{ *(.text )     }
  .rodata     :{ *(.rodata )   }
}
```

C.3 boot.c

```
/*
  DTUSAT - boot
  boot.c
*/
#include "system.h"

void boot() {
  sysinfo_block sysinfo;
  int i, j;
  address ram, flash;

  //wd_start();

  sysinfo_load(&sysinfo);
  //if (sysinfo.err) {
  //  sysinfo_init(&sysinfo);
    failsafe(&sysinfo);
    // }

  /* Check if we should reset boot counter*/
  if (sysinfo.os_comm == OS_RESET_BOOT_COUNTER) sysinfo.bootcount = 0;

  /* Force failsafe from ecos */
  if (sysinfo.os_comm == OS_FORCE_FAILSAFE) failsafe(&sysinfo);

  /* Check numbers of reboot */
  if (sysinfo.bootcount > MAX_BOOT_COUNT) failsafe(&sysinfo);

  /* Update bootcount */
  sysinfo.bootcount++;

  /* Start load af onboard software */
  for (i=0; i<16; i++) { /* loop through modules */
    if (sysinfo.data_module[i].module_status == MODULE_LOAD) {
      for (j=1; j<32;j++) { /* loop through flash memory */
if (sysinfo.block[j] == sysinfo.data_module[i].id) { /* load module */
  flash = (unsigned char *) (FLASH_BASE + 0x10000 * j); /* flash address to load fro
  if (sysinfo.data_module[i].checksum ==
      crcCompute(flash, sysinfo.data_module[i].load_length)) { /*cheksum ok - copy t
    ram = sysinfo.data_module[i].load_addr;
    memcpy(ram, flash, sysinfo.data_module[i].load_length);
    if (sysinfo.data_module[i].checksum !=
```

```
crcCompute(ram, sysinfo.data_module[i].load_length)) {
      sysinfo.err = COPY_TO_RAM_ERROR;
      failsafe(&sysinfo);
    }
    break;
  }
}
      }
      if (j == 32) {
sysinfo.err = MISSING_MODULE_ERROR;
failsafe(&sysinfo);
      }
    }
  }

  /* clear os_comm, so we dont have OS_RESET_BOOT_COUNTER forever */
  sysinfo.os_comm = OS_DO_NOTHING;

  /* save sysinfo to flash */
  sysinfo_save(&sysinfo);
  if (sysinfo.err) failsafe(&sysinfo);

  /* set sysinfo pointer if defined */
  if (sysinfo.sysinfo_os_pointer)
    ((address) sysinfo.sysinfo_os_pointer)  = (address) FLASH_BASE + sysinfo.sysinfo_block *

  /* set pc to entrypoint */
  arm_execute(sysinfo.entrypoint);
  /* Software is running */
}
```

C.4 cmd_handler.c

```
/*
  DTUSAT - boot
  cmd_handler.c

*/
#include "system.h"
#include "cmd_handler.h"

void failsafe(sysinfo_block * sysinfo){

  pckt packet; //the incomming packet
  rpckt return_packet; //return packet

  // we are going to be in here for some time,
  // so we better save the sysinfo to the flash
  sysinfo_save(sysinfo);

  comm_init();
  for(;;){/* runs forewer (until a reboot are send from the
     ground_segment or when dog is not kicked */
    comm_get_packet(&packet);
    // get the cmd form the packet

    //handle the cmd;
    switch(packet.cmd){
    case(REBOOT):
      reboot();
      break;
    case(RAM_TEST):
      ram_test( (cmd_ramtest*) packet.data, &return_packet);
      break;
    case(COPY_TO_FLASH):
      copy_to_flash((cmd_copy*) packet.data,&return_packet);
      break;
    case(COPY_TO_RAM):
      copy_to_ram((cmd_copy*) packet.data,&return_packet);
      break;
    case(GET_STATUS):
      get_status(sysinfo, &return_packet);
      break;
    case(SEND_STATUS):
      send_status( (cmd_status*) packet.data, sysinfo, &return_packet);
```

```
      break;
    case(EXE):
      execute_address( (cmd_exe*) packet.data, &return_packet);
      break;
    case(GET_CHECK_SUM):
      get_check_sum( (cmd_cksum*) packet.data, &return_packet);
      break;
    case(UPLOAD):
      upload((cmd_upload *) packet.data, &return_packet);
      break;
    case(DOWNLOAD):
      download((cmd_download *) packet.data, &return_packet);
      break;
    case(DELETE_FLASH_BLOCK):
      delete_flash_block( (cmd_delflash*) packet.data, &return_packet);
      break;
    default:
      //Command not reconised
      return_packet.error_code=UNDEFINED_CMD;
      return_packet.data_length=0;
      break;

    }
    //adds the origin cmd to the return packet
    return_packet.origin_cmd=packet.cmd;
    //sends the confirm packet to ground
    comm_send_packet(&return_packet);
  }
}

void reboot() {
  arm_reboot();
}

void ram_test( cmd_ramtest* cmd, rpckt* return_packet){
  address return_value;

  return_value=memtest(cmd->start_address, cmd->data_length);

  if (return_value==0){
    return_packet->error_code=NO_ERROR;
    return_packet->data_length=0;}
  else{
    ((return_ramtest*) (return_packet->data))->failure=return_value;
```

```
    return_packet->error_code=RAM_ERROR;
    return_packet->data_length=sizeof(return_value);
  }
}


void copy_to_flash( cmd_copy* cmd, rpckt* return_packet) {
  int err;

  err = flash_write( cmd->to_address, cmd->from_address, cmd->data_length );
  if (err) {
    return_packet->error_code=FLASH_WRITE_ERROR;
    return_packet->data_length=0;
  }else {
    return_packet->error_code=NO_ERROR;
    return_packet->data_length=0;
  }
}

void copy_to_ram(cmd_copy* cmd, rpckt* return_packet) {
  memcpy((void*) cmd->to_address, (void*) cmd->from_address, cmd->data_length );
  return_packet->error_code=NO_ERROR;
  return_packet->data_length=0;
}

void get_status(sysinfo_block* sysinfo, rpckt* return_packet) {
  // return system information without magic number and final checksum
  memcpy( (address)return_packet->data, ( (address)sysinfo ) +1, 251);
  return_packet->error_code=NO_ERROR;
  return_packet->data_length=251;
}

void send_status(cmd_status* cmd, sysinfo_block* sysinfo, rpckt* return_packet) {
  // set system information, without magic number and final checksum

  memcpy( ((address)sysinfo ) +1, (address) cmd, 251);
  sysinfo->magic_number = SYSINFO_MAGIC_NUMBER;

  sysinfo_save(sysinfo);
  return_packet->error_code=sysinfo->err;
  return_packet->data_length=0;
}
```

```
void execute_address( cmd_exe* cmd, rpckt* return_packet){
  arm_execute(cmd->addr);
}

void get_check_sum( cmd_cksum* cmd, rpckt* return_packet){
  checksum cs;

  cs=crcCompute(cmd->start_address, cmd->data_length);

  ((return_cksum*) (return_packet->data))->cs=cs;
  return_packet->error_code=NO_ERROR;
  return_packet->data_length=sizeof(cs);
}

void upload( cmd_upload* cmd, rpckt* return_packet){
  memcpy( cmd->start_address, (address)&(cmd->data), cmd->data_length);

  return_packet->error_code=NO_ERROR;
  return_packet->data_length=0;
}

void download( cmd_download* cmd, rpckt* return_packet){
  memcpy( return_packet->data, cmd->start_address, cmd->data_length);

  return_packet->error_code=NO_ERROR;
  return_packet->data_length=cmd->data_length;
}

void delete_flash_block( cmd_delflash* cmd, rpckt* return_packet){
  int err;

  err= flash_erase_sector(cmd->delete_address);

  if(err!=0){
    return_packet->error_code=FLASH_DELETE_ERROR;
    return_packet->data_length=0;
  }  else{
    return_packet->error_code=NO_ERROR;
    return_packet->data_length=0;
  }
}
```

C.5 cmd_handler.h

```
/*
  DTUSAT - boot
  cmd_handler.h

*/

#ifndef __cmd_handler_h
#define __cmd_handler_h

#include "system.h"

//-------------error codes
#define NO_ERROR 0
#define UNDEFINED_CMD 1
#define RAM_ERROR 2
#define FLASH_WRITE_ERROR 3
#define FLASH_DELETE_ERROR 4

//------------packet constants
#define CMD_LENGTH 1 // The length of the cmd field in bytes
#define DATA_LENGTH_LENGTH 2 // The length of the data_length field in bytes
#define ADDRESS_LENGTH 4 // The length of an address in bytes
#define MAX_PACKET_SIZE 256 // The max byte size of a packet
#define ERROR_CODE_LENGTH 1 // The length off the error_code_length
#define HEAD_PACKET_LENGTH CMD_LENGTH+DATA_LENGTH_LENGTH // the header length of a packet..
#define HEAD_RETURN_PACKET_LENGTH CMD_LENGTH+ERROR_CODE_LENGTH+DATA_LENGTH_LENGTH //the head
#define MAX_RETURN_DATA_SIZE MAX_PACKET_SIZE-HEAD_RETURN_PACKET_LENGTH // the max size of
#define MAX_DATA_SIZE MAX_PACKET_SIZE-HEAD_PACKET_LENGTH // the max size of the data
#define INT_DATA_LENGTH 4




// difinition of the packet
typedef struct {
  unsigned char cmd;
  unsigned short data_length;
  unsigned char data[MAX_DATA_SIZE];
} __attribute__ ((packed)) pckt;
```

```
// difinition of the return_packet
typedef struct {
  unsigned char origin_cmd;
  unsigned char error_code;
  unsigned short data_length;
  unsigned char data[MAX_RETURN_DATA_SIZE];
}  __attribute__ ((packed)) rpckt;

/* Comminucation driver */
void comm_init();
void comm_get_packet(pckt* packet);
void comm_send_packet(rpckt* packet);



//definition of the copy_to_ram / copy_to_flash data format
typedef struct {
  address from_address;
  address to_address;
  unsigned int data_length;
} __attribute__ ((packed)) cmd_copy;

// Upload
typedef struct {
  address start_address;
  unsigned int data_length;
  unsigned char data[MAX_DATA_SIZE-ADDRESS_LENGTH-INT_DATA_LENGTH];
} __attribute__ ((packed)) cmd_upload;

//definition of the ramtest data format
typedef struct {
  address start_address;
  unsigned int data_length;
} __attribute__ ((packed)) cmd_ramtest;

//definition of the download data format
typedef struct {
  address start_address;
  unsigned short data_length;
} __attribute__ ((packed)) cmd_download;

//definition of the ramtest return data format
typedef struct {
```

```
  address failure; //The address where the ram-failuer is
} __attribute__ ((packed)) return_ramtest;

typedef struct {
  address start_address;
  unsigned int data_length;
} __attribute__ ((packed)) cmd_cksum;


typedef struct {
  unsigned char sysinfo[251];
}  __attribute__ ((packed)) cmd_status;


typedef struct {
  address addr;
} __attribute__ ((packed)) cmd_exe;

typedef struct {
  checksum cs;
} __attribute__ ((packed)) return_cksum;

typedef struct {
  address delete_address;
} __attribute__ ((packed)) cmd_delflash;



//-------commands
#define REBOOT 1
void reboot();

#define RAM_TEST 2
void ram_test(cmd_ramtest* cmd, rpckt* return_packet);

#define COPY_TO_FLASH 3
void copy_to_flash(cmd_copy* cmd, rpckt* return_packet);

#define COPY_TO_RAM 4
void copy_to_ram(cmd_copy* cmd, rpckt* return_packet);

#define GET_STATUS 5
void get_status(sysinfo_block* sysinfo, rpckt* return_packet);
```

```
#define SEND_STATUS 6
void send_status(cmd_status* cmd, sysinfo_block* sysinfo, rpckt* return_packet);

#define EXE 7
void execute_address(cmd_exe* cmd, rpckt* return_packet);

#define GET_CHECK_SUM 8
void get_check_sum( cmd_cksum* cmd, rpckt* return_packet);

#define UPLOAD 9
void upload(cmd_upload* cmd, rpckt* return_packet);

#define DOWNLOAD 10
void download(cmd_download* cmd, rpckt* return_packet);

#define DELETE_FLASH_BLOCK 11
void delete_flash_block(cmd_delflash* cmd, rpckt* return_packet);

#endif
```

C.6 crc.c

```
/* This implementation of the 32-bit crc
 * algorithm is a modyfied version of the
 * implementation pressented in "Programming
 * Embedded Systems" by Michael Barr
 *
 * Boot-group
 * 32-bit crc checksum computation
 */



#include "system.h"

#define POLYNOMIAL 0x04C11DB7
#define INITIAL_REMAINDER 0xFFFFFFFF
#define FINAL_XOR_VALUE 0xFFFFFFFF


/*
CCIT:
Divisor 0x1021
Remainder 0xFFFF
Final XOR value 0x0000

CRC16
Divisor 0x8005
Remainder 0x0000
Final XOR value 0x0000

CRC32
Divisor 0x04C11DB7
Remainder 0xFFFFFFFF
Final XOR value 0xFFFFFFFF
*/



#define WIDTH (8*sizeof(checksum))
#define TOPBIT (1<< (WIDTH-1))
```

```
/*
 Should not be included
*/

#ifdef INCLUDE_CRCINIT
void crcInit()
{
checksum remainder;
checksum 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 the polynomail */
for (bit=0;bit<8;bit++)
{

/* Ty 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

checksum crcCompute(address message, unsigned int nBytes)
```

```
{

unsigned int offset;
unsigned char byte;
checksum remainder = INITIAL_REMAINDER;

/* Divide the message by the polynomail, a bit 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()*/
```

C.7 data.c

```
/* The remainder of every possible byte selection (the result of crcInit() ) */

#include "system.h"

const checksum crcTable[256] = {
  0x00000000,0x04c11db7,0x09823b6e,0x0d4326d9,0x130476dc,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,0x164f8078,0x1b0ca6a1,0x1fcdbb16,0x018aeb13,0x054bf6a4,
  0x0808d07d,0x0cc9cdca,0x7897ab07,0x7c56b6b0,0x71159069,0x75d48dde,
  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,0x0315d626,0x07d4cb91,0x0a97ed48,0x0e56f0ff,
  0x1011a0fa,0x14d0bd4d,0x19939b94,0x1d528623,0xf12f560e,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,
  0x18197087,0x1cd86d30,0x029f3d35,0x065e2082,0x0b1d065b,0x0fdc1bec,
  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};

const char zeroTable[256] = {
  0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
  0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
  0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
  0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
  0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
  0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
  0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
  0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
  0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
  0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
  0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
  0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
  0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
  0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
  0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,
  0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0,0x0};
```

C.8 flash_driver.c

```
/* flash driver
 * see Am29LV017D data sheet, page 21 for details
 */

#include "system.h"

#define FLASH_CODE_FIRST 0xAA
#define FLASH_CODE_SECOND 0x55
#define FLASH_SECTOR_ERASE_FIRST        0x80
#define FLASH_SECTOR_ERASE_SECOND       0x30
#define FLASH_UNLOCK_BYPASS 0x20
#define FLASH_UNLOCK_PROGRAM            0xA0
#define FLASH_UNLOCK_RESET_FIRST        0x90
#define FLASH_UNLOCK_RESET_SECOND       0x00


#define OK 0
#define FLASH_TIMEOUT 34

int flash_erase_sector(address flash) {
  volatile char data;
  int i;

  /* Erase sector */
  *flash = FLASH_CODE_FIRST;
  *flash = FLASH_CODE_SECOND;
  *flash = FLASH_SECTOR_ERASE_FIRST;
  *flash = FLASH_CODE_FIRST;
  *flash = FLASH_CODE_SECOND;
  *flash = FLASH_SECTOR_ERASE_SECOND;

  for(i=0;i<1000000;i++) {
      data = *flash;
      if ((data & 0x80) == 0x80 ) break; /* program ok */
      if (data & 0x20) {
data = *flash;
if ((data & 0x80) == 0x80) break; /* program ok */
return  FLASH_TIMEOUT;
      }
    }

    return OK;
```

```
}

/* Returns OK or the address where the timeout occured */
int flash_write(address flash, const address mem,  int len) {
  int i,j;
  volatile char data;

  /* PROGRAM chip */
  /* Am29LV017D data sheet - page 21 */

  *flash = FLASH_CODE_FIRST;
  *flash = FLASH_CODE_SECOND;
  *flash = FLASH_UNLOCK_BYPASS;

  for (i=0; i<len; i++) {
    *flash = FLASH_UNLOCK_PROGRAM;
    flash[i] = mem[i];
    for(i=0;i<1000000;i++) {
      data = flash[i];
      if ((0x80 & data) == (0x80 & mem[i])) break; /* write ok */
      if (data & 0x20) {
data = flash[i];
if ((0x80 & data) == (0x80 & mem[i])) break; /* write ok */
return  i + (int) *flash; /* write failed */
      }
    }
  }

  *flash = FLASH_UNLOCK_RESET_FIRST;
  *flash = FLASH_UNLOCK_RESET_SECOND;

  return OK;
}
```

C.9 init.S

```
# OBC initialization
# Boot-group


/* at91 register pointers */
.set EBI_BASE, 0xFFE00000
.set WD_BASE,  0xFFFF8000
.set PIO_BASE, 0xFFFF0000

/* memory to be tested */
.set RAM0_BASE,0x0
.set RAM0_LENGTH,0x2000
.set RAM1_BASE,0x02000000
.set RAM1_LENGTH,0x00100000

/* Memory required by c-stack */
.set STACK_SIZE,0x400

.align 4
.global __reset

.section .text

__reset:
b reset /* reset */
undefvec:
b reset /* Undef */
swivec:
b reset /* SW */
pabtvec:
b reset /* P abt */
dabtvec:
b reset /* D abt */
rsvdvec:
b reset /* reserved */
irqvec:
b reset /* irq */
fiqvec:
b reset /* fiq */

reset:
/* Debug code  - turns on P0 and P4 */
```

```
ldr r2, PtPIO_BASE
mov r3, #0xFF
str r3, [r2]
str r3, [r2, #0x10]
str r3, [r2, #0x30]
mov r3, #0X11
str r3, [r2, #0x34]

/* Disable interrupts in processor core */
#msr CPSR_f, #0xdf

/* Start wacthdog */
#ldr r10, PtWDTable /* get the address of the watchdog image */
# ldmia r10, {r0-r3} /* Load data to registers */
# str r1, [r3, #4] /* Setup Clock Mode Register */
# str r0, [r3] /* Enable watchdog */
# str r2. [r3, #8] /* Reset watchdog timer */
/* 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

ldr r12, PtDoMemTest /* load next function address */

/* Copy Chip Select Register Image to Memory Controller and command remap */
ldmia r10!, {r0-r9,r11} /* 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 */


PtPIO_BASE:
.long PIO_BASE

PtDoMemTest:
.long DoMemTest

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

```
EBITable:
.long 0x01002F3e /* 0x01000000 ---*/
.long 0x03002F3e /* 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 0x00000006 /* 7 memory regions, standard read */
.long EBI_BASE /* EBI Base address */

PtMEMTable:
.long MEMTable

MEMTable:
.long RAM0_BASE /* r4 */
.long RAM0_LENGTH-1 /* r5 */
.long RAM1_BASE /* r6 */
.long RAM1_LENGTH-1 /* r7 */
.long 0x00000000 /* r8 */
.long 0xFFFFFFFF /* r9 */
.long STACK_SIZE /* r10 */

/* Do simple mem test to find space for stack */
DoMemTest:
/* Debug code - turns on P1 and P5 */
ldr r2, PtPIO_BASE
mov r3, #0x22
str r3, [r2, #0x34]

/* test */
ldr r1, mem
b CRTSetup

/* Load memory variables into registers */
ldr r11, PtMEMTable
ldmia r11, {r4-r10}

mov r0, r4
add r1, r4, r5
mov r2, r1
bl Write /* Test sram */
```

```
mov r0, r6
add r1, r6, r7
mov r2, r1
bl Write /* Test ram */

b DoPANIC /* No memory available */

mem: .long 0x02080000

Write: /* r0=base, r1=higest valid address,
 * r2=pointer to actual address,
 * r3= tmp */

str r9, [r2] /* save HIGH in mem */
ldr r3, [r2] /* load again */
  cmp r9, r3   /* test */
bne Fail
str r8, [r2] /* save LOW in mem */
ldr r3, [r2] /* load 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 */
b Write
Fail: mov r1, r2
b Write


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

WDTable:
.long  0x0000373F /* Set timer to approx. 6 secs. */
.long  0x00002343 /* Enable watchdog timer */
.long  0x0000C071 /* Reset watchdog timer */
.long  WD_BASE /* WD Base address */

CRTSetup:
mov r3, r1 /* setup stackpointers and registers */
mov sp, r3
```

```
sub sl, sp, #STACK_SIZE
mov r2, #STACK_SIZE
mov fp, #0
mov r7, #0

mov r0, #0
mov r1, #0

bl boot

DoPANIC:
/* PANIC - no memory available
 * Blame it on the hardware guys
 * turns on P5-P8 and loop
 */
ldr r2, PtPIO_BASE
mov r3, #0xFF
str r3, [r2, #0x30]
mov r3, #0xF0
str r3, [r2, #0x34]

Loop: b Loop
```

C.10 memtest.c

```
/* This memtest() routine is a modifyed
 * implementation of the routine
 * memTestDevice() pressented in "Programming
 * eEmbedded Systems" by Michael Barr
 *
 * Boot-strap group
 * memtest routine
 */

#include "system.h"

address memtest(int *baseAddress, unsigned long nBytes)
{

unsigned long offset;
unsigned long nWords = nBytes/sizeof(int);

        unsigned char pattern;
unsigned char 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 sencond pass.
*/

for (pattern=1, offset=0;offset<nWords; pattern++, offset++)
{
if (baseAddress[offset] != pattern)
{
return ((address)&baseAddress[offset]);
}
antipattern = ~pattern;
baseAddress[offset]=antipattern;
}
```

```
/*
* Check each location for the inverted pattern an zero it
*/

for (pattern =1, offset=0; offset<nWords;pattern++, offset++)
{
antipattern= ~pattern;
if (baseAddress[offset]!=antipattern)
{
return ((address)&baseAddress[offset]);
}
baseAddress[offset]=0;
}

return (0);  /* NULL */
} /* memtest */
```

C.11 sysinfo.c

```
/*
  DTUSAT - boot
  sysinfo.c

  Functions to read and save sysinfo

*/
#include "system.h"

void sysinfo_load(sysinfo_block* sys_p) {
  sysinfo_block *sysinfo;
  int i;
  /* Find sysinfo block */
  sysinfo = (sysinfo_block*) FLASH_BASE;
  for (i = 31; i >= 0; i--) {
    /* check magic number and that sysinfo_block point to it self */
    if ((sysinfo[i]).magic_number == SYSINFO_MAGIC_NUMBER &&
(sysinfo[i]).sysinfo_block == i) {
      /* calculate checksum */
      if (sysinfo[i].checksum == crcCompute((unsigned char*) &sysinfo[i],252)) {
/* Copy sysinfo to mem */
*sys_p  = sysinfo[i];

/* check flash */
if (!flash_erase_sector( (address) &sysinfo[i]) &&
    !flash_write( (address) &sysinfo[i], (address) &zeroTable, 256))
  sys_p->err = OK;
return;
      }
    }
  }
  if (i<0) sys_p->err=SYSINFO_LOAD_ERROR;
}

void sysinfo_save(sysinfo_block* sys_p) {
  /* calculate checksum */
  sys_p->checksum = crcCompute( (address) sys_p,252);
  sys_p->err = OK;
  /* delete and write a block */
  if (flash_erase_sector( (address) FLASH_BASE) ||
      flash_write( (address) (FLASH_BASE + sys_p->sysinfo_block * 0x100) , (address)
    sys_p->err=SYSINFO_WRITE_ERROR;
```

```
    return;
  }
  return;
}

void sysinfo_init(sysinfo_block* sys_p) {
  int i;
  sysinfo_block* flash;

  flash = (sysinfo_block*) FLASH_BASE;
  if (flash_erase_sector( (address) flash)) {
    sys_p->err = SYSINFO_INIT_WRITE_ERROR;
    return;
  }
  i=0;
  while(flash_write( (address) &flash[i], (address) &zeroTable, 256))
    i++;

  if (i == 32) {
    sys_p->err = SYSINFO_INIT_WRITE_ERROR;
    return;
  }

  /* Make default system information in memory */
  memcpy( (address) sys_p, (address) &zeroTable, 256);

  sys_p->magic_number = SYSINFO_MAGIC_NUMBER;
  sys_p->sysinfo_block = i;
  sys_p->err = SYSINFO_INIT;
  sys_p->os_comm = OS_DO_NOTHING;
}
```

C.12 system.c

```
#include "system.h"

inline void arm_execute(address entrypoint) {
  asm volatile ("mov pc,%0" :: "r" (entrypoint));
}

inline void arm_reboot() {
  /* cancel remap - p.47 */
  asm volatile ("mov r0,#0xE3" ::);
  asm volatile ("msr CPSR_c,r0" ::);
  *(address)(0xFFE00020) = 0x1;
  asm volatile ("mov pc,#0" ::);
}


void memcpy(address s1, const address s2, int n) {
  for (n--;n>=0;n--)
    s1[n] = s2[n];
}



/* Visual Debug Device */

void led_on(int what) {
*(volatile int*)(PIO_CODR) = what;
}

void led_off(int what) {
*(volatile int*)(PIO_SODR) = what;
}

void led_init() {
*(volatile int*)(PIO_PER) = LED_ALL;
*(volatile int*)(PIO_OER) = LED_ALL;
}

void led_wait() {
  volatile int i = 10000;
  while (i) i--;
}
```

```
void led_blink() {
  int led;
  for (led=1;led < 257; led = led*2) {
    led_off(LED_ALL);
    led_on(led);
    led_wait();
      }
  led_off(LED_ALL);
}
```

C.13 system.h

```
/* Header file for boot software */

#ifndef __system_h
#define __system_h


/* Datatypes */
typedef volatile unsigned int at91_reg; /* CPU REGISTER */
typedef volatile unsigned char* address; /* Address */
typedef unsigned int checksum;  /* 32 bits checksums */

/* Type for flash blocks */
typedef unsigned char flash_id;


/* struct for data module */
typedef struct {
  unsigned char id;
  unsigned char module_status;
  unsigned short load_length;
  address load_addr;
  checksum checksum;
}  __attribute__ ((packed)) flash_data_module;

/* struct for sysinfo block */
typedef struct {
  unsigned long magic_number;  /* Magic number 4 bytes */
  unsigned char bootcount;          /* Count how many times we have rebooted 1 byte
  unsigned char os_comm;          /* the OS can communicate with the boot software
       using this variable 1 byte */
  unsigned char err;                /* Last error 1 byte */
  unsigned char sysinfo_block;     /* The number of the current
       sys_info block 1 byte */
  unsigned char sysinfo[12]; /* Space for system information 12 bytes */
  address sysinfo_os_pointer;  /* Pointer to the sysinfo block
    to pass on to the os 4 bytes */
  address entrypoint;       /* Ecos entry point 4 bytes */
 flash_data_module data_module[16]; /* 192 bytes */
  flash_id block[32]; /* 32 bytes */
  checksum checksum;          /* Checksum of the entire sys_info block,
       without the checksum */
}  __attribute__ ((packed)) sysinfo_block;
```

```
/* Functions */
/* Comminucation driver
void comm_init();
void comm_get_packet(pckt* packet);
void comm_send_packet(rpckt* packet);
*/

/* crc.c */
checksum crcCompute(address message, unsigned int nBytes); /* Compute checksum */

/* system.c */
void arm_execute(address addr); /* execute code at addr */
void arm_reboot(); /* reboot computer */
void memcpy(address s1, const address s2, int n); /* standard memcopy function */

/* sysinfo.c */
void sysinfo_init(sysinfo_block*); /* Find space in flash, and create empty system informati
void sysinfo_load(sysinfo_block*); /* Try to find valid system information */
void sysinfo_save(sysinfo_block*); /* Save system information to flash */

/* boot.c */
void boot(); /* C boot program */

/* memtest.c */
address memtest(int* baseAddress, unsigned long nBytes);

/* cmd_handler.c */
void failsafe(sysinfo_block*);

/* flash_driver.c */
int flash_erase_sector(address flash);
int flash_write(address flash, address mem,  int len);

/* watchdog.c */
void wd_start();
void wd_touch();

/* Other stuff */
/* The remainder of every possible byte selection (the result of crcInit() ) */
extern const checksum crcTable[256];
extern const char zeroTable[256];
```

```
/* OS communication
   Commands the OS can give to the boot software
*/
#define OS_DO_NOTHING 0xFF /* Default value, does nothing  */
#define OS_RESET_BOOT_COUNTER 0x8F /* Reset boot counter. Should be set when the OS
#define OS_FORCE_FAILSAFE 0x01 /* Force the boot software into failsafa mode */


/* Defination */
#define MAX_BOOT_COUNT 4 /* How many times to boot before we go into failsafe */
#define FLASH_BASE 0x03000000 /* Base of flash memory */

#define SYSINFO_MAGIC_NUMBER 0x100879C7 /* Magic number for system information */
#define MODULE_LOAD 3 /* Load data module at boot */


/* Errors */
#define OK 0

#define SYSINFO_LOAD_ERROR 1  /* Could not find valid system information */
#define SYSINFO_WRITE_ERROR 2 /* Could not save system information to flash */
#define SYSINFO_INIT_WRITE_ERROR 3 /* Could not create new system information in fla
#define SYSINFO_INIT 4        /* We have been forced to create neew system informati
#define COPY_TO_RAM_ERROR 5 /* Error while copy date to memory */
#define MISSING_MODULE_ERROR 6 /* One or more data modules could not be found */



/* Magic number */
#define FLASH_MAGIC_NUMBER 0xC7 /* Skal det være noget smartere? */

#define FLASH_LOAD_MODULE 0x2E /* Skal modulet loades under opstart*/

/* ID of flash blocks */
#define FLASH_ID_RESERVED 0xFF /* smart ved reboot under flashoperation */
#define FLASH_ID_UNUSED 0x8F /* også smart da sysinfo kan skrives med det samme, og


/* Visual Debug Device Interface Control */
#define PIO_BASE   0xFFFF0000
#define PIO_PER       PIO_BASE + 0x00
#define PIO_OER      PIO_BASE + 0x10
#define PIO_SODR      PIO_BASE + 0x30
```

```
#define PIO_CODR      PIO_BASE + 0x34

#define LED0         0x01
#define LED1 0x02
#define LED2 0x04
#define LED3 0x08
#define LED4 0x10
#define LED5 0x20
#define LED6 0x40
#define LED7 0x80
#define LED_ALL 0xFF

void led_on(int what);
void led_off(int what);
void led_init();
void led_wait();
void led_blink();


#endif /* __system_h */
```

C.14 usart.h

```
// This file is copied from the Atmel AT91 Library V2.0

//*----------------------------------------------------------------------------
//*         ATMEL Microcontroller Software Support  -  ROUSSET  -
//*----------------------------------------------------------------------------
//* The software is delivered "AS IS" without warranty or condition of any
//* kind, either express, implied or statutory. This includes without
//* limitation any warranty or condition with respect to merchantability or
//* fitness for any particular purpose, or against the infringements of
//* intellectual property rights of others.
//*----------------------------------------------------------------------------
//* File Name           : usart.h
//* Object              : USART Header File.
//*
//* 1.0 01/04/00 JCZ    : Creation
//*----------------------------------------------------------------------------

#ifndef usart_h
#define usart_h

#include "system.h"

/*-----------------------------------------*/
/* USART User Interface Structure Definition */
/*-----------------------------------------*/


typedef struct
{
    at91_reg            US_CR ;         /* Control Register */
    at91_reg            US_MR ;         /* Mode Register */
    at91_reg            US_IER ;        /* Interrupt Enable Register */
    at91_reg            US_IDR ;        /* Interrupt Disable Register */
    at91_reg            US_IMR ;        /* Interrupt Mask Register */
    at91_reg            US_CSR ;        /* Channel Status Register */
    at91_reg            US_RHR ;        /* Receive Holding Register */
    at91_reg            US_THR ;        /* Transmit Holding Register */
    at91_reg            US_BRGR ;       /* Baud Rate Generator Register */
    at91_reg            US_RTOR ;       /* Receiver Timeout Register */
    at91_reg            US_TTGR ;       /* Transmitter Time-guard Register */
    at91_reg            Reserved ;
    at91_reg            US_RPR ;        /* Receiver Pointer Register */
```

```
    at91_reg            US_RCR ;        /* Receiver Counter Register */
    at91_reg            US_TPR ;        /* Transmitter Pointer Register */
    at91_reg            US_TCR ;        /* Transmitter Counter Register */
} StructUSART;

/*--------------------------*/
/* US_CR : Control Register */
/*--------------------------*/

#define US_RSTRX                0x0004      /* Reset Receiver */
#define US_RSTTX                0x0008      /* Reset Transmitter */
#define US_RXEN                 0x0010      /* Receiver Enable */
#define US_RXDIS                0x0020      /* Receiver Disable */
#define US_TXEN                 0x0040      /* Transmitter Enable */
#define US_TXDIS                0x0080      /* Transmitter Disable */
#define US_RSTSTA               0x0100      /* Reset Status Bits */
#define US_STTBRK               0x0200      /* Start Break */
#define US_STPBRK               0x0400      /* Stop Break */
#define US_STTTO                0x0800      /* Start Time-out */
#define US_SENDA                0x1000      /* Send Address */

/*-----------------------*/
/* US_MR : Mode Register */
/*-----------------------*/

#define US_CLKS                 0x0030      /* Clock Selection */
#define US_CLKS_MCK             0x00        /* Master Clock */
#define US_CLKS_MCK8            0x10        /* Master Clock divided by 8 */
#define US_CLKS_SCK             0x20        /* External Clock */
#define US_CLKS_SLCK            0x30        /* Slow Clock */

#define US_CHRL                 0x00C0      /* Byte Length */
#define US_CHRL_5               0x00        /* 5 bits */
#define US_CHRL_6               0x40        /* 6 bits */
#define US_CHRL_7               0x80        /* 7 bits */
#define US_CHRL_8               0xC0        /* 8 bits */

#define US_SYNC                 0x0100      /* Synchronous Mode Enable */

#define US_PAR                  0x0E00      /* Parity Mode */
#define US_PAR_EVEN             0x00        /* Even Parity */
#define US_PAR_ODD              0x200       /* Odd Parity */
#define US_PAR_SPACE            0x400       /* Space Parity to 0 */
#define US_PAR_MARK             0x600       /* Marked Parity to 1 */
```

```
#define US_PAR_NO               0x800       /* No Parity */
#define US_PAR_MULTIDROP        0xC00       /* Multi-drop Mode */

#define US_NBSTOP               0x3000      /* Stop Bit Number */
#define US_NBSTOP_1             0x0000      /* 1 Stop Bit */
#define US_NBSTOP_1_5           0x1000      /* 1.5 Stop Bits */
#define US_NBSTOP_2             0x2000      /* 2 Stop Bits */

#define US_CHMODE                   0xC000  /* Channel Mode */
#define US_CHMODE_NORMAL            0x0000  /* Normal Mode */
#define US_CHMODE_AUTOMATIC_ECHO    0x4000  /* Automatic Echo */
#define US_CHMODE_LOCAL_LOOPBACK    0x8000  /* Local Loopback */
#define US_CHMODE_REMOTE_LOOPBACK   0xC000  /* Remote Loopback */

#define US_MODE9                0x20000     /* 9 Bit Mode */

#define US_CLKO                 0x40000     /* Baud Rate Output Enable */

/* Mode Register model */

/* Standard Asynchronous Mode : 8 bits , 1 stop , no parity */
#define US_ASYNC_MODE ( US_CHMODE_NORMAL + \
                        US_NBSTOP_1 + \
                        US_PAR_NO + \
                        US_CHRL_8 + \
                        US_CLKS_MCK )

/* Standard External Asynchronous Mode : 8 bits , 1 stop , no parity */
#define US_ASYNC_SCK_MODE ( US_CHMODE_NORMAL + \
                            US_NBSTOP_1 + \
                            US_PAR_NO + \
                            US_CHRL_8 + \
                            US_CLKS_SCK )

/* Standard Synchronous Mode : 8 bits , 1 stop , no parity */
#define US_SYNC_MODE ( US_SYNC + \
                       US_CHMODE_NORMAL + \
                       US_NBSTOP_1 + \
                       US_PAR_NO + \
                       US_CHRL_8 + \
                       US_CLKS_MCK )

/* SCK used Label */
#define SCK_USED (US_CLKO | US_CLKS_SCK)
```

```
/*-------------------------------------------------------------------*/
/* US_IER, US_IDR, US_IMR, US_IMR: Status and Interrupt Register */
/*-------------------------------------------------------------------*/

#define US_RXRDY            0x1       /* Receiver Ready */
#define US_TXRDY            0x2       /* Transmitter Ready */
#define US_RXBRK            0x4       /* Receiver Break */
#define US_ENDRX            0x8       /* End of Receiver PDC Transfer */
#define US_ENDTX            0x10       /* End of Transmitter PDC Transfer */
#define US_OVRE             0x20       /* Overrun Error */
#define US_FRAME            0x40       /* Framing Error */
#define US_PARE             0x80       /* Parity Error */
#define US_TIMEOUT          0x100       /* Receiver Timeout */
#define US_TXEMPTY          0x200       /* Transmitter Empty */

#define US_MASK_IRQ_TX      (US_TXRDY | US_ENDTX | US_TXEMPTY)
#define US_MASK_IRQ_RX      (US_RXRDY | US_ENDRX | US_TIMEOUT)
#define US_MASK_IRQ_ERROR   (US_PARE | US_FRAME | US_OVRE | US_RXBRK)



#endif /* usart_h */
```

C.15 watchdog.c

```
/*

watchdog timer interface

*/

#include "system.h"
#include "wd.h"

#define WD_BASE 0xFFFF8000


StructWD* const wd_base = (StructWD *) WD_BASE;

inline void wd_start() {
  wd_base->WD_CMR = ( WD_CKEY | WD_HPCV | WD_WDCLKS_MCK1024 );
  wd_base->WD_OMR = ( WD_OKEY | WD_RSTEN | WD_WDEN );
  wd_base->WD_CR = WD_RSTKEY;
}

inline void wd_touch() {
  wd_base->WD_CR = WD_RSTKEY;
}
```

C.16 wd.h

```
// This file is copied from the Atmel AT91 Library V2.0

//*----------------------------------------------------------------------------
//*         ATMEL Microcontroller Software Support  -  ROUSSET  -
//*----------------------------------------------------------------------------
//* The software is delivered "AS IS" without warranty or condition of any
//* kind, either express, implied or statutory. This includes without
//* limitation any warranty or condition with respect to merchantability or
//* fitness for any particular purpose, or against the infringements of
//* intellectual property rights of others.
//*----------------------------------------------------------------------------
//* File Name           : wd.h
//* Object              : Watch Dog Header File.
//*
//* 1.0 01/04/00 JCZ    : Creation
//* 1.1 10/01/02 PFi : Conditional Compilation added in StructWD for AT91M55800
//*----------------------------------------------------------------------------

#ifndef wd_h
#define wd_h

#include    "system.h"

/*------------------------------------------------*/
/*  Watch Dog User Interface Structure Definition */
/*------------------------------------------------*/
typedef struct
{
    at91_reg        WD_OMR ;        /* Overflow Mode Register */
    at91_reg        WD_CMR ;        /* Clock Mode Register */
    at91_reg        WD_CR ;         /* Control Register */
    at91_reg        WD_SR ;         /* Status Register */
    at91_reg        Reserved ;
    at91_reg        WD_TLR ;        /* Test Load Register : test purpose only */
} StructWD ;

/*--------------------------------------------------------*/
/*  WD_OMR: Watch Dog Overflow Mode Register Bits Definition */
/*--------------------------------------------------------*/

#define WD_WDEN             0x1         /* Watch Dog Enable */
#define WD_RSTEN            0x2         /* Internal Reset Enable */
```

```
#define WD_IRQEN            0x4         /* Interrupt Enable */
#define WD_EXTEN            0x8         /* External Signal Enable */
#define WD_OKEY             0x2340      /* Overflow Mode Register Access Key */

/*-----------------------------------------------------*/
/* WD_CMR: Watch Dog Clock Register Bits Definition */
/*-----------------------------------------------------*/

#define WD_WDCLKS           0x3         /* Clock Selection */
#define WD_WDCLKS_MCK8      0x0
#define WD_WDCLKS_MCK32     0x1
#define WD_WDCLKS_MCK128    0x2
#define WD_WDCLKS_MCK1024   0x3

#define WD_HPCV             0x3C        /* High Preload Counter Value */

#define WD_CKEY             (0x06E<<7)  /* Clock Register Access Key */

/*-----------------------------------------------------*/
/* WD_CR: Watch Dog Control Register Bits Definition */
/*-----------------------------------------------------*/

#define WD_RSTKEY           0xC071      /* Watch Dog Restart Key */

/*-----------------------------------------------------*/
/* WD_SR: Watch Dog Status Register Bits Definition */
/*-----------------------------------------------------*/

#define WD_WDOVF            0x1         /* WatchDog Overflow Status */

/*-----------------------------------------------------*/
/* WD_TLR: Test Load Register for test purpose only */
/*-----------------------------------------------------*/

#define WD_TMRKEY           0xD64A0000  /* Access Key */
#define WD_TESTEN           0x2         /* Test Mode Enable */


#endif /* wd_h */
```

Appendix D

Source files for test

These are the test programs used for test purpose in the boot-strap project.

D.1 cmd_test_driver.c

```
/*
 test
*/


#include "cmd_handler.h"
//#include "cmd_handler.c"
#include "system.h"
//#include "testing/ligth.c"



//test_properties
#define RAM_TEST_ADDRESS 0x02040000
#define RAM_TEST_DATA_LENGTH  4

#define COPY_TO_FLASH_FROM_ADDR 0x02020000
#define COPY_TO_FLASH_TO_ADDR 0x3020000
#define COPY_TO_FLASH_DATA_LENGTH 2

#define COPY_TO_RAM_FROM_ADDR 0x02020000
#define COPY_TO_RAM_TO_ADDR 0x02040000
#define COPY_TO_RAM_DATA_LENGTH 2

#define TEST_SYS_BOOTCOUNT  3
#define TEST_SYS_OS_COMM 23
```

```
#define TEST_SYS_ERR 1
#define TEST_SYS_SYSINFO_BLOK 2
#define TEST_SYS_MAGIC_NUMBER SYSINFO_MAGIC_NUMBER

#define TEST_GET_CHECK_SUM_ADDRESS 0x02020000;
#define TEST_GET_CHECK_SUM_DATA_LENGTH 4;

#define TEST_EXE_ADDRESS 0x0000000 //FIXME

#define TEST_UPLOAD_START_ADDRESS 0x02050000 //FIXME

#define  TEST_DOWNLOAD_START_ADDRESS TEST_UPLOAD_START_ADDRESS //0x02020000 //FIXME
#define  TEST_DOWNLOAD_DATA_LENGTH 3;

#define TEST_DELETE_FLASH_BLOCK_DELETE_ADDRESS 0x03020000 // FIXME

//#define TEST_SYS_SYSINFO;

#define PROG_ARRAY_SIZE 368

unsigned char prog_array[PROG_ARRAY_SIZE] = {
  48,48,159,229,3,208,160,225,2,172,77,226,0,16,160,227,1,176,160,225,1,112,160,225,
  159,229,20,32,159,229,0,32,66,224,0,0,160,227,0,16,160,227,73,0,0,235,2,0,0,235,1
  4,2,112,177,4,2,14,240,160,225,254,255,255,234,13,192,160,225,0,216,45,233,4,176,7
  4, 208,77,226,16,0,11,229,255,60,224,227,203,48,67,226,16,32,27,229,0,32,131,229,0
  233,13,192,160,225,0,216,45,233,4,176,76,226,4,208,77,226,16,0,11,229,255,60,224,2
  48,67,226,16,32,27,229,0,32,131,229,0,168,27,233,13,192,160,225,0,216,45,233,4,176
  2,49,160,227,195,55,160,225,255,32,160,227,0,32,131,229,255,60,224,227,239,48,67,2
  131,229,0,168,27,233,13,192,160,225,0,216,45,233,4,176,76,226,4,208,77,226,97,59,1
  42,62,131,226,16,48,11,229,16,48,27,229,0,0,83,227,0,0,0,26,3,0,0,234,16,48,27,229
  226,16,48,11,229,247,255,255,234,0,168,27,233,13,192,160,225,0,216,45,233,4,176,76
  77,226,1,48,160,227,16,48,11,229,16,48,27,229,1,12,83,227,0,0,0,218,8,0,0,234,255,
  206,255,255,235,16,0,27,229,194,255,255,235,224,255,255,235,16,48,27,229,131,48,16
  11,229,242,255,255,234,255,0,160,227,197,255,255,235,0,168,27,233,13,192,160,225,0
  4,176,76,226,180,255,255,235,228,255,255,235,253,255,255,234};



pckt packet_list[12];
int packet_pointer = 11;

//prototype of the init_test_data
void init_test_data();
//prototype of the cmd_func_test_flow
```

```
void cmd_func_test_flow();

void comm_init() {
  //init_test_data();
  cmd_func_test_flow();
}

int test_upanddown_load(){
  /*******Test upload and download*****************
  Testing the upload and download function....
  use the upload function to store some data form a address and forward
  use the download function to download the data form the same address and forward
  I the downloaded data corresponds to the uploaded software the both methods works.
  Of cause there could be a writing error and a similar reading error and test will still re
  But using the Insight/GDB debugger, givs the option to see an address' value, at we will t
  */
  //define UPLOAD 9
  //void upload(cmd_upload* cmd, rpckt* return_packet);

  cmd_upload cmdupload;
  cmdupload.data_length = 3;
  cmdupload.start_address = (address)TEST_UPLOAD_START_ADDRESS;
  unsigned char uploadData[3] = {1,3,5};
  memcpy((address)&cmdupload.data, (address)&uploadData, 3 );

  rpckt upload_return_packet;
  upload(&cmdupload, &upload_return_packet);

  //define DOWNLOAD 10
  //void download(cmd_download* cmd, rpckt* return_packet);

  cmd_download cmddownload;
  cmddownload.start_address = (address)TEST_DOWNLOAD_START_ADDRESS;
  cmddownload.data_length = TEST_DOWNLOAD_DATA_LENGTH;

  rpckt download_return_packet;
  download(&cmddownload, &download_return_packet);
  unsigned char  err_code = download_return_packet.error_code;
  unsigned short datalength;


  switch(err_code){
  case (NO_ERROR):
    datalength = (unsigned short)download_return_packet.data_length;
```

```
    unsigned char downloaddata[datalength];
    memcpy((address)&downloaddata,(address)&download_return_packet.data,datalength)
    if(downloaddata[0]  == uploadData[0] && downloaddata[1]  == uploadData[1] && dow
      return 123;
    }
    break;
  default:
    //led_off(LED_ALL);
    //led_on(LED_ALL);
  }
  return 0;
}


int test_getandset_status(){
  /************Test get status and set status*****************/
  /*
    Get the status and checking that the recieved data corresponds to the init data.
    Send some new sattus values, and get them agin. Check that tha system valuse cor
    Rember to check the address value, using the debugger, to prevent the read- writ

  */
  sysinfo_block sysinfo;
  sysinfo.magic_number = TEST_SYS_MAGIC_NUMBER;
  sysinfo.bootcount = TEST_SYS_BOOTCOUNT;
  sysinfo.os_comm = TEST_SYS_OS_COMM;
  sysinfo.err = TEST_SYS_ERR;
  sysinfo.sysinfo_block = TEST_SYS_SYSINFO_BLOK;
  sysinfo_save(&sysinfo);


  //define GET_STATUS 5
  //void get_status(sysinfo_block* sysinfo, rpckt* return_packet);
  rpckt sysinfo_return_packet;
  get_status(&sysinfo, &sysinfo_return_packet);

  unsigned char  err_code = (int)sysinfo_return_packet.error_code;
  sysinfo_block  sysinfo_return;
  unsigned int datalength;

  switch(err_code){
  case (NO_ERROR):
    datalength = (unsigned short)sysinfo_return_packet.data_length;
    memcpy( ((address) (&sysinfo_return)) +1, (address) &sysinfo_return_packet.data
```

```
    if(  sysinfo_return.bootcount  ==  sysinfo.bootcount  && sysinfo_return.os_comm==sysinfo
      //define SEND_STATUS 6
      //void send_status(cmd_status* cmd, sysinfo_block* sysinfo, rpckt* return_packet);
      sysinfo_block sys;
      memcpy((address)&sys,(address)&sysinfo,256);

      sys.bootcount  = 3;
      sys.os_comm = 21;

      cmd_status sys_packet;
      memcpy((address) &sys_packet.sysinfo,( (address)&sys )+1, 251);
      rpckt sys_return_packet;
      send_status(&sys_packet, &sysinfo, &sys_return_packet);

      // GET_STATUS 5
      rpckt sysinf_return_packet;
      get_status(&sysinfo, &sysinf_return_packet);
      unsigned char  error_code = (int)sysinfo_return_packet.error_code;
      sysinfo_block  sysinf_return;
      switch(error_code){
      case (NO_ERROR):
memcpy( ((address) (&sysinf_return)) +1, (address) &sysinf_return_packet.data, 251);
if( sysinf_return.bootcount == 3 && sysinf_return.os_comm == 21 ){
  return 34;
}
break;
      default:

      }
      break;
    default:

    }
  }
  return 0;
}

void cmd_func_test_flow(){
  if( test_upanddown_load() &&  test_getandset_status() ){
    led_off(LED_ALL);
    led_blink();
    led_blink();
    //led_on(LED3);
    //led_on(LED4);
```

```
    led_off(LED_ALL);
  }else{
    led_off(LED_ALL);
    led_on(LED_ALL);
    led_off(LED_ALL);
  }


  //RAM_TEST 2 tested in overall functions
  //void ram_test(cmd_ramtest* cmd, rpckt* return_packet);

  //define GET_CHECK_SUM 8 tested in overall functions
  //void get_check_sum( cmd_cksum* cmd, rpckt* return_packet);

  //define COPY_TO_FLASH 3 tested in overall functions
  //void copy_to_flash(cmd_copy* cmd, rpckt* return_packet);

  //define COPY_TO_RAM 4 tested in
  //void copy_to_ram(cmd_copy* cmd, rpckt* return_packet);

  //define EXE 7 ????
  //void execute_address(cmd_exe* cmd, rpckt* return_packet);

  //define DELETE_FLASH_BLOCK 11 teste in overall functions
  //void delete_flash_block(cmd_delflash* cmd, rpckt* return_packet);
}

void comm_get_packet(pckt* packet) {
  int datalength;
  switch(packet_pointer){

  case( 0 ):
    //upload
    packet->cmd = UPLOAD;
    cmd_upload cmdupload;
    cmdupload.data_length = 3;
    cmdupload.start_address = (address)TEST_UPLOAD_START_ADDRESS;
    unsigned char uploadData[3] = {2,9,11};
    memcpy((address)&cmdupload.data, (address)&uploadData, 3 );
    datalength = ADDRESS_LENGTH + INT_DATA_LENGTH + 3;
    memcpy((address)&packet->data,(address)&cmdupload,datalength);
    packet->data_length = datalength;
    break;
```

```
case( 1 ):
  //ram test
  packet->cmd = RAM_TEST;
  cmd_ramtest cmdramtest;
  cmdramtest.start_address = (address)RAM_TEST_ADDRESS;
  cmdramtest.data_length = RAM_TEST_DATA_LENGTH ;
  datalength = ADDRESS_LENGTH+INT_DATA_LENGTH;
  memcpy( (address)&packet->data, (address)&cmdramtest ,datalength);
  packet->data_length = datalength;
  break;

case( 2 ):
  // COPY_TO_FLASH  has to faild
  packet->cmd = COPY_TO_FLASH;
  cmd_copy cmdcopy;
  cmdcopy.from_address=(address)COPY_TO_FLASH_FROM_ADDR;
  cmdcopy.to_address=(address)COPY_TO_FLASH_TO_ADDR;
  cmdcopy.data_length= COPY_TO_FLASH_DATA_LENGTH;
  datalength = ADDRESS_LENGTH+ADDRESS_LENGTH+INT_DATA_LENGTH;
  memcpy((address)&(packet->data),(address)&cmdcopy,datalength);
  packet->data_length = datalength;
  break;

case ( 3 ):
  //  DELETE_FLASH_BLOCK packet
  packet->cmd = DELETE_FLASH_BLOCK;
  cmd_delflash cmddelflash;
  cmddelflash.delete_address = (address)COPY_TO_FLASH_TO_ADDR;
  datalength = ADDRESS_LENGTH;
  memcpy((address)&packet->data,(address)&cmddelflash,datalength);
  packet->data_length=datalength;
  break;

case ( 4 ):
  // COPY_TO_FLASH  has to faild
  packet->cmd = COPY_TO_FLASH;
  cmd_copy cmdcopyT;
  cmdcopyT.from_address=(address)COPY_TO_FLASH_FROM_ADDR;
  cmdcopyT.to_address=(address)COPY_TO_FLASH_TO_ADDR;
  cmdcopy.data_length= COPY_TO_FLASH_DATA_LENGTH;
  datalength = ADDRESS_LENGTH+ADDRESS_LENGTH+INT_DATA_LENGTH;
  memcpy((address)&(packet->data),(address)&cmdcopyT,datalength);
  packet->data_length = datalength;
  break;
```

```
case( 5 ):
  // COPY_TO_RAM packet
  packet->cmd = COPY_TO_RAM;
  cmd_copy cmdcpy;
  cmdcpy.from_address=(address)COPY_TO_RAM_FROM_ADDR;
  cmdcpy.to_address=(address)COPY_TO_RAM_TO_ADDR;
  cmdcpy.data_length= COPY_TO_RAM_DATA_LENGTH;
  datalength = ADDRESS_LENGTH+ADDRESS_LENGTH+INT_DATA_LENGTH;
  memcpy((address)&(packet->data),(address)&cmdcpy,datalength);
  packet->data_length = datalength;
  break;

case( 6 ):
  //  SEND_STATUS packet
  packet->cmd = SEND_STATUS;
  cmd_status sys_packet;
  sysinfo_block sys;
  sys.magic_number = TEST_SYS_MAGIC_NUMBER;
  sys.bootcount = TEST_SYS_BOOTCOUNT;
  sys.os_comm = TEST_SYS_OS_COMM;
  sys.err = TEST_SYS_ERR;
  sys.sysinfo_block = TEST_SYS_SYSINFO_BLOK;
  memcpy((address)&(packet->data),(address)&(sys_packet),251);
  packet->data_length = 251;
  break;

case ( 7 ):
  //  GET_STATUS packet
  packet->cmd = GET_STATUS;
  packet->data_length = 0;
  break;

case ( 8 ):
  //  GET_CHECK_SUM packet
  packet->cmd = GET_CHECK_SUM;
  cmd_cksum cmdcksum;
  cmdcksum.start_address = (address)TEST_GET_CHECK_SUM_ADDRESS;
  cmdcksum.data_length = TEST_GET_CHECK_SUM_DATA_LENGTH;
  datalength = ADDRESS_LENGTH+INT_DATA_LENGTH;
  memcpy((address)&packet->data,(address)&cmdcksum,datalength);
  packet->data_length=datalength;
  break;
```

```
  case ( 9 ):
    //  DOWNLOAD packet
    packet->cmd = DOWNLOAD;
    cmd_download cmddownload;
    cmddownload.start_address = (address)TEST_DOWNLOAD_START_ADDRESS;
    cmddownload.data_length = TEST_DOWNLOAD_DATA_LENGTH;
    datalength = ADDRESS_LENGTH+INT_DATA_LENGTH;
    memcpy((address)&packet->data,(address)&cmddownload,datalength);
    packet->data_length = datalength;
    break;


  case ( 10 ):
    //EXE packet
    packet->cmd = EXE;
    cmd_exe cmdexe;
    cmdexe.addr =  (address)prog_array;
    datalength=ADDRESS_LENGTH;
    memcpy((address)&packet->data, (address)&cmdexe ,datalength);
    packet->data_length = datalength;
    break;

  case ( 11 ):
    //REBOOT packet
    packet->cmd = REBOOT;
    packet->data_length = 0;
    break;

  default:
  }
  packet_pointer++;
}

void comm_send_packet(rpckt* packet) {
  int err_code = (int)packet->error_code;
  switch(err_code){
  case NO_ERROR:
    led_off(LED_ALL);
    led_blink();
    led_blink();
    led_on(LED3);
    led_on(LED4);
    break;
  default:
```

```
    led_off(LED_ALL);
    led_on(LED_ALL);
  }
}
```

D.2 crt0.S

```
/* Sample initialization file */

.extern main
.extern exit

/* .text is used instead of .section .text so it works with arm-aout too.  */
.text
.code 32
.align  0

.global _mainCRTStartup
.global _start
.global start
start:
_start:
_mainCRTStartup:

/* Start by setting up a stack */
/*  Set up the stack pointer to end of bss */
ldr r3, .LC2
mov  sp, r3

sub sl, sp, #512 /* Still assumes 512 bytes below sl */

mov  a2, #0 /* Second arg: fill value */
mov fp, a2 /* Null frame pointer */
mov r7, a2 /* Null frame pointer for Thumb */

ldr a1, .LC1 /* First arg: start of memory block */
ldr a3, .LC2 /* Second arg: end of memory block */
sub a3, a3, a1 /* Third arg: length of block */

mov r0, #0 /*  no arguments  */
mov r1, #0 /*  no argv either */

bl main
bl exit /* Should not return */

/* For Thumb, constants must be after the code since only
positive offsets are supported for PC relative addresses. */

.align 0
```

```
.LC1:
.word __bss_start__
.LC2:
.word __bss_end__
```

D.3 crt_wrapper.c

```
/* wrapper script for c runtime */

void __gccmain(void) {
}

int exit(){
while(1);
}
```

D.4 load_module_test.c

```
/* Testing of the boot load module
 * first we test the sysinfo function
 * and then load a little program into the memory
 * use ligth to debug */

#include "../system.h"

unsigned char prog_array[];
void create_flash_image();

int test_sysinfo_load() {
  /* write 3 sysinfo_blocks to flash
   * s1 has invalid magic_number
   * s2 has invalid checksum
   * s3 is ok, and should be loaded
   */

  sysinfo_block s1, s2, s3, sp, *sys_p;
  address addr, addr2;
  int i, err;

  /* create sysinfo blocks */
  addr = (address) &s1;
  for(i=0; i<256; i++)
    addr[i] = i;


  s2 = s1;
  s2.err = 0; /* sysinfo load set err = 0 */
  s2.sysinfo_block = 4;
  s2.magic_number = SYSINFO_MAGIC_NUMBER;

  s3 = s2;
  s3.sysinfo_block = 2;
  s3.checksum = crcCompute( (address) &s3, 252);

  /* write blocks to flash */

  sys_p = (sysinfo_block*) FLASH_BASE;

  err = flash_erase_sector( (address) sys_p);
if (err) return 1;
```

```
  err = flash_write( (address) &sys_p[2], (address) &s3, 256);
if (err) return 2;
  err = flash_write( (address) &sys_p[4], (address) &s2, 256);
if (err) return 3;
  err = flash_write( (address) &sys_p[9], (address) &s1, 256);
if (err) return 4;

/* call sysinfo load */
  sysinfo_load(&sp);
  if (sp.err) return 5;

  /* test that sp == s3 */
    addr = (address) &s3;
    addr2 = (address) &sp;
    for (i=0; i<256; i++)
      if (addr[i] != addr2[i]) return 6;

  /* test ok */
  return 0;
}

int test_sysinfo_save() {
  /* saves sysinfo block to flash */
  sysinfo_block s1, *sys_p;
  address addr, addr2;
  int i;

  /* write data in sysinfo block */
  addr = (address) &s1;
  for(i=0; i<256; i++)
    addr[i] = i;

  s1.magic_number = SYSINFO_MAGIC_NUMBER;
  s1.sysinfo_block = 5;

  /* Save sysinfo i block 5 */
  sysinfo_save(&s1);
    if (s1.err) return 1;

    /* test that data was written */
  sys_p = (sysinfo_block*) FLASH_BASE;

  /* test that s1 was written to flash */
    addr = (address) &sys_p[5];
```

```
    addr2 = (address) &s1;
    for (i=0; i<256; i++)
      if (addr[i] != addr2[i]) return 2;

    /* test ok */
  return 0;
}

int test_sysinfo_init() {
  /* should find a working block, and return it */
  sysinfo_block s1;

  /* call sysinfo_init */
  sysinfo_init(&s1);
  if (s1.err) return 1;

  /* Now s1.sysinfo_block should contain a working block number
   * and sysinfo_save should work fine */
  sysinfo_save(&s1);
  if (s1.err) return 2;

  /* test ok */
  return 0;
}

void test_load() {
  /* create valid flash image
   * and call boot.
   * Should not return
   * Modify create_flash_image() to test
   * various situations.
   */

 create_flash_image();
  boot();
}

void create_flash_image() {
  /* creates sysinfo and flash blocks
   * the program ligthshow.c are located in prog_array
   *
   * Possible tests:
   * 1; load code and enjoy the ligthshow
   * 2; make invalid cheksum and failsafe will start
```

```
   * 3; store data multiple places in flash, and mess some of the up.
   *    the checksum check will then load the valid data, and boot up
   * 4; set bootcount > max_boot_count and failsafe will start
   * 5; set os_comm to OS_FORCE_FAILSAFE and failsafe will start
   * 6; set bootcount > max_boot_count and os_comm to OS_RESET_BOOT_COUNTER
   *    and the ligthshow will start
   */

#define BOOT_COUNT 0;
#define OS_COMM OS_DO_NOTHING;

  sysinfo_block s1;
  address addr;
  int i;

  /* create sysinfo blocks */
  addr = (address) &s1;
  for(i=0; i<256; i++)
    addr[i] = 0;

  s1.magic_number = SYSINFO_MAGIC_NUMBER;
  s1.sysinfo_block = 1;
  s1.bootcount = BOOT_COUNT;
  s1.os_comm = OS_COMM;

  /* write data to flash
   * byte 0-199 in 0x03040000 (block 4)
   * byte 200-368 in 0x03120000 (block 18)
   */
  flash_erase_sector((address) 0x03040000);
    flash_write((address) 0x03040000, (address) &prog_array[0], 200);

  flash_erase_sector((address) 0x03120000);
    flash_write((address) 0x03120000, (address) &prog_array[200], 169);

    /* write two data_modules
     * we use 3 and 8 as example
     */

    s1.data_module[3].id = 23; // id number
    s1.data_module[3].module_status = MODULE_LOAD; // we want the data loaded
    s1.data_module[3].load_length = 200; // load 200 bytes
    s1.data_module[3].load_addr = (address) 0x02040000; // into 0x02040000
    s1.data_module[3].checksum = crcCompute((address) 0x03040000, 200); // compute checksum
```

```
  s1.data_module[8].id = 45; // another id number
    s1.data_module[8].module_status = MODULE_LOAD;
    s1.data_module[8].load_length = 169;
    s1.data_module[8].load_addr = (address) 0x020400C8;
    s1.data_module[8].checksum = crcCompute((address) 0x03120000, 169);

    /* map data_modules to flash */
    s1.block[4] = 23;
    s1.block[18] = 45;
    s1.block[6] = 45; // no data are placed in block 6, so checksum will fail.

    /* and finaly set the entrypoint */
    s1.entrypoint = (address) 0x02040000;

    /* save the sysinfo and goto boot() */
    sysinfo_save(&s1);
}

void main() {
  led_init();
  led_blink();
  led_on(LED0 | LED1);
  if (!test_sysinfo_load()) led_on(LED2);
  if (!test_sysinfo_save()) led_on(LED3);
  if (!test_sysinfo_init()) led_on(LED4);

  test_load();

  led_on(LED7);
  /* test of loadsystem is missing */
}


/* The lightshow program
 * Blinks the LED's on the
 * Extention board connected to
 * the parallel I/O conroller
 */


#define PROG_ARRAY_SIZE 368
unsigned char prog_array[PROG_ARRAY_SIZE] = {
 48,48,159,229,3,208,160,225,2,172,77,226,0,16,160,227,1,176,160,225,1,112,
```

```
 160,225,20,0,159,229,20,32,159,229,0,32,66,224,0,0,160,227,0,16,160,227,73,
 0,0,235,2,0,0,235,112,161,4,2,112,177,4,2,14,240,160,225,254,255,255,234,13,
 192,160,225,0,216,45,233,4,176,76,226,4,208,77,226,16,0,11,229,255,60,224,
 227,203,48,67,226,16,32,27,229,0,32,131,229,0,168,27,233,13,192,160,225,0,
 216,45,233,4,176,76,226,4,208,77,226,16,0,11,229,255,60,224,227,207,48,67,
 226,16,32,27,229,0,32,131,229,0,168,27,233,13,192,160,225,0,216,45,233,4,176,
 76,226,2,49,160,227,195,55,160,225,255,32,160,227,0,32,131,229,255,60,224,
 227,239,48,67,226,0,32,131,229,0,168,27,233,13,192,160,225,0,216,45,233,4,176,
 76,226,4,208,77,226,97,59,160,227,42,62,131,226,16,48,11,229,16,48,27,229,0,0,
 83,227,0,0,0,26,3,0,0,234,16,48,27,229,1,48,67,226,16,48,11,229,247,255,255,
 234,0,168,27,233,13,192,160,225,0,216,45,233,4,176,76,226,4,208,77,226,1,48,
 160,227,16,48,11,229,16,48,27,229,1,12,83,227,0,0,0,218,8,0,0,234,255,0,160,
 227,206,255,255,235,16,0,27,229,194,255,255,235,224,255,255,235,16,48,27,229,
 131,48,160,225,16,48,11,229,242,255,255,234,255,0,160,227,197,255,255,235,0,
 168,27,233,13,192,160,225,0,216,45,233,4,176,76,226,180,255,255,235,228,255,
 255,235,253,255,255,234};
long prog_array_size(){return((long)PROG_ARRAY_SIZE);}
```

D.5 overall_functions_test.c

```
/* Testing of system functions
 * use ligth to debug */

#include "../system.h"
/*#include "ligth.c"*/

int test_flash_erase_sector() {
  /* erase a flash block */
  address flash = (address) 0x03020000;
  int err;

  err = flash_erase_sector(flash); if (err) return 1;

  /* test ok */
  return 0;
}

int memtest_test() {
  int err;
  address ram = (address) 0x02040000;

  err=(int)memtest(ram,10000);

  if (err)
    return (1); /* We asumes that the ram is working perfect... */

  return 0;
}


int test_flash_write() {
  /* write zero to flash */
  /* write ones - should fail */
  /* erase block and write ones */
  address flash = (address) 0x03020000;
  int err;
  unsigned char data = 0xFF;

  err = flash_write(flash, (address) &zeroTable, 24);
  if (err) return 1;

  err = flash_write(flash, &data, 1);
```

```
  if (!(err)) return 2;

  err = flash_erase_sector(flash);
  if (err) return 3;

  err = flash_write(flash, &data, 1);
  if (err) return 4;

   /* test ok */
   return 0;
}

int watchdog_test(){
  volatile int i = 10000;
  int f;

  wd_start();
  for(f= 0; f< 50000;f=f+1001){
    i=f;
    while (i) i--;/* wait longer and longer */
    wd_touch();/* reset the timer */
    if (f%2==0) /* Blink the light number 6 */
      led_on(LED6);
    else
      led_off(LED6);
  }
  return 0;
}


int test_crc_computation(){
  unsigned char * test_string;
  int err;
  address ram = (address) 0x02040000;
  address flash = (address) 0x03020000;
  checksum cksum;
  checksum known_cksum;

   known_cksum=0x3A1A5178;
   test_string="A test of crcComputation";


     cksum=crcCompute(test_string,24);
   if (cksum!=known_cksum)
```

```
     return 1;

  memcpy(ram,test_string,24);

  cksum=crcCompute(ram,24);
  if (cksum!=known_cksum)
    return 2;

  err = flash_erase_sector(flash);
  if (err) return 3;/* a flash error */

  err = flash_write(flash, ram, 24);
  if (err) return 4;/* a flash error */


  cksum=crcCompute(flash,24);
  if (cksum!=known_cksum)
    return 5;

  test_string="A test of CrcComputation";

  cksum=crcCompute(test_string,24);
  if(cksum==known_cksum)
    return 6;

  /* test ok */
  return 0;
}


void main() {
  led_init();
  led_blink(); /*Blink the lights */
  led_on(LED0 | LED1);/*Turn on light 0 and 1 */
  if (!test_flash_erase_sector())
    led_on(LED2); /* Turn on light 2 */
  if (!test_flash_write())
    led_on(LED3); /* Turn on light 3 */
  if (!test_crc_computation())
    led_on(LED4); /* Turn on light 4 */
  if (!memtest_test())
    led_on(LED5); /* Turn on light 5 */
  if (!watchdog_test())
    led_on(LED6); /* Turn on light 6 */
```

```
  led_on(LED7);   /* Turn on light 7 */
}
```

D.6 packet_sender.c

```
/* Send a packet to the satellite and receive the return packet
 * Packets er read i hexformat from stdin end written to stdout.
 * The code are based on examples from Serial Programming HOWTO.
 */

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <termios.h>
#include <stdio.h>

#define BAUDRATE B2400
#define DEVICE "/dev/ttyS1"


#define BUFF_SIZE 255

main()
{
  int fd,c,i, res, ibuff, length;

  struct termios oldtio,newtio;
  unsigned char buf[BUFF_SIZE], cbuff;

  fd = open(DEVICE, O_RDWR | O_NOCTTY );
  if (fd <0) {perror(DEVICE); exit(-1); }

  tcgetattr(fd,&oldtio); /* save current port settings */

  bzero(&newtio, sizeof(newtio));
  newtio.c_cflag = BAUDRATE | CS8 | CLOCAL | CREAD;
  newtio.c_iflag = IGNPAR;
  newtio.c_oflag = 0;

  /* set input mode (non-canonical, no echo,...) */
  newtio.c_lflag = 0;

  newtio.c_cc[VTIME]    = 0;   /* inter-character timer unused */
  newtio.c_cc[VMIN]     = 1;   /* blocking read until 1 chars received */

  tcflush(fd, TCIFLUSH);
  tcsetattr(fd,TCSANOW,&newtio);
```

```
  while (EOF != scanf("%x", &ibuff))
    {
      cbuff = (char) ibuff;
      write(fd, &cbuff, 1);
    }

  write(2,"OK\n", 3);

  i=0; length = 255;

  while(i<length) {
    res = read(fd,&buf[i],1);
    printf("%02X ", buf[i]);
    i++;
    if (i==4) length = 4 + buf[3] * 0x100 + buf[2];
  }

  printf("\n");

  tcsetattr(fd,TCSANOW,&oldtio);
}
```

D.7 runtest.sh

```
#!/bin/sh
# runs automated tests basen on testfiles
# Format:
# < HEXENCODED PACKET
# > EXPECTED OUTPUT
# the script uses the program eo, which can be found at
# http://www.ibiblio.org/pub/Linux/utils/shell/!INDEX.html

TEST_PROG="./packet_sender"

[ $# != 1 ] && { echo "Usage: runtest.sh testfile"; exit 1;}

TESTFILE=$1


[ -f "$TESTFILE" ] || { echo "Can not read $TESTFILE"; exit 1;}

grep "^<" $TESTFILE | sed "s/<//" > $TESTFILE.input
grep "^> " $TESTFILE | sed "s/>//" > $TESTFILE.expected


[ -f "$TESTFILE.output" ] && rm -f $TESTFILE.output

./eo -v $TESTFILE.input "echo \"@\" | $TEST_PROG >> $TESTFILE.output"

diff --ignore-all-space -i -q $TESTFILE.output $TESTFILE.expected && echo "Test ok"

rm -f $TESTFILE.*
```

D.8 seriel_driver.c

```
/*
  Implementation of serial driver
  can be used to test the board without a radiolink

  sync, 8 bit, no par, 2400 baud ,1 stopbit
*/

#include "usart.h"
#include "cmd_handler.h"

#define USART1

#ifdef USART0
#define USART_BASE 0x0FFFD0000
#define USART_PIO  0xE000
#endif

#ifdef USART1
#define USART_BASE 0x0FFFCC000
#define USART_PIO  0x70000
#endif



StructUSART* const  usart_base =  (StructUSART*) USART_BASE;

void comm_init() {
  at91_reg * const pio = (at91_reg*) 0xFFFF0000;
  *pio = USART_PIO; //  enable pio

  led_blink();

    //* Reset receiver and transmitter
    usart_base->US_CR = US_RSTRX | US_RSTTX | US_RXDIS | US_TXDIS ;

    //* Clear Transmit and Receive Counters
    usart_base->US_RCR = 0 ;
    usart_base->US_TCR = 0 ;

    //* Define the baud rate divisor register
    usart_base->US_BRGR = 5120 ;
```

```
    //* Define the USART mode
    usart_base->US_MR = US_CLKS_MCK | US_CHRL_8 | US_PAR_NO | US_NBSTOP_1 | US_SYNC


    //* Write the Timeguard Register
    // usart_base->US_TTGR = timeguard ;

    //* Enable receiver and transmitter
    usart_base->US_CR = US_RXEN | US_TXEN ;


}

void comm_get_packet(pckt* packet) {
  unsigned short i = 0;
  unsigned short length = MAX_PACKET_SIZE;

  led_off(LED_ALL);
  led_on(0x0F);

  while (i < length) {
    if (usart_base->US_CSR & US_RXRDY) {
      ((address)packet)[i++] = usart_base->US_RHR;

      if (i==3) {
length = packet->data_length + HEAD_PACKET_LENGTH ;
if (length > MAX_PACKET_SIZE) length = MAX_PACKET_SIZE;
      }
    }
  }

}


  void comm_send_packet(rpckt* packet) {
    unsigned int i;
    unsigned short length;

    led_off(LED_ALL);
    led_on(0xF0);

    length = packet->data_length + HEAD_RETURN_PACKET_LENGTH ;

    for(i=0; i<length;i++) {
```

```
      while(!(usart_base->US_CSR & US_TXRDY)) ;
      usart_base->US_THR = ((address)packet)[i];
    }
  }
```

D.9 up-download.test

```
# up and download test
# This file shows how to write testfile
# for the runtest.sh script.

# upload 1 byte (0xBC) to 0x02040000
< 09 09 00 00 00 04 02 01 00 00 00 BC
> 09 00 00 00
# download 1 byte from 0x02040000
< 0A 08 00 00 00 04 02 01 00 00 00
> 0A 00 01 00 BC
#upload 16 bytes to 0x02040104
< 09 18 00 04 01 04 02 10 00 00 00 00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF
> 09 00 00 00
# download 7 bytes from 0x0204010A
< 0A 08 00 0A 01 04 02 07 00 00 00
> 0A 00 07 00 66 77 88 99 AA BB CC
```

Index